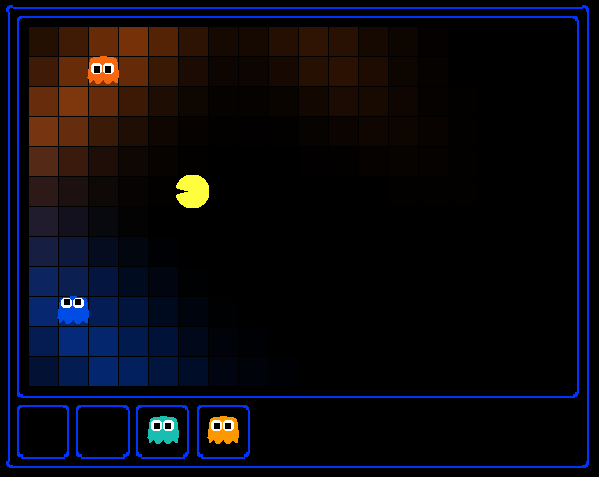
**Project 4: Ghostbusters**

Version 2.0. Last Updated: 24 Nov 2024.

Due: **See Canvas**



I can hear you, ghost.

Running won't save you from my

Particle filter!

**Introduction**

Pacman spends his life running from ghosts, but things were not always so. Legend has it that many years ago, Pacman’s great grandfather Grandpac learned to hunt ghosts for sport. However, he was blinded by his power and could only track ghosts by their banging and clanging.

In this project, you will design Pacman agents that use sensors to locate and eat invisible ghosts. You’ll advance from locating single, stationary ghosts to hunting packs of multiple moving ghosts with ruthless efficiency.

You are allowed to work with a partner on this project if you want to. Working with a partner is not permission to cheat and is not an excuse for ignorance of the material. You must write and submit your own code. Even if you study together, do not share code with your partner. You can discuss algorithms, compare results, help each other debug problems in their code, but do not give them code. And of course, copying code from the internet is cheating. Using an AI assistant as described in the syllabus is not cheating.

The code for this project contains the following files in your repository, assuming you cloned the repository as previously instructed in p0. Create a branch in your repository named **p4** for this assignment.

|  |  |
| --- | --- |
| **Files you'll edit:** | |
| bustersAgents.py | Agents for playing the Ghostbusters variant of Pacman. |
| inference.py | Code for tracking ghosts over time using their “sounds”. |
| **Files you will not edit:** | |
| busters.py | The main entry to Ghostbusters (replacing Pacman.py) |
| bustersGhostAgents.py | New ghost agents for Ghostbusters |
| distanceCalculator.py | Computes maze distances |
| game.py | Inner workings and helper classes for Pacman |
| ghostAgents.py | Agents to control ghosts |
| graphicsDisplay.py | Graphics for Pacman |
| graphicsUtils.py | Support for Pacman graphics |
| keyboardAgents.py | Keyboard interfaces to control Pacman |
| layout.py | Code for reading layout files and storing their contents |
| util.py | Utility functions |

**Files to Edit and Submit:** You will fill in portions of bustersAgents.py and inference.py during the assignment. Please *do not* change the other files in this distribution. Run the autograder on your files, commit the files to GitHub and push them to your remote repository. Put the link to your repository in the **p4** assignment submission in Canvas.

**Evaluation:** Your code will be autograded for technical correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder’s judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

**Academic Dishonesty:** Copying someone else’s code and submitting it as your own is asking for a grade you did not earn and claiming mastery of skills of which you have not demonstrated mastery. We may or may not use a plagiarism tool on your code in this class.

**Getting Help:** You are not alone, though we do expect you to know and practice basic problem-solving skills. If you find yourself stuck on something, contact the instructor or a classmate for help. Class time, Office Hours, Discord and Teams are there for your support; please use them. If you need to, set up an appointment for help. These projects should be rewarding and instructional, not frustrating and demoralizing—but I don’t know when or how to help unless you ask.

**Discord and Teams:** Please be careful not to post spoilers nor executable code.

**Ghostbusters and BNs**

In Ghostbusters, the goal is to hunt down scared but invisible ghosts. Pacman, ever resourceful, is equipped with sonar “ears” that provide noisy readings of the Manhattan Distance to each ghost. The game ends when Pacman has eaten all the ghosts. To start, try playing a game yourself using the keyboard.

python busters.py

The blocks of color indicate where each ghost could possibly be, given the noisy distance readings provided to Pacman. The noisy distances at the bottom of the display are always non-negative, and always within 7 of the true distance. The probability of a distance reading decreases exponentially with its difference from the true distance.

**Your primary task in this project is to implement inference to track the ghosts.** For the keyboard-based game above, a crude form of inference was implemented for you by default: all squares where a ghost could possibly be are shaded by the color of the ghost. Of course, we want a better estimate of the ghost’s position. Fortunately, Bayes Nets provide us with powerful tools for making the most of the information we have. Throughout the rest of this project, you will implement algorithms for performing both exact and approximate inference using Bayes Nets. The project is challenging, so we do encourage you to start early and seek help when necessary.

While watching and debugging your code with the autograder, it will be helpful to have some understanding of what the autograder is doing. There are 2 types of tests in this project, differentiated by their .test files found in the subdirectories of the test\_cases folder. For tests of class DoubleInferenceAgentTest, you will see visualizations of the **inference distributions** generated by your code, but all Pacman actions will be pre-selected according to the actions of the staff implementation. This is necessary to allow comparison of your distributions with the staff’s distributions. For GameScoreTest type tests, your BustersAgent will select actions for Pacman and you will watch your Pacman play and win games.

As you implement and debug your code, you may find it useful to run a single test at a time. Use the -t flag with the autograder to do this. For example, to run only the first test of question 1:

python autograder.py -t test\_cases/q1/1-ObsProb

In general, all test cases can be found inside test\_cases/q\*.

For this project, it is possible sometimes for the autograder to time out if running the tests with graphics. To accurately determine whether your code is efficient enough, you should run the tests with the --no-graphics flag. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

**Question 0 (0 points): DiscreteDistribution Class**

Throughout this project, we will be using the DiscreteDistribution class defined in inference.py to model belief distributions and weight distributions. This class customizes the built-in Python dictionary class as follows:

* the keys are the different discrete elements of our distribution
* the corresponding values are proportional to the belief or weight that the distribution assigns that element
* methods that you will implement here make the rest of your code more convenient.

This question asks you to fill in the missing parts of this class, which will be crucial for later questions even though this question itself is worth no points.

First, implement the normalize method. Recall that normalization converts a set of values into their equivalent probabilities: the values in the distribution to sum to 1, but their relative proportions remain the same. Use the total method to find the sum of the values in the distribution. For an empty distribution or a distribution where all the values are zero, do nothing. **These methods modify the distribution directly, rather than returning a new distribution.**

Second, implement the sample method. This method draws a sample from the distribution, where the probability that a key is sampled is proportional to its corresponding value. Assume that the distribution is not empty and contains at least one non-zero value. The distribution is not necessarily normalized prior to calling this method. You may find Python’s built-in random.random() function useful for this question.

There are no autograder tests for this question, but the correctness of your implementation can be easily checked. We have provided [Python doctests](https://docs.python.org/2/library/doctest.html) as a starting point, and you can feel free to add more and implement other tests of your own. You can run the doctests using:

python -m doctest -v inference.py

Depending on the implementation details of the sample method, some *correct* implementations *might not* pass the doctests that are provided. To thoroughly check the correctness of your sample method, you should instead draw many samples and see if the frequency of each key converges to be proportional of its corresponding value.

**Question 1 (2 points): Observation Probability**

In this question, you will implement the getObservationProb method in the InferenceModule base class in inference.py. This method takes in an observation--which is a noisy reading of the distance to the ghost—and Pacman’s position, the ghost’s position, and the position of the ghost’s jail, and returns the probability of the noisy distance reading given Pacman’s position and the ghost’s position. In other words, we want to return P(noisyDistance | pacmanPosition, ghostPosition).

The distance sensor has a probability distribution over distance readings given the true distance from Pacman to the ghost. This distribution is modeled by the function busters.getObservationProbability(noisyDistance, trueDistance), which returns P(noisyDistance | trueDistance) and is provided for you. Use this function to help you solve the problem. Use the provided manhattanDistance function to find the distance between Pacman’s location and the ghost’s location.

Jail is a special case that we must handle. When we capture a ghost and send it to the jail location, our distance sensor deterministically returns None. So, if the ghost’s position is the jail position, then the observation is None with probability 1 and anything else with probability 0. Conversely, if the distance reading is not None, then the ghost is in jail with probability 0. If the distance reading is None then the ghost is in jail with probability 1. You must handle this case in your implementation.

To test your code and run the autograder for this question:

python autograder.py -q q1

Some of the autograder tests may take a long time to run for this project—be patient. If the autograder doesn’t time out and your passes the tests, you should be fine.

**Question 2 (3 points): Exact Inference Observation**

In this question, you will implement the observeUpdate method in ExactInference class of inference.py to correctly update the agent’s belief distribution over ghost positions given an observation from Pacman’s sensors. You are implementing the **online belief update for observing new evidence**. The observeUpdate method should, for this problem, update the belief at every position on the map after receiving a sensor reading. You should iterate your updates over the variable self.allPositions which includes all legal positions plus the special jail position. Beliefs represent the probability that the ghost is at a particular location, and are stored as a DiscreteDistribution object in a field called self.beliefs, which you should update.

Before typing any code, write down the equation of the inference problem you are trying to solve. You should use the function self.getObservationProb that you wrote in the last question, which returns the probability of an observation given Pacman’s position, a potential ghost position, and the jail position. You can obtain Pacman’s position using gameState.getPacmanPosition(), and the jail position using self.getJailPosition().

In the Pacman display, high posterior beliefs are represented by bright colors, while low beliefs are represented by dim colors. You should start with a large cloud of belief that shrinks over time as more evidence accumulates. As you watch the test cases, be sure that you understand how the squares converge to their final coloring.

*Y*our busters agents have a separate inference module for each ghost they are tracking. If you print an observation inside the observeUpdate function, you’ll only see a single number even though there may be multiple ghosts on the board.

To run the autograder for this question and visualize the output:

python autograder.py -q q2

If you want to run this test (or any of the other tests) without graphics you can add the following flag:

python autograder.py -q q2 --no-graphics

As mentionedpreviously, the autograder *may* time out if running the tests with graphics. To accurately determine whether your code is efficient enough, run the tests with the --no-graphics flag. If the autograder passes with this flag set, then you will receive full points, even if the autograder times out with graphics.

**Question 3 (3 points): Exact Inference with Time Elapse**

In the previous question you implemented belief updates for Pacman based on his observations. Pacman’s observations are not his only source of knowledge about where a ghost may be. Pacman also has knowledge about the ways that a ghost may move. For example, a ghost cannot move through a wall, nor move more than one space in one time step.

To understand why this is useful to Pacman, consider a scenario in which there is Pacman and one Ghost. Pacman receives many observations that indicate the ghost is very near, and then one that indicates that the ghost is very far. The reading indicating the ghost is very far is likely to be the result of a buggy sensor. Pacman’s prior knowledge of how the ghost may move will decrease the impact of this reading since Pacman knows the ghost could not move so far in only one move.

In this question, you will implement the elapseTime method in ExactInference. The elapseTime step should, for this problem, update the belief at **every position on the map** after one time step passes. Your agent has access to the action distribution for the ghost through self.getPositionDistribution. To obtain the distribution over new positions for the ghost, given its previous position, use this line of code:

newPosDist = self.getPositionDistribution(gameState, oldPos)

Where oldPos refers to the previous ghost position. newPosDist is a DiscreteDistribution object, where for each position p in self.allPositions, newPosDist[p] is the probability that the ghost is at position p at time t + 1, given that the ghost is at position oldPos at time t. This call can be fairly expensive, so if your code is timing out, think about whether you can reduce the number of calls to self.getPositionDistribution.

Before typing any code, write down the equation of the inference problem you are trying to solve. To test your ***predict*** implementation separately from your **update** implementation in the previous question, this question will not make use of your update implementation.

Since Pacman is not observing a ghost, the ghost’s actions will not impact Pacman’s beliefs. Over time, 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 what Pacman already knows about their valid movements.

For the tests in this question, we will sometimes use a ghost with random movements and other times we will use the GoSouthGhost. This ghost tends to move *south* so over time, and without any observations, Pacman’s belief distribution should begin to focus on the bottom of the board. To see which ghost is used for each test case you can look in the .test files.

To run the autograder for this question and visualize the output:

python autograder.py -q q3

Recall that if you want to run any without graphics you can add the --no-graphics flag:

python autograder.py -q q3 --no-graphics

As mentioned previously, you may need this to accurately determine whether your code is efficient enough. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

As you watch the autograder output, remember that lighter squares indicate that Pacman believes a ghost is more likely to occupy that location, and darker squares indicate a ghost is less likely to occupy that location. For which of the test cases do you notice differences emerging in the shading of the squares? Can you explain why some squares get lighter and some squares get darker?

**Question 4 (2 points): Exact Inference Full Test**

Now Pacman knows how to use both his prior knowledge and his observations when figuring out where a ghost is. Now he is ready to hunt down ghosts on his own. This question will use your observeUpdate and elapseTime implementations together, along with a simple greedy hunting strategy that you will implement for this question. In the simple greedy strategy, Pacman assumes that each ghost is in its most likely position according to his beliefs, then moves toward the closest ghost. This is different than his previous strategy of moving by randomly selecting a valid action.

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.

To find the maze distance between any two positions pos1 and pos2, use self.distancer.getDistance(pos1, pos2). To find the successor position of a position after an action:

successorPosition = Actions.getSuccessor(position, action)

You are provided with livingGhostPositionDistributions, a list of DiscreteDistribution objects representing the position belief distributions for each of the ghosts that are still uncaptured.

If correctly implemented, your agent should win the game in q4/3-gameScoreTest with a score greater than 700 at least 8 out of 10 times. The autograder will check the correctness of your inference directly--the outcome of games is a reasonable “smoke test” of your implementation.

To run the autograder for this question and visualize the output:

python autograder.py -q q4

Use the --no-graphics flag if runtimes are too slow. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

**Question 5 (2 points): Approximate Inference Initialization and Beliefs**

Approximate inference is very trendy among ghost hunters this season. For the next few questions, you will implement a particle filtering algorithm for tracking a single ghost.

First, implement the functions initializeUniformly and getBeliefDistribution in the ParticleFilter class in inference.py. A particle (sample) is a ghost position in this inference problem. Note that, for initialization, particles should be evenly (not randomly) distributed across legal positions to ensure a uniform prior.

**Note that the variable you store your particles in must be a list.** A list is simply a collection of unweighted positions here. Storing your particles as any other data type, such as a dictionary, is incorrect and will produce errors. The getBeliefDistribution method then takes the list of particles and converts it into a DiscreteDistribution object.

To test your code and run the autograder for this question:

python autograder.py -q q5

**Question 6 (3 points): Approximate Inference Observation**

Next, we will implement the observeUpdate method in the ParticleFilter class in inference.py. This method constructs a weight distribution over self.particles where the weight of a particle is the probability of the observation given Pacman’s position and that particle location. Then, we **resample** from this weighted distribution to construct our new list of particles.

Reuse the function self.getObservationProb to find the probability of an observation given Pacman’s position, a potential ghost position, and the jail position. The sample method of the DiscreteDistribution class will also be useful. Recall that you can obtain Pacman’s position using gameState.getPacmanPosition(), and the jail position using self.getJailPosition().

**There is one special case that a correct implementation must handle.** When all particles receive zero weight, the list of particles should be reinitialized by calling initializeUniformly. The total method of the DiscreteDistribution may be useful.

To run the autograder for this question and visualize the output:

python autograder.py -q q6

Use the --no-graphics flag if runtimes get slow. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

**Question 7 (3 points): Approximate Inference with Time Elapse**

Implement the elapseTime function in the ParticleFilter class in inference.py. This function should construct a new list of particles that corresponds to each existing particle in self.particles advancing a time step, and then assign this new list back to self.particles. When complete, you should be able to track ghosts nearly as effectively as with exact inference.

For this question, we will test both the elapseTime function in isolation and test the full implementation of the particle filter combining elapseTime and observe. As in the elapseTime method of the ExactInference class, you should use:

newPosDist = self.getPositionDistribution(gameState, oldPos)

This line of code obtains the distribution over new positions for the ghost, given its previous position (oldPos). The sample method of the DiscreteDistribution class will also be useful.

To run the autograder for this question and visualize the output:

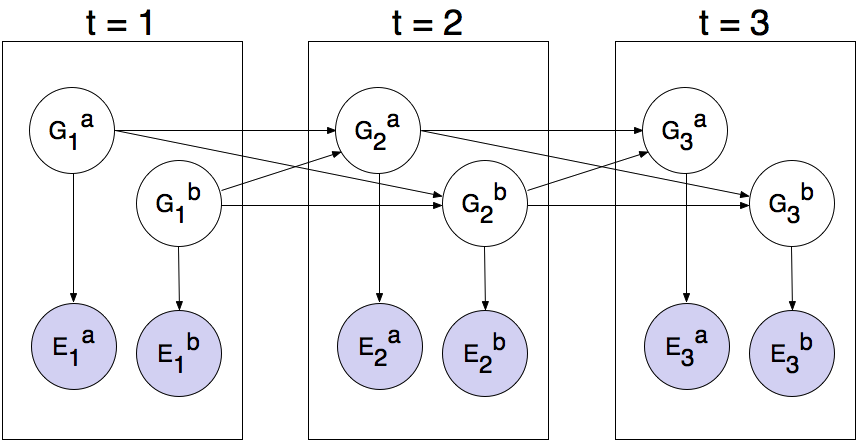
python autograder.py -q q7

Use the --no-graphics flag if runtimes get slow. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

**Question 8 (1 points): Joint Particle Filter Observation**

So far, we have tracked each ghost independently, which works fine for the default RandomGhost or more advanced DirectionalGhost. DispersingGhost however, chooses actions that avoid other ghosts. Since the ghosts’ transition models are no longer independent, all ghosts must be tracked jointly in a dynamic Bayes net!

The Bayes net has the following structure, where the hidden variables represent ghost positions and the emission variables are the noisy distances to each ghost. This structure can be extended to more ghosts, but only two, and , are shown below.



You will now implement a particle filter that tracks multiple ghosts simultaneously. Each particle will represent a tuple of ghost positions that is a sample of where all the ghosts are at the present time. The code is already set up to extract marginal distributions about each ghost from the joint inference algorithm you will create, so that **belief clouds** about individual ghosts can be displayed.

Complete the initializeUniformly method in JointParticleFilter in inference.py. Your initialization should be consistent with a uniform prior. You may find the Python itertools package helpful. Look at itertools.product to get an implementation of the Cartesian product. If you use this, the permutations are not returned in a random order, so must then shuffle the list of permutations to ensure a desirable placement of particles across the board.

Reuse self.legalPositions to obtain a list of positions a ghost may occupy. Recall that **the variable you store your particles in must be a list**.

To run the autograder for this question and visualize the output:

python autograder.py -q q8

Use the --no-graphics flag if runtimes get slow. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

**Question 9 (3 points): Joint Particle Filter Observation**

In this question, you will complete the observeUpdate method in the JointParticleFilter class of inference.py. A correct implementation will weight and resample the entire list of particles based on the observation of all ghost distances.

To loop over all the ghosts, use:

for i in range(self.numGhosts):

...

You can still obtain Pacman’s position using gameState.getPacmanPosition(), but to get the jail position for a ghost, use self.getJailPosition(i), since now there are multiple ghosts each with their own jail positions.

**Your implementation should also again handle the special case when all particles receive zero weight.** In this case, self.particles should be recreated from the prior distribution by calling initializeUniformly.

As you should have done in the update method for the ParticleFilter class, use the function self.getObservationProb to find the probability of an observation given Pacman’s position, a potential ghost position, and the jail position. The sample method of the DiscreteDistribution class will also be useful.

To run the autograder for this question and visualize the output:

python autograder.py -q q9

Use the --no-graphics flag if runtimes get slow. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

**Question 10 (3 points): Joint Particle Filter Time Elapse and Full Test**

Complete the elapseTime method in JointParticleFilter in inference.py to resample each particle correctly for the Bayes net. Each ghost should draw a new position conditioned on the positions of all the ghosts at the previous time step.

As in the last question, you can loop over the ghosts using:

for i in range(self.numGhosts):

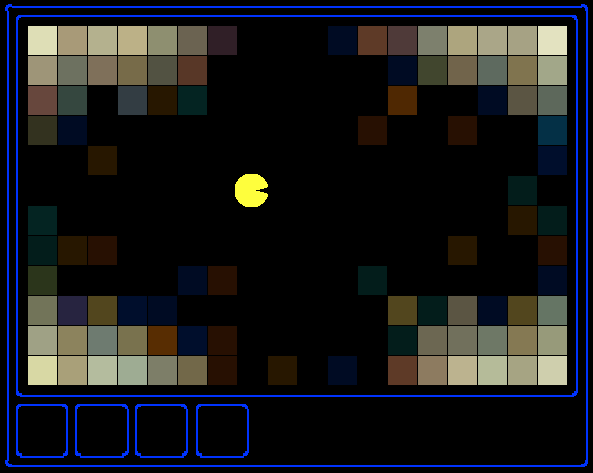
...

Given that i refers to the index of the ghost, to obtain the distributions over new positions for that single ghost, and given the list prevGhostPositions of previous positions of all the ghosts, use:

newPosDist = self.getPositionDistribution(gameState, prevGhostPositions, i, self.ghostAgents[i])

Completing this question involves grading both question 9 and question 10. These questions involve joint distributions, so they require more time and computational power to grade, so please be patient!

As you run the autograder note that q10/1-JointParticlePredict and q10/2-JointParticlePredict test your predict implementations only, and q10/3-JointParticleFull tests both your predict and update implementations. Notice the difference between test 1 and test 3. In both tests, Pacman knows that the ghosts will move to the sides of the gameboard. What is different between the tests, and why?



To run the autograder for this question and visualize the output:

python autograder.py -q q10

Use the --no-graphics flag if runtimes get slow. If the autograder passes with this flag, then you will receive full points, even if the autograder times out with graphics.

**Submission**

To submit your project, run python autograder.py on your solution, then zip all the files as instructed above and submit the zip file to the Project 4 assignment in Canvas.

Check with your instructor on where and how code files are to be submitted if different from what is written in this document.

If you work with a partner:

• In Canvas submission, please specify any partner you may have worked with.

• Each person must make their own submission, with their own code that they wrote.