Project One: Data Structures for Text Analysis

In this project you will construct several data structures given a sequence of words. Part of the program will demonstrate that the data structures have been constructed by displaying on the console the contents of the data structures. More precisely, you are given a text file and you divide it into sequences of non-whitespace characters (called “words”) separated by whitespace characters. The whitespace characters are the blank character, the tab character, and the newline character.

Project Directory Structure

- On agora enter at the command prompt

  mkdir cs253
  cd cs253
  mkdir project1
  cd project1

  to create a cs253 directory for this course and put a project1 directory inside your cs253 directory and then move inside your project1 directory.
- Use vim to create a file named README by doing

  vim README

  Use the vim text editor to create it. See the linux/vim course notes on the course website to help in learning vim.The README file lists the author names and date, explains how to run your program and any aspects of the program that do not work. You will lose fewer points for parts that do not work that you tell me about.
- Create a subdirectory, named app, for your project source code and cd into your app directory.
- In your app directory run the command

  npm init

  to create your package.json file. Follow the prompts and enter the requested information into your package.json file. Using the defaults is fine.
- In the app directory you will need to have one file named

  data_structures.js

  to hold the javascript code for your solution. Use the vim text editor to create it. See the linux/vim course notes on the course website to help in learning vim.
- The file data_structures.js will include the functions
  - wordCount
  - wordFreq
  - condWordCount
  - condWordFreq
  - main

and any related functions you decide you need.

- Your function named main can take the name of the input text file as a command line argument as in the example code on the course website (using the fs node module). Alternatively, your function named main can take one string argument which is the name of the input text file. In this case, after the function definition for main you could call it several times once for each input text file such as below:

  main('rbbrrg_input_text.txt');
  main('empty_input_text.txt');

  I reserve the right to add extra calls to main that use other input text files.
- To run your program, cd to your project directory and then run the command

  node data_structures.js

  if using a main function that takes the name of the input text file as a string argument. Run the command

  node data_structures.js rbbrrg_input_text.txt

  for example, if you are using a command-line argument for the name of the input text file.

Data Structures

1. Your program needs to construct the following data structures (each constructed in its own function).

- an object named wordCount, that for each word in the input word sequence has the number of occurrences of that word
- an object named wordFreq, that for each word in the input word sequence has the percentage (that is, the probability or frequency) of all the word occurrences in the input word sequence that are occurrences of this word
- an object named condWordCount, that for each word A in the input word sequence has a object that is a series of key-value pairs in which the key is a word B that occurs at least once as the word immediately following the word A and the value is the number of times the word B immediately follows the word A.
  - View the first word in the input word sequence as immediately following the last word in the input word sequence. So if the last word in the input word sequence never occurs before in the input word sequence, it will have at least one word that is considered to immediately follow it.
- an object named condWordFreq, that for each word A in the input word sequence has a object that is a series of key-value pairs in which the key is a word B that occurs at least once as the word immediately following the word A and the value is the probability that word B is the word immediately following word A. In other words, the value is the percentage of the time that word B immediately follows word A in the input word sequence.

Example Data Structure Values

Suppose the input word sequence in the input file is

red blue blue red red green

then the number of occurrences of each distinct word is (the order of the key-value pairs in an object is randomly generated and so will not always be in the order below)

wordCount is {'blue': 2, 'green': 1, 'red': 3}

The frequency of each word in the input text is

wordFreq is {'blue': 0.3333333333333333, 'green': 0.16666666666666666, 'red': 0.5}

since two of the six word occurrences are 'blue', one is 'green', and the other three are 'red'.

The conditional frequency of each word in the input text is

condWordCount is {'blue': {'blue': 1, 'red': 1}, 'green': {'red': 1},
'red': {'blue': 1, 'green': 1, 'red': 1}}

The variable condWordCount is stating that after the word 'blue' appears in the input text, one time is 'blue' the next word and one time is 'red' the next word, and so on.

The conditional word frequency is

condWordFreq is {'blue': {'blue': 0.5, 'red': 0.5}, 'green': {'red': 1.0},
'red': {'blue': 0.3333333333333333, 'green': 0.3333333333333333,
'red': 0.3333333333333333}}

The variable condWordFreq is stating that after the word 'blue' appears in the input text, the probability that the next word is 'blue' is 0.5 and the probability that the next word is 'red' is 0.5, and so on. Note that we are saying that the word 'green' has the word 'red' following it since the only occurrence of 'green's is the last word in the text and 'red' is the first word of the text.

An Example Output of the Four Data Structures

Suppose the input file named empty_input_text.txt has either no characters or just whitespace characters (the whitespace characters are the blank character (' '), the tab character ('\t'), and the newline character ('\n'). Then the function call

main('empty_input_text.txt');

in the file data_structures.js must generate the console output

Input can not be empty or only be whitespace.

Another Example Output of the Four Data Structures

Suppose the input word sequence in the input file named rbbrrg_input_text.txt is

red blue blue red red green

Then the function call

main('rbbrrg_input_text.txt');

in data_structures.js must generate the output on the console that looks exactly like (the order of the key-value pairs in an object is randomly generated and so will not always be in the order below)

wordCount is {'blue': 2, 'green': 1, 'red': 3}

wordFreq is {'blue': 0.3333333333333333, 'green': 0.16666666666666666, 'red': 0.5}

condWordCount is {'blue': {'blue': 1, 'red': 1},

'green': {'red': 1},

'red': {'blue': 1, 'green': 1, 'red': 1}}

condWordFreq is {'blue': {'blue': 0.5, 'red': 0.5},

'green': {'red': 1.0},

'red': {'blue': 0.3333333333333333, 'green': 0.3333333333333333, 'red': 0.3333333333333333}}

Text File Input

Here is an example node program illustrating reading from a text file that has been specified by a command line argument. This code is also on the course website.

```
var fs = require('fs');
  var data = fs.readFileSync(process.argv[2],"utf-8");
  fs.writeFileSync(process.argv[3], data);
  console.log("done");
```

fs is a node module and npm package for file system operations. Here is the api documentation

https://nodejs.org/api/fs.html

var fs = require('fs');

This first line in the example code imports the module into your program.

var data = fs.readFileSync(Process.argv[2],"utf8");

This second line in the example code reads all the contents of the file specified by the variable Process.argv[2] into the variable named data. The variable Process.argv[2] holds the third value on the command line. For example, consider the command line below.

node data_structures.js rbbrrg_input_text.txt

This line creates an array in the variable Process.argv which is

[``node", ``data_structures.js', ``rbbrrg_input_text.txt"]

so Process.argv[2] holds the string ``rbbrrg_input_text.txt". This is just like Java. Recall in Java the method header

public static void main(String[] args)

and if the command line is

java HelloWorld Mars

then the args array holds [``java", ``HelloWorld", ``Mars"]

So by passing Process.argv[2] to the fs.readFileSync method we are opening and reading the content of that file. As a result the variable named data will now hold the content of that file. You can then process that content to meet the requirements in the handout.

You can specify the input text file as a command line argument as in the example code above. Alternatively, you can have at the bottom of your source file several calls to your main function each of which takes the name of the input text file as its argument.

Other Comments

- In the case that the input text file is empty or is only whitespace characters, your main function should output to console.log the error message shown in an example later in this handout.
- Your solution can view a word as any sequence of non-whitespace characters. So whitespace separates words. So ``red, blue" is two words ``red," and ``blue".
- Node.js has a module called ``fs" for ``file-system" that can be used for reading and writing files.
- On the course website I have a short example file that runs on agora using node.js and fs to read command-line arguments, to read data from one file and write it to another file, and to write a message to the console. You do not have to use command-line arguments in this project, but the use of reading files is relevant.
- On the course website I have some example input text files.
- You can use the JSON.stringify() method code to return a string representation of a javascript object. You can then pass that string to console.log to display the javascript object on the console.
- The string data type has a split method that takes an argument which is the delimiter to use in splitting the string and which returns an array of the substrings. Here is a link for more details: https://msdn.microsoft.com/library/t5az126b%28v=vs.94%29.aspx.

Administrative

- The project is due at 5 pm on Friday, the 30th of September. You can submit a project multiple times. I will just grade the last version and the time and day of the last version is the time and date that count.
- Late project submissions will be accepted but with late points deducted. Every twenty-four hour period starting at 5 PM is one late day which is five points deducted. So, for example, a project turned in at 5:01 PM on the 28th will have five points deducted.
- Weekends and holidays do not count towards late points in the following sense. A project submitted from 5:01 PM on Friday until 5:00 PM on Monday counts as one extra late day. For holidays the approach is similar.
- You may work in teams of one or two students from the class. You may talk with other students in the class about the concepts involved in doing project one, but anything involving actual code needs to just involve your team. In other words, you can not show your team’s code to students outside of your team and you can not look at the code of another team.

- Make sure to use good programming style and comments. There should be at least one blank line between the end of one function's body and the header of the next function. Follow the airbnb style guide for javascript. There are two versions which are listed below. The first version is for ES5 which is the current javascript standard. The second version is for ES6 (that is, ECMA 2015) which is the draft new javascript standard. You can follow either of these versions.
  1. https://github.com/airbnb/javascript/tree/master/es5
  2. https://github.com/airbnb/javascript
- Your project will be graded based on how it runs on agora when I test it.
- Remember you need a JSDoc comment (that is, a /** style comment) at the start of each source file or test file and just above each function header. In the JSDoc comment at the start of each file you need @author for the author names and @version for the date.
- JSDoc is very similar to javadoc. See the following links for more details
  1. https://en.wikipedia.org/wiki/JSDoc
  2. http://usejsdoc.org/
  3. https://github.com/jsdoc3/jsdoc
- Your code must not be placed on github since github repositories are public.
- Submit your project directory as a gzipped tar file via handin on agora. So the tar file will contain your README file, your tests files in the qa subdirectory, and your source files. If your project is in a directory named project1 and you are in its parent directory, then you can use

  tar -cvzf project1.tar.gz project1

  to create the file project1.tar.gz. The 'c' means to create a tar file, the 'v' means verbose, the 'z' means to turn the tar file into a gzipped file, and the 'f' specifies that the next command line argument is the name of the resulting file. Then use the following command to submit the file.

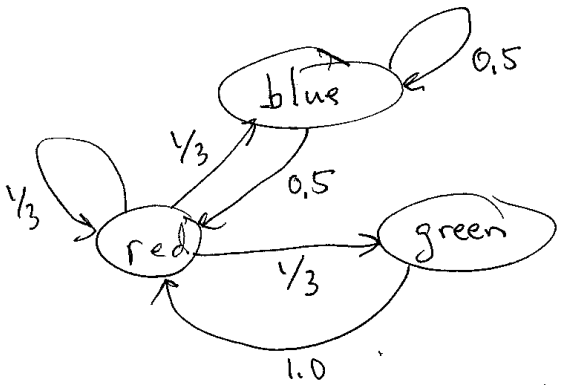
  handin.253.1 1 project1.tar.gz

Appendix: Algorithmic and Mathematical Interpretations (optional)

The data structure condWordFreq is the adjacency list representation for the probabilistically-weighted directed graph describing the input word sequence. The directed graph is said to be probabilistically-weighted since all the edge weights are between 0.0 and 1.0 and all the edge weights coming from the same source vertex sum to 1.0. See the figure below for the weighted directed graph for the example word sequence

red blue blue red red green

In probability theory, the weighted directed graph that is condWordFreq is the state space of a discrete-time markov chain stochastic process. The edge from one state A (that is, vertex in the graph) to another state B (another vertex) is the next step probability of a state transition from state



A to state B. Markov chains have many applications in computer science including in machine learning, bioinformatics, and speech recognition.