Prof. Dr. techn. Wolfgang Nejdl
Julian Laue, Stefan Schestakov, Uwe Hadler
Fachgebiet Wissensbasierte Systeme

November 8th 2024

Artificial Intelligence II
Project 1

Winter Semester 2024/25

Organization

- In order to pass you need to get at least 19 points by the autograder.
- You may participate in this project as a group of 2 persons.
- Submit your solution no later than December 16th, 13:00.
- Submit your solution (only the factorOperations.py, inference.py and bayesAgents.py files) to *aiwise24@l3s.de* using the subject **"AI2Project1: [YourNames]"**. Please be sure to provide your **matriculation numbers** in the email.

Disclaimer

We are reusing a project from UC Berkeley[1]. The required code is available for download via StudIP.

Preliminaries

- If you are unfamiliar with python, you can try the following tutorial `https://inst.eecs.berkeley.edu/~cs188/sp24/projects/proj0//`. It also provides installation instructions.
- If you encounter any bugs, please report them to: aiwise24@l3s.de

Project Structure & Grading

To run the project you need a **python3** environment. You can check your progress by running the following command:

```
python autograder.py
```

[1]`http://ai.berkeley.edu`

When running the autograder you might encounter an exception on modern python versions. The cause of this is that the method `cgi.escape` has been deprecated sine 2011 and removed with Python 3.8. To fix this you can either use an earlier version (3.6 works fine) or make the following two changes to the autograder: Replace `import cgi` with `from html import escape`. Replace `message = cgi.escape(message)` with `message = escape(message)`.
Files you will edit:

- `factorOperations.py`
- `inference.py`
- `bayesAgents.py`

Files should look at:

- `bayesNet.py`

You can ignore the other files. Do not change any other files than `factorOperations.py`, `inference.py` and `bayesAgents.py` and do not change any of the provided function names as this will create problems for the autograder. Your code will be autograded, however as the correctness of the implementation and not the autograder's judgement should be the final judge of your score, we will review grade assignments individually wherever necessary.
A few usefull hints:

- Read the instructions carefully. If hints or advice is given, follow it.
- You should be able to solve the tasks without importing any additional packages. If the autograder does not run on a blank python installation because you import a package you don't use, you will lose a point. Don't import packages which solve the tasks for you. If you do, you will get 0 points for that task.
- Read the error messages you get carefully. Most of the time they point you to the problem.
- Reduce the textual output of your code to a minimum.
- Run the autograder before submitting the project and make sure to submit the correct versions. If the autograder doesn't run for you, it won't run for us.

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 `Factors`:

```
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:

$$(\text{Raining} \rightarrow \text{Traffic} \leftarrow \text{Ballgame})$$

A summary of the terminology is given below:

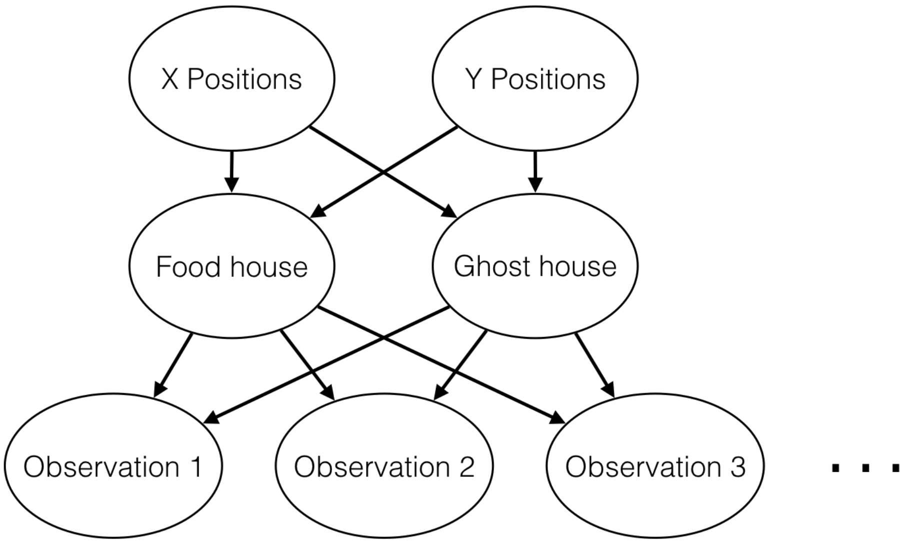
- `Bayes Net`: This is a representation of a probabilistic model as a directed acyclic graph and a set of conditional probability tables, one for each variable, as shown in lecture. The Traffic Bayes Net above is an example.
- `Factor`: This stores a table of probabilities, although the sum of the entries 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:

$$P(Raining), P(Ballgame), P(Traffic|Ballgame, Raining)$$

1 Question: Bayes Net Structure

Implement the `constructBayesNet` function in `bayesAgents.py`. It constructs an empty `BayesNet` 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 determine which house goes on which side of the board. It is either food-left or ghost-left.
- Y positions determine 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.

Grading: To test and debug your code, run

```
python autograder.py −q q1
```

2 Question: Bayes Net Probabilities

Implement the `fillYCPT` function in `bayesAgents.py`. These take the Bayes net you constructed in the previous problem, and specify the factors governing the Y position variables. (We've already filled in the X position, house, and observation factors for you.)
For the structure of the Bayes Net, refer to question 1.

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_VAL`, `BOTH_BOTTOM_VAL`, `LEFT_TOP_VAL`, `LEFT_BOTTOM_VAL`. These variables, and their associated probabilities `PROB_BOTH_TOP`, `PROB_BOTH_BOTTOM`, `PROB_ONLY_LEFT_TOP`, `PROB_ONLY_LEFT_BOTTOM`, 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.

Hint: There are only four entries in the Y position factor, so you can specify each of those by hand.

Grading: To test and debug your code, run

```
python autograder.py −q q2
```

3 Question: 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 `Factors` whose rows do not sum to 1.

Grading: To test and debug your code, run

```
python autograder.py −q q3
```

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 `variableDomainsDicts` are all the same.

4 Question: 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`.

5 Question: 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. If the sum of probabilities in the input `Factor` is 0, you should return `None`.

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. In particular, pay attention to the treatment of unconditioned variables with exactly one entry in their domain.
- 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`.

6 Question: 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 1 (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.)
- You will need to take care of the special case where a `Factor` you have joined only has one unconditioned variable (the docstring specifies what to do in greater detail).

7 Question: Marginal Inference

Inside `bayesAgents.py`, use the `inference.inferenceByVariableElimination` function you just wrote to complete the function `getMostLikelyFoodHousePosition`. This function should compute the marginal distribution over positions of the food house, then return the most likely position. The return value should be a dictionary containing a single key-value pair, `{FOOD_HOUSE_VAR: best_house_val}`, where `best_house_val` is the most likely position from `HOUSE_VALS`. This 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.

Grading: To test and debug your code, run

```
python autograder.py -q q7
```

Hint: You may find `Factor.getProbability(...)` and `Factor.getAllPossibleAssignmentDicts(...)` to be useful.

8 Question: Value of Perfect Information (BONUS)

We will discuss the topics needed for this question at the end of November. If you solve it, you can get an additional 3 points.

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

Part A

First look at `computeEnterValues` in `bayesAgents.py`. This function computes the expected value of entering the top left and top right houses. Again, you can run the inference function you already wrote, on the bayes net `self.BayesNet` to do all the heavy lifting here. First compute $p(\text{foodHouse} = \text{topLeft and ghostHouse} = \text{topRight} \mid \text{evidence})$ and $p(\text{foodHouse} = \text{topRight and ghostHouse} = \text{topLeft} \mid \text{evidence})$. Then use these two probabilities to compute expected rewards for entering the top left or top right houses. Use `WON_GAME_REWARD` and `GHOST_COLLISION_REWARD` as the reward for entering the foodHouse and ghostHouse, respectively.

Part B

Next look at `computeExploreValue`. This function computes the expected value of exploring all of the hidden cells, and then making a decision. 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:

$$E[\text{value of exploration}] = \sum p(\text{evidence}) \max_{\text{actions}} E[\text{value of action}|\text{evidence}]$$

Note that $E[\text{value of action}|\text{evidence}]$ 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`.

Grading: To test and debug your code, run

```
python autograder.py —q q8
```

Hint: 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.