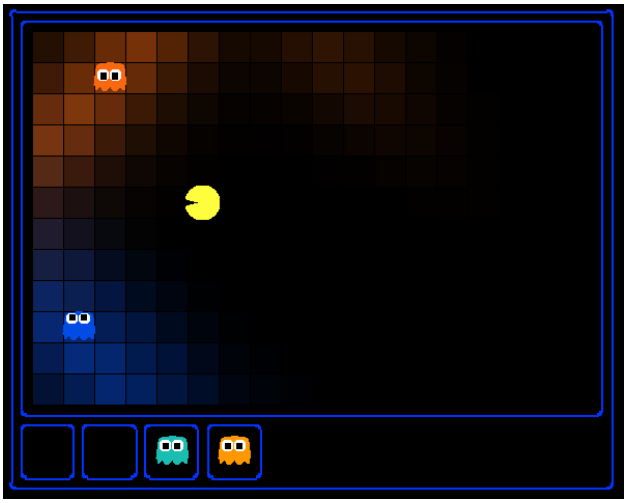
Project 4: Ghostbusters (Due on November 21st)

(Thanks to John DeNero and Dan Klein!)



I can hear you, ghost.
Running won't save you from my
Particle filter!

Note to all students

This is your last fully graded project. So, it will be your most challenging project! and you don't have time for it! So, we do encourage you to start early and seek help when necessary.

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.

The code for this project contains the following files, available on blackboard and can be downloaded from here.

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 should 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

What to submit:

You will fill in portions of bustersAgents.py and inference.py during the assignment. You should submit these files with your code and comments to blackboard. Please do not change the other files in this distribution or submit any of our original files other than multiAgents.py. **If you work with a partner, please write their name in a file named partner.txt.**

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. If your code works correctly on one or two of the provided examples but doesn't get full credit from the autograder, you most likely have a subtle bug that breaks one of our more thorough test cases; you will need to debug more fully by reasoning about your code and trying small examples of your own. That said, bugs in the autograder are not impossible, so please do contact the staff if you believe that there has been an error in the grading.

Academic Integrity: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours, section, and Piazza are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Ghostbusters and BNs

In this version of Ghostbusters, the goal is to hunt down scared but invisible ghosts. Pacman, ever resourceful, is equipped with sonar (ears) that provides 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 Pac-Man. 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. A crude form of inference is implemented for you by default: all squares in which a ghost could possibly be shaded by the color of the ghost. Option -s shows where the ghost actually is.

```
python busters.py -s -k 1
```

Naturally, we want a better estimate of the ghost's position. We will start by locating a single, stationary ghost using multiple noisy distance readings. The default BustersKeyboardAgent in bustersAgents.py uses the ExactInference module in inference.py to track ghosts. The project is challenging, so we do encourage you to start early and seek help when necessary.

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 is an extension of the built-in Python dictionary class, where the keys are the different discrete elements of our distribution, and the corresponding values are proportional to the belief or weight that the distribution assigns that element. The original Pacman set of projects provided by Berkley ask you to implement the

normalization and sampling methods. In here, we have done these implementations for you. We are just asking that you look at them to make sure you understand how they are done and how they can help you for the rest of the project.

Question 1 (6 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), 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. i.e., please 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. You should use this function to help you solve the problem, and use the provided **manhattanDistance** function to find the distance between Pacman's location and the ghost's location.
However, there is the special case of jail that we have to handle as well. Specifically, when we capture a ghost and send it to the jail location, our distance sensor deterministically returns *None*, and nothing else. So, if the ghost's position is the jail position, then the observation is *None* with probability 1, and everything 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. Make sure you handle this special case in your implementation.
To test your code and run the autograder for this question:

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

Notes:

1. It is possible for some of the autograder tests to take a long time to run for this project, and you will have to exercise patience. As long as the autograder doesn't time out, you should be fine (provided that you actually pass the tests).
2. The function returns a probability in floating point so make sure that whatever you return is floated by using float() function.

Question 2 (8 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 observe 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.

Note: your busters' agents have a separate inference module for each ghost they are tracking. That's why if you print an observation inside the update 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
```

***IMPORTANT*:** In general, it is possible sometimes for the autograder to time out if running the tests with graphics. To accurately determine whether or not 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 3 (8 points): Exact Inference with Time Elapse

In the previous question you implemented belief updates for Pacman based on his observations. Fortunately, 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; namely that the ghost can not move through a wall or more than one space in one time step.

To understand why this is useful to Pacman, consider the following scenario in which there is Pacman and one Ghost. Pacman receives many observations which indicate the ghost is very near, but then one which indicates 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 elapsing. Your agent has access to the action distribution for the ghost through **self.getPositionDistribution**. In order 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.* Note that this call can be fairly expensive, so if your code is timing out, one thing to think about is whether or not 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. In order 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 the ghost, this means 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 around 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
```

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 q3 --no-graphics
```

***IMPORTANT*:** In general, it is possible sometimes for the autograder to time out if running the tests with graphics. To accurately determine whether or not 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 4 (6 points): Exact Inference Full Test

Now that Pacman knows how to use both his prior knowledge and his observations when figuring out where a ghost is, 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 which 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. Up to this point, Pacman has moved 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.

Note: the autograder will also check the correctness of your inference directly, but the outcome of games is a reasonable sanity check.

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

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

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 q4 --no-graphics
```

***IMPORTANT*:** In general, it is possible sometimes for the autograder to time out if running the tests with graphics. To accurately determine whether or not 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 5 (6 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 i**nitializeUniformly** 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 in order 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 variables (positions in this case). 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 (8 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.

You should again 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. As a reminder, 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
```

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 q6 --no-graphics
```

***IMPORTANT*:** In general, it is possible sometimes for the autograder to time out if running the tests with graphics. To accurately determine whether or not 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 7 (8 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.

Note that in this question, we will test both the **elapseTime** function in isolation, as well as 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
```

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 q7 --no-graphics
```

Note that even with no graphics, this test may take several minutes to run.

***IMPORTANT*:** In general, it is possible sometimes for the autograder to time out if running the tests with graphics. To accurately determine whether or not 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.

****

Congratulations! By finishing this project, you have almost finished the Pac-Man projects, a set of projects that were developed for UC Berkeley's introductory artificial intelligence course. Hope you are not hating the game by now. You should be proud of yourself and how capable you have made automated Pacman. You are an AI techniques expert by now!

Your projects grade is finished by now. Only one optional project is left which will have extra marks to help you with your grades if needed (or just for fun if you like machine learning).

Well done CS5368 students!

****