Homework 5: Car Tracking 109550171 陳存佩

Part I. Implementation (20%):

Part1

Our target is to convert " current posterior probability " to

" current posterior probability *** P(dt|ct)** "

We go through every grid (using double for loops) by following steps:

First, we use the distance formula to get the distance between our car and one of the grid on the map.

Then, we use the distance to calculate the probability distribution, which means P(dt|ct).

Finally, we update the probability and normalize.

```
def observe(self, agentX: int, agentY: int, observedDist: float) -> None:
    # BEGIN_YOUR_CODE (our solution is 9 lines of code, but don't worry if you deviate from this)
    for row in range(self.belief.numRows): # double for loops: we will go through every grid
        for column in range(self.belief.numCols):
            distance = math.sqrt( (agentX - util.colToX(column)) ** 2 + (agentY - util.rowToY(row)) ** 2 )
            prob_distribution = util.pdf(distance, Const.SONAR_STD, observedDist)
            self.belief.setProb(row, column, self.belief.getProb(row, column) * prob_distribution)
    self.belief.normalize() # makes the sum of all beliefs to be 1
    # END_YOUR_CODE
```

Part2

Our target is to update the posterior probability from current time to next time step.

First, initialize the Belief.

We go through every transition from old tile to new tile which is recorded in self.transProb.

After observe function, we get the probability of car in every position, and we add probabilities which make the car go to new tile.

In the end, we can know where the car most likely go.

```
def elapseTime(self) -> None:
    if self.skipElapse: ### ONLY FOR THE GRADER TO USE IN Part 1
        return
    # BEGIN_YOUR_CODE (our solution is 10 lines of code, but don't worry if you deviate from this)
    initBelief = util.Belief(self.belief.numRows, self.belief.numCols, value=0) #initialize Belief
    for oldTile, newTile in self.transProb:
        initBelief.addProb(newTile[0], newTile[1], self.belief.getProb(*oldTile) * self.transProb[(oldTile, newTile)])
    initBelief.normalize() # makes the sum of all beliefs to be 1
    self.belief = initBelief
    # END_YOUR_CODE
```

Part3-1

In order to reduce the computing cost, we use particle filter.

We use the method like part 1 to update particles.

Then, we create newParticles for grader.

util.weightedRandomChoice() means that we add particles using random weight, and higher probability means more chances to add particles.

```
def observe(self, agentX: int, agentY: int, observedDist: float) -> None:
    # BEGIN_YOUR_CODE (our solution is 12 lines of code, but don't worry if you deviate from this)
    proposed = collections.defaultdict(float)
    for row, column in self.particles:
        distance = math.sqrt( (agentX - util.colToX(column)) ** 2 + (agentY - util.rowToY(row)) ** 2 )
        prob_distribution = util.pdf(distance, Const.SONAR_STD, observedDist)
        proposed[(row, column)] = self.particles[(row, column)] * prob_distribution
    newParticles = collections.defaultdict(int)
    for i in range(self.NUM_PARTICLES):
        particle = util.weightedRandomChoice(proposed)
        newParticles[particle] += 1
    self.particles = newParticles
    # END_YOUR_CODE
    self.updateBelief()
```

Part3-2

We go through every transition from old tile to new tile which is recorded in self.transProbDict.

We use util.weightedRandomChoice() like in part 3-1, so we can interpret probability to particles.

```
def elapseTime(self) -> None:
    # BEGIN_YOUR_CODE (our solution is 6 lines of code, but don't worry if you dev
    newParticles = collections.defaultdict(int)
    for particle in self.particles:
        for _ in range(self.particles[particle]):
            newparticle = util.weightedRandomChoice(self.transProbDict[particle])
            newParticles[newparticle] += 1
    self.particles = newParticles
    # END_YOUR_CODE
```