David Terkula

Time Complexity Analysis Project 4

12/5/16

Professor Lewandowski

The ReadAndHash() method and the PrintTable() method

These are the two methods used in the FileHandler.cpp class which are used to read, hash, write to external file, and output to the console. The getline() method is used to grab input, which according to the c++ reference page for it, has a time complexity of generally linear, or roughly the length of the string. This makes sense as each char has to be read and stored, which is *n time.* Then, a while loop is executed, and it stops running when the last line of the file is read – therefore each line is read in. inside the while loop, each window is hashed, which runs in approximately *n* time as while because in the Hash.cpp class, a for loop is executed over the length of the string to be hashed, and arithmetic and assignment are executed at constant time not significantly altering the time complexity. This part of the program must iterate over each char of the file twice, one time to read and one time to hash. Therefore, the theoretical time complexity of this algorithm to this point should be θ(n) where n is the number of bytes or characters. In the read and hash method, the standardizeText method also considers each character in n time as it must reread the char of the window to remove whitespaces and to standardize the case of the text to lower case to catch simple attempts at fooling the program. Outputting to the file is also unspecified, but should be done generally in *n* time. Printing the data to the console uses two loops over the entire map of hash values and loops in order to find the hash value from the next line in order and to print it. This should take no more than n^2 time depending on the number of duplicate hashes or collisions or entries in the map, so the order of magnitude of this operation is O(n^2) and is certainly omega of n or lower bounded by n. Therefore, the time complexity of just reading, hashing and outputting to a file should be θ(n), but the printing to the console in order could take n^2 time. As a whole the algorithm or the collection of these algorithms is omega(n), and O(n^2). (really the algorithm is more accurate to say it is lower bounded by some factor of n because it looks at each character multiple times over the course of the algorithm but for generalities sake the algorithm is still omega n.

Part 2, comparingFiles

The comparing files method first calls the readDataFile method which has a similar time complexity of the previous readAndHash method, but it should be theoretically lower because it doesn’t do any hashing. Therefore, the readDataFile method should also be θ(n). The compareFile method then uses a for loop over the size of the recreated map of hashvals and vector of ints representing the start lines. This for loop considers each entry, representing and entry of each line if no duplicates, so this will be done in θ(n) time as well. The printing of the file comparisons is simpler than the one used in the first part and it runs in θ(n) time as it just iterates over the map and prints the data. therefore the time complexity of this part is θ(n), assuming θ(n) time from reading the data.

Real execution time.

In order to test to see if the theoretical time for the algorithms was accurate, the size of the file in bytes and the execution time of the program over various file sizes was recorded for each part of the program. For the reading and hashing, and the file comparison, 8 file sizes were tested and measured for their execution time in milliseconds. The program was run 10 times for each file size and the average run time was calculated. This average time was plotted on a scatter plot vs the file size of the files. A line of best fit was then created to judge the relative running time or shape of the line representing the total time running the algorithms. The two graphs for each part are available to see on this repository as well. The running times of both parts of the program appear to be linear in nature or on the order of θ(n) time as hypothesized. Both graphs have a correlation coefficient of atleast .99 meaning that over 99% of the variance between file size and run time can explained in a linear, θ(n), relationship. To validate the usage of the correlation coefficient, the residuals between the points and the fitted line must be observed. By looking at the graph one can see a very tight fit of the data to the regression line, suggesting that a residual plot would show very little difference to the fitted regression line; therefore, the use and interpretation of the correlation coefficient and the linear regression live appears valid.

Between the two graphs, one can see also that the run time of the file comparison was less than the run time for the reading and hashing, given a similar number of bytes per file tested. Therefore, the part 1 was correctly theorized as taking longer despite still generally being θ(n) as well. In reality it may be θ(4n) just as a guess, but still generally falls under θ(n). Therefore, using this test of runtime, it appears the time complexity of both parts is θ(n).