**Anagram Detector Application**

This project is implemented with performance in mind. The goal is to scan a file with list of words and while scan check to see that for any current word scanned, if its a anagram of any of the previous words scanned, print to Console True, else print False.

Please find attached a UML diagram for the layout of all objects used in implementing the functionality of the application. Also you will find some comments within the code to help illustrate parts of the implementation where necessary.

Input: list of words read from file

Assumptions:

1. Words are made up of only lower case ANCII Character set: [a-z]

2. Words can be arbitrary in size ranging from 2 to at most 100 (max 100 for the sake of illustration)

3. Duplicate words can be stored randomly

4. The file may contain lines with no words or multiple words. Such lines will be ignored

Output: to Console

- Boolean Values expressing if the current word is an anagram of a previous word or not

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Thought Process: U D I T

U: Understand the Problem

- What's is an Anagram: Words that contain the same character including the occurrence of those characters

- What's my required Inputs & Outputs?

- How can I store words read from file to check for anagrams

- How should I check for anagrams using the least memory and fastest comparisons

D: Design Solutions & Examine Constraints for each

(Below I explain all the options (in some detail) that I considered)

1. Use a HashSet<String>(instead of HashMap to avoid dealing with duplicate words read) to store all the words read from file. Before adding a word to the HashSet check if the current word is an anagram of any of the words in the HashSet of words. Anagrams can be compared by checking each character (and its occurrence count) of the current word with the comparing word to make sure its an anagram, however, this approach require O(n^2) (and consequently O(n^3) for the entire solution) time is at the least very exhaustive. I quickly noted a more clever way to compare the words in O(n) [linear] time by simply comparing two words by storing them in an ArrayList<Integer> where the index in the array would be the ANCII character value normalized to 0 (ie. subtracting the value by 97 since the 'a' Character in ANCII has a value of 97) and the value at the index would be the occurrence count of the character. Below is the Java code for checking this:

public static boolean makeAnagram(String currentWord, String storedWord){

if(currentWord.length() != storedWord.length()) return false;//words must be same length

Integer[] currentWordChars = new Integer[totalAlphabets];

Integer[] storedWordChars = new Integer[totalAlphabets];

//create a temp Arrays to compare the words

storeWordCharacterInArray(currentWordChars, currentWord);

storeWordCharacterInArray(storedWordChars, storedWord);

for(int i = 0; i < totalAlphabets; i++){

//compare the new word to the current charList to see if anagram is possible

if(currentWordChars[i] != storedWordChars[i]) return false;

}

return true;//and store this word in the HashSet of word in the Heap

}

//for each word store its characters

public static void storeWordCharacterInArray(Integer[] characterList, String word){

char[] charCheck = word.toCharArray();

for(char c: charCheck){

Character cc = c;

int index = cc.charValue()-indexNormalizer;

characterList[index] += 1;

}

}

When analyzing this solution, I noted that this solution would work fine for a small set of words, however, for a large file, our heap memory as well as anagram computation could become very slow due to the comparisons necessary. This solution would have the following Complexities:

Time Complexity: O(n\*m) since we are checking each current word with all the n words read and assuming all words can have at most m (in this case 26) different character.

Space Complexity: O(n\*m) for each word stored where n is number of words and m is maximum characters per word. Since we can have arbitrary word size, m here could be very large and we could potentially exceed our heap space, which would be too

2. Using multithreading to modify the solution from 1. When scanning the word file use an ArrayList to store the words, and when checking this array in the heap for anagram, break the size into sublists for each independent thread to compute. Since most machine today have more than one core, true parallelism can be achieved here resulting in a much faster computation.

Time Complexity: O(m\*n)/l where n is the words read, m is possible characters a word can have (26 for us), and l is the number of cores on the machine running this.

Space Complexity: slightly more efficient than solution 1 since we are using an array list instead of a hash table to store the words (although in the case of many duplicate words we may have a storage inefficiently).

3. After examining my earlier two solutions and feeling displeased about the space and time complexity necessary for the application (despite threading for speed) I starting thinking of what other data structures I could use and after a few minutes remember one called: Trie. I had used it previously for creating a dictionary of words. With a Trie I could store repeated characters from other words together, thus using the least possible memory, and iterate in near constant time to find a word: O(l\*k) where l is the number of character in the word and k is the character set size, both of which are constant values per iteration. Since Trie is not part of any Java library, I built a TrieTree and TrieNode Classes. I then noted that for detecting anagrams I will need to store the characters in a sorted order, hence I will need to sort each word prior to adding it to the Trie. I made some optimization such as checking to make sure the word exists prior to adding it to the Trie.

For each TrieNode, it will have a set of child nodes. I made the decision to use an array list here instead of an array of 26 nodes because I felt it would be more dynamic and require much less memory when starting since an Arraylist by default is of size 10 instead of 26, and often we will have child nodes not linking to all 26 characters.

Time Complexity: O(n\*m) ~ O(1) where n is number of character in a word and m is 26 (the total different characters a word can have under our given assumption), both of which are constant for a given word.

Space Complexity: O(n\*m) where n is the number of unique character we can have (26 here) and m is the size of the largest word (unknown). This way instead of having to store every word, we are only storing every unique character at a given position.

Other more elegant solutions likely exists, however, in the given time I have attempted to implement a solution I find best meet the needs of the task.

I: Identify Solution & Implement

- After considering all three options I decided to go with the 3rd option since it has the best space and time complexity for large datasets with respect to the purpose of the application. It is also designed with scalability in mind and thus can be easily extended to words with large character sets. Moreover, it is designed to work much more efficiently as data sets get larger, when compared to solution 1 and 2. It is a more memory intensive solution for smaller data sets than solution 1 or 2 would be. Given a random distribution of words in the file and the character making up those words, under our current assumptions, a likely threshold word count/file can be found upon which the performance benefit of the 3rd solution can truly be realized. This threshold, however, is a value which would require more testing and trials to determine.

T: Test diligently where necessary

- For each small piece, I performed some tests to make sure my solution piece was correct

- Do heavy testing once the application version 1 is built to find ways to improve the performance (speed and memory).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Testing

Unit: Validating the functions from each of the three classes.

I added some but due to time constraints could not add all such tests as I would have liked.

- Anagram Class

- TrieTree Class

- TrieNode Class

Functional:

* Performed using various datasets with varying file word formats: Black Box Testing approach

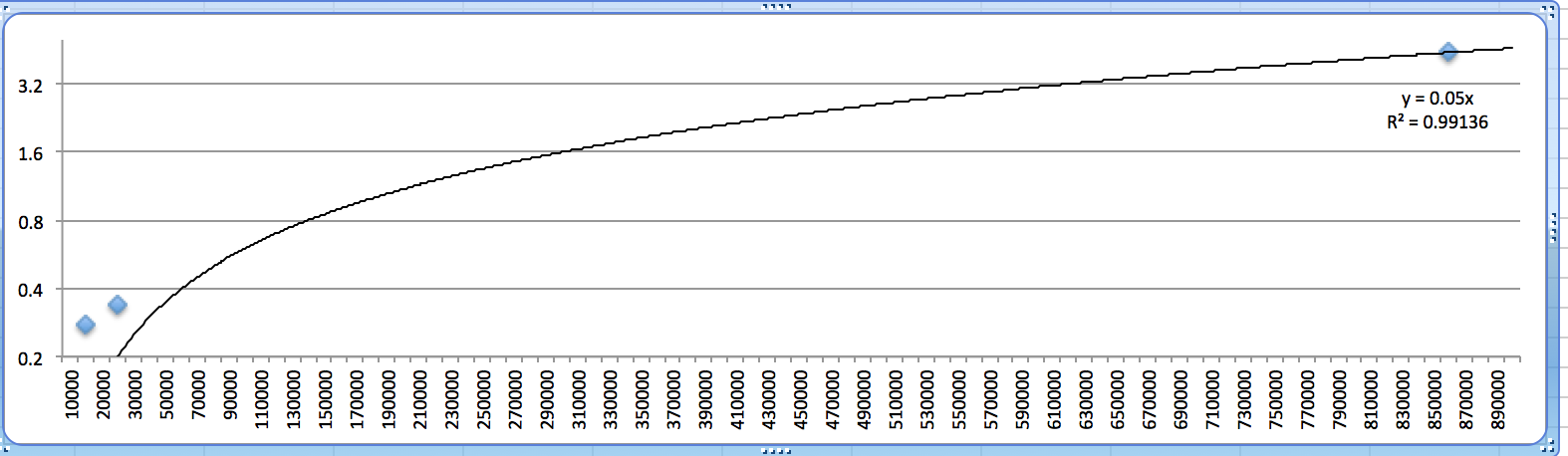
Integration:

- N/A

Performance:

* Using jConsole and jMap while running the application to study its behavior at Runtime.
* By analyzing the time taken for computation over various size datasets I created the following table:

|  |  |
| --- | --- |
| File Size (# of words) | Time taken to detect Anagrams (seconds)  Various trial separated by commas |
| 13484 | 0.48, 0.28, 0.289, 0.30, 0.28 |
| 26109 | 0.55, 0.35, 0.348, 0.347, 0.343 |
| 869230 | 4.44, 4.91, 4.34, 4.38, 4.37 |



Based on this data, the solution can be seen to scale exponentially. This is shown from the log graph above where as we increase the size of our dataset exponentially we increase our time taken for computation linearly.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Other thoughts for saleable solutions where memory is constraint:

Despite using my solution 3 from above where we are most carefully optimizing the memory needed (by storing characters and their positions uniquely, ie a Trie) as well as the speed of the application, it is possible that the word file might contain a set of characters, collectively, which might exceed the memory needed to build a Trie (ie. an older machine with less RAM). In such a case we have one of 2 options:

1. Break the word file into smaller pieces using some sort of hash function (or function) that group certain words (based on character type) together and store them on disk, while tracking the file location for later. For example, for word x: function (x) => y, where y is some key. Then y mod 400(say we want 400 chunks) = y.txt. Next read each of these files in memory one at a time, build a Trie for each (or use a HashSet<String>) to store and compare anagram.
2. If we can access a distributed computing system, using the same method as above to break the file into the 400 chunks and send each of these chunks to different nodes to compute for anagrams. Similar to Hadoop map/reduing operations.