# CS 228, Winter 2010-2011 Programming Assignment #2—Inference

**This assignment is due at 11:59pm on February 17**

## 1 Introduction

In this assignment, you will implement several methods for conducting inference in probabilistic graphical models. You will begin by implementing a message passing framework that will perform *sum product message passing* over both a clique tree and a loopy cluster graph. Then, you will implement two *Markov chain Monte Carlo* sampling methods for inference – *Gibbs sampling* and *Metropolis-Hastings*. Finally, you will test and apply these techniques on two probabilistic networks.

You may work on your own or in a pair. 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 (e.g. www.mathworks.com/academia/student_center/tutorials/launchpad.html. The assignment is due on **February 17 at 11:59pm**.

One of the networks 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 2–5 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 section 4 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.

The other network that will be used to test your code is a simple pairwise Markov network. This Markov net is a $4 \times 4$ grid network of binary variables, parameterized by a set of singleton factors over each variable and a set of pairwise factors over each edge in the grid. This network is created by the function ConstructToyNetwork.m, and in this assignment, you will change some of its parameters and observe the effect this has on different inference techniques.

Broadly speaking, the objective of inference in probabilistic graphical models is to say something about what we think the values of the random variables in the network are (perhaps given some evidence). One approach to doing so is to compute the maximum a posteriori (MAP) joint assignment to the variables in the network. (Though you may not have realized it at the time, this is what the inference algorithm from PA1 did.) The second approach (the one you will explore in this assignment) is to compute the posterior marginal distributions over the variables in the network. Concretely, given a set of factors $\Phi$ that represent a Gibbs distribution $P_\Phi(\mathbf{X})$ over variables $\mathbf{X} = \{X_1, X_2, ..., X_n\}$, we would like to compute $P(X_i)$ for $i = 1, ..., n$.

## 2 Belief Propagation

In the first part of this assignment, you will implement two types of message-passing inference: clique tree calibration and loopy belief propagation.

## 2.1 Clique Trees

Here, you will write code that uses a clique tree to perform inference. We first assign all factors to a clique in the clique tree, thus defining the *initial potential* of each clique. We then calibrate the clique tree by picking an arbitrary root clique and passing messages up and down the clique tree starting and ending at the root. Following calibration, we compute the marginal probability over the scope of each clique by multiplying the initial potential of the clique with all incoming messages.

In this assignment, we provide you with clique trees that enable inference over the relevant networks. (In particular, they satisfy the family preservation and running intersection property.) You must implement the message passing framework that accepts a clique tree and a set of factors as input, and then outputs the marginal probabilities of each random variable. The core operations that occur during message passing are factor-product and factor-marginalization, and these correspond to functions you will implement. In addition to these, you will also implement a function that will instantiate observed variables in factors, allowing you to obtain the posterior marginals given the evidence.

## 2.2 Loopy Belief Propagation

The second message passing algorithm is loopy belief propagation using a loopy cluster graph. In this assignment, we provide you with Bethe cluster graphs (described on pg. 405 in the book) for each network. You will implement loopy belief propagation (LBP), which accepts as input a cluster graph, a set of factors, and a list of evidence, and outputs the approximate posterior marginals for each variable. The message passing framework you implemented for clique trees should directly generalize to the case of cluster graphs. You should therefore have to write relatively little further code for LBP.

In LBP, we do not have a root and a specific up-down message passing order relative to that root. In particular, we can order messages by an arbitrary criterion. The starter code implements a naive message passing order that blindly iterates through all messages without considering other criteria. You will subsequently experiment with alternative message passing orders and analyze their impact on both the convergence of LBP and the values of the marginals at convergence.

# 3 MCMC

A second class of inference methods discussed in class are those based on Markov chain Monte Carlo sampling. As a reminder, a Markov chain defines a transition model $\mathcal{T}(\boldsymbol{x} \rightarrow \boldsymbol{x}')$ between different states $\boldsymbol{x}$ and $\boldsymbol{x}'$. The chain is initialized to some initial assignment $\boldsymbol{x}_0$. At each iteration $t$, a new state, $\boldsymbol{x}_{t+1}$ is sampled from $\mathcal{T}(\boldsymbol{x}_t \rightarrow \boldsymbol{x})$. The chain is run for some number of iterations, over which a subset of the samples is collected. The collected samples are then used to estimate statistics such as the marginals of individual variables. In this assignment, you will implement Gibbs sampling and Metroplis-Hastings, both of which sample from the posterior of a probabilistic graphical model.

A critical issue in the usefulness of a Markov chain is the rate at which it mixes to the stationary distribution. For example, if the stationary distribution has two modes that are far apart in the state space and the MCMC transition probability only allows local moves in the state space, it is likely that the state of the Markov chain will get stuck near one of the modes. This affects both the number of samples required before the chain "forgets" its initial state, and the quality of the estimates — unless many samples are collected, most samples are likely to come from one mode or another. If samples are aggregated before a chain has mixed, then the distribution from which they are drawn will be biased toward the initial state and thus will not be a good approximation to the stationary distribution. In the starter code, we provide you with a visualization function, VisualizeMCMCMarginals, that allows you to analyze a Markov chain to estimate properties such as mixing time and whether or not it is getting stuck in a local optimum. See the description of the code infrastructure below for more details on this function.

## 3.1 Gibbs sampling

Recall that the Gibbs chain is a Markov chain where the transition probability $\mathcal{T}(\boldsymbol{x} \to \boldsymbol{x}')$ is defined as follows. We iterate over the variables in some fixed order, say $X_1, \ldots, X_n$. For the variable $X_i$, we sample a new value from $P(X_i \mid \text{MarkovBlanket}(X_i))$, and update its new value. Note that the terms on the right-hand-side of the conditioning bar use the *newly sampled* assignment to the variables $X_1, \ldots, X_{i-1}$. Once we sample a new value for each of $X_1, \ldots, X_n$, the result is our new sample $\boldsymbol{x}'$.

In this assignment, you will implement a function that computes and samples from $P(X_i \mid \text{MarkovBlanket}(X_i))$. You will then use this function as a transition probability for MCMC sampling. You will also look at examples where Gibbs sampling becomes stuck in local optima and fails to mix properly.

## 3.2 Metroplis-Hastings

Metropolis-Hastings is a general framework (within the even more general framework of MCMC) that defines the Markov chain transition in terms of a proposal distribution $\mathcal{T}^Q(\boldsymbol{x} \to \boldsymbol{x}')$ and an acceptance probability $\mathcal{A}(\boldsymbol{x} \to \boldsymbol{x}')$. The proposal distribution and acceptance probability must satisfy the detailed balance equation in order to generate the correct stationary distribution.

In this assignment, you will implement a general Metropolis-Hastings framework that is capable of utilizing different proposal distributions, specifically the uniform distribution and the Swendsen-Wang distribution (described below). We will provide you with the implementations of these proposal distributions and you will need to compute the correct acceptance probability so that the detailed balance equation is satisfied. Furthermore, you will study the relative merits of each proposal type.

Swendsen-Wang was designed to propose more global moves in the context of MCMC for pairwise Markov networks of the type used for image segmentation or Ising models, where adjacent variables like to take the same value. At its core, it is a graph node clustering algorithm. Given a pairwise Markov network and a current joint assignment $\boldsymbol{x}$ to all variables, it generates clusters as follows: first it eliminates all edges in the Markov network between variables that have different values in $\boldsymbol{x}$. Then, for each remaining edge $\{i, j\}$, it "activates" the edge with some probability $q_{i,j}$ (which can depend on the variables $i$ and $j$ but not on their values in $\boldsymbol{x}$). It then computes the connected components of the graph over the activated edges. Finally, it selects one connected component, $\mathbf{Y}$, uniformly at random from all connected components. Note that all nodes in $\mathbf{Y}$ will have the same label $l$. We then (randomly) choose a new value $l'$ that will be taken by all nodes in this connected component. These variables are then updated in the joint assignment to produce the new assignment $\boldsymbol{x}'$. In other words, the new assignment $\boldsymbol{x}'$ is the same as $\boldsymbol{x}$, except that the variables in $\boldsymbol{Y}$ are all labeled $l'$. Note that this proposed move flips a large number of variables at the same time, and thus it takes much larger steps in the space than a local Gibbs or Metropolis-Hastings sampler for this MRF. (To test your understanding of the Swendsen-Wang proposal distribution, ask yourself this question: if you set all the $q_{i,j}$'s to zero, what does Swendsen-Wang reduce to?)

Let $q(\boldsymbol{Y} \mid \boldsymbol{x})$ be the probability with which a set $\boldsymbol{Y}$ is selected to be updated using this procedure. It is possible to show that

$$\frac{q(\boldsymbol{Y} \mid \boldsymbol{x}')}{q(\boldsymbol{Y} \mid \boldsymbol{x})} = \frac{\prod_{(i,j)\in\mathcal{E}(\boldsymbol{Y},\boldsymbol{X}'_{l'}-\boldsymbol{Y})}(1-q_{i,j})}{\prod_{(i,j)\in\mathcal{E}(\boldsymbol{Y},\boldsymbol{X}_{l}-\boldsymbol{Y})}(1-q_{i,j})} \tag{1}$$

where: $\boldsymbol{X}_l$ is the set of vertices with label $l$ in $\boldsymbol{x}$, $\boldsymbol{X}'_{l'}$ the set of vertices with label $l'$ in $\boldsymbol{x}'$; and where $\mathcal{E}(\boldsymbol{Y}, \boldsymbol{Z})$ (between two disjoint sets $\boldsymbol{Y}, \boldsymbol{Z}$) is the set of edges connecting nodes in $\boldsymbol{Y}$ to nodes in $\boldsymbol{Z}$. (NOTE: The log of the quotient in equation 1 is called "log_QY_ratio" in the code.) Then we have that

$$\frac{\mathcal{T}^Q(\boldsymbol{x}' \to \boldsymbol{x})}{\mathcal{T}^Q(\boldsymbol{x} \to \boldsymbol{x}')} = \frac{q(\boldsymbol{Y} \mid \boldsymbol{x}')}{q(\boldsymbol{Y} \mid \boldsymbol{x})}\frac{R(\mathbf{Y} = l|\boldsymbol{x}'_{-\mathbf{Y}})}{R(\mathbf{Y} = l'|\boldsymbol{x}_{-\mathbf{Y}})} \tag{2}$$

where: $R(\mathbf{Y} = l|\boldsymbol{x}_{-\mathbf{Y}})$ is a distribution specified by you for choosing the label $l$ for $\mathbf{Y}$ given $\boldsymbol{x}_{-\mathbf{Y}}$ (the assignment to all variables outside of $\mathbf{Y}$). Note that $\boldsymbol{x}_{-\mathbf{Y}} = \boldsymbol{x}'_{-\mathbf{Y}}$.

In this assignment, the code for generating a Swendsen-Wang proposal is given to you, but you will have to compute the acceptance probability and use that to define the sampling process for the Markov chain. You will implement 2 variants that experiment with different parameters for the proposal distribution. In particular, you will change the value of the $q_{i,j}$'s and $R(\mathbf{Y} = l|\boldsymbol{x}_{-\mathbf{Y}})$. In variant 1, set the $q_{i,j}$'s to be uniformly 0.5, and set the distribution $R$ to be uniform. In variant 2, $q_{i,j}$ will depend on the strength of the pairwise factor $F_{i,j}$ between $i$ and $j$. In particular, set

$$q_{i,j} := \frac{\sum_u F_{i,j}(u,u)}{\sum_{u,v} F_{i,j}(u,v)}$$

In addition, set $R$ to be the block-sampling distribution (as defined in BlockLogDistribution.m) for sampling a new label.

## 4 Code [60 points]

The following section describes the code you should implement in detail. All code for this assignment should be copied from:

/afs/ir/class/cs228/ProgrammingAssignments/PA2/assignment/

You should be able to access this by logging into the Stanford Unix machines (e.g. corn.stanford.edu) using your SUNet ID. You can also use a copy of Matlab on any of the machines that allow CPU-intensive jobs (e.g. corn, myth). Note that we will test your code on the corn machines, so please verify that it works on these before submitting.

Once you have copied over the code, cd into the Inference/ directory. The script **Inference/TestPA2.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 **Inference/TestPA2.m** and the functions 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. **Important:** Do not modify any of the rand('seed', X) statements in this file or elsewhere in the code. This seeds the random number generator so that your results on random algorithms will sync up with the autograder's.

1. **Message Passing Algorithms**

   (a) **[1 points] Factor Product**:

   - **FactorProduct.m** — This function should compute the product of two factors.

   This function will be tested using sample factors contained in INPUT.Factors.

   (b) **[1 points] Factor Marginalization**:

   - **FactorMarginalization.m** — This function should compute the factor after summing out a given variable in a given factor.

   This function will also be tested using factors from INPUT.Factors.

   (c) **[1 points] Observe Evidence**:

   - **ObserveEvidence.m** — This function should modify a set of factors given the observed values of some of the variables.

   (d) **[20 points] Calibrate Clique Tree**: This task contains two functions:

   - **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.

   These functions will be tested with 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.

   (e) **[5 points] Exact Inference**:

   - **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.

   This function will be tested with the same 5 subtests as in the previous test. 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.

   (f) **[10 points] Approximate Inference**: The next set of functions implement LBP inference:

   - **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. Note that you will implement an alternative message passing order in this function later in the assignment. However, for the purposes of the autograder, **please comment out your novel message passing order code and use the original message passing order before you submit**, or else your code will fail the autograder tests.
   - **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.

Once you have successfully implemented clique tree message passing, it should be relatively simple to implement LBP. The test has 5 subtests, worth 2 points each.

2. **MCMC Methods**

   (a) [**4 points**] **BlockLogDistribution.m** — This is the function that produces the sampling distribution used in Gibbs sampling and (possibly) versions of Metropolis-Hastings. It takes as input a set of variables $\boldsymbol{X}_I$ and an assignment $\boldsymbol{x}$ to all variables in the network, and returns the distribution associated with sampling $\boldsymbol{X}_I$ as a *block* (i.e. the variables are constrained to take on the same value) given the joint assignment to all other variables in the network. That is, for each value $l$, we compute the (unnormalized) probability $\tilde{P}(\boldsymbol{X}_I = l \mid \boldsymbol{x}_{-I})$ where $\boldsymbol{X}_I = l$ is shorthand for the statement "$X_i = l$ for all $X_i \in \boldsymbol{X}_I$" and $\boldsymbol{x}_{-I}$ is the assignment to all other variables in the network. Because this function will be called in the inner-most loop of both Gibbs and Metropolis-Hastings (Swendsen-Wang proposal) sampling, you should do your best to make it fast. Particularly, in the code you should use G.var2factors, which maps a variable to the factors that include the variable in their scope. You should also look at the matlab functions **repmat** and **intersect**. The **repmat** function returns the self-concatenation of the given matrix by the given number of rows and columns. The **intersect** function computes the intersection of two arrays and returns the indices into the original arrays of the elements in the intersection. Finally, use the function **GetValuesOfAssignments** to efficiently compute the factor values of multiple assignments.

   Your solution should only contain one for-loop (because for-loops are slow in Matlab). Note that the distribution will be returned in log-space to avoid underflow issues.

   (b) [**6 points**] **MCMCInference.m** — This function defines the general framework for conducting MCMC inference. It takes as input a probabilistic graphical model, a set of factors, a list of evidence, the name of the MCMC transition to use, and other MCMC parameters such as the target burn-in time and number of samples to collect. You need to implement the logic that transitions the Markov chain to its next state and records the sample. You should also implement the calculations of the $q_{i,j}$'s for both variants of Swendsen-Wang in this function. (The reason that this is done here and not in MHSWTrans.m is to improve efficiency.) You also need to implement the function ExtractMarginalsFromSamples.m (called at the end of MCMCInference), which converts a list of samples to a set of marginals.

   (c) [**4 points**] **GibbsTrans.m** — This function defines the transition process in the Gibbs chain. It should call BlockLogDistribution to sample a value for each variable in the network.

   (d) [**4 points**] **MHUniformTrans.m** — This function defines the transition process associated with the uniform proposal distribution in Metropolis-Hastings. You should fill in the code to compute the correct acceptance probability.

   (e) [**4 points**] **MHSWTrans.m** — This function defines the transition process associated with the Swendsen-Wang proposal distribution in Metropolis-Hastings. You should fill in the code to compute the proposal distribution values and then use these to compute the acceptance probability. Note that there can be different variants of Swendsen-Wang because the value of the $q_{i,j}$'s is arbitrary and the new label $l$ can also be sampled from an arbitrary distribution. You will implement the 2 variants described earlier in the handout. Since the $q_{i,j}$'s are computed in MCMCInference.m, the only thing different between the two variants in this function is the computation of the distribution $R$, from which a new label for $\mathbf{Y}$ is selected.

## 5 Tasks & Questions [40 points]

1. [**3 points**] Now assume you are going to apply your message passing framework to the problem from PA1. That is, your goal is to label each pixel in an image with one of 8 semantic labels. You do this by first segmenting the image into non-overlapping superpixels. You then define a pairwise Markov Network, where each variable corresponds to the label of a superpixel. There are two types of factors in the network – singleton factors over each variable that capture some initial confidence in the label of the superpixel and pairwise factors which capture the similarity bias of adjacent superpixels. Given this problem statement, you want to conduct exact and approximate inference using message passing over a clique tree and cluster graph. As in PA1, you wish to conduct inference for different sized superpixel segmentations. Can you use the same clique tree for different images at the same superpixel level? What about the same image with different superpixel levels? For either scenario, can you use the same cluster graph? Why or why not?

2. [**15 points**] The code skeleton we provide for you in ClusterGraphCalibrate.m defines a specific but rather arbitrary order of message-passing for approximate inference. You will now explore an alternative message passing order. Conduct the following experiments on the insurance network.

   (a) [**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.

   (b) [**7 points**] These observations suggest that convergence may not be even across the network. Build on this intuition to experiment with an alternative order for passing messages. Your order does not have to be regularly repeating nor does it have to visit each message the same number of times. Analyze how this new schedule affects convergence. (Note that you may want to change the way the convergence criterion is computed, depending on the properties of the schedule.) 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.) The helper function MessageDelta may be of interest to you. When you are finished with this question, please comment out the parts of your code that implement the new ordering (but leave comments specifying what it does) in order to pass the autograder tests.

   (c) [**4 points**] Can changing the message passing order affect convergence in cluster graph calibration? Can it affect the value of the final marginals? Why or why not?

3. [**5 points**] Now, consider the toy image network constructed in ConstructToyNetwork.m. Change the values of the on- and off-diagonal weights of the pairwise factor in this network to different values. You should consider the case where the on-diagonal weight is much larger, the on-diagonal weight is much smaller, and where the weights are roughly equal.

For each such model, run LBP and exact inference and compare the results. How does LBP respond to the change in this ratio? Describe and explain what you observe.

4. [**5 points**] Conduct the following experiment on the toy image network. Set the on-diagonal weight of the pairwise factor (see ConstructToyNetwork.m) to be 1.0 and the off-diagonal to be 0.1. Now run Gibbs sampling a few times, first initializing the state to be all ones and then initializing the state to be all twos. What effect does the initial joint assignment have on the accuracy of Gibbs sampling? Explain why this effect occurs. Is this same effect apparent in Metropolis-Hastings with either of the 2 Swendsen-Wang proposal distributions you implemented?

5. [**8 points**] Now, consider again the experiment where we change the on- and off-diagonal weights of the pairwise factor in the toy network; again, you should consider the case where the on-diagonal weight is much larger, the on-diagonal weight is much smaller, and where the weights are roughly equal. For each such model, run exact inference, Gibbs sampling, and the Metropolis-Hastings variants (using random initializations for the sampling methods).

   (a) [**4 points**] For each of the MCMC methods, use VisualizeMCMCMarginals.m to visualize the distribution of the Markov chain for **multiple** runs of MCMC (see Test-Toy.m for an example of how to do this). Describe how the mixing behavior for each chain changes in response to the changes in the pairwise factor, and explain why this happens.

   (b) [**4 points**] Now, compare the final marginals computed by each of the MCMC techniques against the exact marginals as you change the pairwise factor. Explain why you get different behavior for the different chains.

6. [**4 points**] Explain how the Swendsen-Wang proposal distribution avoids the problems of the simpler superpixel-based proposal distribution of Question 1f in the theoretical problem set.

7. [**5 points**] Extra Credit – Consider other variants of the Swendsen-Wang chain that we have given you. You can explore other options for setting $q_{i,j}$ or other (non-uniform) distributions for picking the label $l$ for the selected set $\boldsymbol{Y}$. Note, in particular, that $l$ can also depend on the image and on the current state $\boldsymbol{x}$. However, be careful to correctly compute the new acceptance probability so that your chain is reversible and has the correct stationary distribution. Include a brief justification of why your Markov chain is reversible (it is sufficient to show that your proposal distribution and acceptance probability satisfy the detailed balance equation), why you think your technique should perform well, and a brief analysis of how your technique actually performs on one of the networks from the assignment.

**You must include the answers to these questions in a plain-text or pdf file called README when you submit your code.**

## Infrastructure

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. **GetValueOfAssignments.m** — Given a list of full assignments to the variables of a factor, this function returns a list of values, one for each assignment.
5. **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 $\phi(1, 1, 2) \leftarrow 0.1$.
6. **AssignmentToIndex.m** — Utility function to convert from a variable assignment to a corresponding index into a factor table.
7. **IndexToAssignment.m** — Utility function to convert from an index into a factor table to a corresponding variable assignment.
8. **ConstructToyNetwork.m** — Function that constructs a toy pairwise Markov Network that you will use in some of the questions.
9. **LogProbOfJointAssignment.m** — Returns the log probability of a joint assignment to all variables in the network. Log-space is used to avoid underflow. This function will likely be useful for computing acceptance probabilities in Metropolis-Hastings.
10. **VisualizeMCMCMarginals.m** — This displays two things. First, it displays a plot of the log-likelihood of each sample over time. Recall that in quickly mixing chains, this value should increase until it roughly converges to some constant log-likelihood, where it should remain (with some occasional jumps down).

    This function also visualizes the estimate of the marginal distributions of specified variables in the network as estimated by a string of samples obtained from MCMC over time. In particular, it takes a fixed-window subset of the samples around a given iteration $t$ and uses these to compute a sliding-window average of the estimated marginal(s). It then plots the sliding-window average of each value in the marginal as its estimate progresses over time. The function also can accept samples from more than one MCMC run, in which case the marginal values that correspond to one another are plotted in the same color, allowing you to determine whether the different MCMC runs are converging to the same result. This is particularly helpful if you are trying to identify whether the chain is susceptible to local optima (in which case, different runs will converge to different marginals) or whether the chain has mixed by a given iteration.
11. **TestToy.m** — This function constructs a toy image network where each variable is a binary pixel that can take on a value of 1 or 2. This network is a pairwise Markov net structured as a $4 \times 4$ grid. The parameterization for this network can be found in TestToy.m and

you will tune the parameters of the pairwise factors to study the corresponding behavior of different inference techniques. You can visualize the marginal strengths of this toy "image" by calling the function VisualizeToyImageMarginals.m, which will display the marginals as a gray-scale image, where the intensity of each pixel represents the probability that that pixel has the label 1.

12. **VisualizeToyImageMarginals.m** — Visualizes the marginals of the variables in the toy network on a 4x4 grid.

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 structures[1]:

- `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 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, **Inference/TestPA2.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 **ClusterGraphCalibration.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:

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

- `.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));
```

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, $\phi$. 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]

[2]It 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.

[3]Hint: 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 $B$.

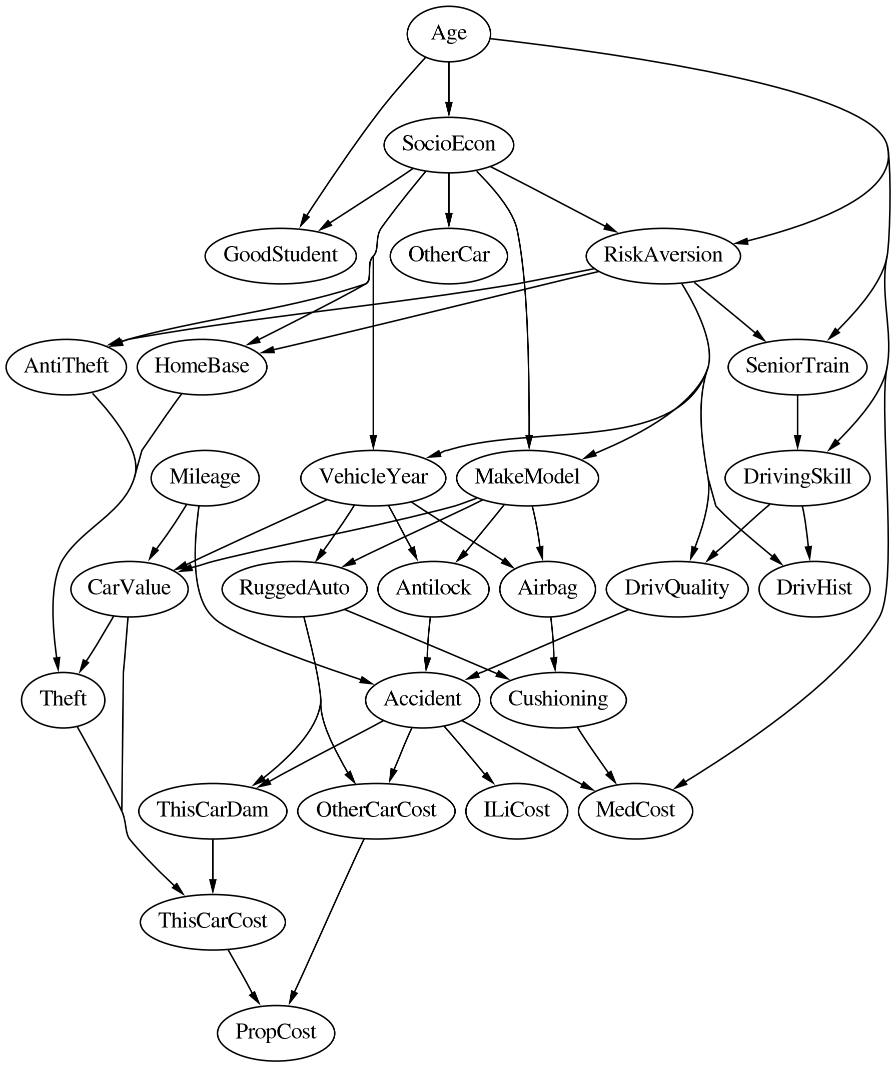

Figure 1: INSURANCE Network for Estimating Expected Insurance Claim Costs.