Project summary

Xin Lin (863522)

Master of Information service management

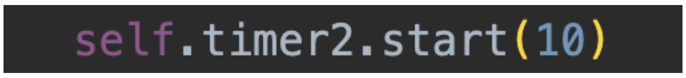
22.May.2024

General description

A program that stimulates the actions of a robot vacuum cleaner in a pre-defined environment is created. In my opinion, the difficulty level my project has been implemented in is **HARD**.

Instructions for the user

Launch the program by runing main.py. You may customize the map by placing boundaries and obstacles in the environment. You can also add as many robots as wish into the simulation. When a robot is destroyed, you can click on the robot to reset it. Otherwise, you can click on the robot to switch vacuum power and cleaning algorithms to observe different behaviors. When you add obstacles, you need to press key W after clicking the button 'Add obstacle' and hold the key pressed down to add as many obstacles as you wish. You will find detailed instruction by clicking 'Rules' button.
If you wish to make the robot movement slower to observe better, you may change the Qtimer interval in method start_cleaning of class GuiWindow. By increasing the value in the bracket you can make the movement slower.



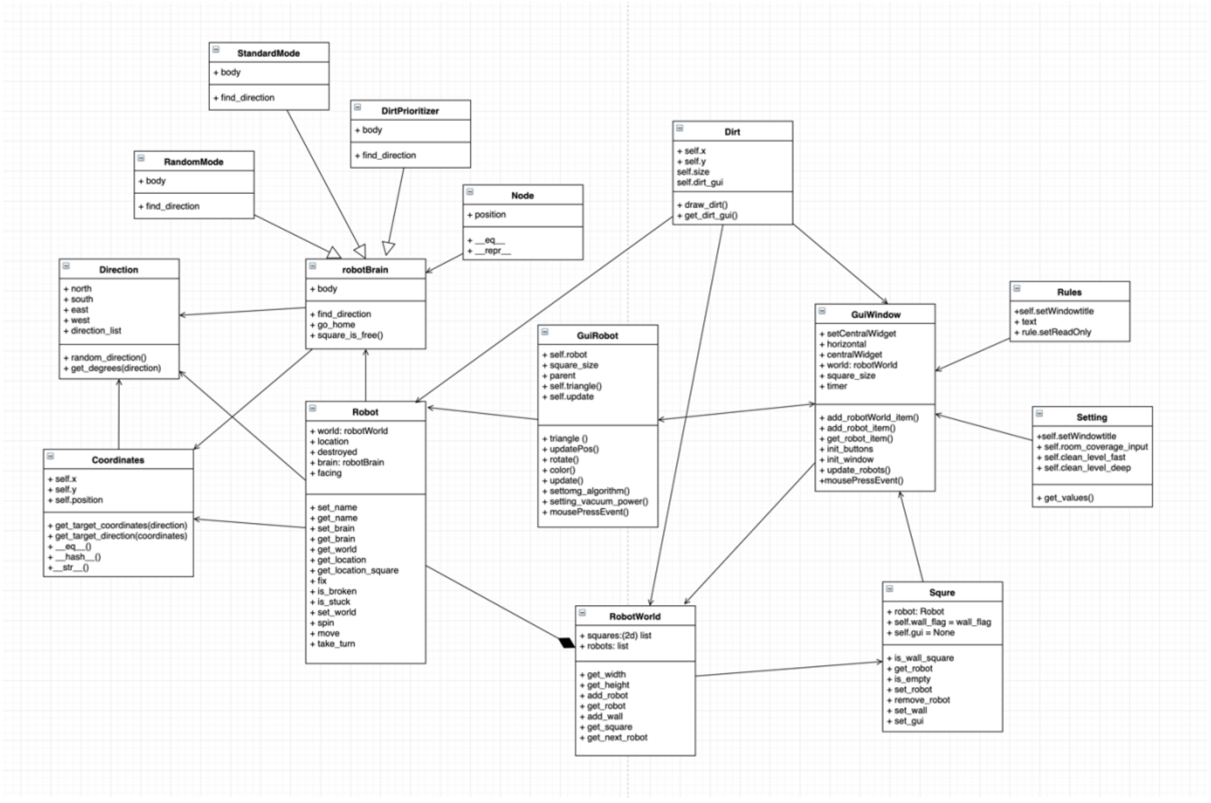
```
self.timer2.start(10)
```

Structure of the program

Like the UML diagram below, I increased several classes. Apart from the 5 classes I originally planned to have, I increased GuiRobot class to take charge of robot graphic items specifically. I also added classes like Direction, Coordinates to manage position, moving related objects. I increased Node class whose objects are initalized when robots mark spots they have visited. Nodes are used for finding the way back to docking station(initial position where robot was set up). Dirt class manages both drawing the dirts(graphic dirts items)and initializing logical dirts objects. Rules and Setting classes take charge of different type of QtWidgets which presenting rules to users and taking user inputs to set up cleaning targets.

The robots have their own coordination system, which is different from grid coordination system. For example, a robot can be placed anywhere on the grid with a location coordinates presenting its physical location on the grid. But it will also have a separate inner coordinates system which will always set the starting position to be (0,0).

The logic flow start with calling start_clean() of GuiWindow class. Then all the robots in the world will take their turns one by one. Every time when a robot takes its turn, it moves only 1 step. After that, the brain of the robot will determine the direction to go. Then move() method in Robot class will move the robot forward 1 step. Then, clean() method is called to remove dirts objects(both logical objects and graphical items). When cleaning targets are met, or when all robots in the world are destroyed, the cleaning task is terminated.



Algorithms

I have three algorithm classes: random mode, standard mode and dirt prioritizer. The idea behind is that, in random mode, the robot will choose a random direction to move forward. In standard mode, the robot will prioritize directions which lead to locations it has never been or visited less frequently. In dirt prioritizer, the robot will operate in the same logic as in standard mode, however, it will have one more feature – the robot can use sensor to detect how much dirt is left at current spot. If the current spot is already cleaned up(no dirt left), it will choose direction to move away. When there is still dirt left, it will stay there until it cleaned the current spot up. No matter in which mode, the robot will start to find its way back to the docking station when the battery level is low(100 out of 400).

The standard mode algorithm works in the way that, the robot saves coordinates it has visited into a dictionary, which means the robot knows how many times it has visited a certain location. When its time for the robot to take a turn, it will check the squares around it (north, south, west, east). Out of the free squares, it will select a target square to move into based on the visited times. The square with minimal visited times will be chosen. Direction to move into that target square will be given to the robot to take action – move() and clean().

The dirt prioritizer algorithm works exactly the same way as the standard mode algorithm which aims to reduce repeatitive movements. One thing extra is to check if the current spot has been cleaned up or not. The robot can retrieve the list of dirt objects at certain location. After cleaning, the robot knows how much dirt it removed and how much is still left there. If there is still leftover to be cleaned, the flag deep_clean will be set to True. The robot will only look for direction to move into when the flag is False. When its this robot's turn again, due to the flag deep_clean being true, it will simply call clean() method again instead of finding new moving direction.

The path-finding algorithm which gets the robot home when battery is low, works in a simiilar fasion as A* algorithm. The robot makes nodes at spots around it that are free to move into. The node object will contain h,g and f values. The h value presents the distance from the target spot to the docking station (0,0). If the target spot is (a,b), then h = a + b. While g value is set to be 0 by default. Every time when the robot visit that node, the g value of that node will be increased by 10. The f value is simply the sum of g and h : f = g + h. Unlike A* algorithm which creates new nodes while examing child nodes, go_home() algorithm will not create new nodes if there is an exisiting node at certain spot. In this way, the node can accurately reocord how many times it has been visited, and can be used to stop the robot repeatitively choosing the wrong path. Since every repeatitive visit will increase g value greatly (by 10), while h value(the distance to docking station) stay always the same, the robot will choose the adjacent node with minimal f value.

I have tried to use A* algorithm directly to find the path home, but A* works only on a logical way to find the optimal path. The fact that the robot has to move to detect if there are obstacles around makes it hard to implement A*. When A* algorithm chooses the minimal F value node to be the next current node, the robot cannot jump directly to that node. I tried to let the robot move backwards based on parent nodes locations, but something was wrong and did not work out. Therefore, instead of finding the global optimal path, I changed to find only the local optimal out of the adjacent nodes.

Data structures

I used list, dictionary and tuple to store data. Tuple is immutable while list and dictionary are mutable. I used tuple to store visited square objects as I expect no duplicated element. Tuple allows me to filter out duplicate ones nicely. List is used to store for example dirctions [north, west, south, east], added robots in the world, dirt objects of a specific location, dirt objects added to the world etc. I use dictionary only when I need to store elements just like in list, but with values. For example, I use dictionary to store turning degrees for each direction, or visited times for certain coordinate objects etc.

Files

No extra file or setting is required to get the program to run. All you need is to run main.py. Just put all the modules in one folder and open the entire folder in Vscode for example.

Testing

I tested window set up, some buttons and button clicking. I have not covered everything I planned in the project plan due to time constraint. I started writing tests only when I finished writing the codes. I think I can do better next time by writing tests while writing project codes. Based on the tests I have done so far, there is no hole identified. All tests passed.

The known shortcomings and flaws in the program

The main shortcomings is that the program works only on a simple 2D grid world, and there are only 4 movable directions. My algorithms work in the ideal 2D world where adjacent obstacles and dirts objects on the floor can be all detected. In real world, the robot might not always get to detect obstacles presenting. It might get stuck on carpets, or thresholds that it does not regocnize as obstacles. Due to the layout, I cannot implement obstacle avoidance algorithems like Bugs either. I would need to make the grid size very small, and make the robot turn, for example, 10 degree per time. I would add features such as robot legs, such that the robot will use legs to across obstacles that it cannot pass.

Another inconvinienice I noticed is that when there are multiple robots operating, users cannot identify different robots with different algorithms. The visual indicator of the robot coverage are also identital to all robots. It will be a bit inconvenient if the user wants to compare the performance between robots.

When the obstacles set up gets really complicated, like a maze, the robot still repeatitively visit areas. The algorithm can be improved to visit only places it has not visited before, unless its not completely clean yet.

The app will also work nicer if there is a timer widget with start and pause buttons. So user can see the time taken by each robots in a very straightforward way.

3 best and 3 worst areas

2 areas needs improvement: user interface, algorithm
Areas well done: customizable map, interactive setting up process and back-home navigation.

I have mentioned how to fix the issues in last section.

Changes to the original plan

Most of the things I planed are implemented. I planned before to have a feature that the robot can detect live creatures and wait for it to leave, and then resume. I realized that the live creature is essentially just the same thing as any other obstacle. The robot is assumed to be able to detect obstacles in adjacent squares, and it will only move into squares that are not occupied. The obstacle is a live creature or just another robot make no difference to the robot.

I also planned to add trash bin feature such that the robot will empty the bin when it arrives home. I did not implement as it is a technically quite easy thing to do. I already implemented the feature to navigate its way back home when battery is low to recharge itself. Empty its trash bin is esentially the same thing.

I am pretty bad at plan the shedule as I have no idea how much time it will take. So basically I did not have any plan. I just work on the project as long as I have time. I started implementing fundamental classes such as Robot, Robot World, Coordinates, Directions etc. Then I started working on the GUI class to get the main window working. I first worked on the square graphic items. Then I worked on mouseEventPress of the square graphic items, which is very important as it is the critical method where obstacles, robots got initialized. At the same time I start to work on robot graphic items and improving the user interface design and functionality. Finally, I started working on algorithms.

Assessment of the final result

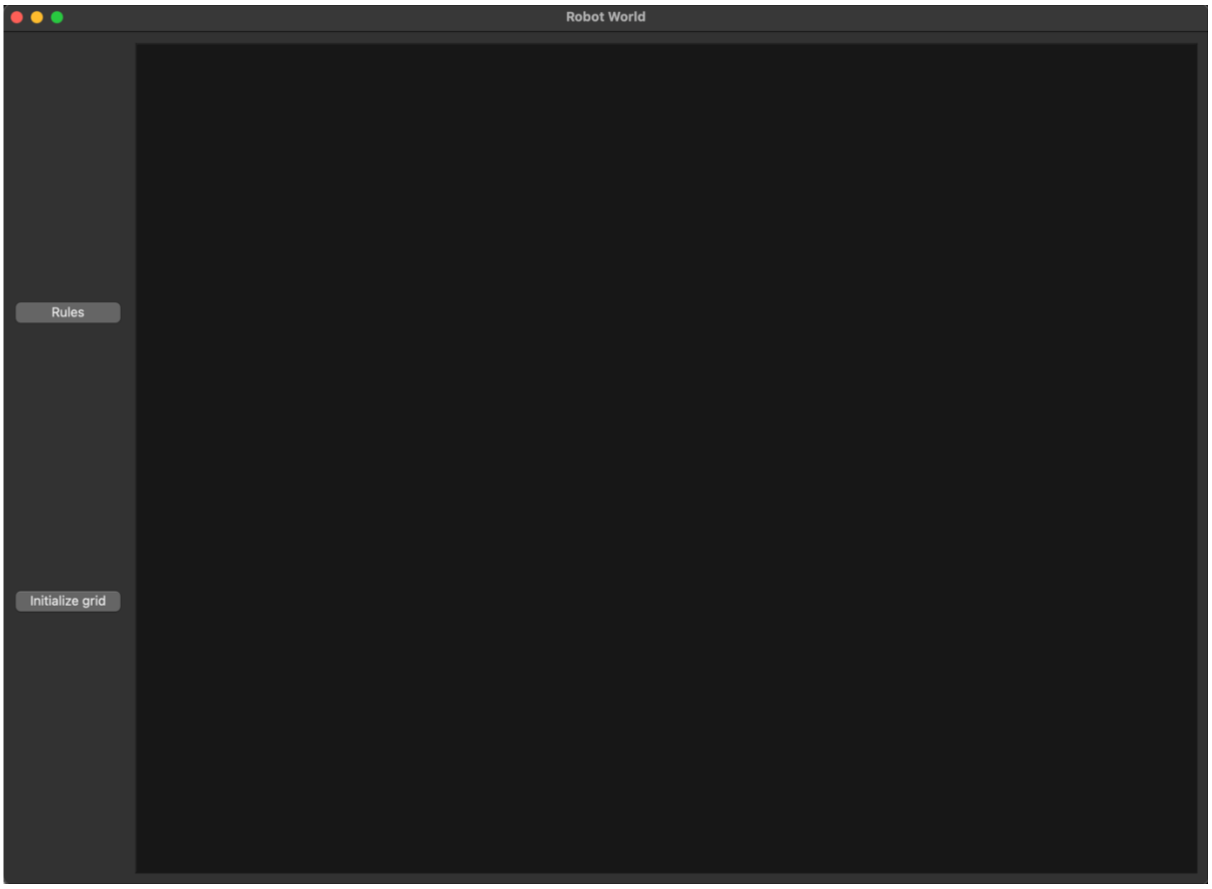
I am pleased with the overall result. The strong part is that I have implemented all the features I wanted to, including path-finding algorithm. The weak parts are the user interface and algorithm can be further improved. As I mentioned earlier, I can add stopwatch to track time taken by the robot, use different colors to differentiate robots and cleaning path, add a log widget which shows which robot accomplished which targets in how much time, to make the grid size very small to simulate robot movement in real world better, to implement Bug algorithm to follow edges of obstacles and walls, and to imrpove the current algorithm such that it do not visit the same spot twice unless its not cleaned completely yet. The main reason for not implementing these is because of time. If time allows I will do it. In terms of data strucutre, I think the ones I have been using work perfectly fine.

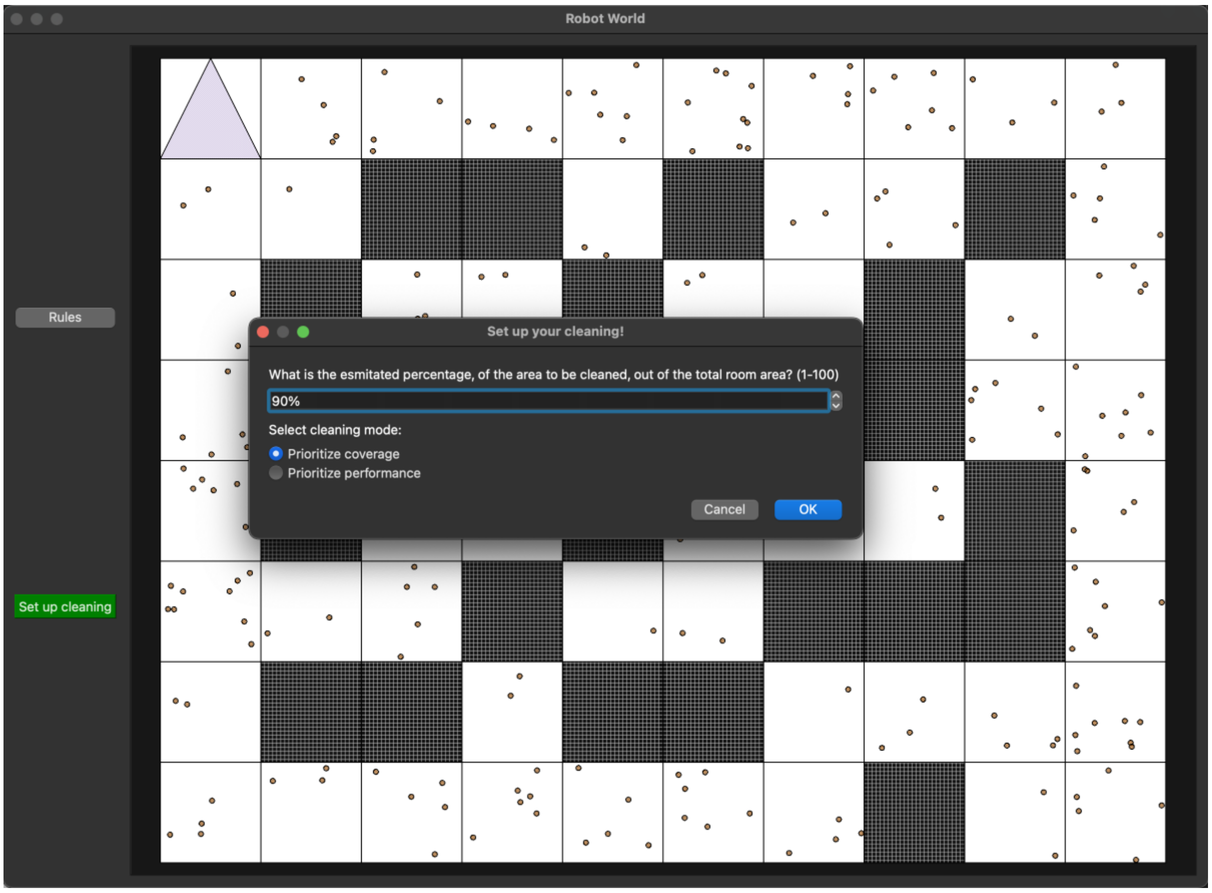
References

Swift.N(2017,Feb 28) , *Easy A* (astar) Pathfinding*, Medium

<https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2>

Attachments





Set up your cleaning!

What is the esmitated percentage, of the area to be cleaned, out of the total room area? (1-100)

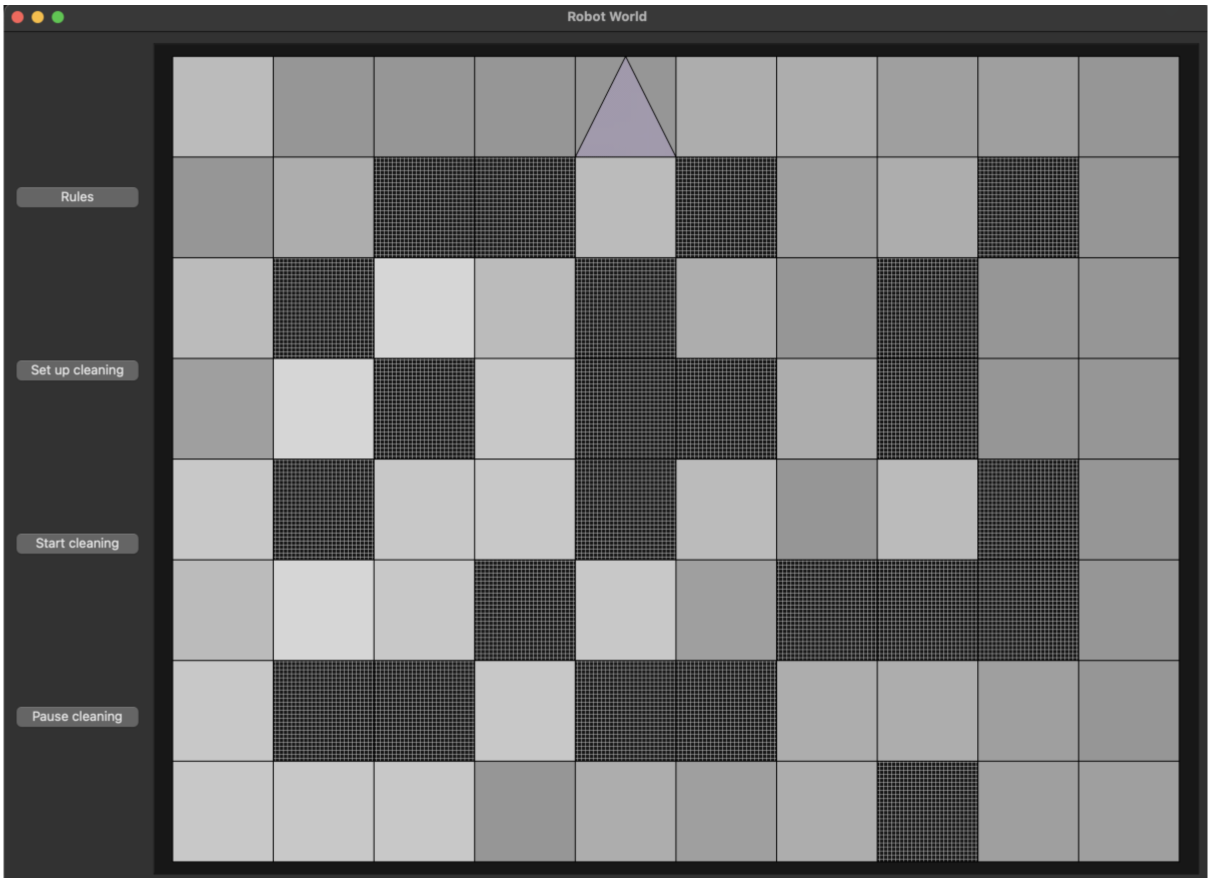
90%

Select cleaning mode:

[✓ Prioritize coverage]

Prioritize performance

Cancel OK


Robot World
Rules
Set up cleaning
Start cleaning
Pause cleaning