Introduction

In this project, you will implement inference algorithms for Bayes Nets, specifically variable elimination and value-of-perfect-information computations. These inference algorithms will allow you to reason about the existence of invisible pellets and ghosts.

This project includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It can be run for one particular question, such as q4, by:

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

It can be run for one particular test by commands of the form:

```
python autograder.py -t test_cases/q4/1-simple-eliminate
```

See the autograder tutorial in Project 0 for more information about using the autograder.

The code for this project contains the following files, available as a zip archive.

Files you'll edit:

`factorOperations.py`	Operations on Factors (join, eliminate, normalize).
`inference.py`	Inference algorithms (enumeration, variable elimination, likelihood weighting).
`bayesAgents.py`	Pacman agents that reason under uncertainty.

Files you should read but NOT edit:

`bayesNet.py`	The BayesNet and Factor classes.

Files you can ignore:

`graphicsDisplay.py`	Graphics for Pacman
`graphicsUtils.py`	Support for Pacman graphics
`textDisplay.py`	ASCII graphics for Pacman
`ghostAgents.py`	Agents to control ghosts
`keyboardAgents.py`	Keyboard interfaces to control Pacman
`layout.py`	Code for reading layout files and storing their contents
`autograder.py`	Project autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question
`bayesNets2TestClasses.py`	Project 4 specific autograding test classes

Files to Edit: You will fill in portions of `factorOperations.py`, `inference.py`, and `bayesAgents.py` during the assignment. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

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.

Treasure-Hunting Pacman

Pacman has entered a world of mystery. Initially, the entire map is invisible. As he explores it, he learns information about neighboring cells. The map contains two houses: a ghost house, which is probably mostly red, and a food house, which is probably mostly blue. Pacman's goal is to enter the food house while avoiding the ghost house.

Pacman will reason about which house is which based on his observations, and reason about the tradeoff between taking a chance or gathering more evidence. To enable this, you'll implement probabilistic inference using Bayes nets.

To play for yourself, run:

```
python hunters.py -p KeyboardAgent -r
```

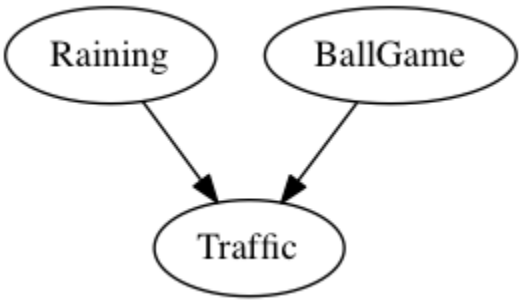
Bayes Nets and Factors

First, take a look at `bayesNet.py` to see the classes you'll be working with - `BayesNet` and `Factor`. You can also run this file to see an example `BayesNet` and associated `Factor`s:

```
python bayesNet.py
```

You should look at the `printStarterBayesNet` function - there are helpful comments that can make your life *much* easier later on.

The Bayes Net created in this function is shown below:



A summary of the terminology is given below:

- Bayes Net: This is a representation of a probabilistic model as a directed acyclic graph and a set of conditional probability tables, one for each variable, as shown in lecture. The Traffic Bayes Net above is an example.
- Factor: This stores a table of probabilities, although the sum of the entries in the table is not necessarily `1`. A factor is of the general form `P(X_1, ..., X_m, y_1,`

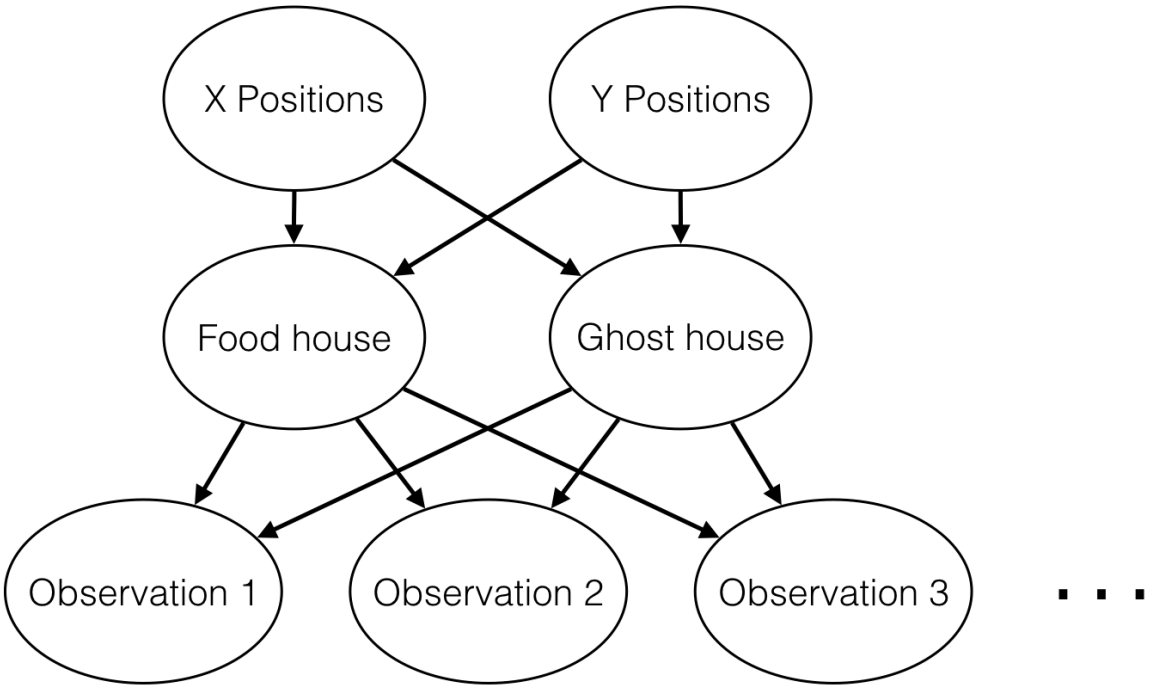
`..., y_n | Z_1, ..., Z_p, w_1, ..., w_q)`. Recall that lower case variables have already been assigned. For each possible assignment of values to the `X_i` and `Z_j` variables, the factor stores a single number. The `Z_j, w_k` variables are said to be conditioned while the `X_i, y_l` variables are unconditioned.

- Conditional Probability Table (CPT): This is a factor satisfying two properties:
    1. Its entries must sum to `1` for each assignment of the conditional variables.
    2. There is exactly one unconditioned variable.
- The Traffic Bayes Net stores the following CPTs: `P(Raining), P(Ballgame), P(Traffic | Ballgame, Raining)`

Question 1 (3 points): Bayes Net Structure

Implement the `constructBayesNet` function in `bayesAgents.py`. It constructs an empty Bayes net with the structure described below. (We'll specify the actual factors in the next question.)

The treasure hunting world is generated according to the following Bayes net:



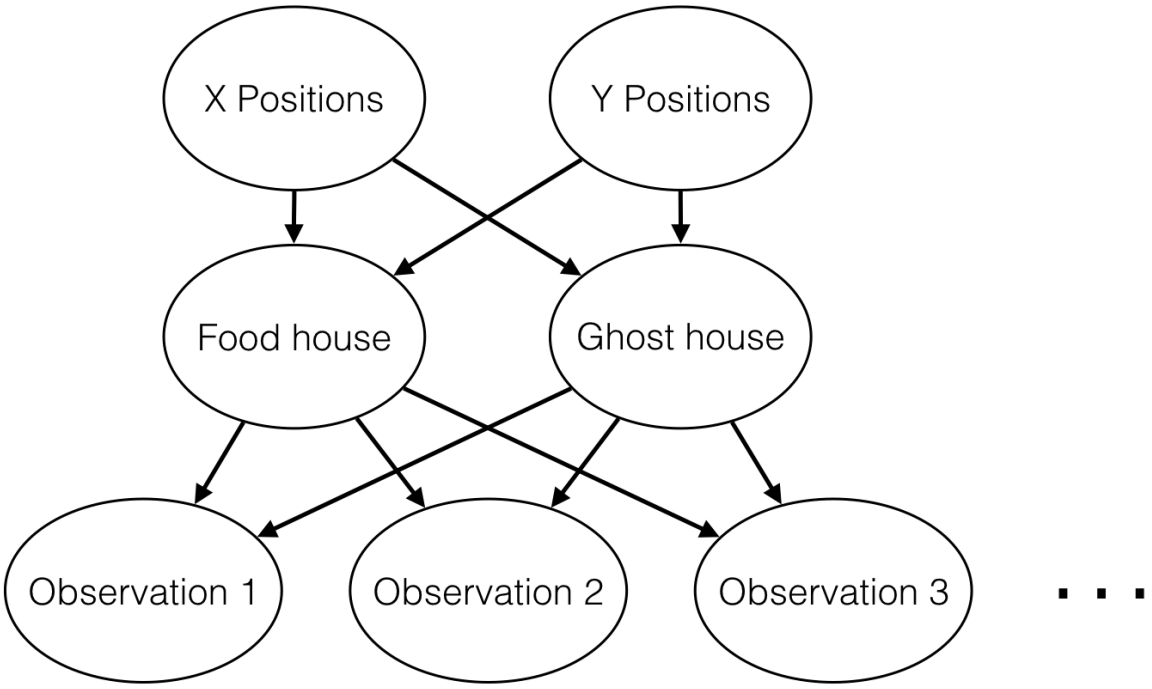
Don't worry if this looks complicated! We'll take it step by step. As described in the code for `constructBayesNet`, we build the empty structure by listing all of the variables, their values, and the edges between them. This figure shows the variables and the edges, but what about their values?

- X positions determines which house goes on which side of the board. It is either food-left or ghost-left.
- Y positions determines how the houses are vertically oriented. It models the vertical positions of both houses simultaneously, and has one of four values: both-top, both-bottom, left-top, and left-bottom. "left-top" is as the name suggests: the house on the left side of the board is on top, and the house on the right side of the board is on the bottom.
- Food house and ghost house specify the actual positions of the two houses. They are both deterministic functions of "X positions" and "Y positions"
- The observations are measurements that Pacman makes while traveling around the board. Note that there are many of these nodes---one for every board position that might be the wall of a house. If there is no house in a given location, the corresponding observation is none; otherwise it is either red or blue, with the precise distribution of colors depending on the kind of house.

Question 2 (3 points): Bayes Net Probabilities

Implement the `fillYCPT` and `fillObsCPT` functions in `bayesAgents.py`. These take the Bayes net you constructed in the previous problem, and specify the factors governing the Y position and observation variables. (We've already filled in the X position and house factors for you.)

Here's the structure of the Bayes net again:



For an example of how to construct factors, look at the implementation of the factor for X positions in `fillXCPT`.

The Y positions are given by values `BOTH_TOP`, `BOTH_BOTTOM`, `LEFT_TOP` and `LEFT_BOTTOM`. These variables, and their associated probabilities, are provided by constants at the top of the file.

If you're interested, you can look at the computation for house positions. All you need to remember is that each house can be in one of four positions: top-left, top-right, bottom-left, or bottom-right.

Observations are more interesting. Every possible observation position is adjacent to a possible center for a house. Pacman might observe that position to contain a red wall, a blue wall, or no wall. These outcomes occur with the following probabilities (again defined in terms of constants at the top of the file):

- If the adjacent house center is occupied by neither the ghost house or the food house, an observation is none with certainty (probability 1).

- If the adjacent house center is occupied by the ghost house, it is red with probability PROB_GHOST_RED and blue otherwise.
- If the adjacent house center is occupied by the food house, it is red with probability PROB_FOOD_RED and blue otherwise.

IMPORTANT NOTE: the structure of the Bayes Net means that the food house and ghost house might be assigned to the same position. This will never occur in practice. But the observation CPT needs to be a proper distribution for every possible set of parents. In this case, you should use the food house distribution.

Hints

There are only four entries in the Y position factor, so you can specify each of those by hand. You'll have to be cleverer for the observation variables. You'll find it easiest to first loop over possible house positions, then over possible walls for each house, and finally over assignments to (wall color, ghost house position, food house position) triples. Remember to create a separate factor for *every one* of the 4*7=28 possible observation positions.

Question 3 (5 points): Join Factors

Implement the `joinFactors` function in `factorOperations.py`. It takes in a list of `Factor`s and returns a new `Factor` whose probability entries are the product of the corresponding rows of the input `Factor`s.

`joinFactors` can be used as the product rule, for example, if we have a factor of the form `P(X|Y)` and another factor of the form `P(Y)`, then joining these factors will yield `P(X, Y)`. So, `joinFactors` allows us to incorporate probabilities for conditioned variables (in this case, `Y`). However, you should not assume that `joinFactors` is called on probability tables - it is possible to call `joinFactors` on `Factor`s whose rows do not sum to `1`.

Grading: To test and debug your code, run

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

It may be useful to run specific tests during debugging, to see only one set of factors print out. For example, to only run the first test, run:

```
python autograder.py -t test_cases/q3/1-product-rule
```

Hints and Observations

- Your `joinFactors` should return a *new* `Factor`.
- Here are some examples of what `joinFactors` can do:
  - `joinFactors(P(X | Y), P(Y))` = `P(X, Y)`
  - `joinFactors(P(V, W | X, Y, Z), P(X, Y | Z))` = `P(V, W, X, Y | Z)`
  - `joinFactors(P(X | Y, Z), P(Y))` = `P(X, Y | Z)`
  - `joinFactors(P(V | W), P(X | Y), P(Z))` = `P(V, X, Z | W, Y)`
- For a general `joinFactors` operation, which variables are unconditioned in the returned `Factor`? Which variables are conditioned?
- `Factor`s store a `variableDomainsDict`, which maps each variable to a list of values that it can take on (its domain). A `Factor` gets its `variableDomainsDict` from the `BayesNet` from which it was instantiated. As a result, it contains all the variables of the `BayesNet`, *not* only the unconditioned and conditioned variables used in the `Factor`. For this problem, you may assume that all the input `Factor`s have come from the same `BayesNet`, and so their `variableDomainsDict`s are all the same.

Question 4 (4 points): Eliminate

Implement the `eliminate` function in `factorOperations.py`. It takes a `Factor` and a variable to eliminate and returns a new `Factor` that does not contain that variable. This corresponds to summing all of the entries in the `Factor` which only differ in the value of the variable being eliminated.

Grading: To test and debug your code, run

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

It may be useful to run specific tests during debugging, to see only one set of factors print out. For example, to only run the first test, run:

```
python autograder.py -t test_cases/q4/1-simple-eliminate
```

Hints and Observations

- Your `eliminate` should return a *new* `Factor`.
- `eliminate` can be used to marginalize variables from probability tables. For example:
  - `eliminate(P(X, Y | Z), Y)` = `P(X | Z)`
  - `eliminate(P(X, Y | Z), X)` = `P(Y | Z)`
- For a general `eliminate` operation, which variables are unconditioned in the returned `Factor`? Which variables are conditioned?

- Remember that `Factor`s store the `variableDomainsDict` of the original `BayesNet`, and *not* only the unconditioned and conditioned variables that they use. As a result, the returned `Factor` should have the same `variableDomainsDict` as the input `Factor`.

Question 5 (4 points): Normalize

Implement the `normalize` function in `factorOperations.py`. It takes a `Factor` as input and normalizes it, that is, it scales all of the entries in the `Factor` such that the sum of the entries in the `Factor` is `1`.

Grading: To test and debug your code, run

```
python autograder.py -q q5
```

It may be useful to run specific tests during debugging, to see only one set of factors print out. For example, to only run the first test, run:

```
python autograder.py -t test_cases/q5/1-preNormalized
```

Hints and Observations

- Your `normalize` should return a *new* `Factor`.
- `normalize` does not affect probability distributions (since probability distributions must already sum to `1`).
- For a general `normalize` operation, which variables are unconditioned in the returned `Factor`? Which variables are conditioned? Make sure to read the docstring of `normalize` for more instructions.
- Remember that `Factor`s store the `variableDomainsDict` of the original `BayesNet`, and *not* only the unconditioned and conditioned variables that they use. As a result, the returned `Factor` should have the same `variableDomainsDict` as the input `Factor`.

Question 6 (4 points): Variable Elimination

Implement the `inferenceByVariableElimination` function in `inference.py`. It answers a probabilistic query, which is represented using a `BayesNet`, a list of query variables, and the evidence.

Grading: To test and debug your code, run

```
python autograder.py -q q6
```

It may be useful to run specific tests during debugging, to see only one set of factors print out. For example, to only run the first test, run:

```
python autograder.py -t test_cases/q6/1-disconnected-eliminate
```

Hints and Observations

- The algorithm should iterate over hidden variables in elimination order, performing joining over and eliminating that variable, until only the query and evidence variables remain.
- The sum of the probabilities in your output factor should sum to one (so that it is a true conditional probability, conditioned on the evidence).
- Look at the `inferenceByEnumeration` function in `inference.py` for an example on how to use the desired functions. (Reminder: Inference by enumeration first joins over all the variables and then eliminates all the hidden variables. In contrast, variable elimination interleaves join and eliminate by iterating over all the hidden variables and perform a join and eliminate on a single hidden variable before moving on to the next hidden variable.)

Question 7 (1 point): Marginal Inference

Inside `bayesAgents.py`, use the `inferenceByVariableElimination` function you just wrote to compute the marginal distribution over positions of the food house, then return the most likely position. This information is used by Bayesian Pacman, who wanders around randomly collecting information for a fixed number of timesteps, then heads directly to the house most likely to contain food.

Question 8 (4 points): Value of Perfect Information

Bayesian Pacman spends a lot of time wandering around randomly, even when further exploration doesn't provide any additional value. Can we do something smarter?

We'll evaluate VPI Pacman in a more restricted setting: everything in the world has been observed, except for the colors of one of the houses' walls. VPI Pacman has three choices:

1. immediately enter the already-explored house
2. immediately enter the hidden house
3. explore the outside of the hidden house, and then make a decision about where to go

You'll implement code to reason about the expected value of each of these actions.

First look at `computeEnterValues`. This function computes the expected value of entering the left and right houses. Again, you can use the inference code you already wrote to do all the heavy lifting here. First compute *p(foodHouse = topLeft and ghostHouse = topRight | evidence)* and *p(foodHouse = topRight and ghostHouse = topLeft | evidence)*. Then use these two probabilities to compute expected values for rushing left and rushing right.

Next look at `computeExploreValue`. This function computes the expected value of exploring all of the hidden cells, and then making a decision. To do this, you'll need to think about all of the things that might happen as a result of your exploration. Maybe you'll find 1 red wall and 6 blue ones; maybe you'll find 2 red walls and 5 blue ones; and so on. We've provided a helper method, `getExplorationProbsAndOutcomes`, which returns a list of future observations Pacman might make, and the probability of each. To calculate the value of the extra information Pacman will gain, you can use the following formula:

E[value of exploration] = Σ p(new evidence) max_{actions} E[action | old evidence and new evidence]

Note that *E[action | evidence]* is exactly the quantity computed by `computeEnterVals`, so to compute the value of exploration, you can call `computeEnterValues` again with the hypothetical evidence provided by `getExplorationProbsAndOutcomes`.

Hints

After exploring, Pacman will again need to compute the expected value of entering the left and right houses. Fortunately, you've already written a function to do this! Your solution to `computeExploreValue` can rely on your solution to `computeEnterValues` to determine the value of future observations.

Prior to submitting, be sure you run the autograder on your own machine. Running the autograder locally will help you to debug and expedite your development process. The autograder can be invoked on your own machine using the command:

```
python autograder.py
```

Good luck with the project!