# COSC 76: Artificial Intelligence

## Programming Assignment 6: Hidden Markov Models

**Student: Amittai Wekesa (github: @siavava)**

**Fall 2021**

### DESIGN DECISIONS

1. To write my Hidden Markov Model solution, I wrote an abstraction, HMM, that abstracts the inner workings of the hidden markov model. The HMM class offers an interface with `forward`, `backward`, and `viterbi` functions that run the respective algorithms on the loaded maze with specified obervations. The `print` function, when called with a file name, prints the current state of the HMM (computed probabilities over the previous run, path, if one exists (only generated by `viterbi`), and probability ditributions). The different algorithms do save their data in HMM instance variables to avoid each function returning a variable that then has to be handled / printed.
2. **NOTE: I found it easier to work with an inverted y-axis (i.e. y increasing downward), so while checking points on the map please take note that y grows down. Additionally, my maze class is 0-indexed.**
3. I also wrote a Matrix abstraction that wraps a numpy array/matrix and offers a nicer interface to operations such as addition, subtraction, matrix- and element-wise multiplication, and division.
4. My forward and backward algorithms use matrix multiplication to compute the next distributions.

### Forward Algorithm

The algorithm steps through the recorded observations in sequence, multiplying the matrix of reading probabilities for that specific reading by thKpe transition matrix, and multiplying the result by the matrix of distributions at the previous step. As a starting point, the robot's position distribution is computed to be equal for every valid location at the beginning. With subsequent calculations, the position quickly converges to one or a few likely locations.

### Backward Algorithm (Forward-Backward)

The backward algorithm steps throuth the reading in reverse order, starting with an initial vector of ones for each position, and multiplying (element-wise) the probability matrix computed at each position by the projected vector then recomputing the vector with the probabilities computed at the previous step.

With backward smoothing, the algorithm was able to get either the same final distributions but better intermediate distributions, especially incases where two predecessors offered a similar observation. For instance, on `Maze 0`, at step 4:

- The forward algorithm suggested that two predecessors had equal highest probabilities:

```
Step 4

 [0.00108424 0.4873853  0.00055405]
 [0.47909387 0.01607997 0.01580256]
```

- With smoothing, the backward algorithm figured out that one predecessor was more likely than the other, despite the two having the same letter.

```
Step 4

 [2.15681700e-05 9.29239296e-01 1.05194025e-03]
 [3.65330049e-02 2.26041475e-03 3.08937755e-02]
```

**Viterbi Algorithm**

After spending a few hours trying to decode viterbi into a matrix multiplication algorithm, I resorted to a classic dynammic programming format.

1. We initialize values for each position to be the initial distribution at that point multiplied by the transition matrix for the robot staying put at that point.
2. In the dynamic programming step, we compute the maximum of (1) a point's current value and (2) the value of each other point multiplied by the transition probability for moving from that point to the current point. If the test passes, we also update the current point's predecessor.
3. After all the computations, we finally normalize the Matrix.

I added steps in my viterbi algorithm to compute the coordinates on the map (x, y) and the actual sequence of readings on that path, and print these and the possible locations with probabilities for each time step.

It was really interesting to see how the probabilities converge to an almost certain point in the case where the observations are long enough.

**Output from Maze 3**

```
sensor readings = abcdefghijk

Viterbi path: [(2, 0), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (2, 4), (2, 3), (1, 3), (1, 4), (1, 5)]
Actual values on map: ['a', 'b', 'd', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
t = 0: loc = 2, prob = 0.07692307692307693
t = 1: loc = 7, prob = 0.859375
t = 2: loc = 8, prob = 0.013079407859535085
t = 3: loc = 13, prob = 0.012928537154719241
t = 4: loc = 18, prob = 0.24663600007736322
t = 5: loc = 23, prob = 0.9013370513450789
t = 6: loc = 22, prob = 0.9716924312387073
t = 7: loc = 17, prob = 0.960339199597682
t = 8: loc = 16, prob = 0.9601792983510272
t = 9: loc = 21, prob = 0.9600102928332321
t = 10: loc = 26, prob = 0.9600056827107533
```

**Caveats**

1. Since the distribution at the beginning is equal for every point, I found that viterbi tends to get a bit thrown off on the first one or two values. For instance, on Maze 5 (see maze5.viterbi), the viterbi algorithm searches for a match to `"pomonacollege"` – and gets close enough, but due to the starting distribution being even picks out `"n"` instead of `"p"`, giving the sequence `"nomonacollege"`. I found that a little tinkering with teh viterbi algorithm (using `>=` instead of `>`) makes it match the exact sequence, but it becomes a case of whether the correct letter appears *before* or *after* the wrong letter.
2. On another Maze full of text, the viterbi algorithm found a close enough match! To see the text in the Maze, check dump.py. Because of all the text I didn't attempt to format the maze into multiple evenly-sized lines, and I also stripped spaces so it's a bit messy to read. To see viterbi output, check text.viterbi.

**Successes**

- All my algorithms worked. To see sample output, check the output folder. Each output stream was generated by the corresponding test in HMM.py