Problem 2

In this Jupyter notebook, we visualize how HMMs work. This visualization corresponds to problem 2 in set 6.

Assuming your HMM module is complete and saved at the correct location, you can simply run all cells in the notebook without modification.

In [37]:
```
import os
import numpy as np
from IPython.display import HTML

#from HMM import unsupervised_HMM
from HMM_helper import (
    text_to_wordcloud,
    states_to_wordclouds,
    parse_observations,
    sample_sentence,
    visualize_sparsities,
    animate_emission
)
```

For some reason, my HMM wouldn't import the updated version of my code so I copy-pasted it below. I removed a lot of the red method descriptions because my code wouldn't fit in otherwise.

In [ ]:
```
class HiddenMarkovModel:
    '''
    Class implementation of Hidden Markov Models.
    '''

    def __init__(self, A, O):
        self.L = len(A)
        self.D = len(O[0])
        self.A = A
        self.O = O
        self.A_start = [1. / self.L for _ in range(self.L)]


    def viterbi(self, x):
        M = len(x)      # Length of sequence.

        # The (i, j)^th elements of probs and seqs are the max probability
        # of the prefix of length i ending in state j and the prefix
        # that gives this probability, respectively.
        #
        # For instance, probs[1][0] is the probability of the prefix of
        # length 1 ending in state 0.
        probs = [[0. for _ in range(self.L)] for _ in range(M + 1)]
        seqs = [['' for _ in range(self.L)] for _ in range(M + 1)]
        t = []
        # initialize row 2 of probs and seqs as 0
        for i in range(self.L):
            probs[1][i] = self.A_start[i] * self.O[i][x[0]]

        # start w row 2 and use top-down approach
        for row in range(2, M+1):
            for col1 in range(self.L):
                temp = []
                for col2 in range(self.L):
                    temp.append(probs[row-1][col2] * self.O[col1][x[row-1]] *
                                self.A[col2] [col1])
                max_index, max_value = max(enumerate(temp), key=operator.itemgette
r(1))
                probs[row][col1] = max_value
                seqs[row][col1] = seqs[row-1][max_index] + str(max_index)

        max_index, max_value = max(enumerate(probs[M]), key=operator.itemgetter
(1))
        max_seq = seqs[len(probs)-1][max_index] + str(max_index)

        return max_seq
```

In [ ]:
```
    def forward(self, x, normalize=False):
        M = len(x)      # len of sequence

        if normalize:
            alphas = [[0. for i in range(self.L)] for j in range(M + 1)]

            # initialize row of alphas
            for i in range(self.L):
                alphas[1][i] = self.O[i][x[0]] * self.A_start[i]

            for row in range(2, M+1):
                for col in range(self.L):
                    temp = []
                    for col2 in range(self.L):
                        temp.append(alphas[row-1][col2] * self.O[col][x[row-1]] *
self.A[col2] [col])
                    alphas[row][col] = sum(temp)
                norm = sum(alphas[row])
                for nCol in range(self.L):
                    alphas[row][nCol] /= norm
            return alphas

        alphas = [[0. for i in range(self.L)] for j in range(M + 1)]

        # initialize first row of alphas
        for i in range(self.L):
            alphas[1][i] = self.O[i][x[0]] * self.A_start[i]

        for row in range(2, M+1):
            for col in range(self.L):
                temp = []
                for col2 in range(self.L):
                    temp.append(alphas[row-1][col2] * self.O[col][x[row-1]] *
                            self.A[col2] [col])
                alphas[row][col] = sum(temp)
        return alphas
```

In [ ]:
```
    def backward(self, x, normalize=False):
        M = len(x)      # len of sequence
        if normalize:
            betas = [[0. for i in range(self.L)] for j in range(M + 1)]

            for i in range(self.L):
                betas[M][i] = 1

            for row in range(M-1, 0, -1):
                for col in range(self.L):
                    for col2 in range(self.L):
                        betas[row][col] += (betas[row+1][col2] *
                                            self.O[col2][x[row]] * self.A[col] [co
l2])
                norm = sum(betas[row])
                for nCol in range(self.L):
                    betas[row][nCol] /= norm
            return betas

        betas = [[0. for i in range(self.L)] for j in range(M + 1)]

        for i in range(self.L):
            betas[M][i] = 1

        for row in range(M-1,0, -1):
            for col in range(self.L):
                for col2 in range(self.L):
                    betas[row][col] += (betas[row+1][col2] * self.O[col2][x[row]]
*
                                    self.A[col] [col2])
        return betas
```

In [ ]:
```
    def count_transitions(self, a, b, X, Y):
        '''
        y_i^j = b
        and y_i^(j-1) = a
        '''
        den = 0
        num = 0

        for i in range(len(X)):
            for j in range(1, len(X[i])):
                if Y[i][j-1] == a:
                    den += 1
                    if Y[i][j] == b:
                        num += 1

        return num, den

    def count_observations(self, w, a, X, Y):
        '''
        y_i^j = w
        and x_i^j = w
        '''
        den = 0
        num = 0

        for i in range(len(X)):
            for j in range(len(X[i])):
                if Y[i][j] == a:
                    den += 1
                    if X[i][j] == w:
                        num += 1
        return num, den
```

In [ ]:
```
    def supervised_learning(self, X, Y):
        for a in range(self.L):
            for b in range(self.L):
                num, den = self.count_transitions(a, b, X, Y)
                self.A[a][b] = num / den

        for a in range(len(self.O)):
            for w in range(len(self.O[0])):
                num, den = self.count_observations(w, a, X, Y)
                self.O[a][w] = num / den

        pass
```

In [ ]:
```
def unsupervised_learning(self, X, N_iters):
        for x in range(N_iters):
            A_den = [0. for i in range(self.L)]
            O_den = [0. for i in range(self.L)]

            A_num = [[0. for i in range(self.L)] for i in range(self.L)]
            O_num = [[0. for i in range(self.D)] for i in range(self.L)]

            for sequence in X:
                M = len(sequence)

                alpha = self.forward(sequence, normalize=True)
                beta = self.backward(sequence, normalize=True)

                for w in range(1, M+1):
                    temp = [0. for i in range(self.L)]
                    for z in range(self.L):
                        temp[z] = alpha[w][z] * beta[w][z]

                    # normalize
                    norm = sum(temp)

                    for z in range(len(temp)):
                        temp[z] /= norm

                    for z in range(self.L):
                        if w != M:
                            A_den[z] += temp[z]
                        O_num[z][sequence[w-1]] += temp[z]
                        O_den[z] += temp[z]

                for w in range(1, M):
                    norm = 0

                    temp = [[0. for _ in range(self.L)] for _ in range(self.L)]

                    for i in range(self.L):
                        for j in range(self.L):
                            temp[i][j] = alpha[w][i] * self.A[i][j]
                            * self.O[j][sequence[w]] * beta[w+1][j]

                    for row in temp:
                        norm += sum(row)

                    for i in range(self.L):
                        for j in range(self.L):
                            temp[i][j] /= norm

                    for i in range(self.L):
                        for j in range(self.L):
                            A_num[i][j] += temp[i][j]

            for i in range(self.L):
                for j in range(self.L):
                    self.A[i][j] = A_num[i][j] / A_den[i]

            for i in range(self.L):
                for j in range(self.D):
                    self.O[i][j] = O_num[i][j] / O_den[i]
```

In [ ]:
```
    def generate_emission(self, M):
        emission = []
        states = []
        state = random.choice(range(self.L))

        for _ in range(M):
            states.append(state)

            t = self.A[state]
            o = self.O[state]

            emission.append(int(np.random.choice(range(self.D), 1, p=o)))

            state = int(np.random.choice(range(self.L), 1, p=t))

        return emission, states


    def probability_alphas(self, x):
        # Calculate alpha vectors.
        alphas = self.forward(x)

        # alpha_j(M) gives the probability that the state sequence ends
        # in j. Summing this value over all possible states j gives the
        # total probability of x paired with any state sequence, i.e.
        # the probability of x.
        prob = sum(alphas[-1])
        return prob


    def probability_betas(self, x):
        betas = self.backward(x)

        # beta_j(1) gives the probability that the state sequence starts
        # with j. Summing this, multiplied by the starting transition
        # probability and the observation probability, over all states
        # gives the total probability of x paired with any state
        # sequence, i.e. the probability of x.
        prob = sum([betas[1][j] * self.A_start[j] * self.O[j][x[0]] \
                    for j in range(self.L)])

        return prob
```

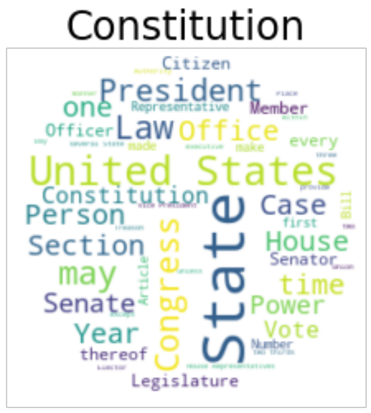
In [ ]:
```
def supervised_HMM(X, Y):
    # Make a set of observations.
    observations = set()
    for x in X:
        observations |= set(x)

    # Make a set of states.
    states = set()
    for y in Y:
        states |= set(y)

    # Compute L and D.
    L = len(states)
    D = len(observations)

    # Randomly initialize and normalize matrix A.
    A = [[random.random() for i in range(L)] for j in range(L)]

    for i in range(len(A)):
        norm = sum(A[i])
        for j in range(len(A[i])):
            A[i][j] /= norm

    # Randomly initialize and normalize matrix O.
    O = [[random.random() for i in range(D)] for j in range(L)]

    for i in range(len(O)):
        norm = sum(O[i])
        for j in range(len(O[i])):
            O[i][j] /= norm

    # Train an HMM with labeled data.
    HMM = HiddenMarkovModel(A, O)
    HMM.supervised_learning(X, Y)

    return HMM

def unsupervised_HMM(X, n_states, N_iters):
    # Make a set of observations.
    observations = set()
    for x in X:
        observations |= set(x)

    # Compute L and D.
    L = n_states
    D = len(observations)

    # Randomly initialize and normalize matrix A.
    random.seed(2020)
    A = [[random.random() for i in range(L)] for j in range(L)]

    for i in range(len(A)):
        norm = sum(A[i])
        for j in range(len(A[i])):
            A[i][j] /= norm

    # Randomly initialize and normalize matrix O.
    random.seed(155)
    O = [[random.random() for i in range(D)] for j in range(L)]

    for i in range(len(O)):
        norm = sum(O[i])
        for j in range(len(O[i])):
            O[i][j] /= norm
```

```
In [ ]:
```

Visualization of the dataset

We will be using the Constitution as our dataset. First, we visualize the entirety of the Constitution as a wordcloud:

```
In [40]: text = open(os.path.join(os.getcwd(), 'data/constitution.txt')).read()
         wordcloud = text_to_wordcloud(text, title='Constitution')
```



Training an HMM

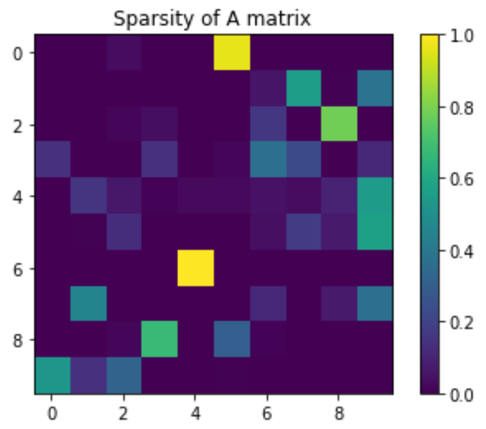
Now we train an HMM on our dataset. We use 10 hidden states and train over 100 iterations:

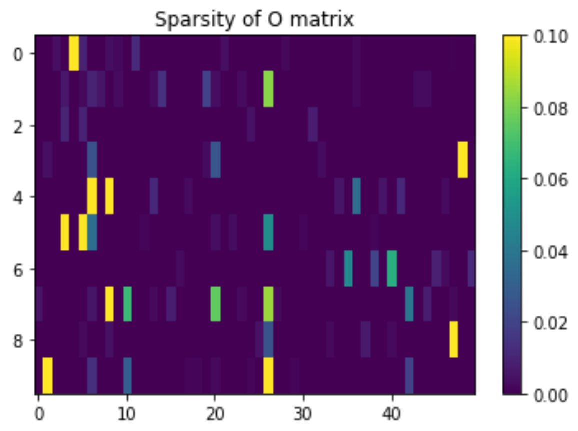
```
In [59]: obs, obs_map = parse_observations(text)
         hmm8 = unsupervised_HMM(obs, 10, 100)
```

Part G: Visualization of the sparsities of A and O

We can visualize the sparsities of the A and O matrices by treating the matrix entries as intensity values and showing them as images. What patterns do you notice?

In [43]:
```
visualize_sparsities(hmm8, O_max_cols=50)
```





Generating a sample sentence

As you have already seen, an HMM can be used to generate sample sequences based on the given dataset. Run the cell below to show a sample sentence based on the Constitution.

In [60]:
```
print('Sample Sentence:\n====================')
print(sample_sentence(hmm8, obs_map, n_words=25))
```

```
Sample Sentence:
====================
And state to georgia for seventeenth ambassadors period court the such power rel
igious the levying all the respective of necessary the number crimes their titl
e...
```

Part H: Using varying numbers of hidden states

Using different numbers of hidden states can lead to different behaviours in the HMMs. Below, we train several HMMs with 1, 2, 4, and 16 hidden states, respectively. What do you notice about their emissions? How do these emissions compare to the emission above?

In [61]:
```
hmm1 = unsupervised_HMM(obs, 1, 100)
print('\nSample Sentence:\n====================')
print(sample_sentence(hmm1, obs_map, n_words=25))
```

```
Sample Sentence:
====================
The union on its vice or office to laws made a and legislature the electors of b
ut each on offices on first existing be law...
```

In [62]:
```
hmm2 = unsupervised_HMM(obs, 2, 100)
print('\nSample Sentence:\n====================')
print(sample_sentence(hmm2, obs_map, n_words=25))
```

```
Sample Sentence:
====================
Third be constitution both he treason first any be so the department the power 1
public the following ii the senator work but electors i...
```

In [63]:
```
hmm4 = unsupervised_HMM(obs, 4, 100)
print('\nSample Sentence:\n====================')
print(sample_sentence(hmm4, obs_map, n_words=25))
```

```
Sample Sentence:
====================
Title the receipts the direct sealed united another particular it and this senat
e one as states such enter may adjournment congress shall the it he...
```

In [64]:
```
hmm16 = unsupervised_HMM(obs, 16, 100)
print('\nSample Sentence:\n====================')
print(sample_sentence(hmm16, obs_map, n_words=25))
```

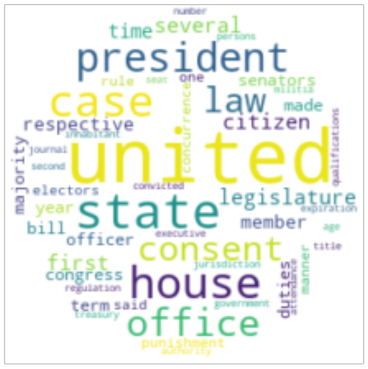
```
Sample Sentence:
====================
Such preserve before determined emolument class except or but the recess of exce
eding state for a president shall be holding order which which exported not...
```

Part I: Visualizing the wordcloud of each state

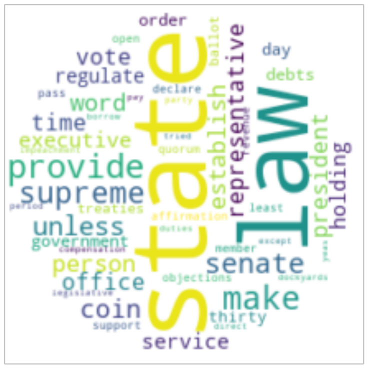
Below, we visualize each state as a wordcloud by sampling a large emission from the state:

```
In [65]: wordclouds = states_to_wordclouds(hmm8, obs_map)
```

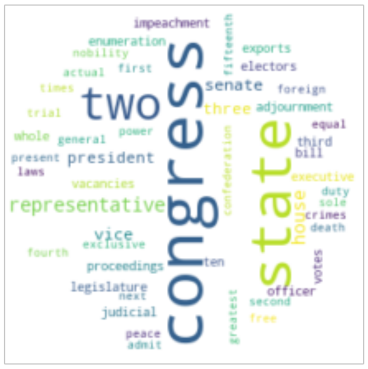
State 0



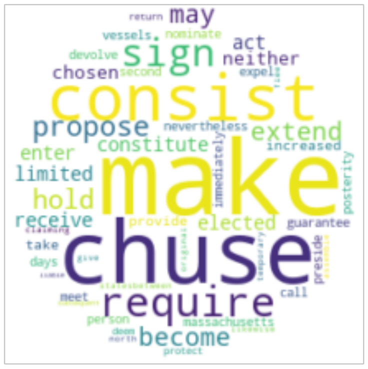
State 1



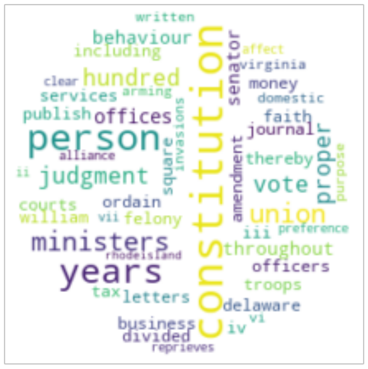
State 2



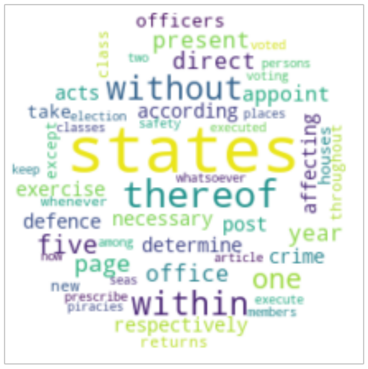
State 3



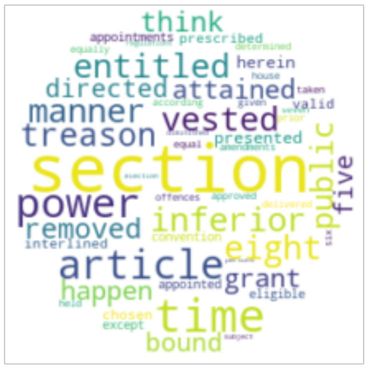
State 4



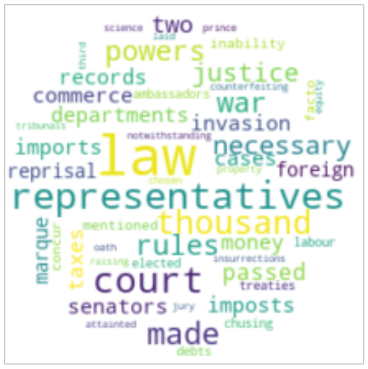
State 5

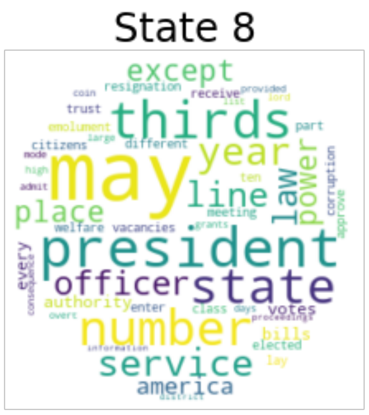


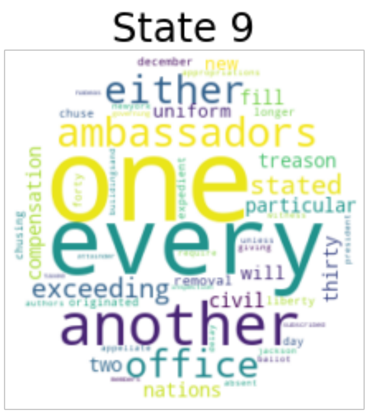
State 6



State 7







Visualizing the process of an HMM generating an emission

The visualization below shows how an HMM generates an emission. Each state is shown as a wordcloud on the plot, and transition probabilities between the states are shown as arrows. The darker an arrow, the higher the transition probability.

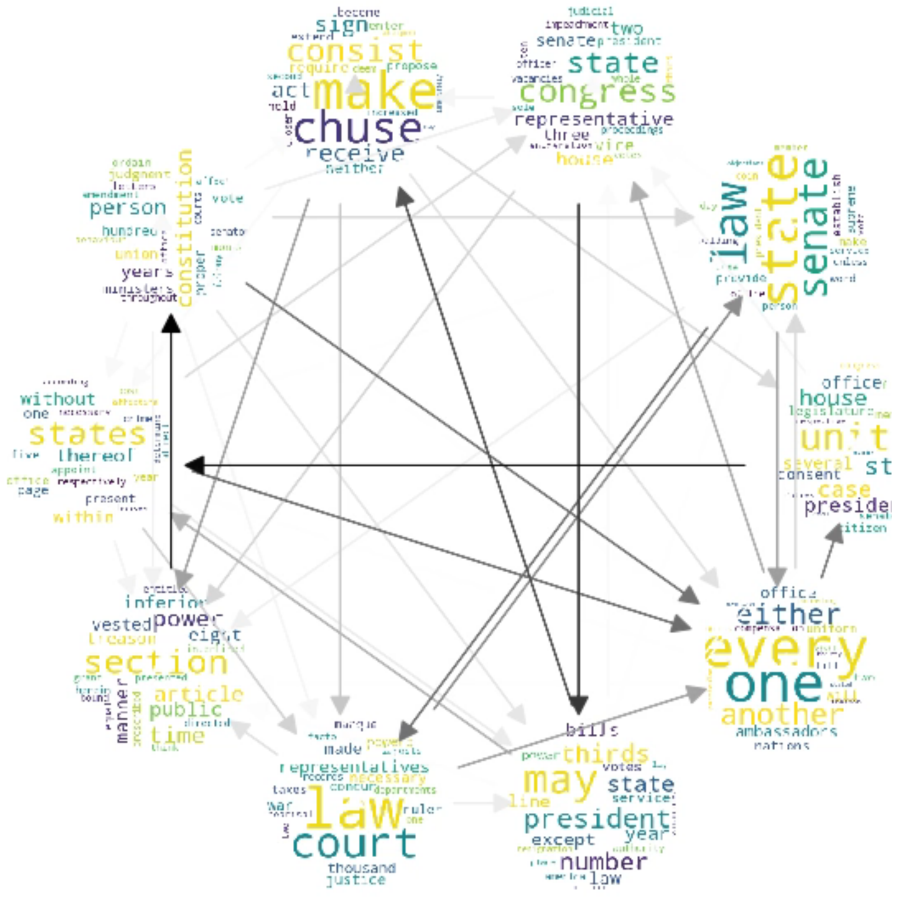
At every frame, a transition is taken and an observation is emitted from the new state. A red arrow indicates that the transition was just taken. If a transition stays at the same state, it is represented as an arrowhead on top of that state.

Use fullscreen for a better view of the process.

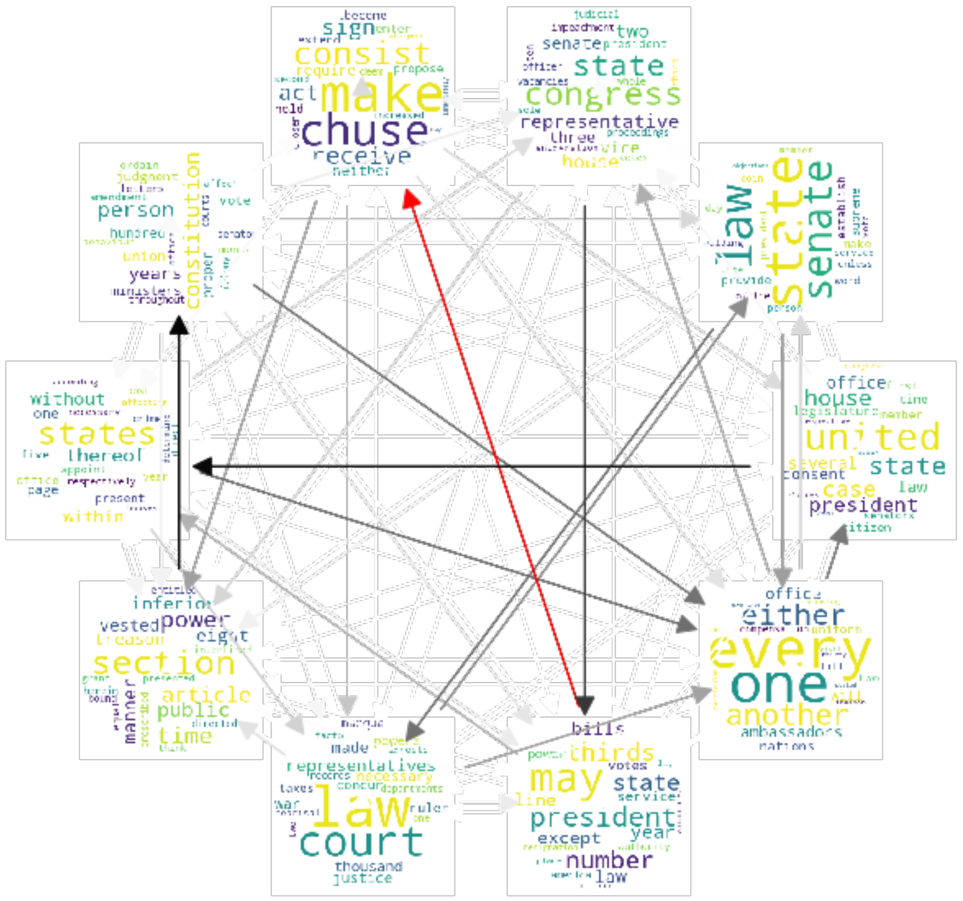
In [66]:
```
anim = animate_emission(hmm8, obs_map, M=8)
HTML(anim.to_html5_video())
```

```
Animating...
```

Out[66]:



Citizen of executing as will congress shall consist



```
In [ ]:
```