

Faculty of engineering and technology  
department of computer systems engineering

**(Arduino powered smart robot)**

**By**

**Abdulhadi Zakor**

**Akram Alkhaled**

**supervisor**

**Dr. Lara Qadid**

**Graduation Project**

**2016**

|  |           |
|--|-----------|
| <b>ABSTRACT.....</b>   | <b>4</b>  |
| <b>CHAPTER 1: INTRODUCTION.....</b>  | <b>5</b>  |
| 1.1 DEFINING THE PROBLEM .....   | 6         |
| 1.2 PROJECT'S TIMELINE .....   | 7         |
| <b>CHAPTER 2: THEORETICAL STUDY .....</b>                                    | <b>9</b>  |
| 2.1 BACKGROUND.....  | 10        |
| 2.1.1 Maze solving .....   | 10        |
| 2.1.2 Types of Mazes.....  | 10        |
| -Linear or railroad maze .....   | 10        |
| -Logic mazes .....   | 10        |
| -Number maze .....   | 10        |
| 2.2 EXPLAINING THE PROBLEM .....   | 11        |
| 2.3 HOW TO SOLVE A MAZE.....   | 12        |
| 2.3.1 Random Mouse Algorithm .....   | 12        |
| 2.3.2 Wall Follower Algorithm.....   | 13        |
| 2.3.3 Pledge Algorithm .....   | 13        |
| 2.4 WHY MAZE SOLVING? .....  | 14        |
| 2.5 THE HARDWARE .....   | 14        |
| 2.5.1 Motors.....  | 15        |
| -DC motor .....  | 15        |
| -stepper motor .....   | 16        |
| Advantages/disadvantages of stepper motors .....                             | 17        |
| 2.5.2 Sensors.....   | 19        |
| -Photo sensors .....   | 19        |
| <b>CHAPTER 3: ANALYTICAL STUDY.....</b>                                      | <b>21</b> |
| 3.1 THE ALGORITHM .....  | 22        |
| 3.2 THE PLATFORM .....   | 26        |
| 3.2.1 Microcontrollers .....   | 26        |
| 3.2.2 What is Arduino .....  | 27        |
| 3.2.3 Difference between microcontrollers, microprocessor, and Arduino ..... | 31        |
| 3.2.3 Arduino Due.....   | 32        |
| 3.3 SENSORS IMPLEMENTING .....   | 34        |
| 3.3.1 Short range sensors .....  | 36        |
| 3.3.2 Long range sensors.....  | 37        |
| <b>CHAPTER 4: DESIGNING THE SYSTEM.....</b>                                  | <b>40</b> |
| 4.1 THE CAR:.....  | 41        |
| 4.1.1 The body.....  | 41        |
| 4.1.2 The wheels .....   | 42        |
| 4.1.3The motors: .....   | 43        |
| 4.1.4 The power source: .....  | 44        |
| 4.2 OPERATING THE MOTORS: .....  | 45        |
| 4-3 SENSORS: .....   | 47        |

|   |           |
|---|-----------|
| 4.4 ARDUINO DUO: .....  | 49        |
| 4.5 THE SOFTWARE: .....   | 49        |
| 4.6 EXPANDING THE MAZE SOLVING THEORY .....                       | 51        |
| 4.6.1 HISTORY OF PAC-MAN ORIGINAL AND THE GAME'S OBJECTIVES ..... | 52        |
| 4.6.2 PAC-MAN PROJECT BACKBONE .....                              | 54        |
| 4.6.2 ALGORITHM DESIGN.....                                       | 54        |
| <b>CHAPTER 5: IMPLEMENTATIONS.....</b>                            | <b>59</b> |
| 5.1 The body .....  | 60        |
| 5.2 THE CODE .....  | 60        |
| 5.3 SOFTWARE IMPLEMENTATION OF PAC-MAN .....                      | 63        |
| 5.4 RESULTS .....   | 83        |
| 5.4.1 The robot: .....  | 83        |
| 5.4.2 Pac-man.....  | 84        |
| CHALLENGES .....  | 87        |
| <b>CHAPTER 6: CONCLUSIONS AND FUTURE WORK .....</b>               | <b>89</b> |
| 4.1 FUTURE IMPROVEMENTS .....                                     | 90        |
| 4.1.1 The robot.....  | 90        |
| 4.1.2 Pac-man.....  | 90        |
| 4.2 CONCLUSION .....  | 91        |
| <b>REFERENCES .....</b>   | <b>92</b> |

## **Abstract**

Through our life, we face variety of problems that need to be solved. And scientists have always viewed these problems in a quest to improve our living, many methods and ideas were invented in order to do that.

With the advancement of technology, machines had the advantages of fast calculating speed in addition to the ability to work in extreme environments and the possibility of installing additional equipment that can expand its senses and functionality, and with these abilities we can use machines to improve our lives. So as students of science our ability to learn and implement an experiment is the reason we build projects such as this one.

We learnt through our studies that we shouldn't stop at one idea or the implementation of it. Thus we took this project further and implemented more algorithms that can help in developing the humans' view on the maze solving problem and ran some simulations to test and improve our theories.

## **Chapter 1: Introduction**

Autonomous robotics is a field with wide reaching applications. From bomb sniffing robots to autonomous devices for finding humans in wreckage to home automation, many people are interested in low-power, high speed, reliable solutions. There are many problems facing such designs: unfamiliar terrain and inaccurate sensing continue to plague designers.

Robotics competitions have intrigued the engineering community for many years. In this report, we provide a study concerning building a machine and prompting it to face a common human problem, which is maze solving. The maze solving problem is one of the highly studied problems for competitions because it require many fields of work.

The maze solving robots formed a hardware implementation for the algorithm. But, taking the algorithm further will be restricted by the hardware limitations. Which will definitely lead to limited results. And because of that, a part of the problem cannot be purely about robotics. But, also about the developed and simulated software that is going ahead of the accompanying slowly-developing hardware.

## **1.1 Defining the problem**

In a situation where a human might find himself stuck in a maze, or more accurately get lost in a place where the roads are not familiar to him, while exploring ruins or visiting a new city and losing the way out, or simply being unable to get around in a given place.

We'll try to build a machine capable of dealing with navigating through a maze. We'll also try to provide the optimal solution to the problem to achieve a way out of our problem.

## 1.2 Project's Timeline

The timeline is shown in the table 1.1 that follows.

Table 1.1: Timeline

| Week                   | Event  |
|------------------------|--|
| 1 <sup>st</sup>        | <b>First session with the supervisor to discuss the schedule</b> |
| 2 <sup>nd</sup>        | <b>Designing the car</b>   |
| 3 <sup>rd</sup>        | <b>Writing the robot's the algorithms</b>                        |
| 4 <sup>th</sup>        | <b>Implementing the left hand algorithm is Arduino platform</b>  |
| 5th&6th                | <b>Designing the motor controlling circuitry</b>                 |
| 7th&8th                | <b>Designing the sensors circuitry</b>                           |
| 9 <sup>th</sup>        | <b>Implementing the hardware</b>                                 |
| 10 <sup>th</sup>       | <b>Researching Pac-man's background and history</b>              |
| 11 <sup>th</sup>       | <b>Designing an implementation</b>                               |
| 12 <sup>th</sup>       | <b>Improving Pac-man's software's implementation</b>             |
| 13th&14th              | <b>typing the report and preparing the presentation</b>          |
| 15 <sup>th</sup> &16th | <b>Printing the report and making the presentation</b>           |

This chapter was an introduction to our project. We showed here the importance of machines and the necessity of designing machines that are capable of solving our problems on their own.

The next chapter will be about the resources we need for our project. The third chapter will talk about the analytical studies we had. While in

chapter four we will design our robot and design an implementation to maze solving using AI. In chapter 5 we will see the results of the robot and the software implementation. Chapter six will have our final conclusions.



## **CHAPTER 2: Theoretical Study**

Solving a problem needs careful planning as we know. But, before we start planning we need to understand the problem and go through it, analyzing it step by step.

In this chapter we learn about mazes, open platforms, and the other hardware we might need during the implementation.

## **2.1 Background**

We explore here the main components of our problem, and discover some of its main ideas and concepts.

### **2.1.1 Maze solving**

A maze is a path or collection of paths, typically from an entrance to a goal. The word is used to refer both to branching tour puzzles through which the solver must find a route, and to simpler non-branching patterns that lead unambiguously through a convoluted layout to a goal. [1]

### **2.1.2 Types of Mazes**

#### **-Linear or railroad maze**

A maze in which the paths are laid out like a railroad with switches and crossovers. Solvers are constrained to moving only forward. Often, a railroad maze will have a single track for entrance and exit. [1]

#### **-Logic mazes**

These are like standard mazes except they use rules other than "don't cross the lines" to restrict motion [1]

#### **-Number maze**

A maze in which numbers are used to determine jumps that form a pathway, allowing the maze to crisscross itself many times.[1]

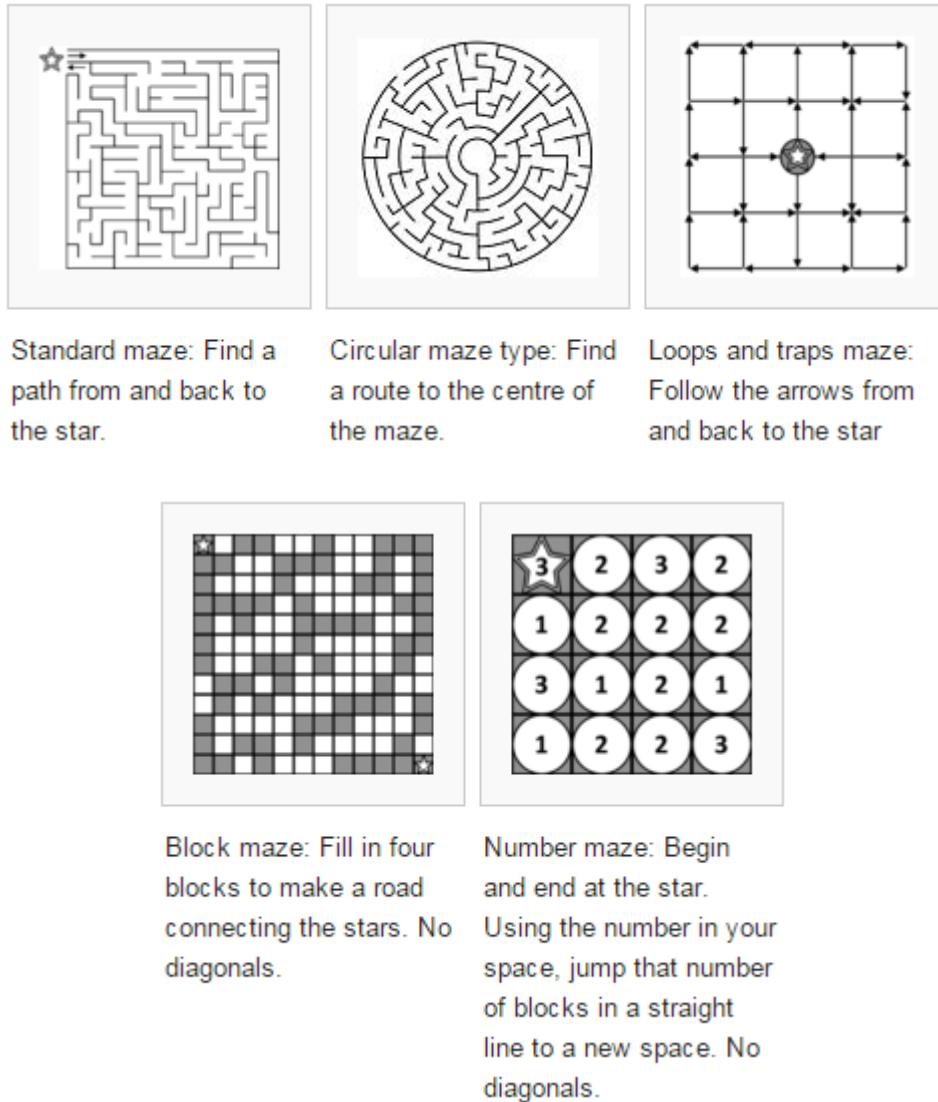


Figure2.1: Types of mazes

## 2.2 Explaining the problem

Maze solving is an old problem that faced people who found themselves stuck in a place they don't know their way out of it, like wars or getting lost. And that leads to many algorithms to solving mazes, some of them are common. But, despite the various numbers of algorithms we have, the perfect algorithm is still a matter of theories.

Imagine that you've found yourself in a place you can't know your way out, like a part of a city you've never been to, what are your odds in going back to the place you came from?

A human being can find his way out by using their basic instincts, but some cases require logical actions based on calculations. And in these situations we find that machines are better due to many factors, including processing speed and immunity to psychological pressure. And thus, we'd be able to build machines that can beat humans in solving a maze by implementing the optimal theory to the maze we're solving.

## **2.3 How to solve a maze**

Solving a maze requires an algorithm. Although, we need to know that algorithms are not necessarily universal; one algorithm can't simply solve all types of mazes.

Here we provide a couple of algorithms that can solve normal mazes.

### **2.3.1 Random Mouse Algorithm**

The Random Mouse algorithm is just what it sounds like: it's the way a mouse would solve a maze. It's just wandering around the maze, hoping you find the end. Start out by going straight, and keep going straight ahead (following any turns that the maze makes) until you come to any point where there is more than one way to go (an intersection or junction). At this point, make a random choice which way to go, and do it. Then go back to just following the maze until you come to the next junction. At the junctions, only include left, right, or straight in your random choice; never turn 180 degrees and go back the way you came. The only time you ever turn around and go back is when you come to a

dead-end (front, left, and right are all blocked). That's all there is to it. If you do a Random Mouse for your robot, expect it to take a LONG time for large mazes.

### 2.3.2 Wall Follower Algorithm

The Wall Follower algorithm is very simple to understand. At the start of the maze, put your left (or right) hand on the wall, and then start walking. Never take your hand off the wall. For a simply connected maze, you will find the exit. That's all there is to it. The picture below shows a maze solved using the Wall Follower algorithm; the red path shows the solution, the gray path shows the other parts of the maze that were visited while following the wall. Look carefully at the maze, and see if you can figure out if this was left- or right-hand wall following.

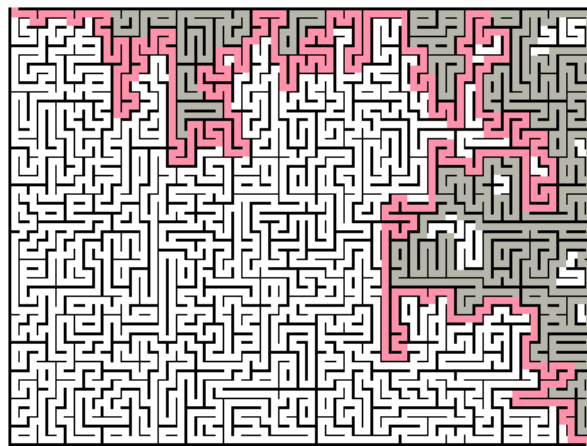


Figure 2.3.2: Wall follower's result

### 2.3.3 Pledge Algorithm

The Pledge algorithm is similar to the Wall Follower algorithm, and it includes wall following as part of its solution. The Pledge algorithm is more powerful than Wall Follower, and can solve some mazes that Wall

Follower can't. However, the Pledge algorithm points out a restriction; you would need to count the number of turns you have made.

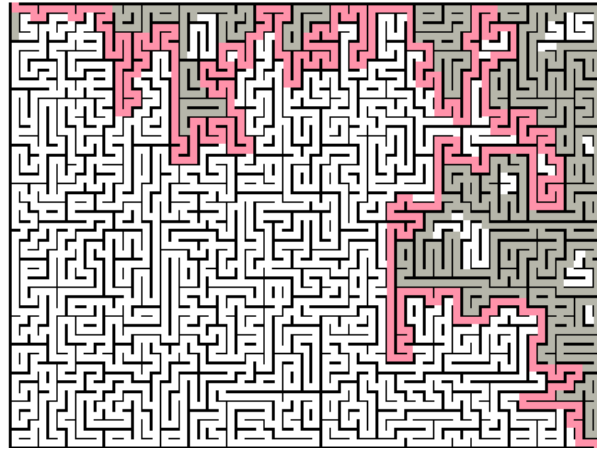


Figure 2.3.3: Pledge's result

These algorithms and many other more show us that we have various models we can use in implementing the solution, but it all depends on the difficulty of the problem itself.

## 2.4 Why maze solving?

The problem itself is not why we created this project, but the fact that there is a community of engineers and scientists who build and test robots that can work and compete against each other in solving mazes is really what made us consider building. We want to show that by prototyping a robot.

## 2.5 The Hardware

When we talk about a maze solving robot, we need to talk about the hardware we need to build the robot, including the motors, the sensors, and the platform we want to use while implementing our robot.

### 2.5.1 Motors

Motors are transducers that convert energy into mechanical energy, and the type of a motor indicates the type of energy the motor consumes and converts.

An electric motor is an electrical machine that converts electrical energy into mechanical energy.

In normal motoring mode, most electric motors operate through the interaction between an electric motor's magnetic field and winding currents to generate force within the motor. In certain applications, such as in the transportation industry with traction motors, electric motors can operate in both motoring and generating or braking modes to also produce electrical energy from mechanical energy.

Here we explore some of the two most common electric motors in the world of control systems [1]

#### -DC motor

Is any of a class of electrical machines that converts direct current electrical power into mechanical power. The most common types rely on the forces produced by magnetic fields. Nearly all types of DC motors have some internal mechanism, either electromechanical or Electronic, to periodically change the direction of current flow in part of the motor. Most types produce rotary motion; a linear motor directly produces force and motion in

a straight line [3]



Figure 2.5.1: Dc Motor

-stepper motor

A stepper motor or step motor or stepping motor is a brushless DC electric motor that divides a full rotation into a number of equal steps. The motor's position can then be commanded to move and hold at one of these steps without any feedback sensor (an open-loop controller), as long as the motor is carefully sized to the application in respect to torque and speed [3]



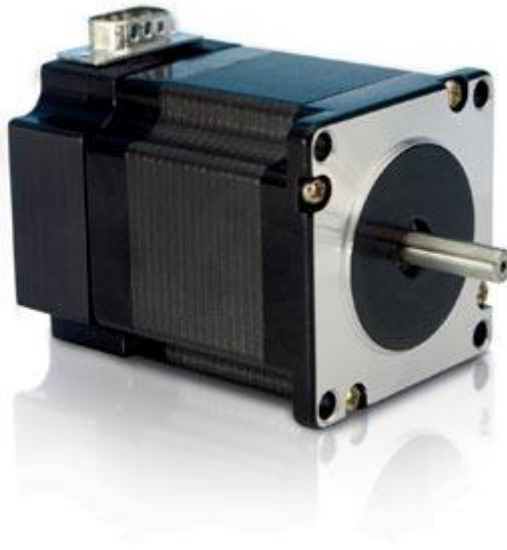


Figure 2.5.2: Stepper Motor

#### Advantages/disadvantages of stepper motors

##### Advantages:

- Low cost for control achieved
- High torque at startup and low speeds
- Ruggedness
- Simplicity of construction
- Can operate in an open loop control system
- Low maintenance
- Less likely to stall or slip
- Will work in any environment
- Can be used in robotics in a wide scale.
- High reliability
- The rotation angle of the motor is proportional to the input pulse.

- The motor has full torque at standstill (if the windings are energized)
- Precise positioning and repeatability of movement since good stepper motors have an accuracy of 3 – 5% of a step and this error is non-cumulative from one step to the next.
- Excellent response to starting/stopping/reversing.
- Very reliable since there are no contact brushes in the motor. Therefore the life of the motor is simply dependent on the life of the bearing.
- The motors response to digital input pulses provides open loop control, making the motor simpler and less costly to control.
- It is possible to achieve very low-speed synchronous rotation with a load that is directly coupled to the shaft.
- A wide range of rotational speeds can be realized as the speed is proportional to the frequency of the input pulses.

#### Disadvantages

- Require a dedicated control circuit
- Use more current than D.C. motors
- Torque reduces at higher speeds
- Resonances can occur if not properly controlled.
- Not easy to operate at extremely high speeds.

### 2.5.2 Sensors

In order for the robot to be able to move and be aware of its surrounding, we need to give our robot "senses" that can provide an idea about the outer world the robot moves in. And for that reason we use sensors.

Sensors are electrical or electronical transducers that converts signals that can't be read by the computers (because computers only operate in binary system) into signals that can be read and understood by the computer we supply with these sensors.

#### -Photo sensors

A photoelectric sensor, or photo eye, is an equipment used to discover the distance, absence, or presence of an object by using a light transmitter, often infrared, and a photoelectric receiver. They are largely used in industrial manufacturing. There are three different useful types: opposed (through beam), retro-reflective, and proximity-sensing (diffused)



Figure 2.5.3: Various Photo Sensors

| Name                | Advantages  | Disadvantages  |
|---------------------|---|--|
| <b>Through beam</b> | <ul style="list-style-type: none"> <li>• Most accurate</li> <li>• Longest sensing range</li> <li>• Very reliable</li> </ul>   | <ul style="list-style-type: none"> <li>• Must install at two points on system: emitter and receiver</li> <li>• Costly - must purchase both emitter and receiver</li> </ul>   |
| <b>Reflective</b>   | <ul style="list-style-type: none"> <li>• Only slightly less accurate than through-beam</li> <li>• Sensing range better than diffuse</li> <li>• Very reliable</li> </ul> | <ul style="list-style-type: none"> <li>• Must install at two points on system: sensor and reflector</li> <li>• Slightly more costly than diffuse</li> <li>• Sensing range less than through-beam</li> <li>•</li> </ul> |
| <b>Diffuse</b>      | <ul style="list-style-type: none"> <li>• Only install at one point</li> <li>• Cost less than through-beam or reflective</li> </ul>                                      | <ul style="list-style-type: none"> <li>• Less accurate than through- beam or reflective</li> <li>• More setup time involved</li> </ul>   |

## **CHAPTER 3: Analytical Study**

In this chapter we explore the handy resources we used to implement the solution to our problem, including the algorithms and the hardware we used to achieve our goal.

The work was concentrated on building a machine capable of solving a maze with human-like actions, and that means a smart software installed on a convenient hardware.

While the software mainly consisted of an algorithm that can be implemented on the hardware to be chosen later, the hardware was a matter of discussion due to the variety of available hardware (at least theoretically).

### **3.1 The algorithm**

During our research we came across many algorithms that can be used to solve a maze, some of them didn't present an actual intelligent behavior (like the random mouse algorithm). Thus we kept searching and found the pledge algorithm to be remarkable, and suitable for our project.

There are basically 2 steps. The first is to drive through the maze and find the end of it. The second is to optimize that path so your robot can travel back through the maze, but do it perfectly without going down any dead ends, or loops.

Step 1 How does the robot find the end of the maze

I use a technique called the left hand on the wall. Imagine you are in a maze and you keep your left hand on the edge of the wall at all times.

Doing this would eventually get you out of a non-looping maze.

This left hand on wall algorithm can be simplified into these simple conditions:

- If you can turn left then go ahead and turn left,
- else if you can continue driving straight then drive straight,
- else if you can turn right then turn right.
- If you are at a dead end then turn around.

In case of a loop, we add the last condition which state:

- if you take three same side turns then change the right hand algorithm, into using the left hand. In that case we must have a counter and the maze should be compatible with our hardware, also if the left hand algorithm face another loop, we change back into the right hand etc. till our maze is fully solved.

The robot has to make these decisions when at an intersection. An intersection is any point on the maze where you have the opportunity to turn. If the robot comes across an opportunity to turn and does not turn then this is consider going straight. Each move taken at an intersection or when turning around has to be stored.

L = left turn

R= right turn

S= going straight past a turn

B= turning around

As the Figure 3.1 the path would be recorded based on the decisions the robot take, and the end path would show the bath as such group of letters that we would need for step 2

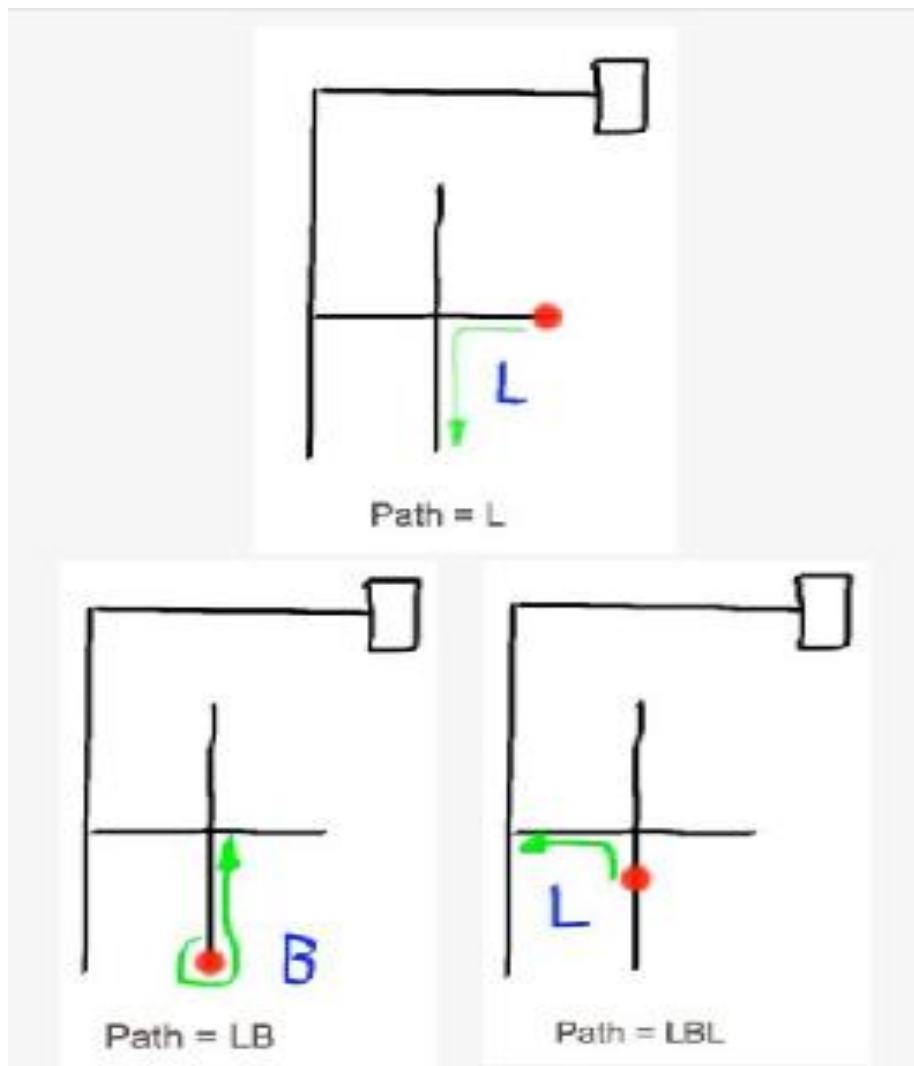


Figure 3.1.1 path storage

Step 2: how does the robot find the best path

As the robot moves through the maze, the root it follows would be registered, and when it finishes the root would be presented as letters showing all the decision it made (ex: final path: "LBLLBSR")

The idea in step two, is to take advantage of the registered path, so we take the registered letters, and change the group of letters that resemble a



wrong decision, which means if our robot goes right (R) to be faced with a dead end, so it goes back (B) then right again (R)

The path at the end would register (Final path: RBR) these three should be changed into making the robot move passing the right turn and not going into an extra un-needed movement so the final path would be from (RBR, into just moving straight S) just like we show in figure 3.1.2

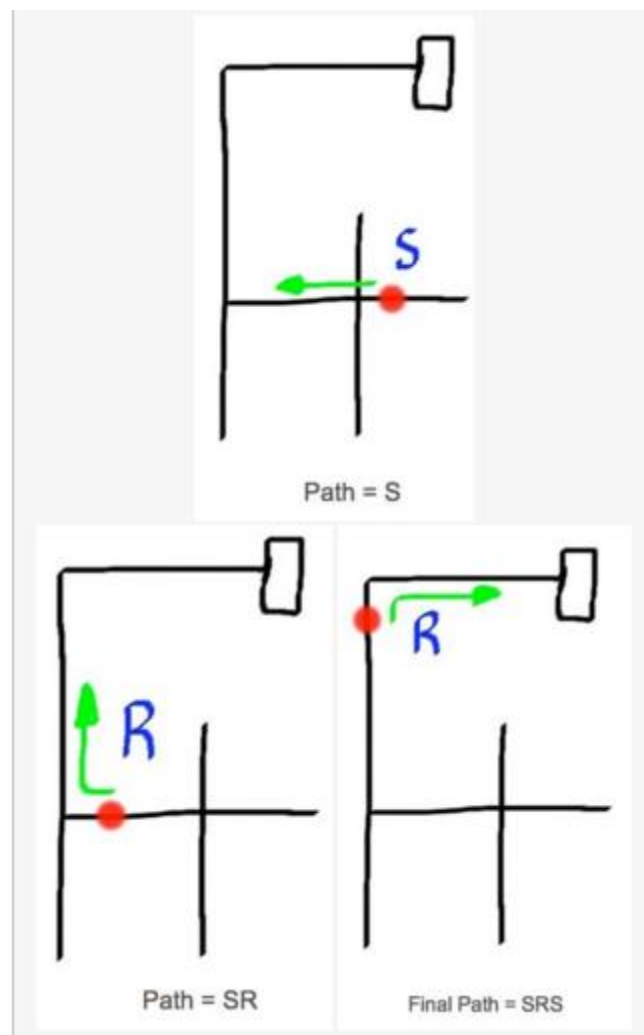


Figure 3.1.2: implementing a wrong turn

Here is an example list that covers optimizing all the wrong in a simple loop-less maze:

LBR = B

LBS = R

RBL = B

SBL = R

SBS = B

LBL = S

This concludes our theoretical study of the algorithm, and now we move into discussing the platform and the actual hardware that we will be using to serve our algorithm.

## **3.2 The platform**

Through our search we had to study the differences between the Arduino platform and a microcontroller guided board.

The differences between the two are huge, and here we provide a brief explanation of the two platforms and a comparison between Arduino platform and a normal microcontroller guided platform.

### **3.2.1 Microcontrollers**

A microcontroller is a small computer (SoC) on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of Ferroelectric RAM, NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for embedded applications, in contrast to the microprocessors used in personal computers or other general purpose applications consisting of various discrete chips.[1]



Figure 3.2.1: Microcontroller chip

Microcontrollers are used in automatically controlled products and devices, such as automobile engine control systems, implantable medical devices, remote controls, office machines, appliances, power tools, toys and other embedded systems. By reducing the size and cost compared to a design that uses a separate microprocessor, memory, and input/output devices, microcontrollers make it economical to digitally control even more devices and processes. Mixed signal microcontrollers are common, integrating analog components needed to control non-digital electronic systems.

A normal microcontroller is programmed using assembly language, and it can be different depending on the manufacturer.[3]

### **3.2.2 What is Arduino**

Is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists, and anyone interested in creating interactive objects or environments. Or more simply, you load on some code and it can read sensors, perform actions based on inputs from buttons, control motors,

and accept shields to further expand its capabilities. Really, you can do almost anything.

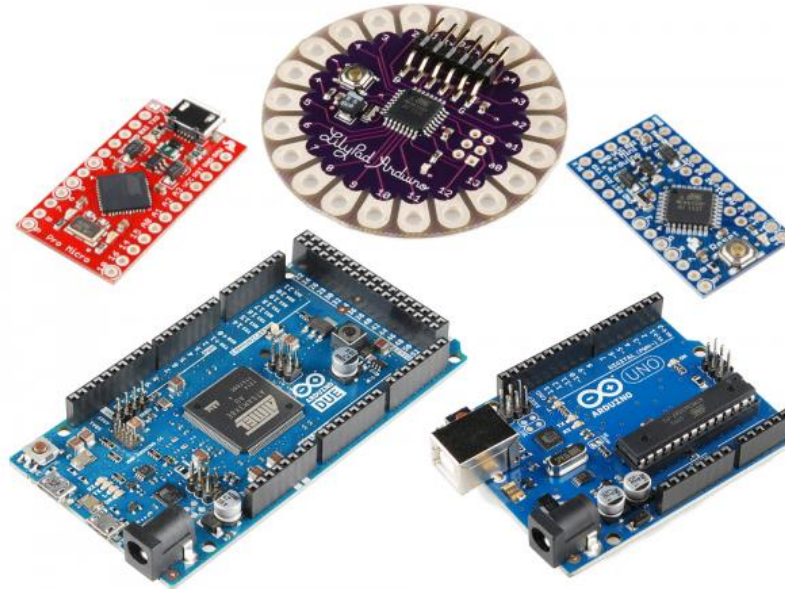










Figure 3.2.2 Samples of Arduino Boards

All Arduino boards have one thing in common: they are programmed through the Arduino IDE. This is the software that allows you to write and upload code in C. Beyond that, there can be a lot of differences. The number of inputs and outputs (how many sensors, LEDs, and buttons you can use on a single board), speed, operating voltage, and form factor are just a few of the variables. Some boards are designed to be embedded and have no programming interface (hardware) which you would need to buy separately. Some can run directly from a 3.7V battery, others need at least 5V. Check the chart on the next page to find the right Arduino for your project [2]

Table 3.1: Most Famous Arduino Boards

| Item  | System Voltage | Clock Speed | Digital I/O | Analog Inputs | PWM | UART | Programming Interface  |
|---|----------------|-------------|-------------|---------------|-----|------|------------------------|
| <b>ATmega328 Boards — 32kB Program Space // 1 UART // 6 PWM // 4-8 Analog Inputs // 9-14 Digital I/O</b>  |                |             |             |               |     |      |                        |
| <br><b><u>Arduino Uno - R3</u></b>                             | 5V             | 16MHz       | 14          | 6             | 6   | 1    | USB via ATmega16U2     |
| <br><b><u>Arduino Pro Mini</u></b><br><b><u>3.3V/8MHz</u></b> | 3.3V           | 8MHz        | 14          | 8             | 6   | 1    | FTDI-Compatible Header |
| <br><b><u>Arduino Ethernet</u></b>                           | 5V             | 16MHz       | 14          | 6             | 6   | 1    | FTDI-Compatible Header |
| <b>ATmega32U4 Boards — 32kB Program Space // 1 UART // 5-7 PWM // 12 Analog Inputs // 9-20 Digital I/O</b>                                      |                |             |             |               |     |      |                        |

|  |      |           |                  |    |    |   |                           |
|--|------|-----------|------------------|----|----|---|---------------------------|
| <br><b><u>Arduino Leonardo</u></b>                        | 5V   | 16M<br>Hz | 20* <sub>-</sub> | 12 | 7  | 1 | Native<br>USB             |
| <br><b><u>Pro Micro 5V/16MHz</u></b>                      | 5V   | 16M<br>Hz | 12               | 12 | 5  | 1 | Native<br>USB             |
| <br><b><u>LilyPad Arduino USB</u></b>                     | 5V   | 16M<br>Hz | 9                | 12 | 5  | 0 | Native<br>USB             |
| <b>ATmega2560 Arduino Mega's — 256kB Program Space // 4 UARTs // 14 PWM //<br/>16 Analog Inputs // 54 Digital I/O</b>                      |      |           |                  |    |    |   |                           |
| <br><b><u>Arduino Mega 2560</u></b><br><b><u>R3</u></b> | 5V   | 16M<br>Hz | 54               | 16 | 14 | 4 | USB via<br>ATMega16<br>U2 |
| <b>AT91SAM3X8E Arduino Due — 512kB Program Space // 4 UARTs // 12 PWM<br/>(2 DAC) // 12 Analog Input // 54 Digital I/O</b>                 |      |           |                  |    |    |   |                           |
| <br><b><u>Arduino Due</u></b>                           | 3.3V | 84M<br>Hz | 54               | 12 | 12 | 4 | USB native                |

The importance of Arduino comes from its flexibility and the easy implantation possibility.

If you need to implement a project using a microcontroller you have to be fully aware of the differences between each of its types.

Prototyping, maintaining, and expanding your projects is very easy using Arduino compared to using a microcontroller platform.

### **3.2.3 Difference between microcontrollers, microprocessor, and Arduino**

**An Arduino** is a board containing an Atmel AVR microcontroller and usually providing a set of connectors in a standard pattern. The microcontroller is typically preprogrammed with a "bootloader" program that allows a program (called a "sketch") to be loaded into the microcontroller over a serial connection (or virtual serial over USB connection) from a PC.

**A microprocessor** is an (integrated circuits) IC that contains only a central processing unit (CPU). The IC does not contain RAM, ROM or other peripherals. The IC may contain cache memory but it is not designed to be usable without any external memory. Microprocessors cannot store programs internally and therefore typically load software when powered on, this usually involves a complex multistage "boot" process where "firmware" is loaded from external ROM and eventually an operating system is loaded from other storage media (e.g. hard disk). It is typically found in a personal computer.

**A microcontroller** is an IC that contains a CPU as well as some amount of RAM, ROM and other peripherals. Microcontrollers can function without external memory or storage. Normally, microcontrollers are either programmed before being soldered or are programmable using In-

System-Programming connectors via a special "programmer" device attached to a personal computer.

Typical microcontrollers are much simpler and slower than typical microprocessors but I believe the distinction is mostly one of scale and application. It is found, for example, in simple appliances such as basic washing machines.[3]

### **3.2.3 Arduino Due**

We will use the Arduino Due in our project. The Arduino Due is a microcontroller board based on the Atmel SAM3X8E ARM Cortex-M3 CPU. It is the first Arduino board based on a 32-bit ARM core microcontroller. It has 54 digital input/output pins (of which 12 can be used as PWM outputs), 12 analog inputs, 4 UARTs (hardware serial ports), a 84 MHz clock, an USB OTG capable connection, 2 DAC (digital to analog), 2 TWI, a power jack, an SPI header, a JTAG header, a reset button and an erase button. Unlike most Arduino boards, the Arduino Due board runs at 3.3V. The maximum voltage that the I/O pins can tolerate is 3.3V. Applying voltages higher than 3.3V to any I/O pin could damage the board. The board contains everything needed to support the microcontroller; simply connect it to a computer with a micro- USB cable or power it with a AC-to-DC adapter or battery to get started. The Due is compatible with all Arduino shields that work at 3.3V and are compliant with the 1.0 Arduino pinout.[2]

The specifications are provided in the site as follows

|                    |             |
|--------------------|-------------|
| Microcontroller:   | AT91SAM3X8E |
| Operating Voltage: | 3.3V        |



|  |                                   |
|--|-----------------------------------|
| Input Voltage (recommended):             | 7-12V                             |
| Input Voltage (limits):                  | 6-16V:                            |
| Digital I/O Pins:                        | 54 (of which 12 provide PWM)      |
| Analog Input Pins:                       | 12                                |
| Analog Output Pins:                      | 2 (DAC)                           |
| Total DC Output Current on all I/O lines | 130 mA                            |
| DC Current for 3.3V Pin                  | 800 mA                            |
| DC Current for 5V Pin                    | 800 mA                            |
| Flash Memory                             | 512 KB all available for the user |
| SRAM                                     | 96 KB (two banks: 64KB            |
| and 32KB)                                |                                   |
| Clock Speed                              | 84 MHz                            |
| Length                                   | 101.52 mm                         |
| Width                                    | 53.3 mm                           |
| Weight                                   | 36 g                              |

The Due can be programmed with the Arduino Software (IDE).  
Uploading sketches to the SAM3X is different than the AVR microcontrollers found in other Arduino boards because the flash memory needs to be erased before being re-programmed. Upload to the

chip is managed by ROM on the SAM3X, which is run only when the chip's flash memory is empty.[2]

### **3.3 Sensors implementing**

In order to execute the algorithm (and keep the robot from crashing into obstacles), the robot must be able to "see" the environment around it. There are some very serious technical challenges associated with this task. First, we need not only to be able to detect obstacles, but also to measure our distance from them at very close proximities. When our robot traverses a diagonal path, the maximum spacing between posts must be close enough to be detected and far enough to keep the robot from crashing.

It is clear that we must have accurate distance measurements at short distances in order to avoid hitting the posts. Problems with traditional short range sensors make this difficult. In order to get the resolution we need to keep our control system stable at high speeds, we need to limit the maximum visible distance of our short range sensors to a maximum of

8cm.

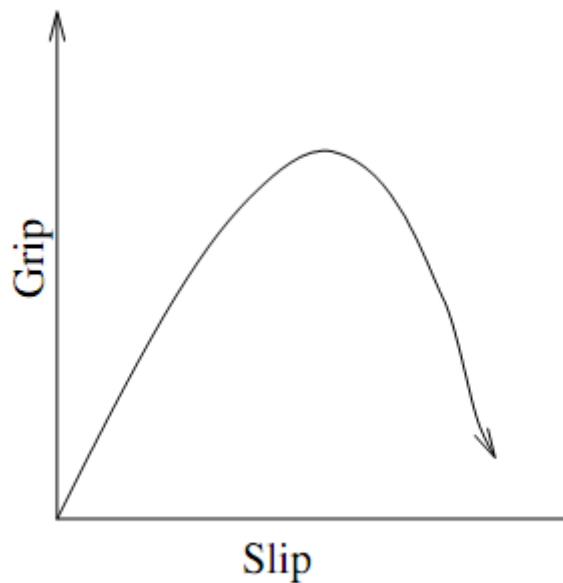


Figure 3.3.1: grip-slip curve

Physically, the wheels must slip when accelerating; figure 3.3.1 is what automotive engineers call a "grip-slip" curve for a typical rubber tire. It is obvious that some amount of wheel slip is necessary to exert the acceleration force. Worse, the actual grip-slip curve is dependent on floor material and thus unknown to us all we know about the floor is that it absorbs light. However, we must have a reliable way to measure the distance we have traveled.

It is clear, then, that any distance measure we make cannot totally rely on the motion of the wheels or the motors. Typical advanced robot designs get around this problem by mounting "idler" wheels on the ground that are not powered by the motors, but merely roll on the ground. While this solution is attractive, it uses precious space under the robot, and, in our design, this space is at a premium. It was therefore necessary to design a

solution on top of the robot. This led to the design of our long range sensing mechanisms.

### 3.3.1 Short range sensors

As discussed in the introduction to this section, our short range sensors need to have a dynamic range. To accomplish this, we use a fairly traditional design for distance measurement. Our short range sensors use one infrared LED and two photo-sensitive detectors, spaced a known distance apart. This configuration is shown in figure 2. Note that the angle of acceptance of the two detectors is very narrow, while the angle of light given off by the LED is large.

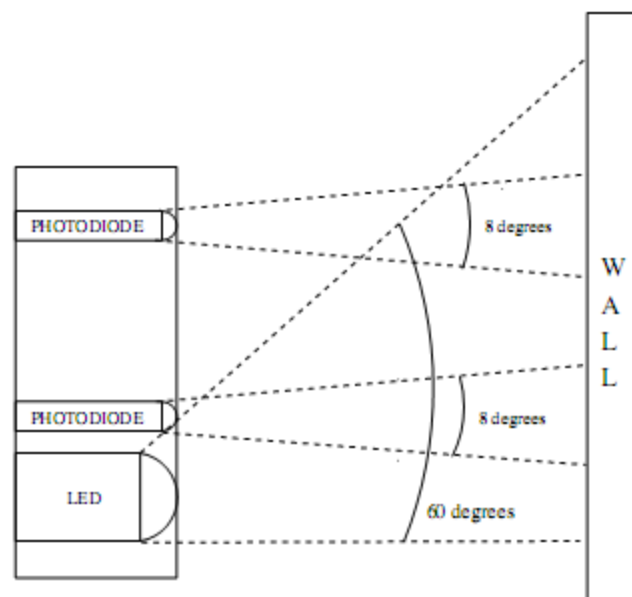


Figure 3.3.2: Sensor Implementation

Therefore, because of the fall off nature of light, the quotient of the voltages produced by the two photo-transistors will be a monotonic function that is directly related to distance. An empirically determined

lookup table will allow us to compute the actual distance if necessary. Another important property of this design is the fact that it is completely insensitive to the operating environment. At startup time, we calibrate the sensors by taking measurements of voltage readings in the contest environment. This allows us to correct for the ambient light levels at run-time. In addition, our two detector design has a significant advantage over the more conventional one detector method: it is totally insensitive to the reflectivity of the surface.

Recall that mazes are typically designed as a lattice of squareposts with slots in the sides to accommodate modular wall units. Because we are taking a quotient of two voltages measured with a vertical displacement, as long as the reflectivity is constant in the vertical dimension, our sensors will behave properly. It is important to note that the above quotient is in fact proportional to the square of the distance, which means we will get much higher resolution measurements as the distance decreases. This is exactly the behavior we desire, since it is very important to remain stable at close proximities to walls, especially in the diagonal path case.

### **3.3.2 Long range sensors**

Our long range sensor mechanisms are much more sophisticated. In order to understand the design, it is first important to discuss how these sensors are to be used. The long range sensors measure distance to an object that is much farther away than can be measured by our short range sensors. However, we expect the accuracy to be much less. The reason for this comes from the worst case scenario for wheel slippage, shown in figure

### 3.3.3

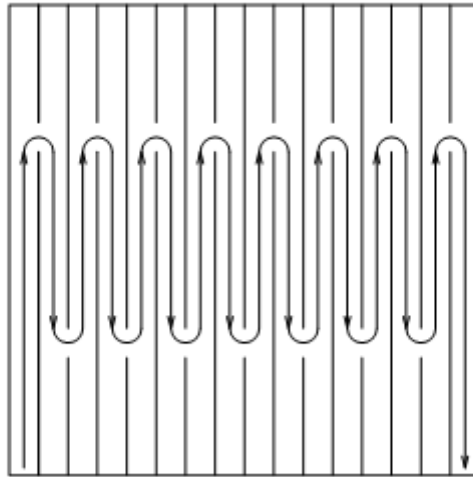


Figure 3.3.3 Worst Case Slippage

Every time we pass an opening in a wall, our short range sensors will detect this. As long as we know what block we're in, we can update our current absolute position with high accuracy. The worst case situation shown in figure 3.3.3 shows a maze configuration where we must speed down a long straightaway and then make a very sharp turn somewhere in the middle. For motor control reasons, the turn must begin before the front of the mouse crosses the opening in the wall. Therefore, we must know where we are to within half a block, or two things might go wrong. First, we might begin the turn at the wrong time, hitting the wall mid-turn. Second, we might not begin braking early enough to make the turn at all without slipping. Wheels slipping in place is a recoverable error, but skidding around a turn poses a much harder problem.

To design a sensor to meet these requirements, we need a special purpose laser range finder. Our design uses a linear triangulation method that exploits properties of similar triangles in a way similar to the grade

school method of computing the height of a building based on its shadow. Our design is shown in figure 3.3.4.

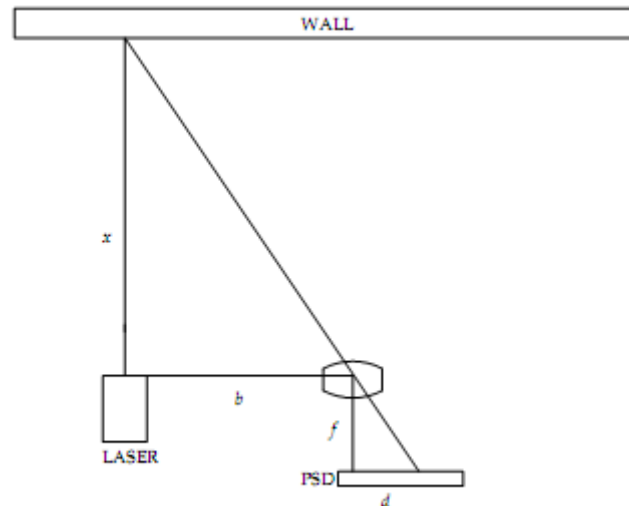


Figure 3.3.4 A Long Range Sensor

The spacing  $b$  between the laser and the lens is known, as is the distance  $f$  between the lens and the rear surface (in fact, it is the focal length of the lens). Because of this, we can compute the distance from the laser to the object  $x$  if we can measure the displacement  $d$  of the image of the laser beam on the rear surface. This turns out to be relatively easy; we simply use a "Position Sensitive Detector", or PSD. PSD's are devices typically found in auto focus cameras, where they are used for approximating distances to objects. This application is very similar to what we are trying to accomplish with our long range sensors. By focusing the image of a laser beam on the PSD, we can obtain a fairly accurate measurement of our distance up to 2 meters away.

## **Chapter 4: designing the system**



Designing the system before implementing it is very important; it gives us the chance to evade mistakes.

And in this chapter we will introduce a full design for the system with the necessary charts and figures.

## **4.1 The car:**

In order to move around the maze we need a car that carries the essential components of the robot and provide a reliable structure that is consistent and suitable for the robot.

The car consists of motors, wheels and a body. We will design them in this chapter.

### **4.1.1 The body**

The car's body has the biggest effect on the robot's movement. Because, it's the platform that carries the robot. The body needs to be light and robust. And we will use plastic as a material to build it.

Like most cars, ours will have two wheel axes; a front and a rear. Their length is going to be greater than the car's length to facilitate rotating. Rotating will be conducted through the rear axis while the front axis will remain fixed. There will be a minor slippage because of that and it will be taken care of by the software and the path correction algorithms we will talk about later. Figure 4.1 shows the car initial design.

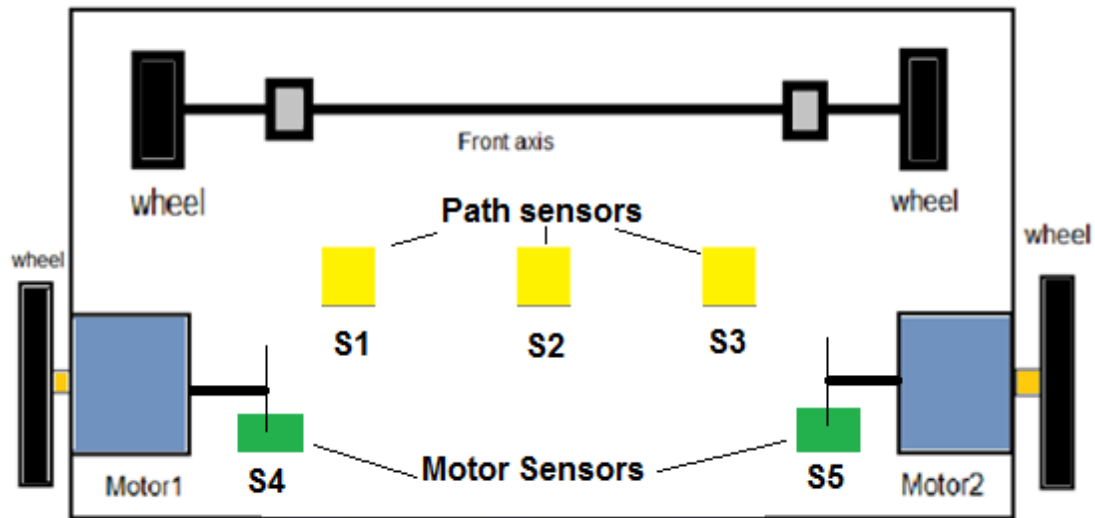


Figure 4.1: the car design

We will make the robot as small as possible to make it lighter. Thus, easier to move. In figure we show the primary design for the body.

#### 4.1.2 The wheels

The wheels are firmly related to slippage and movement accuracy. Their diameter is also related to movement speed; bigger wheels will give higher speed but lower force, while small ones will give lower speed with higher force. And that's for the same rotation speed of a given motor.

The front wheels are going to be smaller than rear in order to avoid contact with the plastic platform. But the base we will fix the front axis to is supposed to provide a suitable altitude to evade hitting low objects and provide us with good mobility.

While the rear axis is not going to be a real axis. The wheels are going to be directly fixed to the motors in order to produce less friction compared to the one produced from having another axis. That will also allow us to decrease the altitude of the robot at the rear to allow better performance of the photo couple. The design is going to be as we show in figure 4-2.

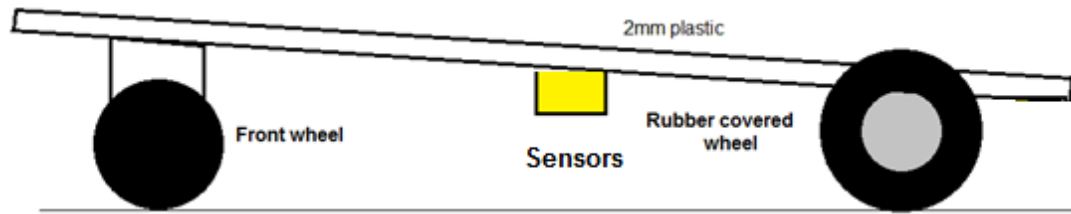


Figure 4.2: the wheels

The Adriano and the motor controlling unit as well as the batteries will be placed firmly on the body's surface as it's shown in figure 4.3.

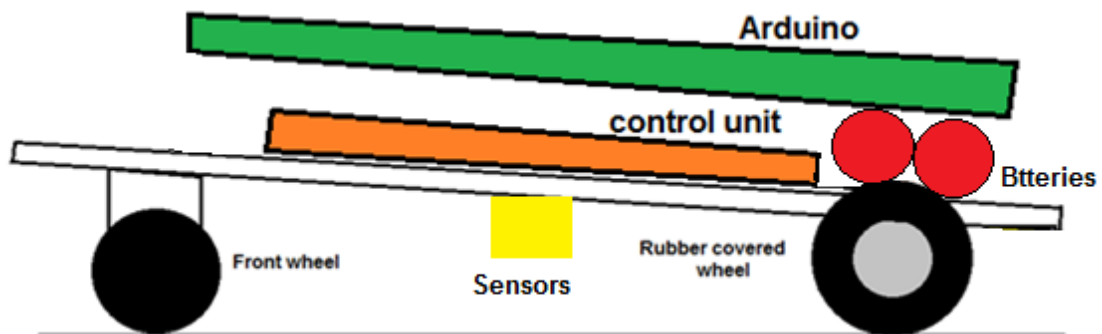


Figure 4.3 placing the components on the car.

#### 4.1.3The motors:

Using two stepper motors is highly beneficial in the robotics field. But, the stepper motors are more prone to noise than DC motors. In order to get rid of the noise signals we need to implement the circuit shown in Figure 4.4

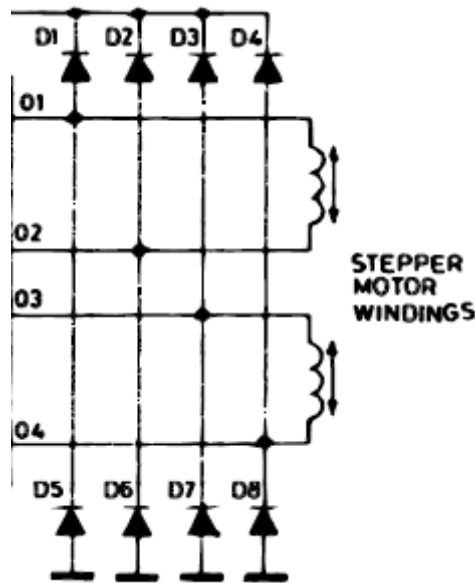


Figure 4.4 noise-free Stepper motor circuitry[4]

Installing the diodes as shown in the previous figure will negate the effects of noise on the motor. Which will guarantee it to operate properly and effectively. But, even when reverse biased. The diodes will still allow a small current to pass. Which will increase the power consumption in the robot. We will also need two bridges to control the motors. Which will also increase the size and the power consumption. Thus, we eliminated the option of using stepper motors in our robot. And will use DC motors and drive them with one L298 H-bridge. Figure 4.5 shows the needed controlling circuitry using the L298 H-bridge and two LM7805 voltage regulators.

#### 4.1.4 The power source:

The robot needs a power source that can be carried with it to enable it to freely roam within the maze. The power source can be batteries. But we need to determine the type of batteries we need to use. During researching we encountered multiple types of batteries, the lead-acid wet batteries and the lithium-ion dry batteries.

By comparing the two types to each other we found that lithium-ion batteries are better for our robot for the reasons we included in the table 4-1.

Table 4-1 comparison between batteries.

| Battery              | Lead-acid       | Lithium -ion     |
|----------------------|-----------------|------------------|
| Rechargeable         | Yes             | Yes              |
| Requires maintenance | Yes             | No               |
| Capacity             | Less than 300mA | More than 4000mA |
| Voltage              | 6/9/12 volts    | 3.7 volts        |

The table compares two batteries of the same size (5cmX3cmX1cm) we notice that the lead-acid battery stores too little energy compared to the lithium-ion battery. And since we are building a robot that solves mazes, we need to have enough energy to ensure solving the maze before the batteries are exhausted.

We will use two lithium-ion batteries and connect them serially to achieve a voltage of  $3.7+3.7=7.4v$

## 4.2 Operating the motors:

We need to be able to control the speed and the direction of two DC motors. And that can be easily achieved by using one L298 H-bridge. Which is commonly used for controlling stepper motors as well. Figure 4.5 shows the controlling circuitry.

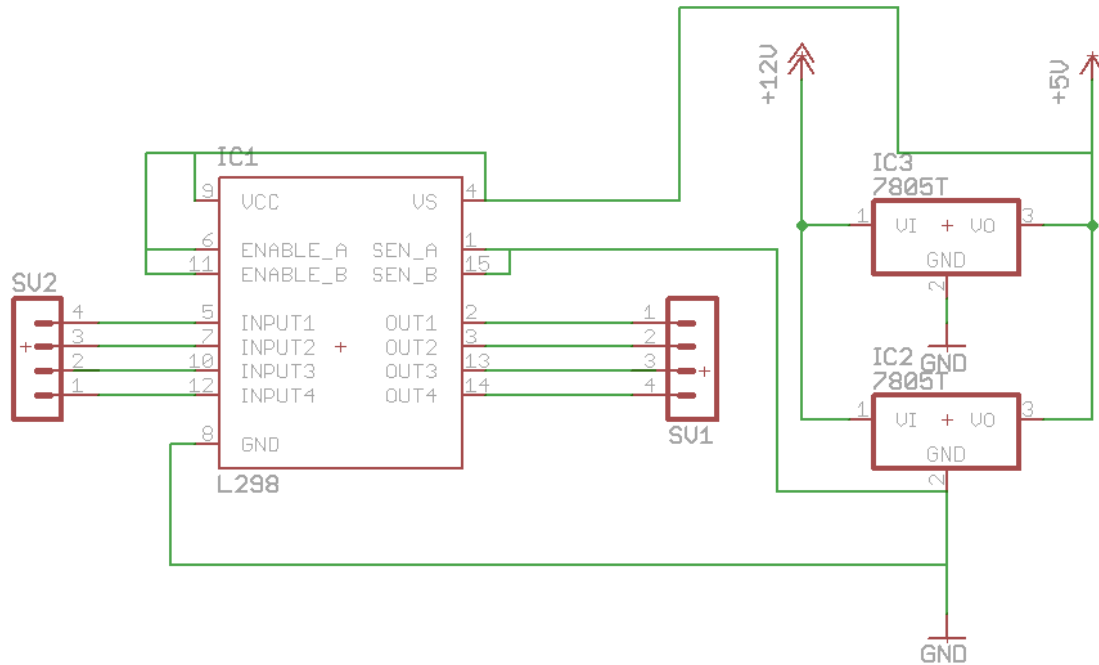


Figure 4.5 the motors' controlling circuitry

The SV1 pin slot is dedicated for the controller's output. Which will form the power supply for the motors. Where OUT1 and OUT2 will be connected to the first motor and OUT3 and OUT4 will be connected to the second motor. While the control signals will come through SV2 pin slot to the input pins in the controller.

Moving DC motors usually faces the speed issue. The motors won't be able to move the body unless they are operated in a voltage high enough to reach a suitable velocity. But, that will lead the robot's speed to increase gradually. Which will make stopping extremely hard. And to solve this issue we need to control the motors' speed, which can be controlled by changing the voltage, the current, or by using Pulse Wave Modulation (PWM) which is easy to achieve using the software.

The following pseudo code shows the required coding to move the robot with DC motors

```
Moveforward()  
{  
    Firstmotor = on;  
    Secondmotor = on;  
    While (stopping condition not met)  
    {  
        Firstmotor = on;  
        Secondmotor = on;  
        Wait(5 milliseconds);  
        Firstmotor = off;  
        Secondmotor = off;  
        Wait(5 milliseconds);  
    }  
}
```

#### **4-3 sensors:**

The photo couplers will be connected to the 3.3v pin in the Arduino. And they will be connected to the analog pins to facilitate reading them.

Having a 220K resistor connected in parallel with the output port.

The robot will have three sensors. One in its center and two closer to the sides to detect routes and response to them. In the figure 4-6 we show how the sensors are connected to the Arduino.

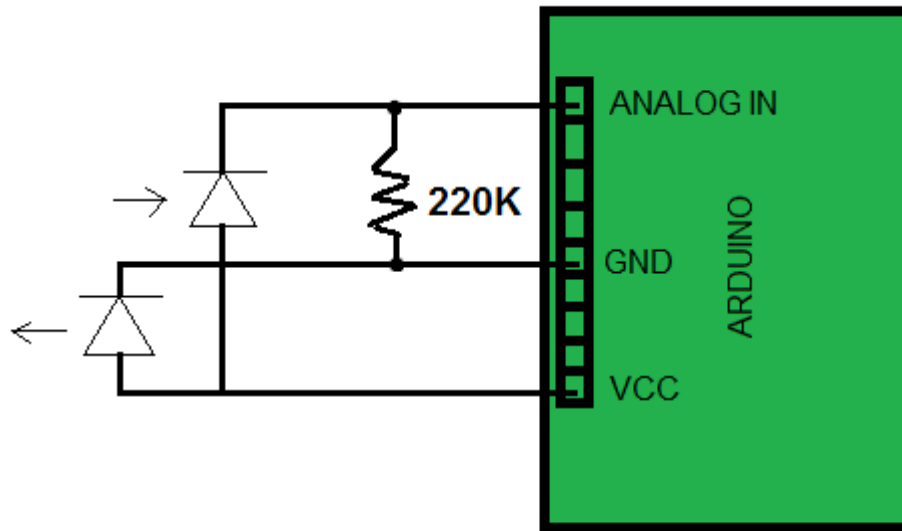


Figure 4.6 connecting a sensor to Arduino.

The receiving diode will change its resistance according to the amount of light reflected from the surface the sending diode lights, which will decrease the voltage on it. Since the maze will be represented with a black line, the robot will see a way when the analog input is in its minimum.



#### 4.4 Arduino Duo:

Our main logical unit is an Arduino Duo board shown in figure 4-12

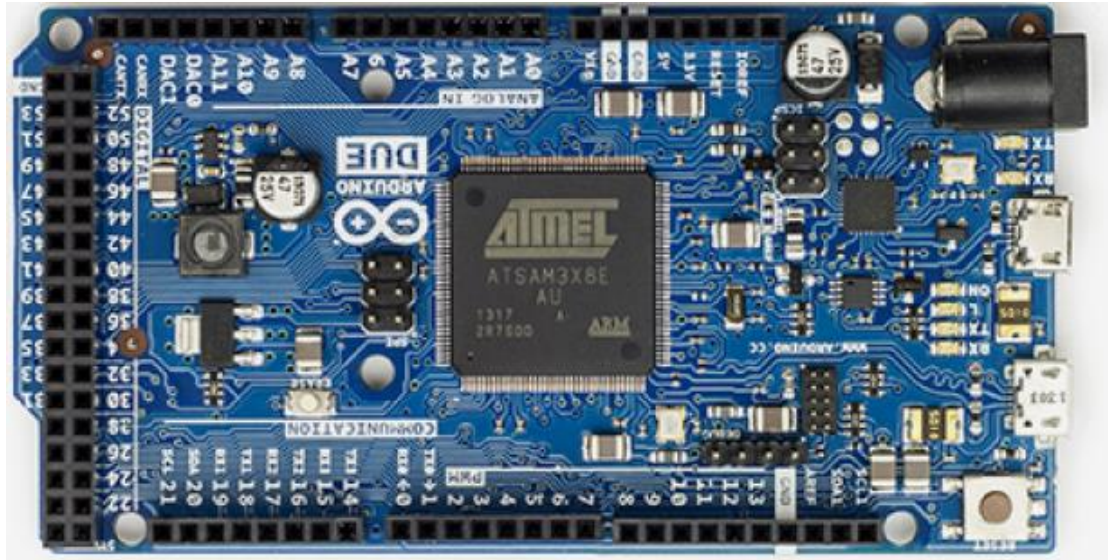


Figure 4.7 Arduino Duo

The software will reside on the Arduino Duo board, where its processor will execute all the processing in this system. It will provide the robot with a processor, a main memory, and a ROM memory. And will provide the sensors with voltage to operate them.

#### 4.5 The software:

The software will operate the robot and determine its behaviors. It is consisted of a set of algorithms to response to the input and solve the maze in an efficient way. Several algorithms are applied to achieve the necessary movements in the robot such as turning left, turning right, turning around, and moving straight.

Arduino software is built into two main functions. The first is **Setup()** which runs only once, and is used to assign the static and the initial arguments. And the second one is **loop()** which runs infinitely. The behavior of the robot is described in the following pseudo-code:

Read sensors;

```
if (nowayleft AND nowayright)
{ straight(); }
```

Else

```
{ leftHandwall (); }
```

The straight() and leftHandwall() methods are also explained in the following pseudo-code.

*Straight()*

```
{
```

```
    Motor1 move forward;
```

```
    Motor2 move forward;
```

```
}
```

*leftHandwall ()*{

```
    if (leftavailable)
```

```
        {turnLeft;
```

```
        Return;}
```

```
    If (rightavailable AND noleftavailable)
```

```
        {turnRight;
```

```
        Return;}
```

```
    If (noLeftavailable AND noRightavailavble AND nosttraightavailabe)
```

```
        {turnaround;
```

```
        Return;}
```

```
}
```

And we will agree to consider having signals from both left and right sensors without having any from the center signal to be the end of the maze. When that condition is met the robot will stop searching and replay the maze in a shortened path.

Shortening the path will be conducted using the algorithm we explained. At each turn around the software will check for possible shortening; the turns will be stored in a matrix. This pseudo code will explain the method Shortpath():

```
Shortpath(){  
    If (lastTurn==left AND thirdlastTurn==right){  
        Delete last three turns;  
        LastTurn = back;}  
  
    elseif (lastTurn==right AND thirdlastTurn==left){  
        Delete last three turns;  
        LastTurn = back;}  
    elseif (
```

turnLeft(), turnRight(), and turnaround() methods operate in the same way. But with different parameters for each motor. The robot will turn around by moving one of the motors forward and the other backwards. By using the right parameters we will achieve a 90 or 180 degrees turn.

We eventually have a full design for the robot and its software. We now need to assemble it according to this design and operate it later.

#### 4.6 Expanding the maze solving theory

In 1980, Namco<sup>1</sup> developed the now-ubiquitous Pac-Man, an arcade game starring an eponymous yellow circle who the player guides through a maze, eating dots, and avoiding enemy ghosts. The goal of this project is to fully implement Pac-Man, from the graphical user interface

to the artificially intelligent ghosts. The aim, in a lot of ways, will be to effectively emulate the Pac-Man experience with AI.

#### **4.6.1 History of Pac-man original and the game's objectives**

Pac-Man is a simple predator-prey style game, where the human player maneuvers an agent (i.e. Pac-Man) through a maze. The aim of the game is to score points, by eating dots initially distributed throughout the maze while attempting to avoid four “ghost” characters. If Pac-Man collides with a ghost, he loses one of his three lives (Whenever Pac-Man occupies the same tile as an enemy, he is considered to have collided with that ghost), and play resumes with the ghosts reassigned to their initial starting location (the “ghost cage” in the center of the maze). Four “PowerPills or PowerDots” are initially positioned near each corner of a maze: when Pac-Man eats a PowerPill he is able to turn the tables and eat the ghosts for a few seconds. The game ends when Pac-Man has lost all of his lives.

In the original game there are no elements of randomness, each ghost makes deterministic decision about which direction to move at a given time step. Since the ghosts movement is dependent on the position and movement of the Pac-Man agent. As long as the Pac-Man agent displays variety in play, the game dynamics will unfold differently. This means that typical human players develop strategies for playing the game based on task prioritization, while planning and risk assessment have part in making any ghost agent AI the basic behaviors of the ghost agents can be described, the precise (programmed) strategies in the original arcade game appear to be unknown (reason is the short of

translating machine code back into assembly language for the Z80 microprocessor and reverse-engineering!).

Pac-Man (and variant) computer games have received some recent attention in Artificial Intelligence research. One reason is that the game provides a platform that is both simple enough to conduct experimental research and complex enough to require non-trivial strategies for successful gameplay.

The game is overly beaten and old and exists in too many variants to enumerate here. (Also one of the major challenges is making it playable and distinct). The development of a unique packman game has infiltrated many fields, for example some collages use the ghost AI for teaching purposes. Plus, there is a version of the Pac-man game titled “Ms. Pac-man” which is officially used for competitions worldwide. The aim of this competition is to provide the best software automated controller for the game agent of Ms. Pac-man. This is a great challenge for computational intelligence, machine learning, and AI in general.

Unlike Pac-Man, Ms. Pac-Man is a non-deterministic game, and rather difficult for most human players. As far as we know, nobody really knows how hard it is to develop an AI player for the game. The world record for a human player scoring (on the original arcade version) currently stands at 921,360, developing a software that can beat it is what this competition is all about. Ms. Pac-Man competition will test the ability of computer-based players at the conference. We are especially interested in players that use computational intelligence methods to

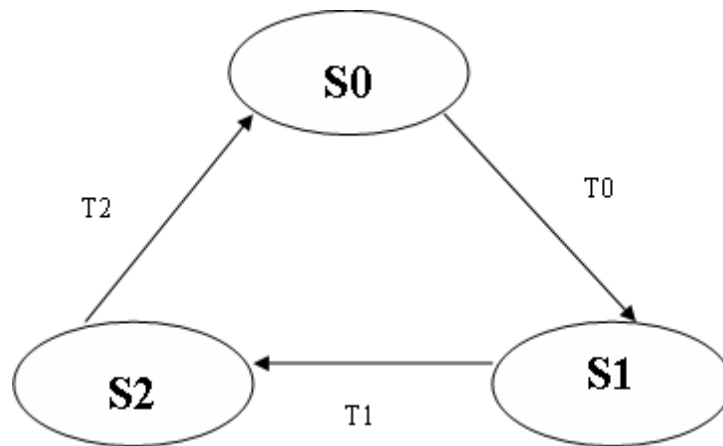
address the problem, but the contest is open to any type of algorithm: you can hand-program it as much as you like. And the results are posted by IEEE reference [5]

#### **4.6.2 Pac-man Project backbone**

This paper aims to describe the process of creating a unique playable and different Pac-man game through our development of the “Ghost agents” AI, our algorithm aims to show the importance of the Randomness factor while navigating through a maze, and how some deterministic factors combined with randomness could create a well-balanced environment for a game, the implementation and the process of building an applet, This is an intelligent Pac-man game written in Java. The original goal of the project is to demonstrate how randomness contributes to the intelligent movements of the ghosts while dealing with the game environment with all of its variables and states, the movement of the ghosts is what makes our game project unique and interesting.

#### **4.6.2 Algorithm design**

The game logic of original Pac-man game is straightforward and simple. In order to make it more interesting, game logic design is another important design part we concentrated on in this implementation. The ghosts are in one of three states as shown in figure 4.8.



**Figure 4.8:** States of a Robot Ghost

in Figure 4.8 we show that Our ghosts are built over the uses of a Finite State Machine , The ghosts have three states and three transitions in the game:

- \_ S0: Robot Ghost is walking randomly in Maze
- \_ S1: Robot Ghost meets the Pac-man (chase state)
- \_ S2: Robot Ghost dies (in the state where Pac-man eats a PowerDot)
- \_ T0: Robot Ghost transitions from S0 to S1
- \_ T1: Robot Ghost transitions from S1 to S2
- \_ T2: Reset Robot Ghost to S0

In the beginning of the game the initial state of the maze is shown in figure 4.9 Where there are four PowerDots at the corners of the maze, and all the ghosts are in a cage, Pac-man can go inside the cage, and any

ghost inside that cage won't be affected by the power pill effect (won't have any of the main states till they leave outside of it)

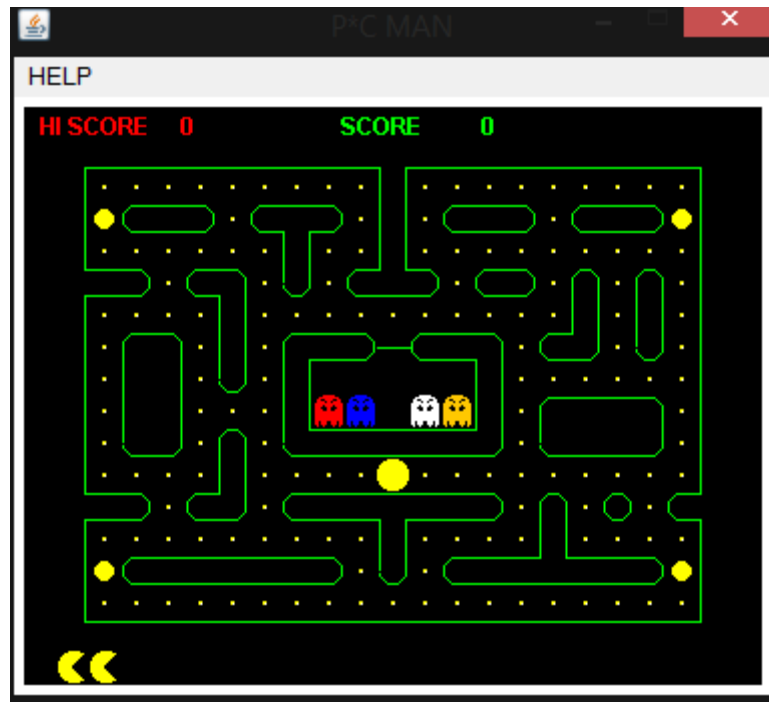


Figure 4.9 the initial state of the game.

When building the ghosts, we programmed them to start as blind for a set numbers of frames, we set their speed in that state (while walking randomly in state S0) as full speed which is same as the speed of Pac-man, the ghosts have knowledge of the maze at all times, and move through iX and iY axes inside the maze (i.e. always knows its place inside the maze), the ghost in this state will be choosing one of the available directions randomly at every intersect. Once the number of the blind frames is zero, the ghost will change states into the second state of “ghost chasing Pac-man S1” this is when the behavior of ghost takes smart AI movement, where they have priorities while choosing the random directions, the ghosts will go roam around the power dots, till it



spots the Pac-man (the PowerDots are located in one of the four corner blocks of the maze, one in each corner, Four in total, its score will be added to the calculations that affect the second state path choice (described more later).

Once the number of the blind frames is zero, The ghosts will change into the third state which is “Flee state S3” as shown in figure 2.3, in that state they will change color, and instead of chasing Pac-man, they will flee to a changeable fixed point outside the maze where they will always keep roaming, the speed of the ghosts will be at half of the normal speed and Pac-man will be able to eat them at that state, resetting them back to the initial starting position (inside the cage in the middle of the maze) where they will be reset, after a fixed frame time, the powerDot effect will fade, and the ghosts will be back to the initial random state to restart the loop of states. We show this state in figure 4.10.

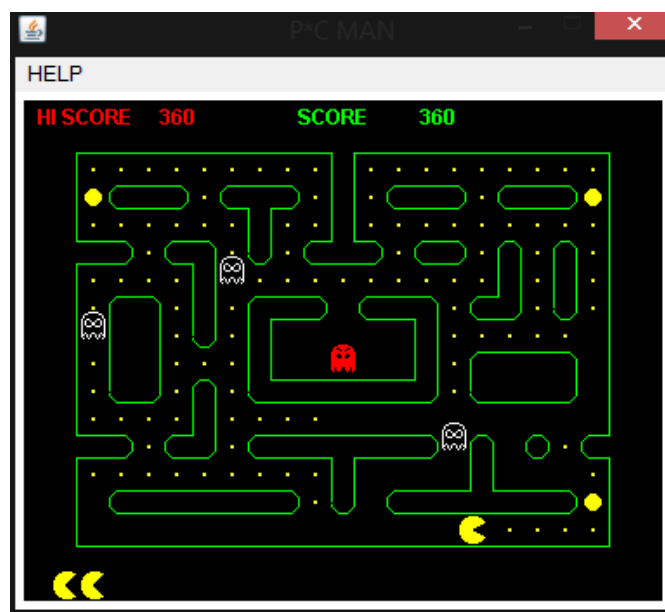


Figure 4.10: showing the flee state S3

And with the suggested implementations both of the software and the hardware of our project are ready to be implemented and developed. We now need to focus on implementing it properly.

## **Chapter 5: implementations**

## 5.1 The body

We installed the parts we designed earlier together and tested them to ensure their performance. Figure 5.1 shows the final form of the robot.



figure 5.1 the robot's final form

## 5.2the code

The Arduino's two main programs has been set and uploaded to the board. **setup()** and **loop()** are implemented as follows.

We need first to define some important variables in the global scope, in order to use them in all of the methods we need to form the right hand algorithm. The following figure shows the defined variables. The code in figure 5.2 shows the variables in the global scope of our program.

```
#define leftSensor      A2
#define rightSensor     A3
#define centerSensor    A4

int leftReading;
int rightReading;
int centerReading;

#define leftMotor1  2
#define leftMotor2  4

#define rightMotor1 7
#define rightMotor2 8

#define led 13

char path[30] = {};
int pathLength;
int readLength;
```

Figure 5.2 the global scope in the software.

The sensors are connected to the analog pins A2, A3, A4 as the figure shows, while the motors are connected to {2,4} and {7,8} which are digital output PWM pins. While in the **setup()** segment we define the pins as input or output as needed. The figure shows the segment

```
void setup() {

    pinMode(leftSensor, INPUT);
    pinMode(rightSensor, INPUT);
    pinMode(centerSensor, INPUT);

    pinMode(leftMotor1, OUTPUT);
    pinMode(leftMotor2, OUTPUT);
    pinMode(rightMotor1, OUTPUT);
    pinMode(rightMotor2, OUTPUT);
    pinMode(led, OUTPUT);
    delay(1000);
}
```

Figure 5.3 Setup() segment

Delay (1000) is a function that causes the processor to wait for 1000 milliseconds before moving to another segment, which is **loop()**. In **loop()** the robot will repeat the algorithm as long as there is an input signal from one of the sensors. As we show in figure 5.4.

```
void loop() {

    readSensors();
    if(leftReading<200 && rightReading<200)
    {
        straight();
    }
    else if ( leftReading>200 && rightReading>200 && centerReading<200)
    {
        done();
    }
    else {
        leftHandWall();
    }
}
```

Figure 5.4 the loop() segment

As we can see, `loop()` contains two methods which are `straight()` and `leftHandWall()`. These methods are implemented in separate functions. `Straight()` will prompt the robot to move forward as there will be no other options. We can see the `done()` represents the stopping function which will only occur when the stopping function is met.

The rest of the code and the functions are included in the appendix A.

### **5.3 Software implementation of Pac-man**

Our process of java coding started with a base idea to build an applet first which will contain all the features separately while the build process should be in blocks and separate classes for each function, because such projects are always made with the purpose of expansion and development, as we see in figure 5.5 the applet window which holds a play button that runs the app we built, and if at any point we decide to add more functions, or different apps, we can just add them separately to the applet page interface, the applet provided by eclipse contain a debugging function that will show any errors, the applet is implemented within “`applet.class`” which implements “`java.applet.Applet`” library provided by java.

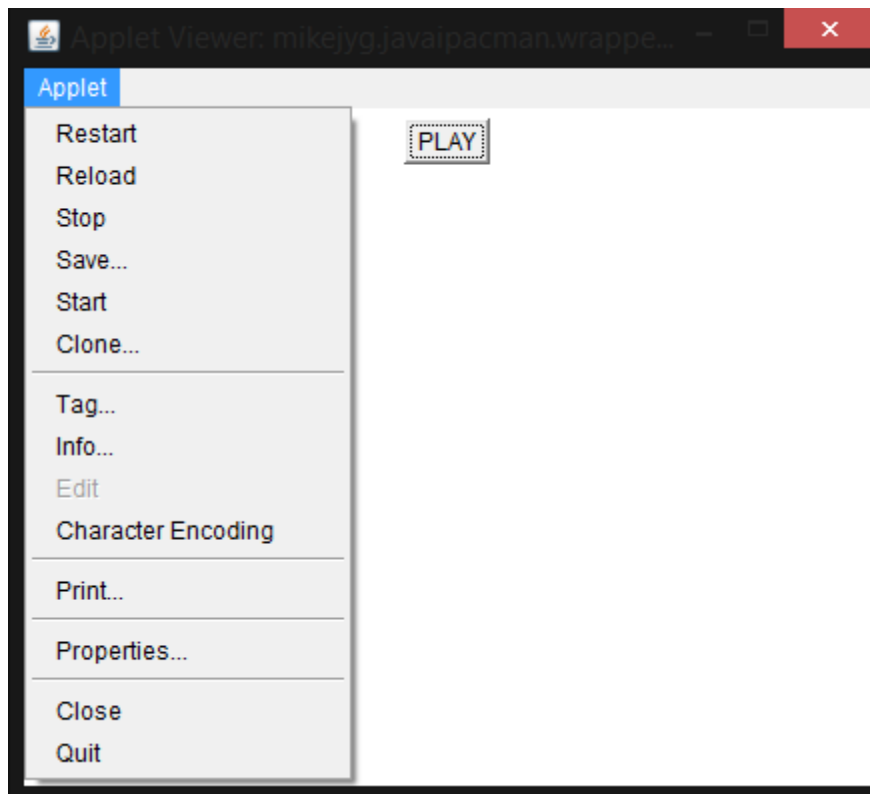


Figure 5.5 applet interface

Once Play is pressed the compiled app will be runned and it will show a window provided in figure 5.6 which is our core game that will start running and taking keyboard input once the user press the arrow Up key



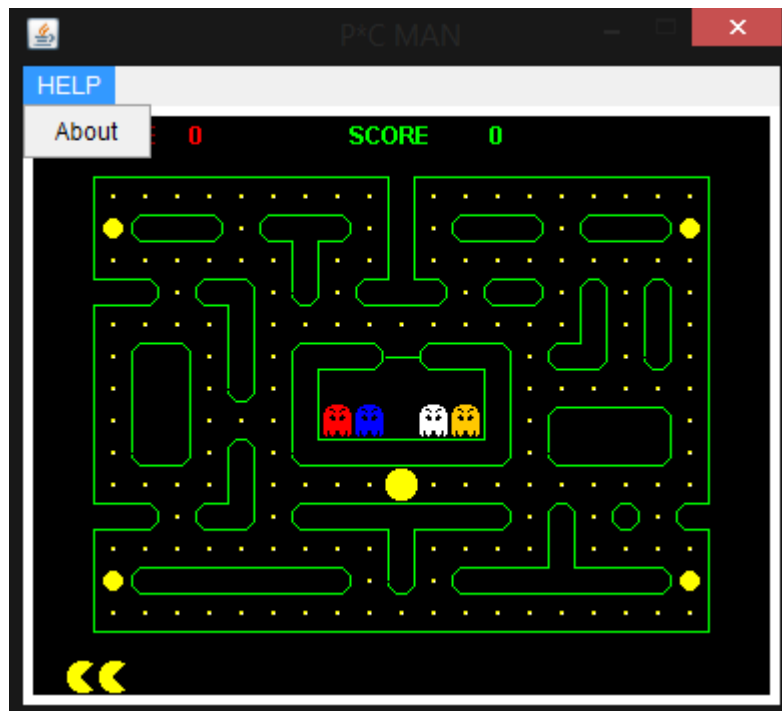


Figure 5.6 application window

## 1- Ghost AI Implementation

### Code Implementation Examples:

First we start with a ghost movement rules and decision making which we set as follows:

- 1) All states are bound to a fixed timer which is programmed as a set of frames (which we can decrease if we want the game Level to be harder, ex: scatter 7 frames, chase 7 frames, then scatter again etc.)

// remain blind for ??? frames

Final int INIT\_BLIND\_COUNT=600;

//at the start state blind counter

blindCount=INIT\_BLIND\_COUNT/((round+1)/2);

They can never (in all states) go back the same direction path as the one they came from.

```
// random calculation factors
```

```
final int DIR_FACTOR=2;
```

```
final int POS_FACTOR=10;
```

```
// the maze the ghosts knows
```

```
cmaze maze;
```

- 2) The ghost in random scatter mode will move based on available paths, but will have the ability to scatter in different directions to cover most of the maze and not leave many openings

```
// randomly select a direction
```

```
if (iDirTotal!=0)
```

```
{
```

```
    iRand=cuty.RandSelect(iDirTotal);
```

```
    if (iRand>=iDirTotal)
```

```
        throw new Error("iRand out of range");
```

```
    // exit(2);
```

```
    for (i=0; i<4; i++)
```

```
    {
```

```
        iM=maze.iMaze[iY/16+ ctables.iYDirection[i]]
```

```
[iX/16+ ctables.iXDirection[i]];

if (iM!=cmaze.WALL && i!= ctables.iBack[iDir]
&& iM!= cmaze.DOOR )

{

    iRand-=iDirCount[i];

    if (iRand<0)

        // the right selection

        {

            iDir=i; break;

        }

    }

}

else

    throw new Error("iDirTotal out of range");

    // exit(1);

    return(iDir);

}
```

3) They will try and choose different paths when in chase mode (the ghost won't just follow you, they will try and surround you from different directions) this would really show when the game is almost over, where the Ai will use calculations and accounting factors to make a surrounding move of the Pac-man agent, some of the factors affecting our decision making process were implemented as followed:

- a) Ghost status.
- b) Ghost position to each other and in the maze.
- c) Ghost movement direction and choice when chasing.
- d) Dot positions.
- e) Only the closest dot is goal.
- f) PowerDot position.
- g) The closer the ghost to a Pac-man, more weight to go to power dot.
- h) Direction based on score (disperse to cover around all of the dots that are not yet eaten by Pac-man) They will disperse in different numbers based on the location score (cover all the maze at all times), the location with higher score will have more ghosts.
- i) If the location scores is not different, or not available, the opposite score will be chosen and the ghost will change priorities to catch Pac-man while moving between the four corners of the maze Then they will compare and calculate for a new decision again.

```
public int GetNextDir()
```

```
throws Error
```

```
{
```

```
int i;

// first, init to 0

for (i=0; i<4; i++)

    iDirScore[i]=0;


// add score for dot

AddDotScore();

// add score for ghosts

AddGhostScore();

// add score for powerdot

AddPowerDotScore();


// determine the direction based on scores

for (i=0; i<4; i++)

    iValid[i]=1;

    int iHigh, iHDir;
```

```
while (true)

{

    iHigh=-1000000;

    iHDir=-1;

    for (i=0; i<4; i++)

    {

        if (iValid[i] == 1 && iDirScore[i]>iHigh)

        {

            iHDir=i;

            iHigh=iDirScore[i];

        }

    }

    if (iHDir == -1)

    {

        throw new Error("cpacmove: can't find a
way?");

    }

}
```

```
        if ( cPac.iX%16 == 0 && cPac.iY%16==0)

        {

            if ( cPac.mazeOK(cPac.iX/16 + ctables.iXDirection[iHDir],
cPac.iY/16 + ctables.iYDirection[iHDir]) )

                return(iHDir);

        }

        else

            return(iHDir);

        iValid[iHDir]=0;

        iDirScore[ctables.iBack[iHDir]] = iDirScore[iHDir];

    }

    //calculating score example

void AddGhostScore()

{

    int iXDis, iYDis;    // distance

    double iDis;        // distance
```

```
int iXFact, iYFact;

// calculate ghosts one by one
for (int i=0; i<4; i++)
{
    iXDis=cGhost[i].iX - cPac.iX;

    iYDis=cGhost[i].iY - cPac.iY;

    iDis=Math.sqrt(iXDis*iXDis+iYDis*iYDis);

    if (cGhost[i].iStatus == cGhost[i].BLIND)
    {

    }

    else
    {

        // adjust iDis into decision factor
    }
}
```



```
iDis=18*13/iDis/iDis;
```

```
iXFact=(int)(iDis*iXDis);
```

```
iYFact=(int)(iDis*iYDis);
```

```
if (iXDis >= 0)
```

```
{
```

```
    iDirScore[ctables.LEFT] += iXFact;
```

```
}
```

```
else
```

```
{
```

```
    iDirScore[ctables.RIGHT] += -iXFact;
```

```
}
```

```
if (iYDis >= 0)
```

```
{
```

```
    iDirScore[ctables.UP] += iYFact;
```

```
}
```

```
else
```

```

        {

                                iDirScore[ctables.DOWN] += -iYFact;

        }

    }

}

}

```

- 4) When the powerDot is taken by Pac-man, the ghosts will be in S3, they will scatter at half of Pac-man speed, and will have a changeable unreachable point to go to, which is located outside the maze frame (meaning the ghost will keep scattering but not to a predictable point, because random moving agent would be presented on the iX iY access)

#### 5) The win state

```

// return 1 if caught the pac!

// return 2 if being caught by pac

int testCollision(int iPacX, int iPacY)

{

    if (iX<=iPacX+2 && iX>=iPacX-2

```

```
    && iY<=iPacY+2 && iY>=iPacY-2)

{

    switch (iStatus)

    {

        case OUT:

            return(1);

        case BLIND:

            iStatus=EYE;

            iX=iX/4*4;

            iY=iY/4*4;

            return(2);

        }

    }

    // nothing

    return(0);

}
```

- 6) Special case movements and conditions for example dealing with the home of the ghosts (the box where they start, and go back to when they get eaten by Pac-man, and where Pac-man cant inter)

```
if (iStatus==BLIND || iStatus==OUT)
```

```
{
```

```
    iStatus=BLIND;
```

```
    iBlindCount=blindCount;
```

```
    iBlink=0;
```

```
    // reverse
```

```
    if (iX%16!=0 || iY%16!=0)
```

```
    {
```

```
        iDir= ctables.iBack[iDir];
```

```
    // a special condition where the ghosts always leave the home (the box  
    in the middle of the maze where they start, and where Pac-man cant  
    inter) they leave blind (in random scatter mode)
```

```
    int iM;
```

```
    iM=maze.iMaze[iY/16+ ctables.iYDirection[iDir]];
```

```
    [iX/16+ ctables.iXDirection[iDir]];
```

```
if (iM == cmaze.DOOR)
```

```
    iDir=ctables.iBack[iDir];
```

## 2- Game Environment implementation

Here we show some of the core classes as a walkthrough of the building blocks and features:

- 1- cpcman: the main class of our Pac-man game containing the base of our build the game variables such as score Pac-man lives (number of time you can restart before game over) and the key components for a healthy initial start of a level

```
// the timer
```

```
    Thread timer;
```

```
    int timerPeriod=12; // in milliseconds
```

```
// the timer will increment this variable to signal a frame
```

```
    int signalMove=0;
```

```
// the canvas starting point within the frame
```

```
    int topOffset;
```

```
    int leftOffset;
```

```
// the draw point of maze within the canvas
```

```
    final int iMazeX=16;
```

```
    final int iMazeY=16;
```

```
// the off screen canvas for the maze
```

```
Image offScreen;
```

```
Graphics offScreenG;
```

```
// the objects
```

```
cmaze maze;
```

```
cpac pac;
```

```
cpowerdot powerDot;
```

```
cghost [] ghosts;
```

```
// game counters
```

```
final int PACLIVE=3;
```

```
int pacRemain;
```

```
// to signal redraw remaining pac
```

```
int changePacRemain;
```

```
block listing some of the key objects initialization
```

```
// initialize maze object
```

```
maze = new cmaze(this, offScreenG);
```

```
// initialize ghosts object

// 4 ghosts

ghosts = new cghost[4];

for (int i=0; i<4; i++)

{

    Color color;

    if (i==0)

        color=Color.red;

    else if (i==1)

        color=Color.blue;

    else if (i==2)

        color=Color.white;

    else

        color=Color.orange;

    ghosts[i]=new cghost(this, offScreenG, maze,color);

}

// initialize power dot object

powerDot = new cpowerdot(this, offScreenG, ghosts);
```

```
// initialize pac object
```

```
pac = new cpac(this, offScreenG, maze, powerDot, ghosts);
```

```
pac = new cpac(this, offScreenG, maze, powerDot);
```

the block which is setting the maze dimension and painting everything

```
// set the proper size of canvas
```

```
Insets insets=getInsets();
```

```
topOffset=insets.top;
```

```
leftOffset=insets.left;
```

```
// updating the frame
```

```
powerDot.draw();
```

```
for (int i=0; i<4; i++)
```

```
    ghosts[i].draw();
```

```
pac.draw();
```

```
// display extra information
```

```
if (changeHiScore==1)
```

```
{
```



```
imgHiScoreG.setColor(Color.black);

imgHiScoreG.fillRect(70,0,80,16);

imgHiScoreG.setColor(Color.red);

imgHiScoreG.drawString(Integer.toString(hiScore),
70,14);

g.drawImage(imgHiScore, 8+ leftOffset, 0+ topOffset,
this);

changeHiScore=0;

}

if (changeScore==1)

{

imgScoreG.setColor(Color.black);

imgScoreG.fillRect(70,0,80,16);

imgScoreG.setColor(Color.green);

imgScoreG.drawString(Integer.toString(score), 70,14);

g.drawImage(imgScore,

158+ leftOffset, 0+ topOffset, this);

changeScore=0;

}
```

```
// update pac life info

if (changePacRemain==1)

{

    int i;

    for (i=1; i<pacRemain; i++)

    {

        g.drawImage(pac.imagePac[0][0],

            16*i+ leftOffset,

            canvasHeight-18+ topOffset, this);

    }

    g.drawImage(powerDot.imageBlank,

        16*i+ leftOffset,

        canvasHeight-17+ topOffset, this);

    changePacRemain=0;

}
```

There is also some key factors for a game environment that we won't be listing the code for such as “Control and moves” block which is the routine running at the background of drawings and the routine that draw each frame (updating the maze objects as game progress), followed by” process key input” (keyboard input for controlling the Pac-man agent), followed by the handler of the menu event (buttons and such), handling the timer (stopping the clock at death, restarting the game etc.)

## 5.4 results

### 5.4.1 The robot:

After running the robot we tested it on the 6X6 maze specified in figure 5.7. the robot took 3:12 minutes to solve it and took the path shown in figure 5.8.

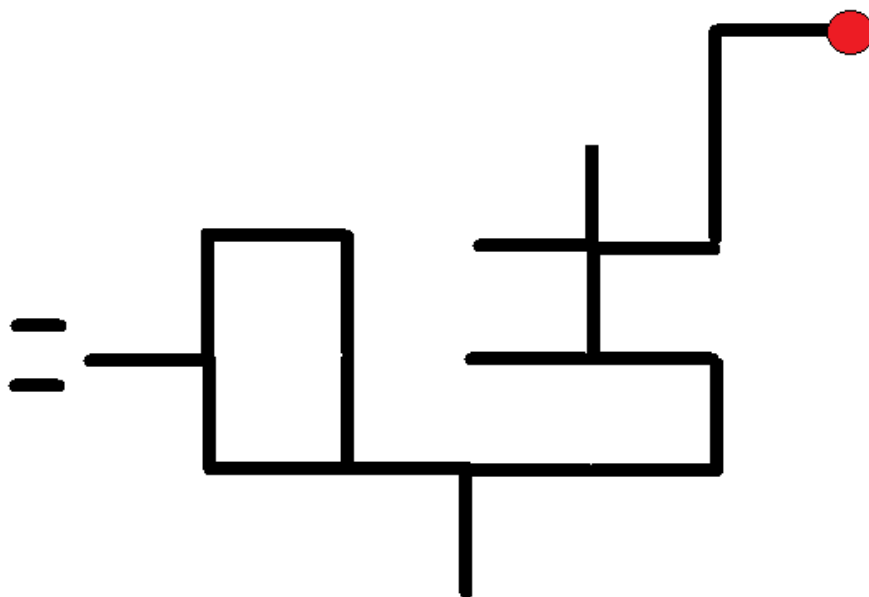


Figure 5.7 the tested maze

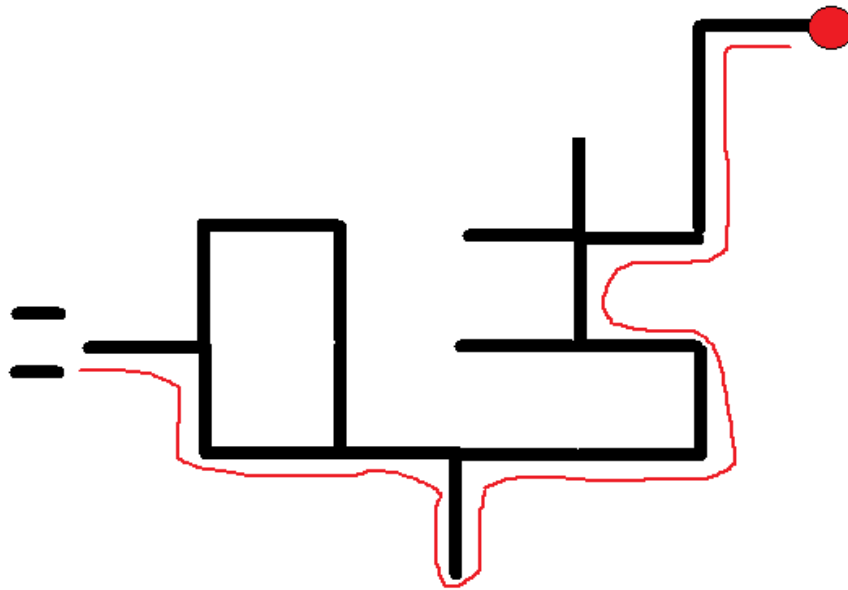


Figure 5.8 the followed path

After activating the shortening algorithm the path and the solving time improved. The maze was solved with only 2:52 minutes and the new path became as we show in figure 5.9

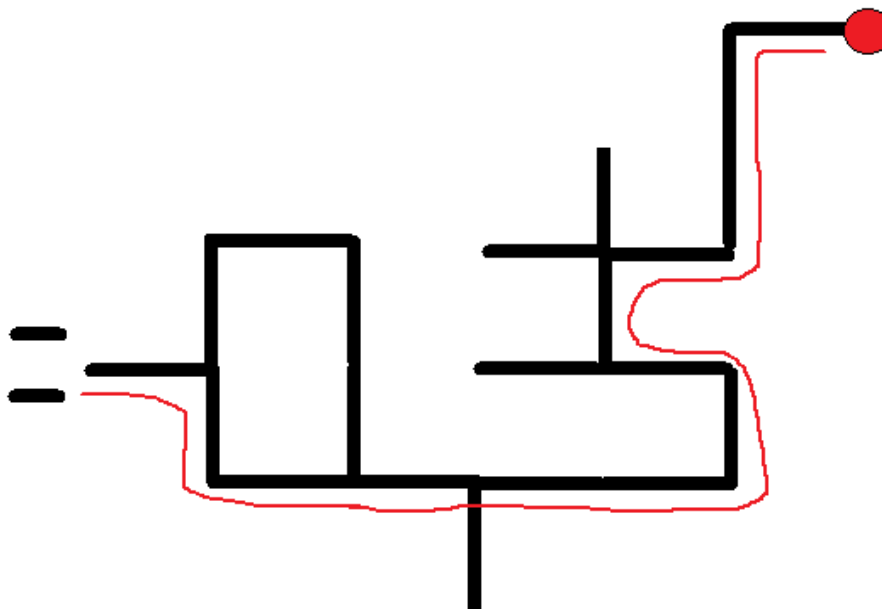


Figure 5.9 the final solution of the maze

#### 5.4.2 Pac-man

What approaches have others tried. Here we go over other existing open-source projects that attempt several. Approaches (but not necessarily complete the actual implementation) and their main features, which we can use to compare or think of expanding our current project further:

- First our current project with the “Ghosts AI” offers a unique game experience, with the randomness factor which will always make the game Dynamic with the overall experience being unpredictable, it was built with development and compile time standards in mind, even our resources were made to compile fully at any point, without the need of resources being loaded (such as adding the maze as a picture, or the ghosts as icons, or making the objects within the maze unique object, which would have cause slow compiling, and bad reaction time from the ghost part), also it takes up the least amount of memory working, so it can be reused elsewhere or expanded upon, one of the examples of our work, is the maze way of being drawn:

```
public static final String[] MazeDefine=
{
    "XXXXXXXXXXXXXXXXXXXXX",    // 1
    "X.....X.....X",        // 2
    "XOXXX.XXX.X.XXX.XXXOX",    // 3
    "X.....X..X.....X",        // 4
    "XXX.XX.X.XXX.XX.X.X.X",    // 5
    "X....X.....X.X.X",        // 6
    "X.XX.X.XXX-XXX.XX.X.X",    // 7
    "X.XX.X.X      X.....X",    // 8
}
```

```

    "X.XX...X      X.XXXX.X",    // 9
    "X.XX.X.XXXXXXX.XXXX.X",    // 10
    "X...X...  ....X",          // 11
    "XXX.XX.XXXXXXX.X.X.XX",    // 12
    "X.....X...X...X...X",    // 13
    "XXXXXXXXX.X.XXXXXXXOX",    // 14
    "X.....X",                // 15
    "XXXXXXXXXXXXXXXXXXXXXXXXX",  // 16

    };

```

- "Pacman Arena is a Pacman clone in full 3D with a few surprises. Rockets, bombs and explosions abound."
- "PANP" is a new generation of the known game Pacman offering a 3D graphic interface, it provides 4 different views of game. The most outstanding feature is the possibility of playing net games against other players. It require OpenGL support."
- "A multiplayer team-based Pacman game with exciting, fast-paced action."
- "Smart Pacman: Take Pac-man's or ghosts' side in the challenge. Ghosts are controlled in the same way as units in a chase game. You can play with AI or with a friend on a single keyboard."
- "Pacman Tag: The playground game of "t" or "tag" combined with the arcade game "Pacman". One player is "it" and has to tag another player to stop their counter from ticking up. Whoever has been it for the least amount of time by the end of the game wins."

- "PacMan X is a fully featured PacMan implementation Pacman for Sharp Zaurus is a multi-board Pacman game developed for the Qtopia environment. It was developed at Auburn University for a Senior Design project. The game is quite playable, with most features from the classic Pacman game in place already."
- "Phoenix PacMan : It is an advanced PacMan Game with four types of monsters, openable walls and lot of items. it written in Java, and requires JDK 1.3"
- "This project intends to build a winter version of Pac Man, which will be programmed under OpenGL/GTK environment."
- "Mango Quest is a 3D arcade game using OpenGL and SDL, which aims to extend the pacman gameplay using a 1st person view and tons of special items. It runs on both Linux and Win32 platforms."
- "The game is PacMacro , the live action version of PacMan. We select a 5x6 block area of a major metropolitan area and play PacMan on a large scale. On the street we have the ghosts: Inky, Blinky, Pinky, and Clyde all chasing down the ever elusive PacMan."
- "A network multiplayer Pacman clone , tweaked for fast and exciting gameplay!"
- "A Multiplayer version of Pacman with the focus on Death- match, programmed in the SDL. Later on it will feature Bots, A Level editor, more game types." [7]

## Challenges

- During the implementation I got many problems with outOfBounds , and empty Lists.
- The tweaking of values for costs was tricky, and it might require a deeper analysis for optimal values. I determined the values by experimentation and tests for an overall best play feeling.
- It was a challenge to have agents for Ghosts, since creating hard enemies is not a good idea from the player point of view, but it is needed to make something different from any other.
- Bugs: the compiling and building process was tough, I had to take many different approaches to implement the maze the graphics the controls and the overall algorithm without it causing any runtime errors, or slow movement, or clock accounting issues.



## **Chapter 6: Conclusions and Future Work**

After fully implementing both the maze solving robot and the Pac-man implementation we came out with some insights and conclusions.

## **4.1 Future improvements**

The project has so much improving potential. Many chances of improvement that require further analysis have appeared during our researching and implementing

### **4.1.1 The robot**

We will work on improving the movement algorithm to improve it and reduce the time required to solve a maze. Both of the solving and the shortening algorithms have improvement potential. And we will seek for weak points and try to evade them in the future.

### **4.1.2 Pac-man**

We have started working on an automation factor for the Pac-man, the computer controlled auto play of the Pac-man agent feature is work in progress and does not work properly yet, we are building it while taking some factors into considerations and are set to follow these steps:

- 1- Building a Pac-man automation that will learn through trying and is able to beat our current project ghosts.
- 2- A Pac-man automation that is able to beat the original Pac-man game with a high score.
- 3- A Pac-man that have a unique changeable AI that can adapt to hard mode (ghost state timers are set to have chase at most times).
- 4- Having the ability to implementing a smarter ghost AI that is able to chase the Pac-man itself in a set of coordinated states unique to each of the ghosts.

## **4.2 Conclusion**

Maze solving might look a narrow field. But, we actually found during our researching the problem is very important in our daily modern life. It also has much improvement potential and can add to our knowledge as computer systems engineers.

## References

[1] <http://wikipedia.org>

[2] <http://Arduino.cc>

[3] Encyclopedia of Electronic Components Volume 1, O'Reilly Media.

[4] Robot Builder's Bonanza, @nd ed 2001

[5] Pac Man papers in IEEE CIG

<http://www.csse.uwa.edu.au/cig08/Proceedings/search.html?cx=000061790505554880205:oanpsvp2jsm&cof=FORID:10&ie=UTF-8&q=pacman&sa=Search#393>

[6] [IEEE Symposium on Computational Intelligence and Games](#)

[7] <http://www.findbestopensource.com/> the biggest site offering a search engine and download of many open source applications.

## الملخص:

خلال حياتنا، سنواجه العديد من المشاكل التي تتطلب الحل. وقد لاحظ العلماء هذه المشاكل واعتبروها من الأمور الضرورية من أجل البحث عن أساليب تحسين حياتنا، ومواجهة العقبات والمشاكل والتطور من خلالها.

مع تقدم التكنولوجيا أصبحت الآلات متقدمة على الإنسان في نواحي سرعة إنجاز العمليات الحسابية والقدرة على تحمل مشاق العمل وظروفه القاسية، كما أنه من الممكن إضافة بعض الطرقات إليها لتوسيع قدراتها. وبذلك الميزات تستطيع الآلات تحسين حياتنا وإنجاز بعض من المهام الصعبة. وفي هذا التقرير، نسعى كطلاب إلى تعلم وتطبيق تجربة ينطلي عليها ما سبق ذكره.

كلية الهندسة والتكنولوجيا

قسم هندسة نظم الحاسوب

# دراسة تصميم روبوت ذكي بمنصة اردوينو

إعداد

أكرم الخالد

عبد الهادي زكور

إشراف

د. لارا قديد

مشروع تخرج

2016