PGM Programming Assignment: Exact Inference

1 Overview

In the first assignment of the Representation course, you implemented a rudimentary inference engine that could correctly answer queries over small networks. Unfortunately, in subsequent programming assignments, your inference engine got passed over in favor of third-party inference engines. It's time to bring your inference engine back!

We have provided you with the correct code for the first assignment's `ComputeMarginal` and all of the functions that it calls, as well as a genetic inheritance network for a 9-person pedigree from the second assignment. At your own risk, try finding the marginal of variable 1 in this network by running the following code:

```
load('PA4Sample.mat', 'NinePersonPedigree');
ComputeMarginal(1, NinePersonPedigree, []);
```

As you can see, the "brute force" inference engine that you implemented in the first assignment is unable to handle anything larger than tiny networks; its running time is proportional to the number of entries in the joint distribution over the entire network, which even in this small 9-person example is on the order of 10^7.

Where brute force inference fails, however, other forms of exact inference can still be very efficient. We will explore clique tree message passing in this assignment, and by its end, you will have created an inference engine powerful enough to handle probabilistic queries and find MAP assignments over the genetic inheritance networks from the second programming assignment and the OCR networks from the third programming assignment, respectively.

2 Belief Propagation

Let's start by taking a look at the belief propagation algorithm, specialized to exact inference over clique trees:

1. Construct a clique tree from a given set of factors Φ.
2. Assign each factor $\phi_k \in \Phi$ to a clique $C_{\alpha(k)}$ such that $\text{Scope}[\phi_k] \subseteq C_{\alpha(k)}$. $\alpha(k)$ returns the index of the clique to which ϕ_k is assigned.
3. Compute initial potentials $\psi_i(\mathbf{C}_i) = \prod_{k:\alpha(k)=i} \phi_k$.
4. Designate an arbitrary clique as the root, and pass messages upwards from the leaves towards the root clique.
5. Pass messages from the root down towards the leaves.
6. Compute the beliefs for each clique: $\beta_i(\mathbf{C}_i) = \psi_i \times \prod_{k \in N_i} \delta_{k \to i}$

As an example, consider the following genetic inheritance network:

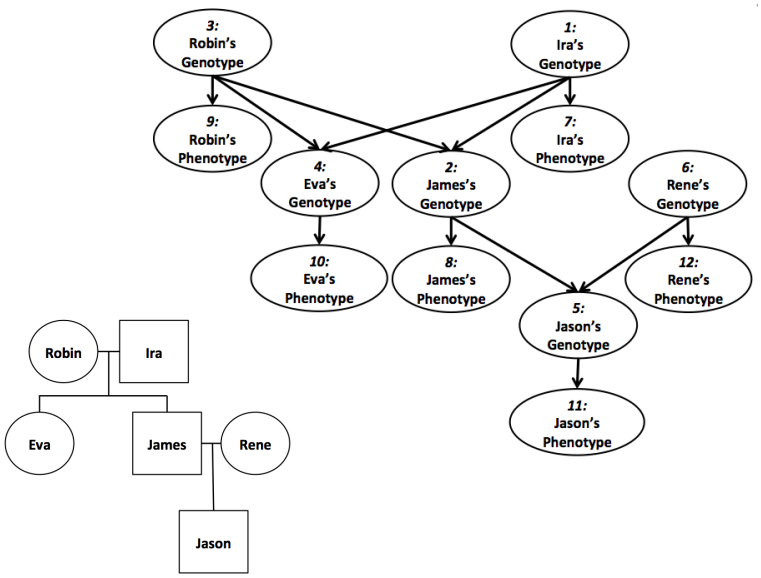

Figure 1: A pedigree of 6 people and its corresponding Bayesian network.

From the list of factors corresponding to this network, we can create the following clique tree:

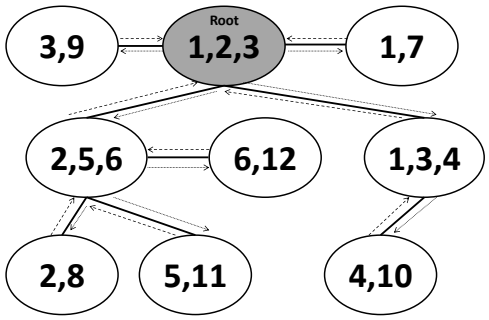

Figure 2: One possible clique tree for the genotype network above.

Next, we initialize the clique potentials in this tree by assigning each of the original factors to a clique. Arbitrarily choosing the clique with variables 1, 2, and 3 to be the root, we can start passing messages up from the leaves to the root, and then down from the root to the leaves. In this clique tree, there are 9 cliques, so $2 \cdot (9 - 1) = 16$ messages suffice to correctly compute all beliefs**.** The dashed lines show the messages from the leaves to the root, and the dotted lines show the messages from the root to the leaves. Finally, we can use the calibrated clique beliefs to answer probabilistic queries on the original network.

In the subsequent sections we will work our way through the steps of this algorithm, implementing two variants: sum-product message passing and max-sum message passing.

3 Clique Tree Construction

In this and future assignments, a `CliqueTree` is a `struct` with the following fields:

- `cliqueList` – This is a struct array of cliques. The representation of a clique is exactly like that of a factor (which you have seen in Assignments 1-3). The `.val` for each clique i should be initialized to its initial potentials ψ_i.
- `edges` – This is an adjacency matrix representing edges between cliques in the clique tree. It is an $n \times n$ matrix, where n is the number of edges. If clique i and clique j share an edge, then `.edges[i,j]` and `.edges[j,i]` are both 1; else, they are 0.

To perform clique tree message passing, we first need to take in a list of factors and construct an appropriate clique tree. We have provided a `CreateCliqueTree` function that creates a clique tree using a modified variable elimination procedure, as described in the "Clique Trees and VE" lecture.

`CreateCliqueTree` relies on the `ComputeInitialPotentials` function, which you will have to write:

- **ComputeInitialPotentials.m (10 points)** – This function should take in a set of factors (possibly already reduced by evidence, which is handled earlier in `CreateCliqueTree`) and a clique tree skeleton corresponding to those factors, and returns a `cliqueTree` with the `.val` fields correctly filled in. Concretely, it should assign each factor to a clique, and for each clique do a factor product over all factors assigned to it. Note that no messages should be computed in this function. The functions in section 4 will accomplish that task. For debugging, you can look at the sample input clique tree `InitPotential.INPUT` and the sample output clique tree `InitPotential.RESULT` in `PA4Sample.mat`.

The clique tree skeleton that `CreateCliqueTree` produces satisfies both family preservation and the running intersection property, so you will not have to worry about writing error-checking / input-validation code.

4 Message Passing in a Clique Tree

4.1 Message Ordering

With our correctly-initialized clique tree in hand, we now need to come up with a correct message passing order. Recall that a clique is ready to transmit a message upstream toward the root after it has received all of the messages from its downstream neighbors, and vice versa; it is ready to transmit a message downstream after it has received its upstream message. (Since we're working with clique trees, each clique will only receive one upstream message.)

More generally, we say that a clique C_i is ready to transmit to its neighbor C_j when C_i has received messages from all of its neighbors except from C_j. In clique tree message passing, each message is passed exactly once. The easiest way to come up with a valid message passing order is to write the following function:

- **GetNextCliques.m (10 points)** – This function looks at all the messages that have been passed so far, and returns indices i and j such that clique C_i is ready to pass a message to its neighbor C_j. Do not return i and j if C_i has already passed a message to C_j. For debugging, you can look at the sample input clique trees `GetNextC.INPUTx` and the sample output clique trees `GetNextC.RESULTx` in `PA4Sample.mat`.

Note that there are computationally cheaper but slightly more involved methods of obtaining the correct message passing order over the tree. The main computational bottleneck in our message passing scheme does not lie in calculating the message passing ordering, though, so this has little effect in practice.

4.2 Sum-Product Message Passing

We can now begin calibrating the clique tree. Recall that a clique tree is calibrated if all pairs of adjacent cliques are calibrated. Two adjacent cliques C_i and C_j are calibrated if they agree on the marginals over their shared variables, i.e.,:

$$\sum_{\mathbf{C}_i - \mathbf{S}_{i,j}} \beta_i(\mathbf{C}_i) = \sum_{\mathbf{C}_j - \mathbf{S}_{i,j}} \beta_j(\mathbf{C}_j) = \mu_{i,j}(\mathbf{S}_{i,j}) \quad \forall \mathbf{S}_{i,j}$$

Write the following function:

- **CliqueTreeCalibrate.m (20 points)** – This function should perform clique tree calibration by running the sum-product message passing algorithm. It takes in an uncalibrated `CliqueTree`, calibrates it (i.e., setting the `.val` fields of the `cliqueList` to the final beliefs $\beta_i(\mathbf{C}_i)$), and returns it. **To avoid numerical underflow, normalize each message as it is passed, such that it sums to 1. For consistency with our autograder, do not normalize the initial potentials nor the final cluster beliefs.** This function also takes in an `isMax` flag that toggles between sum-product and max-sum message passing. For this part, it will always be set to 0. For debugging, you can look at `SumProdCalibrate.INPUT and SumProdCalibrate.RESULT` in `PA4Sample.mat`.

Finally, let's bring together all the steps described above to compute the exact marginals for the variables in our network.

- **ComputeExactMarginalsBP.m (15 points)** – This function should take a network, a set of initial factors, and a vector of evidence and compute the marginal probability distribution for each individual variable in the network. A network is defined by a set of factors. You should be able to extract all the network information that you need from the list of factors. **Marginals for every variable should be normalized at the end, since they represent valid probability distributions.** As before, this function takes in an `isMax` flag, which should be set to 0 for now (at this point, you need to write the function for only when isMax=0). For debugging, you can look at `ExactMarginal.INPUT and ExactMarginal.RESULT` in `PA4Sample.mat` and use Evidence = [] (the sample case makes this assumption). You may also try different values for Evidence if you like.

Congratulations! You have successfully implemented an efficient exact inference algorithm. Now you can use the factors for the genotype network from PA2 and run inference using your own inference engine. You can compare the marginals that you get to the ones from the inference engines that were provided in PA2.

To test your inference engine on the genotype network shown in the figure above, run

```
load('PA4Sample.mat', 'SixPersonPedigree');
ComputeExactMarginalsBP(SixPersonPedigree, [], 0);.
```

This 6-variable network is small enough that we can use the brute-force implementation in Assignment 1 to double check the results. Use

`ComputeMarginal(1, SixPersonPedigree, []);`

With evidence you can try, `M = ComputeExactMarginalsBP(SixPersonPedigree, [1], 0)` and compare `M(5)` with `ComputeMarginal(5,SixPersonPedigree,[1 1])`.

Note that CreateCliqueTree builds an evidence vector already.

Play around with observing evidence, and make sure that it works!

4.3 Max-Sum Message Passing

A typical OCR task could be to scan in a hand-written manuscript, decipher the writing, and give others a typed version of the manuscript that they can easily read. In such a scenario, we would want to output the most likely text corresponding to a set of images, instead of calculating marginal probabilities over individual characters. To do this, we will use max-sum message passing to calculate the MAP assignment.

Our aim is to output the most probable instantiation of the variables in the network instead of solving the marginals for each of them. As mentioned in lecture, we will be working in log-space for this part of the assignment; the probability of any single assignment in a large enough network, even the MAP assignment, is typically low enough to cause numerical issues if we do not work in log space. Note that this algorithm is nearly identical to sum-product message passing, with the sums replaced by maxima and the products replaced by sums in the definitions. You should therefore have to write relatively little code.

Just as we used `FactorMarginalization` in the previous algorithm, let's start by writing a function `FactorMaxMarginalization`:

- **FactorMaxMarginalization.m (10 points)** – Similar to `FactorMarginalization` (but with sums replaced by maxima), this function takes in a factor and a set of variables to marginalize out. For each assignment to the remaining variables, it finds the maximum factor value over all possible assignments to the marginalized variables.

  For debugging, in `PA4Sample.mat`, the sample input factor is `FactorMax.INPUT1`, the sample input variables are in `FactorMax.INPUT2`, and the sample output factor is `FactorMax.RESULT`.

With this, modify `CliqueTreeCalibrate` to do max-sum message passing when `isMax=1`:

- **CliqueTreeCalibrate.m (20 points)** – This function should perform clique tree calibration by running the max-sum message passing algorithm when the `isMax` flag is set to 1. It takes in an uncalibrated `CliqueTree`, does a log-transform of the values in the factors/cliques using natural log, max-calibrates it (i.e., setting the `.val` fields of the `cliqueList` to the final beliefs $\beta_i(\mathbf{C}_i)$), and returns it **in log-space**. **We are working in log-space, so do not normalize each message as it is passed. For consistency with our autograder, do not normalize the initial potentials nor the final cluster beliefs.** This function takes in an `isMax` flag that toggles between sum-product and max-sum message passing. For this part, it will always be set to 1, but make sure that it still does sum-product message passing correctly when `isMax=0`.

  In sum-product message-passing, we accomplished the product part of "sum-product" by running `FactorProduct`. You might want to write a similar helper function, `FactorSum`, that accomplishes the sum part of "max-sum" message passing. You can use `if` statements involving the `isMax` flag to decide when to run `FactorProduct` and when to run `FactorSum`.

For debugging, you may look at `MaxSumCalibrate.INPUT` and `MaxSumCalibrate.RESULT` in `PA4Sample.mat`.

Now we are ready to compute the max-marginals for the variables in our network. Modify **ComputeExactMarginalsBP.m** to support max-marginal calculation:

- **ComputeExactMarginalsBP.m (5 points)** – This function should take a network, a set of initial factors, and a vector of evidence and compute the max-marginal for each individual variable in the network (including variables for which there is evidence). Max-marginals for every variable should **not** be normalized at the end. **Leave the max-marginals in log-space; do not re-exponentiate them.** As before, this function takes in an `isMax` flag, which will be set to 1 now; make sure it still works for computing the (non-max) marginals when it is set to 0.

  For debugging, you can look at `MaxMarginals.INPUT` and `MaxMarginals.RESULT` in `PA4Sample.mat`.

Finally, write a function that takes the max-marginals and extracts the MAP assignment:

- **MaxDecoding (10 points)** – This function should take a list of max-marginals returned by ComputeExactMarginalsBP and return a vector `A`, with `A(i)` the value of i^{th} variable in the MAP assignment. You may assume that the max-marginals passed into this function will be unambiguous (so no backtracking etc. is necessary for decoding).

  In `PA4Sample.mat`, the sample input is `MaxDecoded.INPUT`, and the sample output is `MaxDecoded.RESULT`.

To test out your MAP estimation pipeline, let's try it on a sample word from Assignment 3. Run the following commands:

```
load('PA4Sample.mat', 'OCRNetworkToRun');
maxMarginals = ComputeExactMarginalsBP(OCRNetworkToRun, [], 1);
MAPAssignment = MaxDecoding(maxMarginals);
DecodedMarginalsToChars(MAPAssignment)
```

You should see the word "mornings"!

5 Conclusion

Congratulations! You have successfully built an inference engine that can take relatively large networks, such as the OCR network and the genetic inheritance network, and run exact inference on them. Give yourself a pat on the back - you have now written end-to-end programs that can do credit scoring, genetic counseling, and OCR.

However, the powers of exact inference are limited. The next programming assignment will teach you more about that, and explore approximate inference methods that will let us scale to significantly larger networks. Stay tuned!