CS 476/676, Spring 2016 Problem Set #2b (Due by 11:59pm on Tuesday, Mar. 15)

---

1 Instructions

This assignment contains a set of problems about robot localization.

1.1 What to Hand In

All of your submission files should be handed in as a single archive named `hw1b-username1-username2.zip`, where the `usernames` have been replaced with the JHED IDs of your group members. (Your group is allowed to have 2 members; see Section 1.2 for details on collaboration.)

Your zip archive should contain a writeup called `writeup.pdf`, as well as various other files described in the sections labeled `Deliverables`.

Hand in the zip archive by creating a private note to the instructors on Piazza with the title *Submission 2b from* `<list of names of team members>`, and attaching your zip file to that note. The note should be submitted to the `submission2b` folder.

1.2 Submission Policies

Please note the following:

- **Collaboration:** Please work in groups of size 2 people. The homeworks are a way for you to work through the material you're learning in this class on your own. But, by working in a group, and debugging each other's solutions, you'll have a chance to learn the material in more depth. The recommended format for tackling these problem sets is the following. Write a high level sketch of the solution for all of the problems on your own. Meet as a group to brainstorm your solutions and converge on a solution as a group. It is important that you have a good understanding of how you'd have approached the problem independently before discussing your solution with the other group members. Developing this intuition will serve you well in the final exam where you will be required to work on your own. Persuant to your group meeting, write up the solutions on your own. Thereafter, meet as a group to clean up and submit a final write up as a group. By now, each of you should have a solid understand of the concepts involved, and by meeting as a group, you've had a chance to see common ways in which one can make mistakes. Submit your final solution as a final writeup for the group. Your submission should include the names of every team member. Also, name your file as `hw1b-username1-username2.zip`.

- **Late Submissions:** We allow each student to use up to 3 late days over the semester. You have late days, not late hours. This means that if your submission is late by any amount of time past the deadline, then this will use up a late day. If it is late by any amount beyond 24 hours past the deadline, then this will use a second late. **If you jointly submit an assignment as a team, then every team member will lose late days if the assignment is submitted late.** If you collaborate with team members but independently submit your own version, then late hours will only apply to you.

2 Robot Localization: Model Description

In the remainder of this assignment, we will work through an example problem on robot localization. Assume you have a robot, which is navigating an $I \times J$ grid environment (the *map*). At each time step, the robot is located at some position on the map. Using its sensors, it collects a set of observations, which let it see whether there are any walls or landmarks in the adjacent squares. Then it takes some action (moving one square either north, east, south, or west), which determines its position during the next time step. The observations and actions are both probabilistic: with some low probability, the sensors may fail to detect a wall or landmark, or the robot may fail to move out of its current square.

We will encode all of this as a Bayesian network that includes random variables for each time step, representing the robot's position at that time step, the action the robot took, and what the robot observed from that position. Under this model, one can predict the robot's location on the map, given its observations.

You will implement algorithms to perform both parameter estimation and inference in this Bayesian network. In the first problem, you will do parameter estimation. You will be given a series of samples of the robot's movement through the map, along with what it observed during its run. From this data, you will perform parameter estimation to fill in the conditional probability tables (CPTs) of the model. In the second problem, you will do inference. Given a complete model specification (i.e. the CPDs of the network), you will perform exact inference and return solutions to queries.

These two tasks are detailed in subsections 3.1 and 3.2. In the remainder of this section, we will describe the localization model in more detail. (You will not need to construct the Bayesian network yourself; a set of `network` files have been provided for you, corresponding to different maps and numbers of timesteps.)

2.1 Details of the Bayesian Network

Map Layout The map is an $I \times J$ grid of discrete cells denoted (i, j), where i denotes the row index and j denotes the column index. Each grid cell can either be empty or it can be occupied with an object. There are two types of objects: walls and landmarks. There is only one type of wall (the wall at one position looks the same as the wall in another), whereas there may be multiple types of landmark (however, there can be more than one landmark of each type).

The cell $(1, 1)$ is the bottom left of the map. The top of the map is North, so moving northward increases the row index, while moving eastward increases the column index.

Motion model The random variables *PositionRow_t* and *PositionCol_t* indicate the robot's position in the grid at time t (starting at 0). The values of the variables can be one of $\{1, 2, \ldots, I\}$ and $\{1, 2, \ldots, J\}$, respectively.

At each time step, the robot must take one action. The robot can take one of four actions: `MoveNorth`, `MoveSouth`, `MoveEast`, `MoveWest`. The variable *Action_t* denotes the action the robot takes at time t.

Performing an action at time t will cause a change position at time $t + 1$ with respect to the motion model. If the robot moves North, the row will either increase by 1 (decrease if South) or stay the same. If the robot moves East, the column will either increase by 1 (decrease if West) or stay the same.

The action causes the robot to change position with some probability—these probabilities are given by the *motion model*. (The outcomes of actions are probabilistic and not deterministic because the robot is not perfect. An action may result in a different outcome than expected,

PositionRow_0	PositionCol_0	Action_0	ObserveWall_N_0	ObserveLandmark1_N_0	...
PositionRow_1	PositionCol_1	Action_1	ObserveWall_N_1	ObserveLandmark1_N_1	...
PositionRow_2	PositionCol_2	Action_2	ObserveWall_N_2	ObserveLandmark1_N_2	...
PositionRow_3	PositionCol_3	Action_3	ObserveWall_N_3	ObserveLandmark1_N_3	...
PositionRow_4	PositionCol_4	Action_4	ObserveWall_N_4	ObserveLandmark1_N_4	...
PositionRow_5	PositionCol_5	Action_5	ObserveWall_N_5	ObserveLandmark1_N_5	...
...	...	...	...	...	...

Table 1: Some of the variables in the model. Omitted for space: observation variables for directions other than north (N), observation variables for landmarks other than Landmark1, and variables corresponding to time values beyond 5.

although the intended outcome will have the highest probability. The two possible outcomes are moving in the intended direction, and staying in)

We will assume that the grid **wraps around**, meaning that if the robot moves South from row 1, then it will move to row I, and if it moves West from column 1 it will move to column J.

Observation model Using its sensors, the robot can observe objects (walls or landmarks) at a particular time from its current position.

The binary variable $ObserveWall\_d\_t$ denotes whether or not the robot observes a wall at time t in the direction d, where d can be N, S, E, or W (that is, the robot can observe anything in a non-diagonal adjacent cell). For example, $ObserveWall\_E\_8$ denotes whether the robot in position (i, j) observes a wall in cell $(i, j+1)$ at time 8. The probability of this being `Yes` depends on the position of the robot at time t. Similarly, the binary variable $ObserveLandmark\ell\_d\_t$ denotes whether or not the robot observes landmark ℓ at time t in direction d; the CPD of this variable also depends on the robot's position at time t.

The robot is likely to observe an object if it exists. If there is a wall at $(4, 2)$, then the observation model at time 10 should give high probability to $P(ObserveWall\_S\_10 = \texttt{Yes} | PositionRow\_10 = \texttt{5}, PositionCol\_10 = \texttt{2})$. However, the robot's sensors are not perfect, and so it will not always observe the same objects from a particular position—its observations are thus probabilistic, governed by the *observation model*.

Prior probabilities In this model, the position variables at time 0 and all action variables do not have parents in the model, so they are associated with a prior probability rather than a conditional probability. We simply assume the prior probability for each of these variables is the **uniform distribution**. The `cpd` files do not encode priors.

2.2 Putting it all Together

Let us summarize the information we have presented so far. The Bayesian network described above encodes a joint probability distribution over many variables: examples of these variables are shown in Table 1. Depending on the number of landmarks and time steps, you may be performing inference over a model with tens of thousands of variables. The number of possible configurations of the variables (assignments of values to the variables) is astronomical, since it is exponential in the number of variables.

It would be impossible to construct a probability table that has a line for every possible combination of variable assignments. Thus, we take advantage of graphical models to factorize this distribution into a number of small conditional probability distributions. Before you begin

programming, it may be helpful to look at the `network` and `cpd` files so that you have a full understanding of the model.

2.3 Input Format

The localization network is a Bayesian network. Included in `hw2-files.zip` are the files `network-grid*.txt` and `cpd-grid*.txt`, which contain the models' network structure and CPDs. Note that two of the CPD files are blank: you will need to generate these yourself (section 3.1).

The `network` and `cpd` files take the same format as they have in previous questions and assignments. This subsection contains a review of that format.

The file `network-grid*.txt` encodes the variables and network structure. The first line is an integer (denoted by n here for explanation) which will tell the program that how many random variables there are in the network. The next n lines each contain the names of the variables in the network followed by the values (a comma-separated list) the variable can take (one variable per line). For example, the following line:

```
Action_0 MoveNorth,MoveSouth,MoveEast,MoveWest
```

indicates that the network will contain a variable called *Action_0* which can take the values `MoveNorth`, `MoveSouth`, `MoveEast`, and `MoveWest`.

After the variable declarations, all subsequent lines encode edges in the network. Each line contains one variable name, followed by a right arrow, followed by a second variable name, such that there is a directed edge from the first to the second. For example, the following line:

```
Action_0 -> PositionRow_1
```

indicates a directed edge from the *Action_0* node to the *PositionRow_1* node. For clarity, the input files should only use right arrows (`->`) and not left arrows.

The file `cpd-grid*.txt` encodes the conditional probability distributions in this model. Each line defines a conditional probability, where the set of conditional probabilities that need to be defined are determined by the network structure. The first part of the line is a comma-separated list of variable/value pairs (of the form `VariableName=Value`) for the left-hand side of the conditional probability definition. The second part (following a space) is a comma-separated list of variable/value pairs for the right-hand side of the conditional probability. The third part (following another space) is a floating point number indicating the probability. For example, the following line:

```
PositionRow_2=2 PositionRow_1=1,Action_1=MoveNorth 0.799424
```

should be interpreted to mean:

$$P(PositionRow_2 = \texttt{2} | PositionRow_1 = \texttt{1}, Action_1 = \texttt{MoveNorth}) = 0.799424$$

Each of the networks that we provide includes variables corresponding to 10, 100, or 1000 time steps.

Additionally, we have included the files `map-grid*.txt`. These show a visual representation of the underlying map that was used to generate the training examples and `cpd` files. This is

only to help your understanding; these files should not be used during parameter estimation or inference.

You will see that the input file sizes are very large. This is not a standard or efficient representation, but we use readable string names for variables and values so that you can better interpret the model. (Typically, input files represent variable names as integers.)

Sparse probabilities The motion model defines the probability of moving from row i to row i' after performing an action. However, above we said that the robot can only move to one of two possible cells given an action: either its current cell, or the adjacent cell in the direction of the move (i.e. row $i + 1$ if moving north). This implies that there is 0 probability of moving from i to any row i' that is not allowed by the action. Rather than storing all of the CPDs for row-to-row movements (the transition matrix), the `cpd` file will only store lines corresponding to non-zero entries. You will need to ensure that your code assumes 0 probability for the omitted entries.

(Everything above applies to columns in addition to rows.)

2.4 Programming Requirements

As in Homework 1b, you must print out all the probabilities with 13 significant figures in scientific notation. Below are examples for Matlab, Python, and Java of how to do such a printout for a variable named `x`:

Matlab:

```
x = 1.0 / 7.0
fprintf('%.13e\n', x)
```

Python:

```
x = 1.0 / 7.0
print '%.13e' % (x)
```

Java:

```
double x = 1.0 / 7.0;
System.out.printf("%.13e", x);
```

3 Robot Localization Problems [140 points] [20 extra credit]

3.1 Learning: Parameter Estimation [50 points]

In this section, you will implement a program that estimates the CPDs of a network, given a set *training* examples.

3.1.1 Parameter Sharing

For the observation model, you have parameters of the form:

$$P(Observation\_t | PositionRow\_t, PositionCol\_t)$$

In the CPT, these distributions are given for particular values of t (e.g. time 4 and time 8). However, we have no reason to believe that the particular time value should affect the probability of making a particular observation, for example. Therefore, the distributions you learn should be the same across all values of t: in other words, the time steps *share* the various parameters in the model.

Similarly, for the motion model we will assume that the parameters of the motion model are shared across time values as well as particular positions (i and j values). We will assume the probability of ending up in row 4 after moving forward from row 3 should be the same as moving from row 6 to 7. Thus, you will learn parameters for the following templates:

$$\begin{aligned}
&P(PositionRow_t = i | PositionRow_t\text{-}1 = i - 1, Action_t\text{-}1)\\
&P(PositionRow_t = i | PositionRow_t\text{-}1 = i + 1, Action_t\text{-}1)\\
&P(PositionRow_t = i | PositionRow_t\text{-}1 = i, Action_t\text{-}1)\\
&P(PositionCol_t = j | PositionCol_t\text{-}1 = j - 1, Action_t\text{-}1)\\
&P(PositionCol_t = j | PositionCol_t\text{-}1 = j + 1, Action_t\text{-}1)\\
&P(PositionCol_t = j | PositionCol_t\text{-}1 = j, Action_t\text{-}1)
\end{aligned}$$

You do not have to worry about whether the robot can transition to cells that contain walls and landmarks—for simplicity, it is okay to assign positive probability to occupying these positions.

3.1.2 Training Examples

You will be given many examples of "trajectories" traversed by the robot. A trajectory consists of a sequence of 10–1000 time steps, where you observe the values of all the random variables in the model corresponding to each time step. You will estimate the CPDs from these instances (see 3.1.3).

We have included files named `training-grid*.txt` as training data for the `cpd-grid*.txt` parameters. Each line of the training file has the following format:

```
TrajectoryNumber TimeStep Variable1=Value1 Variable2=Value2 ...
```

The position and action variables are always given. We use a **sparse** representation of the observation variables, of which there may be hundreds: the observations are assumed to have the value `No` unless explicitly given in the line.

The three lines below give an example of times $t = 0$ through $t = 2$ for the 32nd trajectory. At time $t = 1$, for example, you can assume $ObserveWall\_S\_1$=`No` since it is not included.

```
32 0 PositionRow_0=3 PositionCol_0=4 Action_0=MoveEast
32 1 PositionRow_1=3 PositionCol_1=5 Action_1=MoveNorth ObserveWall_E_1=Yes
32 2 PositionRow_2=4 PositionCol_2=5 Action_2=MoveNorth ObserveWall_E_2=Yes
```

3.1.3 Estimation

Suppose you want to estimate $P(ObserveWall\_S\_t = \texttt{Yes} | PositionRow\_t = \texttt{5}, PositionCol\_t = \texttt{2})$. The maximum likelihood estimate (MLE) for this parameter is the number of times a wall was observed to the south of the robot at position $(5, 2)$ divided by the total number of times the robot made observations facing south at position $(5, 2)$.

Problems can arise when estimating parameters for which no examples were observed during training. This can result in an MLE that assigns 0 probability to some values of a variable, and therefore the probability of any joint query that includes that 0-probability value will be 0.

We can address this issue with Dirichlet prior *smoothing*. This assumes there is a prior probability on the values you are estimating, and the prior probability is given by the Dirichlet distribution. The Dirichlet distribution can make it unlikely to have parameters that give tiny probability. Instead of estimating parameters to maximize the likelihood of the data, you will estimate the parameters to maximize the joint likelihood of both the data given the parameters and the parameters given the prior. This is known as *maximum a posteriori* (MAP) estimation.

In this assignment, you will perform a special case of Dirichlet prior smoothing called +1 smoothing or Laplace smoothing. This involves adding 1 to the "counts" of observations when computing the MLE, so that values observed 0 times will still have some probability mass. That is, if your MLE for a value i is given by $n_i / \sum_{i'} n_{i'}$, the smoothed estimate is $(n_i+1)/\sum_{i'}(n_{i'}+1)$, where n_i is the count of value i.[1]

Your parameter estimation program should perform MAP estimation using this +1 technique.

3.1.4 Deliverables [40 points]

You will create and submit a program called `estimate-params`. This program will take 3 arguments at the command line: a path to the network structure file, a path to the training examples, and a path to an output file. Given a network file and training examples, your program should use MAP estimation to learn a CPDs for the network, and then it will store the complete CPT in the output file. The output file should have the same format as the other `cpd` files that have been given to you.

An example call to your program is:

[1] Here is a sketch of why this is. Under a multinomial distribution with parameters θ, the probability of observing the ith value n_i times is $P(n_i|\theta) \propto \theta_i^{n_i}$. Under a Dirichlet distribution with parameters α, the probability of the ith parameter having the value θ_i is $P(\theta_i|\alpha_i) \propto \theta_i^{\alpha_i - 1}$. The two distributions have different normalizing constants, but the core has a similar form. The joint probability of n_i and θ_i conditioned on α is $P(n_i|\theta_i)P(\theta_i|\alpha_i) \propto \theta_i^{n_i}\theta_i^{\alpha_i - 1} = \theta_i^{n_i + \alpha_i - 1}$. Recall that the MLE of θ_i is proportional to n_i. Similarly, it turns out that the MAP estimate of θ_i given α_i is proportional to $n_i + \alpha_i - 1$. Thus, applying +1 smoothing to the MLE as described above corresponds to a MAP estimate conditioned on a Dirichlet(α) prior where $\alpha_i = 2$. See 17.3.2 in the Koller & Friedman book for more details.

```
./estimate-params network-grid10x10-t100.txt training-grid10x10-t100.txt cpd-grid10x10-t100.txt
```

After your program finishes, the file `cpd-grid10x10-t100.txt` should contain an estimated CPT for this network.

You must hand in:

- Your source code, the `estimate-params` executable, and a Makefile file to compile your code (if applicable).
- A paragraph giving a brief overview of how your code works, and anything else that would be useful for us to know in order to understand your code. Please add this to your `writeup.pdf`.

We have given you three models with blank CPD files: `grid10x10-t100`, `grid10x10-t1000`, and `grid15x15-t100`. We have also given you the full CPD file for a fourth model, `grid10x10-t10`, in order to test your code. The file `training-grid10x10-t10.txt` was generated by sampling from the CPDs in `cpd-grid10x10-t10.txt`. Therefore, parameters that you estimate on this training data should closely (but not perfectly) match this CPD file, and you can use it as a sanity check while testing your program. Your probabilities should match this CPD file to 1 decimal place.

You can write your program in whatever programming language you like, as long as it can be executed on the `ugrad` servers, but you must develop an executable named `estimate-params`. As before, this executable should simply be a shell script which acts as a wrapper to your program. Remember that you need to run the command `chmod +x estimate-params` in order to make the script an executable file that we can run.

If you use Matlab, please use the following script:

```
matlab -nodisplay -nosplash -r "try, estimate_params '$1' '$2' '$3',
catch err, disp(err), fprintf('ERROR\n'), end, quit"
```

3.1.5 Analytical Questions [10 points]

Consider the advantages and disadvantages to using the shared parameterization (3.1.1).

1. [**3 points**] An important advantage of sharing parameters is the lessening of the effects of *overfitting*. Explain why. Why is this issue important when doing parameter estimation in this assignment?
2. [**3 points**] Describe at least one property of the robot localization setting (with regard to either the motion model or observation model) that you cannot model with the shared parameterization in this assignment, but that you could model if we used an explicit parameterization (i.e. we learn the entire CPT for particular positions or time steps). It does not have to be a property of the environment that we described in this handout; you can construct something relevant that you might like to model in this setting.
3. [**4 points**] Describe a way to combine the advantages of both approaches. That is, come up with a model design that can potentially learn unique values for all CPT entries, but also has shared parameters which reduce overfitting. Sketch the idea in a few sentences, and explain the advantages over the other models.

3.2 Inference [90 points] [20 extra credit]

This section consists of two parts. First, you will construct clique trees for the networks of interest. Second, you will create a program that can output solutions to sets of queries, given a clique tree as input. You will implement the *belief update message passing* algorithm for exact inference.

3.2.1 Analytical Questions: Clique Tree [20 points]

The models in this assignment have a particular structure. That is, given the map dimensions, the set of landmarks, and the maximum number of time steps in a trajectory, we can fully specify the set of variables that make up each model.

While it is possible to generate a clique tree for any arbitrary graphical model, we only want you to create a clique tree for these three models we have provided:

- `grid10x10-t100.txt`
- `grid10x10-t1000.txt`
- `grid15x15-t100.txt`

You should do this by first considering what the form of a clique tree for these models will be on pencil and paper.

Two typical approaches to converting a Bayesian network to a clique tree are the Variable Elimination algorithm and direct manipulation of the network. Figure 10.10 in Koller and Friedman gives an example of the latter, which is usually carried out in four steps:

1. Convert the directed graph to an undirected graph. That is, moralize the graph by "marrying" the parents of each node and converting directed edges to undirected edges.
2. Create a chordal graph by triangulation. This amounts to removing any chordless cycles of 4 or more nodes (see Definition 2.24, Section 9.4.3.1, and Section 10.4.2 in the textbook). Note that this triangulation must be chosen carefully so as to ensure that the *running intersection property* holds in the resulting clique tree.
3. Find the maximal cliques in the chordal graph.
4. Form a cluster graph over these cliques, where each edge $(\mathbf{C}_i, \mathbf{C}_j)$ weighted by the number of variables their have in common, $|\mathbf{C}_i \cap \mathbf{C}_j|$.
5. Find the maximal spanning tree over this cluster graph, and use it as a clique tree.

After you know the form of a clique tree for these models, you must create a `cliquetree` file representing each one. How you create these files is up to you.[2] Again, you are *not* expected to implement a procedure for finding clique trees on arbitrary graphical models.

[2] We found it only took a few for loops and about a few dozen lines of Python to generate cliquetrees for this model, after we had worked out the form of the trees by hand.

Analytical Questions:

1. **[2 points]** Describe the process you followed to make the clique tree. Note one or two points in the process where you could have obtained a (slightly) different clique tree by making different choices.

2. **[2 points]** Demonstrate that the running intersection property holds in the resulting clique tree. A formal proof is not required, simply explain how this property holds.

3. **[2 points]** Describe how you produced the `cliquetree` file.

4. **[4 points]** For a grid of size N by M and a sequence of T actions and observations (assume they are fully observed), what would be the computational complexity of computing the distribution over the final position at time T if we simply marginalized over a joint CPT? What is the computational complexity of computing the distribution over the final position at time T using message passing in your clique tree?

5. Suppose we wanted to query for the robot state at time 5 and time 15.

   (a) **[7 points]** How would you modify the clique tree so that you could make this query? Prove that your modification is guaranteed to give the correct marginal probability. Hint: to ask a query, there must exist a cluster which contains all of the query variables.

   (b) **[3 points]** Why is it not always valid to answer this type of query using two different clusters where each contains one of the query variables?

3.2.2 Deliverables: Clique Trees [10 points]

For each of the network files `network-grid`I`x`J`-t`T`.txt` provided, you should create a `cliquetree` file named `cliquetree-grid`I`x`J`-t`T`.txt`. If you wrote a script to generate the clique tree files, include that as well.

The `cliquetree` file The `cliquetree` file should specify the structure of an undirected clique tree. The first line contains $|\mathcal{T}|$ the number of cliques (i.e. clusters) in the tree. Each of the following $|\mathcal{T}|$ lines contain a cluster specification, which is simply a comma separated list of variables in that cluster. The remaining lines define undirected edges between clusters, and have the form `cluster1 -- cluster2` , where `cluster1` and `cluster2` are comma separated lists of variables. The file `cliquetree-grid10x10-t10.txt` has been provided as an example of the format (and therefore, you don't need to generate to generate a clique tree for that network file, just the other three).

3.2.3 Inference by Belief Update Message Passing

Now that you've created clique trees for the robot model, you can begin to do inference over those clique trees. You will implement the *belief update message passing* algorithm. This message passing algorithm produces calibrated clique trees, which you will use to answer multiple within-clique queries about the marginal distributions of the variables in the model.

Before you begin, you should read chapters 9 and 10 in Koller and Friedman. Both the theory and practice of exact inference for graphical models of any structure is laid out in these chapters. Your implementation of exact inference will do the following:

- Construct the clique tree using the structure defined in the `cliquetree` file and the graphical model given by the `network` and `cpd` files.

- Compute the sepsets $\mathbf{S}_{i,j} = \mathbf{C}_i \cap \mathbf{C}_j$ for each edge (see Definition 10.2).
- Initialize the beliefs β_i and $\mu_{i,j}$ corresponding to the cliques $\mathbf{C}_i$ and sepsets $\mathbf{S}_{i,j}$ respectively.
- Run belief update message passing, as described in section 10.3 of the textbook, to produce a *calibrated clique tree*. This requires that the algorithm be ordered according to a message passing schedule that ensures convergence to the true unnormalized marginal distributions for each clique. Section 10.3.2 proves that an upward pass and a downward pass based on some arbitrarily chosen root node ensures convergence.
- Read a sequence of queries from a `queries` file.
- For each query, update the factors to reflect the evidence given in each query. There are two cases you must handle (see below for examples):
  - The first case is when the new evidence is *additive*. Additive evidence keeps all existing variable assignments (if any) that are already accounted for in the clique tree, and optionally adds new variable assignments. For this case, you will perform *incremental updates* as described in Section 10.3.3.1. Crutially, you should not reinitialize the clique tree, but should instead apply the appropriate updates, and then run belief update message passing to recalibrate the tree.[3]
  - The second case is when the new evidence is *retractive*. Retractive evidence removes variable assignments that is currently accounted for in the calibrated clique tree, and optionally adds some others. You are not expected to do the additional bookkeeping required to remove evidence from the clique tree. Instead, you may simply reinitialize the beliefs to their start state and then tree the evidence as additive.
- Answer multiple within-clique queries by renormalizing the beliefs of the calibrated clique tree and summing over the variables the relevant clique that are not involved in the query.

3.2.4 Deliverables: Message Passing [60 points]

You will create and submit a program called `bayes-query-sp`. This program will answer queries about the marginal distributions of a model using belief propagation. Note that the program will accept multiple queries which can be (efficiently) answered in one call.

The program `bayes-query-sp` will take network files and queries as input and will output the corresponding probability. The program will be executed with commands of the following format:

```
./bayes-query-sp network_file cpd_file cliquetree_file queries_file
```

The `network` file and the `cpd` file should have the same format as in the parameter estimates question, and will fully specify a Bayesian network. The `cliquetree` file specification is given above.

[3] To introduce evidence by incremental updates, you will keep the clique tree the same, but will simply multiply in an additional indicator factor to reflect the evidence. For each clique that contains a variable for which there is some evidence, you will simply multiply the clique by an indicator function on the observed value of that value. After this multiplication is carried out, the clique tree will no longer be calibrated and it will be necessary to run belief update message passing to convergence again, though possibly on a subset of the nodes.

The `queries` file Each line of the `queries` files will have the following format: the left-hand and right-hand sides will follow the same format as the `cpd` file. A query file containing three queries will have the following form, where the intervening queries are elided.

```
lefthand_side_1 [righthand_side_1]
lefthand_side_2 [righthand_side_2]
...
lefthand_side_n [righthand_side_n]
```

When the `bayes-query-sp` program is run on a `queries` file containing n queries, the last n lines of the output should contain the result for each query, in the same order they were given.

Important Note on Query Types We place an important restrictions on the types of queries that will be given as input to your program. When grading your `bayes-query-sp` program, we may use arbitrary graphical model input. However, we will restrict our queries to have a `lefthand_side` that only contains variables within a single clique. That is, you do not need to implement the textbook's *Algorithm 10.4 Out-of-clique inference in a clique tree.*

Creating the `bayes-query-sp` program You can write your program in whatever programming language you like, as long as it can be executed on the `ugrad` servers, but you must develop an executable named `bayes-query-sp`. As before, this executable should simply be a shell script which acts as a wrapper to your program. Remember that you need to run the command `chmod +x bayes-query-sp` in order to make the script an executable file that we can run.

If you use Matlab, please use the following script:

```
matlab -nodisplay -nosplash -r "try, bayes_query_sp '$1' '$2' '$3' '$4',
catch err, disp(err), fprintf('ERROR\n'), end, quit"
```

Testing and Debugging The 10x10 map is small enough that you should be able to work out the solution to simple queries on paper. We recommend coming up with the solution first, then verifying that your inference program gives the same solution. You are also free to write your own input files to check your solutions on other maps.

Again, it's good to check that your probabilities fall in the range $[0, 1]$ and that the sum over all combinations of values to the left-hand variables is 1.0.

Read Box 10.A from the textbook and make sure you aren't getting stuck on any of the pitfalls it discusses. For example, you may want to work with log-probailities instead of probabilities.

You may find it helpful to write a few lines of code to check that the clique tree file contains a valid clique tree for the network.

Feel free to use your own simpler examples when debugging.

3.2.5 Extra Credit: Max-Product Message Passing [20 extra credit]

In this assignment you implemented belief update message passing, a variant of sum product message passing. Using the calibrated clique tree produced by this algorithm, you found the marginal distributions over query variables given evidence.

Suppose instead that you wanted to find the maximum likelihood assignment to the variables given the evidence, and the probability of that assignment. This can be accomplished by *max-product* message passing, where the summation in the creation of each message is replaced by a max.

For this extra credit problem, you should implement max-product message passing by augmenting your sum-product message passing program. There are two parts, to this extra credit problem. The first is to simply adjust your algorithm so that you can compute the *probability* of the maximum likelihood assignment to a set of query variables given some evidence. The second is to report the *assignment* of the variables that maximizes likelihood.

Deliverables [20 points] You should create a new wrapper program `bayes-query-mp` that runs *max-product message passing*. In your writeup, include a description of the changes you made to accomplish this form of inference. This program should accept the exact same input files as `bayes-query-sp` with a few caveats. The queries file should contain lines of the form:

```
lefthand_side_1 [righthand_side_1]
lefthand_side_2 [righthand_side_2]
...
lefthand_side_n [righthand_side_n]
```

However, each `lefthand_side` should simply be a comma-separated list of variables (without assignments).

To simplify the problem, we assume the `lefthand_side` will always contain all the variables not in the `righthand_side`. This is the typical setting for max-product and avoids having to marginalize over a subset of the unobserved variables. That is, Let L be the set of variables in the `lefthand_side`, R be the set of variables in the `righthand_side`, and X be the set of all variables. Your `bayes-query-mp` program need only support queries for which the following properties hold: (1) the intersection of L and R is the empty set; and (2) the union of L and R is X. Therefore, probabilities that you print out for `bayes-query-mp` should always be $\max_l P(L = l|R = r)$.

1. **[8 points]** The last n lines of the program should report the probability of the maximum likelihood assignment to the `lefthand_side` variables given the `righthand_side` evidence, for each of the n queries.

2. **[12 points]** Extend your program using backpointers to recover the maximum likelihood assignment to the `lefthand_side` variables given the `righthand_side` evidence. Print out each probability as above, but now followed by the maximum likelihood assignment.