Homework 5: Car Tracking

Part I. Implementation (15%):

Part 1.

```python
def observe(self, agentX: int, agentY: int, observedDist: float) -> None:
    # BEGIN_YOUR_CODE
    # Iterate all locations
    for row in range (self.belief.numRows):
        for column in range (self.belief.numCols):
            # Get the distance between the grid and my car
            distance_to_car = math.sqrt(pow((util.colToX(column) - agentX), 2) + pow((util.rowToY(row) - agentY), 2))

            # Calculate the PDF with mean = distance to my car, std = Const.SONAR_STD, value = observedDist
            pdf = util.pdf(distance_to_car, Const.SONAR_STD, observedDist)

            # Update the probability
            self.belief.setProb(row, column, self.belief.getProb(row, column) * pdf)
    # Normalize self.belief
    self.belief.normalize()
    # END_YOUR_CODE
```

In this code snippet, we have a function called observe that updates the probabilities stored in self.belief, which is an instance of the Belief class defined in util.py. This update is performed using an observed distance, observedDist, and your car's position represented by agentX and agentY.

To accomplish this, the code follows these steps:

1. It iterates over all the locations in the grid.
2. For each location, it calculates the distance between the grid and your car using the util.colToX and util.rowToY functions.
3. Then, it computes the probability density function (PDF) by calling util.pdf, passing in the distance to your car as the mean, Const.SONAR_STD as the standard deviation, and observedDist as the value.
4. The code updates the probability for each tile by multiplying the original probability with the PDF obtained in the previous step.
5. After updating all the probabilities, the code normalizes self.belief to ensure that the updated probabilities sum up to 1

Overall, the observe function performs the necessary calculations and updates the probabilities on each tile based on the observed distance, using the provided utilities and methods.

Part 2.

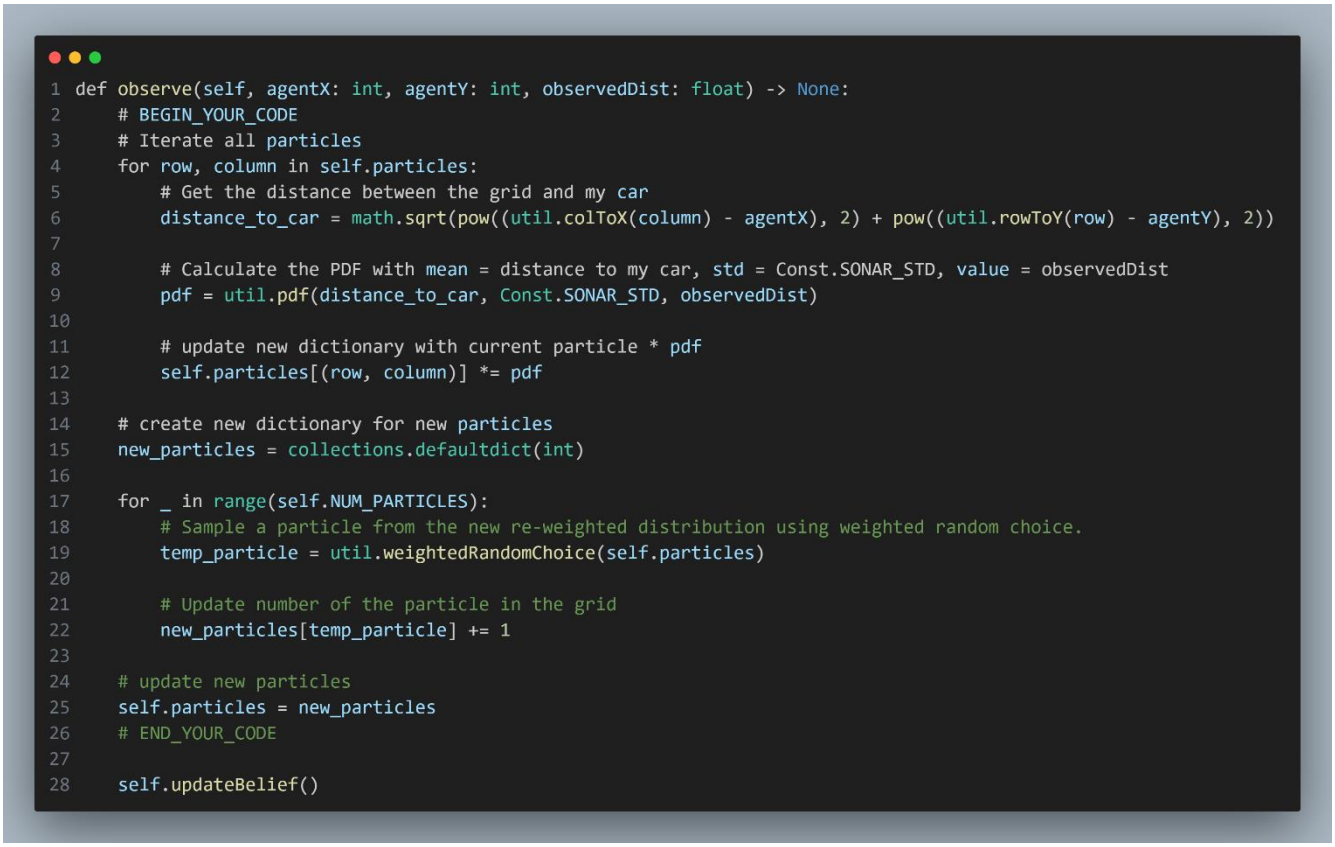
```python
def elapseTime(self) -> None:
    if self.skipElapse: ### ONLY FOR THE GRADER TO USE IN Part 1
        return
    # BEGIN_YOUR_CODE

    # Initialize new belief with value is 0
    new_belief = util.Belief(self.belief.numRows, self.belief.numCols, value=0)

    for (oldTile, newTile), transProb in self.transProb.items():
        # Get the location of old tile and new tile
        old_r, old_c = oldTile
        new_r, new_c = newTile

        # Update the probability with new location and current probability and transition probability
        new_belief.addProb(new_r, new_c, self.belief.getProb(old_r, old_c) * transProb)

    # Update self belief and normalize
    self.belief = new_belief
    self.belief.normalize()
    # END_YOUR_CODE
```

The code snippet demonstrates a function called elapseTime that proposes a new belief distribution based on a learned transition model. The following steps are involved:

1. The code initializes a new belief distribution called new_belief using the Belief class from util.py. All the probabilities in this new belief are initially set to 0.
2. It iterates through each key-value pair in the self.transProb dictionary, where the keys represent the transition from an old tile to a new tile, and the values represent the corresponding transition probabilities.
3. For each (oldTile, newTile), transProb pair, it extracts the row and column indices for both the old and new tiles.
4. The code updates the probability for the new tile using the formula: new_belief.addProb(new_r, new_c, self.belief.getProb(old_r, old_c) * transProb). It multiplies the probability of the old tile (retrieved from self.belief) with the corresponding transition probability and adds it to the probability of the new tile in new_belief.
5. After updating all the probabilities based on the transition model, the code assigns new_belief to self.belief, effectively replacing the old belief distribution with the new one.
6. Finally, the code normalizes the updated belief distribution using the normalize method to ensure that the sum of probabilities is exactly 1.

In summary, the elapseTime function creates a new belief distribution by updating the probabilities of each tile based on the learned transition model. It iterates through the transition probabilities, calculates the new probabilities for the corresponding tiles, and normalizes the updated belief distribution.

Part 3-1.



```python
def observe(self, agentX: int, agentY: int, observedDist: float) -> None:
    # BEGIN_YOUR_CODE
    # Iterate all particles
    for row, column in self.particles:
        # Get the distance between the grid and my car
        distance_to_car = math.sqrt(pow((util.colToX(column) - agentX), 2) + pow((util.rowToY(row) - agentY), 2))

        # Calculate the PDF with mean = distance to my car, std = Const.SONAR_STD, value = observedDist
        pdf = util.pdf(distance_to_car, Const.SONAR_STD, observedDist)

        # update new dictionary with current particle * pdf
        self.particles[(row, column)] *= pdf

    # create new dictionary for new particles
    new_particles = collections.defaultdict(int)

    for _ in range(self.NUM_PARTICLES):
        # Sample a particle from the new re-weighted distribution using weighted random choice.
        temp_particle = util.weightedRandomChoice(self.particles)

        # Update number of the particle in the grid
        new_particles[temp_particle] += 1

    # update new particles
    self.particles = new_particles
    # END_YOUR_CODE

    self.updateBelief()
```

The code snippet showcases a function called observe that updates the particles based on a distance observation. The process involves two steps:

Step 1: Re-weighting the particles based on the observation

- Iterate through each particle in self.particles.
- For each particle, calculate the distance between the grid and your car using the provided agentX and agentY coordinates.
- Compute the probability density function (PDF) using util.pdf, where the mean is the distance to your car, the standard deviation is Const.SONAR_STD, and the value is the observed distance.
- Multiply the current particle count with the PDF to update the weight of the particle.

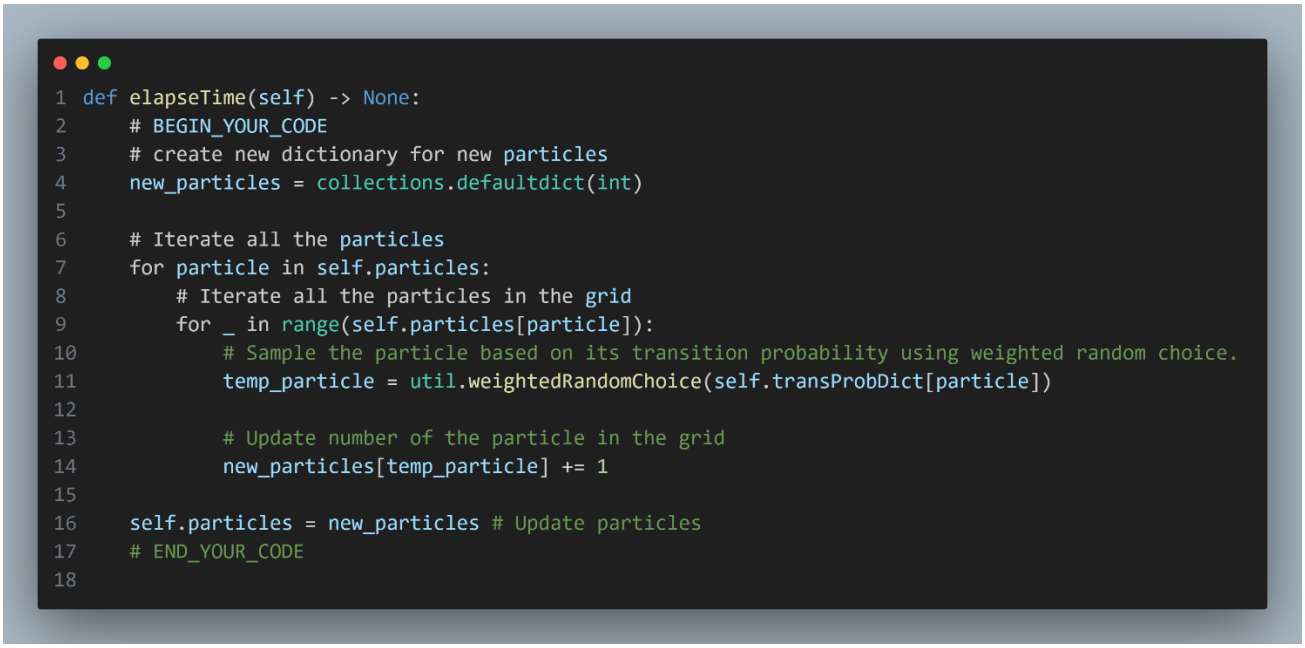
Step 2: Re-sampling the particles

- Create a new dictionary, new_particles, to store the resampled particles.
- Iterate self.NUM_PARTICLES times to generate new particles during resampling.
- Use util.weightedRandomChoice to select a new particle from the updated (unnormalized) particle distribution, ensuring that particles with higher weights have a higher chance of being selected.
- Increase the count of the selected particle in new_particles by 1.

After resampling, the code updates self.particles with the new particle distribution stored in new_particles. Finally, it calls the updateBelief function to update the belief distribution based on the updated particles.

In summary, the observe function updates the particles by re-weighting them according to the emission probability obtained from the observed distance and resampling them to create a new particle distribution. This process allows for more accurate tracking by adjusting the particles' weights based on the observation.

Part 3-2.



```python
def elapseTime(self) -> None:
    # BEGIN_YOUR_CODE
    # create new dictionary for new particles
    new_particles = collections.defaultdict(int)

    # Iterate all the particles
    for particle in self.particles:
        # Iterate all the particles in the grid
        for _ in range(self.particles[particle]):
            # Sample the particle based on its transition probability using weighted random choice.
            temp_particle = util.weightedRandomChoice(self.transProbDict[particle])

            # Update number of the particle in the grid
            new_particles[temp_particle] += 1

    self.particles = new_particles # Update particles
    # END_YOUR_CODE

```

The code snippet demonstrates a function called elapseTime that updates the particles based on a learned transition model. The process involves one step:

Step 1: Proposing new particle locations based on the particle distribution at the current time

- Create a new dictionary, new_particles, to store the updated particles.
- Iterate through each particle in self.particles.
- For each particle, iterate through the particles in the corresponding grid location.
- Sample a new particle location for each existing particle based on the transition probabilities stored in self.transProbDict. Use util.weightedRandomChoice to select a new particle location, taking into account the transition probabilities.
- Increase the count of the selected new particle location in new_particles by 1.

After updating the particle locations, the code assigns the new particle distribution in new_particles to self.particles.

In summary, the elapseTime function updates the particles by proposing new particle locations based on the transition probabilities. It iterates through each particle in each grid location, resampling its next most probable location based on the transition probability dictionary. The result is a new particle distribution that represents the updated belief about the particle locations at the next time step.

Part II. Question answering (5%):

The most challenging aspect I encountered during this homework assignment was understanding the concept of Bayesian networks and correctly updating the probabilities in the code. Since the task involved working with probabilistic reasoning and inference, it was crucial to have a solid understanding of these concepts in order to implement the code accurately.

To tackle this difficulty, I took proactive steps to enhance my comprehension. I dedicated time to watch instructional videos and thoroughly review the course materials related to Bayesian networks. This allowed me to strengthen my understanding and clarify any uncertainties before starting the coding process.

I made sure to carefully read and understand the comments provided in the code. These comments explained the purpose and logic behind each code section, which helped me in planning and executing the implementation correctly.

Furthermore, I leveraged testing as a means of validating my implementation. By creating various scenarios and comparing the results against the expected outcomes, I could identify and rectify any errors or misconceptions in the code. Additionally, I used debugging techniques such as printing intermediate results or utilizing a debugger to assist in locating and resolving any issues encountered during the implementation process.

In summary, the main difficulty I faced while working on this homework was comprehending Bayesian networks and accurately updating the probabilities. Through proactive learning, careful reading of code comments, and thorough testing, I was able to overcome this challenge and successfully implement the code.