Probabilistic Graphical Models Programming Assignment #5: Approximate Inference

This assignment is due at 11:59pm PDT (UDT -7) on 7 November 2012. Important: There is a bug in the current implementation. Due to issues with the random number generator, you cannot use Matlab 2012 versions, we recommend you use a 2011 version, or Octave instead.

1 Introduction

Last week, we focused on implementing exact inference methods using the clique tree algorithm, both for finding exact marginals and for finding the MAP assignment in a PGM. Unfortunately, sometimes performing exact inference is intractable and cannot be done as performing exact inference in general networks is NP-hard.

Consider for example the 3×3 pairwise Markov network and its corresponding Bethe cluster graph[1]:

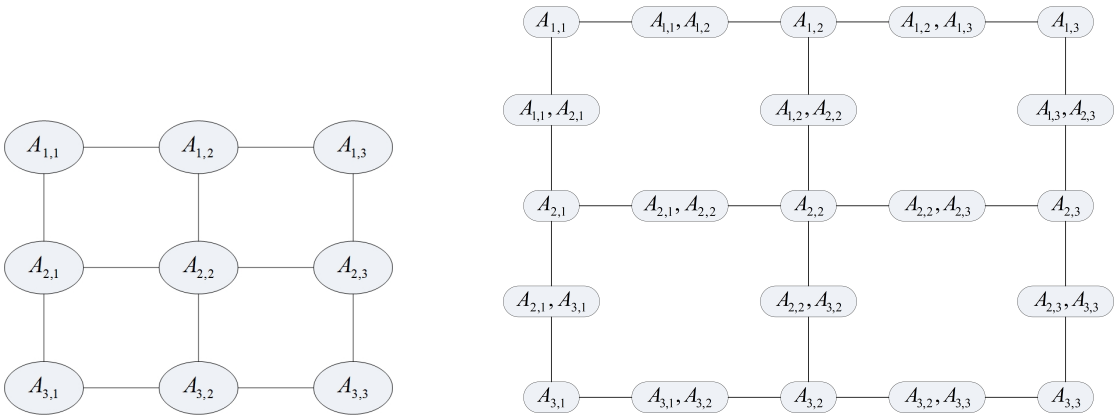


Figure 1: 3×3 pairwise Markov network and its corresponding Bethe cluster graph.

To do exact inference, we would have to construct a clique tree for this network. We cannot simply use the Bethe cluster graph for this network, since from Fig. 1 it is clear that this is not a clique tree. To construct a clique tree, we can for example reparametrize the original network by combining variables along diagonal cuts through the network, producing an equivalent network with the same joint distribution, as follows:

[1] To see that this figure corresponds to the Bethe cluster graph, note that there is a cluster for each pairwise factor $A_{i,j}, A_{p,q}$ and each of these factors is connected to the clusters corresponding to its constituent individual random variables $A_{i,j}$.

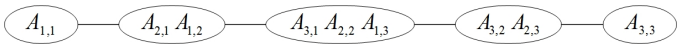


Figure 2: 3×3 pairwise Markov network reparametrized along diagonal cuts.

This new reparametrized network consists of five groups of variables. The clique tree which corresponds to this new network is:

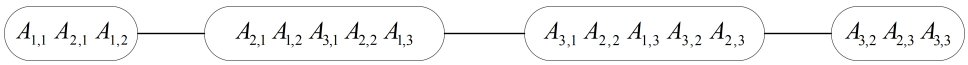


Figure 3: Clique tree corresponding to a 3×3 pairwise Markov network.

Alternatively, we could have combined variables along horizontal or vertical cuts, but diagonal cuts result in a clique tree with the smallest sepsets. The maximal clique in this tree consists of $2n - 2$ pairwise factors. The maximal sepset contains n elements. If all variables are binary, the CPD of the maximal clique will contain 2^{2n-2} entries, which is exponential in the grid size (for the sake of clarity, we have ignored singleton factors here). For larger grids the construction of the clique tree will therefore be intractable and this problem is thus NP-hard (even without having done inference yet).

Luckily, there are a number of approximate inference methods we can use, and in this programming assignment we will investigate two of them: loopy belief propagation (LBP) and Markov chain Monte Carlo (MCMC). The goal of this assignment is to get a feel for how these two important approximate methods work and which limitations they have. You will see how feedback loops affect the convergence and the accuracy of these methods and you will investigate how they can be tuned and/or improved to deal with these effects.

Throughout this assignment it is important to realize that choosing the right approximate method and selecting its parameters is sometimes more of an art than an exact science. While there are some rules of thumb for which methods will work better in a certain situation, one usually has to tune these methods manually and carefully experiment to verify that they actually work in the situation at hand. This is the kind of experimenting you will do throughout this assignment!

A word of warning: this programming assignment is quite intense, so we recommend starting early and working on the optional parts only after finishing all the required tasks.

2 Testing your code

We strongly recommend using the test script provided by us in the course forum. It makes use of the file **exampleIOPA5.mat**, which contains example input and output corresponding to the 13 preliminary tests for this assignment. For argument j of the function call in part i, you should use **exampleINPUT.t**$\#_i$**a**$\#_j$ (replacing the $\#_i$ with i). If there are multiple function calls in one test (for example, we iterate over multiple inputs) then for iteration k you should reference **exampleINPUT.t**$\#_i$**a**$\#_j$.$\{\#_k\}$. For output, look at **exampleOUTPUT.t**$\#_i$ for the output to part i. If there are multiple outputs or iterations, the functionality is the same as for the input example, but with **a**'s replaced by **o**'s.

An example is the second part of this assignment, **CreateClusterGraph.m**, for which the test script looks like:

```
P = CreateClusterGraph(exampleINPUT.t2a1, exampleINPUT.t2a2);
result = isequal(P, exampleOUTPUT.t2);
```

Because some of the code makes use of a random number generator, you can only match the test output when you set the seed using `randi('seed',1);`. You have to do this once for each part, starting at part 7 (**GibbsTrans**). Note that you don't have to worry about this when you use our test script.

Important: the current implementation has two bugs. When testing part 8 (**MCMCInference PART 1**) you have to set `exampleINPUT.t8a42 = MHGibbs`. When testing part 12 (**MCMCInference PART 2**) you have to reset the random number generator before the second iteration by `randi('seed',26288942);`. Note that this is also taken care of in our test script.

2.1 Toy networks

We have also provided a couple of toy networks which you have to use for the assignment questions. You can create these networks using the functions **ConstructToyNetwork.m** and **ConstructRandNetwork.m**. Each of these functions creates a 4×4 pairwise Markov network, with each node a binary variable. The network consists of singleton factors for every node and pairwise factors for adjacent nodes.

Both functions accept two numbers as arguments: an on-diagonal weight and an off-diagonal weight. These weights define the CPDs for the pairwise factors (those CPDs are the same for all the node pairs in the grid network). The on-diagonal weight corresponds to the unweighted probability that adjacent nodes agree on their values (i.e. assignment $(1,1)$ or $(2,2)$). Correspondingly, the off-diagonal weight corresponds to the probability that adjacent nodes disagree on their values (assignment $(1,2)$ or $(2,1)$). A big relative difference in these two numbers indicates strong (anti-)correlations between adjacent nodes. An example of a realization of this toy network is shown in Fig. 4. In this assignment, you will experiment with these weights and observe the effect this has on different inference techniques.

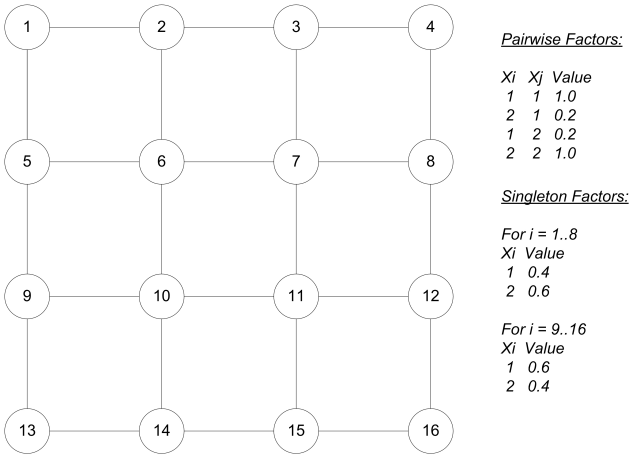


Figure 4: An example of the toy network.

Both **ConstructToyNetwork.m** and **ConstructRandNetwork.m** have as output two fields: G, a data structure that represents the variables in the probabilistic graphical network and F, a list of all factors in the network. G differs somewhat from the data structures you have encountered before and contains the following fields:
.names – a cell array where names$\{i\}$ = name of variable i in the graph;
.card – an array where card(i) is the cardinality of variable i;
.var2factors – a mapping of variables to which factors they are included in;
.edges – an adjacency matrix where edges$(i, j) = 1$ implies factors i and j share an edge.
Even though G is a redundant data structure, as all fields can easily be computed from the list of factors, you will find its fields convenient in the Markov chain Monte Carlo part of this assignment.

3 Loopy Belief Propagation

The first approximate inference method that you will implement is loopy belief propagation (LBP). LBP is not restricted to clique trees, but can run on any cluster graph, like the Bethe cluster graph shown in Fig. 1. In principle, many different cluster graphs can be constructed for a given network. Running LBP on a different graph will affect convergence and the accuracy of the resulting marginals and it is not a priori clear which graph will perform best. In this assignment you will work with the Bethe cluster graph, which is perhaps the simplest way to create a cluster graph from a network as we create a separate cluster for every factor and attach them so as to meet the requirements of a cluster graph. For a pairwise Markov network it has the additional property that there is a direct correspondence between the clusters in the cluster graph and variables or edges in the pairwise Markov network (see Fig. 1).

Just like in the clique tree algorithm, in LBP the cluster graph beliefs are calibrated by passing messages over the graph. However, because a cluster graph may contain feedback loops, the resulting marginals are in general not exact. Moreover, in LBP we can choose any particular message passing order, without being restricted to a root and the corresponding up-down message passing order. In this assignment you will first implement a naive message passing order that blindly iterates through all messages without considering other criteria. Optionally, you can subsequently experiment with alternative message passing orders and analyze their impact on the convergence of LBP and the values of the marginals at convergence.

- **NaiveGetNextClusters.m (5 points)** — 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. In this naive implementation we simply iterate over the cluster pairs. Details on this ordering are given within the code file.

Now we can begin the message passing process to calibrate the cluster, but we'll need some infrastructure to make this work. For example, we need to create a cluster graph in order to run LBP and we need criteria to tell us when we've converged.

- **CreateClusterGraph.m (5 points)** – Given a list of factors, this function will create a Bethe cluster graph with nodes representing single variable clusters and pairwise clusters. The ClusterGraph data structure is the same as the CliqueTree data structure from PA4, but cliqueList is renamed as clusterList. You may assume that the input factor list includes all the required singleton factors. These are numbered consecutively starting at 1 and precede all other factors.

- **CheckConvergence.m (2 points)** – Given your current set of messages and the set of messages that immediately preceded each of our current messages, this function will determine if the cluster graph has converged, returning 1 if so and 0 otherwise. In this case, we assume that the messages have converged if no messages have changed significantly, where "significantly" means by a value of over 10^{-6} in any entry.
- **ClusterGraphCalibrate.m (10 points)** — This function should initialize the initial message values to 1 and then perform loopy belief propagation over a given cluster graph. It should return an array of final beliefs (factors) for each cluster.

We are now ready to bring together all the steps described above to compute approximate marginals for the variables in our network.

- **ComputeApproxMarginalsBP.m (5 points)** – This function should take a set of initial factors and vector of evidence and compute the approximate marginal probability distribution for each variable in the network. You should be able to reuse much of the code you wrote for **ComputeExactMarginalsBP.m** for PA4 in this function.

3.1 LBP Investigation Questions

Answer these questions in the assignment quizzes section on the course website.

1. **(5 points)** – Recall our function **CheckConvergence.m** which uses the difference between a message before and after it is updated (called the "residual") as a criterion for convergence. While running LBP with our naive message ordering on the network created by **ConstructRandNetwork.m** with on-diag weight 0.3 and off-diag weight 0.7, print out and plot the residuals of the message $19 \to 3, 15 \to 40$, and $17 \to 2$, with the iteration number on the x-axis (you may want to change the range of the y-axis). Do these messages converge at the same rate? Which converges fastest? (Note, it will be easiest do this assessment within **ClusterGraphCalibrate.m** and use the helper function **MessageDelta** within that file). You may construct the network and check for convergence using code like (after having made changes to **ClusterGraphCalibrate.m**):
```
[G, F] = ConstructRandNetwork(.3, .7);
M = ComputeApproxMarginalsBP(F, []);
```
2. **(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)** – Even though performing exact inference in $n \times n$ pairwise networks is in general NP-hard, the 4×4 model we study here is small enough that exact inference is still possible. This makes it possible to compare the performance of LBP by comparing the resulting marginals with the exact ones. Now, consider the toy image network constructed in **ConstructToyNetwork.m**. Change the values of the on- and off-diagonal weights of the pairwise factors in this network to different values (which can be done by changing the values passed to this function). First try making the the weights on the diagonal much larger than the off-diagonal weights (1 and 0.2 respectively), then try the opposite where the off-diagonal weights are much larger (0.2 and 1), and then finally try the case where the weights are equal (0.5 and 0.5). For each such model, run LBP and exact inference (using your code from PA4). How do the results on these varied types of graphs compare? Why? (Note that if LBP does not converge within 100,000 iterations, it is okay to truncate the run and report on the pseudo-marginals given at that point.) You may construct the

network and calculate the marginals using code like:

```
[G, F] = ConstructToyNetwork(1, .2);
M = ComputeApproxMarginalsBP(F, []);
ExM = ComputeExactMarginalsBP(F, [], 0);
```

3.2 Improving Message Passing (optional)

In this optional part we invite you to improve on the naive LBP algorithm which you have been implementing so far. In the lectures two ideas have been discussed: smoothing messages and informed message scheduling. The idea of smoothing messages is to dampen strong oscillations in cluster beliefs (which may occur due to tight loops and/or strong potentials pulling in different directions) by taking for each message the weighted average of the new message we just computed, and its previous value at the last time we sent this message.

For informed message scheduling two ideas have been discussed. The first is tree reparameterization (TRP), in which several trees are defined over the network which can be calibrated using the clique tree algorithm. The second is residual belief propagation (RBP), in which messages between clusters are prioritized based on the magnitude of the changes in the marginals before and after message passing. Try to experiment with each of these to see whether you can improve the performance of your implementation, both in terms of convergence and in correctness of the resulting marginals.

There exist even better message passing schedules that improve the convergence of the algorithm, but those are beyond the scope of this assignment. For more information, we refer the interested readers to relevant research papers such as Elidan, McGraw, and Koller (2006)[2].

4 Markov Chain Monte Carlo (MCMC)

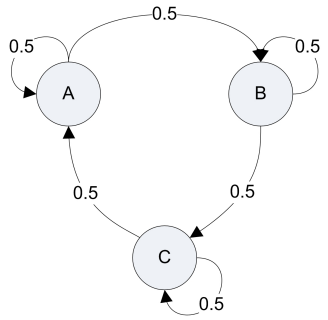
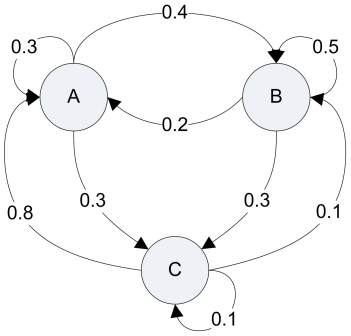


Figure 5: A simple Markov chain in which each state is equally likely.

In Fig. 5, we see a very simple Markov chain. We can tell by examination that if we run it for a while, we're equally likely to end up in any state. So the stationary probabilities are $\pi_A = 0.33, \pi_B = 0.33$ and $\pi_C = 0.33$.

[2] http://www.robotics.stanford.edu/~koller/Papers/Elidan+al:UAI06.pdf



$$\pi_A = 0.2(\pi_B) + 0.8(\pi_C) + 0.3(\pi_A)$$
$$\pi_B = 0.5(\pi_B) + 0.1(\pi_C) + 0.4(\pi_A)$$
$$\pi_C = 0.3(\pi_A) + 0.3(\pi_B) + 0.1(\pi_C)$$
$$\pi_A + \pi_B + \pi_C = 1$$
$$\pi_A \approx 0.39 \quad \pi_B \approx 0.36 \quad \pi_C \approx 0.25$$

Figure 6: A more complex Markov chain in which each state has a different probability.

The last inference method we'll work with is Markov chain Monte Carlo (MCMC)[3]. As with the other methods, we begin with a PGM (a Markov or Bayes network) that we want to do inference on, which is to say we want the marginal probabilities for some or all of the variables of our model. But instead of computing exact probabilities, we will approximate them via simulation, sampling from the joint probability distribution of the model and computing statistics on these samples. This seems contradictory: we want to find out the distribution, or at least the marginals of the distribution, by sampling from the distribution. But if we don't know the distribution, how can we sample from it?

We approach the problem in an indirect way, using a trick, a Markov chain, which gives a way to sample from a distribution, even though we don't know exactly what the distribution is. A Markov chain consists of some states, which in our case will correspond to assignments to the variables of the model. Each state has transition probabilities specifying the probabilities of stepping to each other state from that state. In a Markov chain, we start at some initial state, and in each step we move to another state, obeying the transition probabilities. As discussed in the lectures, some Markov chains have the property that if we run one such chain infinitely long, it will always converge to the same stationary probability distribution Π for the states: that is, if we run it forever, the probability of being in some state s will be the same p no matter what the initial state was.

In Fig. 6, we see a slightly more complicated chain. Nevertheless, it's simple enough to compute the stationary probabilities as shown. With a bigger graph, the equations would get intractably big to solve, but we can calculate those probabilities differently, empirically, by sampling. In sampling, we pick some initial state, say A. Then we randomly choose the next state according to the transition probabilities: If we're at A, we stay at A with probability 0.2 and move to B with probability 0.8. Say we chose to move to B. Then B is our next sample. For the next transition, we can stay at B with probability 0.5 or go to C with probability 0.5. After a long time, we might have a chain like: $AABCABCCAAABBCABCCC$. Then to determine the stationary probabilities we can just count up the number of times we visited each state and divide by the total number of states we visited.

[3]Note that the Markov networks we've been working with as probabilistic graphical models, and Markov chains, both have the name Markov in them. They're both named after the same guy, but otherwise Markov networks and Markov chains are different things.

When we do Markov chain Monte Carlo, we set up a Markov chain where each state corresponds to a complete assignment to the variables in our PGM. Our trick is to choose the right transition probabilities of the Markov chain so that the probabilities in the stationary distribution of the Markov chain match the probabilities of the joint distribution of the PGM.

Then we run the Markov chain. We pick somewhere in the state space to start. We step along from state to state. Stepping along the Markov chain, which we know how to do, is equivalent to sampling from the joint probability distribution of the PGM, which we didn't know how to do. Each state corresponds to a complete assignment of variables, so we keep statistics of the variable assignments for each state we visit and use them to compute the marginal probabilities of our variables.

4.1 Gibbs sampling

Different types of MCMC use different techniques for the trick of getting the right transition probabilities. For this assignment, you'll investigate several approaches, starting with a simple one, Gibbs sampling. In Gibbs sampling, we start at some randomly-chosen initial state in the Markov network. To move to the next state, we iterate through the variables of the PGM. We replace each value in turn with a newly-selected value (which could be the same as the old value), using probabilities conditional on the values of all the other variables – the new values for the variables we already changed this iteration and the old values for the variables we haven't reached yet. When we have gone through all the variables, their new values become our new variable assignment for the new state, and we've made one step in the Markov chain. As the lectures explain, when the chain is regular – which is for example the case when all transition probabilities are strictly greater than zero – it has the same stationary distribution as the distribution of our PGM, so it's a legal MCMC transition (see Fig. 7).

To test your various MCMC versions, you'll use the same **ConstructToyNetwork.m** and **ConstructRandNetwork.m** that you used for the assignment questions for loopy belief propagation, above.

Now you will implement the MCMC framework and the Gibbs sampling procedure. First you will implement two helper functions. The first is **BlockLogDistribution.m**, in which the sampling distribution of a set of similar variables in a network is calculated given all other variables in the network. This function is useful both for Gibbs sampling, where only one variable is sampled at a time, and for Metropolis-Hastings sampling, in which a block of similar variables is sampled, and which you will implement later in this assignment. The second helper function you will implement is **GibbsTrans.m**, in which the actual Gibbs sampling takes place. Finally, you will implement **MCMCInference.m**, which sets up and executes the Markov chain.

- **BlockLogDistribution.m: (5 points)** – This is the function that produces the sampling distribution used in Gibbs sampling and later in this assignment in some versions of Metropolis-Hastings. You'll be given as input a set of variables to change, plus the current assignment of all variables. The task is to change the specified variables, at the same time, to a new value (which is potentially the same as the old value). All of the variables will be changed to the same new value; **BlockLogDistribution.m** will return the probability distribution for choosing that new value. Note that for MCMC using Gibbs sampling, we sample only one variable at a time, not a group of variables. So when you're doing Gibbs sampling, the input to **BlockLogDistribution.m** will be a set with only one variable in it. Later in this assignment, you'll be calling **BlockLogDistribution.m** with bigger sets of variables when you use other versions of MCMC.

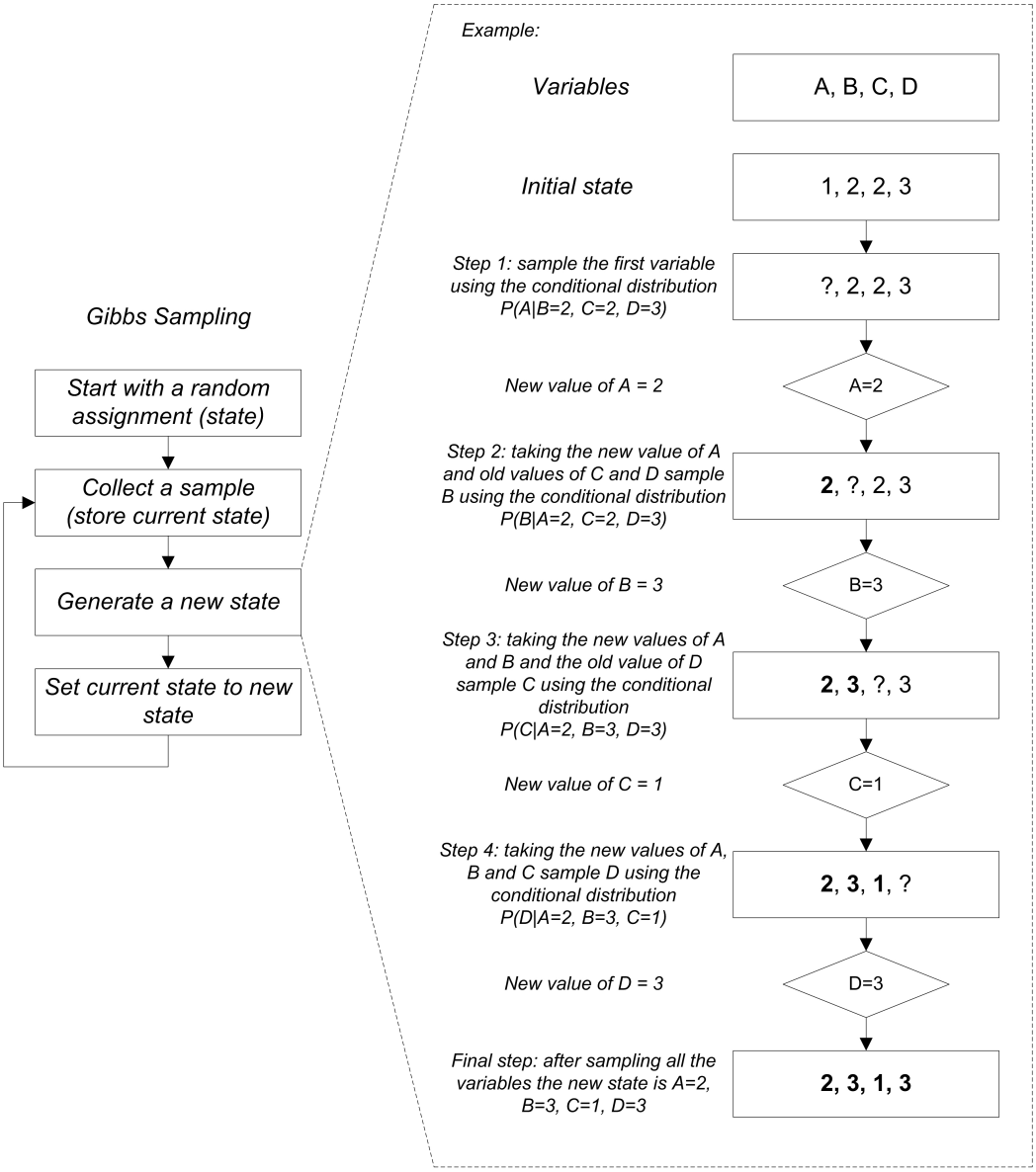


Figure 7: Gibbs sampling process.

So, you'll compute the (unnormalized) log probability distribution for re-sampling all the input variables together, given the joint assignment of all the other variables in the network. You'll return a vector of unnormalized log likelihoods, $[p_1, p_2, \ldots, p_n]$, where n is the cardinality of the variables you're sampling, and p_k is the log likelihood that the variables take value k. You may assume that all the input variables have the same cardinality.

Your solution should only contain one for-loop (because for-loops are slow in Matlab). The helper function **LogProbOfJointAssignment.m** might be useful for taking care of this for-loop, but you may also implement this yourself[4]. The distribution should be returned in log-space to avoid underflow issues. Moreover, the last statement in the provided code, `LogBS = LogBS - min(LogBS)`, adjusts the probabilities so they behave better when you return to probability space; leave it in.

- **GibbsTrans.m (5 points)** – This function defines the transition process in the Gibbs chain as described above (ie. we iteratively resample each variable). It takes as input the current assignment of all variables and should return the next assignment of all variables according to the Gibbs sampling procedure. For each variable in the network, it should call **BlockLogDistribution.m** to sample a new value.

- **MCMCInference.m PART 1: (3 points)** – This is 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 mixing time and number of samples to collect. It returns the marginals for all variables in the network and the set of collected samples over which these marginals have been calculated. Note that one sample simply consists of an assignment to all variables in the network at a particular step in the Markov chain. The calculation of the marginals has already been implemented for you. Your job is to fill the matrix all_samples by implementing the logic that transitions the Markov chain to its next state and to record each resulting sample. As a first step, you only need to get it to work for Gibbs sampling. Note that in stead of directly using **GibbsTrans.m** you should use the function handle `Trans` (more details about this function handle are given in the code). In this way you will keep **MCMCInference.m** a general function, which later in this assignment you will also implement for Metropolis-Hastings sampling.

4.2 Mixing

When we use MCMC, we are wandering through the Markov network state space, where each state corresponds to a full assignment to the PGM variables. We know that, by construction, eventually the probability that we wander next to a state S with variable assignment A is the same as the probability, in the PGM, of assignment A: that's the point, that's why we are doing the Markov chain Monte Carlo, to learn the (marginal) probabilities of variables.

When we start out, for our first few or many steps, we can't end up just anywhere in the network; we remain close to the start because we haven't wandered for long enough yet. But after some N number of steps, knowing where we started doesn't help us know where we'll be. That N is the mixing time, the time when we're lost in the network and could be anywhere. Then we've mixed, we're at the stationary probability distribution of the Markov chain (which, recall, corresponds to the probabilities of variable assignments in the PGM we're studying), and

[4]Strictly speaking, you will probably use a nested for-loop, because the helper function **GetValueOfAssignment.m** also contains a for-loop.

we can start taking samples. That's the mixing time. If we sample before the mixing time, our samples will be biased toward the initial state and other states near it.

Notice that mixing time is a property of the particular Markov chain we're using, not a property of a particular run of the chain. While the definition of mixing time is clear, determining when a chain is mixed is far from obvious. In general there is no formula to assess mixing; it's more of an art than a science. We need to look at our potential sample runs to see if they seem like they are representative of the stationary distribution, which they would be if the chain was mixed.

When we analyze the output of a chain we can sometimes determine that it is not mixed, by comparing two different potential sample runs and discovering that they look nothing like representative samples from the same distribution. For example, we can compare the marginal probability for one variable, computed from each of the two runs. Those two probabilities should be approximately equal, since they should both be approximately the true marginal probability. If they are not equal, we probably don't have mixing. We provide you **VisualizeMCMCMarginals.m** to help with determining the mixing time for a chain.

VisualizeMCMCMarginals.m outputs two different graphs. First, it shows the log-likelihood of each sample for each iteration. What you would like to see in this graph is an apparently quickly mixing chain, where after an initial period of settling in, the chain is mixed, and the texture, the pattern of highs and lows, is more or less the same from one part of the sample to another, and from one run of the chain to another. What you might see instead is a poorly mixing chain, which seems to be getting stuck in different modes, perhaps a low probability for a while, then a high probability for a while, then a different slightly higher or lower probability for a while.

The second graph provided by **VisualizeMCMCMarginals.m** is a sliding-window marginals graph. This computes the marginals for an iteration window up to iteration i, and graphs them for all i. It plots marginals for different variables in different colors, and marginals for the same variable for different runs in the same color. You hope to see that marginals for the same variable for different windows, and for the same variable for different runs, are the same, because you want all of those marginals to be the true marginal from the stationary distribution. If two different runs show different marginals for the same variable, you do not have mixing, alas. Optionally, **VisualizeMCMCMarginals.m** takes the exact marginals, which are plotted in dashed lines. This can be helpful to determine whether the Markov chain could have converged to the stationary distribution.

So we can look at a chain and decide it probably isn't mixed. But how do we decide it is mixed? We can't. All we can do is look at our potential sample runs, decide that they don't seem dissimilar, and hope we have achieved mixing.

4.2.1 Running Gibbs Sampling and Questions

With the inference engine, let's try to understand the behavior of Gibbs sampling (answer online):

1. **(5 points)** – Let's run an experiment using our Gibbs sampling method. As before, use the toy image network and set the on-diagonal weight of the pairwise factor (in **ConstructToyNetwork.m**) to be 1.0 and the off-diagonal weight to be 0.1. Now run Gibbs sampling a few times, first initializing the state to be all 1's and then initializing the state to be all 2's. What effect does the initial assignment have on the accuracy of Gibbs sampling? Why does this effect occur?
   You may construct the network and visualize the value of the marginals during the chain using code like:

```
[G, F] = ConstructToyNetwork(1, .1);
randi('seed',1);
for i = 1:2
   [M all_samples] = MCMCInference(G, F, [], 'Gibbs', 1, 12000, 1, repmat(i, 1, 16));
   samples_list{i} = all_samples;
end
ExM = ComputeExactMarginalsBP(F, [], 0);
VisualizeMCMCMarginals(samples_list, 1:length(G.names), G.card, F, 1500, ExM, 'Gibbs');
```

Gibbs sampling provides us a way to approximate marginals of a distribution via simulation, when we can't compute them analytically. Unfortunately, when there are tight correlations or anti-correlations between variables, Gibbs can have unacceptably long mixing times. In the next section, we'll explore different Markov chain Monte Carlo transitions that can overcome the problem of slow mixing.

4.3 Metropolis-Hastings

In this section, you will create a general framework for Metropolis-Hastings sampling. With Gibbs sampling, we changed the values of our variables one by one, and that gave us our new state. For regular chains, Gibbs sampling is a legal MCMC transition because it produces the right stationary distribution. But with high autocorrelations between variables, trying to change the value of one variable at a time tends to leave us in the same state or the same cluster of similar states, because the one variable we change at a time remains strongly constrained by its neighbors. So mixing is slow. We'd like to use a transition that lets us make bigger moves around the state space, but we still have to make sure our transition is a legal MCMC transition, in that it will result in a Markov chain with the stationary distribution we want.

The Metropolis-Hastings algorithm attempts to overcome these issues. We choose the transition we actually want – one that takes large steps around the state space. Then we modify that transition, using an acceptance probability, so that it turns into a legal MCMC transition. Our MCMC transition then has two steps. First we choose a proposed new state, using our transition probabilities $Q(x \to x')$. Then we accept that new state, with probability $A(x \to x')$. If the transition is accepted, our new state is the proposed new state, but if the transition is rejected, our new state is the same as the old state (see Fig. 8).

For a given proposal transition $Q(x \to x')$, the acceptance probability is given by:

$$A(x \to x') = \min\left[1,\ \frac{\pi(x')Q(x' \to x)}{\pi(x)Q(x \to x')}\right] \tag{1}$$

In this equation, $\pi(x)$ is the stationary probability that we're in state x, and $Q(x \to x')$ is the probability that if we're in state x we'll propose state x' as the new state. This acceptance probability will make the chain satisfy the detailed balance equation, so that the stationary distribution of the resulting Markov chain is the same as the distribution of the target joint probability distribution $\pi(X)$ we are modeling with the graphic model.

We should choose the proposal distribution Q with care. We want a proposal that takes big steps in the state space, which is to say a proposal that ends up changing the values of a lot of variables in one step. But we don't want a proposal that has a low acceptance probability, because that would mean being stuck in one state for a long time rather than moving about. A poor choice of Q will cause a long mixing time, and even after mixing will require an excessive amount of samples to explore the target distribution. There is no recipe for choosing a particular

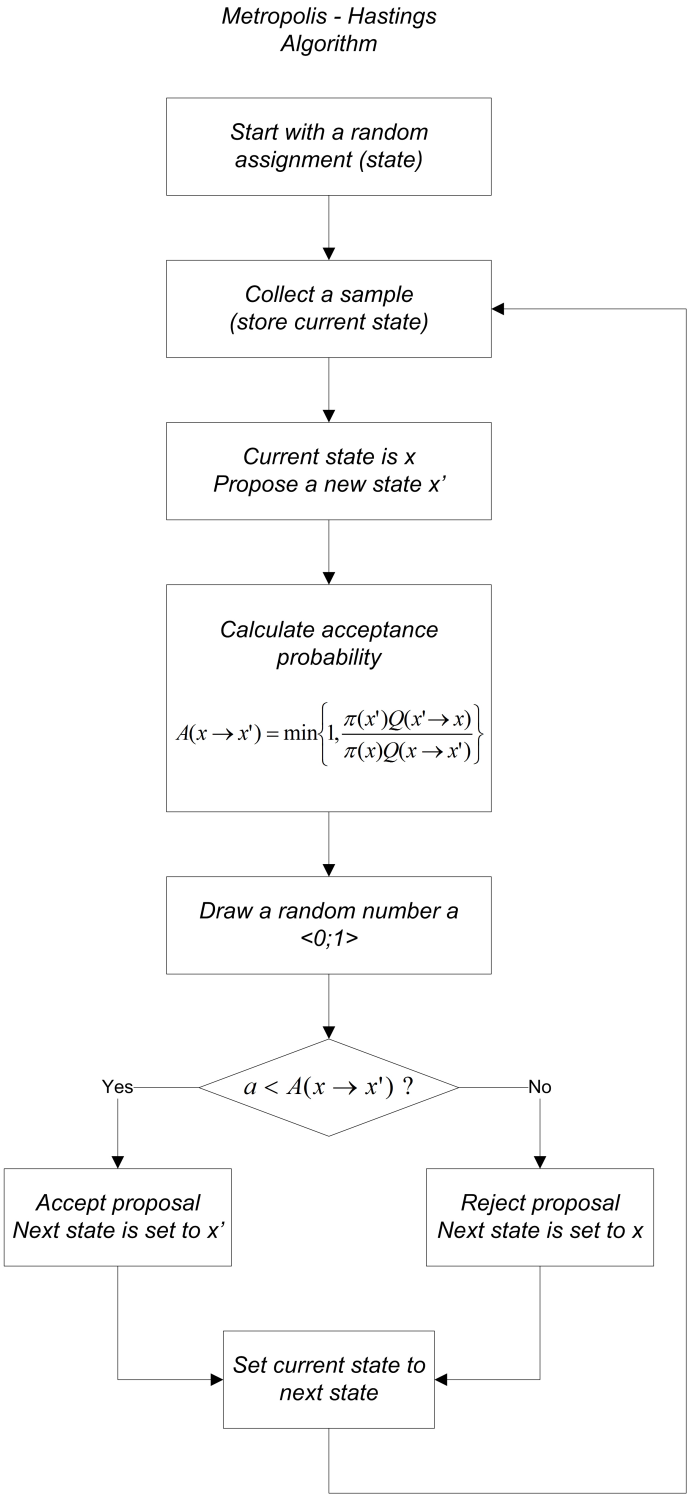


Figure 8: Metropolis-Hastings sampling process.

Q and a good choice for Q depends on the graphical model at hand. In realistic applications one needs to experiment with several different choices, which is precisely what you will do for the Ising grid in the remainder of this assignment.

The Metropolis-Hastings framework you create will be capable of using different proposal distributions. First, you will use your framework with the uniform distribution. Then, you will explore some variations of the Swendsen-Wang distribution. We will provide the implementations of the algorithms to sample from these proposal distributions, and you will need to compute the correct acceptance probability. Finally, you will assess the relative merits of each proposal distribution. First, implement a proposal distribution that uses the uniform distribution:

- **MHUniformTrans.m (5 points)** – This function defines the transition process using a uniform proposal distribution in the Metropolis-Hastings algorithm. It takes as input the current assignment of all variables and should return the next assignment of all variables. This proposed new state is already calculated for you, and you will need to fill in the code to compute the correct acceptance probability. You will use the acceptance probability equation (Eqn. 1). You might find **LogProbOfJointAssignment.m** useful here.

The uniform proposal distribution is useful to test your Metropolis-Hastings framework, but it tends to perform poorly in practice. With a uniform probability over proposed state transitions, we tend to propose many low-probability state transitions, which then get rejected by the acceptance step, so we tend to linger in one state rather than exploring the state space. In Ising models and image segmentation applications, pairwise Markov networks are usually used. In these networks, adjacent variables tend to take on the same values. This makes it hard to explore the space for proposal distributions which change the value of one variable at a time, such as Gibbs or the uniform proposal distribution. The Swendsen-Wang algorithm has been specifically designed for this kind of networks and overcomes this difficulty by changing, at the same time, a group of adjacent variables that have the same value.

4.3.1 Swendsen-Wang

In Swendsen-Wang, we start with a model that has variables connected in pairwise factors in a Markov graph. All the variables can take on the same values. To do MCMC, we start as usual in some state x. To get to the next state, we start with the graph. We break all the edges between variables that have different values. Then, we also break the rest of the edges between any nodes i and j with probability $(1 - q_{i,j})$, where $q_{i,j}$ is a probability that can depend on i and j but not their current assignment. You will compute that $q_{i,j}$ for the two different variations of Swendsen-Wang as described below.

Once we've broken all those edges, we're left with a graph with connected groups of variables, with all variables in a group sharing the same assignment l. We randomly pick one with a uniform distribution. That's Y. We choose a new assignment l', using the probability distribution R, which you will supply for the two different variations. With the new assignment to the variables in Y, we now have the new state x'; in other words, the new assignment x' is the same as x, except that the variables in Y are all labeled l' instead of l. We accept the state with probability A, which you will compute (see Fig. 9).

To compute $q_{i,j}$, R, and A, we use the following equations. Let $C(Y|x)$ be the probability that a set Y is selected to be updated using this procedure, given the current state is x. Then we can compute $Q(x \to x')$, the chance that we'll try a transition from x to x' by changing Y, as

$$Q(x \to x') = C(Y|x)R(Y = l'|x_{-Y}) \tag{2}$$

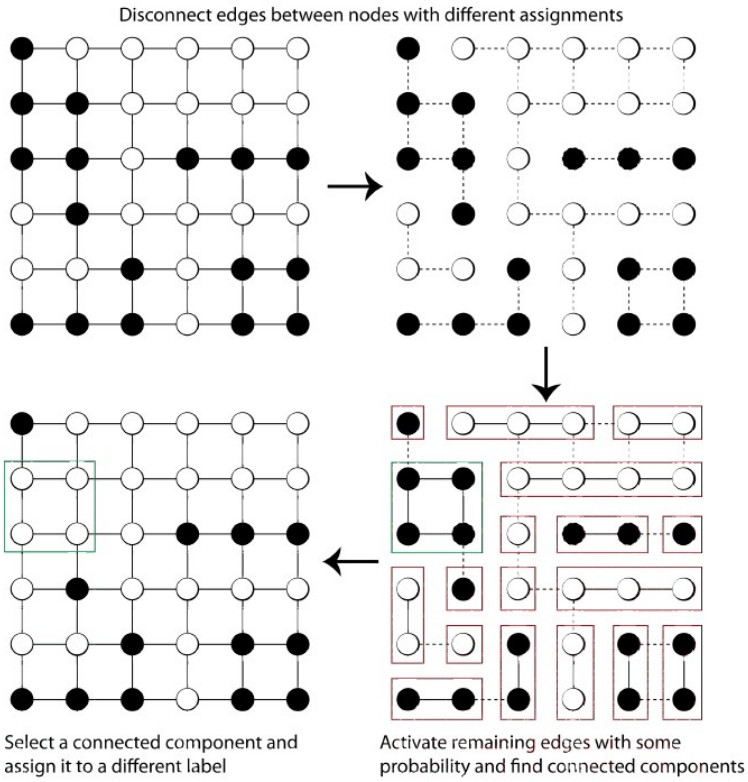


Figure 9: Swendsen-Wang procedure.

That is, if we're in x, we'll propose the state x' if we choose Y, and then change all the variables in Y to l'. Recall that the acceptance probability is given by Eqn. 1. Using Eqn. 2, we have that

$$\frac{Q(x' \to x)}{Q(x \to x')} = \frac{C(Y|x')R(Y = l|x'_{-Y})}{C(Y|x)R(Y = l'|x_{-Y})} \tag{3}$$

It is possible to show that

$$\text{QY\_ratio} = \frac{C(\mathbf{Y}|x')}{C(\mathbf{Y}|x)} = \frac{\prod_{(i,j)\in\mathcal{E}(\mathbf{Y},(\mathbf{X}'_{l'}-\mathbf{Y}))}(1-q_{i,j})}{\prod_{(i,j)\in\mathcal{E}(\mathbf{Y},(\mathbf{X}_{\mathbf{l}}-\mathbf{Y}))}(1-q_{i,j})} \tag{4}$$

where: $\mathbf{X_l}$ is the set of vertices with label l in x, $\mathbf{X}'_{l'}$ is the set of vertices with label l' in $\mathbf{x}'$; and where $\mathcal{E}(\mathbf{Y}, \mathbf{Z})$ (between two disjoint sets $\mathbf{Y}, \mathbf{Z}$) is the set of edges connecting nodes in $\mathbf{Y}$ to nodes in $\mathbf{Z}$.

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 two variants that experiment with different parameters for the proposal distribution. In particular, you will change the value of the $q_{i,j}$'s and $R(\mathbf{Y} = l|x_{-\mathbf{Y}})$. The two variants are as follows:

1. Set the $q_{i,j}$'s to be uniformly 0.5, and set the distribution R to be uniform.

2. Set R to be the block-sampling distribution (as defined in **BlockLogDistribution.m**) for sampling a new label and make $q_{i,j}$ dependent on the pairwise factor $\phi_{i,j}$ between i and j. In particular, set
$$q_{i,j} := \frac{\sum_u \phi_{i,j}(u,u)}{\sum_{u,v} \phi_{i,j}(u,v)} \tag{5}$$

The implementation of Swendsen-Wang consists of two parts. In the first part you will implement the transition process for both variants in **MSHWTrans.m**. In the second part you will incorporate the Swendsen-Wang variants in the MCMC driver, **MCMCInference.m**. Note that you will have to calculate the $q_{i,j}$'s only in **MCMCInference.m**. These are then passed on to **MHSWTrans.m** through an extra field in the input graphical model G, because it would be very ineffecient to recalculate these for each individual transition.

- **MHSWTrans.m (Variant 1) (3 points)** – This function defines the transition process associated with the Swendsen-Wang proposal distribution in Metropolis-Hastings. It takes as input the current assignment of all variables and the name of the Swendsen-Wang variant to use and should return the next assignment of all variables. You should fill in the code to compute the proposal distribution values and then use these to compute the acceptance probability. Implement the first variant for this test. Note that the log of QY_ratio, `log_QY_ratio`, is already computed for you in the code.
  Important: there is a bug in the current implementation. This function makes use of the helper function **scomponents.m**. This function is located in directory `gaimc`. To make it visible, run `addpath 'gaimc'`.

- **MHSWTrans.m (Variant 2) (3 points)** – Now implement the second variant of SW. Note: the first variant should still function after the second variant has been implemented.

With Swendsen-Wang, we will need to compute the values of our $q_{i,j}$'s, so we must update our inference engine:

- **MCMCInference.m PART 2 (4 points)** – Flesh this function out to run our Swendsen-Wang variants in addition to Gibbs. Your task here is to implement the calculations of the $q_{i,j}$'s for both variants of Swendsen-Wang in this function. To calculate the $q_{i,j}$'s, you should use Eqn. 5. In the code, the pairwise factor $\phi_{i,j}$ is represented by `edge_factor`.

Now that we've finished implementing all of these functions, let's compare these inference algorithms. For answering the following questions we strongly recommend using the dedicated test package provided by Leonid Fituni in the course forum. When you use this package, you should use the **TestToyFile.m** in stead of **TestToy.m**, which is referred to below. An advantage of using this package is that all resulting visualizations are saved to disk. Make sure you read the documentation in **TestToyFile.m** and change the appropriate parameters.
Important: There is a bug in the current implementation. There is a problem with the supplied random number generator file **rand.m**, it goes into a cycle where it repeats, which makes the MCMCInference runs not very useful. However, the supplied file is necessary to pass all the tests. When you have passed all the tests above and are ready for the assignment questions, please rename **rand.m** to **randOLD.m**.

1. **(10 points)** – For this question, repeat the experiment for LBP where we ran our Toy Network while changing the on and off-diagonal network. We will consider the cases where the on-diagonal weights are much larger and about equal to the off-diagonal weights.

While running this, use **VisualizeMCMCMarginals.m** to visualize the distribution of the Markov chain for **multiple** runs of MCMC (see **TestToy.m** for how to do this). We will examine how the mixing behavior and final marginals for each chain change in response to the change in the pairwise factor.

(a) **(5 points)** – Set the on-diagonal weight of our toy image network to 1 and off-diagonal weight to 0.2. Now visualize **multiple** runs with each of Gibbs, MHUniform, Swendsen-Wang variant 1, and Swendsen-Wang variant 2 using **VisualizeMCMC-Marginals.m** (see **TestToy.m** for how to do this). How do the mixing times of these chains compare? How do the final marginals compare to the exact marginals? Why?

(b) **(5 points)** – Set the on-diagonal weight of our toy image network to 0.5 and off-diagonal weight to 0.5. Now visualize **multiple** runs with each of Gibbs, MHUniform, Swendsen-Wang variant 1, and Swendsen-Wang variant 2 using **VisualizeMCMC-Marginals.m** (see **TestToy.m** for how to do this). How do the mixing times of these chains compare? How do the final marginals compare to the exact marginals? Why?

2. **(3 points)** When creating our proposal distribution for Swendsen-Wang, if you set all the $q_{i,j}$'s to zero, what does Swendsen-Wang variant 2 reduce to?

5 Conclusion

Congratulations! You've now implemented a full suite of inference engines for exact inference, approximate inference, and sampling. These methods are useful for making predictions and gaining understanding of the world around us, a process we'll explore more in PA6. Of course, one underlying assumption has been that we already know the basic facts of which variables influence one another – an assumption that is certainly not always the case! So stay tuned, we'll look into how to eliminate more of our assumptions later in the course.