EECS 233 Introduction to Algorithms and Data Structures
Programming Assignment 4 ("P4")

Due Date: Thu Dec 4 by midnight

In this assignment, you will be responsible for demonstrating the correctness of your code and each of our methods. Thus, there will be no drafts. To demonstrate correctness, you will do the following:

- Design a small set of text files to test the correctness of your code using well-chosen cases.
- Write a `main` method which processes your test files and outputs both descriptive text and results in narrative form so that they can be easily read and interpreted by the graders.
- Your `main` method should also process the cases specified below in the assignment.
- Write a companion document (in pdf format) to go along with your main method and turn it in with your code. Name this file `testing.pdf`. It should explain the rationale behind your test cases illustrate each test case with graph diagrams and the results you expect.

Assignment

Total Implementation Points: 80 (60% of total) + 10 points extra credit

In this assignment you will implement a graph-based phrase generator. Like in P3, you will calculate simple word statistics from a given text (and you can reuse some of your P3 code), but instead of storing the statistics in a table, you will construct a directed graph in which the nodes store the words and their counts and the weights of the edges represent the number of times one word followed another word in the source text. Once you have constructed the word graph, you will use Dijkstra's algorithm to find shortest paths to generate phrases based on simple word sequence statistics.

1. WordGraph

Implement a `WordGraph` class to encode the word sequence statistics from a given source. In this assignment, you have much more flexibility in your implementation, e.g. you will need to define `WordNode` and `WordPair` classes for the graph vertices and edges, but how you define them as well as what helper classes or methods you will need is up to you. You are free to use any of Java's built-in classes to implement the functionality, but you must implement the graph methods and Dijsktra's algorithm yourself.

- a constructor with a `String` argument builds a `WordGraph` from that file name. You can choose how you wish to tokenize the file and normalize the words. In the graph, nodes represent unique words in the source text and store their counts. Edges are directed and represent a links between words. An edge connecting nodes i and j stores the (non-zero) count of the word pair (w_i, w_j), i.e. the numbers of times w_i follows w_j in the source text. Like in P3, you should use efficient algorithms and data structures to construct the graph. — 5 P.
- `int numNodes()` returns the number of nodes in the graph. — 5 P.
- `int numEdges()` returns the number of edges in the graph. Note that since the graph is directional, each directed connection should count once. Thus, a graph with two nodes that are mutually connected has two edges. — 5 P.
- `int wordCount(String w)` returns the count of word `w`, i.e. the number of times it occurred in the source text. — 5 P.
- `int inDegree(String w)` returns the in degree of the word node `w`, i.e. the number of (unique) words that preceded the word `w`. — 5 P.
- `int outDegree(String w)` returns the out degree of the word node `w`, i.e. the number of (unique) words that followed the word `w`. — 5 P.

- `String [] prevWords(String w)` returns a `String` array of all the words that preceded the word `w`, i.e. all the nodes in the graph that have connections to the node for word `w`. 5 P.
- `String [] nextWords(String w)` returns a `String` array of all the words that follow the word `w`. 5 P.
- `double wordSeqCount(String [] wordSeq)` returns the "cost" of the word sequence given in the `String` array `wordSeq`. Below you will write the an implementation of Dijkstra's algorithm to find plausible word phrases. This requires a notion of path cost, so here we will use the simple definition that cost is the sum total count of each word pair count in the source text: 5 P.

$$C(w_1,\ldots,w_n) = \sum_{i=1}^{n-1} C(w_i, w_{i+1}), \tag{1}$$

  where $C(w_i, w_j)$ is the count of word pair w_i, w_j in the source text. Instead of finding the shortest path, however, you will find the most plausible sequence by finding the path, i.e. word sequence, with highest `wordSeqCount`. You do not have to use the `wordSeqCount` method in your shortest path algorithm below, as it will be easier to compute path cost incrementally, but you can and should use `wordSeqCount` to verify the correctness your `generatePhrase` method.

- `String generatePhrase(String startWord, String endWord, int N)` returns the most plausible word sequence (as a single String, with words separated by spaces) that is at most `N` words long, starts with `startWord`, and ends with `endWord`. You should use Dijkstra's algorithm to find a path from `startWord` to `endWord`. This will fill in the intermediate words by finding the word sequence with highest path cost using the definition above. Note that the word sequence with shortest path cost is not necessarily a sequence of the most common next words for each word. The argument `N` serves as a way to limit the depth of the search, since you do not know the length of the shortest path. Return an empty string if a path does not exist or if all sequences of `N` words or less have been searched. 15 P.
- (Extra Credit) `String generatePhrase(String startWord, String endWord, int N, int r)` is identical to the method defined above, but the extra argument `r` specifies the number of times each words can be repeated in the phrase. For example, `generatePhrase("car","tree",5,0)` indicates that the method should generate a phrase of (at most) 5 unique words. `generatePhrase("the","car",6,1)` would generate a phrase of up to 6 words, but allows up to one repetition of each word. +10 P.
- (Required) Write a `main` method that reads a well-designed set of test files and outputs the results of calls to the methods above to demonstrate their correctness. You should copy the output to the file `main.txt` and submit it along with the test files with your code and the `testing.pdf` file. 20 P.

  The output of your `main` method should follow your `testing.pdf` file. The format is up to you, but you should make it easy for graders to compare the two files and follow your rationale. We will run your code to verify that it generates the same results in `main.txt` and what you reported in `testing.pdf`.

  You be graded on your choice of test cases and how clearly and convincingly you demonstrate the correctness of your code in terms of your choice of test cases and your rationale as explained in your `testing.pdf` file.

  This is intentionally free-form, so please refrain from asking questions of the nature "Is this what you want?" or "Is this convincing?" The point here is for you design your own test cases and demonstrate that your code is correct in a manner that is clear, concise, and convincing. If you are in doubt, ask a classmate to read your results and give you feedback.

2. Grading

Your grade will consist of the following:

compilation	20%
correctness	30%
effort	30%
design	15%
style	5%
Total	100%

a) *Compilation* (20%) Does it compile or not? This is given a weight of 20% for two reasons: 1) If your code doesn't compile, we can't check its correctness by reproducing your output. Non-compiling code implies zero points for correctness. 2) Debugging compilation errors can be difficult and time consuming, but it is a fundamental aspect of programming. To avoid this, develop incrementally and compile often to make sure your code remains in a working state. If you encounter an error, learn how to use a debugger to debug your code efficiently.

b) *Correctness* (30%) As described above, in this assignment we will determine correctness based on your test cases in your `main` method and your description and rationale in `correctness.pdf`. If we cannot reproduce your output in `main.txt`, by compiling and running the code ourselves due compilation errors or an infinite loop you will not receive a correctness grade.

c) *Effort* (30%) The implementation points given at the beginning of the specification constitute 60% of the total grade: 30% is correctness; the other 30% is effort. Like for JUnit tests, if for some reason we cannot reproduce your results, you can still get half the points if it looks like you made a reasonable effort to implement the functionality. Code that is incomplete or obviously wrong will receive zero credit for those methods.

d) *Testing* Note that in this assignment testing and correctness are combined. That said, the general guidelines for testing your code still apply:

- Test as you write the code. This goes along with writing incrementally. Once you implement a part of the assignment, write a test to check its correctness.
- Test code at its boundaries. This is also referred to as testing "corner cases." Do conditions branch in the right way? Do loops execute the correct number of times? Does it work for an empty input? A single input? An exactly full array? Etc.
- Test systematically. Does your test code check every condition? Does every line get executed? Especially test parts you're going to reuse. Otherwise, debugging can be much more difficult.
- Test automatically. Have test suite that automatically runs all the tests. JUnit helps provides this functionality. This helps identify errors as soon as they are introduced, which would otherwise can be very difficult to debug.
- Stress test. Does your code work on large inputs? Random inputs? It's easy to make implicit assumptions that aren't always valid. Stress testing can help uncover these.

e) *Design* (15%) Scored on a scale of 0 to 15. Design and implementation[1] considers how you chose to structure your program to implement the specification. It also concerns how you structure the code within a method and your choice of data structures and algorithms. For grading we will consider things like, how you chose to break down a problem into smaller components, how elegantly your code handles edge cases, etc.

f) *Style* (5%) Scored on a scale of 0 to 5. Style considerations include (from *Practice of Programming*):

- Variable name choices. Are they descriptive and do they convey information about their purpose? Note for simple concepts, it perfectly acceptable, even preferable to use, for example, `i` for an index or `n` for an integer.

[1] The design, style, and testing considerations follow the guidelines and ideas in *The Practice of Programming* by Kernighan and Pike.

- Expressions and statements. Are they clear? Do they make the meaning as transparent as possible? Well-written code should not need much commenting, if any. Is it formatted in a way that aides readability?
- Consistency and Idiom. Is formatting consistent? Are common operations consistent with programming idiom? (the textbook, for example)
- Comments. Do they aid the reader in understanding the code? Comments that restate what is already clear the code are redundant and not helpful. Nor are comments that are not consistent with the code. For more complex methods, the comment should state the running time of the method (or running times if there are different cases).

General Instructions and Guidelines

Turning in your code. You should submit a zipped folder of your java source and JUnit tests files via Blackboard. Follow these instructions:

- Make a separate submission folder with the following name: `EECS233_abc123_P4` where `abc123` is replaced by your own Case Network ID.
- The folder should contain your java source code, your `main.txt` output file, your `testing.pdf` file, and your test case text files.
- Do *not* include the java `.class` files. They take up too much space, and we will compile your files ourselves.
- Your submission folder should not contain subfolders.
- Compress your submission folder using the `.zip` format (do not use `.rar` or some other format)
- The compressed filename should be the same as the folder name, but with the `.zip` extension.
- Submit this file via Blackboard.
- **Ensure your code compiles**. You must ensure that your code is in a stable state and compiles before submitting your code.

Version Control. You should always write your code incrementally, with a short write-test-debug cycle. Once you have a component written (which could be just a single method), write JUnit tests that check that your implementation is correct. Since we cannot grade what we cannot compile, you need to use some kind of version control to ensure that you have a saved working version once each step is completed. This could be as simple as copying the directory or using a sophisticated version control system like "git". Anything is fine, as long as you always maintain a working copy of your code that you can turn in (or revert to if you introduce a bug that you can't track down).

Use an Automated Backup System. Everyone should be using a backup system that backs up your files automatically and seamlessly, such as Dropbox or Google Drive, among many others. They are free services for small space requirements, so there is no excuse anymore for loosing your work to a hard drive failures or accidentally deleted files. If you haven't done so already, set up a such a system now.

General Strategies. It is helpful to work through the logic of the implementation for the problems in the assignment on a piece of paper, *before* you write any code. This can include things like data structure design, implementation, and small examples you use to work through the steps of your algorithms. You do not need to turn this in, but it can be helpful for grading in you include a README.txt file or a pdf scan of your notes that explains your design choices to the grader. Your code comments should describe clearly your high-level logic, design choices, and the running time and/or memory complexity of your algorithms.

Cheating and Plagiarism. It should go without saying that you should solve the problems and write the code on your own. You can consult the book and the internet for java questions, but do not look for solutions to the problems given in the assignments. You are encouraged to get help from your classmates regarding basic

programming issues, debugging problems, features of Eclipse, etc., but like I mentioned in class, be careful about giving (or receiving) too much help. Not only might it spoil someone's learning efforts, it may appear as plagiarism. To ensure fairness, we will run software to check for cheating, so please do not copy code. Also keep in mind that your goal should be to develop the ability to think through problems, find solutions, and implement them in code. Some exam questions will be designed to test to what extent you have achieved this using examples drawn from the assignments.