Introduction to Artificial Intelligence HW4

0711239 李勝維

Part 1: Emission probabilities

```python
def observe(self, agentX: int, agentY: int, observedDist: float):
    # BEGIN_YOUR_CODE
    for col in range(self.belief.numCols):
        for row in range(self.belief.numRows):
            dist = dist2agent(col, row)
            probability_density = util.pdf(
                dist, Const.SONAR_STD, observedDist)
            probability = self.belief.getProb(
                row, col) * probability_density
            self.belief.setProb(row, col, probability)
    self.belief.normalize()
    # END_YOUR_CODE
```

In part 1, we iterate over each tile and updates belief value (probability) based on current probability and the calculated probability density.

We need the distance between the tile and agent to calculate the probability distribution, so we calculate the distance between the tile and agent by function "dist2agent()":

```python
def dist2agent(col, row):
    return math.sqrt((agentX-util.colToX(col))**2 + \
             (agentY-util.rowToY(row))**2)
```

This function calculates the Euclidean distance between agent and given (col, row)

Last, after updating self.belief, we need to normalize it.

Part 2: Transition probabilities

```python
def elapseTime(self) -> None:
    if self.skipElapse:  # ONLY FOR THE GRADER TO USE IN Part 1
        return
    # BEGIN_YOUR_CODE
    new_belief = util.Belief(
        self.belief.numRows, self.belief.numCols, value=0)

    for (oldTile, newTile),transition_probability in self.transProb.items():
        new_belief.addProb(
            row=newTile[0], col=newTile[1],
            delta=self.belief.getProb(*oldTile) * transition_probability)
    new_belief.normalize()
    self.belief = new_belief
    # END_YOUR_CODE
```

In part2, we need to propose a new belief distribution based on a learned transition model.
First, we declare variable “new_belief” and initialize all belief values in it as zero. Variable “new_belief” represents the proposed new belief.

Next, for each new location the car will possibly be (every new tile in self.transProb), we update (fill) the corresponding belief value in “new_belief” by the product of current probability (self.belief.getProb(*oldTile)) and the transition probability.

Last, we normalize new_belief and update self.belief with the proposed new_belief

Part 3-1: Particle filtering(observe)

```python
def observe(self, agentX: int, agentY: int, observedDist: float):
    # BEGIN_YOUR_CODE
    new_weight = dict()
    new_particles = dict()
    # Re-weight
    for (row, col), num in self.particles.items():
        dist = dist2agent(col, row)
        probability_density = util.pdf(
            dist, Const.SONAR_STD, observedDist)
        new_weight[(row, col)] = num * probability_density
    # Re-sample
    for _ in range(self.NUM_PARTICLES):
        key = util.weightedRandomChoice(new_weight)
        if key in new_particles:
            new_particles[key] += 1
        else:
            new_particles[key] = 1
    self.particles = new_particles
```

In function observe(), we re-weight the particle distribution in each tile based on the distance between the tile and agent, then we re-sample particles based on the new weight
First, we declare two new variables “new_weight” and “new_particles”, represents the new weight and new sampled particles correspondingly.

Next, we re-weight particles distribution by the distance between the tile and agent, the function “dist2agent()” actions the same as part1, which calculates the Euclidean distance between the tile and agent. The weight of each tile is updated by the product of current weight and the probability density.

After that, we sample N times, where N equals “self.NUM_PARTICLES”, with the calculated new weight.

Last, we update the distribution of particles by the re-sampled one.

Part 3-2: Particle filtering(elapseTime)

```
def elapseTime(self):
    # BEGIN_YOUR_CODE
    new_particles = collections.defaultdict(int)
    for particle in self.particles:
        num = self.particles[particle]
        for _ in range(num):
            new_weight = self.transProbDict[particle]
            key = util.weightedRandomChoice(new_weight)
            new_particles[key] += 1
    self.particles = new_particles
    # END_YOUR_CODE
```

In this function, we propose a new belief distribution based on a learned transition model.
Since the belief values are based on the particle distribution, we can simply update the particle distribution by re-sample it based on the learned transition model.
First, we declare a variable “new_particle” represents the new particle distribution.

Next, we loop over every tile in self.particles(represents how much particles are in each tile), and re-sample N times, where N equals the number of particles in this tile.

After that, for each particle sample, we get the new_weight (particle distribution) by the learned transition model, self.transProbDict, and randomly get a tile, “key”, and record this particle in “new_particles”.

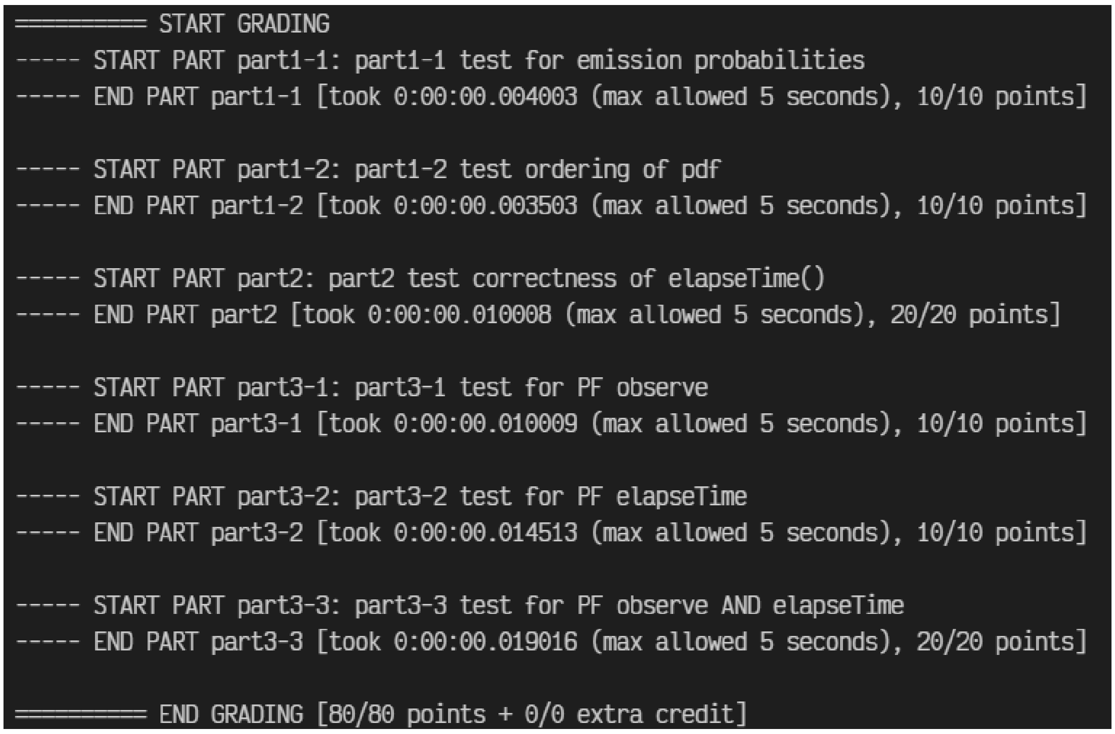
Finally, we update self.particles with the re-sampled one “new_particles”.

Problems I meet and how I solve them:
Q: While implementing part 2, the value of new_belief is not right.
A: I accidentally ignored that different oldTile may lead to same newTile, so instead of set the value of new_belief with the most recent one, I set the value of new_belief with the sum of every possibility

Q: While implementing part 2, I update the new belief value in-place, which leads to an incorrect result.
A: Stores the new belief value in another variable, then update self.belief with the new variable.

Q: While implementing part 3-2, I always get a KeyError from grader.
A: It seems to be that the type of self.particles must be a collection.defaultdict() instead of a python dict().

Grader result:



```
========== START GRADING
----- START PART part1-1: part1-1 test for emission probabilities
----- END PART part1-1 [took 0:00:00.004003 (max allowed 5 seconds), 10/10 points]

----- START PART part1-2: part1-2 test ordering of pdf
----- END PART part1-2 [took 0:00:00.003503 (max allowed 5 seconds), 10/10 points]

----- START PART part2: part2 test correctness of elapseTime()
----- END PART part2 [took 0:00:00.010008 (max allowed 5 seconds), 20/20 points]

----- START PART part3-1: part3-1 test for PF observe
----- END PART part3-1 [took 0:00:00.010009 (max allowed 5 seconds), 10/10 points]

----- START PART part3-2: part3-2 test for PF elapseTime
----- END PART part3-2 [took 0:00:00.014513 (max allowed 5 seconds), 10/10 points]

----- START PART part3-3: part3-3 test for PF observe AND elapseTime
----- END PART part3-3 [took 0:00:00.019016 (max allowed 5 seconds), 20/20 points]

========== END GRADING [80/80 points + 0/0 extra credit]
```