CS5340 Assignment 1: Belief Propagation and Maximal Probability

1. Overview

Welcome to the course! In this first assignment, you will write code to perform inference on simple tree-like graphs using belief propagation. You will write the code for both the sum-product and max-product algorithm.

References: Lecture 4 and 5

Honour Code. This coding assignment constitutes **15%** of your final grade in CS5340. Note that plagiarism will not be condoned! You may discuss with your classmates and check the internet for references, but you MUST NOT submit code/report that is copied directly from other sources!

2. Submission Instructions

Items to be submitted:

- **Source code (lab1.py).** This is where you fill in all your code.
- **Report (report.pdf).** This should describe your implementation and be no more than one page.

Please indicate clearly your name and student number (the one that looks like A1234567X) in the report as well as the top of your source code. Zip the two files together and name it in the following format: **A1234567X_lab1.zip** (replace with your student number).

Submit your assignment by **17 September 2023, 2359HRS** to LumiNUS. 25% of the total score will be deducted for each day of late submission.

3. Getting Started

This assignment as well as the subsequent ones require Python 3.5, or later. You need certain python packages, which can be installed using the following command:

```
pip install -r requirements.txt
```

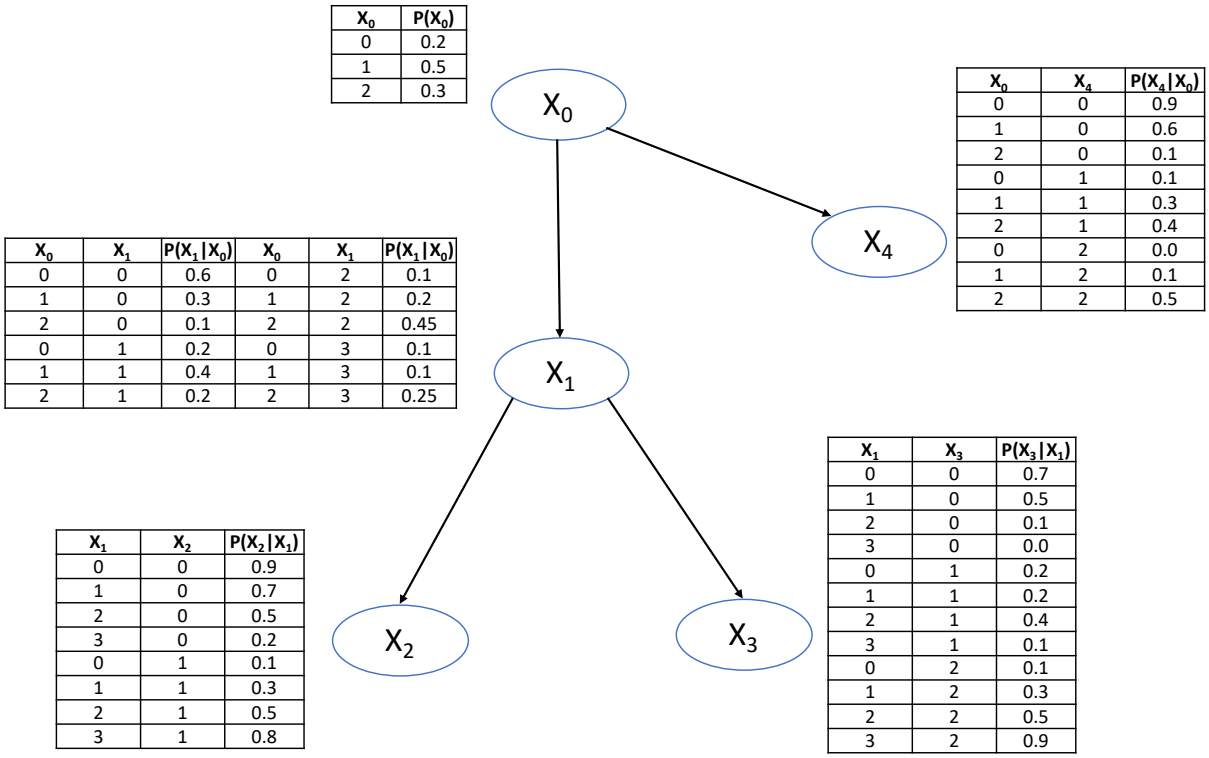
If you have any issues with the installation, please post them in the forum, so that other students or the instructors can help accordingly.

4. Test Cases

To help with your implementation, we have provided a few sample datasets and their expected outputs to help you debug your code. They can be found in the **data/** folder. You can find the code that loads the sample data and checks your program output in **test_lab1.py**.

We will mainly work with the Bayesian network stored in **graph_small.json**. It is shown below in Figure 1 for your convenience. In addition, we also provide a larger graphical model in **graph_large.json** for you to further test your code.

Note that during grading, we will run your code on hidden test cases on top of the two provided graphs.



X_0	$P(X_0)$
0	0.2
1	0.5
2	0.3

X_0	X_4	$P(X_4 \mid X_0)$
0	0	0.9
1	0	0.6
2	0	0.1
0	1	0.1
1	1	0.3
2	1	0.4
0	2	0.0
1	2	0.1
2	2	0.5

X_0	X_1	$P(X_1 \mid X_0)$	X_0	X_1	$P(X_1 \mid X_0)$
0	0	0.6	0	2	0.1
1	0	0.3	1	2	0.2
2	0	0.1	2	2	0.45
0	1	0.2	0	3	0.1
1	1	0.4	1	3	0.1
2	1	0.2	2	3	0.25

X_1	X_3	$P(X_3 \mid X_1)$
0	0	0.7
1	0	0.5
2	0	0.1
3	0	0.0
0	1	0.2
1	1	0.2
2	1	0.4
3	1	0.1
0	2	0.1
1	2	0.3
2	2	0.5
3	2	0.9

X_1	X_2	$P(X_2 \mid X_1)$
0	0	0.9
1	0	0.7
2	0	0.5
3	0	0.2
0	1	0.1
1	1	0.3
2	1	0.5
3	1	0.8

Figure 1: The Bayesian network you will perform inference on

5. Basic Factor Operations

As you recall from the lectures, we use factor tables to represent the conditional probability distributions of a Bayesian Network. You will now implement the core functionality for manipulating the factor tables.

- **`factor product()`:** This function should compute the product of two factors.
- **`factor_marginalize()`**: This function should sum over the indicated variable(s) and return the resulting factor.
- **`observe_evidence()`**: This function should take in a list of factors and the observed values of some of the variables, and modify the factors such that assignments not consistent with the observed values are set to zero.

All the factor operations make use of a custom `Factor` class, which you should familiarise yourself with. Its implementation can be found in **factor.py**, and a description on the factor datatype together with some example code can be found in **factor_readme.py**.

6. NAÏVE SUMMATION

To motivate the need for belief propagation, we will first implement the naïve summation algorithm. That is, you will compute the required (conditional) probabilities by computing the full joint distribution, then marginalising out irrelevant variables.

6.1 COMPUTING THE JOINT DISTRIBUTION

Complete the function **`compute_joint_distribution()`**, which computes the joint distribution over a Bayesian Network. Recall that for a Bayesian network over variables $X_1, \dots, X_n$, the joint distribution can be computed as[1]:

$$P(X_1, \dots, X_n) = \prod_i P\big(X_i \big| Parent(X_i)\big)$$

6.2 COMPUTING MARGINALS

Having computed the joint distribution, we can compute the marginal probabilities for a variable by marginalising out irrelevant variables from the joint distribution. In the event we have observed evidence, we will also need to reduce the joint distribution by setting the assignments inconsistent with the evidence to zero. So, complete **`compute_marginals_naive()`** with your implementation of the full naïve summation algorithm. In summary, there are three steps to perform:

- Compute the joint distribution over all variables
- Reduce the joint distribution by the evidence
- Marginalizing out irrelevant variables

After you perform all these operations, you will notice that the factor obtained is un-normalized, so be sure to normalize the final probability distribution.

If you implemented it correctly, for the small Bayesian network in Figure 1, you will get the following marginals for $P(X_0|X_3 = 1)$.

$\boldsymbol{X_0}$	$\boldsymbol{P(X_0|X_3 = 1)}$
0	0.178
1	0.486
2	0.336

[1] Generally, each node can have multiple parents, and the conditional probability should consider all parents. However, for the tree-like graphs considered in this assignment, nodes have at most one parent.

7. SUM-PRODUCT ALGORITHM

Notice that even for our small graph in Figure 1, the joint distribution has $3 \times 4 \times 2 \times 3 \times 3 = 216$ different assignments. This is very large and not scalable[2]! The standard variable elimination reduces this computational complexity but requires a separate run for every variable we want to compute the marginals for.

An efficient solution is to use the belief propagation (or sum-product) algorithm, which allows us to compute all single-node marginals for tree-like graphs in a single pass by "reusing" messages. Recall that the sum-product algorithm consists of the following phases:

- Apply evidence
- Send messages inward towards the root
- Send messages outward towards the leaves
- Compute the marginal of the relevant variables

The belief propagation algorithm will require you to traverse through the graph. For this purpose, we use a NetworkX graph. We have provided a function **`generate_graph_from_factors()`** that converts a list of factors into a NetworkX graph.

NetworkX provides useful functions for easy graph traversal. For example, neighbours of variable i can be accessed through:

```
>>> graph.neighbors(i)
```

For convenience, unary and pairwise factors will also be stored as node and edge attributes. To access the unary factor of variable i,

```
>>> graph.nodes[i]['factor']
```

Similarly, the pairwise factor between i and j can be accessed as,

```
>>> graph.edges[i, j]['factor']
```

Note that if a node does not have a unary factor, it will not contain the 'factor' field. You will have to handle the various scenarios yourself.

Complete the **`compute_marignals_bp()`** function, which computes the marginal probabilities of multiple variables using belief propagation. If implemented correctly, your inference should agree with **`compute_marginals_naive()`**.

Hint: It might be useful to create additional functions for this part. You can place those functions anywhere in the file.

[2] At your own risk, you can try performing using your naive summation code on the graph in **graph_large.json**.

8. MAP INFERENCE USING MAX-PRODUCT

In the final part you will write code to compute the Maximum a Posterior (MAP) configuration and its probability. To avoid underflow, we will be working in log-space for this assignment; otherwise, the probability of any single assignment in a large enough network is typically low enough to cause numerical issues. You should therefore take logarithm of all the probabilities before message passing, and perform all operations in log-space. That is, max-product is actually implemented using max-sum.

To facilitate this, first implement the relevant factor operations for log-space:

- **`factor_sum()`:** Analogous to factor_product(), but for log space.
- **`factor_max_marginalize()`**: This function should max over the indicated variable(s) and return the resulting factor. You should also keep track of the maximising values in the field `factor.val_argmax`.

Now, you are ready to complete the code for **`map_eliminate()`**. The function should take in a list of factors and optionally evidence, and compute the most probable joint assignment and its log probability.

If implemented correctly, the MAP conditioned on $X_3 = 1$ should be:

$$\text{argmax}_{X_0,X_1,X_2,X_4} P(X_0, X_1, X_2, X_4 | X_3 = 1) = \{X_0 = 0, X_1 = 0, X_2 = 0, X_4 = 0\},$$

with log probability $\log\big(P(X_0 = 0, X_1 = 0, X_2 = 0, X_4 = 0, X_3 = 1)\big) = -3.940$.

Notice that $X_0 = 0$ in the MAP assignment. This is despite $X_0 = 1$ maximising the conditional *marginal* probability, as you have computed in the previous sections.