1 Message Passing: Sum-Product (Review)

From last class, we know that

\[ q_{n \rightarrow m}(x_n) = \prod_{m' \in M(n) \setminus m} r_{m' \rightarrow n}(x_n) \]

\[ r_{m \rightarrow n}(x_n) = \sum_{x_m \in x_n} f_m(x_m) \prod_{m' \in N(m) \setminus \{n\}} q_{n' \rightarrow m}(x_{n'}) \]

\( q_{n \rightarrow m}(x_n) \) means the information propagated from variable node \( n \) to factor node \( f_m \); \( r_{m \rightarrow n}(x_n) \) is the information propagated from factor node \( f_m \) to variable node \( n \). And our goal is to compute the marginal probability for each variable \( x_n \):

\[ P(x_n) = \frac{1}{Z} \prod_{m \in M(n)} r_{m \rightarrow n}(x_n). \]

The joint distribution of two variables can be found by, for each joint assignment to both variables, performing message passing to marginalize out all other variables, and then renormalizing the result:

\[ P(x_i, x_j) = \frac{1}{Z_{\{i,j\}}} \left( \prod_{m \in M(i)} r_{m \rightarrow i}(x_i) \right) \left( \prod_{m \in M(j)} r_{m \rightarrow j}(x_j) \right) \]

In the original problem, the marginal probability of variable \( x_n \) is obtained by summing the joint distribution over all the variables except \( x_n \):

\[ P(x_n) = \frac{1}{Z} \sum_{x_1} \cdots \sum_{x_{n-1}} \sum_{x_{n+1}} \cdots \sum_{x_n} \prod_{m} f_m(x_m). \]

And by pushing summations inside the products, we obtain the efficient algorithm above.
2 Max-Sum

In practice, sometimes we wish to find the set of variables that maximizes the joint distribution

\[ P(x, N) = \frac{1}{Z} \prod_m f_m(x_m) \]

Removing the constant factor, it can be expressed as

\[
\max_{x_1, \ldots, x_N} \prod_m f_m(x_m) = \max_{x_1} \ldots \max_{x_N} \prod_m f_m(x_m)
\]

Figure 1 shows an example, in which the shadowed variables \( x_j, x_k, \) and \( x_l \) block the outside information flow. So to compute \( P(x_i|x_j, x_k, x_l) \), we can forget everything outside them, and just find assignments for inside variables:

\[
\max_{\text{inside var}} \prod_m f_m(x_m)
\]

Like the sum-product algorithm, we can also make use of the distributive law for multiplication and push maxs inside the products to obtain an efficient algorithm. We can put max whenever we see \( \sum \) in the sum-product algorithm to get the max-sum algorithm, which now actually is max-product (Viterbi) algorithm. For example,

\[
r_{m \rightarrow n}(x_n) = \max_{x_m \backslash x_n} f_m(x_m) \prod_{n' \in N(m) \backslash \{n\}} q_{n' \rightarrow m}(x_{n'})
\]

Since products of many small probabilities may lead to numerical underflow, we take the logarithm of the joint distribution, replacing the products in the max-product algorithm with sums, so we obtain the max-sum algorithm.

\[
\max \prod f_m \rightarrow \max \log \prod f_m \rightarrow \max \sum \log f_m
\]
3 Tree Decomposition

If we consider a decision problem instead of a numerical version, the original max-product algorithm will be:

\[
\text{find } x_1, ..., x_N \text{s.t. } \bigwedge_m f_m(x_m).
\]

We need to find some assignments to make it 1, which can be seen as a reduction from the 3-SAT problem (constraint satisfaction). So the problem is \(NP\)-complete in general.

To solve the problem, we force the graph to look like a tree, which is tree decomposition. Figure 2 shows an example.

Given a Factor Graph, we first need to make a new graph (Dependency Graph) by replacing each factor with a clique, shown in Figure 3. Then we apply the tree decomposition.

Tree decomposition can be explained as: given graph \(G = (V, E)\), we want to find \((\{X_i\}, T)\), \(X_i \subseteq V\), \(T = \text{tree over}\{X_i\}\). It should satisfy 3 conditions:

1. \(\bigcup_i X_i = V\), which means the new graph should cover all the vertex;
2. For \((u, v) \in E\), \(\exists X_i\) such that \(u, v \in X_i\);
3. If \(j\) is on the path from \(i\) to \(k\) in \(T\), then \((X_i \cap X_k) \subseteq X_j\) (running intersection property).

Using this method, we can get the new graph in Figure 3 with \(X_1 = \{A, B\}\), \(X_2 = \{B, C, D\}\), and \(X_3 = \{D, E\}\). The complexity of original problem is \(O((N + M)(k + l)2^{l-1})\), with \(l = \max_m |N(m)|\). By tree decomposition, we can obtain \(l = \max_i |X_i|\). Figure 4 shows the procedure to do tree decomposition on a directed graphical model.
Figure 3: Dependency graph for tree decomposition (vertex for each variables)

Figure 4: The procedure of tree decomposition on a directed graphical model (we can directly get Dependency Graph by moralization)
A new concept is the treewidth of a graph:

\[ \text{treewidth}(G) = \min_{(X_i,T)} \max_i |X_i| - 1 \]

For example, \( \text{treewidth}(\text{tree}) = 1 \), \( \text{treewidth}(\text{cycle}) = 2 \), and the worst case, \( \text{treewidth}(K_n) = n - 1 \) (\( K_n \) is a complete graph with \( n \) vertices). If the treewidth of the original graph is high, the tree decomposition becomes impractical.

Actually, finding the best tree decomposition is \( \mathcal{NP} \)-complete. One practical way is **Vertex Elimination**:

1. choose vertex \( v \) (heuristicly, choose \( v \) with fewest neighbors);
2. create \( X_i \) for \( v \) and its neighbors;
3. remove \( v \);
4. connect \( v \)'s neighbors;
5. repeat the first four steps until no new vertex.

**Vertex Elimination** cannot ensure to find the optimum solution. Figure 5 shows an example of this method on a single cycle.

Another way to do tree decomposition is **Triangulation**:

1. find cycle without chord (shortcut);
2. add chord;
3. repeat the first two steps until triangulated (no cycles without chords).

The cliques in the new graph are \( X_i \) in the tree decomposition.

![Figure 5: An example of Vertex Elimination on a single cycle](image-url)