The Maze Runner

Alexander Kirillov

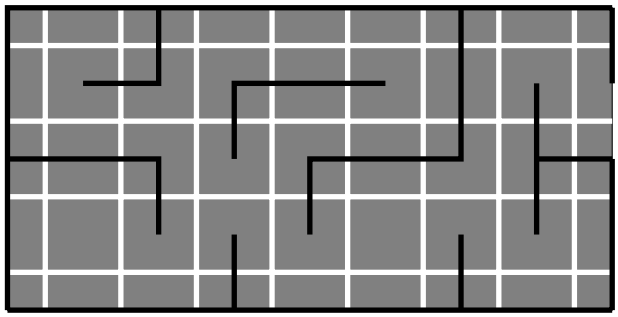
URL: `http://sigmacamp.org/mazerunner`

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

Introduction

This document describes a robotics project: building and 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 LEGO NXT sets and programmed them in RobotC.

Description of the challenge. The maze is made of 4×8 ft sheet of plywood, divided (by pencil marks) into 1 ft squares. The surface was painted black, with white masking tape (0.94 in wide) running through the centers of each 1 ft square, forming a navigation grid. Between some squares, there are walls made out of 5 in high boards; there are also walls along the perimeter of the plywood (with one exit), turning the whole surface into a maze. The figure below shows a typical maze (to better see the walls, maze surface is shown as gray; in real life, it was black).



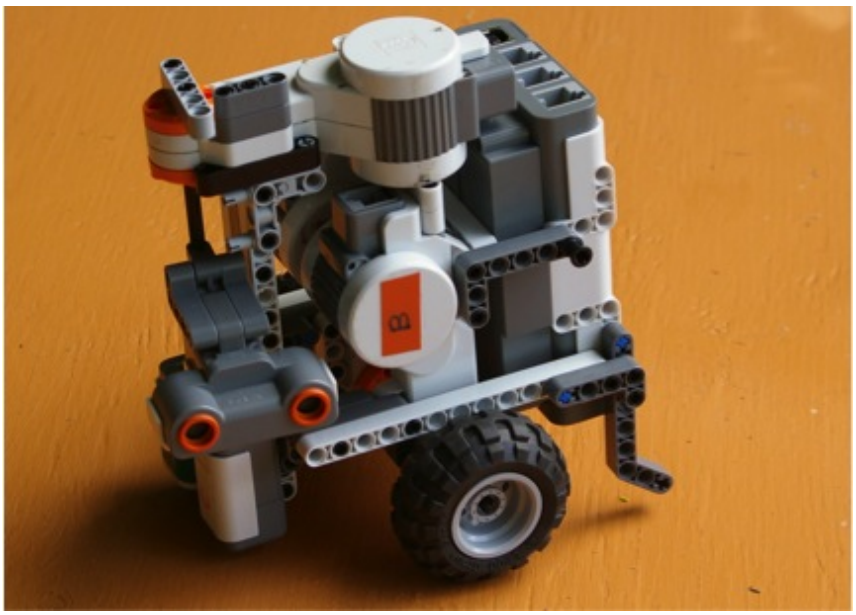
The goal is to build and program a robot to find its way out of the maze, using one of the three algorithms described at the end of this document. Students were given the building instructions for the robot, a *RobotC primer* by Alexander Kirillov, and laptops with RobotC 4.10 (Release Candidate version) and with "natural language" function library, described in the primer.

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 (building instructions, sample programs, *RobotC primer* and function library) from `http://sigmacamp.org/mazerunner`.

Lesson 1

Build the robot using the building instructions provided (see file `building_instructions.pdf`). Picture below shows the final result (without the cables).



Lesson 2

- Basics of RobotC. Connecting the robot. First commands: `motor[motorA]=100;` and `wait1Msec(500);`. Writing, downloading, and running the first program: have the robot run for one second.
- Using Natural Language. Commands `NLGoForward(distance)`, `NLStopMotors()`, and `NLTurnLeft(angle)`. Programming the robot to complete a 25-cm square (without using loops, just copy-and-paste 4 times).
- Basic control structures:

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

- Sensors. Setting sensors using Robot→Motor and Sensor Setup tool. Determining the reasonable value of the light cutoff (in this case, 40) by using `View` function of the NXT.
- Using a light sensor to have the robot go forward until it detects a white line:

```
#pragma config(Sensor, S1, LightSensor, sensorLightActive)
/*!!Code automatically generated by 'ROBOTC'
      configuration wizard               !!*/

// natural language setup
float MOVEUNIT=21.1;
float TURNUNIT=2.12;
#include "nat_language.c"
task main(){
  NLStartForward(50);//start forward at 50 % power
  while (SensorValue[LightSensor]<40){
   // do nothing
  }
  NLStopMotors();
}
```

Lesson 3

- `while (true)` loop. Following the line using two light sensors; in pseudocode, the program is

```
while (true) {
  if (both sensors see black) {
    go forward
  }
  if (left sensor sees black, right sees white) {
    go forward steering to the right
  }
  if (right sensor sees black, left sees white)  {
    go forward steering to the left
  }
}//end of while loop
```

- Variables. Basic types of variables: `int`, `float`, `bool`. Replacing hard-coded value of light cutoff (40) by a variable:

```
int LIGHT_CUTOFF=40;
```

Using a variable to break out of the loop:

```
bool AtIntersection=false;
while (!AtIntersection) {
  if (both sensors see black) {
    go forward
  }
  if (left sensor sees black, right sees white) {
    go forward steering to the right
  }
  if (right sensor sees black, left sees white) {
    go forward steering to the left
  }
  if (both sensors see white) {
    AtIntersection=true;
    NLStopMotors();
  }
}//end of while loop
NLGoForward(7); //to advance so that the center of the robot
                //is at the intersection
```

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

Lesson 4

- Using the Ultrasonic sensor. Coding the simplified wall following algorithm:

```
while (true){
 if (US sensor doesn't see a wall){
    NLTurnLeft(90);
    GoToIntersection();
 } else {
   // a wall to the left - go forward
   GoToIntersection();
 }
}//end of while loop
```

- Improving the above code, adding checks if there is a wall ahead; if there is, also checking if there is a wall to the right:

```
while (true){
 if (US sensor doesn't see a wall){
    NLTurnLeft(90);
    GoToIntersection();
 } else {
   // a wall to the left - check for wall ahead
   turn sensor 90 deg
   if (US sensor doesn't see a wall) {
     //wall to the left, no wall ahead
     turn sensor 90 degrees back
     GoToIntersection();
   } else {
     // wall to the left and ahead;
     // need to check for wall to the right
     turn sensor all the way to the right
     if (US sensor doesn't see a wall) {
       // no wall to the right
       turn sensor back all the way to the left
       NLTurnRight(90);
       GoToIntersection();
     } else {
       //walls on 3 sides - a dead end
       turn sensor back all the way to the left
       NLTurnRight(180);
       GoToIntersection();
     }
   }
 }
}//end of while loop
```

Common problem: rotating the sensor back 90 degrees sometimes fails, as the sensor can never complete the rotation and the program is stuck. Solution: rotate by time or use `NLSafeRotateMotor()`.

Lesson 5

- Improving the line follower algorithm: instead of nested ifs, use a function `TurnAlongWall()`:

```
void TurnAlongWall(){
 if (US sensor doesn't see a wall){
    NLTurnLeft(90);
    return ;
 }
 turn sensor 90 deg
 if (US sensor doesn't see a wall) {
    //wall to the left, no wall ahead
    turn sensor 90 degrees back
    return;
 }
 //if we are here, we have a wall to the left and ahead
 turn sensor all the way to the right
 if (US sensor doesn't see a wall) {
    // no wall to the right
    turn sensor back all the way to the left
    NLTurnRight(90);
    return;
  } else {
       //walls on 3 sides - a dead end
       turn sensor back all the way to the left
  NLTurnRight(180);
  return;
  }
}//end of TurnAlongWall

task main{
  while (true) {
    TurnAlongWall();
    GoToIntersection();
  }

}
```

(This is not a complete program: it is missing sensor and natural language setup instructions).

- Ultrasonic Sensor readings are not quite reliable. To fix this, one can take three readings, wiggling the sensor a bit in between; if at least one of the readings is below the cutoff, there is a wall. The simplest way to do it would be to create a function `SeeWall()`, which would return a boolean value.

  A final version of the wall follower program, with all the improvements above, is attached as `WallFollower.c`

  Note that wall follower algorithm has it drawbacks. If you enter the maze from the outside and travel in it always using the wall follower algorithm, you are guaranteed to find your way out. However, if you start in the middle of the maze, you might not be able to

get out using the wall follower algorithm: you could be going in circles around an "island" in the middle of the maze.

Further challenges: Pledge algorithm

(We didn't have time to cover this in SigmaCamp.) 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

```
// pragma statements - sensor setup
// natural language setup
int TotalTurns=0;

void GoToIntersection(){
...
}

void TurnAlongWall(){
...
}

//follows wall until total number of turns is zero
void FollowWall(){
...
}

//goes forward until it meets the wall
void ForwardToWall(){
...
}


task main{
  while (true) {
    ForwardToWall();
    NLTurnRight(90); TotalTurns=TotalTurns+1;
    FollowWall();
  }
}
```

The most difficult part is function `FollowWall()`, which would be essentially the same as wall follower algorithm of the previous program, stopping when `TotalTurns` is zero.

A full version of the program is available as `PledgeAlgorithm.c`

Mapping the maze

One can add to any of the previous algorithms the ability to record one's travel and build the map of the maze. Here is one strategy.

(1) We represent the maze as a grid of vertices (intersections), each having two coordinates: `x` and `y`. We take the original position of the robot to be (0,0), and direction to be the positive direction of `y` axis ("north").

(2) Vertices are connected by passages. We will record information about the maze by recording the status of each passage. Possible values are "BLOCKED" (passage is missing, or blocked by a wall), "UNKNOWN", "TRAVELED", etc.

(3) We will have global variables `position_x`, `position_y`, `direction`, describing robot's position and direction. All functions causing the robot to move/rotate will update these variables accordingly.

It is natural to divide the problem in two stages:

(1) Find a way to store information about passages and write functions for basic manipulations of this data, such as setting or reading the status of a passage

(2) Modify one of the previous programs for maze navigation so that it records and later prints information about the maze, using the functions described in the previous step

These two stages can be done independently of each other. The second, higher level stage, which uses functions like "label the passage from point `(x,y)` in the direction `d` with status `s`" does not need to know the details of where and how these statuses are stored. The only thing it needs is the list of available commands together with their exact syntax and list of arguments. In programming, this is usually referred to as API (application programming interface).

API. So here is the API:

```
//Possible directions: NORTH, SOUTH, EAST, WEST
//Possible statuses: BLOCKED, UNKNOWN, OPEN, TRAVELED

//Functions:

/* *************************
* Direction-related functions
*****************************/

int left_of(int dir){
...
}//end of left_of()

int right_of(int dir){
...
}//end of right_of

int opposite_of(int dir){
...
}//end of opposite_of
```

```
/* ************************
 * Maze map related-functions
 * ************************ */


/* ************************
 * Initializes the map,
 * setting status of each passage to UNKNOWN
 * ************************ */
void initialize_map(){
...
}



/* ************************
 * Sets the status of a passage going from vertex (x,y)
 * in direction dir
 * ************************ */
void set_status(int x, int y, int dir, int status){
}
/* ************************
 * Returns the status of a passage going from vertex (x,y)
 * in direction dir
 * ************************ */
int get_status(int x, int y, int dir){
 ...
}

/* ************************
 * Prints the map to NXT screen
 * ************************ */
void print_map(){
  ...
}
```

A note on data types: there are two approaches. The simple one is to use integers to represent different directions (e.g., 0=NORTH, 1=EAST...) and similarly for statuses. The problem with this is that the program is difficult to read; at all moments we need to remember what integer value represents what direction.

Another approach is to define a new data type and explicitly list what values variables of this type are allowed to take. Both approaches are possible with RobotC. We choose an intermediate approach: statuses and directions will be represented by integers; however, instead of writing say `direction=0;`, we introduce named constants `NORTH`, `SOUTH`, `EAST`, `WEST` and only use them — we will never use numbers such as 0 or 1 to represent directions. Similarly, we will use the following named constants for status of a passage:

`UNKNOWN` – no information (passage not visited yet)

`BLOCKED` – passage does not exist (blocked by a wall)

`OPEN` – passage exists but not travelled yet

`TRAVELED` – passage traveled at least once

Implementation of the API. For first reading, you can skip this subsection and just use provided file `MazeMap.c`, including it in the usual way: `include ‘‘MazeMap.c’’`. This file provides all the functions listed above. If you want to write your own implementation, read on.

To implement the API, we need to find a way to store information about the passages. We will use 2-dimensional arrays: one array to store information about vertical (north/south) passages, another for horizontal (east/west) passages.

Since we can not create arrays of variable size, we need to know in advance what is the maximal size of the maze. Therefore, we introduce a constant `MAZE_RADIUS` and allow coordinates of vertices to range from `-MAZE_RADIUS` to `MAZE_RADIUS`. Since array index must be non-negative, we must do some conversion between `x` and `y` coordinates and array indices. This leads to the following code

```
//directions
#define NORTH 0
#define EAST 1
#define SOUTH 2
#define WEST 3

//statuses
#define UNKNOWN 0
#define BLOCKED -1
#define OPEN 1
#define TRAVELED 2

//maze bounds
#define MAZE_RADIUS 7
#define MAZE_SIZE 15 //=2*MAZE_RADIUS+1

int vertical_passage[MAZE_SIZE][MAZE_SIZE];
/* ********************************
* vertical_passage[u][v] contains status of the passage
* going  up (NORTH)
* from vertex with coordinates x=u-MAZE_RADIUS, y=v-MAZE_RADIUS
* Thus, as u ranges from 0 to MAZE_SIZE-1=2*MAZE_RADIUS,
*   x will range from -MAZE_RADIUS to MAZE_RADIUS
* and similarly for y
* ********************************/
int horizontal_passage[MAZE_SIZE][MAZE_SIZE];
/* ********************************
* horizontal_passage[u][v] contains status of the passage
* going right (EAST)
* from vertex with coordinates x=u-MAZE_RADIUS, y=v-MAZE_RADIUS
* Thus, as u ranges from 0 to MAZE_SIZE-1=2*MAZE_RADIUS,
*   x will range from -MAZE_RADIUS to MAZE_RADIUS
```

```
* and similarly for y
* ********************************/
```

Now the code for reading/setting the status is easy (just remember that the passage leading south from `(x,y)` is the same as passage leading north from `(x, y-1)`).

To print the map, we will use the command `drawLine(x0,y0,x1,y1)`, which draws a line from point `x0,y0` to `x1,y1`. Note that for NXT screen, `x` ranges from 0–99, while `y` ranges from 0–63, so again, we need to convert maze coordinates to NXT display coordinates. Since our array indices range from 0–`2*MAZE_RADIUS`, we should set one maze unit to be equal to `step=63/(2*MAZE_RADIUS)` (note that it is integer division!), which for `MAZE_RADIUS=7` gives `step=4`. Now the conversion from array indices `(u,v)` to screen coordinates `(x,y)` will be `x=u*step`, `y=v*step`.

```
void print_map(){
  int step=63/(2*MAZE_RADIUS);
  int u,v;
  eraseDisplay();
  for (u=0; u<MAZE_SIZE; u++) {
    for (v=0; v<MAZE_SIZE; v++) {
      if (horizontal_passage[u][v]==TRAVELED) {
        drawLine(u*step,v*step,(u+1)*step,v*step)
      }
      if (vertical_passage[u][v]==TRAVELED) {
        drawLine(u*step,v*step, u*step,(v+1)*step)
      }
    }
  }
}
```

Using the API for maze navigation. Thsi part is easy. We take a maze navigation program such as `WallFollower.c` and modify it, introducing global variables

```
int position_x=0, position_y=0, direction=NORTH;
```

and every time we make a turn, we update the variable `direction`; for example, we replace `NLTurnLeft(90)` by `NLTurnLeft(90); direction=left_of(direction);`. We also add to function `GoToIntersection()` code to mark the passage we just used as traveled, and update the variables `position_x`, `position_y`. Finally, after traveling each passage we will print a map to NXT screen using the function `print_map()`.

The final program is attached as file `WallFollowerWithMap.c`.

Challenge for advanced students

Combining (and modifying as necessary) the pieces above, write an implementation of the Trémaux's algorithm.

Maze algorithms

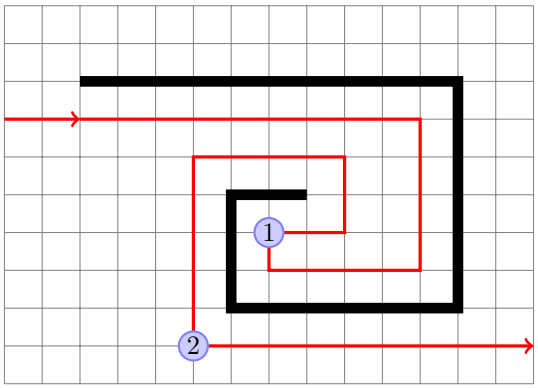
Wall follower. Start following passages, and whenever you reach a junction always follow the leftmost open passage. This is equivalent to a human walking in the a maze by putting their hand on the left wall and keeping it on the wall as they walk through. This method is not guaranteed to find an exit: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

Trémaux's algorithm. This maze solving method is designed to be able to be used by a human inside of the maze. It will find a solution for any maze.

As you walk down a passage, draw a line behind you to mark your path. When you hit a dead end, turn around and go back the way you came. When you encounter a junction you haven't visited before, pick a new passage at random. If you're walking down a new passage and encounter a junction you have visited before, treat it like a dead end and go back the way you came. (That last step is the key which prevents you from going around in circles or missing passages in braid Mazes.) If walking down a passage you have visited before (i.e. marked once) and you encounter a junction, take any new passage if one is available, otherwise take an old passage (i.e. one you've

marked once). When you finally reach the exit, paths marked exactly once will indicate a direct way back to the start. If the maze has no solution, you'll find yourself back at the start with all passages marked twice.