Bayesian Network with Discrete Random Variables and its Python Implementation

Yuwei (Johnny) Meng

Department of Statistical Sciences, University of Toronto

Contents

1 Preface & Acknowledgements

One day before class, I was talking with Jae and Liam, two other students in STA410, about coming up with a cool name for our Python packages, and I was like "Oh man I hate coming up with names!" I do think that cool package names are good, and I admire people coming up with names like `PyTorch` and `Beautiful Soup` (when I first learned to use `Beautiful Soup` I was very confused how this is an HTML parsing library). But yeah, coming up with names doesn't work for me. Just `BayesNetwork`, or `BayesianNetwork`. It's cool. It's straightforward. No need to ponder on why `Beautiful Soup` is for HTML.

Bayesian Network, a variational model for exact inference. The first time I familiarized with this concept was in CSC384, and I thought that this model seems very versatile and powerful. The exposure to the D-Separation, Variable Elimination, and Viterbi algorithms in this course kept my interest in this model. Upon finishing CSC384, I never thought that I would encounter this concept again in CSC412, presented in a more theoretical and mathematical way. What's more, for the third time I was taught Hidden Markov Model in STA410, and I just can't throw this concept out of my mind now. Well, it's something that interests me, so I quickly chose this model to do as my STA410 project.

Here I would like to express my sincere gratitude to **Professor Alice Gao** in CSC384 for introducing Bayesian Network to me in a very entertaining way, to **Professor Denys Linkov/Murat Erdogdu** in CSC412/STA414 for incorporating theories of this model into the course, and definitely to **Professor Scott Schwartz** in STA410 for providing this project opportunity to me to have this model implemented in Python.

2 Installation & User Guide

The Python package can be installed by running the following command:

```
pip install bayes-network-bulldf
```

Upon installation, the following imports are available to use the functionalities developed in the package:

```
# Access the BayesNetwork class
from bayes_network_bulldf import BayesNetwork

# Read in a bayesian network from a text file
from bayes_network_bulldf import read_bayes_network_from_txt

# Variable elimination algorithm
from bayes_network_bulldf import variable_elimination

# Ancestral sampling algorithm
from bayes_network_bulldf import ancestral_sampling

# Access the HiddenMarkovModel class
from bayes_network_bulldf import HiddenMarkovModel

# Read in a Hidden Markov Model from a text file
from bayes_network_bulldf import read_hmm_from_txt

# Viterbi algorithm
from bayes_network_bulldf import viterbi

# Filtering via forward algorithm
from bayes_network_bulldf import filtering

# Smoothing via forward-backward algorithm
from bayes_network_bulldf import smoothing
```

3 Bayesian Network

3.1 Motivation

Consider a set of random variables $\mathbf{X} = (X_1, \cdots, X_n)$ where each $X_i \in \{x_i, \neg x_i\}$ is binary. To specify the joint distribution of $\mathbf{X}$, we need to list out all the possible combinations:

$$\begin{aligned}
&\mathbb{P}(x_1, x_2, \cdots, x_n);\\
&\mathbb{P}(\neg x_1, x_2, \cdots, x_n);\\
&\mathbb{P}(x_1, \neg x_2, \cdots, x_n);\\
&\vdots
\end{aligned}$$

For n random variables, the number of joint probabilities is $2^n - 1$, which quickly becomes intractable as n grows large, requiring a more efficient framework for modeling joint probabilities. Bayesian Network, or Directed Acyclic Graphical Model (DAGM), is one such framework exploiting *Theorem 1* below.

Theorem 1. (Chain Rule of Probability) Let $X_1, \cdots, X_n$ be random variables. Then the joint probability can be expressed as

$$\begin{aligned}
\mathbb{P}(X_1, \cdots, X_n) &= \mathbb{P}(X_1)\mathbb{P}(X_2, X_3, \cdots, X_n | X_1)\\
&= \mathbb{P}(X_1)\mathbb{P}(X_2|X_1)\mathbb{P}(X_3, \cdots, X_n|X_1, X_2)\\
&= \cdots\\
&= \mathbb{P}(X_1)\mathbb{P}(X_2|X_1)\mathbb{P}(X_3|X_1, X_2)\cdots\mathbb{P}(X_n|X_1, X_2, \cdots, X_{n-1})\\
&= \prod_{i=1}^{n}\mathbb{P}(X_i|X_1, X_2, \cdots, X_{i-1}).
\end{aligned}$$

■

Thus, if we have prior knowledge about the conditional independence relationships between the random variables we are modeling, then we can express the joint distribution in a more compact form, which motivates the use of Bayesian Networks.

3.2 Introduction to Bayesian Network

An example of Bayesian Network is shown in *Figure 1* below.

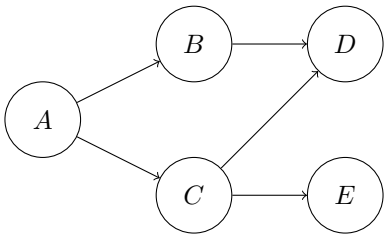

Figure 1: Bayesian Network Example

In this Bayesian Network, each vertex represents a random variable. The arrows denote conditional dependencies between random variables. For example, the edge $A \longrightarrow B$ implies that B is directly conditionally dependent on A. With this structure, we can model the joint distribution of these random variables in the following way:

Definition 1. Let $\mathbf{X} = (X_1, \cdots, X_n)$ be a set of random variables modeled by a Bayesian Network. Then the joint probability can be expressed as

$$\mathbb{P}(X_1, \cdots, X_n) = \prod_{i=1}^{n} \mathbb{P}(X_i | \text{ Parents}(X_i)).$$

■

Example 1. For the Bayesian Network shown in *Figure 1*, the joint probability can be expressed as

$$\mathbb{P}(A, B, C, D, E) = \mathbb{P}(A) \cdot \mathbb{P}(B|A) \cdot \mathbb{P}(C|A) \cdot \mathbb{P}(D|B, C) \cdot \mathbb{P}(E|C).$$

■

Note that *Definition 1* is a special case of *Theorem 1*, given that we have prior knowledge of the conditional independence relationships between the random variables. As a result, we can reduce the number of conditionals in the product and express the joint distribution in a more compact form. To illustrate this concept more clearly, let's define some probabilities for the Bayesian Network shown in *Figure 1*. These probabilities will be used throughout the rest of this document.

Example 2. Let $A, B, C, D, E \in \{0, 1\}$ be binary random variables with the following conditional probabilities:

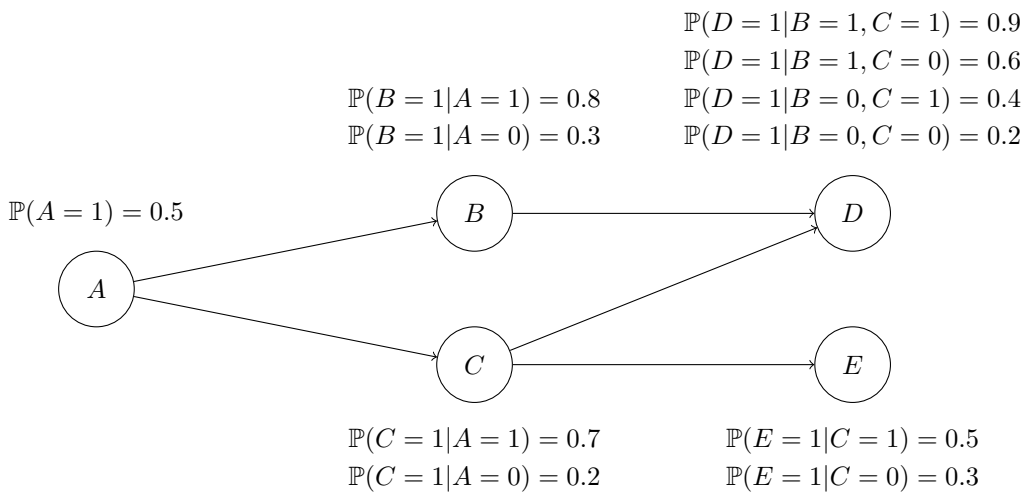

Figure 2: Bayesian Network with Probabilities Example

■

Note that only 11 conditional probabilities are needed to specify the joint distribution of the 5 random variables in our Bayesian Network, which is a significant improvement compared to the $2^5 - 1 = 31$ joint probabilities in the general case. The exact number of required conditional probabilities depends on the structure of the Bayesian Network, though, but as long as there are some conditional independence relationships between the random variables, we can always expect a reduction in the number of required conditional probabilities.

3.3 Implementation

The implementation of Bayesian Network in this project consists of multiple modules which are discussed separately in the following subsections.

3.3.1 `Distribution`

The `distribution.py` file defines an abstract class called `Distribution`, as well as two inherited classes named `UnconditionalDistribution` and `ConditionalDistribution`. The purpose of these classes is to provide a data structure that encodes the probability distribution of a vertex in the Bayesian Network.

The abstract class `Distribution` has the attribute and method definitions:

```
domain: set
```

This attribute defines the set of possible values for the random variables.

```
__call__(self, *args) -> float
```

This method evaluates the probability of the random variable taking on a specific value in the domain.

3.3.2 `UnconditionalDistribution`

The `UnconditionalDistribution` class defines the distribution of a random variable that does not have a parent in the Bayesian Network, as well as the distribution of a random variable conditioned on its parents. Besides the inherited attribute `domain`, it has the following attribute:

```
distribution: dict[Any, float]
```

This attribute serves as a lookup table for the unconditional distribution of the random variable. The keys of the dictionary are the possible values of the random variable, and the values are the corresponding probabilities.

```
__init__(self, domain: set, distribution: dict[Any, float]) -> None
```

This is the constructor of the class. It takes in the domain and the distribution as arguments. The constructor checks if the provided domain and distribution are valid. If `domain` does not match the keys of `distribution`, or the probabilities in `distribution` do not sum to 1, a `ValueError` will be raised.

This class also implements the `__call__` method. The expected argument is a value in the domain of the random variable, and the method outputs the probability of the random variable taking on that value. If the argument is not in the domain, a `ValueError` will be raised.

Example 3. For the Bayesian Network shown in *Figure 2*, the distribution of vertex A would have the following behaviors:

```
>>> dist_A = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.5, 1: 0.5})
>>> dist_A(0)
0.5
>>> dist_A(1)
0.5
```

■

3.3.3 `ConditionalDistribution`

This class defines the distribution of a random variable that has parents in the Bayesian Network. It inherits the `domain` attribute from the `Distribution` class, and it has the following attributes:

```
distributions[frozenset[str], UnconditionalDistribution]
```

The keys of this dictionary are `frozensets` of conditionals of the random variable, and the values define the distribution of the random variable conditioned on the values of its parents.

```
__init__(self,
         domain: set,
```

```
distributions: dict[frozenset[str], UnconditionalDistribution]) -> None
```

This constructor takes in the domain and the distributions as arguments. The constructor checks if the provided domain and distributions are valid. If `domain` does not match the domain of each `UnconditionalDistribution` object in `distribution`, a `ValueError` will be raised.

This class also implements the `__call__` method. The expected arguments are a value in the domain of the random variable and a dictionary of the values of its parents. The method outputs the probability of the random variable taking on that value, given the values of its parents. If the value is not in the domain, or if the parents are not in the dictionary, or if the conditioning values are not in the domain of the parents, a `ValueError` will be raised.

This is slightly confusing, so the following example illustrates this concept.

Example 4. For the Bayesian Network shown in *Figure 2*, the distribution of vertex D would have the following behaviors:

```
>>> dist_D_B1C1 = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.1, 1: 0.9})
>>> dist_D_B1C0 = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.4, 1: 0.6})
>>> dist_D_B0C1 = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.6, 1: 0.4})
>>> dist_D_B0C0 = UnconditionalDistribution(domain={0, 1}, distribution={0: 0.8, 1: 0.2})
>>> dist_D = ConditionalDistribution(domain={0, 1}, distributions={
...     frozenset({'B: 1', 'C: 1'}): dist_D_B1C1,
...     frozenset({'B: 1', 'C: 0'}): dist_D_B1C0,
...     frozenset({'B: 0', 'C: 1'}): dist_D_B0C1,
...     frozenset({'B: 0', 'C: 0'}): dist_D_B0C0
... })
>>> dist_D(1, {'B': 1, 'C': 0})
0.6
>>> dist_D(0, {'C': 0, 'B': 0})
0.8
```

■

The use of a `frozenset` data structure is a compromise to the need for a hashable data structure. The advantage is that we don't need to memorize the order of the parents when we are looking up the distribution because there is no inherent order for a `set` object. When looking up probabilities, the order in which we pass in the parents thus does not matter, as long as the values are correct, which ensures the efficiency of the lookup process.

3.3.4 `Vertex`

The `Vertex` class represents a vertex in the Bayesian Network. It has the following attributes:

```
name: str
```

This attribute defines the name of the vertex.

```
domain: set
```

This attribute defines the set of possible values for the random variable.

```
parents: dict[str, Self]
children: dict[str, Self]
```

These two attributes hold the parents and children of the vertex in the Bayesian Network. The keys of the dictionaries are the names of the vertices, and the values are the corresponding `Vertex` objects.

```
distribution: Optional[Distribution]
```

This attribute holds the distribution of the vertex. Depending on whether the vertex has parents or not, this attribute

can be either an `UnconditionalDistribution` or a `ConditionalDistribution` object.

The `Vertex` class also implements the following methods:

```
__init__(self, name: str, domain: set) -> None
```

This constructor takes in the name and domain of the vertex as arguments. It initializes the `parents`, `children`, and `distribution` attributes to empty dictionaries or `None`.

```
add_parent(self, parent: Self) -> None
add_child(self, child: Self) -> None
```

These two methods add a parent or child to the vertex. If the added vertex is already a parent or child of the vertex, a `ValueError` will be raised.

```
in_domain(self, value: Any) -> bool
```

This method returns `True` if the value is in the domain of the vertex, and `False` otherwise.

```
set_distribution(self, distribution: Distribution) -> None
```

This method sets the distribution of the vertex. This method contains many checks to ensure that `distribution` is valid. If `distribution` is not compatible with the vertex, a `ValueError` will be raised.

```
__call__(self, *args) -> float
```

This method basically calls the `__call__` method of the `distribution` object with the arguments `*args`. The expected arguments are the same as the `__call__` method of the `distribution` object and outputs a probability. If the `distribution` attribute for the vertex is `None`, a `ValueError` will be raised.

3.3.5 BayesNetwork

This class represents the Bayesian Network. It has the following attribute:

```
vertices: dict[str, Vertex]
```

This attribute holds a list of the vertices in the Bayesian Network. The keys of the dictionary are the names of the vertices, and the values are the corresponding `Vertex` objects.

The `BayesNetwork` class implements the following methods:

```
__init__(self) -> None
```

The constructor creates a `BayesNetwork` object and initializes the `vertices` attribute to an empty dictionary.

```
__len__(self) -> int
```

This method returns the number of vertices in the Bayesian Network.

```
add_node(self, vertex: Vertex) -> None
```

This method adds a vertex to the Bayesian Network. If the vertex is already in the network, a `ValueError` will be raised.

```
add_edge(self, parent: Vertex, child: Vertex) -> None
```

This method adds an edge from the parent vertex to the child vertex in the Bayesian Network. If either vertex is not in the network, a `ValueError` will be raised.

```
find_roots(self) -> list[Vertex]
```

This method returns a list of the root vertices in the Bayesian Network. A root vertex is defined as a vertex that does not have any parents in the network.

```
__call__(self, *args) -> float
```

This method evaluates the joint probability of the random variables in the Bayesian Network. The expected argument is a dictionary with the keys being the names of the random variables and the values being the corresponding values. The method outputs the joint probability of the random variables taking on those values. If values for one or more random variables are not provided, a `ValueError` will be raised. To compute the probability of a subset of the random variables, use the *Variable Elimination* algorithm discussed in *Chapter 4*

3.3.6 `read_bayes_network_from_txt`

The above description might be very confusing, so it is not recommended to create a Bayesian Network from scratch. Instead, I provide a function to read in a Bayesian Network from a text file. An example of this text file is provided in this `bn_ex.txt` file. The text file requires 3 parts, explained using the example file and *Figure 2*:

```
A: {0, 1}
B: {0, 1}
C: {0, 1}
D: {0, 1}
E: {0, 1}
```

This part defines the vertices in the Bayesian Network. Each line contains the name of the vertex and its domain, enclosed in curly braces. The vertices are separated by new lines.

```
A -> B
A -> C
B -> D
C -> D
C -> E
```

This part defines the edges in the Bayesian Network. Each line contains the name of the parent vertex and the name of the child vertex, separated by an arrow (`->`). The edges are separated by new lines.

```
P(A = 1) = 0.5
P(A = 0) = 0.5

P(B = 1 | A = 1) = 0.8
P(B = 0 | A = 1) = 0.2
P(B = 1 | A = 0) = 0.3
P(B = 0 | A = 0) = 0.7

P(D = 1 | B = 1, C = 1) = 0.9
P(D = 0 | B = 1, C = 1) = 0.1
P(D = 1 | B = 1, C = 0) = 0.6
P(D = 0 | B = 1, C = 0) = 0.4
P(D = 1 | B = 0, C = 1) = 0.4
P(D = 0 | B = 0, C = 1) = 0.6
P(D = 1 | B = 0, C = 0) = 0.2
P(D = 0 | B = 0, C = 0) = 0.8

...
```

This part defines the unconditional and conditional distributions of the vertices in the Bayesian Network. The conditional distributions are separated by new lines. Note that it is necessary to provide the probabilities for both `P(A = 1)` and `P(A = 0)` in the text file, even though the complement can be easily computed.

As long as the format is followed, the order in which the vertices, edges, and distributions are defined does not matter because the function uses regular expression to parse the text file. The function will create a `BayesNetwork` object and add the vertices, edges, and distributions to the object. The function then returns the `BayesNetwork` object. It is strongly recommended to use this function to create a Bayesian Network.

To wrap up this section, below is an example of how to use the function to create a Bayesian Network from a text file.

Example 5. Assume that the text file is named `bn_ex.txt` and is in the same directory as the Python file.

```
>>> bn = read_bayes_network_from_txt('bn_ex.txt')
>>> bn.find_roots()
[A: {0, 1}]
>>> len(bn)
5
```

Regarding joint probability computations:

$$\begin{aligned}\mathbb{P}(A=1,B=0,C=1,D=1,E=0) &= \mathbb{P}(A=1)\mathbb{P}(B=0|A=1)\mathbb{P}(C=1|A=1)\mathbb{P}(D=1|B=0,C=1)\\ &\quad\cdot\mathbb{P}(E=0|C=1)\\ &= 0.5\cdot 0.2\cdot 0.7\cdot 0.4\cdot 0.5\\ &= 0.014.\end{aligned}$$

The code output confirms this result:

```
>>> bn({'A': 1, 'B': 0, 'C': 1, 'D': 1, 'E': 0})
0.013999999999999999 # Numerical roundoff error
```

■

4 Variable Elimination

4.1 Motivation

In the previous section, we introduced how we could easily compute the joint probability of the random variables in a Bayesian Network. However, *Definition 1* must be used when the joint probability involves all random variables in the network and cannot be applied when we want to calculate the joint probability of a subset of the random variables or a conditional probability of the random variables. This motivates the use of the *Variable Elimination* algorithm.

Below is the example Bayesian Network again for reference:

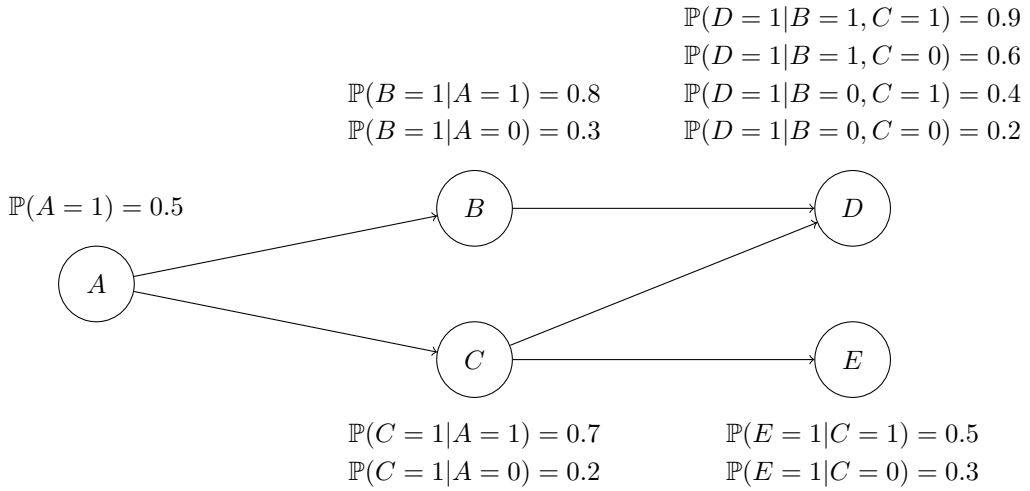

Figure 3: Bayesian Network with Probabilities Example

Some examples of probabilities we might want to compute are as follows:

$$\begin{aligned}&\mathbb{P}(A,B,C,D);\\&\mathbb{P}(B,C|D);\\&\mathbb{P}(B,E|A);\\&\mathbb{P}(A,B|C,D);\\&\mathbb{P}(E);\\&\vdots\end{aligned}$$

With the *Variable Elimination* algorithm, these probabilities can be computed easily.

4.2 Theory

The *Variable Elimination* algorithm is based on the following theorem:

Theorem 2. (Marginal Probability) Given discrete random variables X, Y, the marginal probabilities can be

computed as

$$\begin{aligned} p_X(x) &= \sum_y p_{XY}(x,y) \\ &= \sum_y p_{X|Y}(x|y)p_Y(y); \\ p_Y(y) &= \sum_x p_{XY}(x,y) \\ &= \sum_x p_{Y|X}(y|x)p_X(x). \end{aligned}$$

■

With *Theorem 2*, the *Variable Elimination* algorithm is as follows:

Theorem 3. (Variable Elimination Algorithm) Given a Bayesian Network with vertices $\mathbf{X} = \{X_1, \cdots, X_n\}$ and a desired probability $\mathbb{P}(Q|E)$ with $Q \subseteq \mathbf{X}$, $E \subseteq \mathbf{X}$, $Q \cap E = \varnothing$, define $H = \mathbf{X} \setminus Q \cup E$ as the hidden variables. Let D_X denote the domain of random variable X. Then

$$\mathbb{P}(Q|E) = \text{normalize}\left(\sum_{X_h \in H} \sum_{x \in D_{X_h}} \prod_{\psi \in \mathcal{F}} \psi(X_h|E)\right),$$

where the notation $\psi(X_h|E)$ denotes the "factor" that concerns the hidden variable X_h conditioned on E, and $\mathcal{F}$ defines the set of all the "factors". ■

Theorem 3 defines some new concepts such as *hidden variables* and *factors* that might be confusing, so I devote the following section to give an example of using the *Variable Elimination* algorithm to compute a probability.

4.3 Example

Given Bayesian Network *Figure 3*, suppose we want to compute conditional probability

$$\mathbb{P}(A, E|D = 0).$$

4.3.1 Step I – Creating Initial Factors

For each node in the Bayesian Network, we create a factor consisting of its probabilities defined by the network. Each factor has a scope denoting the set of random variables it concerns. For the Bayesian Network in *Figure 3*, we would create the following initial factors:

$$f_1(A) = \begin{pmatrix} A=1 & 0.5 \\ A=0 & 0.5 \end{pmatrix}$$

$$f_2(A,B) = \begin{pmatrix} A=1 & B=1 & 0.8 \\ A=1 & B=0 & 0.2 \\ A=0 & B=1 & 0.3 \\ A=0 & B=0 & 0.7 \end{pmatrix}$$

$$f_3(A,C) = \begin{pmatrix} A=1 & C=1 & 0.7 \\ A=1 & C=0 & 0.3 \\ A=0 & C=1 & 0.2 \\ A=0 & C=0 & 0.8 \end{pmatrix}$$

$$f_4(B,C,D) = \begin{pmatrix} B=1 & C=1 & D=1 & 0.9 \\ B=1 & C=1 & D=0 & 0.1 \\ B=1 & C=0 & D=1 & 0.6 \\ B=1 & C=0 & D=0 & 0.4 \\ B=0 & C=1 & D=1 & 0.4 \\ B=0 & C=1 & D=0 & 0.6 \\ B=0 & C=0 & D=1 & 0.2 \\ B=0 & C=0 & D=0 & 0.8 \end{pmatrix}$$

$$f_5(C,E) = \begin{pmatrix} C=1 & E=1 & 0.5 \\ C=1 & E=0 & 0.5 \\ C=0 & E=1 & 0.3 \\ C=0 & E=0 & 0.7 \end{pmatrix}$$

Note that whether the probabilities are conditional or not do not matter in the factors. The factors only care about the random variables they concern. As a reference, the set of factors we have after step 1 is

$$\mathcal{F}_1 = \{f_1, f_2, f_3, f_4, f_5\}.$$

4.3.2 Step II – Restricting Evidence

The evidence we have is $D = 0$, and so we can discard entries of D that are irrelevant. Specifically, f_4 turns into

$$f_6(B,C) = \begin{pmatrix} B=1 & C=1 & 0.1 \\ B=1 & C=0 & 0.4 \\ B=0 & C=1 & 0.6 \\ B=0 & C=0 & 0.8 \end{pmatrix}$$

Note that we can also remove D from the scope of f_4 because we have restricted it based on the evidence.

After step 2, the set of factors is

$$\mathcal{F}_2 = \{f_1, f_2, f_3, f_5, f_6\}.$$

Note that each factor is used only once and we can safely discard it afterwards.

4.3.3 Step III – Eliminating Hidden Variables

In our probability, A, E are the *Query* variables because we are interested in their joint probabilities. D is the *Evidence* variable because we have evidence of it. The remaining variables, B and C, are *Hidden* variables, and we will be removing them in this step.

The computational complexity of the *Variable Elimination* algorithm is $O(mk^{N_{\max}})$, where m is the number of initial factors, k is the maximum number of values in the domain of the variables, and $N_{\max}$ is the number of variables inside the largest sum. However, this complexity largely depends on the order in which we eliminate the hidden variables, and finding the optimal order of elimination is NP-hard. For a small Bayesian Network like the one in our case, the order doesn't matter much, so I will predetermine the order to be $B \to C$.

To eliminate B, we will gather all factors that concern the variable B, in our case they are f_2 and f_6. We will consider every valid combination of the variables and multiply the probability values if the conditions match. After

multiplying, we get

$$f_7(A,B,C) = \begin{pmatrix} A=1 & B=1 & C=1 & 0.8\cdot 0.1 = 0.08 \\ A=1 & B=1 & C=0 & 0.8\cdot 0.4 = 0.32 \\ A=1 & B=0 & C=1 & 0.2\cdot 0.6 = 0.12 \\ A=1 & B=0 & C=0 & 0.2\cdot 0.8 = 0.16 \\ A=0 & B=1 & C=1 & 0.3\cdot 0.1 = 0.03 \\ A=0 & B=1 & C=0 & 0.3\cdot 0.4 = 0.12 \\ A=0 & B=0 & C=1 & 0.7\cdot 0.6 = 0.42 \\ A=0 & B=0 & C=0 & 0.7\cdot 0.8 = 0.56 \end{pmatrix}$$

Next, we will sum out B and obtain

$$f_8(A,C) = \begin{pmatrix} A=1 & C=1 & 0.08+0.12 = 0.2 \\ A=1 & C=0 & 0.32+0.16 = 0.48 \\ A=0 & C=1 & 0.03+0.42 = 0.45 \\ A=0 & C=0 & 0.12+0.56 = 0.68 \end{pmatrix}$$

Now we have eliminated variable B. As a reference, the set of factors we have after this step is

$$\mathcal{F}_3 = \{f_1, f_3, f_5, f_8\}.$$

We will do the same thing for variable C. We gather factors f_3, f_5 and f_8 that concern C, and we multiply them together:

$$f_9(A,C,E) = \begin{pmatrix} A=1 & C=1 & E=1 & 0.7\cdot 0.5\cdot 0.2 = 0.07 \\ A=1 & C=1 & E=0 & 0.7\cdot 0.5\cdot 0.2 = 0.07 \\ A=1 & C=0 & E=1 & 0.3\cdot 0.3\cdot 0.48 = 0.0432 \\ A=1 & C=0 & E=0 & 0.3\cdot 0.7\cdot 0.48 = 0.1008 \\ A=0 & C=1 & E=1 & 0.2\cdot 0.5\cdot 0.45 = 0.045 \\ A=0 & C=1 & E=0 & 0.2\cdot 0.5\cdot 0.45 = 0.045 \\ A=0 & C=0 & E=1 & 0.8\cdot 0.3\cdot 0.68 = 0.1632 \\ A=0 & C=0 & E=0 & 0.8\cdot 0.7\cdot 0.68 = 0.3808 \end{pmatrix}$$

Then, we sum out C:

$$f_{10}(A,E) = \begin{pmatrix} A=1 & E=1 & 0.07+0.0432 = 0.1132 \\ A=1 & E=0 & 0.07+0.1008 = 0.1708 \\ A=0 & E=1 & 0.045+0.1632 = 0.2082 \\ A=0 & E=0 & 0.045+0.3808 = 0.4258 \end{pmatrix}$$

After this step, the remaining factors are

$$\mathcal{F}_4 = \{f_1, f_{10}\}.$$

4.3.4 Step IV – Combining and Normalization

At this step, all remaining factors should concern only our query variables. We will combine the remaining factors (f_1, f_{10}) by multiplying them like we did previously:

$$f_{11}(A,E) = \begin{pmatrix} A=1 & E=1 & 0.5\cdot 0.1132 = 0.0566 \\ A=1 & E=0 & 0.5\cdot 0.1708 = 0.0854 \\ A=0 & E=1 & 0.5\cdot 0.2082 = 0.1041 \\ A=0 & E=0 & 0.5\cdot 0.4258 = 0.2129 \end{pmatrix}$$

We are almost done. We just need to normalize the remaining values so that they follow a probability distribution. The current sum is $0.0566 + 0.0854 + 0.1041 + 0.2129 = 0.459$. We can normalize the values by dividing each value

by the sum, rounded to 3 decimal points:

$$\mathbb{P}(A, E|D = 0) = \begin{pmatrix} A = 1 & E = 1 & 0.0566/0.459 = 0.123 \\ A = 1 & E = 0 & 0.0854/0.459 = 0.186 \\ A = 0 & E = 1 & 0.1041/0.459 = 0.227 \\ A = 0 & E = 0 & 0.2129/0.459 = 0.464 \end{pmatrix}$$

4.4 Implementation

The *Variable Elimination* algorithm is implemented in the `variable_elimination.py` file. Each step mentioned above gets its own helper function, but the entire pipeline can be executed by calling the `variable_elimination` function. This function has the following signature:

```
variable_elimination(bn: BayesNetwork, query: set, evidence: dict={}) -> dict
```

Running code for the example illustrated above, we would get the outputs

```
>>> bn = read_bayes_network_from_txt('bn_ex.txt')
>>> variable_elimination(bn, query={'A', 'E'}, evidence={'D': 0})
{{'A: 1', 'E: 1'}: 0.12331154684095862,
 {'E: 1', 'A: 0'}: 0.22679738562091506,
 {'A: 1', 'E: 0'}: 0.18605664488017432,
 {'E: 0', 'A: 0'}: 0.46383442265795205}
```

which match our manual calculations above.

5 Ancestral Sampling

A Bayesian Network is nonetheless a model for a set of random variables, and thus we can sample from it. One way to sample from a Bayesian Network is *Ancestral Sampling*. The joint probability formula for a Bayesian Network defined in *Definition 1* suggests that we can sample each random variable based on the conditional distribution given its parents in a topological order, which also gives birth to the name *Ancestral Sampling*.

Example 6. Given the Bayesian Network in *Figure 2*, we can sample from it as follows:

1. Sample the root node A as a Bernoulli random variable with probability $\mathbb{P}(A)$;
2. Sample the next layer of nodes B and C conditioned on the sampled value of A using the conditional distributions $\mathbb{P}(B|A)$ and $\mathbb{P}(C|A)$;
3. Sample the final layer of nodes D and E conditioned on the sampled values of B and C using the conditional distributions $\mathbb{P}(D|B,C)$ and $\mathbb{P}(E|C)$;
4. The sampled values of A, B, C, D, E form a sample of the joint distribution of the random variables. Repeat this process to obtain more samples.

■

This method is implemented in the `ancestral_sampling.py` file. The function has the following signature:

```
ancestral_sampling(bn: BayesNetwork, n: int=1) -> Union[dict, list[dict]]
```

This function takes in a `BayesNetwork` instance and the number of samples n, whose default value is 1. The function returns a dictionary containing the sampled values of the random variables. If $n > 1$, the function returns a list of dictionaries, each representing a sample. Below is an example usage of the function:

```
>>> bn = read_bayes_network_from_txt('bn_ex.txt')
>>> ancestral_sampling(bn)
{'A': 1, 'B': 0, 'C': 1, 'D': 1, 'E': 0}
>>> ancestral_sampling(bn, n=5)
[{'A': 0, 'B': 0, 'C': 1, 'D': 1, 'E': 1},
 {'A': 0, 'B': 0, 'C': 0, 'D': 1, 'E': 1},
 {'A': 0, 'B': 0, 'C': 0, 'D': 0, 'E': 0},
 {'A': 1, 'B': 1, 'C': 0, 'D': 1, 'E': 0},
 {'A': 0, 'B': 0, 'C': 1, 'D': 0, 'E': 1}]
```

6 Hidden Markov Model (HMM)

6.1 Introduction to HMM

A *Hidden Markov Model*, or an HMM, is a special case of Bayesian Networks. An HMM has only two types of nodes: *Hidden* nodes and *Observation* nodes. In real-world applications, the hidden nodes represent states of a system that are not directly observable, while the observation nodes represent observable events that are generated by the hidden states. For $t \in \{0, 1, \cdots\}$, let Z_t denote the hidden state at time t and X_t denote the observation at time t. An HMM, as a Bayesian Network, has the following structure:

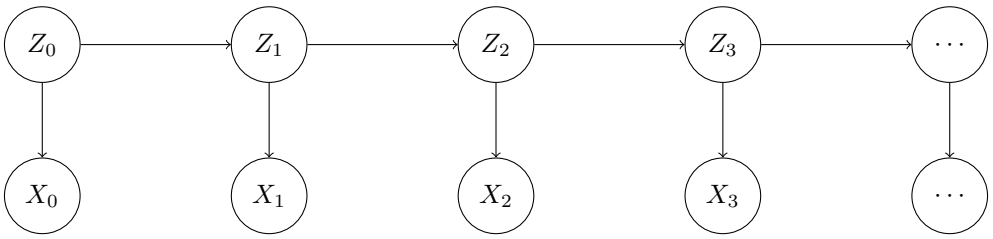

Figure 4: Hidden Markov Model Structure

Note that an HMM can have infinitely many nodes as $t \to \infty$. At each time step t, the hidden state Z_t generates an observation X_t, modeled by the conditional dependence $Z_t \longrightarrow X_t$. This implies that observation X_t only depends on the current hidden state Z_t, which is an important property of HMMs. Additionally, note the conditional dependence $Z_{t-1} \longrightarrow Z_t$, which implies that the hidden state at time t only depends on the hidden state at time $t-1$, and not on other hidden or observation states.

Given the HMM in *Figure 4*, from previous sections we know that we can define different distributions for the hidden states and observation states. However, since the time step t can approach ∞, defining the distributions for all hidden states and observation states separately is impractical. Therefore, we will make an assumption about our HMM here called *stationarity*:

Definition 2. (Stationarity of HMMs) An HMM is *stationary* if it satisfies the following conditions:

1. $\mathbb{P}(Z_t|Z_{t-1})$ is the same for every time step t;
2. $\mathbb{P}(X_t|Z_t)$ is the same for every time step t.

■

With the assumption defined in *Definition 2*, we only need to define the following three sets of distributions for an HMM, which is much easier to do:

Definition 3. A stationary HMM is specified by the following distributions:

1. **Initial Distribution** – $\pi(i) = \mathbb{P}(Z_0 = i)$. The probability that the initial hidden state is i.
2. **Transition Distribution** – $\psi(i, j) = \mathbb{P}(Z_t = j|Z_{t-1} = i)$. The probability that the hidden state at time t is j given that the hidden state at time $t-1$ is i.
3. **Emission Distribution** – $\phi(i, k) = \mathbb{P}(X_t = k|Z_t = i)$. The probability that the observation at time t is k given that the hidden state at time t is i.

■

Similar to *Figure 2* where we defined some probabilities for illustration, we will define the following HMM and use these probabilities for subsequent examples:

Example 7. Let $Z_t \in \{c, h\}$ denote whether the weather is *cold* or *hot* on day t. Let $X_t \in \{0, 1, 2\}$ denote the number of ice creams you eat on day t. We define the following probability distributions:

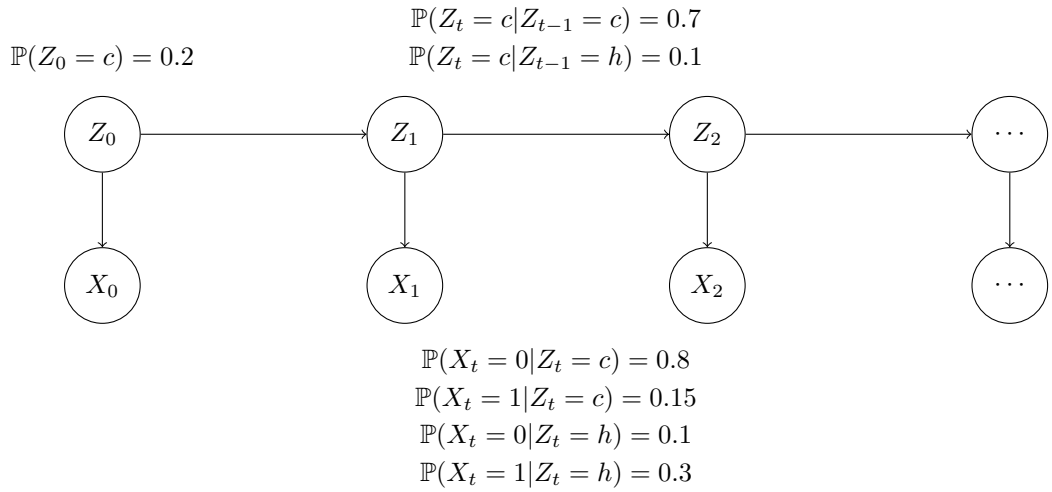

Figure 5: HMM with Probabilities Example

■

Besides computing the joint probabilities using *Definition 1* or the *Variable Elimination* algorithm on an HMM, given a sequence of observations $\vec{x} = [x_0, x_1, \cdots, x_T]$, there are 4 common types of tasks that we can perform with an HMM:

1. **Filtering** – $\mathbb{P}(Z_T|\vec{x})$. Compute the probability of the hidden state at *current* time T based on all observations. This can be done using the *Forward* algorithm discussed in *Chapter 8.2*;
2. **Smoothing** – $\mathbb{P}(Z_t|\vec{x})$ for $0 \leq t < T$. Compute the probability of the hidden state at *past* time t based on all observations. This can be done using the *Forward-Backward* algorithm discussed in *Chapter 8.3*;
3. **Prediction** – $\mathbb{P}(Z_t|\vec{x})$ for $t > T$. Compute the probability of the hidden state at *future* time t based on all observations. This is currently not implemented;
4. **Most Likely Sequence** – $\arg\max_{\vec{z}} \mathbb{P}(\vec{z}|\vec{x})$. Compute the most likely sequence of hidden states given the observations. This can be done using the *Viterbi* algorithm discussed in *Chapter 7*.

6.2 Implementation

6.2.1 `HiddenMarkovModel`

The `HiddenMarkovModel` class is implemented in the `hmm.py` file. This class extends the `BayesNetwork` class discussed in *Chapter 3.3*, because an HMM is essentially a special type of Bayesian Network. The class has the following attributes:

```
time_step: Optional[int]
```

This attribute stores the number of time steps for the HMM.

```
hidden_domain: Optional[set]
observation_domain: Optional[set]
```

These two attributes store the domains of the hidden and observation variables, respectively.

```
initial_distribution: Optional[UnconditionalDistribution]
```

```
transition_distribution: Optional[ConditionalDistribution]
emission_distribution: Optional[ConditionalDistribution]
```

These three attributes define the distributions of the HMM as specified in *Definition 3*.

In addition to the methods implemented in `BayesNetwork` as documented in *Chapter 3.3*, this class implements the following methods:

```
__init__(self) -> None
```

This constructor initializes a `HiddenMarkovModel` instance. It takes no arguments and initializes all the attributes to `None`.

```
set_time_step(self, time_step: int) -> None
set_hidden_domain(self, hidden_domain: set) -> None
set_observation_domain(self, observation_domain: set) -> None
set_initial_distribution(self, initial_distribution: UnconditionalDistribution) -> None
set_transition_distribution(self, transition_distribution: ConditionalDistribution) -> None
set_emission_distribution(self, emission_distribution: ConditionalDistribution) -> None
```

These methods basically set the corresponding attributes of the `HiddenMarkovModel` instance.

6.2.2 `read_hmm_from_txt`

Similar to `BayesNetwork`, initializing a `HiddenMarkovModel` instance from scratch is cumbersome. Therefore, a helper function named `read_hmm_from_txt` is provided. The example file for the HMM specified in *Figure 5* is in this `hmm_ex.txt` file. This file has 5 parts:

```
X: {0, 1, 2}
Z: {c, h}
```

This part specifies the domains of the observation and hidden variables, respectively.

```
T = 2
```

This part specifies the number of time steps for the HMM.

```
P(Z = c) = 0.2
P(Z = h) = 0.8
```

This part specifies the initial distribution of the HMM.

```
P(Z = c | Z = c) = 0.7
P(Z = h | Z = c) = 0.3
P(Z = c | Z = h) = 0.1
P(Z = h | Z = h) = 0.9
```

This part specifies the transition distribution of the HMM.

```
P(X = 0 | Z = c) = 0.8
P(X = 1 | Z = c) = 0.15
P(X = 2 | Z = c) = 0.05
P(X = 0 | Z = h) = 0.1
P(X = 1 | Z = h) = 0.3
P(X = 2 | Z = h) = 0.6
```

This part specifies the emission distribution of the HMM.

Note that it is important to follow the exact format above as the function uses regular expressions to parse the file. Given this function, we can initialize and work with an HMM in the following manner:

Example 8. Assume that the text file is named `hmm_ex.txt` and is in the same directory as the Python file.

```
>>> hmm = read_hmm_from_txt('hmm_ex.txt')
>>> hmm.find_roots()
[Z0: {'h', 'c'}]
>>> hmm.time_step
2
>>> hmm.hidden_domain
{'h', 'c'}
>>> hmm.observation_domain
{0, 1, 2}
>>> len(hmm)
6 # 3 hidden states and 3 observation states
```

Regarding joint probability computations:

$$\begin{aligned}\mathbb{P}(Z_0 = c, Z_1 = h, Z_2 = h, X_0 = 0, X_1 = 1, X_2 = 2) &= \mathbb{P}(Z_0 = c) \cdot \mathbb{P}(Z_1 = h|Z_0 = c) \cdot \mathbb{P}(Z_2 = h|Z_1 = h) \\ &\quad \cdot \mathbb{P}(X_0 = 0|Z_0 = c) \cdot \mathbb{P}(X_1 = 1|Z_1 = h) \cdot \mathbb{P}(X_2 = 2|Z_2 = h) \\ &= 0.2 \cdot 0.3 \cdot 0.9 \cdot 0.8 \cdot 0.3 \cdot 0.6 \\ &= 0.007776\end{aligned}$$

The code output confirms this result:

```
>>> hmm({'Z0': 'c', 'X0': 0, 'Z1': 'h', 'X1': 1, 'Z2': 'h', 'X2': 2})
0.007776
```

■

7 The Viterbi Algorithm

7.1 Motivation

Suppose we have logged the number of ice creams we have eaten through out the summer in our diary, and later we want to recall what the weather was like on each day of the summer. Since we have no recorded information on the weather, our only hope is to recover the weather information based on the numbers of ice creams we have eaten. This motivates the *Viterbi* algorithm, which computes the most likely sequence of hidden states based on a given sequence of observations.

The *Viterbi* algorithm uses dynamic programming in a way that future states are computed based on past states. We will illustrate this process in the following section. Below is the example HMM again for reference.

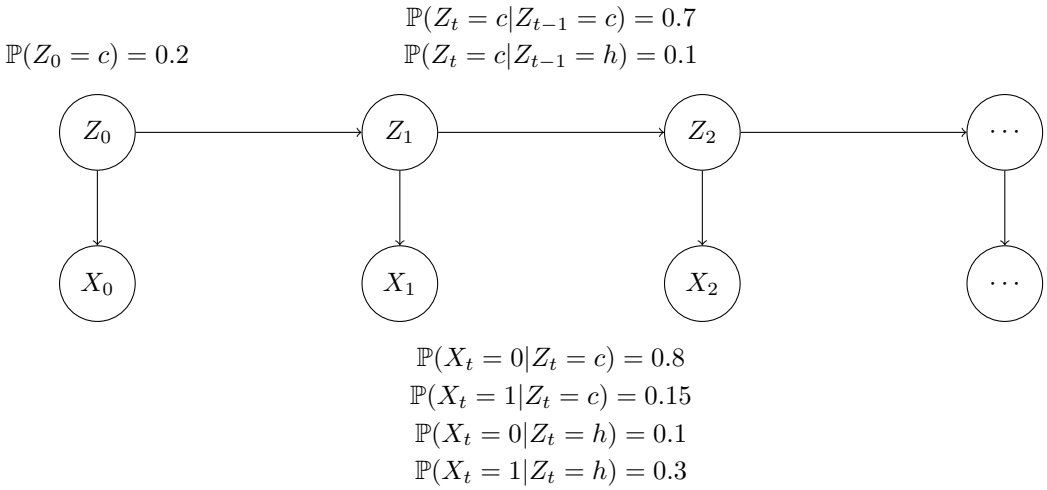

Figure 6: HMM with Probabilities Example

7.2 Example

Suppose on a 4-day interval, we have eaten the following numbers of ice creams:

```
X = [0, 1, 2, 1]
```

We will consider the numbers sequentially.

7.2.1 Day 0

On day 0, the number of ice cream we have eaten is 0, and we would like to compute the most likely hidden state on day 0. This motivates the use of *Bayes' Rule*:

$$\begin{aligned} z_0 &= \arg\max_z \mathbb{P}(Z_0 = z|X_0 = 0) \\ &= \arg\max_z \frac{\mathbb{P}(X_0 = 0|Z_0 = z)\mathbb{P}(Z_0 = z)}{\mathbb{P}(X_0 = 0)} \\ &= \arg\max_z \mathbb{P}(X_0 = 0|Z_0 = z)\mathbb{P}(Z_0 = z). \end{aligned}$$

This separates into 2 cases:

$$\mathbb{P}(X_0 = 0|Z_0 = c)\mathbb{P}(Z_0 = c) = 0.8 \cdot 0.2 = 0.16; \qquad \mathbb{P}(X_0 = 0|Z_0 = h)\mathbb{P}(Z_0 = h) = 0.1 \cdot 0.8 = 0.08.$$

Therefore, it is more likely that $z_0 = c$ than $z_0 = h$.

7.2.2 Day 1

On day 1, we have eaten 1 ice cream. Note that the probability of interest at this time step is the following, given the conditional independence relationships in the HMM:

$$\begin{aligned}\mathbb{P}(Z_0, Z_1|X_0 = 0, X_1 = 1) &= \frac{\mathbb{P}(X_0 = 0, X_1 = 1, Z_0, Z_1)}{\mathbb{P}(X_0 = 0, X_1 = 1)} \\ &\propto \mathbb{P}(X_0 = 0, X_1 = 1, Z_0, Z_1) \\ &= \mathbb{P}(Z_1, X_0 = 0, X_1 = 1|Z_0)\mathbb{P}(Z_0) \\ &= \mathbb{P}(Z_0)\mathbb{P}(X_0 = 0, X_1 = 0|Z_0, Z_1)\mathbb{P}(Z_1|Z_0) \\ &= \mathbb{P}(Z_0)\mathbb{P}(X_0 = 0|Z_0)\mathbb{P}(Z_1|Z_0)\mathbb{P}(X_1 = 1|Z_1).\end{aligned}$$

In the last line of the above equation, note that the first two terms come from the previous time step $t = 0$, the third term is the transition probability, and the last term is the emission probability. Since the first two terms do not contain Z_1, we should always choose the most likely hidden state from the previous time step, suggesting a dynamic programming approach. With $Z_0 = c$, we can compute day 1 as follows:

$$\begin{aligned}\mathbb{P}(z_0, Z_1 = c|X_0 = 0, X_1 = 1) &\propto \mathbb{P}(z_0)\mathbb{P}(X_0 = 0|z_0)\mathbb{P}(Z_1 = c|z_0)\mathbb{P}(X_1 = 1|Z_1 = c) \\ &= 0.16 \cdot 0.7 \cdot 0.15 \\ &= 0.0168; \\ \mathbb{P}(z_0, Z_1 = c|X_0 = 0, X_1 = 1) &\propto \mathbb{P}(z_0)\mathbb{P}(X_0 = 0|z_0)\mathbb{P}(Z_1 = h|z_0)\mathbb{P}(X_1 = 1|Z_1 = h) \\ &= 0.16 \cdot 0.3 \cdot 0.3 \\ &= 0.0144.\end{aligned}$$

Therefore, it is more likely that $z_1 = c$ than $z_1 = h$.

7.2.3 Day 2

We will use the same technique to compute the probability of day 2:

$$\begin{aligned}\mathbb{P}(z_0, z_1, Z_2 = c|X_0 = 0, X_1 = 1, X_2 = 2) &\propto \mathbb{P}(z_0, z_1|x_0, x_1)\mathbb{P}(Z_2 = c|z_1)\mathbb{P}(x_2|Z_2 = c) \\ &= 0.0168 \cdot 0.7 \cdot 0.05 \\ &= 0.000588; \\ \mathbb{P}(z_0, z_1, Z_2 = h|X_0 = 0, X_1 = 1, X_2 = 2) &\propto \mathbb{P}(z_0, z_1|x_0, x_1)\mathbb{P}(Z_2 = h|z_1)\mathbb{P}(x_2|Z_2 = h) \\ &= 0.0168 \cdot 0.3 \cdot 0.6 \\ &= 0.003024.\end{aligned}$$

Therefore, it is more likely that $z_2 = h$ than $z_2 = c$.

7.2.4 Day 3

We will use the same technique to compute the probability of day 3:

$$\begin{aligned}\mathbb{P}(z_0, z_1, z_2, Z_3 = c|X_0 = 0, X_1 = 1, X_2 = 2, X_3 = 1) &\propto \mathbb{P}(z_0, z_1, z_2|x_0, x_1, x_2)\mathbb{P}(Z_3 = c|z_2)\mathbb{P}(x_3|Z_3 = c) \\ &= 0.003024 \cdot 0.1 \cdot 0.15 \\ &= 0.00004536; \\ \mathbb{P}(z_0, z_1, z_2, Z_3 = h|X_0 = 0, X_1 = 1, X_2 = 2, X_3 = 1) &\propto \mathbb{P}(z_0, z_1, z_2|x_0, x_1, x_2)\mathbb{P}(Z_3 = h|z_2)\mathbb{P}(x_3|Z_3 = h) \\ &= 0.003024 \cdot 0.9 \cdot 0.3 \\ &= 0.00081648.\end{aligned}$$

Therefore, it is more likely that $z_3 = h$ than $z_3 = c$.

7.2.5 Conclusion

Overall, given the observations

```
X = [0, 1, 2, 1]
```

the most likely sequence of hidden states computed using the *Viterbi* algorithm is

```
Z = [c, c, h, h]
```

7.3 Implementation

The *Viterbi* algorithm is implemented in the `viterbi.py` file. The algorithm is split into two parts: `viterbi_base` and `viterbi_step`. The main function `viterbi` uses the split functions to execute the algorithm and compute the most likely sequence of hidden states. The main function has the following signature:

```
viterbi(hmm: HiddenMarkovModel, observations: list) -> list
```

Running code for the example illustrated above, we would get the outputs

```
>>> hmm = read_hmm_from_txt('hmm_ex.txt')
>>> viterbi(hmm, [0, 1, 2, 1])
['c', 'c', 'h', 'h']
```

which match our manual calculations above.

7.4 Notes

7.4.1 Normalization

From our manual calculations above in *Chapter 7.2*, we notice that the computed probabilities got exponentially smaller as the number of time steps increased. This quickly leads to numerical stability and underflow issues. To prevent this, the Python implementation normalizes the returned probabilities at each time step so that the probabilities stay at a manageable level.

7.4.2 `HiddenMarkovModel.time_step`

Note that the `HiddenMarkovModel` has an attribute called `time_step` that is supposed to specify the number of states in the HMM. This attribute only matters when computing the joint probabilities of the variables in the HMM using the `__call__` method or the *Variable Elimination* algorithm. The *Viterbi* algorithm only uses the `initial_distribution`, `transition_distribution`, and `emission_distribution` attributes of the HMM, and thus the `viterbi` function described above can compute the most likely sequence of hidden states of any length under the given HMM setup. The length of the returned sequence is solely determined by the length of the input observations.

8 The Forward-Backward Algorithm

8.1 Motivation

In the previous section where we explored the *Viterbi* algorithm, the problem we were interested in solving was to find the most likely sequence of hidden states given a sequence of observations. In this section, we explore the case where we don't care the entire sequence of hidden states, but only the probability distribution of the hidden state at a certain time step t. As mentioned in *Chapter 6.1*, the following types of tasks are generally of interest to us:

1. **Filtering** – $\mathbb{P}(Z_T|\vec{x})$. Compute the probability of the hidden state at *current* time T based on all observations;
2. **Smoothing** – $\mathbb{P}(Z_t|\vec{x})$ for $0 \leq t < T$. Compute the probability of the hidden state at *past* time t based on all observations;
3. **Prediction** – $\mathbb{P}(Z_t|\vec{x})$ for $t > T$. Compute the probability of the hidden state at *future* time t based on all observations;
4. **Most Likely Sequence** – $\arg\max_{\vec{z}} \mathbb{P}(\vec{z}|\vec{x})$. Compute the most likely sequence of hidden states given the observations.

In this chapter, we will explore algorithms that solve problems 1 and 2. Below is the example HMM again for reference.

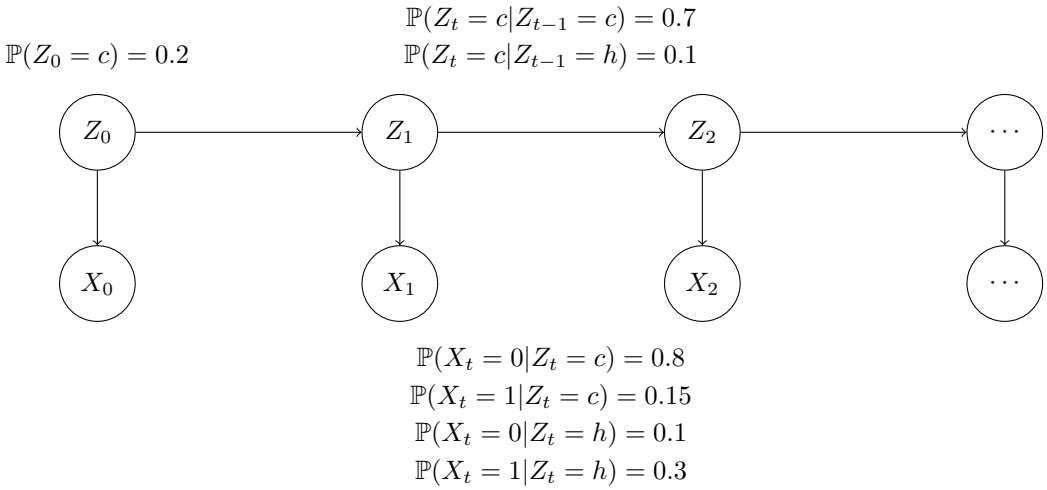

Figure 7: HMM with Probabilities Example

8.2 The Forward Algorithm

8.2.1 Theory

Suppose we have observations

```
X = [x0 = 0, x1 = 1, x2 = 2, x3 = 1, x4 = 1, x5 = 2, x6 = 0]
```

Since we have 7 observations from x_0 to x_6, we define $T = 6$ as the last time step of the observations. For $t \in \{0, 1, \cdots, T\}$, we define the *filtered marginal* α_t as

$$\alpha_t(j) = \mathbb{P}(Z_t = j|x_{0:t}).$$

In other words, $\alpha_t(j)$ is the probability that the hidden state at time t is j given its current and preceding observations.

Note that for the base case $t = 0$, we have

$$\begin{aligned}\alpha_0(j) &= \mathbb{P}(Z_0 = j|x_0) \\ &= \frac{\mathbb{P}(x_0|Z_0 = j)\mathbb{P}(Z_0 = j)}{\mathbb{P}(x_0)} \\ &\propto \mathbb{P}(x_0|Z_0 = j)\mathbb{P}(Z_0 = j) \\ &= \pi(j)\phi(j, x_0),\end{aligned}$$

which is the product of the initial probability and the emission probability of the HMM.

Now, the *Forward* algorithm comes into play to compute the filtered marginals α_t for $t > 0$. This algorithm has two steps: A *Prediction* step and a *Update* step. For $t > 0$, by the *Law of Total Probability*, the prediction step runs as follows:

$$\begin{aligned}\tau_t(j) &= \mathbb{P}(Z_t = j|x_{0:t-1}) \\ &= \sum_i \mathbb{P}(Z_t = j|Z_{t-1} = i, x_{0:t-1})\mathbb{P}(Z_{t-1} = i|x_{0:t-1}) \\ &= \sum_i \mathbb{P}(Z_t = j|Z_{t-1} = i)\mathbb{P}(Z_{t-1} = i|x_{0:t-1}) \\ &= \sum_i \psi(i, j)\alpha_{t-1}(i).\end{aligned}$$

With prediction τ_t, we can update the filtered marginal α_t as follows:

$$\begin{aligned}\alpha_t(j) &= \mathbb{P}(Z_t = j|x_{0:t}) \\ &= \mathbb{P}(Z_t = j|x_{0:t-1}, x_t) \\ &\propto \mathbb{P}(x_t|Z_t = j, x_{0:t-1})\mathbb{P}(Z_t = j|x_{0:t-1}) \\ &= \mathbb{P}(x_t|Z_t = j)\mathbb{P}(Z_t = j|x_{0:t-1}) \\ &= \phi(j, x_t)\tau_t(j).\end{aligned}$$

Therefore, with the above recurrence relation, we can compute the filtered marginals α_t at any time step t for $t \in \{0, 1, \cdots, T\}$ given observations $x_{0:T}$. Now, notice our objective probability for the *Filtering* task

$$\mathbb{P}(Z_T|\vec{x}) = \mathbb{P}(Z_T|x_{0:T}) = \alpha_T,$$

which is exactly the filtered marginal at the last time step T. Therefore, the *Filtering* task can be computed by a pass of the *Forward* algorithm.

8.2.2 Implementation

The *Forward* algorithm is implemented in the `forward_backward.py` file in the `forward` function, which has the following signature:

```
forward(hmm: HiddenMarkovModel, observations: list, t: int=0) -> dict
```

This function takes in a `HiddenMarkovModel` instance, a list of observations, and a time step t, and it outputs the filtered marginal α_t at time step t. To run the *Filtering* task, a function called `filtering` is provided that has the following signature:

```
filtering(hmm: HiddenMarkovModel, observations: list) -> dict
```

This function basically calls `forward` with $t = T$, where T is the length of the observations minus 1.

With the HMM defined in *Figure 7*, we can run code as follows:

```
>>> hmm = read_hmm_from_txt('hmm_ex.txt')
>>> filtering(hmm, [0, 1, 2, 1, 1, 2, 0])
{'h': 0.5070962425796098, 'c': 0.4929037574203903}
```

The outputs tell us that it is almost equally likely that day 6 is either hot or cold based on the observations.

8.3 The Forward-Backward Algorithm

8.3.1 Theory

In the previous section, we discussed how the *Forward* algorithm computes the *forward* probabilities α_t given the current and preceding observations. We can do the same for the *backward* direction by defining the *backward* filtered marginal β_t as follows:

$$\beta_t(j) = \mathbb{P}(x_{t+1:T}|Z_t = j).$$

In other words, $\beta_t(j)$ is the probability of observing $x_{t+1:T}$ given that the hidden state at time t is j. For the base case $t = T$, we assume $\beta_T(j) = 1$ for all j. Then we can see the recurrence as follows with the *Law of Total Probability*:

$$\begin{aligned}
\beta_t(j) &= \mathbb{P}(x_{t+1:T}|Z_t = j) \\
&= \mathbb{P}(x_{t+1}, x_{t+2:T}|Z_t = j) \\
&= \sum_i \mathbb{P}(x_{t+1}, x_{t+2:T}|Z_t = j, Z_{t+1} = i)\mathbb{P}(Z_{t+1} = i|Z_t = j) \\
&= \sum_i \mathbb{P}(x_{t+1}|Z_{t+1} = i)\mathbb{P}(x_{t+2:T}|Z_{t+1} = i)\mathbb{P}(Z_{t+1} = i|Z_t = j) \\
&= \sum_i \phi(i, x_{t+1})\beta_{t+1}(i)\psi(j, i).
\end{aligned}$$

Now that we have both the *forward* probabilities α_t and the *backward* probabilities β_t, we can start considering the objective probability for the *Smoothing* task, for some $t \in \{0, 1, \cdots, T-1\}$:

$$\begin{aligned}
\mathbb{P}(Z_t|\vec{x}) &= \mathbb{P}(Z_t|x_{0:T}) \\
&= \frac{\mathbb{P}(Z_t, x_{0:T})}{\mathbb{P}(x_{0:T})} \\
&\propto \mathbb{P}(Z_t, x_{0:T}) \\
&= \mathbb{P}(Z_t, x_{0:t})\mathbb{P}(x_{t+1:T}|Z_t, x_{0:t}) \\
&= \mathbb{P}(Z_t, x_{0:t})\mathbb{P}(x_{t+1:T}|Z_t) \\
&\propto \mathbb{P}(Z_t|x_{0:t})\mathbb{P}(x_{t+1:T}|Z_t) \\
&= \alpha_t\beta_t.
\end{aligned}$$

Overall, the *Smoothing* task can be solved with a forward pass to compute the forward marginal α_t and a backward pass to compute the backward marginal β_t. The final output would then be the product of the two marginals.

8.3.2 Implementation

The *Forward-Backward* algorithm has the following function signature:

```
forward_backward(hmm: HiddenMarkovModel, observations: list, t: int=0) -> dict
```

This function computes and returns the normalized product of the forward and backward marginals at time step t. To run the *Smoothing* task, a function called `smoothing` is provided that has the following signature:

```
smoothing(hmm: HiddenMarkovModel, observations: list, t: int) -> dict
```

This function basically calls the `forward_backward` function to compute the probability of hidden state Z_t given the observations. With the HMM defined in *Figure 7*, we can run code as follows:

```
>>> hmm = read_hmm_from_txt('hmm_ex.txt')
>>> smoothing(hmm, [0, 1, 2, 1, 1, 2, 0], 2)
{'c': 0.018344806972238406, 'h': 0.9816551930277615}
>>> smoothing(hmm, [0, 1, 2, 1, 1, 2, 0], 5)
{'c': 0.04663014310969049, 'h': 0.9533698568903095}
```

Note that similar to the implementation of the *Viterbi* algorithm, the *Forward-Backward* algorithm also does not use the `time_step` attribute of the `HiddenMarkovModel` instance. The algorithm would get the variable T from the length of the input observations. If the input t to the function is not compatible with T (i.e. calling `smoothing` with $t \geq T$), the function will raise an `ValueError`.

8.4 Connection with Variable Elimination

Note that in both *Smoothing* and *Filtering* tasks, we are interested in computing a probability of a random variable in a Bayesian Network conditioned on some other random variables in the network, and from *Chapter 4* we know that the *Variable Elimination* algorithm is exactly designed for this purpose. Therefore, we would expect to get the same probability outputs for the same sequence of observations. Given the same HMM setup as in *Figure 7* but with `T = 7` stored in a file called `hmm_ex_t7.txt`, we would get the outputs

```
>>> hmm = read_hmm_from_txt('hmm_ex_t7.txt')

>>> filtering(hmm, [0, 1, 2, 1, 1, 2, 0])
{'h': 0.5070962425796098, 'c': 0.4929037574203903}
>>> variable_elimination(
...     hmm,
...     {'Z6'},
...     {'X0': 0, 'X1': 1, 'X2': 2, 'X3': 1, 'X4': 1, 'X5': 2, 'X6': 0}
... )
{{'Z6: c'}: 0.4929037574203903, {'Z6: h'}: 0.5070962425796098}

>>> smoothing(hmm, [0, 1, 2, 1, 1, 2, 0], 2)
{'c': 0.018344806972238406, 'h': 0.9816551930277615}
>>> variable_elimination(
...     hmm,
...     {'Z2'},
...     {'X0': 0, 'X1': 1, 'X2': 2, 'X3': 1, 'X4': 1, 'X5': 2, 'X6': 0}
... )
{{'Z2: h'}: 0.9816551930277616, {'Z2: c'}: 0.018344806972238416}

>>> smoothing(hmm, [0, 1, 2, 1, 1, 2, 0], 5)
{'c': 0.04663014310969049, 'h': 0.9533698568903095}
>>> variable_elimination(
...     hmm,
...     {'Z5'},
...     {'X0': 0, 'X1': 1, 'X2': 2, 'X3': 1, 'X4': 1, 'X5': 2, 'X6': 0}
... )
{{'Z5: h'}: 0.9533698568903095, {'Z5: c'}: 0.04663014310969049}
```

So, if the *Variable Elimination* algorithm does the same job and is more general, why do we need to implement the *Forward-Backward* algorithm? The answer is that the *Forward-Backward* algorithm is more efficient than the

Variable Elimination algorithm on an HMM. The following code illustrate this:

```
>>> import time
>>> trials = 100

>>> fb_total_time = 0
>>> for _ in range(trials):
...     start_time = time.time()
...     filtering(hmm, [0, 1, 2, 1, 1, 2, 0])
...     fb_total_time += time.time() - start_time

>>> fb_total_time / trials
2.892017364501953e-05

>>> vea_total_time = 0
>>> for _ in range(trials):
...     start_time = time.time()
...     variable_elimination(
...         hmm,
...         {'Z6'},
...         {'X0': 0, 'X1': 1, 'X2': 2, 'X3': 1, 'X4': 1, 'X5': 2, 'X6': 0}
...     )
...     vea_total_time += time.time() - start_time

>>> vea_total_time / trials
0.00014547586441040038
```

For `T = 7` we don't see much difference, but the difference would become much more significant for larger T.