AI Challenge 5: Ghostbusters

COSC4550/COSC5550
Artificial Intelligence
University of Wyoming

1 Overview

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.

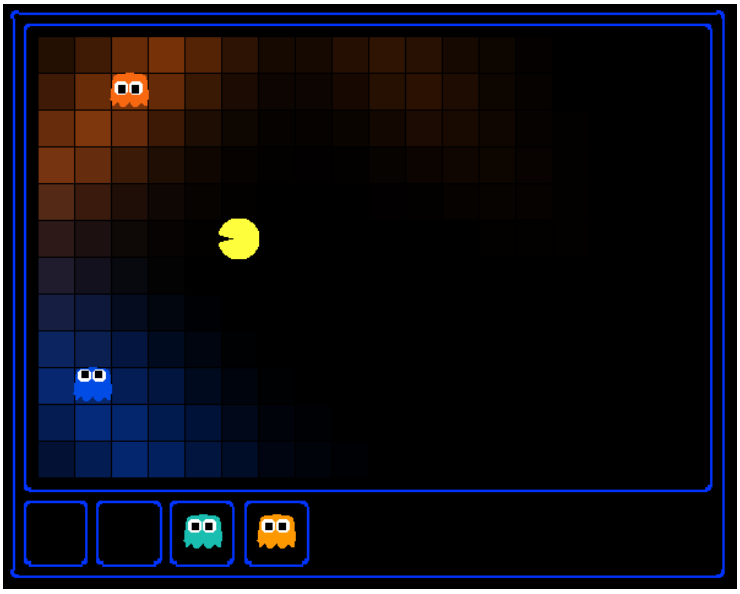

Figure 1: The full Pacman game.

Acknowledgment: This assignment is based on the one created by Dan Klein and John DeNero given as part of Berkeley's CS188 course. This assignment was also inspired by the modifications made by Peter Stone in his CS343 course of 2012. We thank Dan and John for creating the assignment and granting the permission to use it and we thank Peter for the ideas on how to adapt the assignment for this course.

1.1 Chapters

Chapters are from the book 'Artificial Intelligence, A Modern Approach', third edition, by Stuart Russel and Peter Norvig. The relevant chapter for this challenge is chapter 15, sections 1, 2, and 5. The most relevant sub-sections are 15.1.2 (Transition and senor models), 15.2.1 (Explains filtering for exact inference), and 15.5.3 (Explains particle filtering). Chapters 13 and 14 may help in understanding the material presented in chapter 15.

1.2 Program files

The archive for this challenge contains a number of Python files. These files have been tested on, and should work with, Python version 2.7.3 and Python version 2.6.6. More recent versions of Python 2.x probably work as well, but these files do not work with Python 3.x.

The code base is once again very similar to previous challenges, and it is still unwise to intermingle these files with the files from previous challenges. The important files for this challenge are:

student_inference.py	This file should be extended with your implementation of the various inference algorithms, and your greedy buster agent.
busters.py	The main entry to Ghostbusters (replacing pacman.py)
game.py	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
util.py	Utility functions.
autograder.py	Use the autograder to check the correctness of your solutions.

While working 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, as 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 preselected by the autograder. This is necessary in order to allow comparison of your distributions for grading. 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 question 1-a, use:

```
python autograder.py -t test_cases/q1a/1-ExactObserve
```

In general, all test cases can be found inside *test_ cases/q**. To speed up some of the tests you can use the option `--no-graphics` to run the test without graphics.

1.3 Deliverables

For this challenge you should submit your versions of *student_ inference.py*. Please rename this file to *[yourname]_ inference.py* before submitting it.

Important: Make absolutely sure that your implementation will run all questions without any modifications being necessary on our part. It should run on either python 2.7.3 or python 2.6.6 and when in doubt you can always test your implementation on hive. You will only receive partial credit for implementations that do not run.

2 Questions

The goal of this game 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 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. A crude form of inference is implemented for you by default: all squares in which a ghost could possibly be are shaded by the color of the ghost. Option `-s` shows where the ghost actually is.

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

2.1 Question 1-a (3 points): Exact Inference Observation

Naturally, we want a better estimate of the ghost's position. In this question, you will update the agent's beliefs according to the data from his sensor. Note that you do not have to handle the transition model yet, the transition model will be handled in question 1-b.

For this question you'll have to update the observe method in *ExactInference* class of *student_ inference.py* to correctly update the agent's belief distribution over ghost positions given an observation from Pacman's sensors. A correct implementation should also handle one special case: when a ghost is eaten, you should update the beliefs of the agent to belief that, with certainty, the ghost is now in its prison. See the comments in observe for how to do this.

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

```
python autograder.py -q q1a
```

As you watch the test cases, be sure that you understand how the squares converge to their final coloring. In test cases where Pacman is boxed in (which is to say, he is unable to change his observation point), why does Pacman sometimes have trouble finding the exact location of the ghost?

2.1.1 Hints and Observations

- Figuring out what to write is the challenge of this question, but once you have the answer your code should be very short.
- Before typing any code, write down the equation of the inference problem you are trying to solve.
- Try printing *noisyDistance*, *observationDistribution*, and *pacmanPosition* (in the observe function) to get started.
- The effect of the *observe* function should be to update the *self.beliefs* field of the inference module. This field is a *util.Counter* object (like a dictionary) that holds the agents current beliefs about the world (e. g. X_t).
- Before any readings, Pacman believes the ghost could be anywhere: a uniform prior (see *initializeUniformly*).
- 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.
- Note that, while there is no transition model for this question, the agents new beliefs still depend on his current beliefs.
- An important function is:

  ```
  busters.getObservationDistribution(noisyDistance)
  ```

  This function returns a map between true distances and their likelihood given the provided noisy distance, which can be used to obtain all probabilities in the form of $\mathbf{P}(noisyDistance|trueDistance)$. In general this function describes the complete sensor model $\mathbf{P}(E_t|X_t)$. See the comments in the *observe* function for suggestions on how to handle this data.

- Your busters agents have a separate inference module for each ghost they are tracking. That's why if you print an observation inside the observe function, you'll only see a single number even though there may be multiple ghosts on the board.

2.1.2 Grading: 3 points

To get full credit your implementation has to be correct and pass all tests of question 1-a on the autograder. Note that the autograder does not only check if your algorithm is correct, it will also time your algorithm, and your implementation may not take more than 5 minutes to answer a question or a time-out exception will be thrown. If your algorithm fails on one or more tests you'll get partial credit depending on the number of tests you passed.

2.2 Question 1-b (4 points): Inference over time

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 timestep.

In this question you will update Pacman's beliefs based on the transition model. You will have to implement this in the *elapseTime* method in *ExactInference*. Your agent has access to the action distribution for any *GhostAgent* (see the Hints and Observations section and the comments in the code for how to access this distribution).

To run the autograder for question 1-b only:

```
python autograder.py -q q1b
```

Note that, in this case, Pacman is not utilizing his observations about the ghost. Therefore Pacman will start with a uniform distribution over all spaces, and then update his beliefs according to how he knows the Ghost is able to move. 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.

2.2.1 Hints and Observations

- 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.

- We assume that ghosts move independently of one another, so while you can develop all of your code for one ghost at a time, adding multiple ghosts should still work correctly.
- An important combination of functions is:

  `self.getPositionDistribution(self.setGhostPosition(gameState, oldPos))`

  This will return the position distribution of an agent given his previous position *oldPos*. That is, these functions return a map containing $\mathbf{P}(newPos|oldPos)$ for each possible new position and in general these functions describe the complete transition model $\mathbf{P}(X_t|X_{t-1})$. See the comments in the *elapseTime* function for hints on how to use this distribution.

2.2.2 Grading: 4 points

To get full credit your implementation has to be correct and pass all tests of question 1-b on the autograder. If your algorithm fails on one or more tests you'll get partial credit depending on the number of tests you passed.

2.3 Question 1-c (4 points)

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 *observe* 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 its 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 *student_ inference.py*. Your agent should first find the most likely position of each remaining (uncaptured) ghost, then choose an action that minimizes the distance to the closest ghost. If correctly implemented, your agent should win the game in *q3/3-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 q1c
```

Note: when you run with graphics you will almost certainly run into a time-out exception in the `3-gameScoreTest.test`. This is just because of the slow graphics, run with `--no-graphics` to pass this final test.

2.3.1 Hints and Observations

- Because this question depends on your Exact Inference working correctly, the autograder will start testing your Exact Inference first. Remember that you can use:
  `python autograder.py -t test_cases/q1c/3-gameScoreTest`
  to skip the Exact Inference tests, and immediately start testing your greedy agent.
- When correctly implemented, your agent will thrash around a bit in order to capture a ghost.
- The comments of chooseAction provide you with useful method calls for computing maze distance and successor positions.
- Make sure to only consider the living ghosts, as described in the comments.

2.3.2 Grading: 4 points

To get full credit your implementation has to be correct and pass all tests of question 1c on the autograder. If your algorithm fails on one or more tests you'll get partial credit depending on how many tests you passed.

2.4 Question 2-a (5 points): Approximate Inference

Approximate inference is very trendy among ghost hunters this season. Next, you will implement a particle filtering algorithm for tracking a single ghost. Note that, once again, this question will not use the transition model yet, the transition model will be handled in question 2-b.

Implement the functions *initializeUniformly*, *getBeliefDistribution*, and *observe* for the *ParticleFilter* class in *student_ inference.py*. *initializeUniformly* should initialize a uniform, but *not* random, distribution of samples. *getBeliefDistribution* should use the current samples to calculate and return a belief distribution similar to the one used in question 1. *observe* should produce a new set of samples based on the old set of samples and the current observation.

A correct implementation should also handle two special cases. (1) When all your particles receive zero weight based on the evidence, you should re-sample all particles from the prior to recover. (2) When a ghost is eaten, you should update all particles such that the agent will believe that the ghost is in its prison cell, as described in the comments of *observe*.

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

```
python autograder.py -q q2a
```

2.4.1 Hints and Observations

- 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.

2.4.2 Grading: 5 points

To get full credit your implementation has to be correct and pass all tests of question 2-a on the autograder. If your algorithm fails on one or more tests you'll get partial credit depending on the number of tests you passed.

2.5 Question 2-b (4 points): Approximate Inference with Time Elapse

Implement the *elapseTime* function for the *ParticleFilter* class in *student_ inference.py*. This function should create a new sample based on the old sample and Pacman's knowledge on how Ghosts move. 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*.

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

```
python autograder.py -q q2b
```

2.5.1 Hints and Observations

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

  ```
  python autograder.py -t test_cases/q2b/2-ParticleElapse
  ```

- If you have implemented all functions correctly you should now be able to test your agents in the following games. Use this command to run the game interactively:

  ```
  python busters.py -k 1 -s -a inference=ParticleFilter
  ```

  Or use this command to let the greedy buster agent play the game:

  ```
  python busters.py -p GreedyBustersAgent -l oneHunt -k 1 -n 10 -s -a inference=ParticleFilter
  ```

2.5.2 Grading: 4 points

To get full credit your implementation has to be correct and pass all tests of question 2-b on the autograder. If your algorithm fails on one or more tests you'll get partial credit depending on the number of tests you passed.

2.6 Question 3-a (COSC5550 5 pts., COSC4550 2 pts. bonus): Joint Particle Filter Observation

(For those enrolled in the graduate student version of the class only. Bonus points for undergraduates)

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.

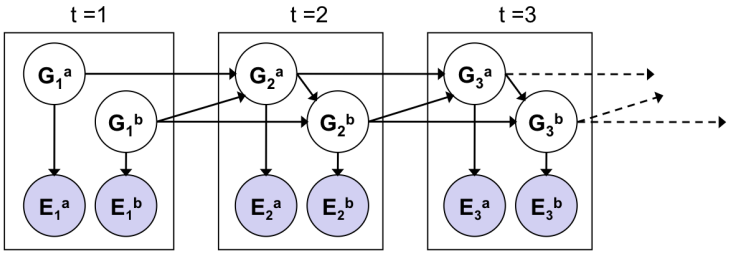

Figure 2: The Dynamic Bayesian Network for the Joint Particle Filter

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.

Once again, this question will only handle the sensor model, question 3-b will handle the transition model. Complete the *initializeParticles*, *getBeliefDistribution*, and *observeState* methods in *JointParticleFilter* to weight and re-sample the whole list of particles based on new evidence. These functions should work the same as the functions in question 2-a except that they should now work on samples that are tuples instead of singles.

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 re-sample 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:

```
python autograder.py -q q3a
```

2.6.1 Grading: 5 points

To get full credit your implementation has to be correct and pass all tests of question 3-a on the autograder. If your algorithm fails on one or more tests you'll get partial credit depending on the number of tests you passed. If you are enrolled in the undergraduate version of this course your implementation needs to pass all tests of question 3-a on the autograder to get the 2 point bonus, partial bonus is not awarded.

2.7 Question 3-b (COSC5550 3 pts., COSC4550 1 pt. bonus): Joint Particle Filter with Elapse Time

(For those enrolled in the graduate student version of the class only. Bonus points for undergraduates)

Complete the *elapseTime* method in *JointParticleFilter* in *student_ inference.py* to re-sample each particle correctly from 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.

To run the autograder for this question use:

```
python autograder.py -q q3b
```

2.7.1 Hints and Observations

- The *q3b/1-JointParticleElapse* and *q3b/2-JointParticleElapse* tests check only your *elapseTime* implementations while the *q3b/3-JointParticleElapse* tests both your *elapseTime* and *observe* implementations.
- Since these questions involve joint distributions, they require more computational power (and time) to grade, so please be patient!

2.7.2 Grading: 3 points

To get full credit your implementation has to be correct and pass all tests of question 3-b on the autograder. If your algorithm fails on one or more tests you'll get partial credit depending on the number of tests you passed. If you are enrolled in the undergraduate version of this course your implementation needs to pass all tests of question 3-b on the autograder to get the 1 point bonus, partial bonus is not awarded.

3 FAQ

Q: *For exact inference, what should the new believes look like after Pacman ate a ghost?*

A: The new believes should have a 1.0 (100%) at the jail position, and 0 everywhere else; we are certain the ghost is in jail.

Q: *My code runs extremely slow with the autograder, what is wrong?*

A: First, make sure you use the `--no-graphics` command. Second, check your code for slow operations, especially for operations that get slower over time. For example, make sure that the `self.particles` list does not increase in size over time.