CS461 Artificial Intelligence
Homework 4

Spring 2023

Due: May 23, 2023, 23:59

Instructions

- You will submit your Python codes with.py format to Moodle. Only modify the specified parts (highlighted by `*** YOUR CODE HERE ***` comments) of the provided files, you will lose the whole grade otherwise. There are other parts of the codes you can (but may not need to) change, which are explicitly mentioned in the comments.
- You will submit your codes individually or in groups of two. Please also include the name and the student ID of each member in a text file named `members.txt`.
- You will submit only four files: `bustersAgents.py`, `inference.py`, `factorOperations.py`, and `members.txt`. Any other file will be ignored.
- You will need `Python 3.6` installed on your computer to run the environment. Your implementation must not change the package/version requirements. Please note that you might face errors running the scripts if you work on other versions of Python. It's recommended to use the Anaconda distribution platform to easily install multiple independent versions of Python.
- Only one member should upload a .zip file including the mentioned python files.
- The codes will be checked for plagiarism. This check will include publicly available implementations for this homework. Submit your own work.
- Your codes will also be evaluated in terms of efficiency. Make sure that you do not have unnecessary loops and obvious inefficient calculations in your code.
- We follow no extension policy. However, you will lose 25% of the grade per day of late submission, up to 2 days.
- The files have been tested before being uploaded to Moodle. However, if you still have problems using and running the codes, you can contact the course TAs: Barış Bilgin Şenol (bilgin.senol@bilkent.edu.tr) and Navid Ghamari (navid.ghamari@bilkent.edu.tr).

0 Python Environment Setup

This section describes the Python environment that will be used during the grading of your homework. You do **not** have to follow the same setup, as long as you meet the requirements listed in the instructions and get proper grades. However, it is recommended to follow the same setup to avoid inconsistencies that you may encounter while working on the assignment.

The Python environment will be managed by `conda`. If you have not used it before, take this as a chance to get used to that good practice.

1. Download and install Anaconda [1]. Miniconda is also fine if you prefer a minimal installation [2].
2. Search for the `Anaconda Prompt` on your computer. It will launch a command line interface with `base` environment already activated.
3. Create a new `Python 3.6` environment by entering the following command on the Anaconda Prompt. This will create a new `conda` environment named `cs461-hw4` and install `Python 3.6` along with some base packages. Refer to [3] for further information on Conda basics and managing environments.

```
conda create --name cs461-hw4 python=3.6
```

4. Confirm the prompt "Proceed ([y]/n)?" by pressing enter.
5. Enter the following command to the Anaconda Prompt. This will activate the environment you just created. You should see the environment name `cs461-hw4` in parentheses at the beginning of the line.

```
conda activate cs461-hw4
```

6. Enter the following command to verify the Python version.

```
python --version
```

7. You should see an output similar to the one below:

```
Python 3.6.13 :: Intel Corporation
```

8. That is it! Run the necessary checks shown under each question in the assignment within this environment. Also, don't forget to select your interpreter as this environment in your code editor of choice.

1 Bayes Nets

Until this point, we have worked on the standard Pac-Man game, with the regular objectives of eating pellets and running from ghosts. In this assignment [4], you will consider a different version where your Pac-Man agent is blind and ghosts are permanently scared of Pac-Man. However, Pac-Man receives some noisy readings of Manhattan distance to each ghost from the environment. Your objective is to hunt down all the ghosts. You can try playing the game yourself with the keyboard by running:

```
python busters.py
```

The colored blocks in the game indicate the region where each of the ghosts could possibly be located. You receive noisy distance metrics that are always non-negative and within 7 of the actual distance. Naturally, we want a better estimation of where the ghosts are located than this simple implementation. Your objective is to implement algorithms to perform inference based on Bayes Nets.

Question 1 [8 Points]

Bayes Net Structure. Your first task is to construct an empty Bayes Net structure. A Bayes Net is not complete without the probabilities, however, they are assigned automatically in the code. If you wish, you can look into `printStarterBayesNet` function in `bayesNet.py` to get a better understanding. We will try to build the net by following the diagram in Fig. 1. You need to add variables and edges based on the diagram and add all possible position tuples for the Pacman and the two ghosts. The observations you see is equal to the Manhattan distances with some added noise. You will complete the `constructBayesNet` function in the `inference.py` file.

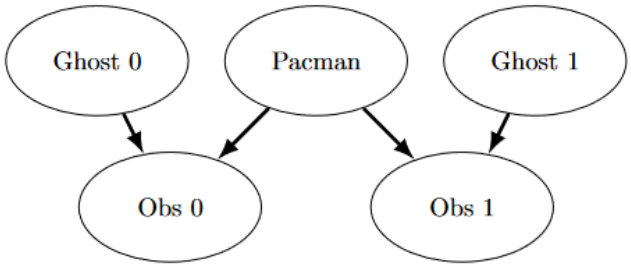

Figure 1: Simplified Bayes Net diagram

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q1
```

Question 2 [12 Points]

Join Factors. Next, you will implement `joinFactors` function to incorporate probabilities for the conditioned variables. For example, combining factor $P(X|Y)$ with $P(Y)$ should yield $P(X, Y)$. The function takes in a list of `Factor`s and you should return a new `Factor`. You will complete the `joinFactors` function in the `factorOperations.py` file.

You may find it useful to see only one set of factors during debugging. For example, to only run the first test, run the following command:

```
python autograder.py -t test_cases/q2/1-product-rule
```

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q2
```

Question 3 [8 Points]

Elimination. You will complete the `eliminate` function in the `factorOperations.py` file. This function takes in a `Factor` and a variable to eliminate as inputs and should return a new `Factor` that excludes that variable. Mathematically, this process involves summing all the entries in the `Factor` that differ only in the value of the variable being eliminated.

Hint: Consider which variables are unconditioned (or conditioned) in the returned `Factor`.

You may find it useful to see only one set of factors during debugging. For example, to only run the first test, run the following command:

```
python autograder.py -t test_cases/q3/1-simple-eliminate
```

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q3
```

Question 4 [8 Points]

Variable Elimination. You will complete the `inferenceByVariableElimination` function in the `inference.py` file. It answers a probabilistic query, which is represented using a `BayesNet`, a list of query variables, and the evidence.

- The algorithm for solving this problem should go through the hidden variables in the order they need to be eliminated; join the factor over the variable, eliminate it, and repeat until only the query and evidence variables are left.
- The output `Factor` you create should have probabilities that add up to 1, which ensures that it's a valid conditional probability based on the evidence.
- If you need help using the required functions, take a look at the `inferenceByEnumeration` function in `inference.py`. Keep in mind that this function first joins over all variables and then eliminates the hidden ones, while variable elimination alternates between joining and eliminating hidden variables.
- You'll need to handle the case where a factor you're joining only has one unconditioned variable, as explained in the docstring for this function.

You may find it useful to see only one set of factors during debugging. For example, to only run the first test, run the following command:

```
python autograder.py -t test_cases/q4/1-disconnected-eliminate
```

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q4
```

Question 5 [4 Points]

DiscreteDistribution Class. Due to the exponential growth of the graph when using timesteps, variable elimination is not a viable option for exact inference. Instead, we will utilize the Forward Algorithm for Hidden Markov Models (HMMs) for exact inference and Particle Filtering for faster, albeit approximate inference.

For the remaining parts of the project, the `DiscreteDistribution` class defined in `inference.py` will be used to model belief and weight distributions. This class is an extension of the built-in Python dictionary class, where the keys represent discrete elements of the distribution, and the corresponding values are proportional to the assigned belief or weight. Although this question is worth no points, it is crucial to complete this class's missing parts, which will be required for later questions.

The first task is to fill in the `normalize` method, which normalizes the distribution values such that they sum to one while keeping their proportions unchanged. The `total` method should be utilized to calculate the distribution's sum of values. If the distribution is empty or has zero values, do nothing. It is worth noting that this method modifies the distribution itself, rather than returning a new distribution.

The second task is to fill in the `sample` method, which selects a sample from the distribution where the probability of selecting a key is proportional to its corresponding value. It is assumed that the distribution is not empty, and not all of its values are zero. Note that it is not necessary to normalize the distribution before calling this method. The built-in `random.random()` function may be useful for this task.

Observation Probability. Implement the `getObservationProb` method in the `InferenceModule` base class in `inference.py`. The method takes in Pacman's position, the ghost's position, the position of the ghost's jail, and an observation which is a noisy reading of the distance to the ghost, and returns the probability of the noisy distance given Pacman's position and the ghost's position. Use the provided `busters.getObservationProbability(noisyDistance, trueDistance)` function, which models the probability distribution over distance readings, and the `manhattanDistance` function to calculate the distance between Pacman and the ghost.

There is a special case when the ghost is in jail, where the distance sensor returns `None`. If the ghost's position is the jail position, then the observation is `None` with probability 1, and everything else with probability 0. Handle this case in your implementation: Essentially, we have a different set of rules when the observation is `None`.

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q5
```

Question 6 [12 Points]

Exact Inference Observation. In this question, you need to implement the `observeUpdate` method in `ExactInference` class of `inference.py`, which updates the agent's belief distribution over ghost positions based on an observation from Pacman's sensors. You will be implementing an online belief update for new evidence. You should update the belief at every position on the map using the `self.allPositions` variable, including the special jail position. Beliefs represent the probability of the ghost's presence at a certain location, and are stored as a `DiscreteDistribution` object in `self.beliefs`.

Before coding, try to write the equation of the inference problem you are trying to solve. Use the `self.getObservationProb` function you wrote in the previous question, which returns the probability of an observation given Pacman's position, a possible ghost position, and the jail position. You can get the Pac-Man and jail positions using `gameState.getPacmanPosition()` and, `self.getJailPosition()` respectively. The Pac-Man display shows high posterior beliefs with bright colors, and low beliefs with dim colors. Start with a large cloud of belief that shrinks as more evidence accumulates. Make sure you understand how the squares converge to their final colors.

Note: Each busters agent has a separate inference module for each ghost being tracked. So, printing an observation inside the `observeUpdate` function will show a single number even if there are multiple ghosts on the board.

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q6
```

Alternatively, you can run the test in no graphics mode:

```
python autograder.py -q q6 --no-graphics
```

Question 7 [12 Points]

Exact Inference with Time Elapse. In the last question, you updated Pacman's beliefs about the ghost's location based on observations. However, Pacman also knows how the ghost can move and cannot move. This knowledge can help Pacman when he receives an erroneous reading.

This question requires you to implement the `elapseTime` method in `ExactInference`, which updates the belief at every position on the map after one time step. You can obtain the distribution over new positions for the ghost, given its previous position, by using `self.getPositionDistribution`. This method returns a `DiscreteDistribution` object, where for each position, it calculates the probability that the ghost is at that position at time $t + 1$, given that the ghost is at position `oldPos` at time t.

Before typing any code, write down the equation of the inference problem you are trying to solve. Pacman's beliefs will come to reflect places on the board where he believes ghosts are most likely to be, given the geometry of the board and ghosts' possible legal moves. This question will not use the update implementation from the previous question to test your predict implementation separately. The Bayes Net diagram below shows what is happening, but rely on the description above for implementation.

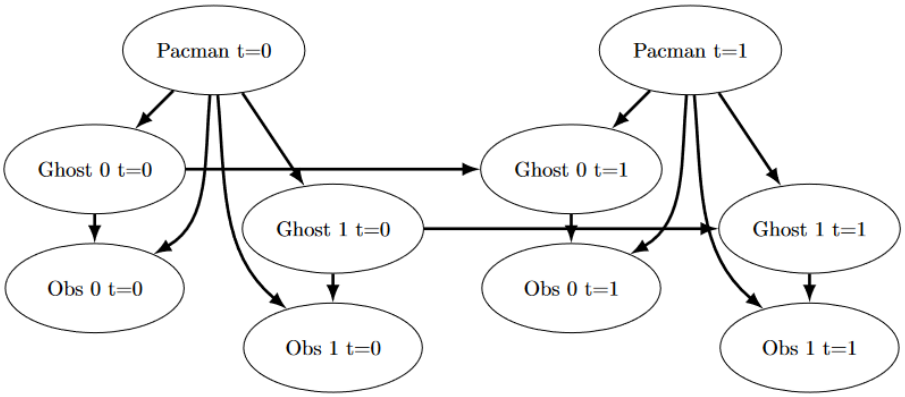

Figure 2: Bayes Net / HMM diagram

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q7
```

Alternatively, you can run the test in no graphics mode:

```
python autograder.py -q q7 --no-graphics
```

Question 8 [4 Points]

Exact Inference Full Test. Pacman will use `observeUpdate` and `elapseTime` implementations to keep an updated belief distribution and choose an action based on the latest distribution at each time step. The simple greedy strategy involves Pac-Man assuming that each ghost is in its most likely position according to his beliefs, and then moving towards the closest ghost. Your task is to implement the `chooseAction` method in `GreedyBustersAgent` in `bustersAgents.py`. Your agent should first find the most likely position of each remaining uncaptured ghost, then choose an action that minimizes the maze distance to the closest ghost.

Use `self.distancer.getDistance(pos1, pos2)` to find the maze distance between any two positions `pos1` and `pos2`, and use `Actions.getSuccessor(position, action)` to find the successor position of a position after an action. The autograder will check if your agent can win the game in `q8/3-gameScoreTest` case with a score greater than 700 at least 8 out of 10 times. How the greedy agent works can be represented with the following modification to the previous diagram:

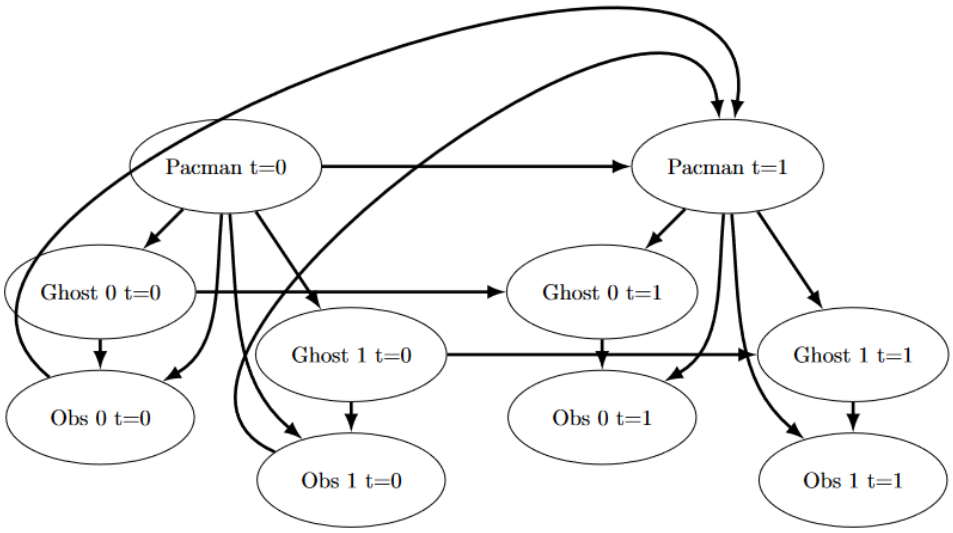

Figure 3: Modified Bayes Net / HMM diagram

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q8
```

Alternatively, you can run the test in no graphics mode:

```
python autograder.py -q q8 --no-graphics
```

Question 9 [8 Points]

Approximate Inference Initialization and Beliefs. For the next few questions, you will be implementing a particle filtering algorithm for tracking a single ghost.

Implement `initializeUniformly` and `getBeliefDistribution` in the `ParticleFilter` class in `inference.py`. For initialization, particles should be evenly (not randomly) distributed across legal positions to ensure a uniform prior. Use a list to store particles and convert it into a `DiscreteDistribution` object in the `getBeliefDistribution` method.

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q9
```

Question 10 [12 Points]

Approximate Inference Observation. Implement `observeUpdate` method in `ParticleFilter` class in `inference.py`. This method computes a weight distribution over `self.particles` based on the probability of the observation given Pac-Man's position and each particle's location. Next, resample from this weighted distribution to create a new list of particles.

Use `self.getObservationProb` to calculate the probability of an observation given Pac-Man's position, a potential ghost position, and the jail position. Use the sample method of the `DiscreteDistribution` class to resample from the weight distribution. Use `gameState.getPacmanPosition()` to get Pacman's position and `self.getJailPosition()` to get the jail position.

If all particles have zero weight, reinitialize the list of particles by calling `initializeUniformly`. The `total` method of the `DiscreteDistribution` can be used for this purpose.

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q10
```

Alternatively, you can run the test in no graphics mode:

```
python autograder.py -q q10 --no-graphics
```

Question 11 [12 Points]

Approximate Inference with Time Elapse. In the `ParticleFilter` class of `inference.py`, implement the `elapseTime` function that updates each particle in `self.particles` by advancing a time step and assigning the new list of particles back to `self.particles`. This will help track ghosts almost as accurately as exact inference. Use the following code to get the distribution over new positions for the ghost, given its previous position (`oldPos`):

```
newPosDist = self.getPositionDistribution(gameState, oldPos)
```

The `sample` method of the `DiscreteDistribution` class may also be useful. Keep in mind that both the `elapseTime` function alone and the full implementation of the particle filter combining `elapseTime` and `observeUpdate` will be tested.

Grading. You can test your implementation by running the following command:

```
python autograder.py -q q11
```

Alternatively, you can run the test in no graphics mode:

```
python autograder.py -q q11 --no-graphics
```

2 Grading

Although your homework will be graded using an autograder with the mentioned details, your implementations will also be checked. In any case, only the correct implementations will be accepted.

References

[1] Anaconda web page. https://www.anaconda.com/. Accessed: 2023-04-04.

[2] Miniconda documentation. https://docs.conda.io/en/latest/miniconda.html. Accessed: 2023-04-04.

[3] Conda cheat sheet. https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf. Accessed: 2023-04-04.

[4] Berkeley project web page. https://inst.eecs.berkeley.edu/~cs188/sp23/projects/proj5/. Accessed: 2023-04-29.