Coding Assignment

Attempt 1

Since we must iterate through the documents token-wise to find all the suffixes, thus the tokenizer can be used to divide the *original* documents into many tokens, and then used the start index of each token to signify the start of a suffix in the suffix array. Each `Suffix` object thus has 3 variables:

- `entry` (an integer storing the docID of the document that the suffix belongs to)
- `offset` (an integer storing the start index of the suffix)
- a string pointer storing the entire "suffix" itself (this is used only for the query, not for the suffixes)

In total, each suffix in the suffix array only uses 12 bytes of memory. For more details, see the codes in `attempt1` directory. To sort the suffix array, I wrote a comparator class that can compare 2 suffixes lexicographically. For each suffix to be compared, I used the `entry-offset` information to extract from the *original* document the entire suffix as a string, and then compare the strings directly against each other with the built-in method `String.compareTo()`. This works well for most cases, but doesn't work if the character sequence in the query phrase is slightly different from the original document. For example, the query "`low speed`" can't be matched with the document phrase "`low\nspeed`" (as already explained here: https://piazza.com/class/iyn035m9zih56d?cid=10). This led me to my second attempt.

Attempt 2

I used the normalization technique here to normalize both the documents and the queries. This resolves the issues in the first attempt because the data will be normalized such that the new line (`\n`), the tab (`\t`), or capitalization will have no effects on the search (note that we can also specify more normalization rules if we want, to make the search more "tolerant"). The idea is to normalize the original documents, and then only allow the `PhraseSearchEngine` and the `SuffixArray` to work on the *normalized* data. This means that our search engine must be able to store not only the `originalData`, but also the `normalizedData` as well.

Storing the normalized data is very easy. All we need is an additional string array (`normalizedDocs[]`) inside the `SuffixArray` class, where the string at index i will represent the normalized data of the document with id $= i$. We use a normalizer (the static `SuffixArray.normalize()` method) to normalize the documents and then put the normalized data into the string array. Note that I didn't use the `BrainDeadNormalizer` in the prekode for this task, simply because I didn't want to mess up any of the provided codes. Besides, I want all the implementations to be self-contained within the `SuffixArray` class.

Using the method described here, then on average, it takes around 1.2 ms to query the `cran.xml` file, and around 0.2 ms for `wescience.txt`, which is quite fast. The memory required for the suffix array can be calculated by the formula: $(M + 12n)$ bytes. Here, M is the total number of bytes required to store the normalized data (which is a little bit less that the data in the original documents). We use n to denote the total number of suffixes in the documents (which is also equal to the number of tokens). And since each suffix requires 12 bytes, so we must have $12n$ in the formula.

Paper Assignment

Exercise 1

The answer is: `ng$s*` (where `$` means the end of a word). In other words, we search for a permuterm that has the prefix `ng$s`.

Exercise 2

	ε	A	L	I	C	E
ε	**0**	**1**	**2**	**3**	**4**	**5**
P	**1**	(1,0,1) **1**	(1,1,2) **2**	(2,2,3) **3**	(3,3,4) **4**	(4,4,5) **5**
A	**2**	(2,1,1) **1**	(1,1,2) **2**	(2,2,3) **3**	(3,3,4) **4**	(4,4,5) **5**
R	**3**	(3,2,1) **2**	(2,1,2) **2**	(2,2,3) **3**	(3,3,4) **4**	(4,4,5) **5**
I	**4**	(4,3,2) **3**	(3,2,2) **3**	(3,2,3) **2**	(2,3,4) **3**	(3,4,5) **4**
S	**5**	(5,4,3) **4**	(4,3,3) **4**	(4,3,2) **3**	(3,2,3) **3**	(3,3,4) **4**

Notations: Each cell in the matrix above has 4 numbers. The boldface number is d, which is the edit distance for that particular cell. The 3 numbers within the parentheses are x, y, and z, which respectively represent the edit distance of the cell on the left, on the diagonal, and on the top of the current cell. Note that the cells for the empty string (ε) only have the edit distance d, but not x, y, and z. Also, let $s = s_1s_2s_3s_4s_5 = paris$, and $t = t_1t_2t_3t_4t_5 = alice$ (where each of the s_i and t_i represents a single character in each of the 2 strings).

Algorithm: For each cell $[i, j]$, we compute the edit distance as follow:

- If $s_i = t_j$, then $d = \min(x+1,\ y,\ z+1)$
- If $s_i \neq t_j$, then $d = \min(x+1,\ y+1,\ z+1)$

Exercise 3

a) Since it's not mentioned that Kurt needs a tolerant retrieval system, so we can actually use a hash table here to represent the data-structure. A hash table can only find exact match, but it is very efficient when used for searching because it takes $O(1)$ time for the hash function to perform the search. However, if Kurt wants a tolerant search engine (i.e. an engine that can handle wildcard query or prefix/suffix search), then we can use a suffix array and the binary search (exactly like in the coding exercise). Alternatively, we can use a trie, which is very similar to a finite state machine, and hence we can construct a simple finite automaton to use for searching.

b) Here we can use a trie as the datastructure for the dictionary. To perform the search, we are going to walk through the trie, while computing the edit table (Levenshtein distance) at the same time. The edit table is computed using dynamic programming, and we will compute each column at a time. The entire algorithm is described in the paper: http://www.uio.no/studier/emner/matnat/ifi/INF3800/v15/slides/tries_for_approximate_string_matching.pdf. Basically, the idea is to have the pattern string (i.e. the query string w) on the vertical axis of the edit table, and the target string (i.e. the matched result) on the horizontal axis, such that each character in the target string occupies a single column in the table. Since all strings below a node in the trie share the same prefix, so it must share the same column in the edit table. This means that every time we go down a node, we have to compute the edit distances for that column at the same time. If the edit distances grow bigger than k, then we reject that computational branch. Otherwise, we continue that branch until we reach the leaf node (where a matched string will be returned). So basically, the trie represents the search space (i.e. the dictionary itself), and the algorithm is to use the information from the columns of the edit table to cut down the branches of the trie as soon as possible in order to reduce the search space.

Exercise 4

a) All the entries in the permuterm index generated by *sting* are: *sting$*, *ting$s*, *ing$st*, *ng$sti*, and *g$stin*.

b) Binary search can be used to search the suffix array. The suffix array for *mississippi* is:

Index	**Suffix**
10	i
7	ippi
4	issippi
1	ississippi
0	mississippi
9	pi
8	ppi
6	sippi
3	sissippi
5	ssippi
2	ssissippi

Assignment B

Exercise 1:

If you wanted to search for `s*ng` in a permuterm wildcard index, what key(s) would one do the lookup on?

(Exercise 3.3, page 56)

Exercise 2:

Compute the edit distance between `paris` and `alice`. Write down the `5x5` array of distances between all prefixes as computed by the algorithm in Figure 3.5.

(Exercise 3.6, page 62)

Exercise 3:

String matching algorithms

a) Kurt has a large dictionary with millions of elements. As part of a document processing system, he wants to develop a module which can efficiently detect whether a word in the document is also present in the dictionary. Describe (i) what kind of data structure Kurt should use to represent his dictionary, and (ii) what kind of algorithm should be used to perform the search.

b) Kurt also has a big dictionary containing the surface forms of the most common words in Norwegian. As part of a spellchecking application, Kurt wants to be able to query the dictionary with a word *w*, and get back the set of all words which have an edit distance of at most *k* from *w*. More formally, the resulting set is thus defined as *{w' : editdistance(w' ,w) <= k }*. You can assume that the maximum distance *k* is small. Describe how Kurt should represent his dictionary and perform the search.

Exercise 4: (Optional for INF3800)

Permuterm indexes & Suffix arrays

a) Write down the entries in a permuterm index that are generated by the term `sting`. If you wanted to search for `s*ng` in a permuterm wildcard index, what key(s) would one do the lookup on?

b) Consider the term `mississippi`. Write down the suffix array for this term, and explain how you can use this to efficiently locate all occurrences of the substring is.

(This exercise is taken from the V14 final exam)