2D-AutoNaviRobot Simulation Results

School of Electrical Engineering and Computer Science

University of Ottawa

By:

Yitong Du

Oct. 2014

**Part I. General Idea**

The general idea of this simulation is to simulate a game scene from one of the game I made before.

To make the task easier to complete, I made some simplification, so the final version looks like this:

There is a 9 x 9 grid, every 0.5 second, one obstacle will spawn randomly in one of the non-destination grid (a grid with an obstacle in it is considered as impassable), and there is also a robot always seeks for the shortest path to the destination node. (In this case, the destination node is fixed.)

The system is consist of two Models:

**Sensor Model:** A sensor to “sense” the variation of the grid (to make the task simple, the variation of the grid is generated by a random function built within Sensor Model instead of using a real sensor), and send the variation (can be null, which means no variation) to Robot Model every 0.5 second.

Sensor Model has 2 states: Idle and SendVariation.

**Robot Model:** It firstly receive the message from Sensor Model, then it will update its “map”, and move to a random grid next to it (up, down, left or right), going nowhere if all adjacent grids is impassable." (It only moves one step, and after that, it stops, waiting for the next “map” update.

Robot Model has 2 states: Idle and SendLocation.

**Part II. Model Specifications**

I use a 9 x 9 grid to represent the “terrain”, the coordinate starts from the left bottom (0, 0), and end with the most right top grid (8, 8).

**Sensor Model** uses a 9 x 9 2D-Array to memories the states of the 9 x 9 grid. Two and only two values can be assigned to each element of the 2D-Array: “1” represents state “impassable” and “0” represents state “passable”. There is an exception:

The current location of the robot. (This info is received from **Robot Model.**)

Every one second, Sensor Model randomly generate 1 non-obstacle grids (value “0”) by assigning “1” to them. The “exception grid” mentioned above won’t be taken into consideration when generating “obstacles” (won’t assign 1 to that grid). Figure 1 shows a sample for one possible variation of the grids.

After that, it send the variation message (in this case, the message is 65.0) to **Robot Model**, which is the robot controller.

1. Meaning of variables in **Sensor Model**:
2. *int grid[9][9];* This is the array where the state of every grid is stored, “1” stands for impassable, while “0” stands for passable.
3. *int robotLocation;* This is a integer variable used to store the coordinate of the robot received from **Robot Model**, it works like this:

The tens digit represents for the x-coordinate while the units digit represents for the y-coordinate, for example, figure “37” represents for grid[3][7]. The idea of using one figure instead of two figures (x-coordinate and y-coordinate) is to reduce the communication cost between two Models.

1. *double variation;* This is a double variable used to store the variation of the grid that has been changed. The idea is similar to that of “robotLocation”.
2. *int gridCount;* This variable is used to calculate the number of impassable grids.
3. *int spwanNum; int spwanCount;* I assign rand() % (9 \* 9 - gridCount - 1) + 1 to spwanNum, and this number represents the value of the “spwanNum”th passable and non-occupied-by-robot grid will be turned into “1”. The idea of using these two variables is to avoid the CPU waste in the traditional “*do{generate a random grid}while(grid is occupied)”* loop when the majority of the grids is impassable.
4. Meaning of Functions in **Sensor Model**:
5. *initFunction()* It initialise the grid[9][9] with the value {0}, and set state to idle, then calls *passivate()*. This allows the Model to wait until an input is detected.
6. *externalFunction()* This function assigns message value to *robotLocation*, and set state to sendVariation.
7. *outputFunction()* This function calculates the variation message and send it to output.
8. *internalFunction()* This function changes the state back to idle.

**Robot Model** receives the variation message, it updates its own 9 x 9 2D-Array to make it synchronized with that from **Sensor Model.**

After the synchronization, the robot move to a random grid next to it (up, down, left or right), going nowhere if all adjacent grids is impassable." (It only moves one step, and after that, it stops, waiting for the next “map” update. (Figure 2 shows a possible scenario.)

After the algorithm is called, there will be two possible situations:

1. Path is blocked: Our robotwill stay in the current location doing nothing.
2. A possible move is found: The robot will move one block according to the *nextMove* variable generated this turn, it then send back its current location to **Sensor Model**, and wait until the next message comes.

According to the rules above, **Sensor Model** and **Robot Model** will keep transmitting messages until there is no way to the destination or the robot has reached the destination.

(8, 8)

(0, 8)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | 1 |  |  |  |
|  |  |  | 1 |  |  |  | 1 |  |
|  |  |  |  |  |  | **Robot** |  |  |
|  |  |  |  |  |  | 0->1 |  |  |
|  |  |  |  |  |  |  |  |  |
| 1 | 1 |  |  |  |  |  |  |  |
|  |  |  | 1 |  |  |  | 1 |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

(0, 0)

(8, 0)

*Figure 1*

The robot will ignore this cell when the adjacent cell has a state value of “1”.

The robot can move to this cell when the adjacent cell has a state value of “0”.

|  |  |  |
| --- | --- | --- |
| Not directly accessible | 1 | Not directly accessible |
| 0 | Robot’s  Location | 0 |
| Not directly accessible | 0 | Not directly accessible |

*Figure 2*

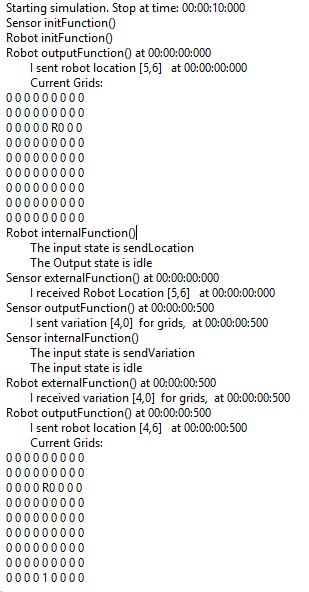
1. Meaning of variables in **Robot Model**:
2. *int grid[9][9];* This is the array where the state of every grid is stored, “1” stands for impassable, while “0” stands for passable. It is used to synchronize those in **Sensor Model**.
3. *int robotLocation;* *double variation;* These idea of using these two variables are as same as those in the **Sensor Model**.
4. *int attempt;* This random number attempts to determine which direction to go to.
5. *int nextMove;* If *“grid[tens digits of nextMove][units digits of nextMove]”* is occupied, the system will go back for another attempt.
6. Meaning of Functions in **Robot Model**:
7. *initFunction()* It initialise the grid[9][9] with the value {0}, Model is set active immediately.
8. *externalFunction()* This function assigns message value to *variation*, and set state to sendLocation.
9. *outputFunction()* This function calculates a possible move and send the new location message to output.
10. *internalFunction()* This function changes the state back to idle.

**Part III. Simulation Result**

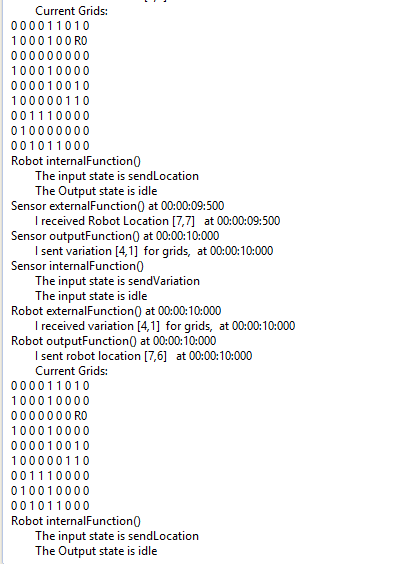
Figure 3 shows the initialization of the simulation and the first round of communication of the two models.

Figure 4 is a screenshot of communication between two models in the 9.5th s and 10th s.

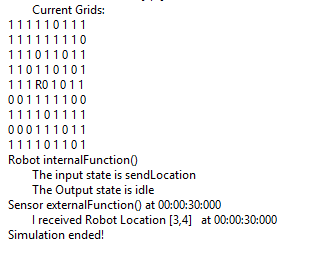
Figure 5 is a complete simulation of a 30 second period.



*Figure 3*

**

*Figure 4*



*Figure 5*

**Part IV\* Future Work**

Due to limit time, I didn't completely implement my original idea (which is to use Dijkstra’s Algorithm to find the shortest path to a fixed destination). My future work will be applying the Dijkstra’s Algorithm to the Robot model.