PROGRAM FOUR / CSC 1310

1,000 GRAPHS AND A KEYBOARD

IMPORTANT DATES

Assignment Date: Tuesday, April 12, 2022 in class
Due Date: Thursday, May 5, 2022

DESCRIPTION OF PROGRAM – WHAT DOES THIS PROGRAM DO?

This program is a lot like giving a typewriter to 1,000 monkeys, except instead of monkeys we're using graphs. This is an AI algorithm called a **Markov chain**. There are a handful of applications for Markov chains, but the most common is text generation (for example, the autocomplete text on your phone's keyboard). The output text is based on a set of training data, which is based on the probability of which words follow other words in the training text. Those probabilities are used to generate a **corpus**, which is a probabilistic model that we've calculated based on that source text. Once that corpus is generated, we represent it as a graph and generate text from the path of our traversal. For more on how Markov chains work, see these two resources. The first on Markov chains in general https://setosa.io/ev/markov-chains/ and the second on how they're used for text generation https://blog.demofox.org/2019/05/11/markov-chain-text-generation/

You have some options for this program, since we covered two different ways of representing a graph, as an adjacency list or an adjacency matrix, you get to decide which one you'd like to use here. You also have an opportunity for bonus points. Since you get to pick one, if you do the work implement **both** the adjacency list and matrix for this program you can get up to 20 bonus points. **Two incomplete representations will not be eligible for bonus points** you have to have one finished before implementing the other is worth points, if you have two incomplete only one of them will be graded.

Of the four programs, this one has the fewest lines of code you need to write by far. All the text processing is done for you, that includes generating a corpus based on training data and reading it into your object. You only need to write one header file, which is partially written for you since the constructor generates a graph by reading from the corpus. That said, I can't just make things easier for you by providing clear, concise instructions for every individual step. You'll find a description here for code that's new to you, but more of this program comes down to your own interpretation of the instructions provided. Remember, if you need to ask if something is right, my response is going to be to ask "does it work?"

FILES YOU WILL SUBMIT IN YOUR ZIP FILE

This program contains multiple files as described below:

- **Driver.cpp** – **given to you** – the main function only has two lines, so I didn't really see any reason to make you write it yourself at this point. It creates a graph and then calls the generate function on that graph.
- **markovList.h** – **partially given** – one option for your Markov chain, implemented using and adjacency list.
- **markovMat.h** – **partially given** – one option for your Markov chain, implemented using an adjacency matrix.
- **generateCorpus.py** – **given to you** – this is what I used to generate a corpus from some training data and format it in a way that would be easier for c++ to parse. You can use it to generate your own corpus from another set of training data.
- **Corpus.txt** – **given to you** – a sample corpus provided to you, generated from pirates.txt as sample text.
- **CorpusTest.txt** – **given to you** – a small sample corpus that you can use to make sure you're constructing your graph correctly.

- **pirates.txt** – **given to you** – Howard Pyle's book of pirates, pulled from project guttenberg.

DRIVER.CPP

The main function literally only needs two lines of code: create a Markov chain by calling its constructor, and calls its function to generate text. This is provided for you.

GENERATECORPUS.PY

This is a python script which will read a .txt file (sent in by user input) and generate a corpus from that file. This means that you can play around with whatever training text you want! Do note, that if you feed in another set of training data it will be written to corpus.txt, which means that if you don't want to overwrite your previous corpus you'll need to either rename it or move it to another folder.

If you're curious to get into the weeds about how Markov chains work, do take a look through this script to see what it's doing to make a corpus from training data.

MARKOV CLASS

Both implementations of our graph use a map from c++ stl, they basically work the same as an array (similar to a hash table) but we can use a string as an index instead of an integer. The adjacency list maps every string in the corpus to a list of its adjacent nodes. The adjacency matrix maps every word in the corpus to an integer, and that integer is the matrix index associated with that word. You can find documentation for the stl map here https://www.cplusplus.com/reference/map/map/

For an adjacency list you'll also need a node struct that contains the destination node, a weight (the percent probability of traversing to that node), and of course a pointer to another node. For an adjacency matrix you'll want a struct to bundle your map with your 2d array.

There are a couple of places you'll need to use an interator to to loop through every element in the map (and to grab a random node when you start the generate function). You can use these in a for loop in pretty much the same way you'd use an integer to loop through an array, but the syntax is a bit different. You can find sample syntax here https://www.cplusplus.com/reference/map/map/begin/

The generate function takes in an integer, which is the number of words the function generates, and it returns a string, which is the output text that it has generated. The code to grab a random starting node is provided for you. Once it grabs that random node, it loops for the number of words it's generating (passed as the parameter). At every iteration of this loop it makes a random roll to determine which node it traverses to, you'll use `(float)rand()/RAND_MAX` as the value for this variable. It's going to check this value against to weight of every edge. For the adjacency list that means traversing through every node in the list (none of which have a weight of zero); for the matrix that means looping through every index in the matrix (most of which will have a weight of zero, you'll need an iterator for this). If the random value is less than the weight for an edge (aggregated with the sum of edges that have already been checked) then it does a traversal to that node. So if node A has edges to nodes B, C, D, and E with weights of 0.1, 0.4, 0.3, and 0.2 respectively, if we roll a 0.75 the it will traverse to node D (0.75 is not less than 0.1, so it checks the next node. 0.75 is not less that 0.5 (0.1+0.4), so it checks the next node. 0.75 is less than 0.8 (0.1+0.4+0.3), so it does a traversal to node D). Once it does a traversal it adds the destination node to the string we're going to return at the end.

The minimum functions you need for this program to run correctly have prototypes in the code provided to you, you may add other functions if you think they'll be useful. **You only need to implement the graph using one of the two representations, you don't need both for full credit.**

EXTRA CREDIT OPPORTUNITY – DO BOTH

Can't decide which one you want to use to implement your program?

If you implement the program using both graph representations you can get up to **20 bonus points**. One of them must be complete to be eligible for bonus points. If one is complete but not the other you'll get partial bonus points, but if neither are complete you'll only get partial credit for one of the two at the discretion of the grader.