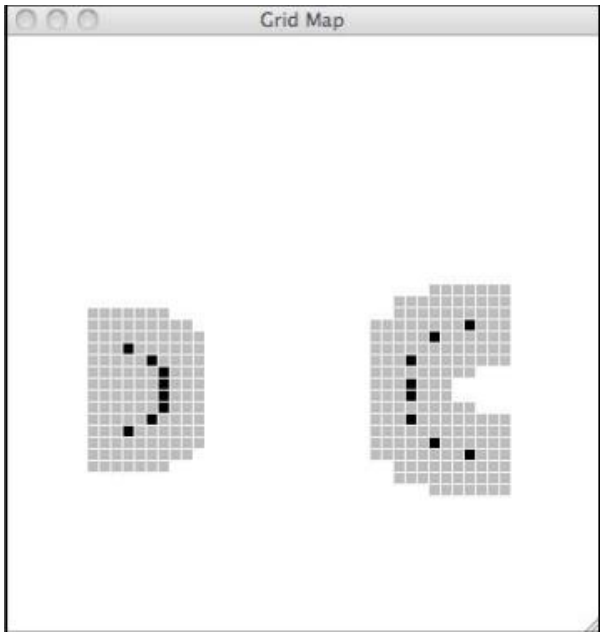
1. Mapmaker, mapmaker, make me a map

Step1

Check Yourself 1. We have defined a `SensorInput` class for testing in idle that simulates the `io.SensorInput` class in soar. Consider these two possible sensor input instances (each has a list of 8 real-valued sonar readings and a pose).

```
testData = [SensorInput([0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2],
                        util.Pose(1.0, 2.0, 0.0)),
            SensorInput([0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4, 0.4],
                        util.Pose(4.0, 2.0, -math.pi))]
```

Be sure you understand why they give rise to the map shown below. Remember that the black squares are the only ones that are marked as occupied as a result of the sonar readings; the gray squares are the places that the robot cannot occupy (because it would collide with one of the black locations).



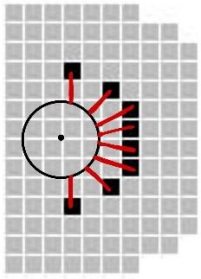
Explanation for the map above:

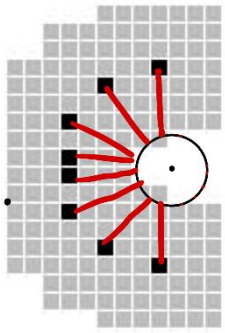
The testData is a list which contains two instances of *SensorInput*.

Every instance has two arguments.The first argument represent the readings from robot's 8 sonars,and the second argument represent the pose of the robot.

Therefore, we can see that there are two parts of gray squares and black squres in this map which correspond to the two inputs.

The black squares: the only ones that are marked as occupied as a result of sonar reading.





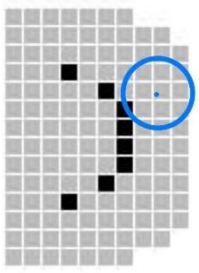
Refer to the documents:

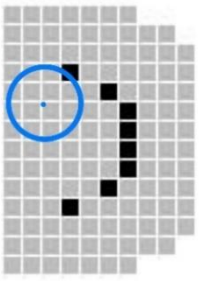
robotRadius=0.2m, grid.xStep=grid.yStep=0.1m

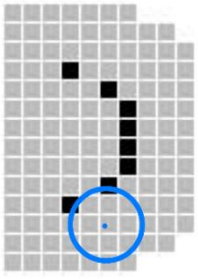
util.Pose(1.0,2.0,0) : It indicates that the x coordinate of the robot in the map is 1.0m, y coordinate is 2.0m, and the Angle between the head and the positive direction of x is 0 (the head of the robot is horizontal to the right).

util.Pose(4.0,2.0,0) : It indicates that the x coordinate of the robot in the map is 4.0m, the y coordinate is 2.0m, and the Angle between the head and the positive direction of x is pi (the head of the robot is horizontal to the left).

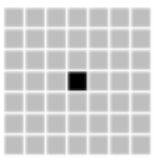
The gray squares: When the center of the robot moves arbitrarily in a square, if the obstacle encountered, the square will be gray (the robot cannot occupy).





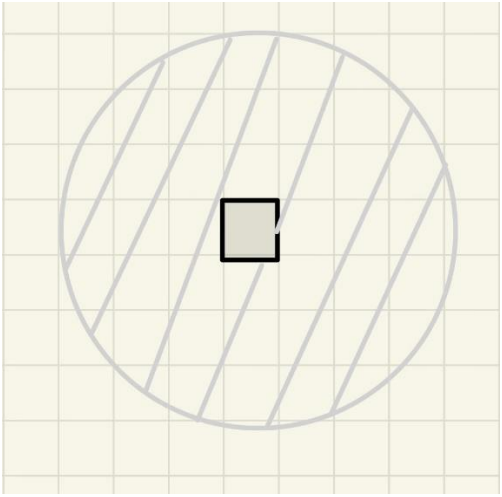


We start with a single obstacle to explain the rules of the gray grid:



Result1

In this basic map, each grid has a side length of 0.1 m. In a circle with a radius of 0.2 m, our ideal situation is like this:



This is a circle with a radius of 0.3 meters

Obviously, we are coloring the whole grid, the above situation is not possible(We don't use 0.2m as the radius because in extreme cases, the robot will be tangent to the obstacle, which will be dangerous)

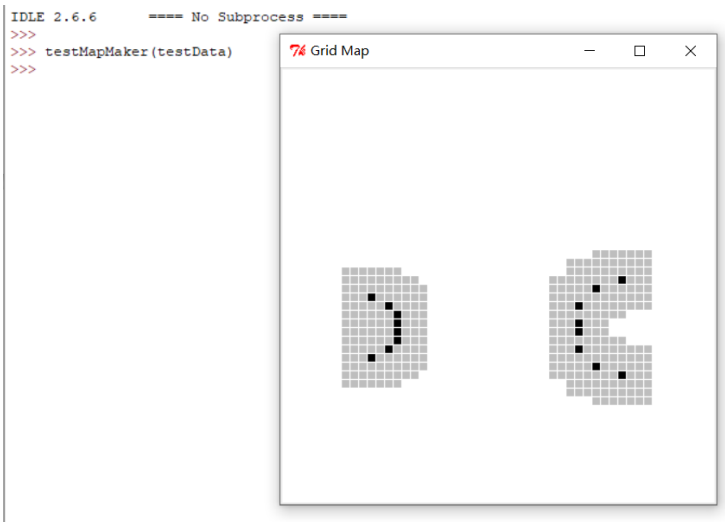
We further study squareColor function. Whether the gray color is colored or not depends on the result of robotCanOccupy. In the document, we understand that this is a function considering "a range of boxes" around spcified box. This is the smallest cube under the condition that the robot does not collide with obstacles, as shown in Result1

We can now say that the gray grid on the map is a superposition of 7*7 grids centered around obstacles (black grids)

Step2 Implement the *MapMaker* class

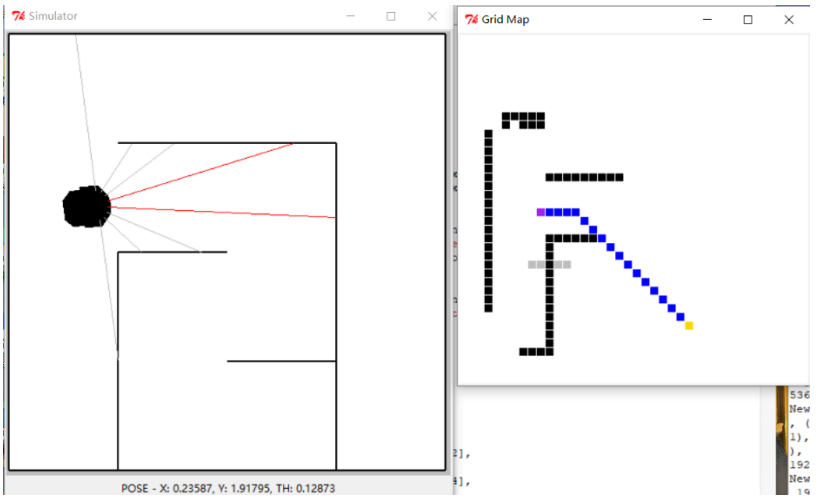
```
class MapMaker(sm.SM):
    def __init__(self, xMin=False, xMax=False, yMin=False, yMax=False, gridSquareSize=False):
         self.startState = dynamicGridMap.DynamicGridMap(xMin,xMax,yMin,yMax,gridSquareSize) # change this
    def getNextValues(self, state, inp):
        for i in range(8):
            start=state.pointToIndices(sonarDist.sonarHit(0,sonarDist.sonarPoses[i],inp.odometry))
            #the location of the sonars in the robot(It has been converted into the indices of the grid)
            very_end=state.pointToIndices(sonarDist.sonarHit(sonarDist.sonarMax,sonarDist.sonarPoses[i],inp.odometry))
            #the Mapmaker knows that the grid cell at the very end of a sonar ray is occupied by an obstacle.
            if inp.sonars[i]<sonarDist.sonarMax:
                item=state.pointToIndices(sonarDist.sonarHit(inp.sonars[i],sonarDist.sonarPoses[i],inp.odometry))
                #the location of the item detected by sonars(It has been converted into the indices of the grid)
            state.setCell(item)
        return (state,state)
```

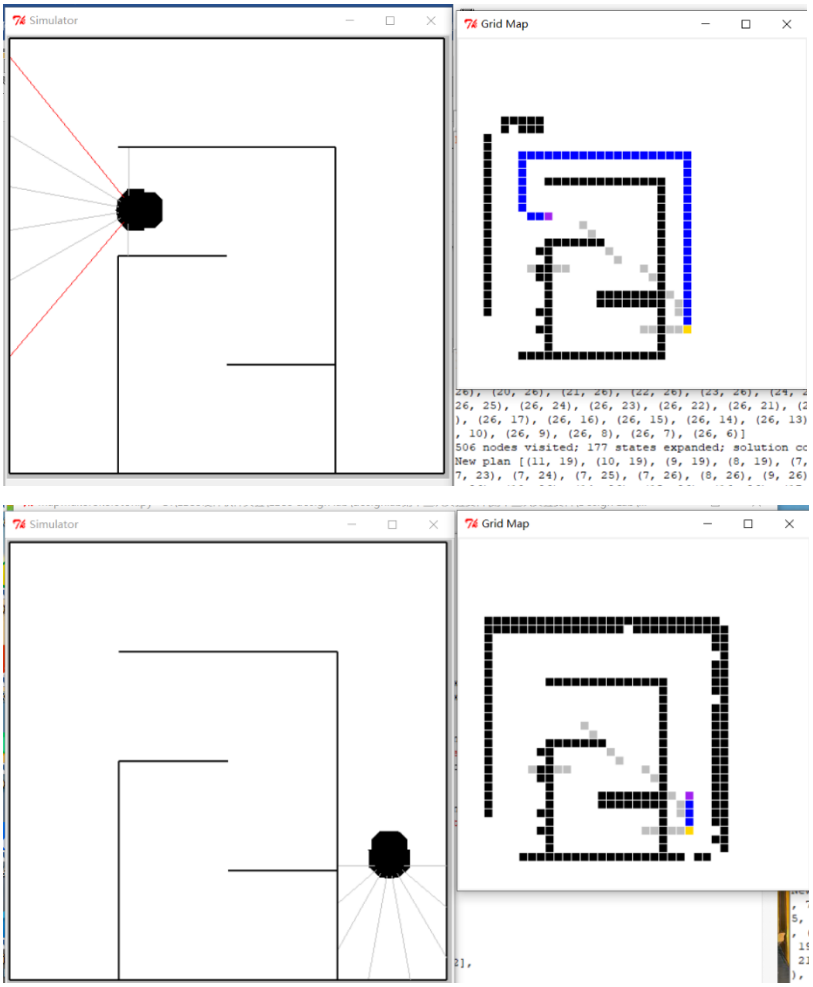
Step3 Test your map maker



The result of the test of our map maker is the same as the figure shown in dl14.

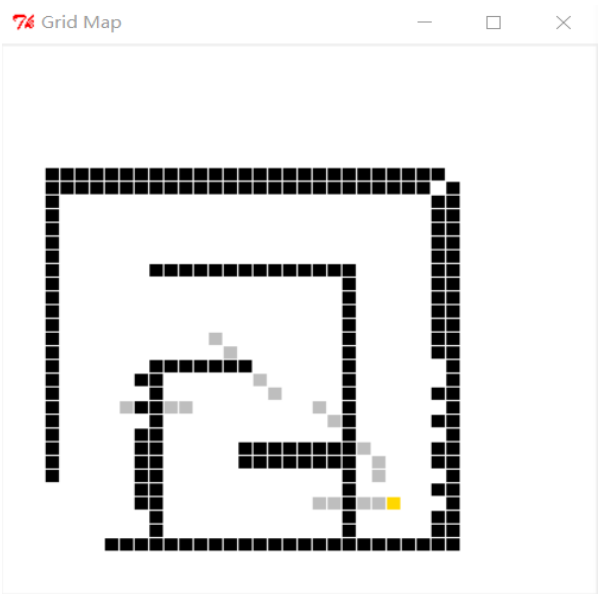
Step4 Test in soar





Checkoff 1. **Wk.14.2.1**: Show the map that your mapmaker builds to a staff member. If it does anything surprising, explain why. How does the dynamically updated map interact with the planning and replanning process?

Final map:



How does the dynamically updated map interact with the planning and replanning process?

At first, the planning process will plan a ‘way’ to the goal according to the information it has known, but this ‘way’ is usually ‘wrong’ (that means in this way, the robot has a probability to collide into a obstacle) With the robot moving, it will update the dynamical map because it get some new information(by observation model) about the environment around it. If the new information tell the truth that the old way is not reliable, the replanning process begin to work, it will make the planning process plan a new way to reach the goal.

3 A noisy noise annoys an oyster

Step 5

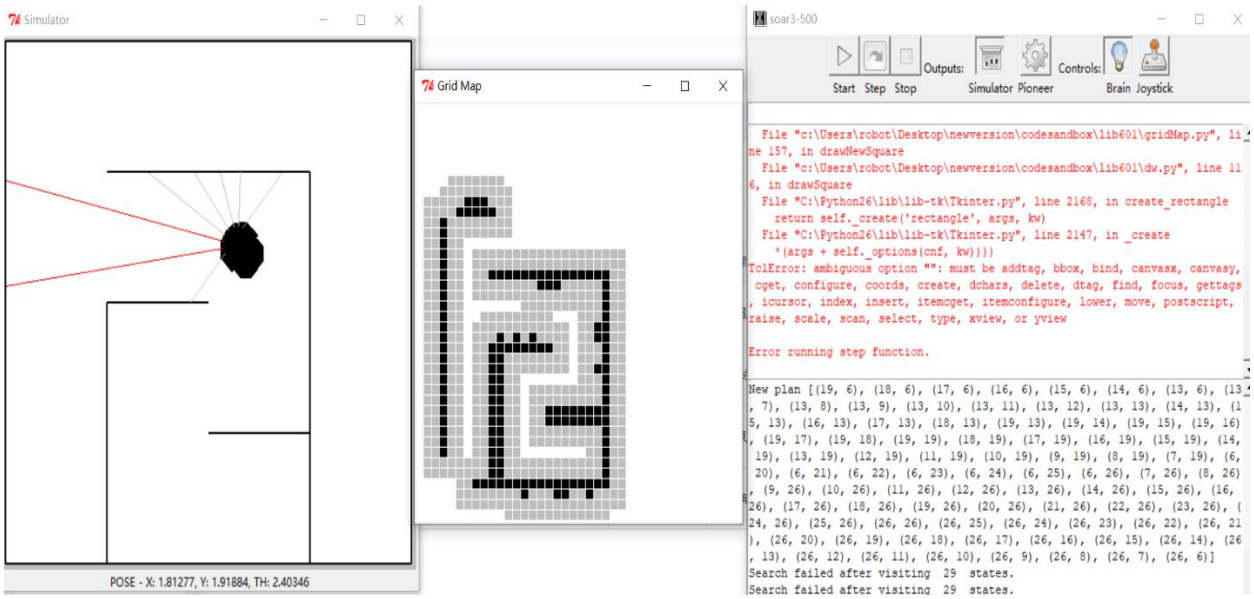
```
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: noNoise
```

and change it to one of

```
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: smallNoise
soar.outputs.simulator.SONAR_VARIANCE = lambda mean: mediumNoise
```

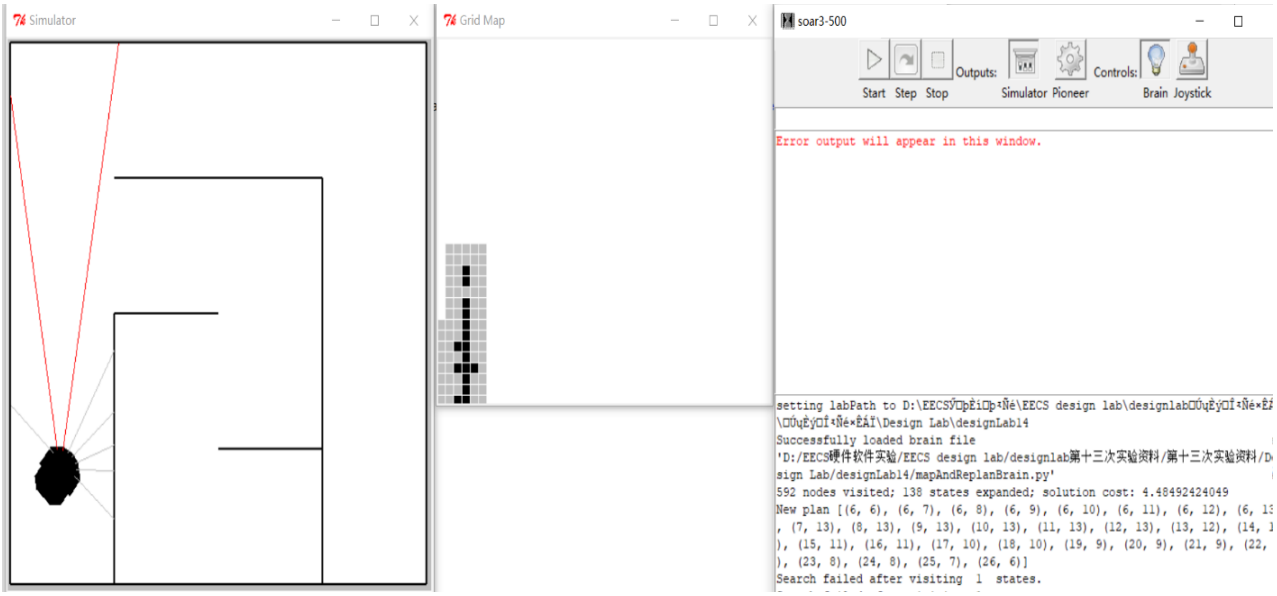
Check Yourself 2. Run the brain again in these noisier worlds. Why doesn't it work? How does the noise in the sensor readings affect its performance?

SmallNoise:



Search failed after visiting 29 states.

mediumNoise:



Explanation:

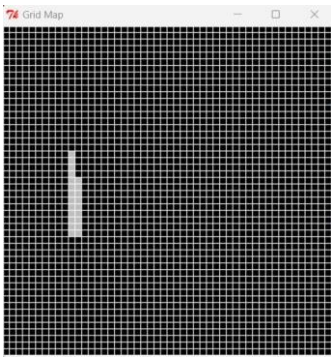
At this time, our sonar is not 100% reliable, which means that compared with the no noise, there is a "very large" probability of marking "occur" in the unobstructible places. Because of the huge size of the robot, a large area will be incorrectly marked as unreachable by the robot after marking a wrong obstacle. Then recall ReplannerWithDynamicMap method for route planning. It is to be expected that, on the one hand, this is definitely not the shortest (or least expensive) path compared to the correct path, on the other hand, there may be no solution.

Step6 Improve your *MapMaker* class

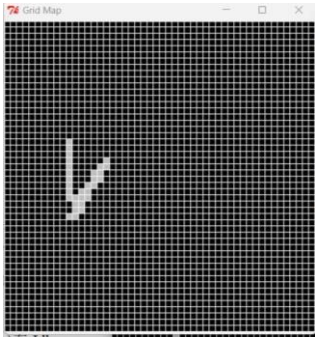
```
class MapMaker(sm.SM):
    def __init__(self, xMin=False, xMax=False, yMin=False, yMax=False, gridSquareSize=False):
         self.startState = dynamicGridMap.DynamicGridMap(xMin,xMax,yMin,yMax,gridSquareSize) # change this
    def getNextValues(self, state, inp):
        for i in range(8):
            start=state.pointToIndices(sonarDist.sonarHit(0,sonarDist.sonarPoses[i],inp.odometry))
            #the location of the sonars in the robot(It has been converted into the indices of the grid)
            if inp.sonars[i]<sonarDist.sonarMax:
                item=state.pointToIndices(sonarDist.sonarHit(inp.sonars[i],sonarDist.sonarPoses[i],inp.odometry))
                # the location of the item detected by sonars(It has been converted into the indices of the grid)
                state.setCell(item)
                # mark this grid which is occupied
            else:
                item=state.pointToIndices(sonarDist.sonarHit(sonarDist.sonarMax,sonarDist.sonarPoses[i],inp.odometry))
                # when the sonar reading is greater than the maximum good value\
                # we make the cells along the first part of the ray clear.
            clearlist=util.lineIndices(start,item)
            for i in range(len(clearlist)-1):
                state.clearCell(clearlist[i])
            # We think of these cells as the set of grid locations that could reasonably be marked as being clear,
            # based on a sonar measurement.
        return (state,state)
```

We regard the sonar value of the obstacle successfully marked as good. We can further use the gap determined between the sonar and the obstacle to mark the grid accessible to the robot for clearCell method. For the maximum accuracy of the program, we will also process the sonar value beyond the maximum sonar reading (sonarDist. sonarMax) This is necessary, and we will give reasons why:

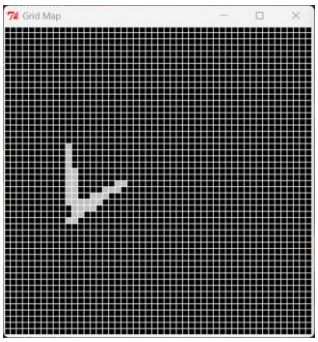
If we consider a single sonar reading, which is a sonar message that detects an obstacle, we generate a map：



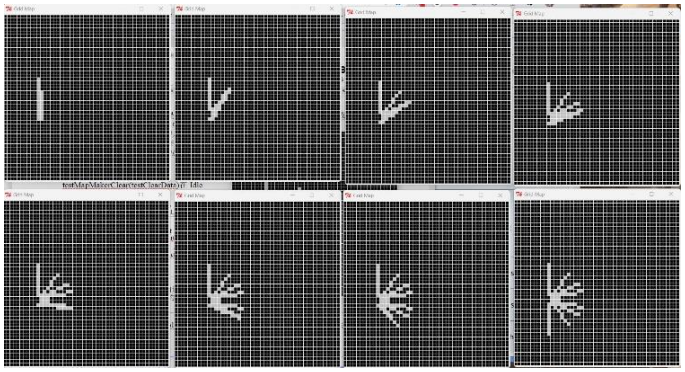
We found that there was a large error, and we further considered two adjacent sonars that together generated the map:



Now we see that the image generated by the first sonar is stable, so in order to generate the correct path, should each sonar generate a map with its neighbor?



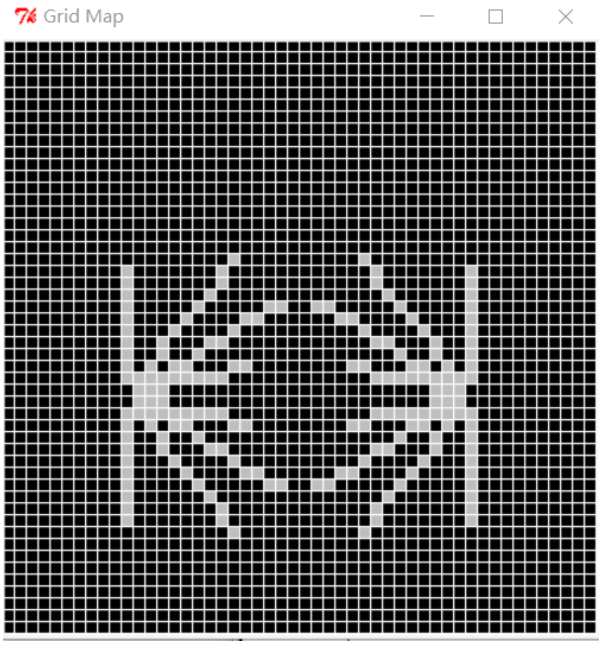
We use two uncontiguous sonar readings and find that neither is accurate, so we can say that our updates to the map must be applied to all sonar readings



This will be our strategy:

For the reading of marked obstacles, we ‘set’ the end of the sonar, and the gap between them is' clear '. For the sonar without detecting obstacles, we consider it passable within 1.5 meters, so all the grids in the first 1.5 meters of the sonar are 'clear'.

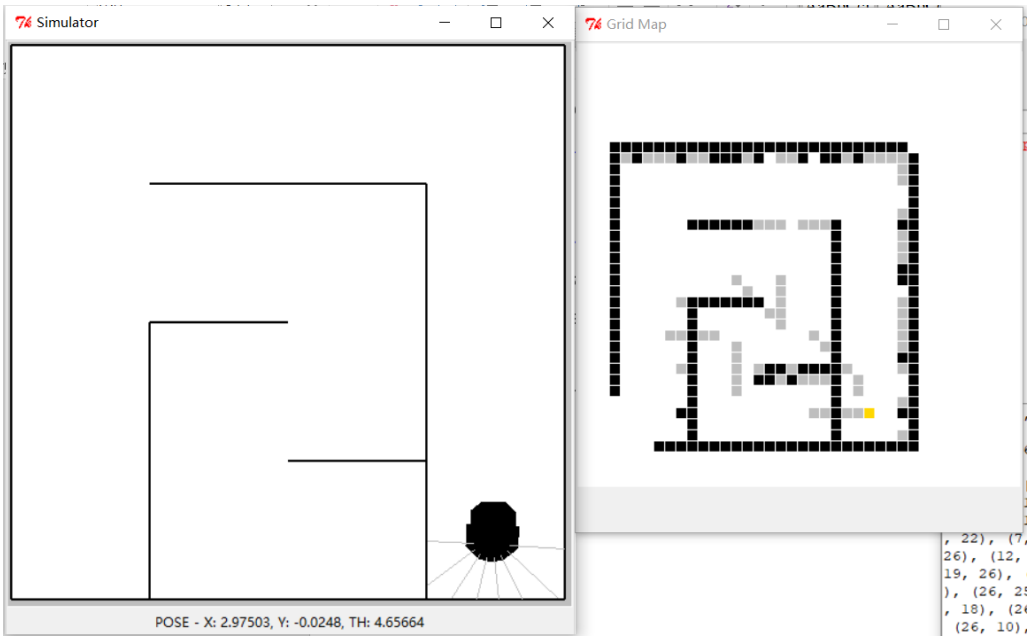
Step7 Test your new MapMaker in Idle



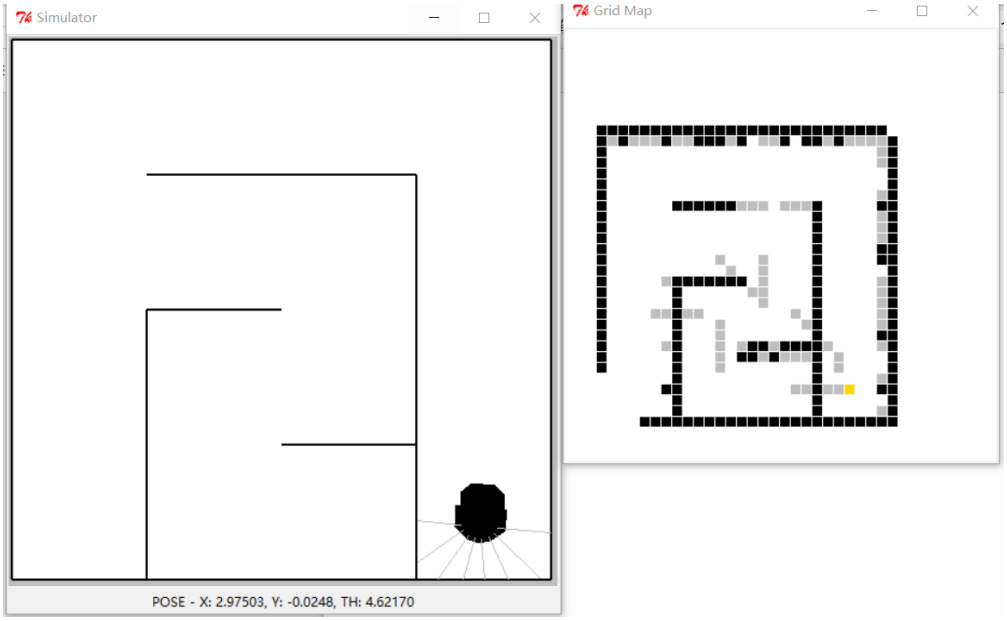
Step 8 Test in soar

Checkoff 2. **Wk.14.2.2**: Show your new map maker running, first with no noise and then with medium noise. We don't necessarily expect it to work reliably: but you should explain what it's doing and why.

no-noise:



medium-noise:



The two figures above are quite similar.

About our new map maker:This reduces the number of times the program has to re-plan the path and improves stability to some extent, but in soar simulations we have found that the grid we have marked as an obstacle may be 'clear' at other times, which helps to eliminate the error caused by noise. But on the other hand, it can also have a negative impact on our path planning.

4. Bayes Map

> *Check Yourself 4.* Remember that the sonar beams can sometimes bounce off of obstacles and not return to the sensor, and that when we say a square is clear, we say that it has nothing anywhere in it. What do you think the likelihood is that we observe a cell to be free when it is really occupied? That we observe it as a hit when it is really not occupied? What should the prior (starting) probabilities be that any particular cell is occupied?
> Decide on possible values for the state of the cell. Assume that the observation can be either `'hit'`, if there is a sonar hit in the cell or `'free'` if the sonar passes through the cell. **To forestall confusion, pick names for the internal states that are neither `'hit'` nor `'free'`.**
> If you are having trouble formulating the starting distribution, observation and transition models for the state estimator, talk to a staff member.

We assume that the likelihood that we observe a cell to be free when it is really occupied is 0.1 and the likelihood that we observe a cell as a hit when it's really not occupied is also 0.1.

Step9 Create an instance of ssm.StochasticSM that models the beha vior of a single grid cell.

```
# Observation model:  P(obs | state)
def oGivenS(s):
    if s=='occupied':
        return dist.DDist({'hit':0.9,'free':0.1})
    elif s=='not_occupied':
        return dist.DDist({'hit':0.1,'free':0.9})
    else:
        return None
# Transition model: P(newState | s | a)
def uGivenAS(a):
    def nsGivenS(s):
        if s=='occupied':
            return dist.DDist({'occupied':1.0,'not_occupied':0.0})
        elif s=='not_occupied':
            return dist.DDist({'occupied':0.0,'not_occupied':1.0})
        else:
            return None
    return nsGivenS

startDistribution=dist.DDist({'occupied':0.5,'not_occupied':0.5})
cellSSM = ssm.StochasticSM(startDistribution,uGivenAS,oGivenS)  # Your code here
```

Step10 Test your grid cell model

mostlyHits=[('hit', None), ('hit', None), ('hit', None), ('free', None)]

```
>>> testCellDynamics(cellSSM, mostlyHits)
[DDist(not_occupied: 0.100000, occupied: 0.900000), DDist(not_occupied: 0.012195, occupied: 0.987805),
DDist(not_occupied: 0.001370, occupied: 0.998630), DDist(not_occupied: 0.012195, occupied: 0.987805)]
```

mostlyFree=[('free', None), ('free', None), ('free', None), ('hit', None)]

```
>>> testCellDynamics(cellSSM, mostlyFree)
[DDist(not_occupied: 0.900000, occupied: 0.100000), DDist(not_occupied: 0.987805, occupied: 0.012195),
DDist(not_occupied: 0.998630, occupied: 0.001370), DDist(not_occupied: 0.987805, occupied: 0.012195)]
```

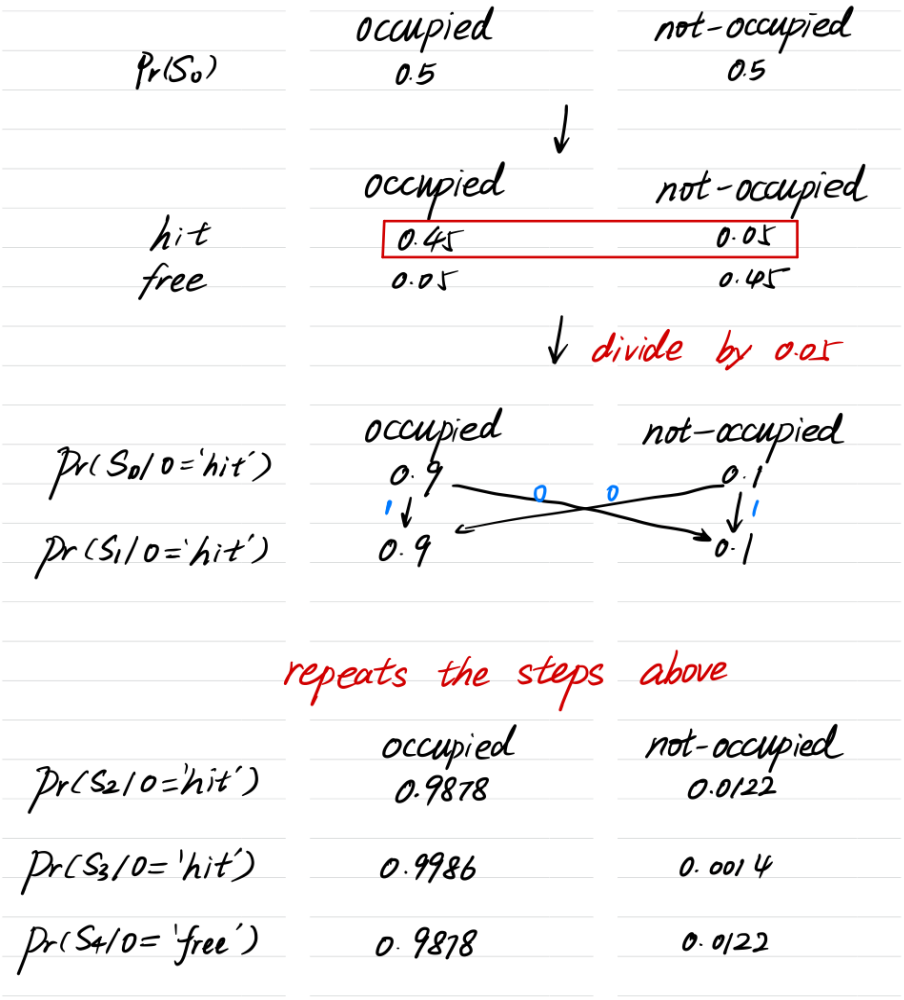
Question: Why are the Nones here?

In this model, instead of having one state estimation problem with 2^{400} states, we have 400 state estimation problems, each of which has 2 states (the grid cell can either be occupied or not). So,for one single grid which is a state estimator,we don't need to consider the action.That's because the action has no influence on the state distribution for one single state estimator,

The input (o, a), where o is an observation and a is an action; we will be, effectively, ignoring the action parameter in this model, you can simply pass in None for a.

Explannation for the resulf of our state estimator:

Take the input:*mostlyhit* for an example:



Step11

Wk.14.2.3	Solve this tutor problem on making collections of object instances.

Problem Wk.14.2.3: Aliasing Instances

This problem examines the problem of shared instances. We'll use the following simple class to illustrate.

```
class MyClass:
   def __init__(self, v):
       self.v = v
```

Part 1: Try 1

Consider the following code:

```
def lotsOfClass(n, v):
   one = MyClass(v)
   result = []
   for i in range(n):
       result.append(one)
   return result

class10 = lotsOfClass(10, 'oh')

class10[0].v = 'no'
```

1. What is the value of `class10[0].v`: no
2. What is the value of `class10[3].v`: no

Part 2: Try 2

Define a new version of lotsOfClass that has separate instances of the objects in each location of the list.

```
def lotsOfClass2(n,v):
    result=[]
    for i in range(n):
        result.append(MyClass(v))
    return result
```

Part 3: Try 3

Define another version of lotsOfClass that has separate instances of the objects in each location of the list. Use `util.makeVectorFill` (see Software Documentation) to accomplish the same thing.

```
def lotsOfClass3(n,v):
    def f(n):
        return MyClass(v)
    return util.makeVectorFill(n,f)
```

Step12 Implement the *BayesGridMap* class

1.the *makeStartingGrid* method:

```
def makeStartingGrid(self):
    def f(x,y):
        return seFast.StateEstimator(cellSSM)
    startGrid=util.make2DArrayFill(self.xN,self.yN,f)
    for i in range(self.xN):
        for k in range(self.yN):
            startGrid[i][k].start()
    return startGrid
```

Define a funtion called *f,* it accepts two arguments and returns an intance of *seFast.StateEstimator*

```
cellSSM = ssm.StochasticSM(startDistribution,uGivenAS,oGivenS)
```

The intance of *seFast.StateEstimator* accept an intance of ssm.StochasticSM which provide a startDistribution, a obervation model and a transition model.

We use the *util.make2DarrayFill* method to make every cell an instance of seFast.StateEstimator. Besides, we should be sure to call the start method on each of the state-estimator state machines just after you create this grid.

2. the *setCell* method and *clearCell* method:

```
def setCell(self, (xIndex, yIndex)):
    SM=self.grid[xIndex][yIndex]
    SM.step(('hit',None))
    self.drawSquare((xIndex,yIndex))
def clearCell(self, (xIndex, yIndex)):
    SM=self.grid[xIndex][yIndex]
    SM.step(('free',None))
    self.drawSquare((xIndex,yIndex))
```

The two methods are used to manage the state update of the state estimator machines in each cell yourself. whenever we get evidence about the state of a cell, we have to call the *step* method of the estimator.

3.the *occProb* method

```
def occProb(self, (xIndex, yIndex)):
    SM=self.grid[xIndex][yIndex]
    occprob=SM.state.d['occupied']
    return occprob
```

SM.state is an instance of dist.DDist which represents the probability distribution of the state. SM.state.d is an "dictionary", so we can return the probability when the state is "occupied".

4.the *occupied* method:

```
def occupied(self, (xIndex, yIndex)):
    threshold=0.9
    if self.occProb((xIndex,yIndex))>=threshold:
        return True
    else:
        return False
```

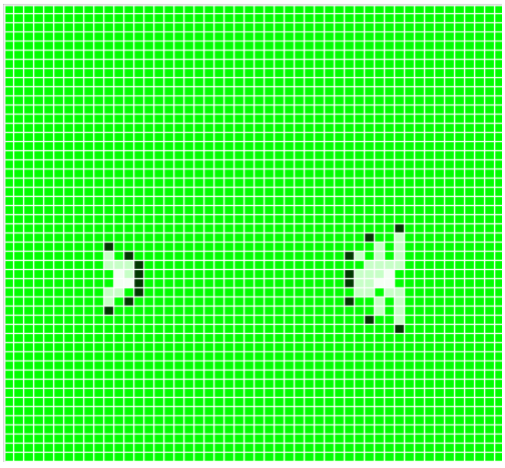
This method is used to detemine whether a cell should be marked as occupied. By experimenting with this method a lot, we find that a good threhold for our model is about 0.9(not more than 0.9)

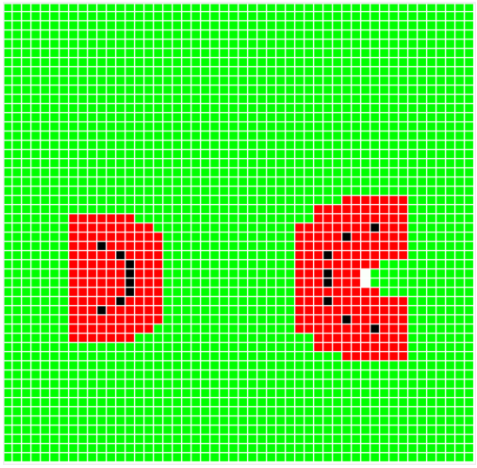
Step13 Test your code in IDLE

Check Yourself 5. Try it with two updates. Try it with testClearData. Be sure it all makes sense.

result:

When we specify the threshold to 0.95,by try it with one update and try it with two updates, we get two different map.





Explanation: We can see that the first map(one update) have red squares.Instead,there are some gray(close to white) squares in it. But in the second map(two updates), there are exactly some red squares which represents the squares that robot can't occupy. To explain this,we need to back to our BayesMap model.

In step10, we have test our model with certain testdate.And we got that:

	occupied	not-occupied
$Pr(S_0)$	0.5	0.5
$Pr(S_1 \mid o='hit')$	0.9	0.1
$Pr(S_2 \mid o='hit')$	0.9878	0.0122

When we update the state estimator machines in each cell for **one time**. The state of the cells which observed as 'hit' will have a probability distribution: "occupied":0.9 "not-occupied":0.1 . But our threshold is 0.95. For this reason, this cell will **not** be marked as occupied, it will execute the first judgment condition:

```
if self.robotCanOccupy((xIndex,yIndex)):
    return colors.probToMapColor(p, colors.greenHue)
elif self.occupied((xIndex, yIndex)):
    return 'black'
else:
    return 'red'
```

and the color of the cell will be drawn by its probability of "occupied". For these whose probability of "occupied" is near to 1(not more than 0.95),the color of them will be black.

In our model, the cells along the sonar ray are specified as "clear". Therefore, the observation of them are "free".

	occupied	not-occupied
$Pr(S_0)$	0.5	0.5
$Pr(S_1 \mid o='free')$	0.1	0.9
$Pr(S_2 \mid o='free')$	0.0122	0.9878

The state of the cells which observed as ‘free’ will have a probability distribution: ”occupied”:0.1 ”not-occupied”:0.9 .

For these whose probability of “occupied” is near to 0(not more than 0.95),the color of them will be gray(close to white).

However, in the second map, Things get different. That’s because when we update the state estimator machines for two times,

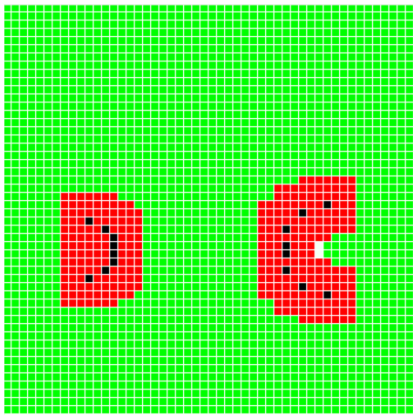
The state of the cells which observed as ‘hit’ will have a probability distribution: ”occupied”:0.9878 ”not-occupied”:0.0122

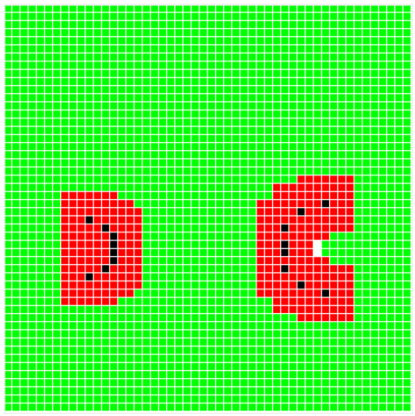
Our codes will execute the two conditions below

```
if self.robotCanOccupy((xIndex,yIndex)):
    return colors.probToMapColor(p, colors.greenHue)
elif self.occupied((xIndex, yIndex)):
    return 'black'
else:
    return 'red'
```

So, there will be some red squares which represents the squares that robot can’t occupy.

But if we specify the threshold to 0.90.The two maps will be very similar.

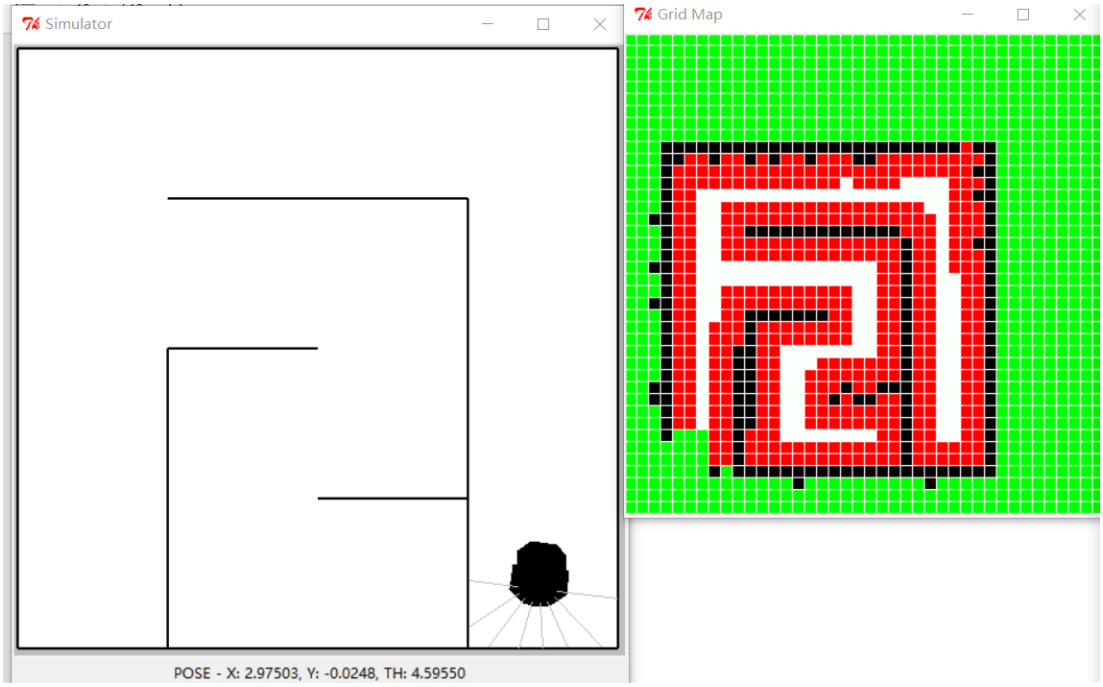




In conclusion, a suitable threshold is close to 0.90(no more than 0.90)

Step16 test in soar

1.Small noise:

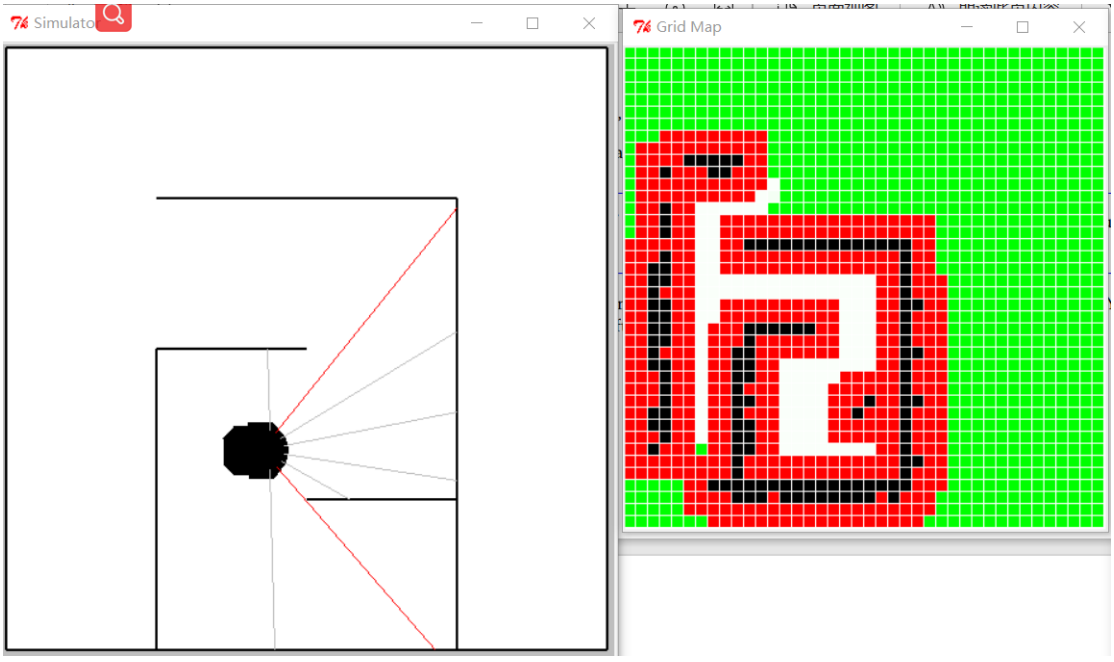


2.Medium noise:

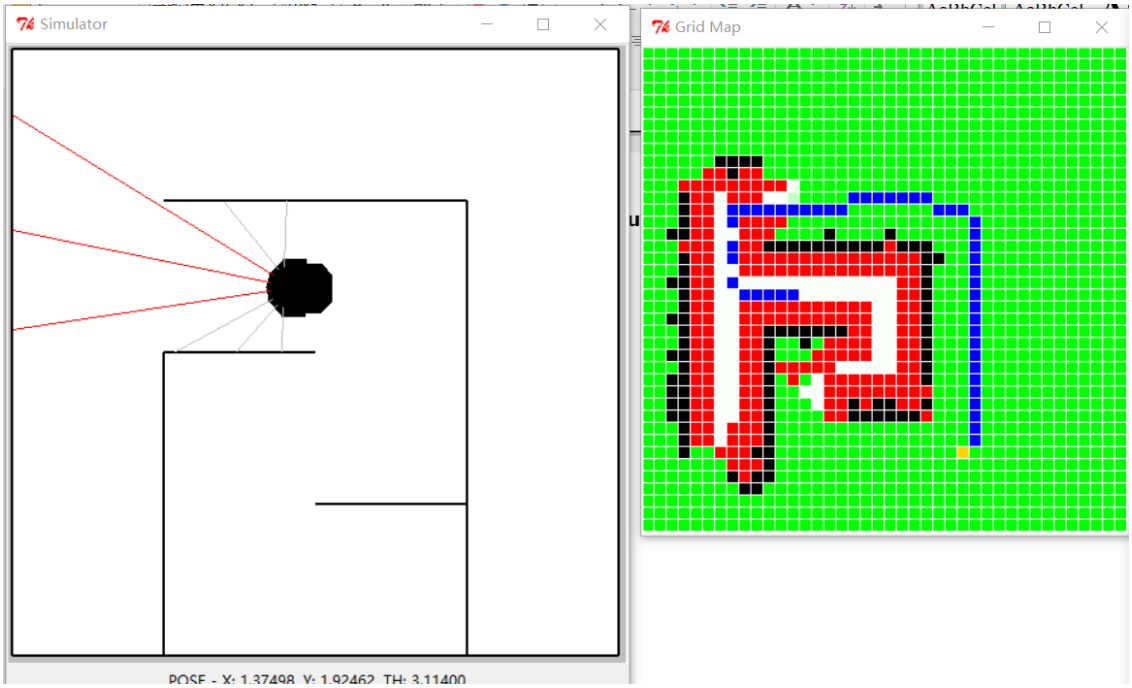
We test many times and find some different results.

1).

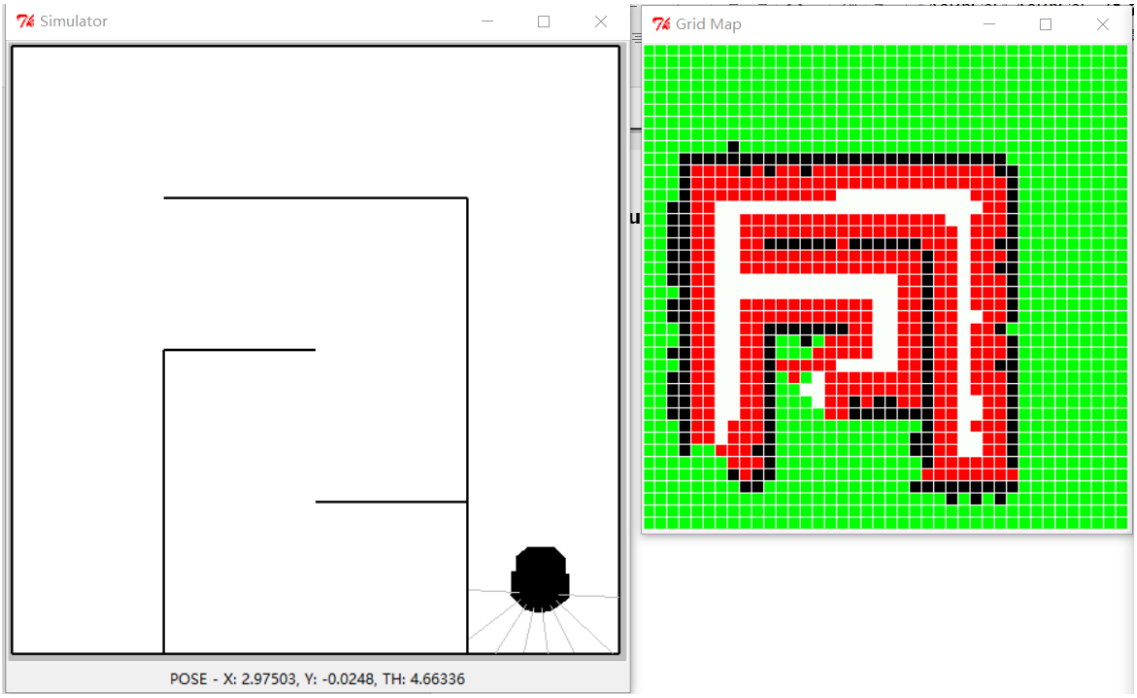
After searching some Nodes,search **failed** when robot was in the Dead end(死胡同)



2) After the robot turned right into a dead end, it quickly replaned its search strategy and made a U-turn(180°转弯)

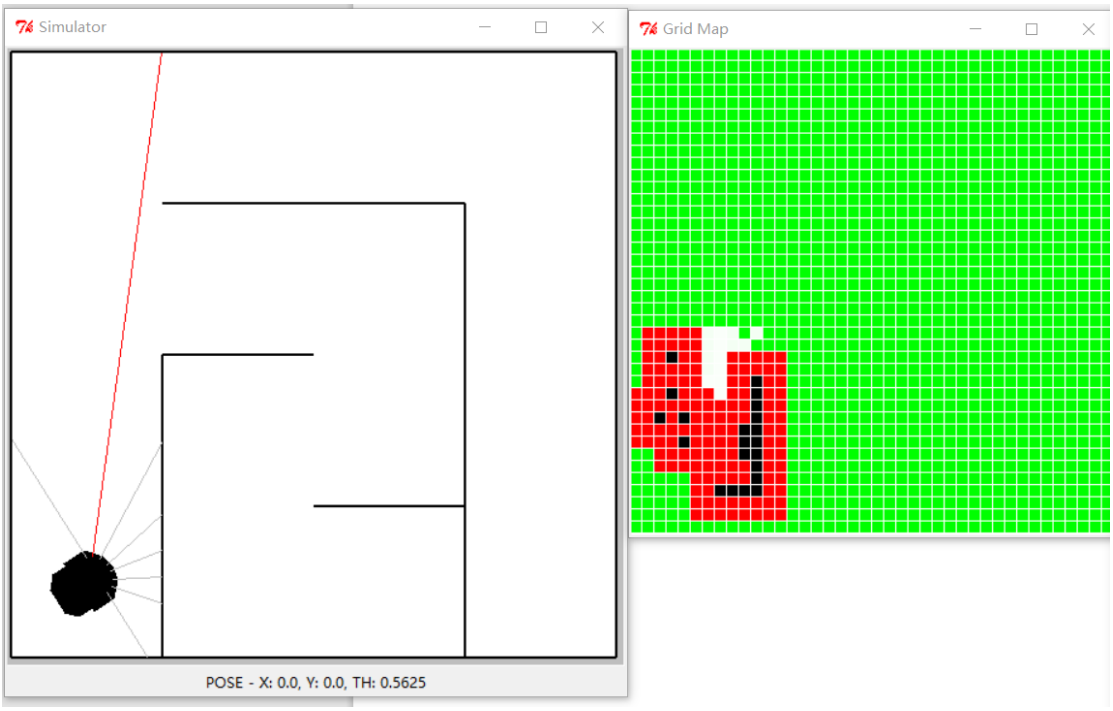


And later, it successfully reach the goal.



3.big noise

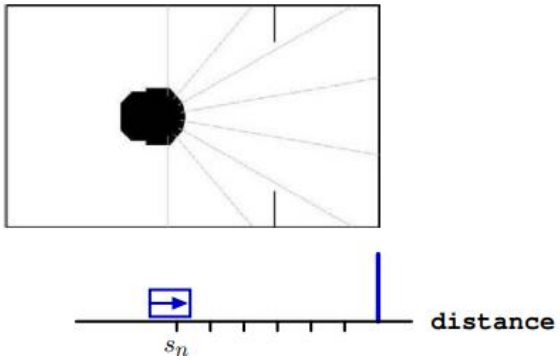
The search failed quickly.



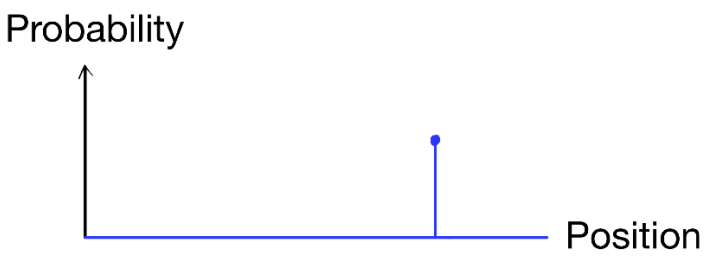
Why does the robot have problem working in high noise?

Explannation:

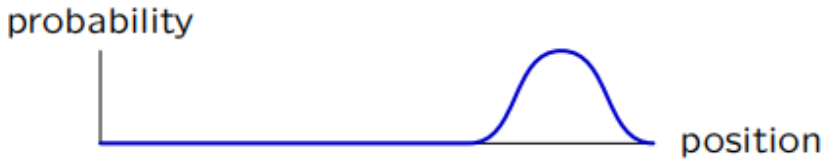
About our observation model:



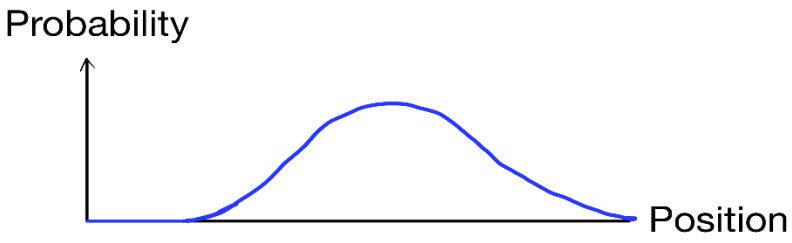
In a no-noise situation, the figure about the probability distribution will be:



But if noise exists,things get different.The noise has an influence on the sonar readings, and in this case,the sonar readings are no longer ideal. So,in a medium-noise situation, the figure about the probability distribution will be:



And in a big-noise situation, the figure will be:

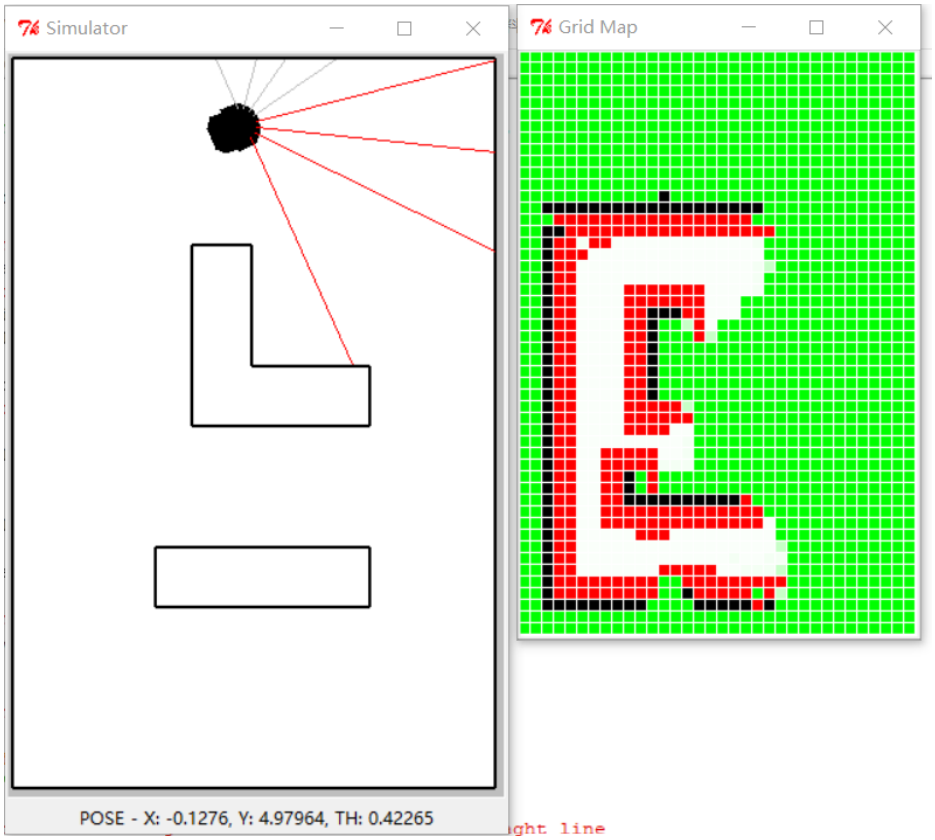


In the **big-noise** situation, if a cell is observed as 'hit', quite a lot cells along the sonar ray will also have a probability to be observed as 'hit'(and this probability can't be ignored) because the probability distribution is very dispersive.And this result will arise a problem to the search process of our "planner" model, because robot will **wrongly** think that many cells around it are already occupied by obstacles and quite a lot cells around it can't be occupied(in case of collision). As a result, this will arise a problem for robot's **"planner"** model, it will have difficulty making a search strategy to reach the goal.

However, in the **medium-noise** situation, the probability distribution is relatively concentrated to a **peak**(that means the distribution is **narrow**).So,the problem it brings to "planner" is relatively small.

5 .Optional: Go, speed racer, go!

Step17 the baseline of our 'score'



Without any optimization strategy, our Total steps are **163**
And the elapesd time in seconds are about **17.731**

Step18 optimize our model and speed up!

1. The first optimization strategy:

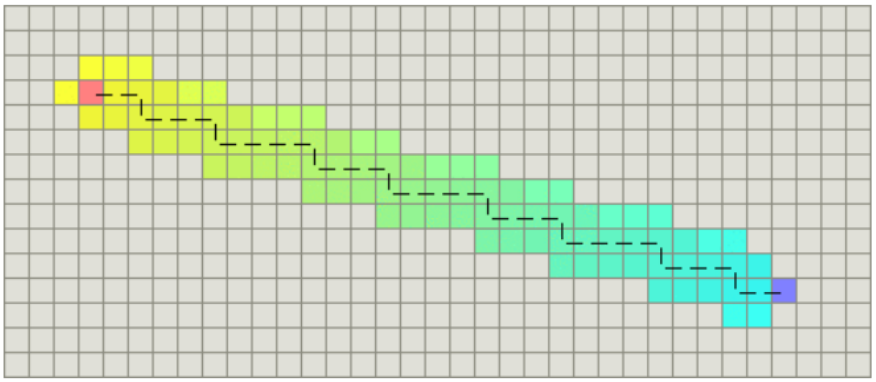
This strategy is came up with by myself. To explain it, we need to back to the ***planner*** we built in SL14.

In SL14, we worked on speeding up planning time by using A* search with a **heuristic.** The ideal heuristic should be :**as close as possible to actual cost (without exceeding it) and easy to calculate.**

Here are some heuristics for our A* search.

1) **Manhattan distance.**

h(n) = D * (abs (n.x – goal.x) + abs (n.y – goal.y))

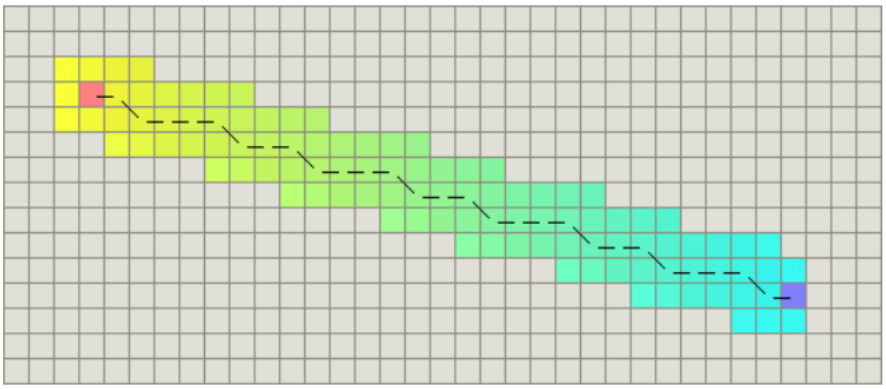


But in our model, it will be not useful. That's because in our model,the state machine should allow 8 possible actions: moving to the four directly adjacent and the four diagonally adjacent grid cells. Robot can move diagonally.

So, let's turn to another two heuristics:

2) **diagonal distance**:

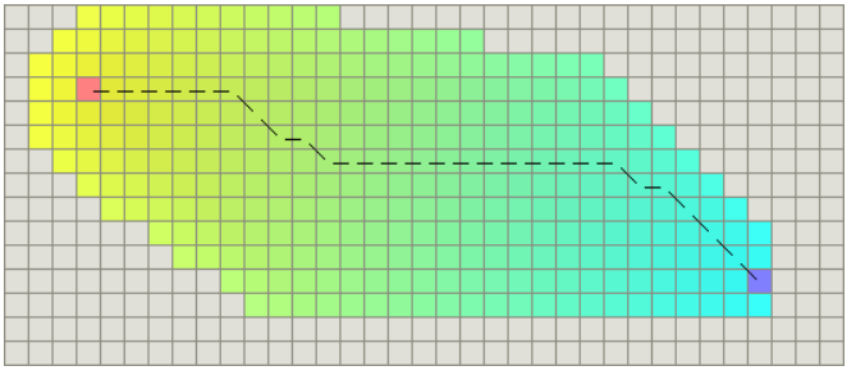
h(n) = D * max(abs(n.x - goal.x), abs(n.y - goal.y))



However, it still has a problem. In our model, a diagonal motion is longer than a horizontal or vertical one which means a diagonal motion will cause a bigger cost. But the principle to use the diagonal distance is that the cost of a diagonal motion and a horizontal one are same.

3) **Euclidean distance**

h(n) = D * sqrt((n.x-goal.x)^2 + (n.y-goal.y)^2)



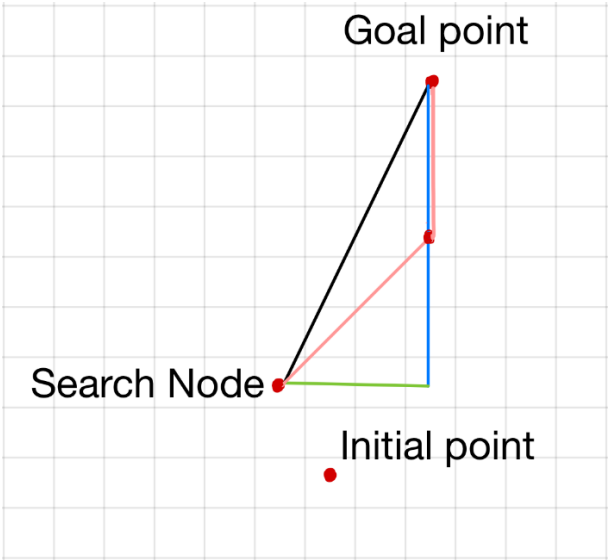
It seems good because our robot can move at any angle. But it still has a problem if we think carefully. The problem is related to the **cost** we defined in *GridDynamics.*

```
cost =  math.sqrt((dx * self.theMap.xStep) ** 2
                  + (dy * self.theMap.yStep) ** 2)
```

The cost only have 2 patterns, **one** for robot moving to **the four directly adjacent**, **the other** for robot moving to **the four diagonally adjacent grid cells.** So, the first cost is **grid.xtep** and the second cost is **grid.xtep***$\sqrt{2}$ **.**

Even though robot can move at any angle, the cost is not the Euclidean distance.

To make the best heuristic, I think about another **"distance"**:



The pink line is our new “distance”. The black line represents the Euclidean distance between a search Node and the goal point. Our actual cost is **the length of the pink line.**

So,we define a new heuristic:

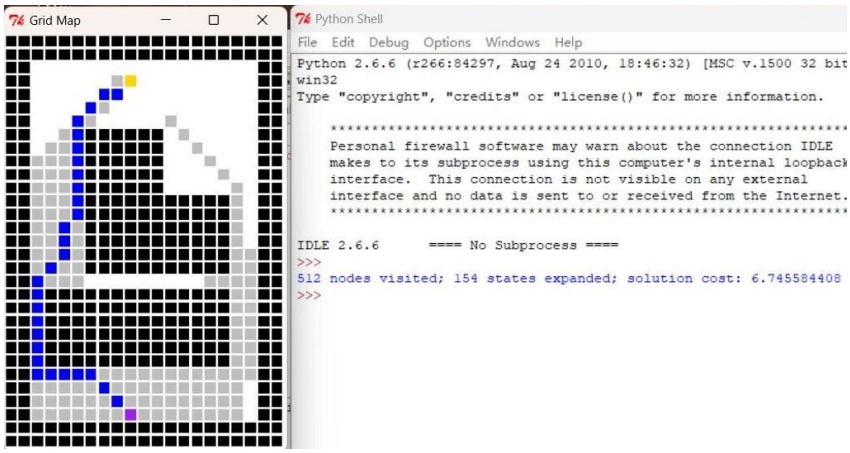
```
def h(s):
    return min(abs(s[0]-goalIndices[0]),abs(s[1]-goalIndices[1]))*(map.xStep*math.sqrt(2))\
      +(max(abs(s[0]-goalIndices[0]),abs(s[1]-goalIndices[1]))-min(abs(s[0]-goalIndices[0]),abs(s[1]-goalIndices[1])))*map.xStep
```

The pink line is made up of two parts.One part is the hypotenuse of an isosceles triangle and the other part is the line that is vertical.

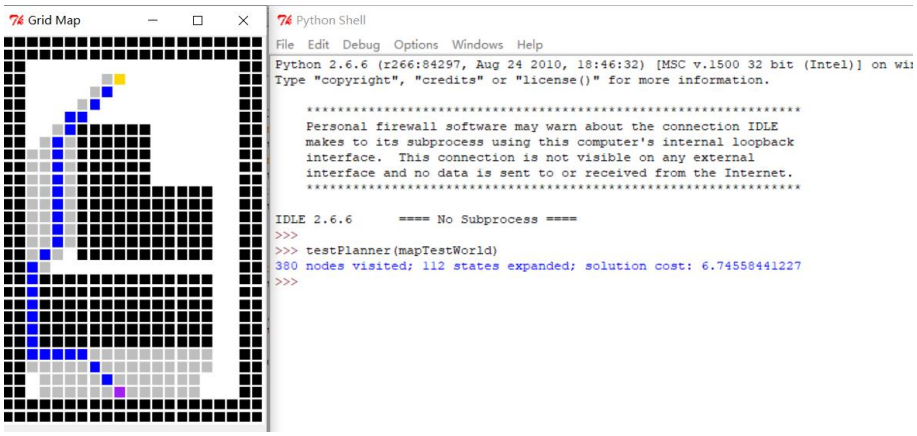
min(abs(s[0]-goalIndices[0]),abs(s[1]goalIndices[1]))
*(map.xStep*math.sqrt(2)) represent the first part of the pink line.

(max(abs(s[0]-goalIndices[0]),abs(s[1]-goalIndices[1]))-
min(abs(s[0]-goalIndices[0]),abs(s[1]-goalIndices[1])))*map.xStep
represent the second part of the pink line.

we test it in IDLE:

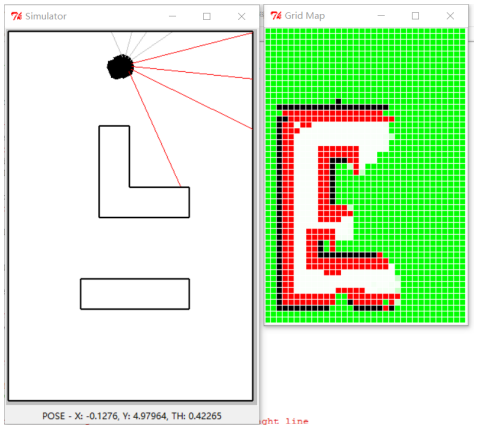


Using **Euclidean distance:**512 nodes visited;154 state expanded



Using our **distance:**380 nodes visited;112 state expanded

We test in soar:




```
[(13, 33)]
Total steps: 112
Elapsed time in seconds: 12.1519999504
```

Reload Brain and World

our Total steps are **150**,and the elapesd time in seconds are about **16.311**

We can see that both the total steps and the elapsed time in seconds are reduced, but the reduction is not significant. So we need other strategies.

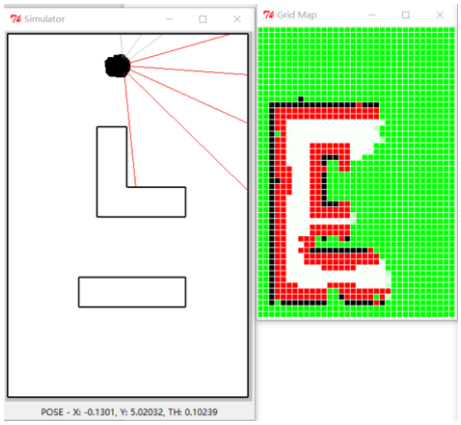
2.the second strategy

1. Plan with the original set of actions, but then post-process the plan to make it more efficient. If the plan asks the robot to make several moves along a straight line, you can safely remove the intermediate subgoals, until the location where the robot finally has to turn.

this strategy is to cut the subgoals when robot move along a straight line. We successfully implement it in our code.

```
if plan:
    # The call to the planner succeeded;  extract the list
    # of subgoals
    state = [s[:2] for (a, s) in plan]
    print 'New plan', state
    del_list=[]
    # remove the intermediate subgoals when robot moves a striaght line
    for i in range(len(state)-2):
       if state[i][0]==state[i+1][0]==state[i+2][0]:
           # x_i=x+1_i=x+2_i, robot moves a straight line
           del_list.append(state[i+1])
       elif state[i][1]==state[i+1][1]==state[i+2][1]:
           # y_i==y+1_i==y+2_i, robot moves a straight line
           del_list.append(state[i+1])
       elif state[i+1][0]-state[i][0]==state[i+2][0]-state[i+1][0] \
            and state[i+1][1]-state[i][1]==state[i+2][1]-state[i+1][1]:
           del_list.append(state[i+1])
           # dx1=dx2, dy1=dy2 , robot moves a straight line
       else:
           pass
    state=[i for i in state if not i in del_list]
    # Draw the plan
    map.drawPath(state)
```

We can simply test it in soar:




```
[(13, 33)]
Total steps: 112
Elapsed time in seconds: 12.1519999504
```

Reload Brain and World

our Total steps are **112**
and the elapesd time in seconds are about **12.15**

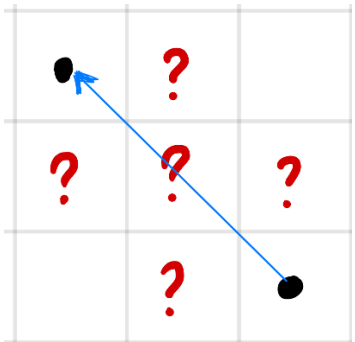
3.the third strategy:

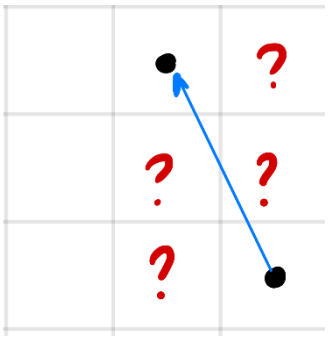
3. Increase the set of actions, to include moves that are more than one square away. You can use the procedure `util.lineIndicesConservative((ix1, iy1), (ix2, iy2))` to get a list of the grid cells that the robot would have to traverse if it starts at `(ix1, iy1)` and ends at `(ix2, iy2)`. This list of grid cells is conservative because it doesn't cut any corners.

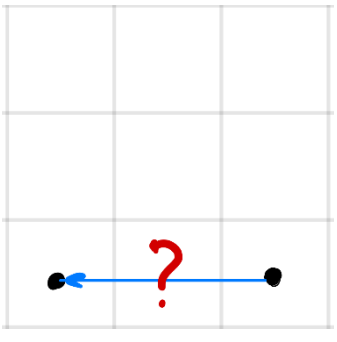
The third strategy is to editing our *GridDynamics* class in order to increase the set of actions. In the original *GridDynamic* class, robot can only move one square away for one action, and it allows 8 possible actions. So,we decide to increase it---robot can move two squares away for one action.

We need consider a lot of additional situations in case that robot will collide into a obstacle.

case1: **case2:** **case3:**

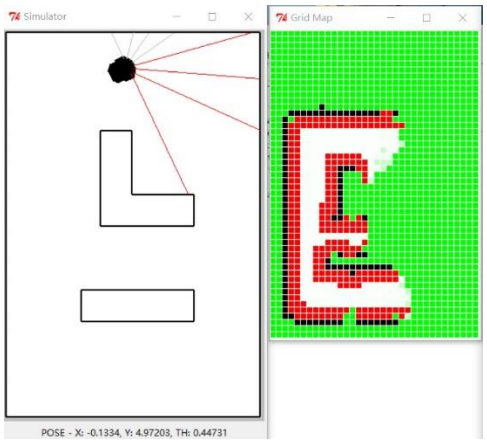






We must ensure that in these cases of action, the cells with a "?" can also be occupied(that means there are no obstacles in the cells with a "?")

Test in soar:




```
Total steps: 104
Elapsed time in seconds: 14.1749999523
```

our Total steps are **104**

and the elapesd time in seconds are about **14.17**

Although this strategy reduce the total steps, it also increase the elapesd time.And we find that when we use this strategy, it will be very unstable when robot doing search, and the graphics software we are using often crashes. So, we don't encouage you to use this strategy.

(maybe the problem is related to the *GridDynamics* class we update.)

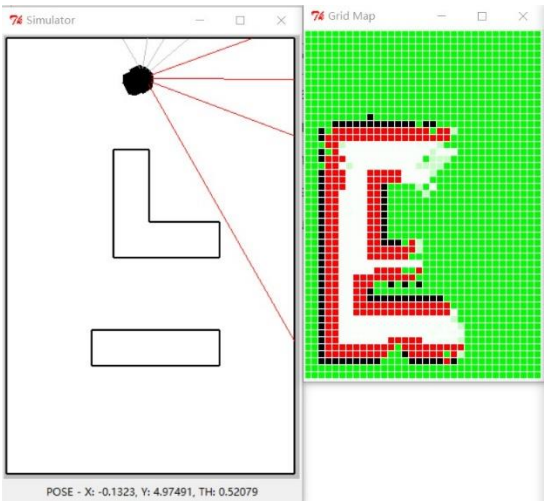
4. The forth strategy

We speed up the execution of the paths by changing the **gains** in the *move.MoveToDynamicPoint* behavior (in *move.py*).

```
move.MoveToDynamicPoint.forwardGain = 1.5
move.MoveToDynamicPoint.rotationGain = 1.5
move.MoveToDynamicPoint.angleEps = 0.05
```

By testing many times, we think that when forwardGain is about 1.5,and rotationGain is about 1.5 ,the result is the best.

This might not be very accurate because the way we chose the best gains is testing in soar.So ,maybe there are better gains.




```
[(13, 33)]
Total steps: 81
Elapsed time in seconds: 8.78199982643
```

Reload Brain and World

our Total steps are **81**

and the elapesd time in seconds are about **8.782**

it actually improves a lot. But in a real robot, we may need to adjust the gains more carefully(it can’t be too high)