Capstone Project

1. Introduction

1.1 Definition

The 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; the robot must make it here from its starting square in order to register a successful run of the maze.

Our robot can run two times.

For completing this project, we need our robot to be able to move to the goal position as quick as possible without any outside helps. Also in this project, we can only touch the code of robot itself.

1.2 Metrics

- `score = number of steps for first run / 30 + number of steps for second run`
- 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

2.1.1 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 as the `Maze` class, and we can only indirectly interact with it through our robot
- Our starting point is `[0, 0]`, and its 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.
- Game will be ended after time runs out, and `score` will be `1000`

2.1.2 Robot

- with `sense()` function, we can get a list with three integers `[a, b, c]`, indicates the distance from the robot to walls at it's left, heading and right direction of the itself.
- Our robot can only move along side the heading direction both 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
- We will get feedbacks if our robot hits the wall, and it will stop move along that directly.

What inputs we can feed to our robot?

- we can tell robot to move number of squres along the heading direction with integer `i`, and `-3 <= i <= 3`.
- we can tell robot to make turns, and the only valid inputs are `[-90, 0, 90]`.

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 [reference here].

2.3 Benchmarks

Our robot should be able to finish the game on time.

---

3 Implementation

3.1 Data Pre-processing

In this project, there will be no data provided before our robot running, and all data are collected as the process goes are the sensor signals in robot's left, front and right directions. Therefore no data Preprocessing are needed.

3.2 System Workflow

the whole system repeats a series actions as loops several times.

- In general, this loop starts by receiving the `sensor_inputs` given by the outer system `Maze`.
- Then our `planner` system will `update_map` based on these inputs, and performs `find_goal_position`.
- robot then will perform `if_reach_goal` to check if it stands in the goal_position area, as there are four goal points in total. If it's confirmed, our `planner` will give singal `['Reset', 'Reset']` to notify the `Maze` system.
- Else, it will `find_path_to_goal` 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 the loop.

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.

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. Of course, there are plenty of details inside `find_path_to_goal` and `find_move_from_path` in the above workflow, and we will discuss them in the **2.2 Algorithm** section

3.2.1 `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 transfer `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 positions. So we take in the sensor inputs and update walls accordingly inside `Map`.

3.2.2 `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 the desired square.

That goal position might be wrong at start since we has no information about walls in that area. But it provide us the direction that where our robot should explore, and walls will be updated as we 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 four positions which forming that square.

3.2.3 `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 four gird in the goal list, and heading towards any of other grids. The `Maze` system ifself 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 back to the loop.

3.2.4 `execute`

After move command is receivd, 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.

There we need to make sure the movements in two system are synchronized. This will become problematic when collision happens. When `collision` happens, our robot in `Maze` system will freeze and enter the next round, but it sends no feedback to us. 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 aware of them. In that case, we should stricted our robot do not move backwards, unless it has moved through exact route between exact two positions before. That is because our robot has no sensor to scan behind it and our `Planner` system cannot tell if there is any walls behind.

3.3 Algorithms

There are two main actions in the workflow that require algorithms to be able to perform self-planning and self-driving, they are `find_path_to_goal` and `find_move_by_path`. We will discuss these phase in this section.

3.3.1 `find_path_to_goal`

To be more specific, the path we are look for here is the shortest path between the `starting point` and the `goal` position. The reason that we are not look for the path from robot's `current position` to the `goal` is that this `path` will determine our score in second round, and we need to make sure the `path` really is the shortest.

The algorithm we used here is the `value table` that has been introduced in ***Udacity's AI for robot*** lesson [reference here]. The updating 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` 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:

- it collects all neighbours of `[x, y]` that are walkable, that is no wall between two spots.
- it find the neighbour that has the lowest value in the value table.
- update the value of position `[x, y]` as `value[neighbour] + cost`, where we assign `cost = 1`.

if no updates for a scan through, it means done.

For each loop, the `value_update` will only performed if any new walls are added to the `Map`, as no value will be changed if the `Map` has no changes.

After `value_table` is updated, we start from the starting position, and find the neighbour with the lowest value, up to the goal position, and this will give us the `path` that looking for.

With the `path`, we will be able to `find_next_move` from that path,

3.3.2 `find_move_from_path`

In this phase, we begin with checking if our robot is in the `path` we have just found above. If yes, then our robot will just keep walking along the `path`.

In the case, as our robot was on the previous `path` before. This means our `Planner` has a shorter path now from starting point to the `goal`. We will need our robot go through the new path to check if it is the shortest. As we know both new `path` and disapeared `path` are starting from the `starting point`, so our robot will just go back through its previous steps and will eventually reach the new `path`. In order for doing that, we add a `stack`: `self.record` to the `Robot` class and to record its movements, if our robot needs to perform `go_back`, we just `pop` out the latest element of the `stack` and perform the `reverse_move` of that element. Of course, the `go_back` action should not be recorded. We keep doing the `go_back` until our robot reach the new `path`, that is robot's `location` is in the `path`.

To find the desired move command, robot will look at the best walkable spot (with the smallest `value`) with range `3` in four directions, then check if that spot is in the `path`. If `True` then figure out the rotation `angle` and return `[angle, 3]`, otherwise repeat the same precedure with ragne `2` and `1`. With range `1`, a returned command is guaranteed, as we used the same funtion to find the `path`.

3.4 Running results

By putting all the actions above together, we get a working system for the project. The performance for each given 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
result	33.067	41.367	58.567