Homework 3: Message Passing Inference

University of Arizona CSC 535: Probabilistic Graphical Models

Homework due at 11:59pm on October 19, 2016

We will examine marginal inference in graphical models. We will use factor graph representations to implement the sum-product variant of belief propagation (BP). In addition to the material presented during lecture, the following resources will be helpful in understanding details of the sum-product and max-product BP algorithms:

1. Secs. 20.1-20.2 of Kevin Murphy's "Machine Learning: A Probabilistic Perspective"
2. Sec. 8.4 of Chris Bishop's "Pattern Recognition and Machine Learning"
3. Kschischang, Frey, & Loeliger, *IEEE Trans. Information Theory* 47, pp. 498-519, 2001.

You must write your own implementation of each algorithm. You may not copy code from other students or existing software packages.

Question 1:

For a graph with cycles the sum-product algorithm may not always compute the correct marginal distributions. Here we explore the variable elimination algorithm, which *is* guaranteed to find exact marginals, albeit with additional computational expense. Consider the undirected graphical model in Figure 1, which is a small example of the spatial lattices used in computer vision applications.

a) Consider the elimination algorithm discussed in lecture, and in Sec. 20.3 of Murphy's textbook and Sec. 8.4.1 of Bishop's textbook. Suppose we compute the marginal of node 1 via the following elimination ordering: $\{5, 2, 6, 8, 4, 9, 7, 3, 1\}$. *What is the largest clique that is produced?*

b) Suppose we instead compute the marginal of node 1 via the following elimination ordering: $\{9, 7, 3, 6, 8, 5, 2, 4, 1\}$. *What is the largest clique that is produced? What is the order of complexity for each elimination order? Which is order more efficient?*

c) Using intuition from these examples, give an upper bound on the treewidth of an $n \times n$ *grid. Explain your answer, which should be much smaller than the number of nodes* n^2.

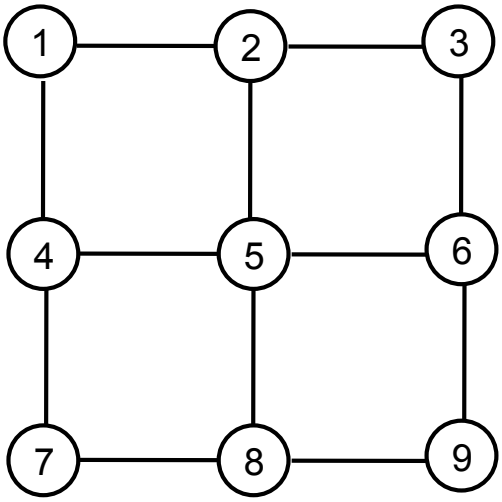

Figure 1: An undirected graphical model of two-dimensional spatial dependencies among nine variables arranged in a 3×3 grid. All variables are discrete, and have the same number of states.

Question 2:

We have provided Matlab code implementing a data structure to store the graph adjacency structure, and numeric potential tables, defining any discrete factor graph. We also provide code that explicitly builds a table containing the probabilities of all joint configurations of the variables in a factor graph, and sums these probabilities to compute the marginal distribution of each variable. Such "brute force" inference code is of course inefficient, and will only be computationally tractable for very small models.

We recommend (but do not require) that you use these same data structures for your own implementation of the sum-product algorithm, by implementing `run_loopy_bp_parallel.m` and `get_beliefs.m`. To gain intuition for the graph structure, examine the output of `make_debug_graph.m`. Think of the code we provide as a starting point: you are welcome to create additional functions or data structures as needed. You may use other programming languages if you wish, but will be responsible for translating the Matlab-compatible data we provide into other formats.

a) *Implement the sum-product algorithm. Your code should support an arbitrary factor graph linking a collection of discrete random variables. Use a parallel message update schedule, in which all factor-to-variable messages are updated given the current variable-to-factor messages, and then all variable-to-factor messages given the current factor-to-variable messages. Initialize by setting the variable-to-factor messages to equal 1 for all states. Be careful to normalize messages to avoid numerical underflow.*

Hints: While it is acceptable to make significant use of loops in your sum-product code, other implementation strategies will lead to faster experiments. In Matlab, a standard vector is a multi-dimensional array in which the first dimension has size equal to the vector's length, and all other dimensions have length one. The `reshape` command can convert vectors to arrays where some other dimension (chosen to match a factor array)

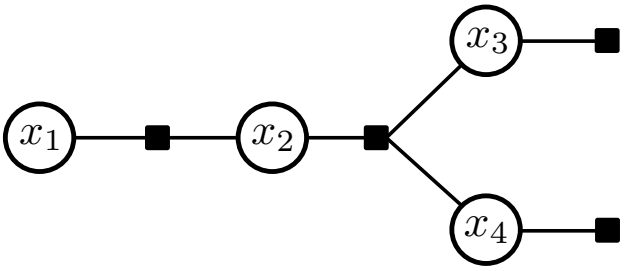

Figure 2: A tree-structured factor graph in which four factors link four random variables. Variable x_2 takes one of three discrete states, and the other three variables are binary.

has length greater than one. To easily compute the product of a message array and a factor array, apply `bsxfun` to the `@times` function. The `sum` command provides an optional dimension argument, to allow marginalization of any dimension in an array.

b) *Consider the four-node, tree-structured factor graph illustrated in Figure 2. Variable x_2 takes one of three discrete states, and the other three variables are binary. Numeric values for the potential functions are defined in* `make_debug_graph.m`*. Run your implementation of the sum-product algorithm on this graph, and report the marginal distributions it computes. Verify that these are consistent with the results of* `marg_brute_force.m`*.*

Question 3:

We investigate the design of algorithms for reliable communication over noisy channels. We focus on error correcting codes based on highly sparse, *low density parity check* (LDPC) matrices, and use the sum-product variant of the loopy belief propagation (BP) algorithm to estimate partially corrupted message bits. For background information on LDPC codes, see Chap. 47 of MacKay's *Information Theory, Inference, and Learning Algorithms*, which is freely available online: `http://www.inference.phy.cam.ac.uk/mackay/itila/`.

We consider rate 1/2 error correcting codes, which encode N message bits using a $2N$-bit codeword. LDPC codes are specified by a $N \times 2N$ binary parity check matrix H, whose columns correspond to codeword bits, and rows to parity check constraints. We define $H_{ij} = 1$ if parity check i depends on codeword bit j, and $H_{ij} = 0$ otherwise. Valid codewords are those for which the sum of the bits connected to each parity check, as indicated by H, equals zero in modulo-2 addition (i.e., the number of "active" bits must be even). Equivalently, the modulo-2 product of the parity check matrix with the $2N$-bit codeword vector must equal a N-bit vector of zeros. As illustrated in Fig. 3, we can visualize these parity check constraints via a corresponding factor graph. The parity check matrix H can then be thought of as an adjacency matrix, where rows correspond to factor (parity) nodes, columns to variable (codeword bit) nodes, and ones to edges linking factors to variables.

a) Implement code that, given an arbitrary parity check matrix H, constructs a corresponding factor graph. The parity check factors should evaluate to 1 if an even number of adjacent

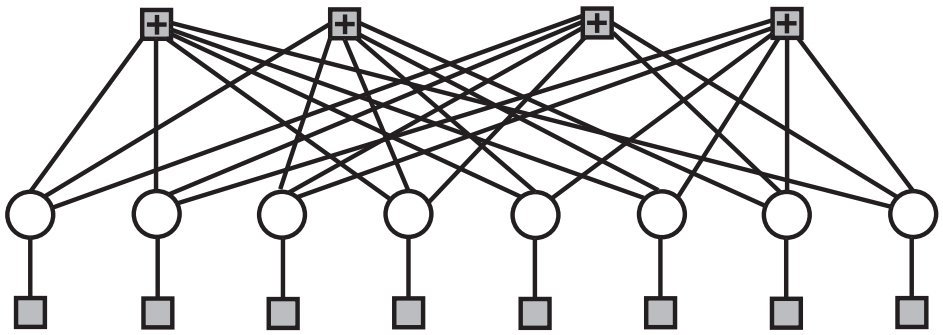

Figure 3: A factor graph representation of a LDPC code linking four factor (parity constraint) nodes to eight variable (message bit) nodes. The unary factors encode noisy observations of the message bits from the output of some communications channel.

bits are active (equal 1), and 0 otherwise. Your factor graph representation should interface with the sum-product BP code developed in Question 2. Define a small test case, and verify that your graphical model assigns zero probability to invalid codewords.

b) *Load the $N = 128$-bit LDPC code provided in* `ldpc36-128.mat`. *To evaluate decoding performance, we assume that the all-zeros codeword is sent, which always satisfies any set of parity checks. Using the* `rand` *method, simulate the output of a binary symmetric channel: each transmitted bit is flipped to its complement with error probability $\epsilon = 0.05$, and equal to the transmitted bit otherwise. Define unary factors for each variable node which equal $1 - \epsilon$ if that bit equals the "received" bit at the channel output, and ϵ otherwise. Run the sum-product algorithm for 50 iterations of a parallel message update schedule, initializing by setting all variable-to-factor messages to be constant. After the final iteration, plot the estimated posterior probability that each codeword bit equals one. If we decode by setting each bit to the maximum of its corresponding marginal, would we find the right codeword?*

c) *Repeat the experiment from part (b) for 10 random channel noise realizations with error probability $\epsilon = 0.06$. For each trial, run sum-product for 50 iterations. After each iteration, estimate the codeword by taking the maximum of each bit's marginal distribution, and evaluate the Hamming distance (number of differing bits) between the estimated and true (all-zeros) codeword. On a single plot, display 10 curves showing Hamming distance versus iteration for each Monte Carlo trial. Is BP a reliable decoding algorithm?*

d) *Repeat part (c) with two higher error probabilities, $\epsilon = 0.08$ and $\epsilon = 0.10$. Discuss any qualitative differences in the behavior of the loopy BP decoder.*

For the LDPC codes we consider, we also define a corresponding $2N \times N$ generator matrix G. To encode an N-bit message vector we would like to transmit, we take the modulo-2 matrix product of the generator matrix with the message. The generator matrix has been constructed (via linear algebra over the finite field GF(2)) such that this product always produces a valid $2N$-bit codeword. Geometrically, its columns are chosen to span the null space of H. We use a systematic encoding, in which the first N codeword bits are simply

copies of the message bits. The problems below use precomputed (G, H) pairs produced by Neal's LDPC software: `http://www.cs.utoronto.ca/~radford/ldpc.software.html`.

e) Load the $N = 1600$-bit LDPC code provided in `ldpc36-1600.mat`*. Using this, we will replicate the visual decoding demonstration from MacKay's Fig. 47.5. Start by converting a 40×40 binary image to a 1600-bit message vector; you may use the* `logo` *image we provide, or create your own. Encode the message using the provided generator matrix G, and add noise with error probability $\epsilon = 0.06$. For this input, plot images showing the output of the sum-product decoder after 0, 1, 2, 3, 5, 10, 20, and 30 iterations. The* `rem` *method may be useful for computing modulo-2 sums. You can use the* `reshape` *method to easily convert between images and rasterized message vectors.*

f) Repeat part (e) with a higher error probability of $\epsilon = 0.10$, and discuss differences.