PGMs session 4 Inference: Belief Propagation

As you would have discovered from last week's work, determining a full joint probability might be an extremely expensive exercise when there are many variables. We are now going to study a number of *message-passing* algorithms to do this more efficiently.

1 Preparation: Read Barber Section 5.1 – Marginal Inference.

Read Section 5.1 in Barber. This might seem short, but the Koller videos (below) will substantially expand on it.

- Section 5.1 starts out with variable elimination and then shows how this leads to the sum-product algorithm in linear and tree-structured networks.
- This is specifically developed here for *factor graphs*. This consists of nodes for the individual variables that are connected to factor nodes according to the potentials that these variables partake in. In these graphs there are two distinct types of *messages*:
  - From a variable node to a factor node the message is the product of all the *other* messages that lead into that variable.
  - From a factor node to a variable node this product (from the previous step) is multiplied with the factor potential, and then all the other variables (except the destination variable) gets marginalised out.

2 Preparation: Watch the Koller videos on inference, variable elimination, and message passing

Watch the following videos in the Coursera course Probabilistic Graphical Models 1: Representation, Week 5:

- *Knowledge Engineering & Conclusion: Knowledge Engineering (23:05)*

Watch the following videos in the Coursera course Probabilistic Graphical Models 2: Inference, Week 1:

- *Overview: Conditional Probability Queries (15:22)*
- *Overview: MAP Inference (9:47)*
- *Variable Elimination: Variable Elimination Algorithm (16:17)*
- *Variable Elimination: Complexity of Variable Elimination (12:48)*
- *Variable Elimination: Graph-Based Perspective on Variable Elimination (15:25)*
- *Variable Elimination: Finding Elimination Orderings (11:58)*

Watch the following videos in the Coursera course Probabilistic Graphical Models 2: Inference, Week 2:

- *Message Passing in Cluster Graphs: Belief Propagation Algorithm (21:21)*
- *Message Passing in Cluster Graphs: Properties of Cluster Graphs (15:00)*
- *Message Passing in Cluster Graphs: Properties of Belief Propagation (9:31)*
- *Optional: Loopy Belief Propagation: BP in Practice (15:38)*
- *Optional: Loopy Belief Propagation: Loopy BP and Message Decoding (21:42)*

Notes on these videos:

- *Knowledge Engineering & Conclusion: Knowledge Engineering (23:05)* is for general interest; we have not discussed template and plate models as yet, so do not be overly worried when you encounter them.
- *Overview: Conditional Probability Queries (15:22)* and *Overview: MAP Inference (9:47)* provide overviews of work still to be done. (Do note in *Overview: MAP Inference (9:47)* the example that shows that the max of the joint is not necessarily equal to the max of the marginals.)
- The *Variable Elimination* videos discuss variable elimination. This is an important topic which will also find use in later work.
- The *Message Passing in Cluster Graphs* videos introduce belief propagation (i.e., the sum-product / Shafer-Shenoy algorithm). In contrast to Barber, who approaches this from a factor-graph viewpoint, Koller uses a more general formulation that paves the way for the junction-tree algorithm (future work).
- *Optional: Loopy Belief Propagation: BP in Practice (15:38)* Shows some variants and tricks to improve convergence in loopy belief propagation.
- *Optional: Loopy Belief Propagation: Loopy BP and Message Decoding (21:42)* shows the historical link between loopy belief propagation and error correction coding.
- Note that the factor graph approach translates to a specific case of this more general formulation – the Bethe cluster graph. The two types of messages that we saw in the factor graph is a simplification arising from the fact that in a Bethe cluster graph, the sepset corresponds exactly to the particular variable on that specific link.
- When the variable clusters are connected in loops, the algorithm might converge to the wrong solution, or might even oscillate. Quite often though, it converges to the correct solution.

Make sure you understand the following:

- A 'message' really is a *distribution/potential* describing what the originators of the message believes about the variables they have in common with the target cluster. The target cluster's own opinion about itself should not be included in this. (Otherwise you get the inflated potential runaway ego situation we so often observe in cults, be it political or religious.)
- Initially, messages are set to unity. This just means they have no opinion about their variables yet – the potential of the cluster rules.
- Numerical issues: With unnormalised potentials (as we typically use in MRFs) we run a risk of exhausting our numerical range with either under- or overflows. We basically have two avenues to cure this (see discussion in Barber at the end of Section 5.1.2, and also Section 5.1.4):
  - Normalise the messages before passing them on. The works fine, but do note that it will no longer be possible to calculate the partition function value Z by finding the normalisation constant at any arbitrary variable cluster. You will have to normalise all your clusters separately (resulting in a $Z = 1$), or alternatively find the full joint distribution and calculate Z from that.
  - Use logarithmic potentials. With these we need to extend our inventory of factor operations somewhat:
    - Potential multiplication now change to potential summation.
    - Observing evidence/conditioning is done as before.
    - To do marginalisation you need to first convert your logarithmic potentials to linear form again. However, direct/naive linearisation does not work – you immediately fall prey to the original over/underflow problem again. The trick (known as the logsumexp trick) is to realise that:

$$\begin{aligned}\log(\sum_i e^{L_i}) &= \log(e^{\max\{L_j\}} \sum_i e^{L_i - \max\{L_j\}}) \\ &= \max\{L_j\} + \log(\sum_i e^{L_i - \max\{L_j\}})\end{aligned}$$

- The *cluster belief* is the product of the cluster potential and all the messages entering it. It serves as a pseudo-marginal distribution.
- The *sepset belief* is the product of the two messages travelling in opposite directions through it. It serves as a pseudo-marginal distribution.

- Convergence implies *calibration*: This simply means that on convergence, we can choose any two of the beliefs available (cluster as well as sepset), and calculate marginal distributions for the variable(s) they have in common. The distributions obtained in this manner must be identical. It should not happen that one part of the system has a different idea about the distribution of some variable(s) than what some other part has.
- Note that the product of the cluster beliefs divided by the sepset beliefs is the same as the product of the original potentials (known as reparameterisation). While they therefore result in the same joint potential, the message passing procedure have at the same time refined them to provide the marginal distributions for each cluster.
- Those familiar with the use of the forward-backward algorithm in the context of HMMs will see that the belief propagation algorithm generalises that algorithm.

3 Practical: LBP for the Hamming (7,4) error-correction code

Objectives:

- Implement/use sum-product messages to calculate marginal distributions without first calculating the full joint probability; i.e., (approximately) calculate the marginal distributions while avoiding the exponential blow-up of the full joint distribution.
- Verify that after convergence the system achieves calibration; i.e., the various beliefs in the system agrees on the marginal distributions of the variables it knows about.
- Verify the re-parameterisation property; i.e., that we can recover the full joint distribution from the various cluster and sepset beliefs. Of course we normally would not want to do this, since it once again introduces the exponential factor size blow-up.
- (Optional) Compare the behaviour of (the more generic) cluster graphs with that of factor graphs.

Task 1: In the following, use only the basic factor operations we encountered previously – we first want to foster an understanding of the basics of message passing. We are going to re-implement the hard-decoding (BSC) version of the Hamming (7,4) code of last week, but this time we will embed them in a special graph structure called a cluster graph (CG). Construct it as follows:

(a) First construct all the factors as you did in Session 3. Also observe and reduce the factors using your same choice of received bits, $r_0, \ldots, r_6$ as for Session 3.

(b) The scope of each factor is the set of random variables it contains. In contrast to Session 3, we are not going to now create a joint factor by multiplying all available factors out; we instead want to arrange them into a graph structure. However, if the scope of a factor is a subset of another factor, we can (optionally) simplify things by assimilating it in that other factor; we do this by simply multiplying the factors. (Note, when a factor is a subset of more than one other factor, we will only multiply it out with *one* of them; i.e., each factor occurs only once in the system.) Absorb all the $p(r_i|b_i)$ factors (that have been observed and reduced) into appropriate factors.

(c) The (in this case three) factors you are now left with, need to be linked up in a graph structure which obeys the running intersection property (RIP). RIP requires that for all variables X_i in the PGM, when considering only those nodes/clusters containing X_i (as well as the links between them), the resulting sub-graph must form a *tree*. (Study the Koller video *Message Passing in Cluster Graphs: Properties of Cluster Graphs (15:00)* until you are clear about what RIP entails.) To form this graph involves two steps:

   - Decide which factors are to be linked and
   - specify their *sepset*; i.e., the set of variables those two factors will exchange information about. Note, the sepset is a *subset* of the random variables those two factors have in common.

   In particular, pay careful attention to which sepsets contain b_0 (the variable that is common to all three parity checking factors). After having done this yourself you can confirm that you end up with something similar to Figure 1 (In this figure, the factors $\Phi_O(b_i, r_i)$ – which are the $p(r_i|b_i)$ factors – have not yet been absorbed into larger factors). We represent the clusters inside ellipses, and their sepsets inside rectangles.

Now we are ready to start passing messages around between the nodes in this graph:

(d) Since we have a loopy structure, we have to place initial messages at all relevant places (consult the videos). Each message is a `emdw DiscreteTable` factor, the scope of which is being specified by the variables in the corresponding sepset. Each message is contained in a separate such variable –

you might want to consider a coding scheme such as `m01` being the message that is sent from cluster 0 to cluster 1, etc. Initialise all messages as uninformative distributions (i.e., make them uniform distributions).

(e) Now repeatedly pass messages between the various nodes using the loopy belief propagation algorithm. You have to determine a schedule for passing messages. For this specific application a natural option is to go once in one direction around the loop, and then once in the other direction around the loop. You will also have to take precautions against numerical under/overflow. For this it probably is easiest to simply normalise each message.

(f) Iterate/repeat this message passing until *calibration* is achieved between all clusters (i.e., the sepset beliefs do not change any more). Consult the Koller video *Message Passing in Cluster Graphs: Properties of Belief Propagation (9:31)* for determining how a cluster belief and a sepset belief is calculated. You will have to decide how you are going to quantify such a *change* in belief[1]. When a graph is calibrated, all beliefs are in agreement as to what the distribution for the variable(s) are.

(g) It might also be interesting to plot the convergence behaviour as the biggest change in sepset belief as a function of the iteration number.

(h) Decoding is complicated by the fact that picking the max of the marginal distributions is not necessarily equivalent to picking the maximum of the joint assignment – an elegant solution for this dilemma forms part of the MAP/max-sum message passing algorithm which we will discuss at a later stage. For now we will have to content ourselves by interpreting the maximum values of the cluster beliefs – this should give us a fair idea as to what is happening. (Optional:) Alternatively we can form the full joint by making use of reparameterisation (see the last part of the Koller video *Message Passing in Cluster Graphs: Properties of Belief Propagation (9:31)*) and then picking the max of the joint distribution. The division operator in `emdw` will be useful here for dividing by the sepset potentials. This should return the most likely answer.

(i) How do the results compare with your results from last week?

(j) Due to the small size of the problem we are considering, the computational load will be higher than what we previously encountered. This seems to negate our very reason for doing message passing. Confirm that you are clear why message passing can alleviate the computational load in bigger problems.

(k) Can you create situations with bad convergence behaviour?

(l) Investigate what happens when every pair of parity checking clusters do share b_0 i.e a loop of b_0 sepsets violates the uniqueness requirement of RIP.

Task 2: You now have experience in what is involved in LBP message passing. Let's now automate it somewhat: Use the example file and the explanation below to implement LBP using the built-in `emdw` functionality. Also compare your results with that of Task 1.

Some instructions are discussed below – see the `LBP_example.cc` example file for more details. And remember that the Terminal command `grep` should be your friend – if you haven't made the acquaintance yet, now is a good time.

Some declarations you will need: Stick this in close to the top of your .cc file:

```
// standard library headers
#include <iostream>  // cout, endl, flush, cin, cerr
#include <vector>    // vector
#include <map>       // map

// emdw headers
#include "emdw.hpp"
#include "discretetable.hpp"
#include "clustergraph.hpp"
#include "lbp_cg.hpp"
#include "lbu_cg.hpp"
#include "messagequeue.hpp"

// These are to avoid doing std:: and emdw:: all the time. Don't do
// this in an .hpp file
using namespace std;
using namespace emdw;
```

[1]Hint: A nice way to measure the 'distance' between distributions, is the Kullback-Leibler divergence (go and look it up). That is already available for `DiscreteTable` via the `distance` member function.

Compiling a valid RIP cluster graph: In Task 1 above you by inspection figured out which variable(s) to remove from certain sepsets so that the graph obeys the RIP. In a more complex system this might be somewhat tricky. Fortunately the LTRIP (layered trees running intersection property) algorithm will effortlessly do that for us. Consult https://arxiv.org/abs/2110.02048 for more detail (and at the same time find out why you would prefer cluster graphs over factor graphs). This is implemented in `emdw`'s `ClusterGraph` class. Some specifics to note:

- Collect all the relevant `Factor`s in a `vector`:
  ```
  vector< rcptr<Factor> > factorPtrs;
  ```
  Typically we use the `push_back` function to add the `rcptr<Factor>`s to the back of this vector. Their order is unimportant.
- Assign all the observed variables in a `map<RVIdType, AnyType>`:
  ```
  map<RVIdType, AnyType> obsv;
  ```
  The key/first part identifies the particular RV, while the value/second part allocates its observed value. Go and check out `AnyType` in the userguide. You will do the assignments with something like:
  ```
  obsv[r0] = int(1); //or T(1). Note explicit type
  ```
- Now build the `ClusterGraph`:
  ```
  // for a factor graph, use BETHE instead of LTRIP
  // for a junction tree, use JTREE instead of LTRIP
  ClusterGraph cg(ClusterGraph::LTRIP, factors, obsv);
  //cout << cg << endl;
  ```
- You can generate a graphviz .dot file of it via:
  ```
  // export the graph to graphviz .dot format
  cg.exportToGraphViz("your_name_for_it");
  ```
  After locating the .dot file, apply the dot command shown on your screen to generate a .pdf showing the graph.

Calibrating the graph via LBP:

- Initialise the message queue and messages:
  ```
  map<Idx2, rcptr<Factor> > msgs;
  MessageQueue msgQ;
  ```
- Apply the relevant inference algorithm to calibrate the graph
  ```
  // loopyBP_CG implements the loopy belief propagation
  // (Shafer-Shenoy) algorithm. Using loopyBU_CG instead results
  //  in the loopy belief update (Lauritzen-Spiegelhalter)
  // algorithm.
  unsigned nMsgs = loopyBP_CG(cg, msgs, msgQ);
  cout << "Sent " << nMsgs << " messages before convergence\n";
  ```

And query the calibrated graph:

- Ask about the distribution of a particular RV or RVs. (You can ask about the joint of several IDs by specifying more than one variable, separated by commas. This is only available for variables that somewhere occur jointly in a cluster.):
  ```
  // If you calibrated with loopyBU_CG, you will have to use
  // queryLBU_CG here.
  rcptr<Factor> qPtr = queryLBP_CG(cg, msgs, {varId})->normalize();
  cout << *qPtr << endl;
  ```
- Ask about the probability that the queried RV(s) takes on a particular value:
  ```
  double p1 = qPtr->potentialAt({varId}, {T(1)});
  cout << p1 << " ";
  ```

Task 3 (optional): Just for comparison you might also want to repeat the above, but using factor graphs (or junction trees) instead (single variable to change in the code). Compare the results, especially in terms of convergence behaviour.

Further exploration: The Koller video *Optional: Loopy Belief Propagation: BP in Practice (15:38)* contains a number of very useful ideas to explore: tree reparameterisation, residual belief propagation, synchronous vs. asynchronous propagation, and message damping. All are worthwhile to look at – some of them are being used in Task 2 above.

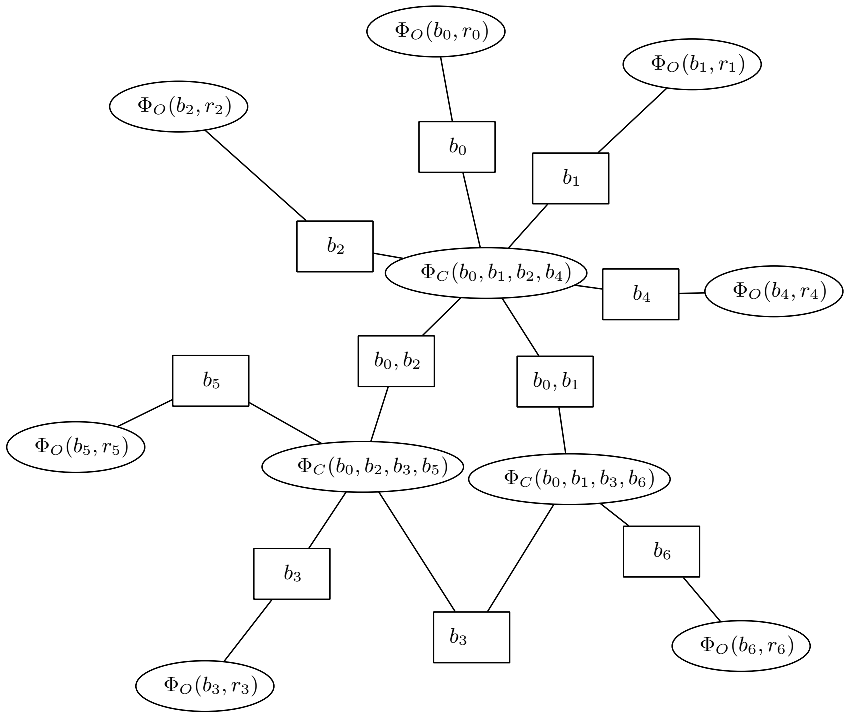

Figure 1: A cluster graph for the Hamming (7,4) error-correcting code. Note the sepset in the bottom center where b_0 has been omitted to achieve valid RIP. With the r_i's observed, this graph can be simplified to only an inner loop of three clusters by assimilating each resulting $\Phi_O(b_i)$ into the Φ_C cluster it is connected to.