Robot Motion Planning:

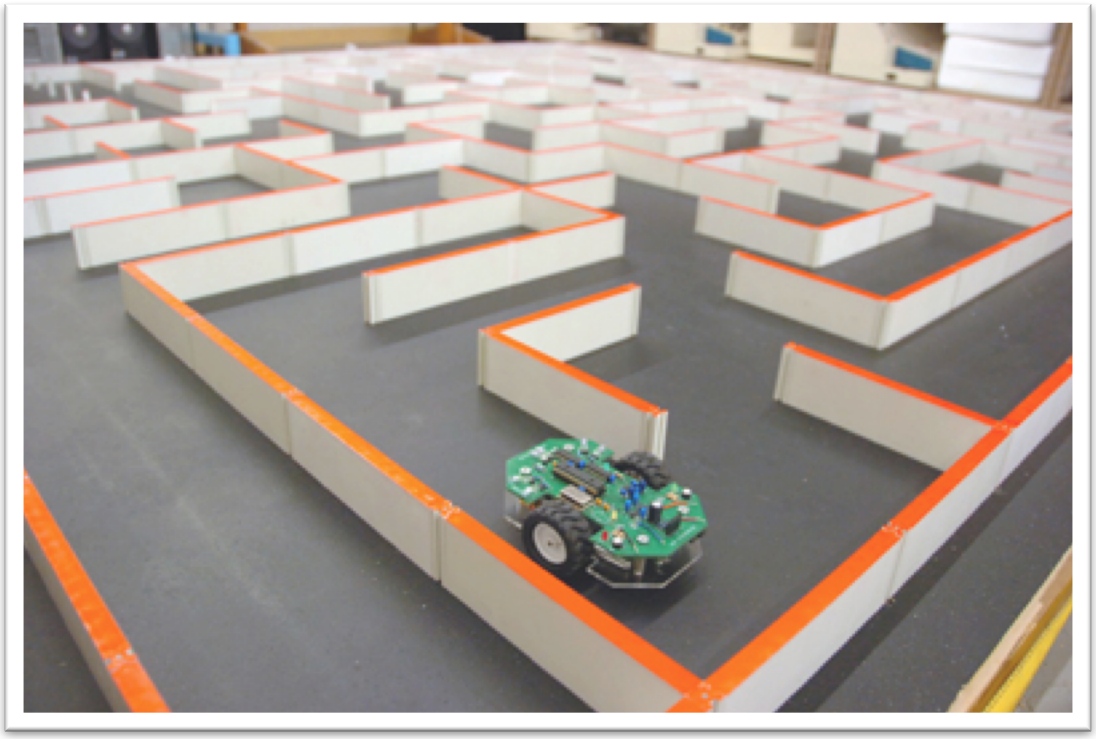
Plot and Navigate a Virtual Maze

Michail Kovanis

Capstone Project

Machine Learning Engineer Nanodegree

24 August 2017



[1] Image taken from: https://www.ocf.berkeley.edu/~jordanwo/images/micromouse.jpg

Definition

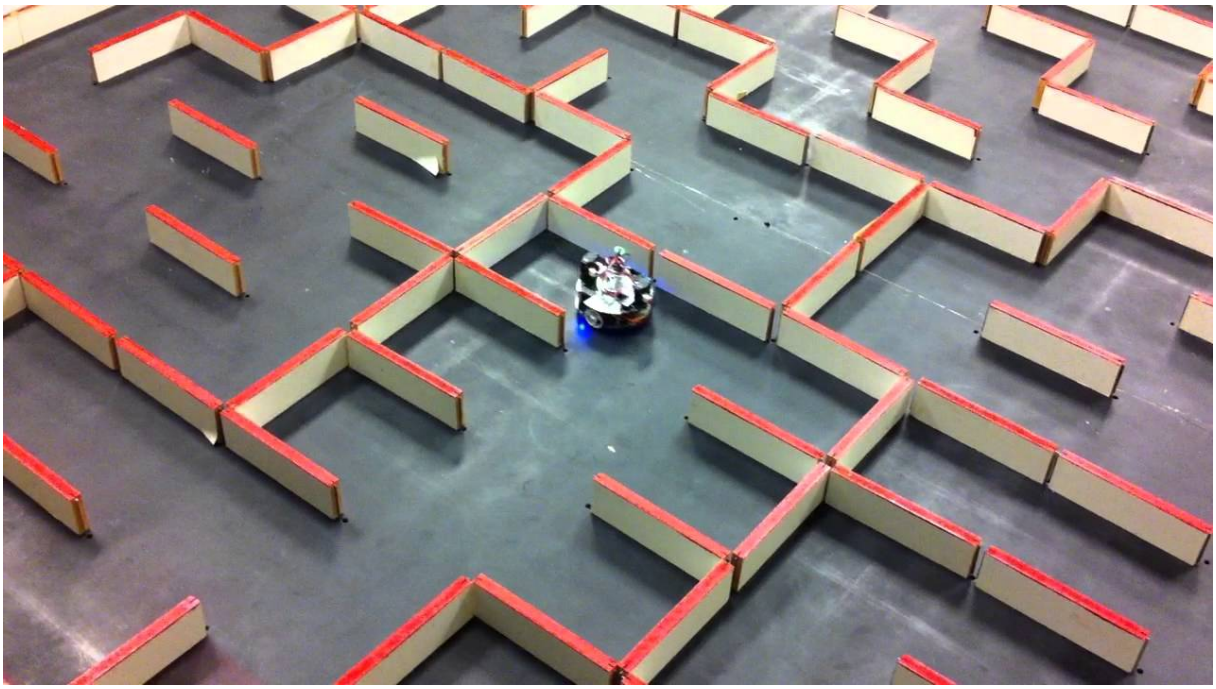
Project overview

A Micromouse competition is an event in which various teams compete on the ability of their micro-robots to solve a maze problem (Figure 1). They were initially introduced by the IEEE Spectrum magazine 40 years ago and are currently being held in various places around the world each year [1, 2].

Each robot is allowed one run to map the maze and one run to run as fast as possible to the destination square of the maze. Both these runs should be unassisted by humans. The maze consists of a 16×16 grid of cells, which are each *18cm* wide and *18cm* long. The robot is evaluated on the time it takes to map and solve the maze and it's penalized for every time manual assistance was required (touch penalty) as seen in Equation 1 [2, 3].

$$Score = run\ time + search\ penalty + touch\ penalty \quad (1)$$

Figure 1: A micro-mouse solving a maze[2]



[2] Image taken from: https://www.youtube.com/watch?v=d0ljAzYHA4I

Problem statement

In this project, I considered an $n \times n$ grid of squares ($12 \leq n \leq 16$), where in its center is situated the goal room (2×2 grid). I programed a virtual micro-robot to navigate through a virtual maze and obtain the fastest running times possible in a set of test mazes. For each maze, the micro-robot performed two runs, one to map the maze and return to the starting square, and one to solve it. The solving algorithm was able to efficiently map and solve a variety of mazes.

Metrics

The model was evaluated by the following metric:

$$score = \frac{run_1}{30} + run_2 \ (2)$$

where run_1 is the number of time steps spent to map the maze and return to the start square, and run_2 is the number of time steps it took the robot to solve the maze. The touch penalty mentioned in Equation 1 is not applicable here. The maximum allowed number of steps is 1000.

Moreover, since an algorithm may not solve the maze in every run I further evaluated it in how many runs, out of 1000, it found a solution in each maze.

Analysis

Data exploration and visualization

The data that I used for this project came from three txt files provided by Udacity, which contain the specifications of the test mazes. The files were already in the appropriate format and thus I didn't need to further process them.

On the first line of each file is the value of the variable n and on the following n lines are n comma-separated values describing the edges of a square that the robot is free to move. Each i^{th} element of the grid represents a four-bit number, in which every side is represented by a

number as follows: 1 for the upper side, 2 for the right, 4 for the bottom, and 8 for the left. If a side is closed, then its number is multiplied by 0, otherwise by 1. Each element of the grid is represented by the sum of all the side numbers each multiplied by 0 or 1 (if closed or open, respectively). Finally, the first row of the $n \times n$ grid of the txt file represents the leftmost column of the maze and the *[0, 0]* element is the bottom-left square of the maze (where the robot starts its run).

The robot is thought to be in the centre of a block and to point to one of the following directions of the maze: up, down, left or right. At the start of each run the robot always points upwards. The robot has three sensors telling it whether a wall exists on its left, up or right. If no wall exists, then the robot may move towards a block. Its movements can be only forward and backward (up to three blocks), therefore to move left or right it should perform a rotation (-90 or 90 degrees respectively).

An example of a maze is seen on Figure 2 and it is Maze 1. This maze is 12×12 and has multiple solutions. Its goal room is situated at the centre of the maze, contains the blocks *[5, 5], [5, 6], [6, 5], [6, 6]* and its entrance is through *[6, 6]*. As seen with the eye, the most obvious (and probably optimal) solution is the one shown in red on Figure 3 and consists of 33 movements. Another path that the robot could follow and that is very close to be optimal (34 movements) is the one shown in blue on Figure 3.

Figure 2: Maze 1

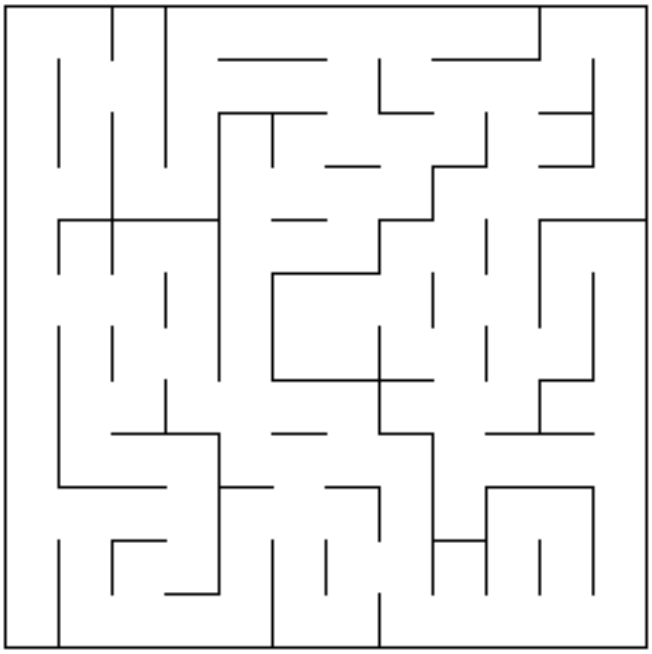
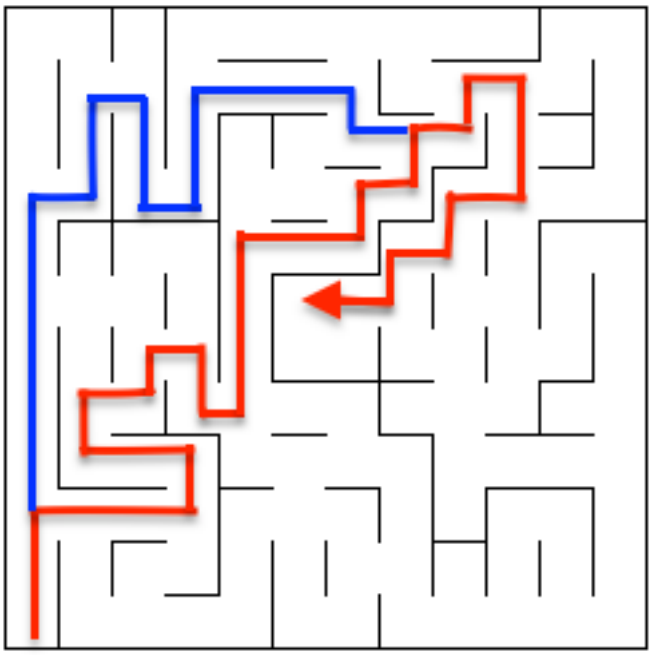


Figure 3: A solution to Maze 1



Algorithms and techniques

I implemented the A* search algorithm to program a robot to map and solve a maze [4, 5]. This algorithm is an informed-search algorithm, which can solve a maze problem by searching among all possible paths to the goal square. The algorithm first considers the paths that seem to be the shortest to the solution, by applying an evaluation function *f(n)* to a square that the robot may move to at a certain time step:

$$f(n) = g(n) + h(n) \quad (3)$$

where *g(n)* is the minimum number of time steps from the start square until square *n* and *h(n)* the physical distance from the center of the square *n* to the center of the maze. The algorithm evaluates *f(n)* only for the squares that can be accessed by the robot. The robot always selects to go to the node *n* with the lowest value of *f(n)*. Ties were resolved randomly. The value of *h(n)* was calculated only when a square was candidate to be visited by the robot. Finally, the robot was keeping track of all the previous steps until a square *n*, thus making it possible to return to the starting square and perform the final run.

Benchmark

As a benchmark model, I used a basic configuration of the Breadth-first search algorithm. This algorithm is like a simpler version of A*, in which one does not evaluate the cost of a movement but rather whether a block has been visited before or not. If not, then the robot moves to that block. In this implementation, when multiple blocks were not visited, the robot always preferred the rightmost block to its sensors. If all blocks were visited, then a block was selected randomly.

Moreover, I considered as a worst-case performance one which requires *744* time steps to map the maze and return to the start square (run_1), *256* time steps to solve it (run_2), and a score equal to *280.8*.

Implementation and refinement

In each time step I took the readings of the sensors. If no block was available to move to, I rotated the robot 90 degrees and re-measured its environment. If only one block was available to move to, then the robot moved towards it. If more than one blocks were available, then the A* algorithm was implemented. The blocks considered were only those, in which the robot had not been in the past and those previously accessed but from a path of higher cost. The algorithm found the block with the least cost and the robot moved towards it. Ties were resolved randomly.

In this implementation of the A* search algorithm I allowed for exploration of the maze only until the robot reached the goal. Then the robot returned to the start, rotated to point upwards and it started the final run, in which it solved the maze. I did not allow for more exploration because this would have been more likely to add an unnecessary extra exploration score of ~20 (if let until > 800 steps as mentioned in the proposal), which would be welcome only in mazes with solutions that vary a lot in size.

Before reaching to this final version I implemented a variation of this algorithm. One, in which the robot would go towards the rightmost free block that has not been accessed before or that had been accessed from a path of higher cost. For example, if the A* algorithm considered the left and the upper block, then in this implementation the robot would always go to the upper one independently of the cost.

Results

Model evaluation and validation

Table 1: Final results after 1,000 runs

		Maze 1	Maze 2	Maze 3	Total score
A*	Score (mean ± std)	43 ± 4	59 ± 5	69 ± 9	171
	% Solved	99.9%	99.8%	94.7%	N/A
A* (variation)	Score (mean ± std)	46 ± 5	66 ± 5	75 ± 5	187
	% Solved	99.6%	97.5%	91.6%	N/A
Breadth-first	Score (mean ± std)	48 ± 7	70 ± 8	92 ± 32	210
	% Solved	89.7%	86.1%	70.8%	N/A
Threshold	Score (Theoretical)	280.8	280.8	280.8	842.4

Justification

I ran 1000 simulations of the Micromouse competition for all three algorithms and mazes. At a first look it seems that the standard A* search algorithm is the best approach. It almost always solved mazes 1 and 2, while it didn't solve only about 5% of the times the third one. The variation of A* performed relatively equal as compared to the percentage it solved maze 1, however it solved about 2.5% less times mazes 2 and 3. Finally, as compared to the Breadth-first algorithm, both A* implementations solved between 10% and 25% more times the mazes.

Regarding the total score, all three algorithms performed obviously better than the worst-case threshold I calculated (more than $3 \times std$). Moreover, as seen on Table 2, the scores achieved by the standard A* search algorithm are all statistically significant ($P < 0.0001$) as compared to all other algorithms in all mazes [6].

Table 2: t-tests of the standard A scores as compared to the other algorithms*

	Maze 1	Maze 2	Maze 3
	A*	A*	A*
A* (variation)	P < 0.0001	P < 0.0001	P < 0.0001
Breadth-first	P < 0.0001	P < 0.0001	P < 0.0001

Conclusion

Free-form visualization

Apart from these three mazes, I created one extra *16 × 16* maze to further test my maze-solving algorithm. The maze and its solutions can be seen on Figures 4 & 5 respectively. The maze has four solutions, one of 14 steps (red), one of 70 steps (dark blue) and two of 55 and 58 steps (lighter shades of blue). All solutions are very straightforward and one should expect that a good algorithm would consistently solve the maze.

Table 3: Performance of algorithms on maze 4

	Score (mean ± std)	% solved
A*	49 ± 18	100%
Breadth-first	40 ± 26	96.8%

As it can be seen on Table 3, A* always solved the maze while Breadth-first search didn't sometimes (even though it performed well). A* is an algorithm that in the beginning of run 0 will choose in random whether to go up or right, while Breadth-first search will always go right. Therefore, the score of A* is on average higher as it is an average of the shortest path and the medium sized paths. Moreover, in this kind of maze A* would be guaranteed to find a solution, which it did. On the other hand, even though Breadth-first search would always go towards the shortest

path, its tendency to pick the rightmost block among equals makes it reach a point where it finds a dead-end in the maze. Then, it starts taking decisions in random until it reaches another non-visited block. This procedure makes it fail a few times to find a solution, even though the maze is easy, while it also explains the higher variability in the final score.

Figure 4: Maze 4

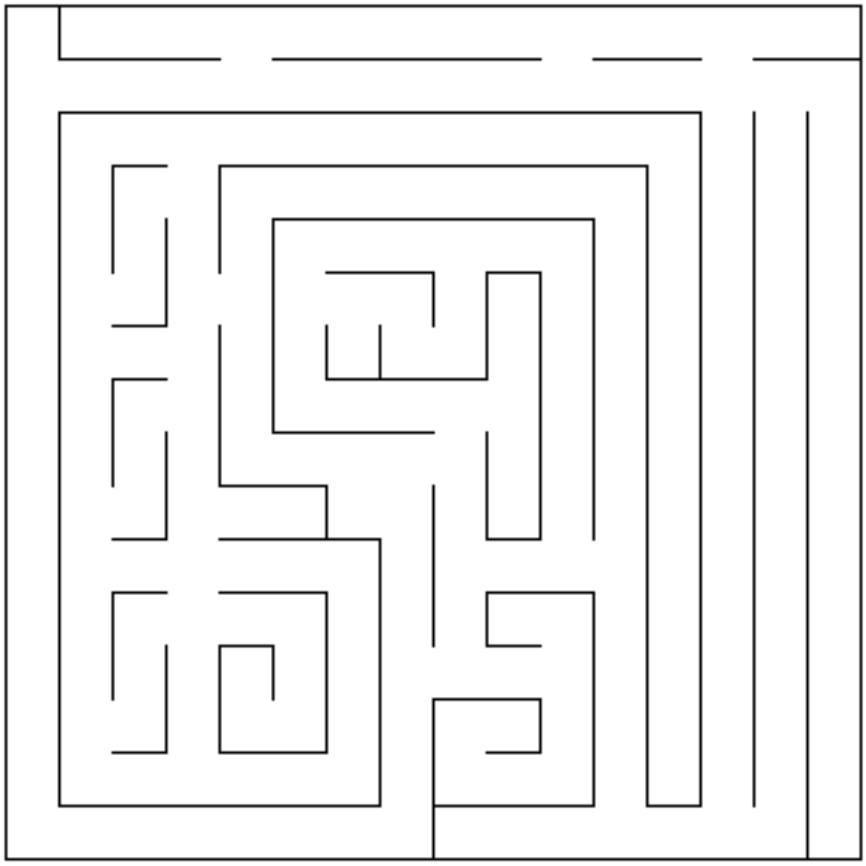
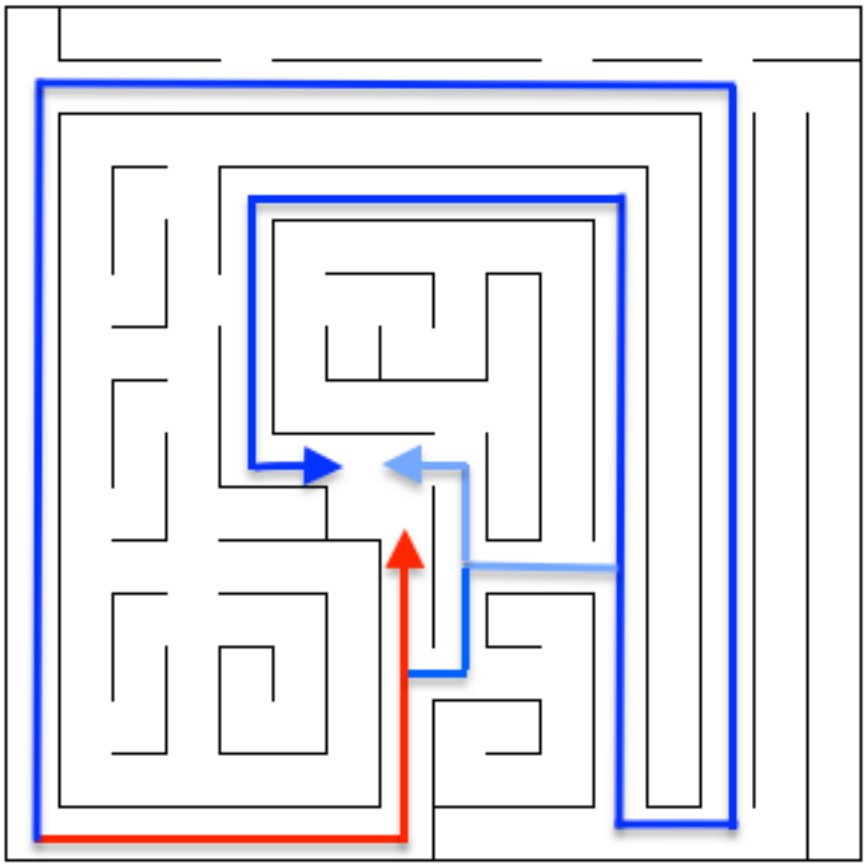


Figure 5: Solutions of maze 4



Reflection

In this project, I programmed a micro-robot to navigate and solve virtual mazes. I used two main algorithms and one variation of one of them. The algorithms were the A* search (plus a variation) and the Breadth-first search, which acted as a benchmark. The A* search is an algorithm that among equally non-visited blocks will choose to go towards the one that has the shortest distance to the goal. On the other hand, Breadth-first search only looks for unvisited blocks, without considering the distance to the goal. The main version of the A* search algorithm performed best among all other algorithms implemented. It solved the most times the three mazes provided plus the one extra I designed. Its score was the lowest on average for the three first mazes and a bit higher for the fourth.

One very interesting aspect of this project is that it is very difficult to make an algorithm solve the maze every time. Even if the maze is relatively easy, there is no guarantee that the algorithm will not fail at least one time after many runs. Moreover, initially I expected both

algorithms would have no difficulty solving maze 4 all the times. I was surprised to see that there were a few times that Breadth-first got completely lost in the maze and failed. Even if the maze was designed to direct towards a solution it wasn't enough for this algorithm.

Improvement

In real world implementations of this project there are additional things that one might consider. For example, real micro-robots and blocks have dimensions that need to be considered. Therefore, it is critical that one considers ways to accelerate and stop the robot without it hitting a wall. Also, the speed of the robot during exploration should not be very high, or else it might not be able to "sense" open walls that it might be passing next to. On the contrary, during the final run the speed of the robot needs to be as high as possible. Finally, another way to see this problem could be apart from having open and closed blocks, to have also ones that can be traversed but with a time penalty. Thus, the robot would have to assess both the normal paths and the shortcuts with the time delay to find the optimal path to the goal.

References

1. Sone Y. Japanese robot culture: Performance imagination and modernity: Palgrave macmillan; 2017.
2. Cai J, Wu J, Huo M, Huang J, editors. A micromouse maze sovling simulator. 2010 2nd International Conference on Future Computer and Communication; 2010 21-24 May 2010.
3. Harrison P. Micromouse Online 2017. Available from: http://www.micromouseonline.com/micromouse-book/rules/.
4. Wikipedia. A* search algorithm. Available from: https://en.wikipedia.org/wiki/A*_search_algorithm - Description.
5. Games RB. Introduction to A* 2014 [cited 2017 21 Aug 2017]. Available from: http://www.redblobgames.com/pathfinding/a-star/introduction.html.
6. MEDCALC. Comparison of means calculator 2017. Available from: https://www.medcalc.org/calc/comparison_of_means.php.