Points: 90

HMM Sequence Generation

Write a Python program `hmmGen.py` that will read in the parameters of a Hidden Markov Model (HMM) and generate a random string displaying the require frequency distribution of emitted characters and the probability of the string being emitted. Here is the usage message for the program:

```
Usage: hmmGen.py [options] file

This will generate a string from an HMM read from a file

Options:
  -h, --help            show this help message and exit
  -n NUM, --num=NUM     num chars to generate
  -i INITSTATE, --init=INITSTATE
                        initial state for string
```

The number of characters to emit is given in `-n` option. IMPORTANT: The HMM simulation begins in the state given by the `-i` option and **after** the emission of the character for that initial state.

The program reads from a file whose name is given on the command line. The data in the file is of the form:

```
numberOfStates
stateName
numberOfAdjacentStates
prob stateName
prob stateName
prob stateName
  .
  .
  .
numberOfEmissionChars
prob char
prob char
prob char
  .
  .
  .

stateName
numberOfAdjacentStates
  .
  .
  .
```

It is probably most clear to give an example input. Here is the two state HMM for the Loaded Dice Problem:

```
2
```

```
L
2
.05 F
.95 L
6
.75 1
.05 2
.05 3
.05 4
.05 5
.05 6
F
2
.1 L
.9 F
6
.1667 1
.1667 2
.1667 3
.1667 4
.1667 5
.1667 6
```

In this example the decimal numbers are probabilities. The values after the probabilities are either state names or emission characters (in the case of a die it is the numbers 1 through 6). The integers standing alone on a line are the lengths of the lists to follow such as number of states or emission characters. The first 2 says there are two states total. The next line has an L which is the name of the first state. The next line says there are 2 states to can get to from L. etc. There can be any number of states, transitions and emission characters.

Your first task is to open the file and read the data in forming an internal data structure for this problem. For this problem make a data structure that looks like:

```
{"L" : (
         [(.05, "F"), (.95, "L")],
         [(.75, "1"), (.05, "2"), (.05, "3"), (.05, "4"), (.05, "5"), (.05, "6")]
       ),
 "F" : (
         [(.1, "L"), (.9, "F")],
         [(.1667, "1"), (.1667, "2"), (.1667, "3"), (.1667, "4"), (.1667, "5"), (.1667, "6")]
       )
}
```

Notice that the HMM is represented as a dictionary of states indexed by state name. The value in the dictionary is a tuple of 2 lists. The first list is a list of tuples as is the second. They both represent the list of (probability, value) pairs.

One way to read in the data and create the data structure might be to set create routines to read different parts of the structure. For example:

```
def readHMM(file, line) :         # read in the whole HMM.   Calls readState.
def readState(hmm, file, line) :  # read in one state of HMM.   Calls readPList for 2 prob lists.
def readPList(file, line) :       # reads in a list of prob value pairs.
```

Another routine you might helpful is one which selects from the probability list with probabilities given in the list: `def pickFromPList(plist)`

You aren't required to write it this way but if you are looking for some direction this might help you organize your thoughts.

After the data is read in and the data structure is created. You can then generate the required length strings in the format demonstrated. See the usage message above for the inputs. For testing both -i and -n will be given.

Your program should first simply print the data structure created. Then starting with the initial state run the HMM and generating emission characters and storing a string of both the states and emission characters. When you have generated the required number of new states and their characters then print the two strings followed by the probability that that string was generated. IMPORTANT: assume that the probability calculation begins with the probability of transitioning from the initial state given on the command line and then multiplying by the probability of the emission at the new state, etc.

For example given the following command:

```
hmmGen.py -iF -n50 hmmExample.dat
```

with the file `hmmExample.dat` containing the HMM definition above, it randomly generated the following:

```
{'L': ([(0.05, 'F'), (0.95, 'L')],
  [(0.75, '1'), (0.05, '2'), (0.05, '3'), (0.05, '4'), (0.05, '5'), (0.05, '6')]),
'F': ([(0.1, 'L'), (0.9, 'F')],
  [(0.1667, '1'), (0.1667, '2'), (0.1667, '3'), (0.1667, '4'), (0.1667, '5'), (0.1667, '6')])}
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLFLLL
16255245232263161624245223441111111111115111115123
1.8995639467884633e-34
```

Submission

As always, do your own work. Do not copy from others or from the internet. Using what we did in class is perfectly legal. Submit a single file named `hmmGen.py` to the class submission page. The test script will ignore the spacing you choose and focus on non-whitespace. Remember no late papers and so always turn something in for partial credit.