Assignment 3. Bayes' Nets

Gary Geunbae Lee
CSED442 - Artificial Intelligence

Contact: TA Seonghyeon Lee (sh0416@postech.ac.kr)

Acknowledgement

This assignment comes from the University of Texas at Austin. Thanks for sharing their assignment and we appreciate their work.

General Instructions

In this assignment, 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
```

You will implement your code in the following three files.

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

You should read and understand this file.

- `game.py` : General class for game.
- `bayesNet.py` : The BayesNet and Factor classes.

You can ignore these files.

- `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 teset and solution files
- `textClasses.py` : General autograding test classes
- `bayesNets2TestClasses.py` : Project 4 specific autograding test classes

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.

Bayes' Nets and Factors

First, take a look at `bayesNet.py` to see the classes you'll be working with. 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. This function creates the Bayes' Net in the following figure.

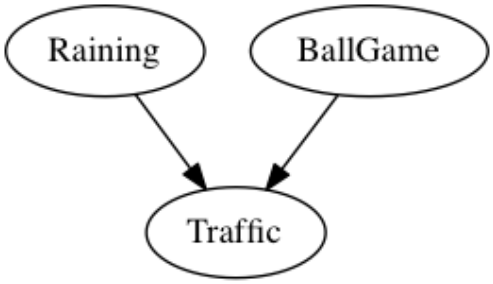

Figure 1: The Bayes' Net created in printStarterBayesNet

A summary of the terminology is given below:

- **Bayes' Net** is a representation of a probabilistic model as a directed acyclic graph and a set of conditional probability tables, one for each variable.

- **Factor** stores a table of probabilities, although *the sum of the entries in the table is not necessarily 1.* A general form of factor is $P(X_1, ..., X_m, y_1, ..., y_n | Z_1, ..., Z_p, w_1, ..., w_q)$ where lower case variables have already been assigned. For each possible combination of the X_i and Z_j, the factor stores a single number. The Z_j and w_k variables are said to be conditioned while the X_i and y_l variables are unconditioned.
- **Conditional Probability Table (CPT)** satisfy two properties: (1) Its entries must *sum to one* for each assignment of the conditional variables and (2) there is *exactly one unconditioned variable.* For example, the Bayes' Net in `printStarterBayesNet` stores the following CPTs: $P(Raining), P(BallGame), P(Traffic | Ballgame, Raining)$

Problem 1. Bayes Net Structure (3 points)

Implement the `constructBayesNet` function in `bayesAgents.py`. It constructs an emtpy Bayes net with the structure described below. We will specify the actual factors in the next problem. The treasure hunting world is generated according to the following Bayes net:

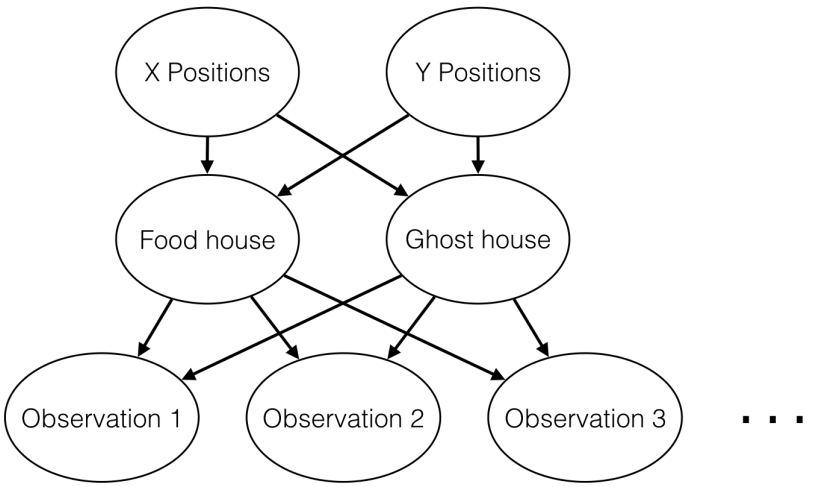

Figure 2: The Bayes' Net for treasure hunting world

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. You might use the following information to complete constructing the empty Bayes net.

- **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.

Problem 2. Bayes Net Probabilities (3 points)

Implement the `fillYCPT` and `fillObsCPT` functions in `bayesAgent.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.) 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**. All you need to remember is that each house can be in one of four positions: *top-left*, *top-right*, *bottom-left*, and *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 *none* wall. These outcomes occur with the following probabilities (again defined in terms of constants at the top of the file):

- If the cell is neither the ghost house wall or the food house wall, an observation is *none* with certainty (probabilitiy 1).
- If the cell is the wall of the **ghost house**, it is *red* with probability `PROB_GHOST_RED` and blue otherwise.
- If the cell is the wall of the **food house**, it is *red* with probability `PROBL_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 when both house are overlapped**.

Hint

There are only four entries in the **Y position** factor, so you can specify each of those by hand. However, you'll have to be cleverer for the **observation** variable. 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.

Problem3. Join Factors (5 points)

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` is called on probability tables - it is possible to call `joinFactors` on `Factor`s whose rows do not sum to one.

Hints and Observations

First, your `joinFactors` should return a new `Factor`. Second, 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(`$P(V|W)$`, `$P(X|Y)$`, `$P(Z)$`) = `$P(V, X, Z|W, Y)$

You should consider which variables are unconditioned in the returned `Factor` and which are conditioned. Lastly, `Factor`s store a `variableDomainsDict`, which maps each variable to a list of values that it can take on. 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.

Problem 4. Eliminate (4 points)

Implement the `eliminate` function in `factorOperation.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.

Hints and Observations

First, 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) = `$P(Y|Z)$

You should consider which variables are unconditioned in the returned `Factor` and which are conditioned. Lastly, `Factor`s store a `variableDomainsDict`, and it contains all the variables of the `BayesNet`, not only the unconditioned and conditioned variables used in the `Factor`. For this problem, the returned `Factor` should have the same `variableDomainsDict` as the input `Factor`.

Problem 5. Normalize (4 points)

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 one.

Hints and Observation

First, your `normalize` should return a new `Factor`. Second, you should consider which variables are unconditioned in the returned `Factor` and which are conditioned. **Make sure to read the docstring of `normalize` for more instructions**. Lastly, `Factor`s store a `variableDomainsDict`, and it contains all the variables of the `BayesNet`, not only the unconditioned and conditioned variables used in the `Factor`. For this problem, the returned `Factor` should have the same `variableDomainsDict` as the input `Factor`.

Problem 6. Variable Elimination (4 points)

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. You should read the docstring in the code carefully.

Hints and Observation

Your `inferenceByVariableElimination` should return a new `Factor` which stores the probability $P(Query|Evidence = E)$. The algorithm should iterate over hidden variables in elimination order, performing joining over and eliminating that variable, until the only the query and evidence variables remain. Also, 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.)

Problem 7. Marginal Inference (1 points)

Inside `bayesAgents.py`, implement `getMostLikelyFoodHousePosition` by using the function `inferenceByVariableElimination` you just wrote to compute the marginal distribution over positions of the food house. It will 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.