Plot and Navigate a Virtual Maze

Capstone Project for Machine Learning Nanodegree in Udacity

Written by: **Chen YI**
Email: 1021869638@qq.com

1. Introduction

1.1 Definition

There is a maze formed by `n x n` squares, `n` is even and 12 <= n <= 16. Along the outside perimeter of the grid, and on the edges connecting some of the internal squares, are walls that block all movement.

The robot will start in the square in the **bottom- left corner of the grid, facing upwards**. The starting square will always have a wall on its right side (in addition to the outside walls on the left and bottom) and an opening on its top side. In the center of the maze is the goal room consisting of a `2 x 2` square. Our robot can run two times, and total number of steps used will be recorded.

In this project, we will be building a planner system to guide our robot to reach the goal position as quick as possible without any outside helps. Our planner system will only receive robot's sensor signals. Also in this project, we cannot touch the codes on both robots and maze.

1.2 Metrics

- `score = number of steps for first run / 30 + number of steps for second run`
- or `score = timeout` if time runs out
- Metric is the smaller `score` are better.

---

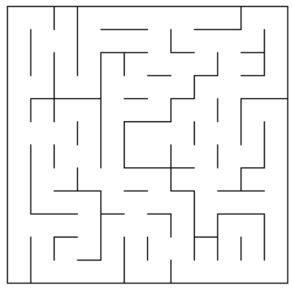
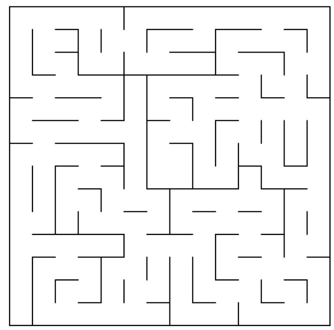
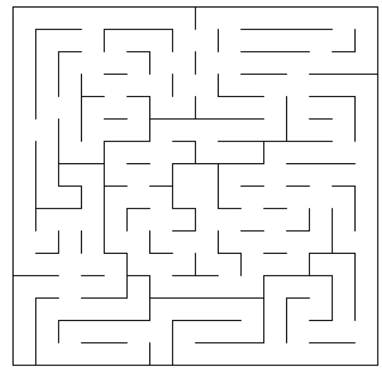
2 Analysis

As there are two systems we need to describe, for the clarification, we will call the outer given system the `Maze` system, and our self-made system the `Planner` system.

2.1 Data Exploration

First let us explore the set-ups of this project.

2.1.1 Example Map Visualization

Maze	`test_maze01.txt`	`text_maze02.txt`	`text_maze03.txt`
Size	12	14	16
Visual			

2.1.2 Environment

- There are only four directions in the system, up, down, right, and left are represented by `up, down, right, left, u, d, r, l`.
- the maze is constructed by `n x n` squares, and we can only indirectly interact with it by moving our robot inside it.
- Our `start` is `[0, 0]`, and that is in the left bottom corner of the `Maze`.
- there are only two runs, the first run is for sensing walls and drawing the map, and second run is for reaching goal position with the shortest path.
- `timeout` is 1000.
- the `Maze` is a discrete system, and there will be no fraction numbers for the coordinates, only integers.
- `Maze` will boardcast if any collision happens.
- Game will be ended either the robot reach the goal at the second run or time runs out.

2.1.3 Robot

- with `sense()` function, the robot will get a list with three integers `[a, b, c]`, indicates the distance from the robot to walls at it's left, heading and right directions.
- the robot send that list to `Planner` and expects the `Planner` returns move command to it.
- Our robot can only move along side the heading direction either forwards or backwards
- Our robot can only turn 90 degrees either left or right of the heading direction.

- For each turn, our robot can execute exactly one turning command and one moving command.
- the robot will stop move along that direction if it hits a wall, but no feedbacks that can be collected for us.

2.1.4 Commands to the robot

- We can tell robot to make turns, and the only valid inputs are `[-90, 0, 90]`.
- Then we tell robot to move number of squres along the heading direction with integer `i`, and `-3 <= i <= 3`.
- These is no other commands.

2.2 Algorithms and Techniques

The main algorithm we used in this project is the `value table` that has been introduced in ***Udacity's AI for robot*** lesson (Udacity, 2016). There are two reasons to use it.

- Firstly, in this project, the `Maze` system is an error-free and discrete system. This simple grid system very suits `value table`.
- Secondly, although the computation cost for `value table` is quadratic to the maze size, but as we know the maze size is less than or equal to 16. The cost can considered bearable.

After the `value talbe` is found, we can simply use it to find best neighours of any spot and hence find path from any spot to the `goal`. We will discuss it more in the **implementation section**.

2.3 Benchmarks

If we let us robot move randomly, it cannot finish the game on time, that is `score = 1000`. Therefore our planner should be able to finish the game before time runs out, that is `score < 1000`. This is the basic benchmark.

---

3 Implementation

3.1 Data Pre-processing

In this project, there will be no data provided before our robot running, and all data we will collect are the sensor signals from the robot as it moves. Therefore no data pre-processing are needed.

3.2 Implementation

3.2.1 Overall Workflow

As there are two runs in total. In the first run, our robot will explore the map. And in the second run, our robot will try to reach the goal as fast as possible.

In the `run1`, the `planner` system conducts a series actions as porcedures.

- In general, this porcedure starts by receiving the `sensor_inputs` from the robot.
- our `planner` system will then `update_map` based on these inputs, and performs `find_goal_position`.
- then `planner` will perform `if_reach_goal` to check if our robot stands in the goal_position area, as there are four goal points in total.
- if `if_reach_goal` flags up, our `planner` will guide the robot move around four `goal` locations to check if there no undetected walls in this area. If it's confirmed, our `planner` will give singal `['Reset', 'Reset']` to notify the `Maze` system for the run 2, or otherwise it back to the searching stage.
- After this, `planner` will `find_path_to_goal` based on the `value table`, and `find_move_from_path` to return a move command.
- then our system first `execute` this result from above in the `planner` system for movement sychorinization, and then `return` the move command to the outer `Maze` system, to finish one loop.

To be more specific, the `path` we are looking for above is the shortest path between the `start` and `goal` positions. The reason is that this `path` will determine our score in second round, and we need to make sure the `path` really is the shortest.

In the second run, the `planner` will ignore the sensor signals and just perform `find_path_to_goal`, `find_move_from_path`, `execute` and `return_command` repeatly.

We will discuss `update_map`, `find_goal_position`, `if_reach_goal` and `execute` in this section as they are basic elements that make the whole system works. There will be plenty details inside `find_path_to_goal` and `find_move_from_path` as `value table` are involved, and we will discuss them in the **3.3 Algorithm** section

3.2.2 `update_map`

We start the game with a blank map, and as our robot moves, we can get more information about walls in the given maze. Therefore we need something to store that information. I construct a new class called `Map`, and it's a matrix with `maze_dim * 2 + 1` by `maze_dim * 2 + 1` dimension where `maze_dim` is the maze's dimension.

All entries are initialized as `0`, then we use `pos_map(maze_location)` that transfering `Maze` cooridnates into `Map` coorindates, and use `1` to indicate if there is wall between `pos_map(two adjcent maze positions)`.

The reason we do that is that we need to know if two adjacent grids in `Maze` are walkable, that is if there is a wall between them. Therefore `Map` class has a function `is_connect(pos1, pos2)` to tell us that.

We use `update_map(robot_location, direction, distance)` to update walls as our robot senses walls at its current position in the given direction and given distance. distance and positions here are `maze` distance and cooridnates. So we can take in the sensor inputs and update walls accordingly inside `Map`.

3.2.3 `find_goal_position`

As we cannot touch the `Maze` directly, and hence cannot tell where the goal is exactly at begining. But we know the walls in our `Map` class, and also we know that the goal position is a `2 x 2` square that has no walls inside it in the center of `Maze`, so we can search through the center area of the maze, and find that desired square.

That goal position might be wrong at start since we has no information about walls in the map. But it provide us the direction that where our robot should explore, and walls will be updated as the robot moves. If our robot find any walls that kills our current goal square, `find_goal_position` will look for another possible square that matches the description. As the process going, our goal position should become accurate.

Since the `goal` square has four corners, this function will `return` a `list` of 4 coordiantes which form a square.

3.2.4 `if_reach_goal`

We need our `Planner` system to tell us if we should stop and notify `Maze` system. In order to do that, our `if_reach_goal` function will compare robot's current location with list of goal positions to see if current location matches any of them. return `True` if it matches.

Another important thing is that after `if_reach_goal` flags up, we still need to make sure that the `2 x 2` square really is the goal position. That is we need our robot to walk through each one of 4 positions in the goal `list`, and heading towards any of other grids. The robot's sensors will do the rest job for us. If the sensor detect any new walls inside that square, the `Map` will updated and our `find_goal_position` will find a new goal position, and `Planner` will be back to normal porcedures.

3.2.5 `execute`

After move command is found, we need our robot to `execute` that command in `planner` before return it to `Maze` system, and the robot's position must be always same in both system. That is we need to make sure the movements in two system are synchronized, and we know that our robot starts with `[0, 0]`, and hence we update its coodinates acording the move comand.

This will become problematic when collision happens. When `collision` happens, our robot will freeze and enter the next round, but it sends no feedback to `Planner`. Hence we need to make sure the robot has the same behaviours in `Planner` system.

This does not mean that we need to avoid collision. This simply means if our robot hits any wall, our `Planner` system must be awaring. In that case, we should stricted our robot do not move backwards, unless it has moved through the exact path between two exact coordinates before. The reason is that our robot has no sensor to scan behind it and our `Planner` system cannot tell if there is any walls behind.

3.3 Algorithms

3.3.1 `value table`

We are using `value table` to help us implimenting `find_path_to_goal` and `find_move_through_path`.

The `update value table` code are below:

```
def update_value(self, target, cost):
    '''
    update value map based on the target positions as 0 
value, and given cost
    '''
    self.values = [[99 for row in range(self.maze_dim)] for 
col in range(self.maze_dim)]
    adjs = ['u', 'r', 'd', 'l']
    changed = True
    while changed:
        changed = False
        for row in range(self.maze_dim):
            for col in range(self.maze_dim):
                if [row, col] in target:
                    if self.get_value([row, col]) != 0:
                        self.write_value([row, col], 0)
                        changed = True
                for neighbour in adjs:
                    util = self.get_value([row, col]) + cost
                    location = [row + dir_move[neighbour][0], 
col + dir_move[neighbour][1]]
                    if 0 <= location[0] < self.maze_dim and 0 
<= location[1] < self.maze_dim:
                        if self.map.is_connect(location, 
[row, col]):
                            # print util, location, 
self.values[row][col]
                            if self.get_value([location[0], 
location[1]]) > util:
```

```
self.write_value([location[0], location[1]], util)
                                changed = True
```

The `value table` is a matrix that has the same dimension as the `Maze` system, and all entries are initialized as `99`. After the system `update_map` and `find the goal positions`, the system will perform `update_value`. The algorithm will first marked `goal` positions (four of them to be exact) as `0`. Then the algorithm go through each position `[x, y]` of that table (other than the goal positions) to do the following:

- get the `value[x, y]`, and assign `util = value[x, y] + cost`, where `cost = 1` in this project.
- it collects all neighbours of `[x, y]` that are walkable, that is no wall between two spots.
- check if `value[neighbour] > util`, if `True` then assign `value[neighbour] = util`
- repeat this process until no more changes, and it means done.

In practice, for minimizing the computational costs, the `planner` will only `update` value table if any new walls are added to the `Map`, as no value will be changed if the `Map` has no changes.

3.3.2 `find_best_spot_in_distance_n`

This is a direct application of `value talbe` that underneath `find_path_to_goal` and `find_move_from_path`.

The procedure is simple, for a given coordinate `X` and given distance number `n`, the procedure will check through all the walkable spots within the range of `n` steps from `X` in four directions. It will find the spot `Y` that has the smallest value that `value_talbe(Y) + cost from X to Y`. Then `return` the coordinate `Y` and step numbers `n` from `X` to `Y`.

Here we want to find farthest spot to go in one turn with max step number `n` that with the smallest value, therefore we use non-strictly smaller sign for value comparison, and loop through the spots with distance from `1` up to `n`. This will make sure that the `return` result really is the best spot we are looking for.

3.3.3 `find_path_to_goal`

To be more specific, the `path` found in this porcedure is the shortest path between `start` to `goal`.

After `value_table` is updated, we start from the `start`, execute `find_best_spot_in_distance_1` recurssivly on the output coordinate. We also record those coordinates sequentially on a `list` namely `path`. This loop will end when the `goal` position reached, and return that `path`.

3.3.4 `find_move_from_path`

After the `path` has been found, we want our robot to go through it to check that this `path` really is the shortest path between `start` to `goal`. That is there is no undetected walls to stop this `path`. If any new walls are detected, `value table` will be updated, and the shortest `path` might change accordingly.

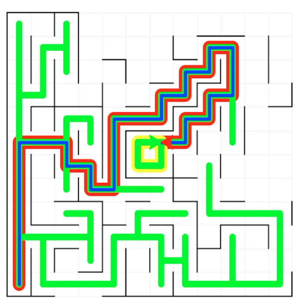
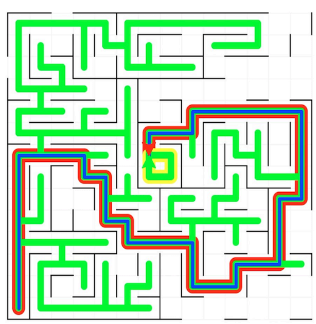
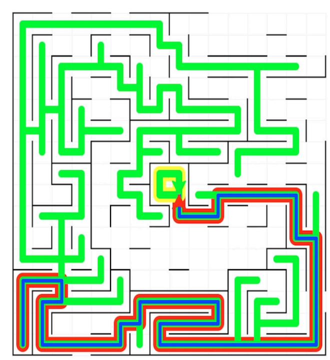
Therefore we begin by checking if our robot is on the `path` by checking `if robot_position in path`. If `True`, then our robot will just keep walking along the `path`.

If our robot is not `in` the `path`, we know our robot was on the previous `path` before. Also both new `path` and previous `path` are starting from the `start` and separated somethere later, our robot just need go back through its previous actions and will eventually lands on the new `path`. For doing that, we add a `stack` namely `record` to the `planner` and to record its move commands, if our robot needs to go back to meet the new `path`, we just `pop` out the latest element from the `record` and `execute` the `reverse_move` of that element. Of course, the `go_back` action itself should not be recorded. We keep doing the `go_back` until our robot reach the new `path`, that is `robot_location in path == True`.

To find the desired move command following the `path`, robot will execute `find_best_spot_in_step_3` where `3` is the maxmium moving steps for one turn by setting. Then we check if the returning coordinates `in path`, and if `True` then figure out the rotation `angle` and return `[angle, 3]`, otherwise repeat the same precedure with step number `2` and `1`. With step number `1`, a returned coordiante will `in path` guaranteed, as we used the same funtion to find the `path`.

4. Running results

The performance for each example mazes are the following. please note that the following result are deterministic, as we have not added any randomly factors into the system:

maze	`test_maze01.txt`	`text_maze02.txt`	`text_maze03.txt`
size	12	14	16
score	32.667	41.533	54.733
final value table	23 22 21 14 13 12 11 10 11 10 09 10 24 21 20 15 14 13 12 09 08 07 08 09 25 22 19 16 15 12 11 10 09 06 07 08 24 23 18 17 14 13 12 11 04 05 06 07 25 24 23 22 15 14 13 02 03 06 07 08 24 23 22 21 16 00 00 01 04 05 08 09 25 24 21 20 17 00 00 02 03 06 07 08 26 23 22 19 18 19 20 05 04 05 06 07 27 24 25 26 19 20 19 18 05 06 07 08 28 27 28 27 20 19 18 17 06 13 12 09 29 26 25 26 21 20 17 16 13 12 11 10 30 25 24 23 22 19 18 15 14 13 12 11	26 25 24 23 22 19 18 17 16 15 14 13 12 13 27 26 25 24 21 20 19 18 17 10 11 12 11 12 28 27 28 23 22 21 20 21 22 09 08 09 10 11 29 30 29 30 31 36 05 06 05 06 07 08 09 10 32 31 32 33 32 35 04 03 04 05 06 07 08 09 33 34 35 34 35 34 01 02 03 08 07 08 09 10 36 35 34 33 34 33 00 00 04 07 08 09 10 11 37 36 33 32 31 32 00 00 05 06 25 10 11 12 38 37 32 31 30 31 30 25 24 23 24 23 14 13 39 38 33 30 29 28 29 24 23 22 21 22 15 14 40 39 38 39 40 27 26 25 24 21 20 19 16 15 41 36 37 36 29 28 27 26 23 20 19 18 17 18 42 35 34 35 30 29 28 25 22 21 20 19 20 19 43 34 33 32 31 30 31 24 23 22 21 20 21 20	30 29 28 27 26 25 24 23 22 21 22 21 20 21 20 21 31 30 29 28 27 26 25 22 21 20 21 20 19 18 19 20 32 31 30 29 28 27 24 23 20 19 18 17 16 17 18 19 33 32 31 28 27 26 25 22 21 18 17 16 15 14 15 14 34 33 32 29 28 29 24 23 20 19 18 17 14 13 12 13 35 34 33 30 31 30 11 12 13 14 15 16 13 12 11 12 36 35 32 31 10 11 10 11 14 15 16 07 08 09 10 11 37 36 47 46 09 08 09 00 00 06 07 06 07 08 09 10 38 37 38 45 08 07 06 00 00 05 04 05 06 07 08 11 39 42 43 44 07 06 05 06 01 02 03 04 05 08 09 12 40 41 44 45 06 05 04 03 02 03 04 05 06 07 10 11 41 42 43 44 45 06 05 04 03 04 05 06 07 18 17 12 46 45 44 45 46 47 06 05 06 05 06 23 22 21 16 13 47 42 43 44 45 46 33 32 31 30 29 22 19 20 15 14 48 41 38 37 36 35 34 25 26 27 28 21 18 17 16 15 49 40 39 38 37 36 35 24 23 22 21 20 19 18 17 16
final visual table			

The green line is the robot's trace in run1, and the red line is the robot's trace in run2. The blue line is the shortest path from `start` to `goal`.

The robot's running record are also available here:
- [robot running records for example mazes]

4.1 Justification

The above results shows that our robot did not fully explore the maze, but this will not affect the shortest path it will get.

The resulting score numbers are acceptable, as our robot find the shortest path on time. Now we get a better benchmark for further improvements.

4.2 Refinement

4.2.1 Obersvations

From the above records, we can spot a problem of our current algorithm. Since there are multiple potential shortest path candidates, our robot has to switch between them all the time as we are detecting more walls to make those candidates longer and longer. As a result, there will be so many repeated runnings that are redundant. I don't think that's a problem caused by the value table algorithm, this seems being caused by the workflows. Maybe we are trying to find the shortest path too early when the robot is exploring the map in `run1`.

4.2.2 New Method

One alternative method in `run1` is that when our robot's current path is being blocked by new detected walls, instead revsering moves and going to a new path between `start` to `goal`, it now continuous its jounery to the goal by recalculate a new path between itself and `goal`. When the robot reach `goal`, it goes back to `start` by the same algorithm with `goal = [0, 0]`.

We repeat the above process, and for each time we reach either `goal` or `start`, we find the shortest path between `start` and the real `goal`, and compare it with the previous shortest path. When two path are the same, it means we found the desire path and move to `run2`.

The idea behind this method is that our robot now puts its priority on learning now. Instead focusing on finding the shortest path first, we now focusing on the detecting the walls that stop the walking between `start` and `goal`. The more walls we found in between, the longer the shortest path will become. As the process goes on, the path we found in each loop will converge to the real shortest path of the `maze`.

4.2.3 New Workflow

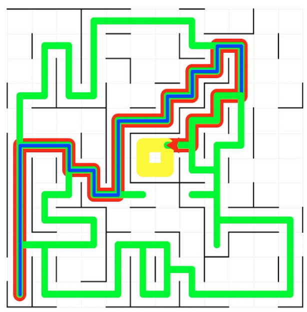
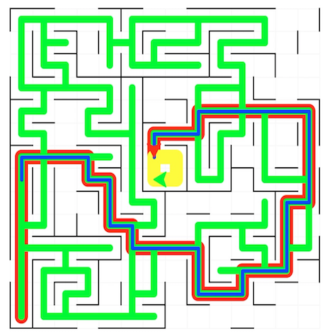
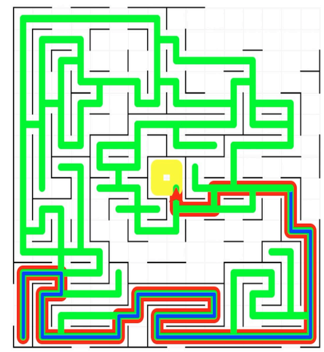
This alternative method does not require the change in `value table` or `update value table`, so I will not put my code in here. The new workflow in `run1` is this:
- `update map` and `update value table` based on given sensor signals.
- instead finding the shortest path between `start` to `goal`, we now find the shortest path between `robot`'s current position to `goal`.
- Then we find move commands based on the path found and `execute` and `return` the result.
- when the `robot` reach real `goal`, we find the shortest path between `start` to `goal` and store it.
- then we guide the robot back to the `start` by setting `goal = [0, 0]` and `update` the value table.
- after the robot reach `start`, we set `goal` back to the real one and find a new shortest path between `start` to `goal`, then we compare it with the old shortest `path` we stored eariler, two result will come out:

- If the two paths not same, then we replace the previous path with the new one and repeat the above precess, and for each time we find a new shortest path, we compare it with the previous one.

- if the new path is the same as the previous one, it means we found the real shortest path in the maze and we will send `reset` singal to the `outer` system and move to `run2`.

4.2.4 New Results

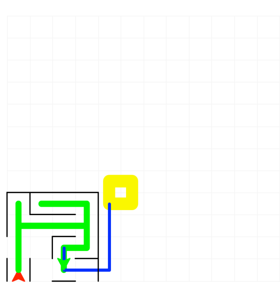
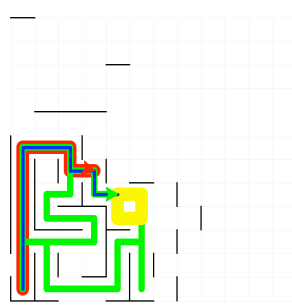
We can see from the result table below, new method gets much better score than the original workflow.

maze	`test_maze01.txt`	`text_maze02.txt`	`text_maze03.txt`
size	12	14	16
new score	26.8	31	38.4
previous score	32.667	41.533	54.733
visual table			

Record videos can also be found on the youtube playlist.

5 Conclusion

5.1 Self-Made map and corner cases.

Maze	bad_maze01.txt	bad_maze02.txt
Size	12	12
Visual		
Score	`Starting run 0.` `Maze is bad, no route to the goal`	`Starting run 0.` `Cannot reset - robot has not hit goal yet.`

In `bad_maze01`, there is no route to the `goal`, and in `bad_maze02`, there is another sqaure that fits the `goal`'s description. Both caes are not restricted by the problem setting, and our `planner` cannot finish the game in both cases.

5.2 Reflection

The really problem in this project is not find any path to the `goal`, but really is finding the shortest path from `start` to `goal` as fast as possible. And yes, we did. The results from our `planner` is determinstic, as there is not room for randomness by the setting, as there will be no locally optimazition or error. This is a project for the **Machine Learning** course, but seems we did not use much ML techniques in this project.

Some other thoughts:

- will exploration result be stored somewhere and reused later?
  - we can output the `value talbe` and store it, but would that be resueable?
- what would happened if walls are changed during the robot running?
  - our `planner` can only `add` walls on `map` but not `remove` walls at this stage. but this workable.

Clearly, as we used matrix and value table for our planner, it can not be used in a continuous system, or mazes with large size at this stage. But we can cut the big maze into small pieces, and maybe it solvable.

5.3 Further Improvements

To make the maze more complex, we can make the environment into continuous. That is coordinates are floating numbers, and walls will have thickness. Also Errors can be introduced to make it even harder. In this environment, a more advanced algorithm will be required.

I would recommend SLAM in that case, and it's short for Simultaneous Localization and Mapping.

6.Reference

Udacity, 2016, *Artificial Intelligence for Robotics (cs373)*, [https://www.udacity.com/course/artificial-intelligence-for-robotics–cs373] (last rechieved in November 2016).