The Maze Runner – Zumo version

Alexander Kirillov

URL: `http://sigmacamp.org/mazerunner-zumo`

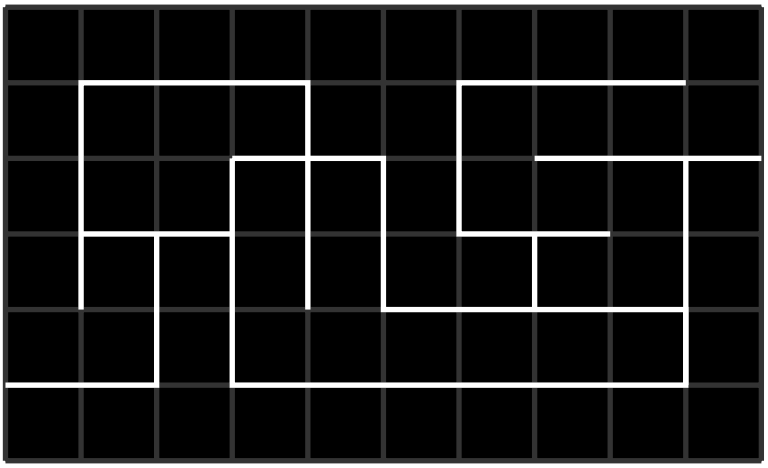
E-mail address: `shurik179@gmail.com`

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc-sa/3.0/` or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA. If you distribute this work or a derivative, include the history of the document.

Introduction

This document describes a robotics project: programming a robot which can find its way out of a maze. This was one of the projects done by students at SigmaCamp (`sigmacamp.org`), a summer science camp. Students were using Zumo-324U robots manufactured by Pololu (`https://www.pololu.com/product/3124`) and programmed them in Arduino IDE.

Description of the challenge. The maze is made of approx. 3×5 ft sheet of plywood, painted black. Maze is made out of white masking tape (0.94 in wide); these lines follow rectangular grid with 0.5 ft squares.



The goal is to build and program a robot to find its way out of the maze, using one of the three algorithms described at the end of this document.

This document details lesson-by-lesson progress of the challenge. Note that some things described here are an improved version of what was actually done in SigmaCamp.

You can download all the files necessary for this project from `http://sigmacamp.org/mazerunner2`.

Lesson 1

- Basics of Arduino IDE. Connecting the robot. First commands: `goForward(20)`, `delay(500)`, . Writing, downloading, and running the first program. Programming the robot to run repeatedly a 25-cm square. Comments.

```
#include <Wire.h>
#include <Zumo32U4.h>
#include "ZumoMaze.h"
void setup() {
  // put your setup code here, to run once:
  buttonA.waitForButton();
  delay(500);//wait for 0.5 second
}

void loop() {
  // put your main code here, to run repeatedly:
  goForward(250);
  turn(90);
}
```

- Basic control structures:

```
// if statements
if (condition) {
 ...
} else {
 ...
}
//while loops
while(condition){
...
}
```

- Sensors. Using light sensor array. Commands `calibrateLineSensors()`, `sensorOnWhite()`, `sensorOnBlack()`.
- Using a light sensor to have the robot go forward until it detects a white line:

```
readSensors();
setMotors (70,70);
while (allOnBlack()) {
 delay (20);
}
stopMotors();
```

Lesson 2

- Variables. Basic types of variables: `int`, `float`, `boolean`. Defining variables:

```
float speed=70;
```

- Project: have the robot go forward for 10cm and checking whether it saw a white line along the way:

```
boolean sawLine=false;
setMotors(speed, speed);
while (distance traveled <10) {
 delay(50);
 if (sensor sees line) {
   sawLine=true;
 }
}
```

- Using light sensor array to determine line position. Ideas of feedback and proportional correction. Following the line using light sensor array:

```
setMotors(70+0.5*linePosition(), 70-0.5*linePosition());
```

Improvement: Replacing hard-coded value of speed and coefficient by variables.

```
float  speed=70;
float  Kp=0.5;
setMotors(speed+Kp*linePosition(),
          speed-Kp*linePosition());
```

- Logical operations: and (`&&`), or (`||`), not (`!`)
  Going to an intersection:

```
boolean atIntersection=false;
setMotors(speed+Kp*linePosition(),
          speed-Kp*linePosition());
while (!atIntersection) {
    readSensors();
    atIntersection=sensorOnWhite(0) || sensorOnWhite(4);

}//end of while loop
stopMotors();
```

- Functions. Turning the above piece of code into a function `GoToIntersection()`. Functions with arguments.

Lesson 3

- Coding the simplified wall following algorithm:

```
boolean leftPassage;
void loop (){
    GoToIntersection();
    checkIntersection();
    if there is a passage to the left, turn left; otherwise do nothing
}
```

  Function `checkIntersection()` should do two things:
  - Move forward so that the center (not front!) of the robot is above the intersection.
  - While doing this, check whether there is a passage to the left and set variable `leftPassage`

- Improve the above code so that it also check for passages to the right and ahead adding checks if there is a wall ahead:

```
boolean leftPassage, rightPassage, centerPassage;
void loop (){
    goToIntersection();
    checkIntersection();
    turnAsNeeded();
}
```

  (all the logic should be in the function `turnAsNeeded`, which would use variables `leftPassage`, `rightPassage`, `centerPassage`).

Lesson 4

- Adding checks for dead ends and for end of the maze. Testing.

Lesson 5: Pledge algorithm

To fix the problems with the wall follower algorithm, one could use the Pledge algorithm, described at the end of this document. To code it, we introduce a global variable `TotalTurns`; every clockwise turn by 90 degrees increases the variable by 1, and every counterclockwise turn, decreases by 1. The overall structure of the program should be

```
void loop(){
    forwardToWall();
    turn(90); TotalTurns=TotalTurns+1;
    followWall();
}
```

where function `forwardToWall()` would go forward as far as possible, and function `followWall()` would follow the wall on the left, until the following conditions are met:

(1) we are facing the same direction as we had initially

(2) the total number of turns is zero

(In fact, as one easily sees, condition 2 implies condition 1).

Maze algorithms

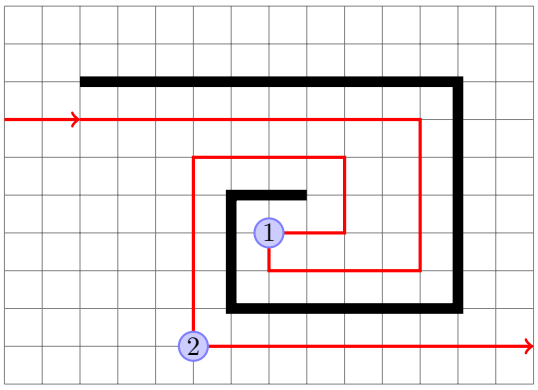
Wall follower. Start following passages, and whenever you reach a junction always follow the leftmost open passage. This is equivalent to a human walking in the a maze by putting their hand on the left wall and keeping it on the wall as they walk through. This method is not guaranteed to find an exit: the robot could be going in circles around an "island" inside the maze.

Pledge algorithm. This is a modified version of wall following that's able to jump between islands, to solve mazes wall following can't. It's a guaranteed way to reach an exit on the outer edge of any 2D maze from any point in the middle. However, it is not guaranteed to visit every passage inside the maze, so this algorithm will not help you if you are looking for a hidden treasure inside the maze.

Start by picking a direction, and always move in that direction when possible. When you hit a wall, start wall following, using the left hand rule. When wall following, count the number of turns you make, e.g. a left turn is -1 and a right turn is 1. Continue wall following until your chosen direction is available again **and** the total number of turns you've made is 0; then stop following wall and go in the chosen direction until you hit a wall. Repeat until you find an exit.

Note: if your chosen direction is available but the total number of turns is not zero (i.e. if you've turned around 360 degrees or more), keep wall following until you untwist yourself. Note that Pledge algorithm may make you visit a passage or the start more than once, although subsequent times will always be with different turn totals.

The figure below illustrates this method



Thick black lines show the walls of the maze; the red line shows the path of the robot. At point 1, robot turns to so that it is again heading the same direction as in the beginning; however, the number of turns at this point is not zero, so the robot continues following the wall. At point 2, the robot is again heading in the original direction, and the number of turns is zero, so it stops following the wall. Had the robot left the wall at point 1, it would be running in circles.

Arduino programming

Basic program structure.

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Comments. Single-line comment: everything from `//` to end of line.

Multi-line comment: everything between `/*` and `*/`:

```
 x=x+1; //this is a single line comment
 /*  and this is a
   multi-line comment */
```

Control structures.

```
// if statements
if (condition) {
 ...
} else {
 ...
}
//while loops
while(condition){
...
}
```

Logical operators. Used to construct conditions:

AND: `&&`

OR: `||`

NOT: `!`

```
while (time<1000 && !sensorOnWhite()){
...
}
```

Variables and constants.

Variable declaration:

```
int n=10;
```

Basic variable types: `int`, `long` (long integer, gives larger range), `float`, `boolean`, `String` (note uppercase!).

Constant declarations:

```
constant int CUTOFF=40;
```

Pre-defined boolean constants: `true`, `false`.

Strings. Defining a string: `String s="Hello␣World!";`

(note the double quotes).

Turning numbers into strings:

```
float x=5.17
String s=String(x);
```

Concatenating (putting together) strings:

```
String s="x="+String(x);
```

Functions.

Common functions.

`int`	`abs(int x)`	Absolute value. Instead of `int`, can also use `float`
`int`	`max(int x, int y)`	Maximum of two numbers. Instead of `int`, can also use `float`
`long`	`millis()`	Number of milliseconds since the program started
`void`	`delay(unsigned long ms)`	Pause program executions for specified number of millisecons

Zumo Bot programming

Basic program structure.

```
#include <Wire.h>
#include <Zumo32U4.h>
#include "ZumoMaze.h"
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Files `ZumoMaze.h` and `ZumoMaze.cpp` must be placed in the same directory as the sketch.

If you are using line sensors, make sure to include `calibrateLineSensors()` in `setup()`.

LCD Screen and buttons.

void	`printLcd(String s1, String s2)`	Prints the two strings to LCD, as line 1 and line 2. Note that each LCD line can only show 8 symbols.
void	`buttonA.waitForButton()`	Pause program execution until button A is pressed and released.
void	`buttonB.waitForButton()`	
void	`buttonC.waitForButton()`	

Motor control.

`void`	`goForward(long distance, int speed=70)`	Go forward/backward for specified distance, in mm. Distance must always be positive. Optional parameter `speed` determines speed (must be between 0-100)
`void`	`goBackward(long distance, int speed=70)`	
`void`	`turn(int angle, int speed=70)`	Turn by specified angle (in degrees). Positive angle gives clockwise rotation, negative gives counterclockwise rotation. Optional parameter `speed` determines speed (must be between 0-100)
`void`	`setMotors(int left, int right)`	Set the speed of both motors. Speed of each must be between -100 and 100
`void`	`stopMotors()`	Self-explanatory
`void`	`resetEncoders()`	Resets the encoders (rotation counters) on both motors. Note that encoders are also reset when you use `goForward()`, `goBackward()`.
`int`	`distanceTraveled()`	Returns distance traveled since the last encoder reset, in mm. Distance is computed by using the average of both motor encoders. Negative distance corresponds to motion backward.

Line sensors.

Zumo robot contains five sensors that measure the reflected light and are used to determine surface color underneath the robot, which in turn is used fro line following and similar tasks.

void	`calibrateLineSensors()`	Calibrates line sensors, by having the robot rotate 360 degrees and determining the highest and lowest value. This function must be called at least once before you use the sensors (usually it is done in `setup()` function).
void	`readLineSensors()`	Reads the values of line sensors (they are saved in internal variable `lineSensorValues`, which you rarely need to use directly.)
boolean	`sensorOnWhite(int sensorNum)`	Checks whether given front sensor is on white/black. `sensorNum` must be between 0 (leftmost) to 4 (rightmost). See note below.
boolean	`sensorOnBlack(int sensorNum)`	
boolean	`allOnWhite()`	Are all sensors on white? (See note below.)
boolean	`allOnBlack()`	Are all sensors on black? (See note below.)
int	`linePosition()`	Checks the values of all 5 sensors and returns the position of the white line under the robot. Returns number between -100 (lien all the way to the left of the robot) and 100 (line all the way to the right). Value 0 shows that the robot is centered on the line.

Note that functions `sensorOnWhite()`,`sensorOnBlack()`, `allOnWhite()`, `allOnBlack()` use sensor readings obtained last time `readSensors()` or `linePosition()` were called; to get up-to-date values, make sure you use `readSensors()` before immediately before calling `sensorOnWhite()` and similar commands.