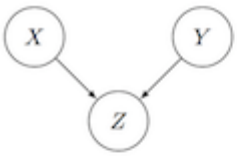
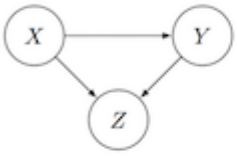
In [ ]:
```
import math  # Just ignore this :-)
```

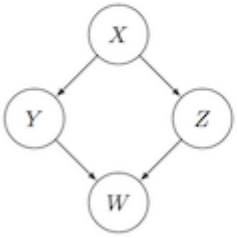
CTiB E2023 - Week 11 - Exercises

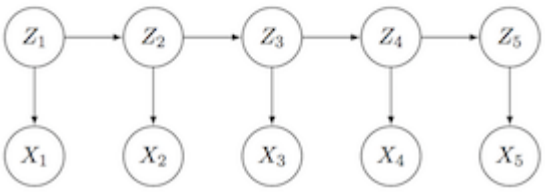
Theoretical exercises

Exercise 1: From graph to joint probability. As explained in the section *Conditional Probabilites and Dependency Graphs* on p296-298 in [ICT] a a dependency graph is a graphical notation to describe the dependency relationships when specifying a joint probility. For the following four graphs, write down the joint probabilty of the random variables.









A: $P(X) \cdot P(Y) \cdot P(Z|X,Y)$

B: $P(X) \cdot P(Y|X) \cdot P(Z|Y,X)$

C: $P(X) \cdot P(Y|X) \cdot P(Z|X) \cdot P(W|Y,Z)$

D: $p(Z_1) \cdot p(X_1 \mid Z_1) \cdot \prod_{i=2}^{5} p(Z_i \mid Z_{i-1}) \cdot p(X_i \mid Z_i)$

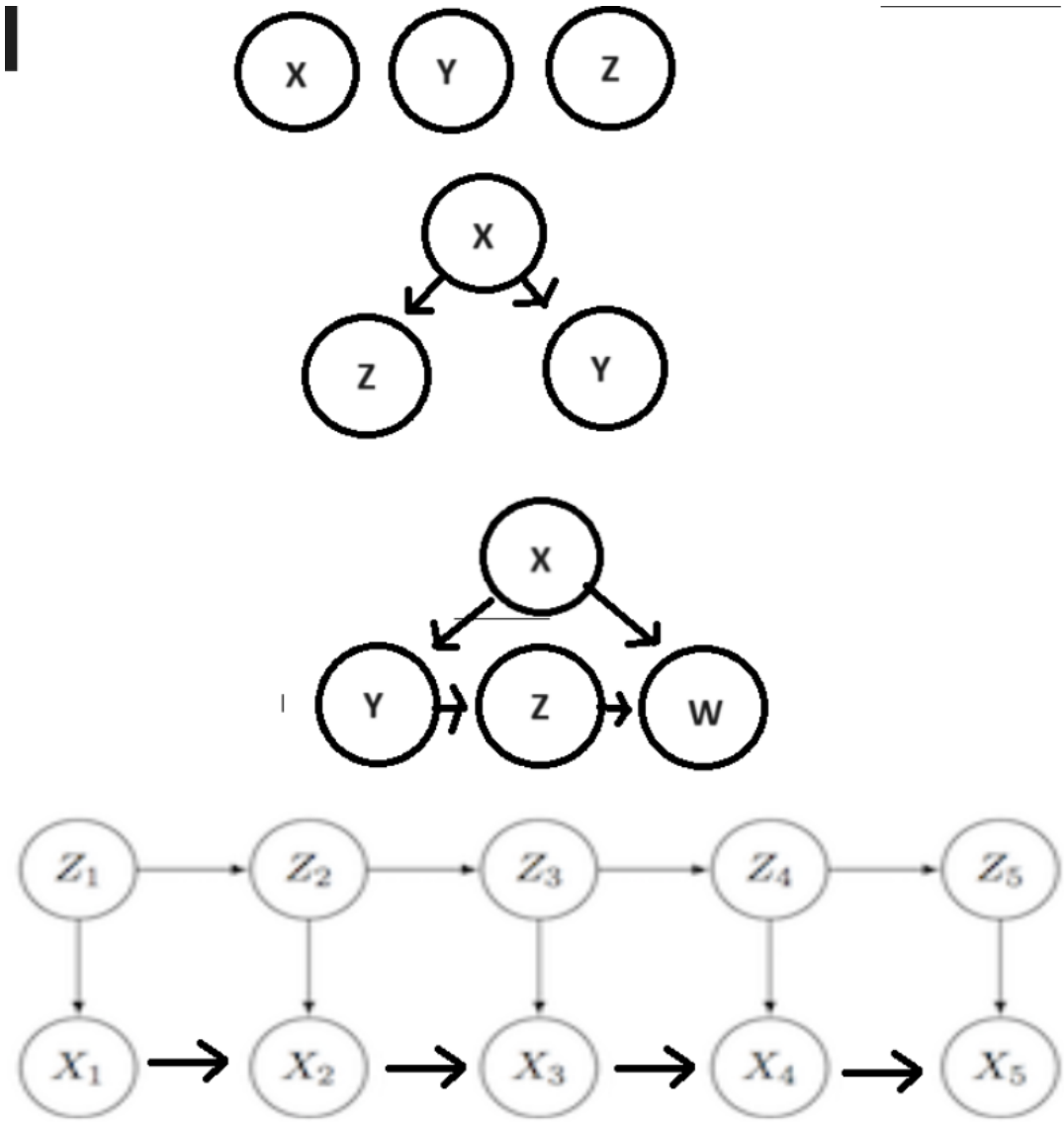
Exercise 2: From joint probability to graph. Draw the following four joint probabilities as dependency graphs:

$p(X)p(Y)p(Z)$

$p(X)p(Y \mid X)p(Z \mid X)$

$p(X)p(Y \mid X)p(Z \mid Y)p(W \mid X, Z)$

$p(Z_1)p(X_1 \mid Z_1) \prod_{i=2}^{5} p(X_i \mid Z_i, X_{i-1}) \prod_{i=2}^{5} p(Z_i \mid Z_{i-1})$



Exercise 3: How much time does it take to compute the joint probability $P(\mathbf{X}, \mathbf{Z}|\Theta)$ in terms of N and K, where $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$, $\mathbf{Z} = \mathbf{z}_1, \ldots, \mathbf{z}_N$, and K is the number of hidden states in the hidden Markov model Θ, using the formula on slide 10 from the lecture on Nov 13?

Since we go through the markov chain sequentially and perform the same calculation for each position in the sequence the running time is $O(N)$. If you need to calculate the probability of all possible paths you have to make K^N calculations, so the running time would be $O(N \cdot K^N)$

Practical exercises

In the exercise below, you will see an example of how a hidden Markov model (HMM) can be represented, and you will implement and experiment with the computation of

the joint probability as explained in the lecture on Nov 13 (week 11).

1 - Representing an HMM

We can represent a HMM as a triple consisting of three matrices: a $K \times 1$ matrix with the initial state probabilities, a $K \times K$ matrix with the transition probabilities and a $K \times |\Sigma|$ matrix with the emission probabilities. In Python we can write the matrices like this:

In [ ]:
```
init_probs_7_state = [0.00, 0.00, 0.00, 1.00, 0.00, 0.00, 0.00]

trans_probs_7_state = [
    [0.00, 0.00, 0.90, 0.10, 0.00, 0.00, 0.00],
    [1.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00],
    [0.00, 1.00, 0.00, 0.00, 0.00, 0.00, 0.00],
    [0.00, 0.00, 0.05, 0.90, 0.05, 0.00, 0.00],
    [0.00, 0.00, 0.00, 0.00, 0.00, 1.00, 0.00],
    [0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 1.00],
    [0.00, 0.00, 0.00, 0.10, 0.90, 0.00, 0.00],
]

emission_probs_7_state = [
    #   A     C     G     T
    [0.30, 0.25, 0.25, 0.20],
    [0.20, 0.35, 0.15, 0.30],
    [0.40, 0.15, 0.20, 0.25],
    [0.25, 0.25, 0.25, 0.25],
    [0.20, 0.40, 0.30, 0.10],
    [0.30, 0.20, 0.30, 0.20],
    [0.15, 0.30, 0.20, 0.35],
]
```

How do we use these matrices? Remember that we are given some sequence of observations, e.g. like this:

In [ ]:
```
obs_example = 'GTTTCCCAGTGTATATCGAGGGATACTACGTGCATAGTAACATCGGCCAA'
```

To make a lookup in our three matrices, it is convenient to translate each symbol in the string to an index.

In [ ]:
```
def translate_observations_to_indices(obs):
    mapping = {'a': 0, 'c': 1, 'g': 2, 't': 3}
    return [mapping[symbol.lower()] for symbol in obs]
```

Let's try to translate the example above using this function:

In [ ]:
```
obs_example_trans = translate_observations_to_indices(obs_example)
```

In [ ]:
```
obs_example_trans
```

Out[ ]:
```
[2,
 3,
 3,
 3,
 1,
 1,
 1,
 0,
 2,
 3,
 2,
 3,
 0,
 3,
 0,
 3,
 1,
 2,
 0,
 2,
 2,
 2,
 0,
 3,
 0,
 1,
 3,
 0,
 1,
 2,
 3,
 2,
 1,
 0,
 3,
 0,
 2,
 3,
 0,
 0,
 1,
 0,
 3,
 1,
 2,
 2,
 1,
 1,
 0,
 0]
```

Use the function below to translate the indices back to observations:

In [ ]:
```
def translate_indices_to_observations(indices):
    mapping = ['a', 'c', 'g', 't']
    return ''.join(mapping[idx] for idx in indices)
```

In [ ]:
```
translate_indices_to_observations(translate_observations_to_indices(obs_e
```

Out[ ]: 'gtttcccagtgtatatcgagggatactacgtgcatagtaacatcggccaa'

Now each symbol has been transformed (predictably) into a number which makes it much easier to make lookups in our matrices. We'll do the same thing for a list of states (a path):

In [ ]:
```
def translate_path_to_indices(path):
    return list(map(lambda x: int(x), path))

def translate_indices_to_path(indices):
    return ''.join([str(i) for i in indices])
```

Given a path through an HMM, we can now translate it to a list of indices:

In [ ]:
```
path_example = '33333333333321021021021021021021021021021021021'

translate_path_to_indices(path_example)
```

Out[ ]:
```
[3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 3,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1,
 0,
 2,
 1]
```

Finally, we can collect the three matrices in a class to make it easier to work with our HMM.

In [ ]:
```
class hmm:
    def __init__(self, init_probs, trans_probs, emission_probs):
        self.init_probs = init_probs
        self.trans_probs = trans_probs
        self.emission_probs = emission_probs
```

```
# Collect the matrices in a class.
hmm_7_state = hmm(init_probs_7_state, trans_probs_7_state, emission_probs

# We can now reach the different matrices by their names. E.g.:
hmm_7_state.trans_probs
```

Out[ ]:
```
[[0.0, 0.0, 0.9, 0.1, 0.0, 0.0, 0.0],
 [1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0],
 [0.0, 0.0, 0.05, 0.9, 0.05, 0.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0],
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0],
 [0.0, 0.0, 0.0, 0.1, 0.9, 0.0, 0.0]]
```

For testing, here's another model (which we will refer to as the 3-state model).

In [ ]:
```
init_probs_3_state = [0.10, 0.80, 0.10]

trans_probs_3_state = [
    [0.90, 0.10, 0.00],
    [0.05, 0.90, 0.05],
    [0.00, 0.10, 0.90],
]

emission_probs_3_state = [
    #   A     C     G     T
    [0.40, 0.15, 0.20, 0.25],
    [0.25, 0.25, 0.25, 0.25],
    [0.20, 0.40, 0.30, 0.10],
]

hmm_3_state = hmm(init_probs_3_state, trans_probs_3_state, emission_probs
```

2 - Validating an HMM (and handling floats)

Before using the model we'll write a function to validate that the model is valid. That is, the matrices should have the right dimensions and the following things should be true:

1. The initial probabilities must sum to 1.
2. Each row in the matrix of transition probabilities must sum to 1.
3. Each row in the matrix of emission probabilities must sum to 1.
4. All numbers should be between 0 and 1, inclusive.

Write a function `validate_hmm` that given a model returns True if the model is valid, and False otherwise:

In [ ]:
```
def validate_hmm(model):

    if abs(sum(model.init_probs)-1) > 0.0001:
        return False
```

```
    for row in model.trans_probs:
        if abs(sum(row)-1) > 0.0001:
            return False

    for row in model.emission_probs:
        if abs(sum(row)-1) > 0.0001:
            return False

    return True
```

We can now use this function to check whether the example model is a valid model.

In [ ]:
```
validate_hmm(hmm_7_state)
```

You might run into problems related to summing floating point numbers because summing floating point numbers does not (always) give the expected result as illustrated by the following examples. How do you suggest to deal with this?

In [ ]:
```
0.15 + 0.30 + 0.20 + 0.35
```

Out[ ]:
```
0.9999999999999999
```

The order of the terms matter.

In [ ]:
```
0.20 + 0.35 + 0.15 + 0.30
```

Out[ ]:
```
1.0
```

Because it changes the prefix sums

In [ ]:
```
0.15 + 0.30
```

Out[ ]:
```
0.44999999999999996
```

In [ ]:
```
0.20 + 0.35 + 0.15
```

Out[ ]:
```
0.7000000000000001
```

In [ ]:
```
0.15 + 0.30
```

Out[ ]:
```
0.44999999999999996
```

On should never compare floating point numbers. They represent only an 'approximation'. Read more about the 'problems' in 'What Every Computer Scientist Should Know About Floating-Point Arithmetic' at:

http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

3 - Computing the Joint Probability

Recall that the joint probability $p(\mathbf{X}, \mathbf{Z}) = p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N)$ of a hidden Markov model (HMM) can be compute as

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N) = p(\mathbf{z}_1) \left[\prod_{n=2}^{N} p(\mathbf{z}_n \mid \mathbf{z}_{n-1}) \right] \prod_{n=1}^{N} p(\mathbf{x}_n \mid \mathbf{z}_n)$$

Implementing without log-transformation

Write a function `joint_prob` given a model (e.g. in the representation above) and sequence of observables, $\mathbf{X}$, and a sequence of hidden states, $\mathbf{Z}$, computes the joint probability cf. the above formula.

In [ ]:
```
def joint_prob(model, x, z):
    acc_prob = None

    if len(x) > 0:
        acc_prob = model.init_probs[z[0]] * model.emission_probs[z[0]][x[

        for i in range(1, len(x)):
            acc_prob *= model.trans_probs[z[i-1]][z[i]] * model.emission_

    return acc_prob
```

Now compute the joint probability of the $\mathbf{X}$ (`x_short`) and $\mathbf{Z}$ (`z_short`) below using the 7-state (`hmm_7_state`) model introduced above. (*Remember to translate them first using the appropriate functions introduces above!*)

In [ ]:
```
x_short = 'GTTTCCCAGTGTATATCGAGGGATACTACGTGCATAGTAACATCGGCCAA'
z_short = '33333333333321021021021021021021021021021021021021'

x = translate_observations_to_indices(x_short)
z = translate_path_to_indices(z_short)

joint_prob(hmm_7_state, x, z)
```

Out[ ]: 1.9114255184318882e-31

Implementing with log-transformation (i.e. in "log-space")

Now implement the joint probability function using log-transformation as explained in the lecture. We've given you a log-function that handles $\log(0)$.

In [ ]:
```
def log(x):
    if x == 0:
        return float('-inf')
    return math.log(x)

def joint_prob_log(model, x, z):
    acc_prob = None

    if len(x) > 0:
        acc_prob = log(model.init_probs[z[0]]) + log(model.emission_probs
```

```
        for i in range(1, len(x)):
            acc_prob += log(model.trans_probs[z[i-1]][z[i]]) + log(model.

    return acc_prob
```

Confirm that the log-transformed function is correct by comparing the output of `joint_prob_log` to the output of `joint_prob`.

In [ ]:
```
x_short = 'GTTTCCCAGTGTATATCGAGGGATACTACGTGCATAGTAACATCGGCCAA'
z_short = '333333333333210210210210210210210210210210210210210'

x = translate_observations_to_indices(x_short)
z = translate_path_to_indices(z_short)

print(log(joint_prob(hmm_7_state, x, z)))
joint_prob_log(hmm_7_state, x, z)
```

```
-70.73228857440486
```

Out[ ]:
```
-70.73228857440486
```

Comparison of Implementations

Now that you have two ways to compute the joint probability given a model, a sequence of observations, and a sequence of hidden states, try to make an experiment to figure out how long a sequence can be before it becomes essential to use the log-transformed version. For this experiment we'll use two longer sequences.

In [ ]:
```
x_long = 'TGAGTATCACTTAGGTCTATGTCTAGTCGTCTTTCGTAATGTTTGGTCTTGTCACCAGTTATC
z_long = '333332102102102102102102102102102102102102102102102102102102102
```

Now compute the joint probability with `joint_prob` the 7-state (hmm_7_state) model introduced above, and see when it breaks (i.e. when it wrongfully becomes 0). Does this make sense? Here's some code to get you started.

In the cell below you should state for which i computing the joint probability (of the first i symbols of `x_long` and `z_long`) using `joint_prob` wrongfully becomes 0.

In [ ]:
```
for i in range(1, len(x_long), 1):
    x = x_long[:i]
    z = z_long[:i]

    x_trans = translate_observations_to_indices(x)
    z_trans = translate_path_to_indices(z)

    if i % 100 == 0 or i > 520:
        print(joint_prob(hmm_7_state, x_trans, z_trans))

    if joint_prob(hmm_7_state, x_trans, z_trans) == 0:
        print(i)
        break
```

```
1.8619524290102183e-65
1.617577499700581e-122
3.067543059784318e-183
4.860704144303008e-247
5.2587243422067765e-306
5.5202e-319
1.93204e-319
1.7386e-320
3.48e-321
1.215e-321
3.26e-322
1e-322
3e-323
1e-323
0.0
530
```

Your answer here:

For the 7-state model, `joint_prob` (of the first i symbols of `x_long` and `z_long`) becomes 0 for **i = 530**.

Since a floating point number is saved with a specific amount of bits (usually 64-bit or 32-bit blocks) there is a limit to how small a number the computer can save. If you get a number below this the computer just rounds to 0. Here we could get the probability up until the 529nd position in the markov chain ($1e-323$), after which the computer can no longer represent such a small number so it rounds to 0. If you look at the probabilities calculated as i approaches 530 you see that there are less and less significants.