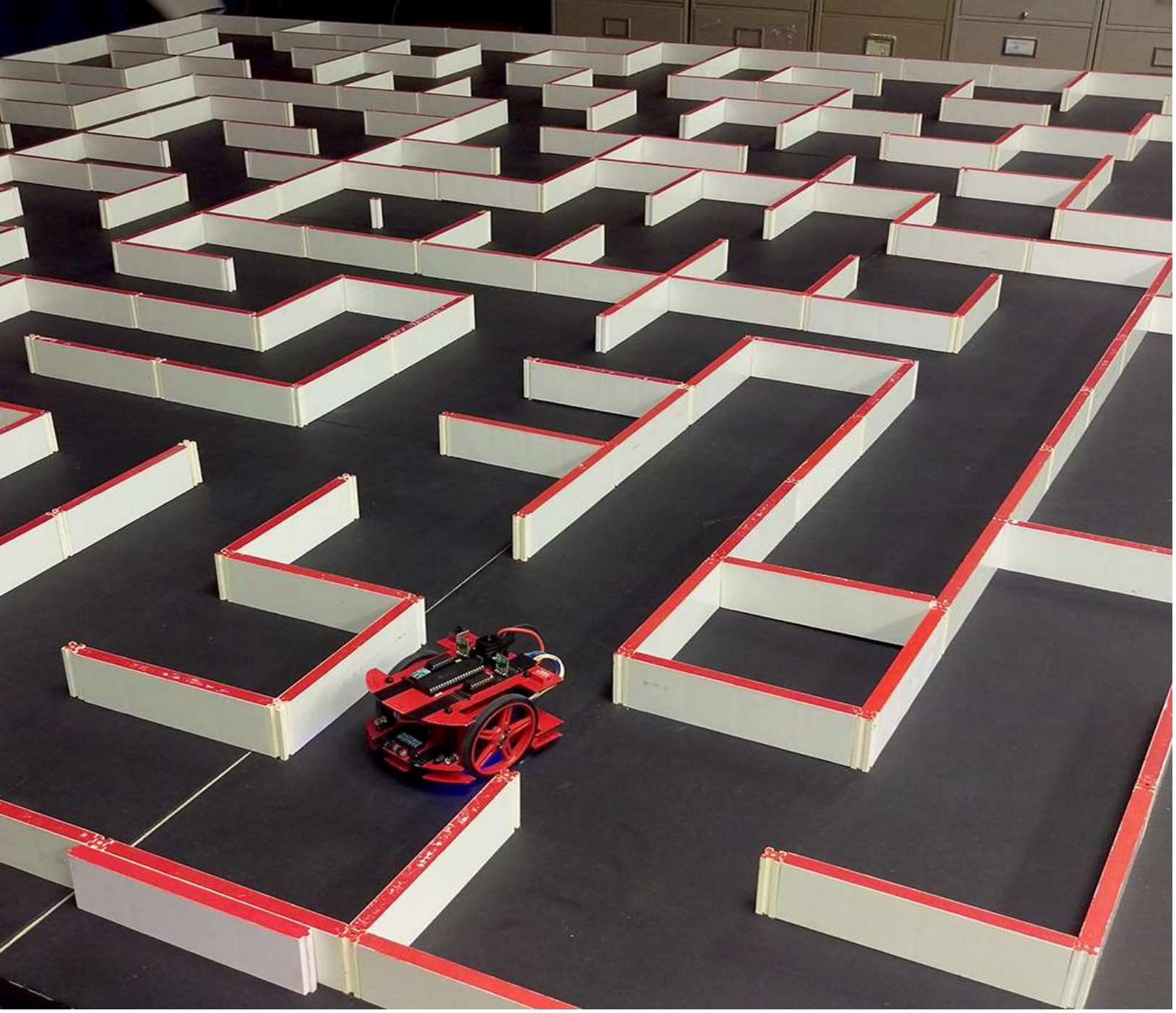


Capstone project

Plot and Navigate a Virtual Maze

Project report

Report Structure

1 How the project was approached

1.1 Algorithms

One of the first point addressed in project analysis, was the search for algorithms to be used for maze navigation. End choice was on 'flooding' algorithm, described in several places over the internet (specifically I found useful a Medium blog and some more technical papers, both quoted in the acknowledgement section). An adaptation of the algorithm to a maze discovery situation was also described in the referenced Medium blog. Flooding algorithm was first implemented in a basic version and in a second stage adjusted to better fit to project specifications.

1.2 Robot class specification

Robot class specification was the second step of the project. Attributes and methods were defined considering two major tasks:

- flooding algorithm implementation (how do I *move* in the maze)
- maze discovery status (what I *know* about the maze)

1.3 Discovery run

One of most intriguing part of the project was how to define navigation algorithm during run 0, where the robot should not only reach the maze center (the easy task) but also discover most of the information that will be needed to be effective during run 1 (the tricky one). This part of the project was addressed as follows:

- define a number of possible maze discovery strategies (called 'exploration modes')
- design code needed to map these strategies and to instruct robot to follow them
- design a "maze generator" code, to define several randomly generated mazes (all aligned with basic maze specification)
- run the robot on all the mazes with all exploration modes and collect info about performances

- collect all results into a Pandas dataframe and analyze the outcome in a dedicated Jupiter notebook
- finally, instruct the robot to follow most promising exploration mode

1.4 Complete project requirement

This include identification and analysis of challenging mazes and collection of recommendations for possible further development

1.5 Collect outcome

Outcome of the project consists of:

- robot.py code, including robot class definition and flooding algorithm implementation
- createmaze.py code, including software to create random mazes, align them to needed specification and save them in the predefined text format (the same used by maze.py and showmaze.py functions)
- a Jupiter notebook ('TrialMazeAnalysis.ipynb') showing the analysis done on the trial maze test set
- this project report.

1.6 Report organization

The rest of the report is organized as follows

- Section 2: Presentation of flooding algorithm, its application to our contest and of the adaptation introduced to better fit requirements.
- Section 3: Robot specification. Include explanations and comments on the code included in the robot.py package
- Section 4,5: Data exploration. Section 4 discuss the creation of a trial set of mazes, while chapter 5 discuss robot parameter tuning over the trial set,
- Section 6: Additional mazes exploration. The section identify those mazes that resulted more challenging in chapter 5 and discuss the problems experienced by the algorithm on one of them

- Section 7: Improvement. Known limitation of the code and possible improvement are discussed in this section
- Section 8: Acknowledgement

2 The 'flooding' algorithm

2.1 Algorithm description

As suggested by the name, the approach taken by flooding algorithm is to mimic the flow of a liquid from the target of the maze (normally the central box in our case) to all adjacent and communicating cells. More in practice, if the maze is known, the algorithm can be descripted by the following pseudo code:

- *start with all cells in the maze as 'unmarked'*
- *assign weight '0' to all cells in the maze target*
- *assign weight '1' (i.e. 0+1) to all **unmarked** cells directly communicating with a cell with weight 0*
- *repeat last step starting from all cells which received a new weight in the last iteration and adding '1' to communicating adjacent cells*
- *stop iteration when all cells have been marked with a weight (assuming that the whole maze is reachable from the target)*
- *navigation from starting point to target can now start, moving at each step toward a reachable cell with the lowest possible weight*

The algorithm is also illustrated by following pictures, based on maze 'test_maze_01.txt' provided as sample input:

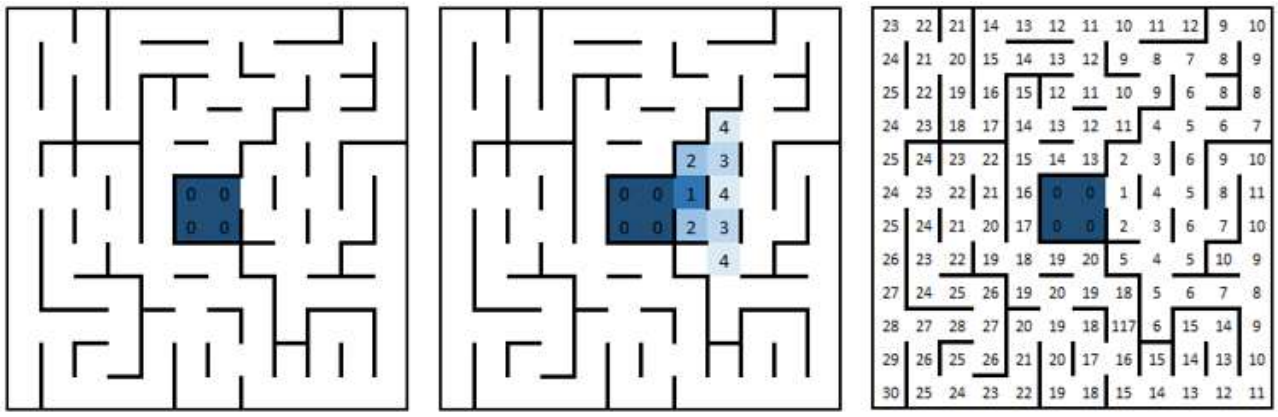


Figure 1: Flooding algorithm: step 0 (left), step 4 (center) and final outcome (right)

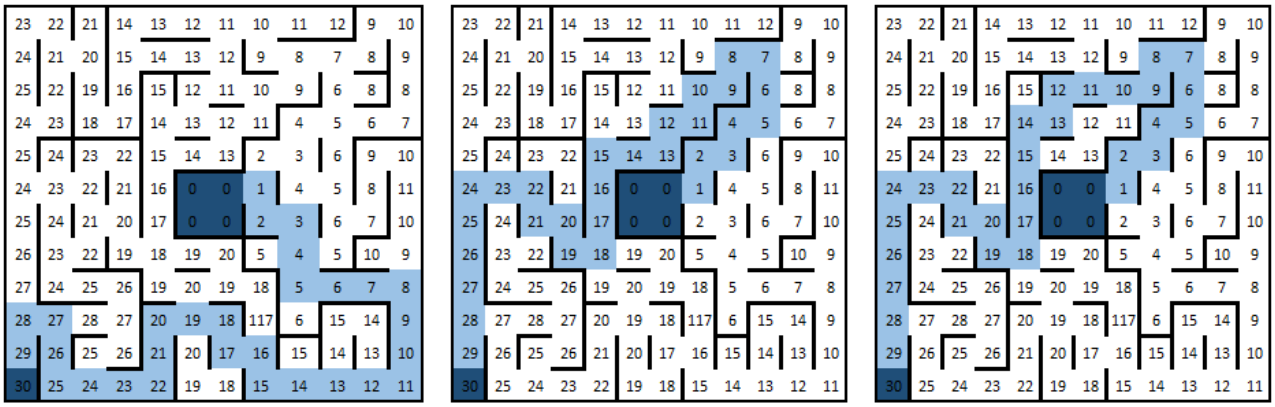


Figure 2 Flooding algorithm: calculated shortest paths from start to target (3 equivalent options)

2.2 Flooding algorithm for unknown maze

From description above, it should be clear that the flooding algorithm is capable to calculate the shortest path once the maze is known. However, the robot start with very limited knowledge about the maze, so how does the algorithm apply to our case? Common approach for this case is to proceed as follows:

- *Start assuming that there are no walls in the maze apart from those on maze perimeter and those known in starting cell*
- *Learn from sensors about walls on left, front and right direction*
- *(Re)calculate flooding algorithm based on current knowledge about maze walls*
- *Move robot considering current flooding weight*
- *If the target is not reached (flood weight of reached cell is >o), repeat from 'learn from sensor' step*

In other words, the robot always assumes that the only walls in the maze are those directly experienced from its sensors during its history of moves; at every move however, its knowledge about the maze gets better, and so are the decisions that flood-based algorithm is recommending. In addition, to avoid possible loops, in all cases where there is more than one equal-weight choice, the path is chosen using a random approach. Next pictures show first steps of flooding calculation on the maze shown before.

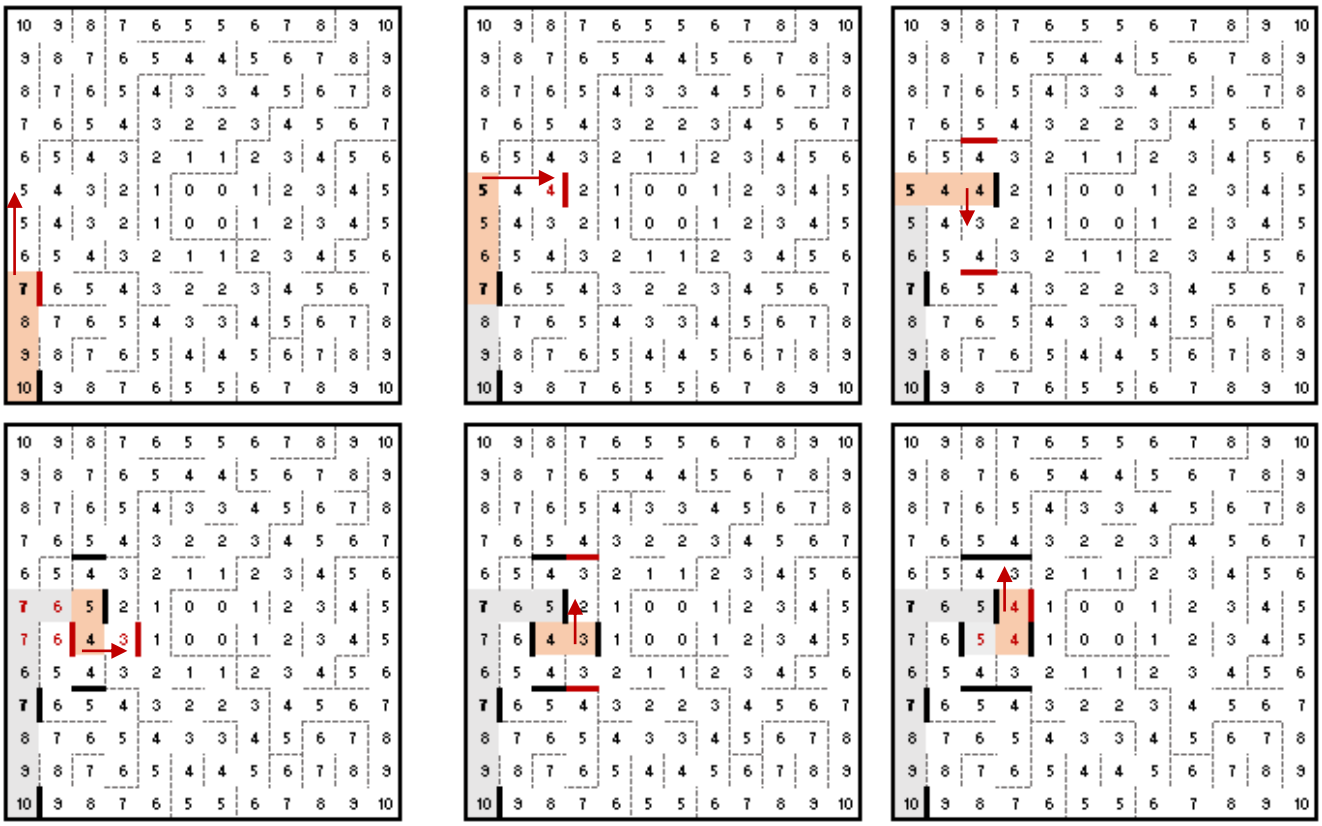


Figure 3 Flooding algorithm while maze learning: steps 1-6.
Orange boxes indicate last move, red arrow next move, red lines newly discovered walls, black lines previously discovered walls, dotted lines unknown walls. Numbers in each box indicated the marking received from from the latest flooding calculation

2.3 Adaptations

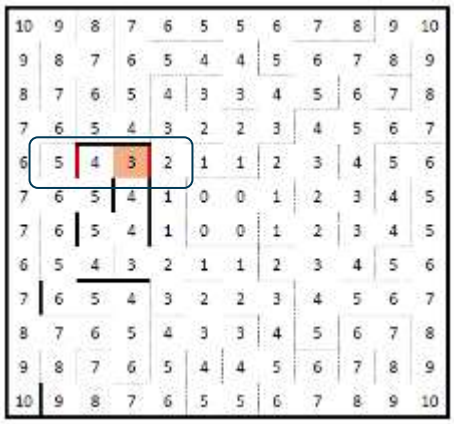
Paragraph above describe somehow the basic version of the algorithm, as found in literature; in our code, this version was slightly modified to consider two targets:

- Increase code efficiency limiting flooding recalculation after walls discovery at each step
- Include the possibility for the robot to run up to three cells per move

2.3.1 Efficiency in flooding recalculation

As described above, when maze is unknown, moving in the maze requires a flooding recalculation every time sensors input discover new walls before a move (meaning a recalculation at nearly every move). However, it can be noticed that not all the weights need to be recalculated, as those with values lower than a limit remain valid even after wall discovery. Again, we explain the adaptation with an example. Imagine that, after a

given step, the robot-internal representation of the maze is the one described in the picture below:



The robot is currently in the light orange cell, direction north, and has just sensed two new walls on left and right, outlined in red; (in the picture, dotted lines represent walls in the maze not yet known to the robot, while heavy black lines represent already discovered walls). Let focus on the impact of the newly discovered walls. We have 4 boxes directly impacted by the walls, currently with weight 5,4,3 and 2 (from left to right). We can clearly see that weight 3 is not valid anymore, as it was calculated assuming communication with cell with weight 2, communication that we now know to be forbidden. As we know that weight 3 is wrong, we also know that weight 4 and 5 in the adjacent cells might have to be adjusted. However, *there is no way that the new walls can require recalculation of cell with weight 2*, as clearly that cell must have received the flow from a cell that was not impacted by new walls. In other words, at every new sensor input, we can:

- Look at the set of cells adjacent to new walls (the four boxes considered in our sample)
- Consider the minimum weight present in that set of cells (2 in our sample), let's say it is '*m*'
- Mark all boxes with a weight greater than '*m*' as unmarked
- Find all cells in the maze with a weight exactly equal to m
- Start flood iteration from the set of cells with weight m, calculating those with weight m+1

Coming back to our sample, we then need to cancel from the maze representation all weights greater than 2, find cells with weight 2 and restart flooding calculation from that cells. This is shown in pictures below:

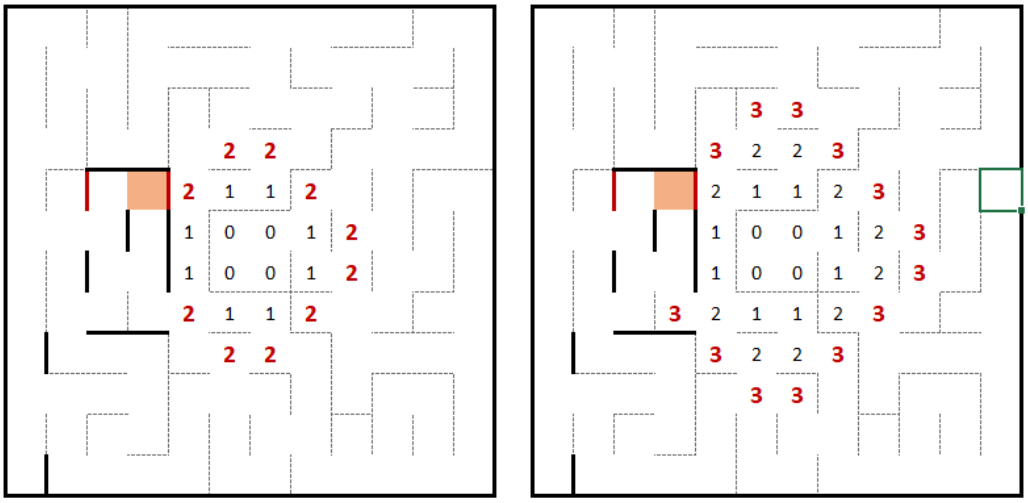


Figure 4 Optimized flooding recalculation

2.3.2 Three-cells moves

The version of the algorithm described above actually works regardless of the number of boxes traversed per move, as long as moving happens in only one direction and sensors detection is unlimited in number of boxes. However, one possible defeat of the basic version is that weight for each box does not count moves needed to reach the target, but boxes to be traversed. In addition, we could wonder if there was some performance gain in working on moves instead than boxes.

The adaptation included to work on moves was to adjust flooding weight iteration in a way that all boxes reachable by the robot in a single move are considered adjacent. Pictures below shows the difference in a generic algorithm iteration:

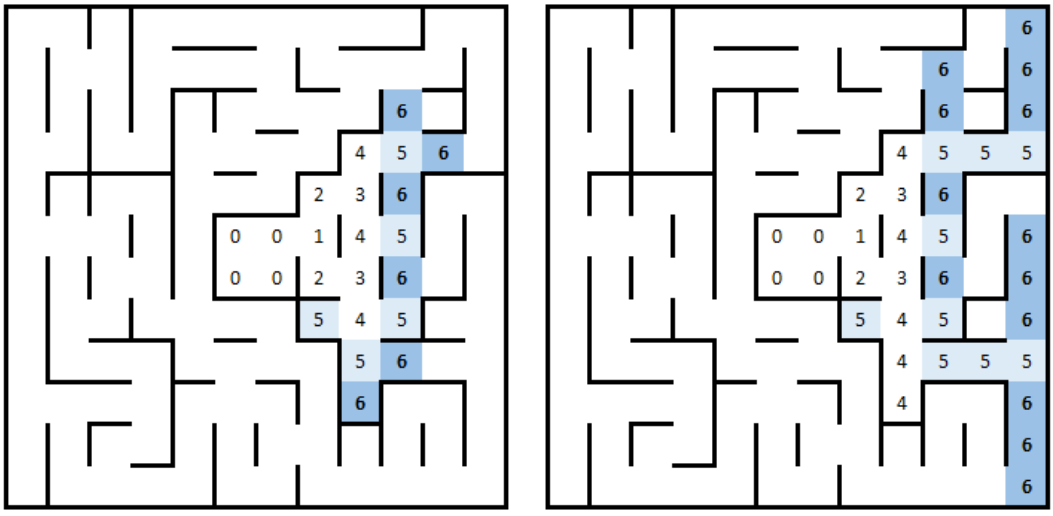


Figure 5 Step 6 of flooding calculation with basic algorithm (left) and with 3-boxes move adaptation (right)

It is interesting to compare the final flooding matrix with the two options. This comparison is shown below:

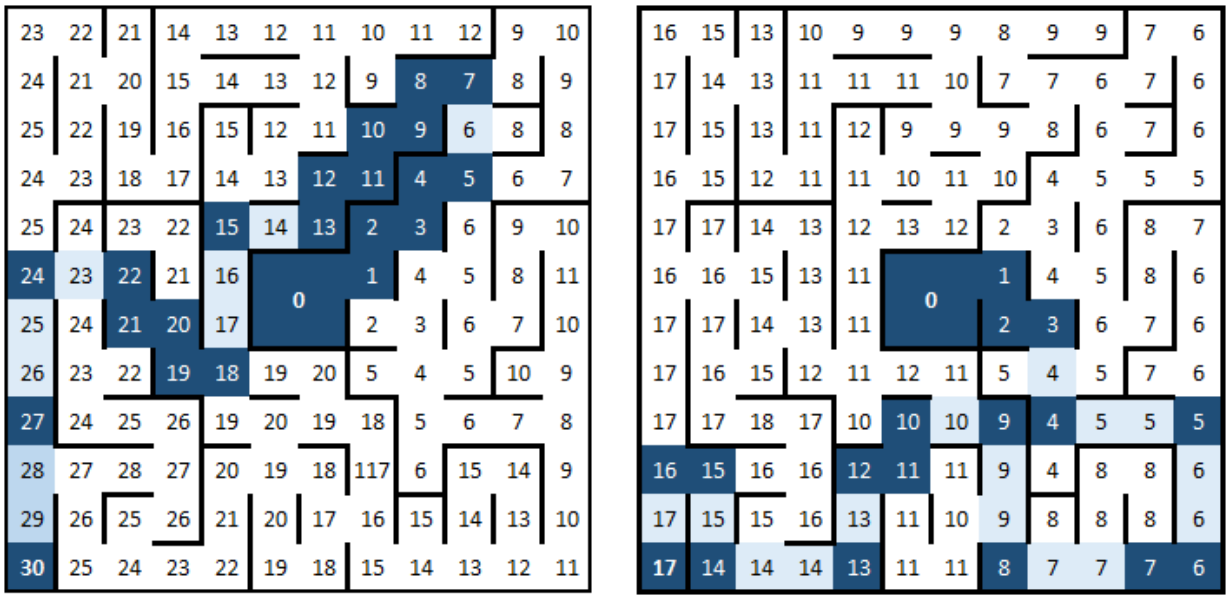


Figure 6 Flooding matrices (for known mazes) with one box move (left) and 3 boxes moves (right)

On both pictures, shown values correspond to the outcome of the flooding calculation (using pure adjacency on the left and 3-box move rule on the right). Colored boxes show the path taken by our robot (that in both cases is capable of 3 box moves) when given the two flooding matrices (dark blue are actual stops of the robot, while light blue correspond to traversed cells). The value in the left bottom box shows number of boxes on the shortest path from start to center (left picture) and the minimum number of moves from start center (right). Some numbers (taken from picture) are useful here:

	Basic flooding	3 box flooding
Number of moves	21	17
Traversed boxes	30	32

In other words, adapted flooding is more effective to reach the target with the lowest number of moves, while the basic version of the algorithm provides the actual shortest path but is less effective in number of moves. For our purpose, it appears that the modified flooding is more performant, so this version was included in the code.

2.4 Sample run on test_maze_01.txt

As a concluding example for this section, let's explore a complete run of the algorithm, initially simply designed as follows:

1) Run 0: reach maze center and ask for reset
2) Run 1: reach center

Please consider that as random selection is used in case of equal distance options, a new run of the algorithm can result in a different outcome. Please also note that different exploration modes will be introduced in next chapters, resulting in improved robot performance.

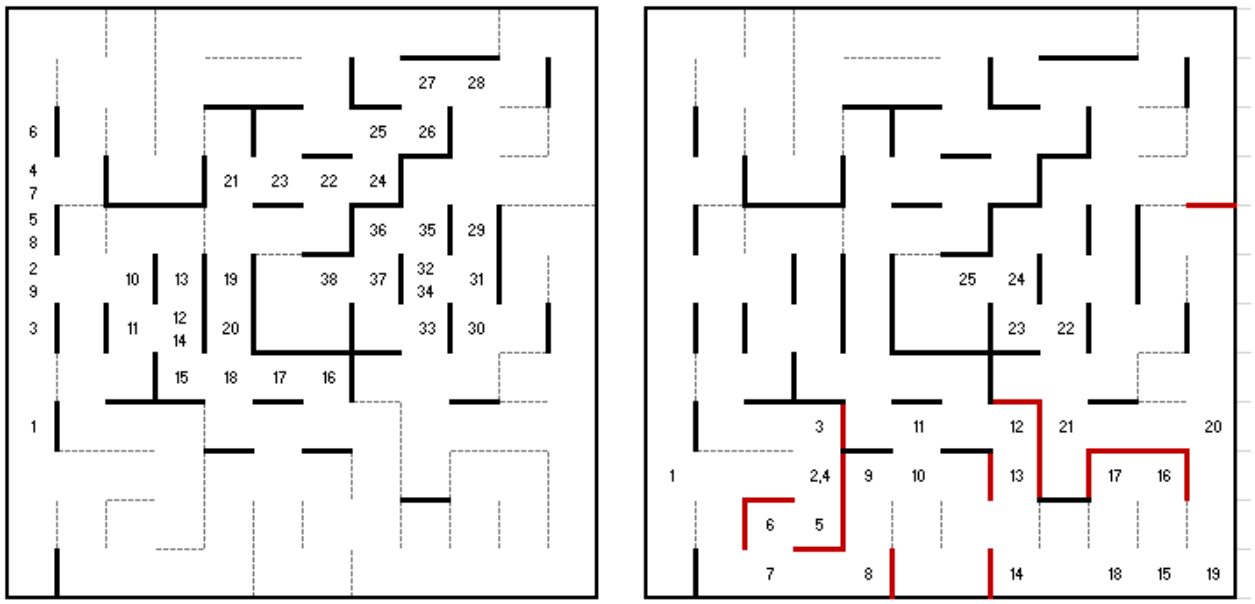


Figure 7 Run 0 (left) and Run 1 (right) outcome. Numbers shows robot position after each step. Heavy lines show walls learned by robot during run 0 (in black) and run 1 (in red)

As shown in the picture, during run 0 the robot chose to go north and then to try to move right, driven by flooding algorithm and with poor knowledge of maze walls. At the end of run 0, the robot spent 38 steps (in this example) and acquired a pretty good knowledge about the center of the maze, where most walls got known. On the contrary it acquired almost no knowledge about the bottom part of the maze, where all vertical walls are still to be discovered.
Due to the unbalance of maze knowledge, on run1 the robot tries to go to the bottom part of the maze (remember that unknown walls are assumed to be open). This resulted in several new walls discovered during run 1 (red walls in the right picture) and consequently in longer final path. Outcome for run 1 was 25 steps, that is around 50% more of the minimum needed number (calculated before as 17). Final score was 26.3.
Given the scoring mechanism, we could expect that if the robot would have spent more time during run 0 to explore the unknown part of the maze, this could have resulted in better efficiency on run 1, resulting in turn into a better score.
This topic will be better analyzed in the next section.

3 Robot specification

3.1 Class attributes

The information maintained by the robot (i.e. Robot Class attributes) must allow the robot (as a minimum):

- to track its position in the maze and its current direction
- to maintain a picture of the maze (based on starting info and on history of inputs received from sensors)
- to have the latest flooding algorithm output (aligned with current maze image)
- to know the target it is currently requested to reach
- to be somehow aware of the ‘mode’ he is moving in the maze (i.e. run0 versus run 1, plus eventually additional info during run 0)

In addition, to maintain a picture about how much the robot knows about the maze, two additional attributes were included, collecting info on the knowledge about status (walled or not walled) of a specific passage. These latest attributes have no actual effect on robot behaviour and have only been introduced to gain understanding of

performance of robot exploration algorithms. Class attributes are summarized in the table below.

Attribute	Type	Description
Maze_dim	Integer	Size of the maze (12, 14 or 16). Referred as 'N' in this table
Location	List of 2 integers	Column and row for current position. Bottom left = [0,0], bottom right = [N,0] , top left= [0,N]
Heading	Str	Current direction ('u'=up, 'd'=down, 'l'=left, 'r'=right)
Walls	N x N numpy array of integers	Store the robot-internal representation of the maze, based on history of info collected from sensors. 4-bit based as per project spec.
Target	list of 2-sized tuple of integers	Each tuple represents one of the 4 central box in the maze.
Flood	N x N numpy array of integers	Flooding distance from each box toward the current target, based on current mase knowledge (walls)
Mode	Str	Navigation mode, can take 3 values: • 'explore_to_target': run 0, robot is making first exploration toward centre boxes • 'explore_from_target': run 0, robot has reached the centre and is further exploring the maze before moving to run 1. Exploration in this phase is driven by exp_path attribute • 'go': robot is in run 1 and is trying to reach the centre on the shortest path
Exp_path	Integer	Key to a global dictionary of possible exploration path
Exp_target	list of 2-sized tuple of integers	Each tuple represents a box that is part of the current exploration target.
Exp_step	Integer	Current index for a list of exploration steps
Restart	Boolean	When True, indicates that flooding algorithm must be reset (because navigation target changed)
Known_hor	N x N np array of Boolean as integers	A value of 1 indicates that the status of a specific horizontal passage is known. It means that the information stored in 'walls' about that specific wall was confirmed by sensor.
Known_vert	N x N np array of Boolean as integers	As per 'known_hor', but for vertical passages.

3.2 Class Methods

Class methods can basically be divided into two sets, those related to attribute initialization and those associated to calculation and management of robot moves. They are also summarized in the table below.

Methods	Description
init_walls	Initialize walls variable (only perimetral walls an those on bottom left box are set)

init_known_walls	Initialize known_hor and known_vert variables
init_exp_target	used to initialize the robot information that will drive maze exploration in run 0, specifically while in 'exploration_from_target' mode
next_move	Trigger calculation and manage communication of next robot move
__move_robot	Perform actual robot move, updating location and heading attributes
__calculate_next_move	Called by next move for actual move calculation, including flooding algorithm logic
__explore	Calculate best moving option from present location in a specific direction, with limit in max move length
__update_walls	Updates walls attribute given current sensor input

3.3 Exploration algorithm

As discussed before, showing the first run example, in order to have good performance on run 1, it is important that during run 0 the robot reaches a 'good enough' knowledge of the maze, so that run 1 may approach the shortest possible path.
As the scoring methods outweigh steps in run 1 over those in run 0 by a factor of 30, it is convenient for the final score to remain in run 0 for some time after reaching the maze centre, if this can result in a reduced number of run 1 steps. But which is then the most appropriate moving logic for run 0?
The approach that was taken in the project was the following:

1) Introduce into the code some level of flexibility, so that the robot may run different 'exploration paths' with minimal or no coding
2) Define a set of alternative 'exploration paths' to be compared
3) Develop some python functions for random maze creation, capable to generate mazes in line with project definition
4) Using functions above, create a 'trial set' of mazes, with different sizes and wall density
5) Run the robot code on all exploration paths on all generated mazes; as there is some random decision in the algorithm, we run the code 10 times for each path-maze pair
6) Collect all the outcome in a pandas dataframe and analyse results in a dedicated Jupiter notebook
7) Finally, introduce in the code the exploration mode that is expected to better suit the kind of maze under analysis

3.3.1 Exploration path coding

The basis for exploration path coding is given by the definition of a global variable named ‘exploring_path’. This variable come in place after the robot reached the centre for first time (this is assumed to be always first step in run 0). After reaching the centre, the robot inspect content of the ‘exploring_path’ variable and gets instruction of next targets to be reached before moving to run 1.
The variable is a dictionary, where the key is an integer representing the exploring path to be used, while the value is a list of places in the maze that need to be reached in sequence to execute that specific mode. Let’s start with an example, to clarify the concept. Imagine that global exploring_path variable is defined as follows:

```
exploring_paths = {0: [],
                   1: [ ['br','tl','tr'] ]
                   7: [ ['tl'], ['tr'], ['br'], ['bl'] ] }
```

Once the robot reaches the centre for first time, the robot uses its exp_path attribute as a key for the proper exploring_path dictionary entry. Let’s explore the 3 cases shown in the example above:

- Exp_path attribute == 0,
  the dictionary will return an empty list. The robot will interpret this information as an indication to complete the exploring phase and to move to run 1. In this case, the robot will not perform any additional navigation after having reached the centre for first time.
- Exp_path attribute == 1,
  the dictionary will return a list with a single entry ([['br','tl','tr']]), that is in turn a list (['br','tl','tr']). The robot in this case is requested to proceed exploration - after having reached the centre- touching any one of the internal list destinations. The string in the list are interpreted as follow:
  ‘br’ = bottom right maze corner (maze_dim,0)
  ‘bl’ = bottom left corner (0,0)
  ‘tr’ = top right corner (maze_dim, maze_dim)
  ‘tl’ = top left corner (0, maze_dim)

'cc' = any of the four cells in the maze center

This implies that the list ['br','tl','tr'] means "reach any corner different from bottom-left one"

- Exp_path attribute == 7,
  the dictionary will return a list with 4 entries [['tl'], ['tr'], ['br'], ['bl']] where each entry is single entry least. Request to the robot in this case is to touch in sequence each of the boxes included in the inner list, meaning that from centre it will move to top left corner, then to top right, bottom right and bottom left. Only after having exhausted the outer list (i.e. after having touched the bottom left corner) the robot will ask to reset its position and to start run 1

The approach above was pretty efficient in analysing different exploration strategies, as adding a new strategy simply requires the creation of a new entry in a global dictionary. The full list of strategies tested is the project is copied and commented below:

```
exploring_paths = {0:[],
                   1:[['br','tl','tr']],
                   2:[['bl']],
                   3:[['br','tl','tr'],['bl']],
                   4:[['tl'],['cc'],['tr'],['cc'],['br'],['cc']],
                   5:[['bl'],['cc'],['bl'],['cc']],
                   6:[['bl'],['cc'],['bl'],['cc'],['bl'],['cc']],
                   7: [['tl'], ['tr'], ['br'], ['bl']]}
```

0: no further exploration after reaching the center

1: centre > any corner different from bottom left > Reset

2: centre > bottom left > Reset

3: centre > any corner different from bottom left > bottom left >Reset

4: centre > top left > centre > top right > centre > bottom right > centre > Reset

5: centre > bottom left > centre > bottom left > centre > Reset

6: centre > bottom left > centre > bottom left > centre > bottom left > centre > Reset

7: centre > top left > top right > bottom right > bottom left > Reset

4 Random Maze trial set

4.1 Generation of random maze trial set

As discussed in the project approach summary, in order to tune robot parameter, a random maze generator code was developed. The code randomly defines square mazes with parametric size n, with a few additional constraints:

- Maze perimeter is fully walled
- The bottom left box has a wall on the right
- The 4 boxes in the centre are fully walled except for a single open wall
- There is at least one path to maze centre from each box in each maze

The generator code, originally developed for the project and provided in module 'createmaze.py', is pretty simple and reuses flooding routines to evaluate connectivity. Algorithm can be roughly described by the following pseudo code, with parameters N and M:

1. Create an N x N maze with no walls apart from those on perimeter and the right wall on left bottom box
2. Randomly add 7 walls surrounding the 2x2 centre of box (so that there is only one exit from the centre)
3. Select randomly a box and a potential wall on that box
4. Verify (using flooding routine) if the addition of the new wall cause unreachability of some part of the maze from maze centre. If not, add the wall to the maze
5. Repeat from step 3, stop when no wall is added after M consecutive attempt
6. Save the identified maze as a text file, following coding assumptions used in the sample mazes

The parameter M, measuring the number of failed attempts of creating a new wall after which the maze is considered completed, is empirically related to the wall density in the maze; the trial set of mazes consists of 20 different mazes per pair (N,M) with N=12,14 or 16 and M=5, 20, 60, 120. The total number of mazes was expected to be 20 x 3 x 4 =240; actually, the count was limited to 237 due to a minimal issue on file naming during maze creation.

4.2 Benchmarks for comparison

Collected data allows several possible comparison metrics related to robot performance, some of them used in the rest of the document for specific analysis. Below some proposals:

4.2.1. Performance score

Most obvious benchmark for robot algorithm comparison is the scoring value suggested as part of the exercise

Score = run 0 steps / 30 + run 1 steps

4.2.2. Maze knowledge after run 0

Purpose of run 0 is basically to explore the maze, in order the get the best knowledge of its structure and allow best performance on run 1. The knowledge of maze is maintained in robot class using two dedicated attributes, known_vert and known_hor. These are matrices of Boolean values indicating if the robot has gained at a given stage a confirmed knowledge of presence/absence of a wall in a given position. Note that the variables do not provide any info about actual presence of a wall (this is stored elsewhere) they only give information about knowledge. As an example, if a robot sensor determines that the distance from next wall in a given direction is X boxes, the robot acquires knowledge about status of X walls (X no wall + 1 wall).
In the picture below, the robot is represented by the blue triangle and is directed East. The front sensor of the robot provides a distance of 6 boxes from nearest wall, indicating absence of right wall on 6 boxes and presence of right wall on the 7th.

Figure 8 Knowledge acquired by a robot sensor

The number of possible walls (including perimeters) in an N x N maze is:
- N x (N+1) vertical positions
- N x (N+1) horizontal positions

Overall, the robot knowledge about the score (in percentage) can therefore be defined as:

Maze_Knowledge = [Sum(known_vert) + Sum(known_hor)] / [2 x N x (N+1)]

Given the approach used for navigation, a perfect knowledge about the maze (Maze_Knowledge = 1) will grant the shortest path to be taken on run 1.

4.2.3 Run 1 steps versus shortest path length

When the structure of the maze is known (e.g. during robot algorithm validation), it is normally possible to calculate the shortest possible path (in number of moves) from origin to target. The two ratios:

- Score effectiveness = Robot Score / Shortest path length

and

- Run 1 effectiveness = Run 1 steps / Shortest path length

provide measures about effectiveness of robot navigation algorithms

4.3 Robot analysis on random mazes

To generate trial data, the robot navigation algorithm was run on all mazes for 10 times (to account for randomicity of path selection in the algorithm) for all defined exploration path options (8). The total number of run was 237 mazes x 8 exploring paths x 10 run = 18960 runs. Information from each run was used to generate a pandas dataframe, that was in turn saved as a .csv file. Dataframe was then loaded in a Jupiter book (also attached to project submission) for data exploration. Each entry in the dataframe includes information about the maze (size, file name, number of walls, shortest path length) and about the robot performance in the maze (steps in run 0 and 1, known walls after each run, exploration path used, scores, etc). The major outcome from notebook exploration are reported in the next chapter.

5 Performance analysis over maze trial set

5.1 Exploration mode comparison – score metric

As a first analysis on the trial maze set, let' compare average robot performance for different exploration path. In the graph below average score is used as a performance metric for comparison.

Figure 9 Exploration path performance

To ease the comment of the graph, let's report again here the definition of different paths:

0: no further exploration after reaching the centre

1: centre > any corner different from bottom left > Reset

2: centre > bottom left > Reset

3: centre > any corner different from bottom left > bottom left >Reset

4: centre > top left > centre > top right > centre > bottom right > centre > Reset

5: centre > bottom left > centre > bottom left > centre > Reset
6: centre > bottom left > centre > bottom left > centre > bottom left > centre > Reset
7: centre > top left > top right > bottom right > bottom left > Reset

The first consideration that could be made from the graph is that there is a major benefit in continuing exploration in run 0 after reaching the centre for first time (as mode 0 is the only one that interrupts run 0 immediately after reaching centre and this mode is by far the less performant in the graph).
On the opposite site, we can see that the most performant choice is option 5, that instructs the robot to loop between start box (bottom left) and centre box for 3 times before asking for reset. Most probably the reason is that this option is the one that concentrate maze exploration on the crucial path (the one that will be used for scoring) so is the one that more probably allows to take best decision on run 1.
This consideration is reinforced by the fact that second best option is option 6, that is identical to option 5 but with an increased number of loops.

5.2 Exploration mode comparison – additional consideration

On top of best average performance (discussed above) the trial set of mazes allows to explore additional questions, such as possible dependencies of result above on maze size, maze density or maze complexity. Dependencies on maze size is easily explored:

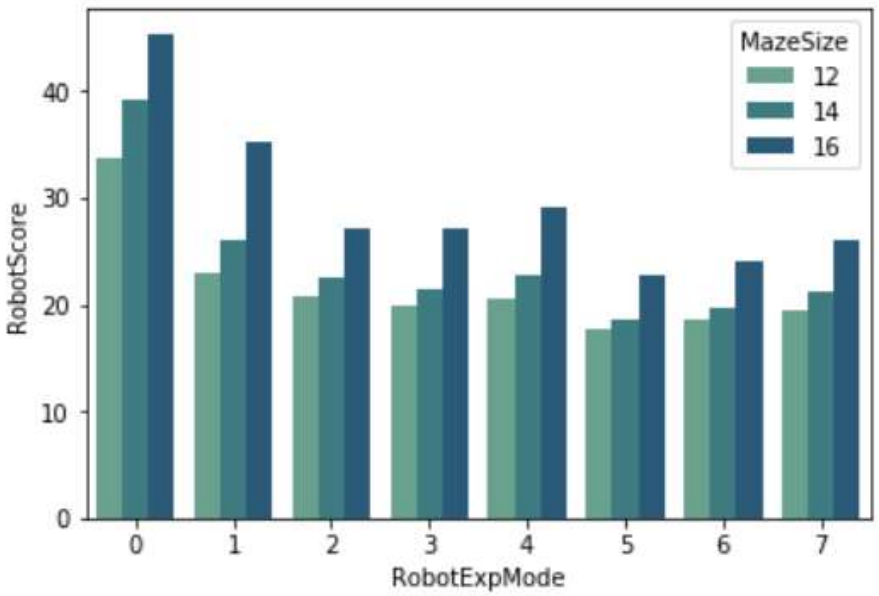
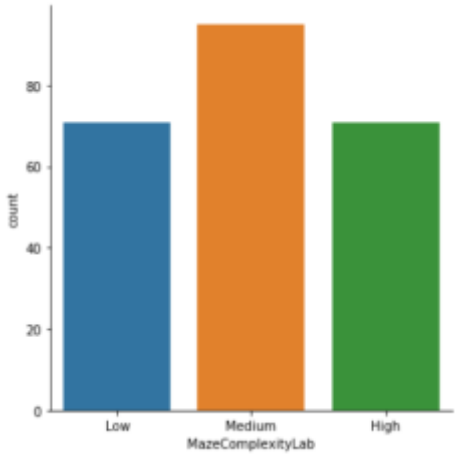


Figure 10 Exploration path performance with maze size as hue

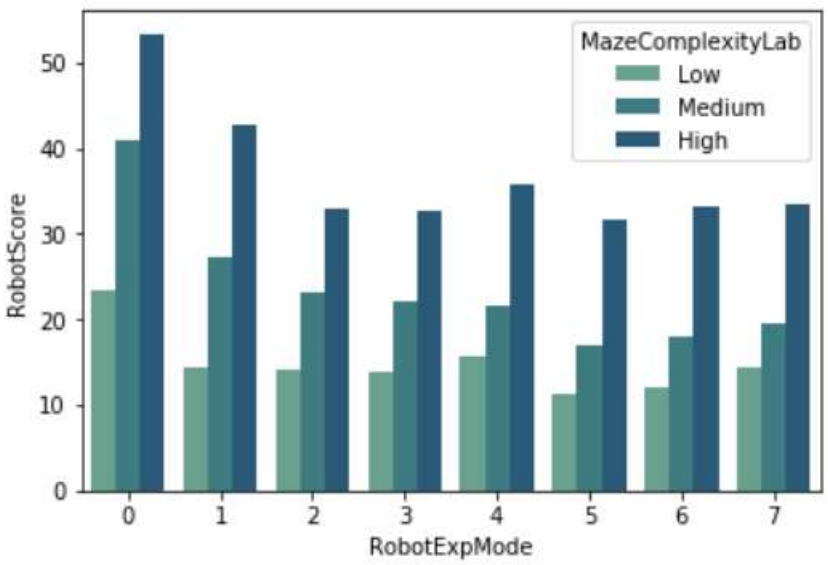
Picture above clearly shows that the order of performance (0=Worst, 5=Better) is independent from maze size (at least for the three sizes explored in the trial set). Regarding dependency on maze complexity, we should first define somehow a measure of complexity for a specific maze. The proposal described in the notebook is to define complexity as the ratio between actual shortest path on the maze and a theorical shortest path on a totally empty maze (both counted in term of robot moves). Using this metric, we defined three cases:

- Simple mazes: ratio lower than 2
- Medium complexity mazes: ratio between 2 and 3.5
- Complex mazes: ratio greater than 3.5

In the definition above, thresholds were chosen in a way that number of mazes in each class is comparable, as shown in the picture below.



Using complexity as a hue for exploration mode comparison bar chart, we obtain the graph below.



It is interesting to note that even if modes 0, 1 and 4 remain the worst choices in all cases, for more complex mazes difference among modes 2,3,5,6 and 7 gets less evident. Nevertheless, mode 5 remain the optimal choice even in this graph for all complexities, even if with a smaller margin.

5.3 Exploration path 5 on test_maze_01.txt

In section 2 we have shown performance of the algorithm on test_maze_01.txt using exploration path 0. As stated, the algorithm resulted in the reported example in 38 steps on run 0 and 25 steps on run 1, for a total score of 26.3. Running on the same maze on exploration path 0 and 5, we obtain the following outcome (run was repeated 10 time to account for randomicity in path selection):

Execution	Run 0 steps		Run 1 steps		Score		% Maze known	
	Mode 0	Mode 5	Mode 0	Mode 5	Mode 0	Mode 5	Mode 0	Mode 5
1	51	147	36	17	37.7	21.9	58%	86%
2	46	145	25	17	26.5	21.8	55%	86%
3	34	148	34	17	35.1	21.9	49%	87%
4	46	139	25	17	26.5	21.6	54%	89%
5	40	152	32	17	33.3	22.1	54%	91%
6	39	145	25	17	26.3	21.8	54%	86%
7	59	159	24	17	26.0	22.3	68%	89%
8	33	147	25	19	26.1	23.9	52%	87%

9	32	130	25	17	26.1	21.3	51%	86%
10	47	132	31	17	32.6	21.4	56%	86%
Average	**42.7**	**144.4**	**28.2**	**17.2**	**29.6**	**22.0**	**55%**	**87%**

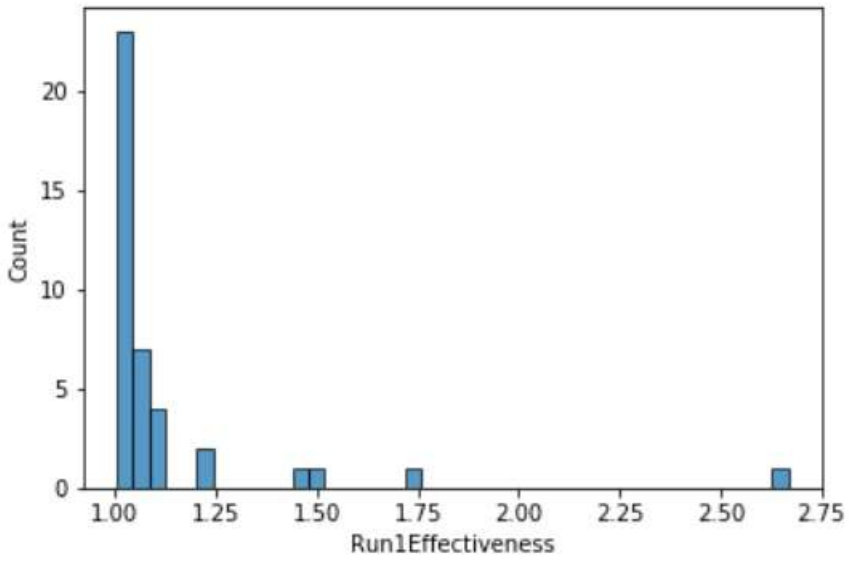
As visible from the table, with the new exploration mode, the length of run 0 increased noticeably (more than 3 times in average), but in all cases (9 over 10) the knowledge acquired in run 0 allowed for shortest path to be taken on run 1. Final score was also considerably lower than previous output (>20% less in average), while the average knowledge about maze moves from 55% to 87%

6 Challenging mazes

One of the requirements of the analysis was to provide a new maze that could result challenging for the chosen algorithm. To address this task, we looked at actual performance of the algorithm (using exploring mode 5) over the whole set of randomly generated mazes. More challenging mazes should be those that results in lower average performance; as a performance index we can use the ranking parameter defined in paragraph 4.2.3, namely:

- Score effectiveness = Robot Score / Shortest path length
- Run 1 effectiveness = Run 1 steps / Shortest path length

Looking first at Run 1 effectiveness, we can initially filter for those mazes with an average value greater than 1 (as a value of 1 means that run 1 was on the shortest possible path). This results in 40 mazes over the set of 237. Distribution of Run 1 effectiveness over these mazes is plotted in the picture below:



The plot shows 6 mazes were the values is significatively higher than 1, with one maze where the average path taken by the robot is almost three times longer than shortest path. A similar analysis done on Score Effectiveness parameter identified the same 4 mazes in the last 4 positions for both indices. These are:

MazeName	Run1Effectiveness	ScoreEffectiveness
ran_14_5_177437.txt	2.67	2.96
ran_12_120_157616.txt	1.75	2.12
ran_14_5_176343.txt	1.50	1.77
ran_16_5_177757.txt	1.47	1.67

Let's consider the worst one, ran_14_5_177437. In the naming convention used, 14 indicates the size of the maze, while the second integer in the name (5 in this case) shows the number of failed attempts to create a new wall done before considering the maze completed. 5 is the lower value used in maze generation, and should bring to a maze not particularly dense in term of walls. Let's have a look:

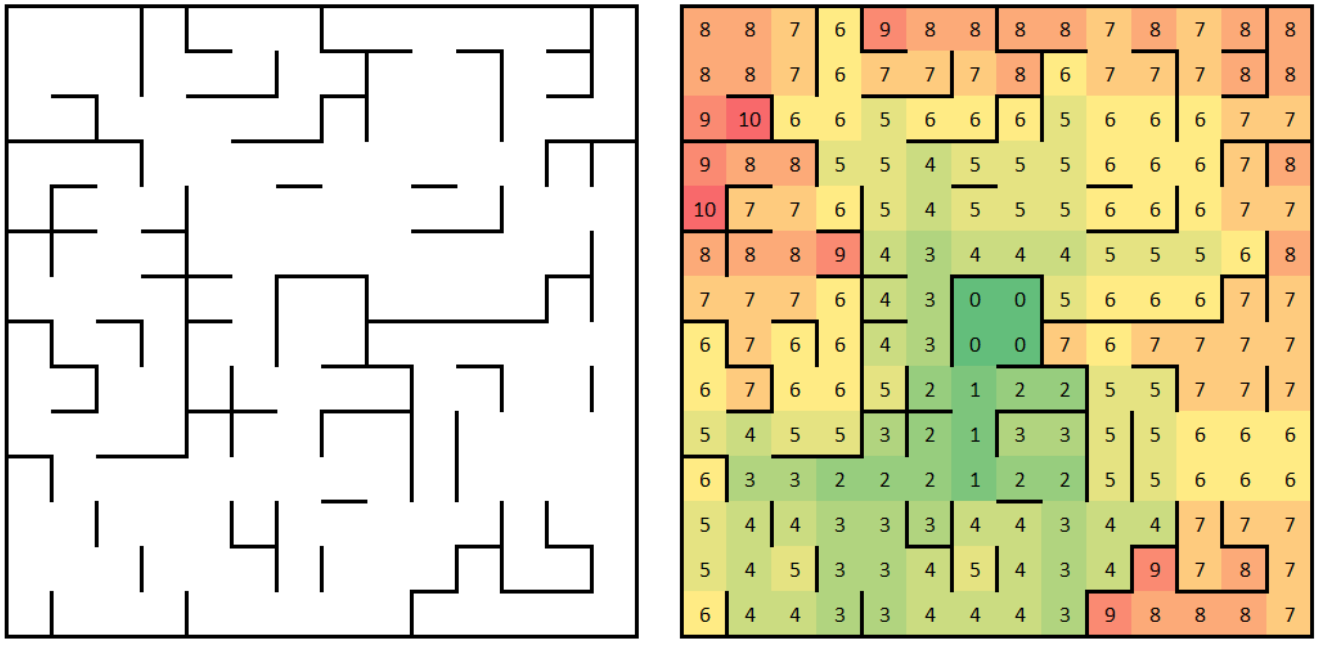


Figure 11 Maze ran_14_5_177437, flooding weight shown in the picture of the right.

As visible in the right picture, the shortest number of moves to reach the centre is 6. Let's have a closer look at robot performance in the 10 recorded runs (remember that we had 10 runs per mode per maze in the dataset) in the table below. Table is sorted in order of decreasing performance.

Iteration	Run0 Moves	Run1 Moves	Run0 Known	Run1 Known	Run0 Maze Discovery	Robot Score
9	47	6	145	145	35%	7.6
7	62	6	194	199	46%	8.1
6	65	6	222	227	53%	8.2
1	69	6	234	239	56%	8.3
2	87	6	280	285	67%	8.9
4	39	14	107	138	25%	15.3
3	39	26	107	184	25%	27.3
5	39	30	107	199	25%	31.3
8	39	30	107	199	25%	31.3
10	39	30	107	199	25%	31.3

The table shows that on the maze under analysis, in half of the cases the run 0 is constant and much shorter (39 steps) than in the other half. When this happens, the

knowledge acquired about the maze remains poor (25%) at the end of run 0, resulting in turn in extremely bad performance on run 1. In order to understand what happens on this maze, a new set of runs was repeated, this time enabling the tracking of the robot during its navigation. Interestingly the behaviours shown in the table was easily reproduced, with nearly half of the new run showing bad performance and the remaining half behaving properly. More interestingly the two groups of runs were characterized by the same set of initial steps, shown in the picture below:

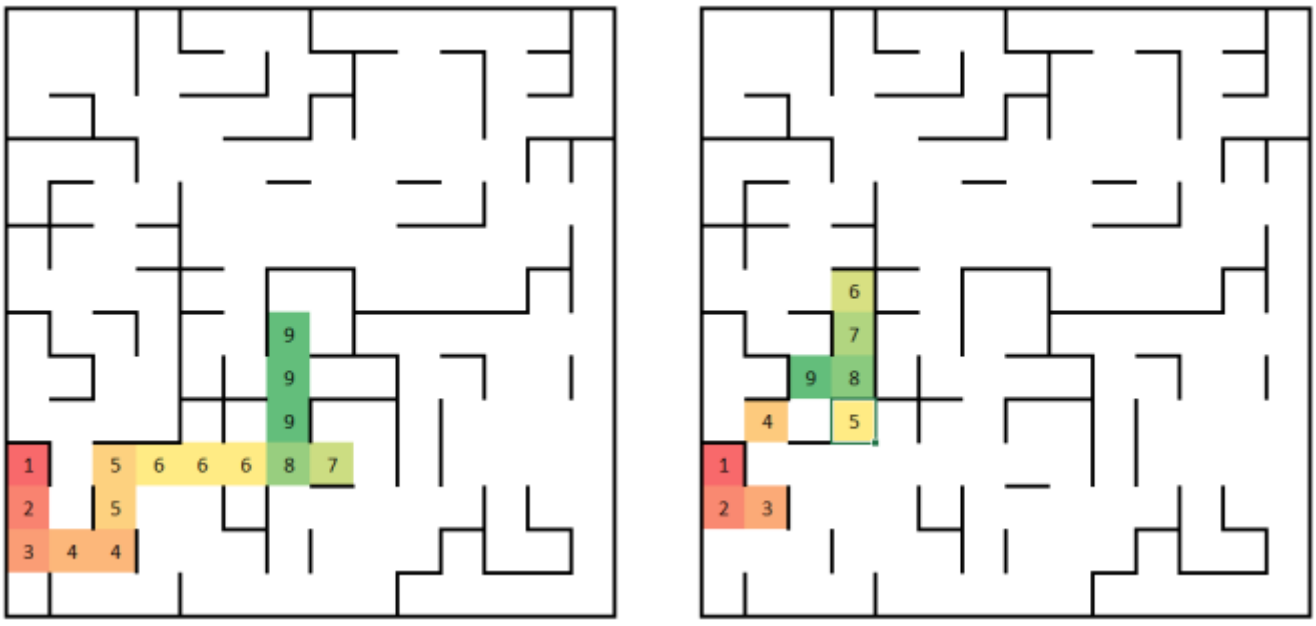


Figure 12Initial steps for bad (left) and good (right) performance runs

Looking at the pictures, we can draw the following considerations:

- The maze has an easy and short path toward the centre moving from bottom left corner in the right direction
- Instead, there is a much more complex path if it moves toward top remaining on the left
- We can argue that at initial stage of flooding, when the robot has minimal clue to take decisions, a random decision is triggering the path chosen (this happens in practice on move 3, when either box [0,1] or box [1,2] is chosen).
- When move 3 is toward box [1,2] the combination of walls is forcing the robot to move toward the top of the maze, entering a longer path before reaching the target. This allows the robot to acquire better information on the maze; in addition, when moving from the centre back to the starting point, the robot will

follow a different path, also exploring the bottom part of the maze. This results in a good overall knowledge at the start of run 1 and in turn into the capability to find the shortest path.

- On the opposite side, when random decision is toward box [0,1], the robot easily find a path toward the centre; in addition, after returning to starting box for run 1, the flooding information calculated based on learned walls systematically force the robot on the path [0, 2]> [1, 2]> [1, 4]> [3, 4]. When in [3,4], the robot will discover the complex part of the maze during run 1, leading to bad performance in the run.
- The explanation above must be completed by an additional tricky observation that is probably the actual key of the problem and a hint to be explored for possible solutions. In exploration mode, (i.e. run 0), when moving from the centre toward the starting point, the robot reaches the starting point with a move from north to south. This means that, when starting the second trip toward the centre (still in run 0), the robot is directed toward south, and will exit the starting box with a backward move. In the code, backward moves have always been limited to 1 box length (as there is no backward sensor and we cannot be sure about the space we have backwards). Finally, this implies that the robot, in its exploration, will never try the move [0,0]>[0,2], that is the actual trap toward the upper part of the maze. After reset, the robot is put in position [0,0] in direction upward, it will move to [0,2] and then toward the longer path during run 1.
- Given the considerations above, we could consider the option to slightly modify the code, instructing the robot to add to additional rotation only steps when back to [0,0] in run 0. This option has not however been explored and is left for further analysis.

7 Improvements

7.1 Improving proposed code

7.1.1 More on exploration paths

I believe that the run 0 exploration implementation described in the document is an effective tool to explore possible exploration strategy. However, the actual code provided has a few known limitations that could potentially results in improved performance once removed.

- It is assumed that the robot will always start run 0 moving toward the centre, and only after reaching the centre the exploration path is queried. Different strategies can be explored, i.e. removing the first step toward the centre and giving to exploration path variable the full control of run 0
- The exploration path value consists of a list of exploration steps that must be completed before closing run 0. An additional control added to the sequence of steps may decide to interrupt the exploration before, to save run 0 steps that are not bringing value. For instance, we can decide to fix a threshold on maze knowledge, so that when the percentage of passages known is over the threshold the robot stops run 0
- The 'challenging maze' analysis outlined a potential source of inefficiencies, given by the fact that the mouse comes back to origin always in direction 'south', so that during exploration will only leave that box moving backward 1 move. An alternative option could be to add one 90 degree / 0 box rotation move before actually leaving the starting box

7.1.2 Longer backward moves

- We limited each backward move to 1 box maximum, as the robot has no backward sensor and cannot trust wall information contained in its 'wall' attribute. However, the combined use of attributes walls, known_hor and known_vert can provide reliable information on max allowed backward move. Moving backward for more boxes could also improve performance

7.1.3 Improve robustness over maze design

- Mazes are assumed to be connected, meaning that from each box of the maze it is possible to reach each other box in the maze. If this is not the case, execution could interrupt ungracefully. This should be managed in a more robust version of the code

7.2 Moving to continuous environment

The situation proposed for evaluation for continuous environment is represented in the picture below:

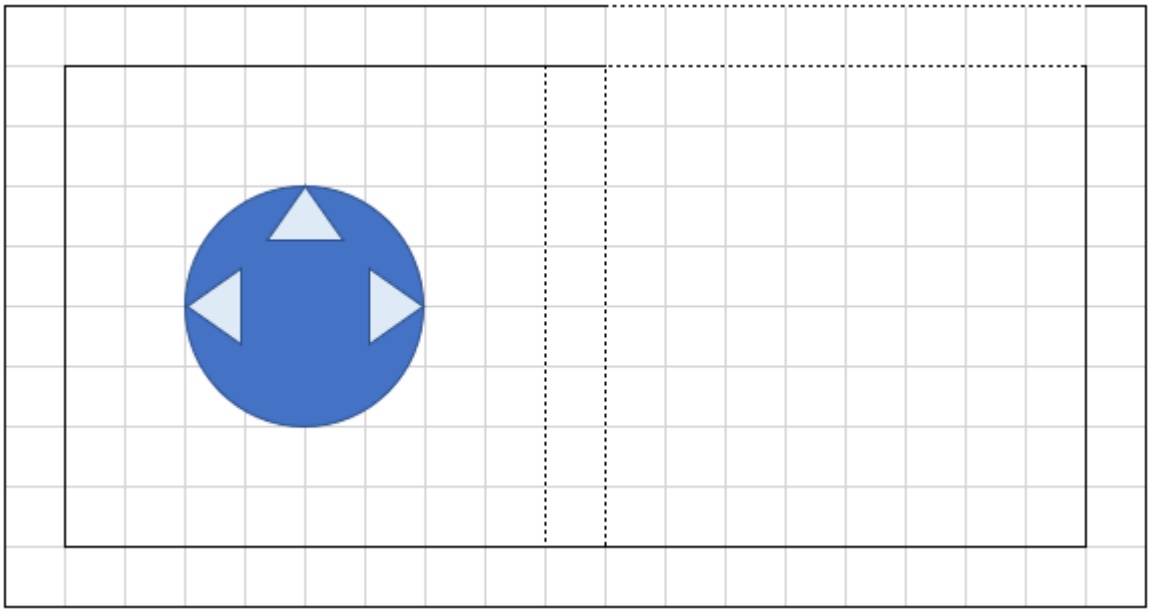


Figure 13 Circular robot in a continuous environment

The box is assumed to be 1 unit large, while walls are 0.1 unit and the robot has a circular shape with 0.2 units radius. I imagine we can assume that rotation of the robot can be 'ideal' so that the robot con rotate maintaining its center in the original position. In this case, designing rotation should not have specific constraints with a circular robot of that size (of course situation would be different for a shape different from a circle). In the moving design, we must always grant the fact that at the end of the move the robot will reach the exact centre of the new cell. This should happen if the robot moves its centre of exactly 0,9 unit (assuming each wall is 'shared' among adjacent cells, as shown in the picture.

8 Acknowledgment

8.1 Algorithms

As stated in the proper section, project started with an internet research for algorithms to be used in maze navigation. Several material could be found on that topic, personally I found useful the Medium blog
https://medium.com/@minikiraniamayadharmasiri/micromouse-from-scratch-algorithm-maze-traversal-shortest-path-floodfill-741242e8510
and I also had a look at the technical paper
'Maze Solving Algorithm', from Gan Zhen Ye , Dae-Ki Kang.

The basic idea for maze generation algorithm (add a random wall and check maze connectivity) also came from internet research, even if I cannot reference a specific site for that

8.2 Code

All code provided in robot.py and randommaze.py modules was originally produced by me for this project. However, some data structures, specifically some global variables and the 'walls' attribute organization were copied by the 'maze.py' modules that was provided as a part of the project.
All code was tested on PyCharm suite; in my software version, maze.py, showmaze.py and tester.py had to be slightly adjusted, specifically to add parenthesis around 'print' argument.