Augmentative Report for Programming Assignment #4

Susung Park (2014311254)

Data Structures, Class #41

December 16, 2018

1 How to Use

Compiling Compiling the project is simple. Move to the base directory of the project, and then type `gcc main.c -o main` in the console.

Prerequisites Before actually running the compiled program, proper input files should be given. Since this assignment receives two input files, both input files should be properly configured. Name of the dictionary file should be `dict.txt` and that of input text file should be `hw4_input.txt`. Both files should be located in `./io/`.

Executing Program After compilation is finished (which will take less than 1 sec), simply type `./main`. Before executing check if the prerequisites are properly configured.

Outputs Outputs of the program is compliant to the requirements mentioned in the assignment PDF file. The output file is given the name `hw4_output`.

2 Algorithms

Since both the dictionary and input text file is likely to be referenced frequently, the whole content of both files are stored in RAM to reduce IO operations.

2.1 Reading the Dictionary

Methodologies Reading the dictionary is simple; the program reads the text character-by-character, until it reaches EOF, while parsing each word when met non-alphanumeric character. This way, every individual word inside the dictionary file can be splitted into single word. Then, each word is packed into a linked-list structure named `DictElem`, which has the pointer that points to the word string, another pointer that points to next element, and an integer that stores frequency information as its member.

Complexity Since this operation is executed sequentially over the whole dictionary file, time complexity of this operation would be $O(n)$, having n as the total number of characters in the dictionary file.

2.2 Reading the Text

Methodologies Using the same strategy mentioned in **2.1**, the input text can be splitted into words, simultaneously removing non-alphanumeric delimiters, such as parentheses, commas, and colons. The splitted words are stored in a linked-list structure named `TextWord`, which has a pointer to the word string and another pointer to next element as its member.

Complexity Since this operation is executed sequentially over the whole input text file, time complexity of this operation would be $O(m)$, having m as the total number of characters in the dictionary file.

2.3 Verifying Containment of Dictionary Word

Methodologies For verifying the containment of dictionary word inside the input text, `strstr()` function could be a good candidate of usage, but since we have to be case-insensitive, this function can't be used. Therefore, another custom function, named as `strinclude(char* str1, char* str2)` is defined. This function checks if second string is included in first string, being case-insensitive. The function compares two strings in a *convolution* form.

Complexity Having the length of first string l_1 and second string l_2 respectively, $l_1 > l_2$ should be satisfied. Under this, testing each element will require $\frac{l_2}{2}$ comparisons in average. If the dictionary and input text file have number of characters as n_1 and n_2 each, and the average length of each individual word is k, we can write $l_1 \approx \frac{n_1}{k}$ and $l_2 \approx \frac{n_2}{k}$. Then, we can find the total time complexity as follows.

$$O(n_1 n_2 \frac{n_2}{2}) = O(n_1 n_2^2) \tag{2.1}$$

Therefore, given the number of words similar in the dictionary and input text file, we find that this algorithm shows cubic time complexity.

2.4 Extracting Words with Nonzero-Frequency

Methodologies After verifying containment and counting the frequencies of each word, they had to be sorted. This was implemented by insertion sorting. The program scanned through the dictionary, detected words with nonzero frequencies, and then added them into a list of nonzero-frequency words. Insertion sort was conducted while inserting each word into the list, having the key as the frequency of each word.

Going one more step, we had to sort this not only by word frequency, but also by ASCII-code-order. Therefore, after the *pushing* operation of insertion sort was over, additional pushing operation by ASCII-order was performed, while the frequency of word did not change.

Complexity Since the algorithm includes insertion sorting, time complexity of this part is $O(n^2)$, having n as the number of nonzero-frequency words.

3 Benchmarks

Dictionary with 10,000 words and input text with 100KB size took approximately 5 seconds to analyze. Performance and specifications of the computer used for this benchmark is shown below.

1. Intel ® i7-6700HQ, 16GB DDR3 RAM
2. Ubuntu 16.04, 64-bit
3. No optimization options enabled in GCC compiler