

CS 228, Winter 2009

Programming Assignment #1—Inference in Graphical Models

In this assignment you will implement the *Sum Product Message Passing* algorithm for exact inference in Graphical Models, and then extend the algorithm to perform *Loopy Belief Propagation* for approximate inference. All code is to be written in Matlab. If you are unfamiliar with Matlab, you should read one of the excellent Matlab tutorials available on the web (for example, http://www.mathworks.com/academia/student_center/tutorials/launchpad.html).

For this assignment, you may work on your own or in a pair. If you need to find a partner, you may post on the course wiki: <http://pmai.wikidot.com/matching>. The assignment is due at **11:59pm on Thursday Feb. 5**. Submission procedures can be found on the course website at <http://www.stanford.edu/class/cs228/submissions.html>.

The primary network that we will use to test your code is the INSURANCE network (shown in Fig. 1 below). The INSURANCE network is a real-world network that was developed for estimating the expected claim costs for a car insurance policyholder. There are 27 variables in the network with between two and five values per variable. If you are curious, you can find the original paper that mentioned this network at <http://citeseer.ist.psu.edu/binder97adaptive.html>. You will be provided with a description of the network that can easily be read into Matlab (see problem 1 below). We also provide a constructed clique tree and cluster graph for the network so that your implementation will simply focus on implementing inference over these structures.

Because your code will be general enough to handle other networks, and in order to give a more practical flavor to the power of inference, we also provide a “Things and Stuff” (TAS) network, which is based on recent research by Jeremy Heitz, a PhD student of Prof. Koller. (If you’re interested, you can find the paper, examples, and other materials at

<http://ai.stanford.edu/~gaheitz/Research/TAS/>). Details on the TAS networks are below.

1. [60 points] Code

All code for this assignment should be copied from `/afs/ir/class/cs228/ProgrammingAssignments/PA1/assignment`. You should be able to access this by logging into the Stanford Unix machines (e.g., `cardinal.stanford.edu`) using your SUNet ID. You can also use a copy of Matlab on any of the machines that allow CPU-intensive jobs (tree, vine, bramble, hedge).

Your job is to fill in the missing portions of code – marked with `YOUR CODE HERE` – in the files described below. The script `TestPA1.m` is the main routine that sets up the tests to be run and also evaluates whether your implementation is correct. Initially, running this script without making any code changes should result in all tests failing (though they should run without error).

The tests run in `TestPA1.m` and the function that must be implemented for each one are enumerated below. For each test, sample inputs and outputs are loaded in, and the test passes if the output produced by your implementation matches the correct outputs that are loaded in. Note that when you submit your final implementation it will be tested with different inputs and outputs.

- (a) [1 points] **Factor Product**: This test simply checks whether your implementation of factor multiplication is correct, using two sample factors contained in `INPUT.Factors`.
- **FactorProduct.m** — This function should compute the product of two factors.
- (b) [1 points] **Factor Marginalization**: This test checks whether your implementation of factor marginalization is correct, using factors from `INPUT.Factors`.
- **FactorMarginalization.m** — This function should compute the factor after summing out a given variable in a given factor.
- (c) [1 points] **Factor Product**: Same as Test 1, with larger factors.
- (d) [1 points] **Factor Marginalization**: Same as Test 2, with larger factors.
- (e) [1 points] **Observe Evidence**: This test checks whether your implementation of observing evidence for a given set of factors is correct.
- **ObserveEvidence.m** — This function should modify a set of factors given the observed values of some of the variables.
- (f) [20 points] **Calibrate Clique Tree**: This test checks whether your implementation of clique tree calibration is correct. It has 5 subtests, worth 4 points each. For each subtest, a sample instantiation of evidence is used from `INPUT.Inference.EVIDENCE`. The observed evidence is encoded in the network, the clique tree is calibrated, and final beliefs (factors) for each clique in the tree are compared to the correct factors in `OUTPUT.Inference.Exact.RESULTS`.
- **FindReady.m** — This function should find a clique that is ready to transmit a message to its neighbor. It should return the indices of the two cliques the message is ready to be passed between.
 - **CliqueTreeCalibrate.m** — This function should perform clique tree calibration by implementing the Sum Product Message Passing algorithm. It should return an array of final beliefs (factors) for each clique.
- (g) [5 points] **Exact Inference**: This test checks whether your implementation of exact inference is correct. It has 5 subtests, as in Test 6, worth 1 point each. For each subtest, after the clique tree is calibrated, the marginals of each variable in the network are computed and compared to the correct marginals in `OUTPUT.Inference.Exact.RESULTS`.
- **ExactInference.m** — This function should take a clique tree, set of initial factors and vector of evidence and compute the marginal probability distribution for each variable in the network.
- (h) [25 points] **Approximate Inference**: This test checks whether your implementation of approximate inference is correct. Once you have successfully implemented Exact Inference, it should be relatively simple to implement Approximate Inference. The test has 5 subtests, as in Tests 6 and 7, worth 5 points each.
- **ClusterGraphCalibrate.m** — This function should perform loopy belief propagation over a given cluster graph. It should return an array of final beliefs (factors) for each clique.
 - **ApproxInference.m** — This function should take a cluster graph, set of initial factors and vector of evidence and compute the marginal probability distribution for each variable in the network.

Now that you have successfully implemented inference, we will use it to solve a (somewhat) real-life application. The goal of a TAS network is to find objects in real images by using context. Both examples you consider will be satellite photos of a city (see Fig. 2), and the task is to put a bounding box around all the cars in the image. The context being leveraged is the background “stuff” such as roads, houses, and trees — this will take advantage of the fact that, for example, cars are more likely to appear on roads than in trees.

The TAS model you use will build on the output of two existing computer vision algorithms. The first is a bounding box detector that is specifically trained to put a box around an object of interest (in your case, either cars or boats) in an image, without considering any context. These candidate bounding boxes (colored by score) are shown in (see Fig. 2), and you can see that the candidate boxes proposed by this algorithm are not as accurate as one might hope — there are boxes that don’t contain the object of interest but have high scores (false positives) and boxes that contain the object but don’t score very high (false negatives). The TAS network aims to improve on these candidate detections by using context to lower the scores of the false positives and strengthen the scores of the false negatives. Note that, for the purpose of this exercise, we have limited the total number of candidates to allow for faster inference; a typical bounding box detector might return a few hundred candidate detections.

The second algorithm whose output you will use is a segmentation labeler. This algorithm first splits the image into regions, and then labels each region. For example, in the first satellite image, we provide you with 7 segments, each of which has been labeled as either “road,” “roofs,” or “trees.” Note that, for the purpose of this exercise, we have given you unrealistically good output; typical state-of-the-art methods yield segments that cut across true segment borders, and the labels are not always accurate.

The TAS network is shown in Fig. 3. The variables T_i are binary and represent whether the object of interest (e.g., car) is in window i . The variables S_j take on several values, representing the type of background in segment j (e.g., “road”). The variables R_{ij} represent the spatial relationship between window T_i and segment S_j ; in our case we restrict this variable to have one of three values: “in” (the window is in the segment), “near” (the window is within 40 pixels of the segment), and “far” (the window is farther than 40 pixels from the segment). The CPDs for this network have been prepared for you: $P(T_i = 2)$ (the assignment $T_i = 2$ means that it is a car, whereas $T_i = 1$ means that it is not) is the score assigned by the bounding box detector to window i (the score is between 0 and 1); $P(S_j)$ is likewise the probability distribution given by the segment labeler; and $P(R_{ij}|T_i, S_j)$ is defined intuitively so that, for example, cars are more likely to appear in roads and near roofs.

You can visualize the labels that the segmenter produces (and that you use as priors on S_j) by executing:

```
imagesc(image_data.seglabels);
```

The scores that are output by the bounding box detector are used as priors on T_i ($P(T_i)$). We can visually evaluate the priors visually by executing:

```
view_detections(image_data,image_data.scores,0.8);
```

In this visualization, the bounding boxes are color coded by the score of the detector, and all scores below a given threshold (0.8) yield boxes with dotted borders.

For a more quantitative analysis, you can see an ROC curve by executing:

```
det_rpc(image_data, image_data.scores);
```

This curve shows how many false positives and false negatives we get as we vary the threshold. Each point on the curve corresponds to a threshold and shows the precision (y-axis) versus the recall (x-axis).

After inference, we hope that the posteriors over T_i ($P(T_i|\mathbf{R})$) will provide superior “scores” than the priors. Once we have run inference, we can evaluate the posteriors in a similar manner, using them in the place of the prior scores.

Did your results improve?

TestPA1.m executes inference on the two TAS images provided, and awards **[5 points]** for correct outputs on the two images. In order for the test to pass, you must fill in code in **TestTAS.m** to populate the posteriors. Note that `view_detections` is called twice for each image, and each time requires a keystroke or mouse click to continue. There is also an extra credit question below (the last part of **2f**) that asks you do some more coding.

2. [40 points] Questions

(a) [3 points]

- i. [1 points] If you observe that the driver is a psychopath (`RiskAversion = 1`), what happens to the marginal distribution over Theft (`1 = True, 2 = False`), compared to the same distribution with no observations?
- ii. [1 points] What happens to the marginal distribution of Theft if you observe that the driver is a psychopath AND is a good student (`GoodStudent = 1`)?
- iii. [1 points] What extra active path(s) between `RiskAversion` and `Theft` are opened by observing `GoodStudent` (as opposed to not observing `GoodStudent`)?

(b) [2 points] If the full joint probability over all variables in the Insurance network were expressed in a single table, how many entries would it contain? How about the two TAS networks?

(c) [3 points] What is the size of the largest clique (measured by the number of entries in the belief for that clique) in the provided clique tree of the three networks? What variables do those largest cliques contain? What is the size of the largest cluster (by the same measure) in the provided cluster graphs?

(d) [3 points] Can you use the same clique tree for the two TAS networks? What about cluster graph? If you can use the same graph, explain how, and if not, explain why not.

(e) [4 points] In which situations should you choose approximate inference over exact inference?

(f) [14 points] In exact inference, we use `FindReady.m` to define a specific message-passing order by choosing a ready clique to pass a message. Multiple messages may be ready at once, allowing some flexibility in the ordering.

- i. [3 points] Can there be a difference between two such orderings in the final marginals? in the running time of exact inference?

- ii. [3 points] If the message ordering is instead chosen randomly and exact inference is run until convergence, can there be a difference in the final marginals? in the running time?

The code skeleton we provide for you in `ClusterGraphCalibrate.m` defines a specific but rather arbitrary order of message-passing for approximate inference.

- i. [4 points] If we choose a different ordering of message-passing for approximate inference, can there be a difference in the final marginals? in the running time until convergence?
 - ii. [4 points] The function `CheckConvergence` uses the difference between a message before and after it is updated (called the “residual”) as a criterion for convergence. Print out and plot the residuals of the message $29 \rightarrow 1$, $35 \rightarrow 6$, and $27 \rightarrow 54$, with the iteration number on the x-axis. Do these messages converge at the same rate? Describe qualitatively their convergence behavior relative to one another.
 - iii. [5 points] (Extra Credit – see course website for extra credit policy) These observations suggest a particular message ordering. Implement such a schedule. In order to do this, you will need to calculate what a message *would be* in the next iteration without actually passing it yet. Analyze how this schedule affects convergence. It is okay if the overhead of the new schedule actually causes convergence to be slower in terms of time; instead, determine the effect on the number of messages passed before convergence. If you want to reduce the overhead so that it doesn’t run too slowly, you may need to cache certain values and only recompute them as needed; this is completely optional for the purpose of this problem.
- (g) [1 points] What is the maximum difference between an entry in a final marginal from exact inference and the same entry in the marginal from approximate inference? What is the average such difference?
 - (h) [4 points] What happens to the detection results when you observe the correct segmentation labels? Give a qualitative evaluation of the detections and of the ROC curve. What happens if you observe Segment 2 incorrectly as “road” in the second TAS network?
 - (i) [4 points] If you observe the segmentation labels, what extra independencies arise in the network (compared to observing only the relations)? Give a simpler way to perform inference in this case, providing the necessary equations.
 - (j) [2 points] Does TAS help the segmentation as well? Compare the posterior beliefs over each segment to the prior beliefs. Does the correct assignment have a higher probability for each segment in the posterior?

You must include the answers to these questions in an ASCII text file called `README` when you submit your code.

Details

You have been provided with the following code and data:

1. **insurance.mat** — Data structures defining the INSURANCE network. Includes definition of the Bayesian Network, Clique Tree for exact inference and Cluster Graph for approximate inference. Also defines the initial set of factors (CPTs) for the INSURANCE network.

2. **ViewGraph.m** — This function interfaces to the external application `dot` for visualization of directed and undirected graphs. Call with `ViewGraph(G)` where `G` has fields `.names`, `.nodes`, and `.edges` (see the details section below). Note that this function may not work on some machines because of issues with installation of the visualization software. It should work from `myth.stanford.edu`, so try to ssh there if you want to do any visualization.
3. **GetValueOfAssignment.m** — Given a full assignment to the variables of a factor, this function returns the value for the assignment.
4. **SetValueOfAssignment.m** — This function sets the value of a full assignment to the variables in a factor to a given value. Returns the factor with the new value assigned, e.g. `phi = SetValueOfAssignment(phi, [1 1 2], 0.1)`; sets the value $\phi(1, 1, 2) \leftarrow 0.1$.
5. **AssignmentToIndex.m** — Utility function to convert from a variable assignment to a corresponding index into a factor table.
6. **IndexToAssignment.m** — Utility function to convert from an index into a factor table to a corresponding variable assignment.

We have provided a lot of the infrastructure code for you so that you can concentrate on the details of the inference algorithms. The file `insurance.mat` contains four Matlab data structures¹:

- `insurance_network` contains the definition of the INSURANCE network;
- `insurance_clique_tree` defines (one possible) Clique Tree for the INSURANCE network to be used for exact inference; and
- `insurance_cluster_graph` defines (one possible) Cluster Graph for the INSURANCE network to be used for approximate inference.
- `insurance_factors` is a vector containing the initial factors for the INSURANCE network.

The files `TAS1.mat` and `TAS2.mat` each contain similar structures for the TAS network. They also contains data for the image in the structure `image_data`, which has the following fields:

- `dets`: Each row has the x and y coordinates of two corners of the bounding box of a single candidate detection
- `scores`: Has the scores corresponding to each candidate detection, as given by the object detector
- `gt`: The true detections
- `image_filename`: The name of the corresponding image
- `segindices`: A matrix specifying which segment each image pixel belongs to
- `seglabels`: A matrix specifying the label of the segment of each pixel as assigned by the segmenter

¹Use the Matlab command `load insurance.mat` to load these variables.

See above for how to visualize these elements.

The function **ViewGraph.m** can be used to visualize different graphs. (As noted above, you may need to do this from myth.stanford.edu.) More detail on how to use the data structures is provided below.

All of the code skeletons are provided for you. You should look at the comments in the code to guide you on what to implement. You are free to implement the algorithms however you like as long as you maintain the interfaces to each function. In particular, **TestPA1.m** should run (as supplied) without crashing since we will be using a similar script for grading your submission.

Your first step should be to implement the **FactorProduct.m** and **FactorMarginalization.m** functions. You should test these and make sure they work before attempting to implement the Sum Product Message Passing algorithm. Next implement the **ObserveEvidence.m** function. This function should take a set of factors and modify them by zeroing out the illegal variable combinations given the evidence. Finally implement the **CliqueTreeCalibrate.m** function.

Now that exact inference is working, it should be a fairly simple change to implement **Cluster-GraphCalibration.m** for approximate inference. Make sure you renormalize the messages at each iteration to prevent overflow.

Network Data Structure

A graph $G = (V, E)$ is defined by a set of vertices, V , and edges, E . One way to represent the edges is using an *adjacency matrix*, A , where

$$A_{ij} = \begin{cases} 1 & \text{if there is a directed edge from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

A graph is undirected if for every edge (u, v) there is an edge (v, u) , which implies $A = A^T$.

We use the Matlab **struct** with the following fields to define a network:

- **.names** provides the names of each variable. For example, given graph G , the number of variables is `length(G.names)` and the name of the i -th variable is `G.names{i}`.
- **.dim** is a vector holding the dimensionality of each variable.
- **.nodes** provides a cell array of cliques (nodes in the graph). A node is a vector containing indices of variables. For example `G.nodes{1}` may return `[2 5 7]` indicating the clique over variables (X_2, X_5, X_7) .²
- **.edges** is an adjacency matrix over the nodes of the graph.

Factor Data Structure

In the code we will use Matlab structures to implement the factor datatype. The code

```
phi = struct('var', [1 2 4], 'dim', [2 2 3], 'val', ones(12, 1));
```

²It is worth making a special note about working with cell arrays in Matlab. Cell arrays are very useful because they allow different size matrices to be placed into a single array. When braces (e.g. `G.nodes{i}`) are used to access a cell array, the contents of the cell is returned; when brackets (e.g. `G.nodes(i)`) are used, the 1-by-1 cell is returned. You will usually want to use braces.

creates the all-ones factor, $\phi(X_1, X_2, X_4)$, with $\dim(X_1) = 2$, $\dim(X_2) = 2$ and $\dim(X_4) = 3$. An assignment to the variables (X_1, X_2, X_3) is implemented using a vector, so the assignment $(X_1 = 1, X_2 = 2, X_3 = 1)$ is `[1 2 1]` in Matlab. You can use the provided functions **SetValueOfAssignment.m** and **GetValueOfAssignment.m** to set and get the value of an assignment in the factor, ϕ . Note that *the order of the variables is defined by the factor and not the assignment* so when working with more than one factor, you may need to map the order of variables between the factors.³

³Hint: you can use `setdiff(A.var, B.var)` to find all the variables in factor A that are *not* in factor B , and `intersect(A.var, B.var)` to find all the variables in factor A that are also in factor B .

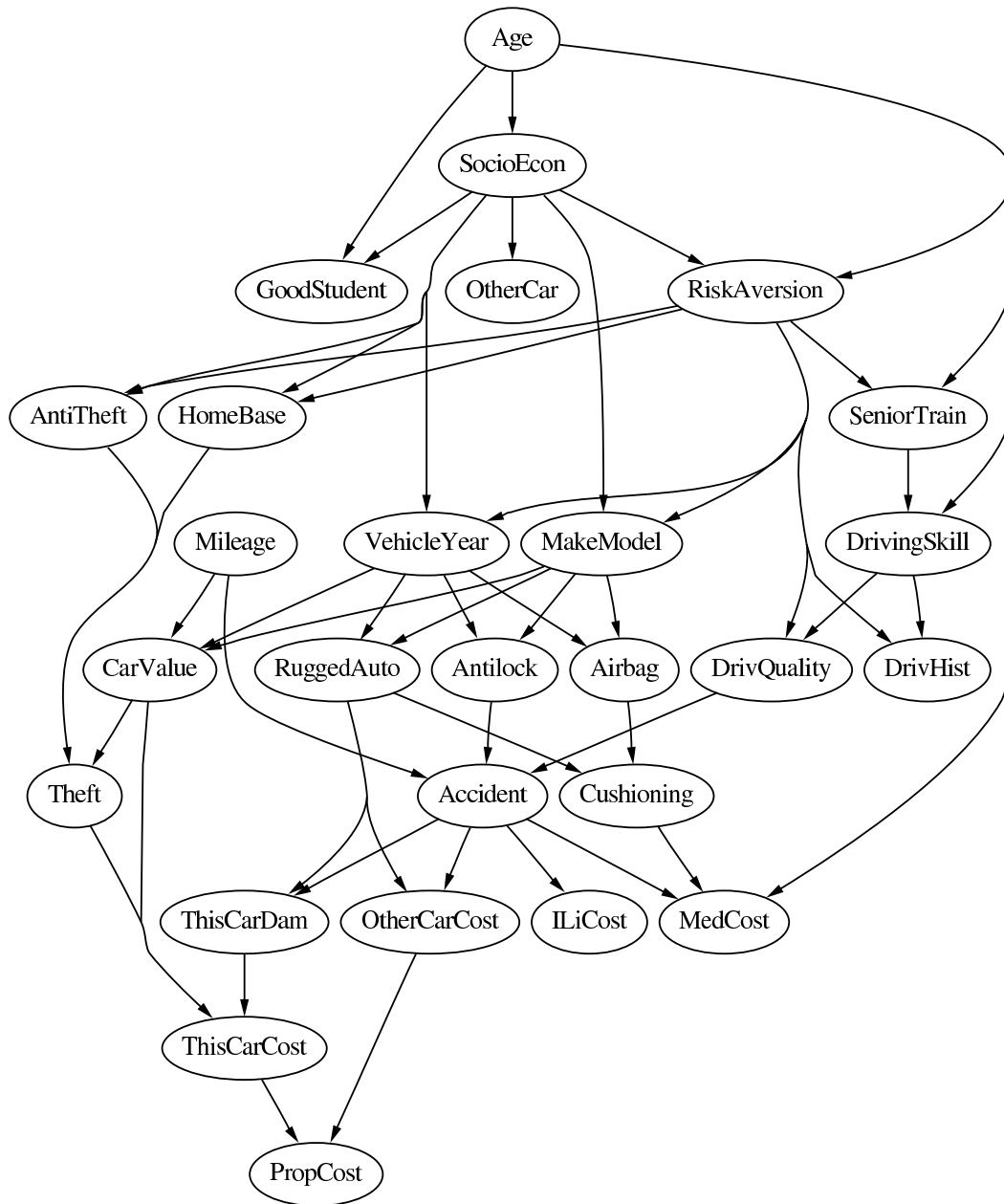


Figure 1: INSURANCE Network for Estimating Expected Insurance Claim Costs.



Figure 2: Satellite image for TAS network.

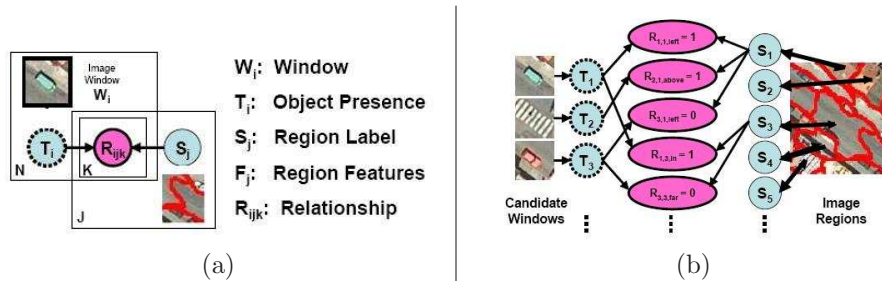


Figure 3: TAS network