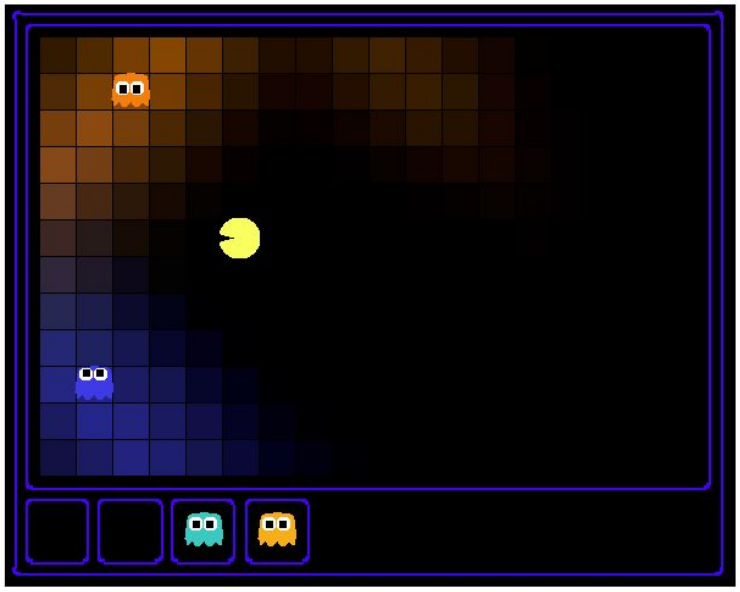
CS 386: Lab Assignment 10

(TA in charge: Anand Dhoot)

Acknowledgement: This lab assignment is based on Project 4: Ghostbusters, which is a part of a recent offering of CS188 at UC Berkeley. The code and resources provided here are almost entirely drawn from the Berkeley project. We thank the authors at Berkeley for making their project available to the public.

We continue where we left off last week: implementing inference to identify the (approximate) location of ghosts in the game of Ghostbusters. In this assignment, we will focus on implementing algorithms for performing inference using Particle Filters and Dynamic Bayes Nets.



Code

The base code for this assignment is available in this zip file. Here is the list of files present in the `tracking` directory.

Files you will edit	
inference.py	Code for tracking ghosts over time using their sounds

Files you must not edit	
busters.py	The main entry to Ghostbusters (replacing Pacman.py)
bustersAgents.py	Agents for playing the Ghostbusters variant of Pacman
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

Task 0: Introduction to Ghostbusters (Ungraded)

As described in Lab Assignment 9, in this version of Ghostbusters, the goal is to hunt down scared but invisible ghosts. The Pacman 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. Your primary task in this assignment is to implement inference (identifying the possible squares where ghosts could be located) to track the ghosts.

Note

- This assignment is relatively long. Make sure you keep moving along. If you haven't finished tasks 1 and 2 at the end of 90 minutes, you might not be able to finish all four tasks in the allotted 180 minutes.
- Most of the boilerplate code has already been written for you so that you can focus on the crux of probabilistic inference. Go through the comments before writing code in functions specified for detailed descriptions of built-in functions and other details about the code.
- 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 two types of tests in this project, as differentiated by their `*.test` files found in the subdirectories of the `test_cases` folder. For tests of class `DoubleInferenceAgentTest`, your will see visualisations of the inference distributions generated by your code, but all Pacman actions will be preselected according to the actions of our implementation. This is necessary in order to allow comparison of your distributions with our distributions. The second type of test is `GameScoreTest`, in which your `BustersAgent` will actually 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. In order to do this you will need to use the -t flag with the autograder. For example if you only want to run the first test of Task 1, use the following command.

  ```
  python autograder.py -t test_cases/q1/1-ParticleObserve
  ```

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

Descriptions of Solutions

You will be evaluated on four tasks as a part of this assignment. In each case your code will have to clear the tests presented by the autograder. Additionally, create a text file called `descriptions.txt` and fill it with an informative description of the approach you used to solve each task. We need to be assured that you have understood the underlying concept, and are not merely going through the motions of "applying a formula". If your descriptions are not clear, you may lose marks for the corresponding tasks.

Task 1: Approximate Inference Observation (4 marks)

In this task, you will implement a particle filtering algorithm for tracking a single ghost.

Implement the functions `initializeUniformly`, `getBeliefDistribution`, and `observe` for the `ParticleFilter` class in `inference.py`. A correct implementation should also handle two special cases.

1. When all your particles receive zero weight based on the evidence, you should resample all particles from the prior to recover.
2. When a ghost is eaten, you should update all particles to place that ghost in its prison cell, as described in the comments of `observe`. When complete, you should be able to track ghosts nearly as effectively as with exact inference.

Run the autograder for this question as follows and visualise the output.

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

Hints

- A particle (sample) is a ghost position in this inference problem.
- The belief cloud generated by a particle filter will look noisy compared to the one for exact inference (the one in the previous assignment).
- `util.sample` or `util.nSample` will help you obtain samples from a distribution. It is preferable that you use `util.sample`. If, however, your implementation is timing out, try using `util.nSample`, which needs to be called in a slightly different way from `util.sample` (see `util.py` for its definition).

Task 2: Approximate Inference with Time Elapse (3 marks)

Implement the `elapseTime` function for the `ParticleFilter` class in `inference.py`. 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`.

Run the autograder for this question and visualise the output.

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

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. As an example, you can run

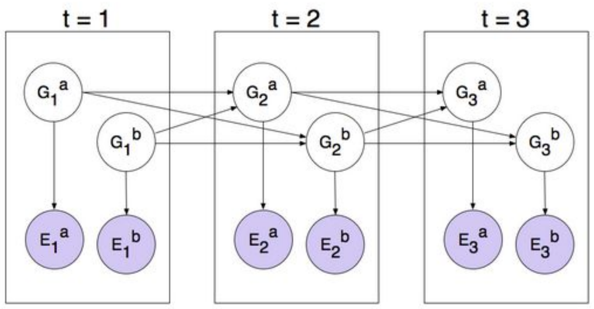
```
python autograder.py -t test_cases/q2/2-ParticleElapse
```

and observe that the distribution becomes concentrated at the bottom of the board.

Task 3: Joint Particle Filter Observation (5 marks)

So far, we have tracked each ghost independently, which works fine for the default `RandomGhost` or more advanced `DirectionalGhost`. However, the prized `DispersingGhost` 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 G represent ghost positions and the emission variables E are the noisy distances to each ghost. This structure can be extended to more ghosts, but only two (a and b) 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 `initializeParticles`, `getBeliefDistribution`, and `observeState` methods in `JointParticleFilter` to weight and resample the whole list of particles based on new evidence. As before, a correct implementation should also handle two special cases.

1. When all your particles receive zero weight based on the evidence, you should resample all particles from the prior to recover.
2. When a ghost is eaten, you should update all particles to place that ghost in its prison cell, as described in the comments of `observeState`.

You should now effectively track dispersing ghosts. To run the autograder for this question and visualize the output, use this command.

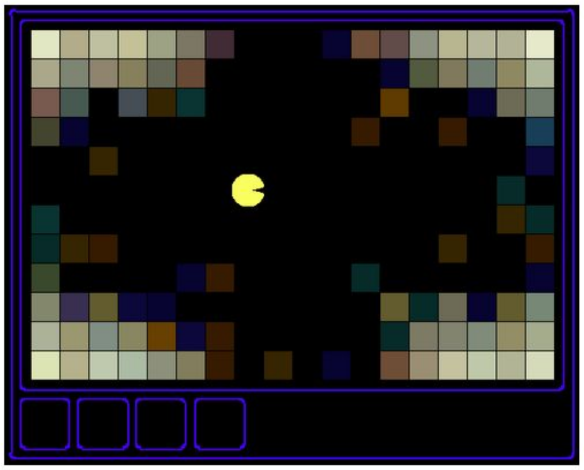
```
python autograder.py -q q3
```

Task 4: Joint Particle Filter with Elapse Time (3 marks)

Complete the `elapseTime` method in `JointParticleFilter` in `inference.py` to resample each particle correctly for the Bayes Net. In particular, each ghost should draw a new position conditioned on the positions of all the ghosts at the previous time step. The comments in the method provide instructions for support functions to help with sampling and creating the correct distribution.

Note that these questions involve joint distributions and they require more computational power (and time) to grade, so please be patient!

As you run the autograder note that `q4/1-JointParticleElapse` and `q4/2-JointParticleElapse` test your `elapseTime` implementations only, and `q4/3-JointParticleElapse` tests both your `elapseTime` and `observe` 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 use this command.

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