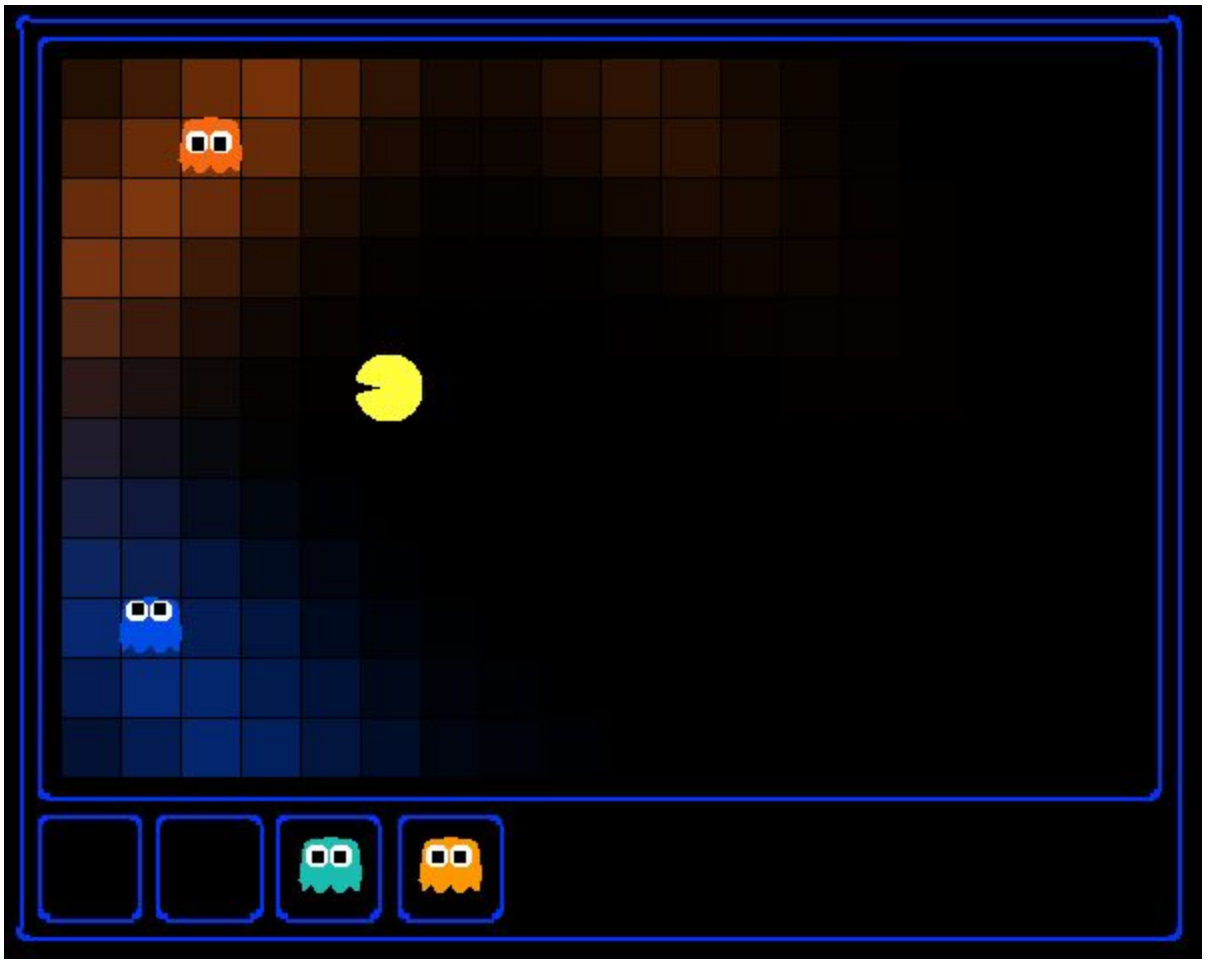
Homework 2: Probabilistic Reasoning



Instructions

1. For this assignment you may form teams of up to two people.
2. Homework is in Python 2.7
3. **Each of the partners must submit the homework**. We understand that they are identical, it's just the way Blackboard software works.
4. The final submission must be structured like this:
    a. FirstName_LastName_hw#.pdf - report, named after you - doing so helps us if we download all the submissions to a directory. Example: Dainis_Boumber_hw1.pdf
    b. FirstName_LastName_resources.zip - code goes here. Do not use other formats such as 7z or tar.xz for compatibility).
    Example: Dainis_Boumber_resources.zip

5. Your homework report must be formatted in a style not too different from a standard scientific conference or journal. The submission must be in single column format. The file should be in pdf format. Use figures and tables when needed to explain your results.
6. Do **NOT** not just give a number or say yes or no. Show work. Explain in detail.

Download: tracking.zip from

https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/tracking/v1/001/tracking.zip

Files to Edit and Submit: fill in portions of bustersAgents.py and inference.py.

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.**

Autograder To run the autograder, run the command:

python autograder.py

Note that the autograder will skip any question that raises *util.raiseNotDefined()*. You can also select individual questions for the autograder to run. For example, if you want to run question 2 on a project, you would run the command:

python autograder.py -q q2

Setup

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 the 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 in which a ghost could possibly be are shaded by the color of the ghost. Naturally, 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 lab 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, as differentiated by their *.test files found in the subdirectories of the test_cases folder. For tests of class DoubleInferenceAgentTest, your will see visualizations of the inference distributions generated by your code, but all Pacman actions will be preselected according to the actions of the staff implementation. This is necessary in order to allow comparison of your distributions with the staff's 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 question 1, use:

python autograder.py -t test_cases/q1/1-ExactObserve

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

Question 1: Inference Based on Observations (30 pts)

In this question, you will update the observe 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. A correct implementation should also handle one special case: when a ghost is eaten, you should place that ghost in its prison cell, as described in the comments of observe.

As you watch the test cases, be sure that you understand how the squares converge to their final coloring. In test cases where is Pacman 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?

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 observe function, you'll only see a single number even though there may be multiple ghosts on the board.

Hints:

- You are implementing the online belief update for observing new evidence. Before any readings, Pacman believes the ghost could be anywhere: a uniform prior (see initializeUniformly). After receiving a reading, the observe function is called, which must update the belief at every position.
- Before typing any code, write down the equation of the inference problem you are trying to solve.
- Try printing noisyDistance, emissionModel, and PacmanPosition (in the observe function) to get started.
- 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.
- Beliefs are stored as util.Counter objects (like dictionaries) in a field called self.beliefs, which you should update.
- You should not need to store any evidence. The only thing you need to store in ExactInference is self.beliefs.

Question 2: Inference based on Priors (30 pts)

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.

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. Your agent has access to the action distribution for any GhostAgent. In order to test your elapseTime implementation separately from your observe implementation in the previous question, this question will not make use of your observe implementation.

Since Pacman is not utilizing any observations about the ghost, this means that 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.

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 q2

As an example of the GoSouthGhostAgent, you can run

python autograder.py -t test_cases/q2/2-ExactElapse

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

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?

Hints:

- Instructions for obtaining a distribution over where a ghost will go next, given its current position and the gameState, appears in the comments of ExactInference.elapseTime in inference.py.
- We assume that ghosts still 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.

Question 3: Full Test (40 pts)

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 bustersAgents.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 q3

Note: 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

Hints:

- 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.