CS657 HW1
Red ID: 817408235
Name: Wen Lin

PROJECT REPORT

Introduction

This is an introduction to the rule based 2-D maze solver, which including six parts, position, moveDirection, grid, sensor, robot, and visualize. Position, grid and sensor belong to the simulation of a robot, and robot is the strategy chooser, who decides which cell in the maze will be next step. Visualize is the last part that indicates the path a robot choose to find the destination in a maze. Details on design issues and features will be listed following.

Design

The parts of the maze runner are independent. In this case, different parts just use the API provided by other parts and has no idea on how each part was implemented.

Part1 Position

Position is a class that can be used to index the maze grid. (x, y) is the format of position. Four methods are provided in position:

- `position move(positoin step)`
  Move the current position to another position according to the given step, which offers the moving direction. Update the current position to the new position.
- `boolean equal(position posToCompare)`
  Given two positions, check if these two positions are the same. If two positions are the same, return true, otherwise return false.
- `void assign(position assignTo)`
  Given two positions, assign a position object to another position object.
- `void printer()`
  Print the information of a position object, which are x and y.

Part2 moveDirection

moveDirection is a class provides move direction for the robot. Directions (left, front, right) are decided according to the current angle of the robot.

- `dirUpdate(int angle)`
  With the input angle, `dirUpdate` will return the corresponding direction, represented by position.

Part3 Grid

Grid simulates the maze board. Grid can create arbitrary percent of obstacles according to the size of the maze board. Grid needs 7 parameters to create a grid object, which are grid width, grid height, percentage of obstacles, start position `(x, y)`, and destination position `(x, y)`. Functions provided by grid class are listed as following.

- `void gridInitializer()`
  Initialize all cells in a grid with 1, which means all cell are legal choice.

- `void obstacleCreator(int obsPercent)`
  Taking the percentage of obstacle as an input, generate certain number of obstacles. Obstacles are indicated by the value 0, obstacles are unavailable when a robot moves.

- `int getValue(position indexPox)`
  Given the index position, getValue() will return the value in the index position.

- `void printGrid()`
  Given a grid object, printGrid will print all the values in the grid.

Part 4 Sensor

A sensor is the part store all the data, which are available moves (including three directions, left, front and right), and maze grid information. A sensor provides a robot the moves that are legal (in maze grid and not obstacle). When the program starts, a grid will be initialized in a sensor. To create a sensor, following parameters are needed: grid width, grid height, percentage of obstacles, x of start position, y of start position, x of destination, and y of destination. Functions provided by a sensor are:

- `boolean inMaze()`
  A sensor will check and make sure itself is currently in the maze grid and avoid the obstacles in the maze grid.
- `boolean isObstacle()`

Check if the current position is an obstacle.

- void leftChecker() & void frontChecker() & void rightChecker()
  A sensor will check three directions and collect available moves.
- boolean leftIsBlocked() & boolean frontIsBlocked() & boolean rightIsBlocked()
  If there has choice available from a certain direction, it means the direction is blocked.
- `boolean destInFront() & boolean destInLeft() & destInRight()`
  If one of the collections of a sensor contains the destination, then the function of a certain direction will return true.
- `boolean destNotIn()`
  If destination not in the three collections, return true.
- `boolean allEmpty()`
  Collections contain no elements, return true. Otherwise, return false.
- `void cellPrinter()`
  Print the elements in all collections.

Part 5 Robot

A robot is a maze runner, whoes task is to find a path to the destination in the maze. A robot is composed by sensor, current position, current angle, an angle counter, move directions, a path container contains the solution, original point, and a final grid. A robot has the following functions:

- `void angleComputer()`
  Compute the angle according to the start position, and the destination position.
- `void angleTuner()`
  Change the current angle when need, always change in anti-clockwise.
- `void pathMarker()`
  Path marker will mark the cell as an element of the solution every time the robot moves to it. It also changes the cell value to 5, which means the cell has been tried. After move to the cell, robot will re-compute the angle and re-initialize the move directions.
- `void pathPrinter()`
  Print elements in the path container when a solution is found.
- `pathFinder()`
  Path Finder contains all the rules used by the robot. There are 10 rules, which represent 10 situations. One and only one rule will be fired every time when the condition of the rule is satisfied. Rules will be analyzied in features.

Part 6 Visualize

Visualization file is a class that will turn the path of the robot to a panel.

Features

Rules

There are 10 rules in this rule-based maze solver. Analysis of each are listed as following.

- Rule 1: Destination has been found.
  When the current position of the robot is the destination of the maze, rule1 will be fired. Rule1 will also be fired when the value of the destination of the maze is 5.
- Rule 2: No solution
  If the robot cannot find a available move after change angle 7 times, the robot will go back to the last step in path and pop the current position from the path, since it is not a cell in the solution. When the robot pop all the elements in the path, return to the original of maze and the angle counter equals to 7, which means the robot tried all possible solutions but did not find a solution. Then no solution rule will be fired.
- Rule 3: Destination in the front collection of the sensor
  If the destination in the front collection of the sensor, then the robot will always choose front as its direction.
- Rule 4: Destination in the left collection of the sensor
  If the destination in the left collection of the sensor, then the robot will always choose left as its direction.
- Rule 5: Destination in the right collection of the sensor
  If the destination in the right collection of the sensor, then the robot will always choose right as its direction.
- Rule 6: Choose front direction and move
  If the destination is not in the collections of the sensor, and front direction has available cell, the robot will choose to move front. It is a wise choice, since after every move the robot will re-compute the angle, and front direction will always be the direction points to the destination.
- Rule 7: Choose left direction and move
  When the destination is not in the collections and front is blocked, if left move is available, robot will move left.
- Rule 8: Choose right direction and move
  If all the rules above haven't fired, the rule 8 will be fired.
- Rule 9: Change angle

If the robot angle counter is 7 and all collections are empty, change angle rule will be fired.

- Rule 10: Move back to the last step

  All the rules above failed, rule 10 fired. When rule 10 fired, the robot will go back to the last step in the path and pop the current position in the path. After going back, the robot will start a new search, since the cell has been tried is marked, robot will not try that again.

Shortest path

In order to find the destination with minimum steps, the robot will change its angle accordingly. The change of angle helps the robot always take the direction that will find the destination. The robot also divides a problem into many sub-problems. When the robot move to a new cell, the start position of the robot is the current cell, then the robot just focuses on solving how to reach the destination from current position.

Outcome and discussion

Here are the pathes a robot choose with different percentage of obstacles in the maze. As the outcmes show, some times the robot cannot find a perfect solution for the maze, even when there are less obstacles.