Eza Rasheed
er6qt
03-06-19
postlab6.pdf

BIG THETA RUNNING TIME:

The Big-Theta running time of my application for the word-search portion of my program was linear. This is because the running time depends on the number of words in the list that the for loop is iterating through. In order to go through my quad loop and search(find) the desired word, my program searched through a list of strings from the beginning to the end, linearly looking at each element one by one, to see where the target word was, and then inserted the word into the hash table. Looking at the quad-nested for loops, two of the for loops looped through constants, d(direction) and l(length), as they had a set range, while the loops for the number of rows(r) and number of columns(c) varied in the grid. There was a find call within these nested for loops suggesting that there is another for loop from the find function in the double nested for loop. Therefore, it can be seen that these three for loops are significant to the finding/word-search portion of the dictionary, meaning the Big-Theta running time would be r*c*w, where r is rows, c is columns, and w is the find function looking for the target word. Running time is affected by loops that affect one another and are independent. Therefore, this would be the equation for big-theta as the other loops can be viewed as insignificant to the running time.

TIMING INFORMATION:

With my original hash function, without any optimizations made to it, when I compiled my program using the -O2 flag and executed the input files words2.txt and 300x300.grid.txt, my

implementation speed was 9.82653 seconds. After making my optimizations, I picked a new hash function which used pow to make performance worse in terms of time. My time after my optimizations was 2.09093 seconds, but when I used pow (as can be viewed in my commented-out code) I got it to run in 9.70791 seconds. Calculating to the power is slow, takes a longer time to compute, and is not very stable. It also "takes out the logarithms" (Bloomfield). It is much more complex and time-consuming than the basic arithmetic operators. Another hash function that I know would cause problems time-wise but did not actually get to implement would be if I set my value to start at the [0]th position. This is because it would only look at the first character in the string word and many of the same results would appear for the checking on that one character. This would then to many collisions. When I picked a new hash table size to make performance worse, I was told to try setting my table size to 0. Upon doing this, I did, as a matter of fact, observe a slow speed. My result came to 57.9612 seconds, a HUGE difference. I hypothesize that this is because the table size would become very large, and so when you would try finding the target word, you would have to linearly check/search each spot for the elements until it is found. This, ofcourse, increases the amount of time spent doing word searching. The files I used to test these modifications were "words2.txt" with "300x300.grid.txt," and the machine I used to run these tests was my personal laptop, HP Spectre x360, using VirtualBox.

OPTIMIZATIONS:

For my optimizations, I modified my hash function, stored the max word length (22) to a variable, and buffered my input. All my optimizations were executed by using words2.txt with 300x300.grid.txt, and this was all tested on my HP Spectre x360 laptop.

1ST OPTIMIZATION:

The first optimization I made was to modify my already fast hash function. I multiplied my unsigned int by 37 and added it with each successive digit. It is quite controversial that by multiplying by 37, you are actually slowing down the speed because you are taking up more space. But because I implemented 37 without exponentiating (using pow), which slows down speed immensely, and just used the basic multiplication and addition operators with it, my speed went from **9.82653** seconds to **8.89331** seconds. 37 is a prime number which "sets really quickly" and is a "quick lookup." Although a small change, it did improve my performance.

2ND OPTIMIZATION:

The second optimization I used which most significantly impacted my speed positively was storing the max word length into its own variable and having my hash table point to it. Storing the max size of a word rather than hardcoding in 22 saved time because this way, there was not repetitive searching going on for words that were longer than the longest word that could be found in the separate dictionary files; words2.txt's max word length was less than 22. I did this by making a public variable called "max_size" in my hashTable file. I then implemented the actual modification for max word size in my insert function in which I used an if statement to check if the length of the word being passed in was greater than the max size, then set the max size to be the length of the word being passed in(inserted). I then went to my main method, and in my quad nested for loop for checking the word length, instead of iterating from 3 to 22, I iterated from 3 to the max size of the word being searched. This lowered my speed from **8.89331** seconds to **4.68586** seconds. Evidently, this had a great impact on runtime speed.

3RD OPTIMIZATION:

My last successful optimization was buffering my input, storing it in memory in a data structure, and then printing it out after the timing code (word searching) was finished. I had numerous print statements within my quad loop as I was printing out the format of the outputs after I checked each direction (there were 8). This greatly slows down run time speed because the program prints out every time a case is "matched". I encountered quite a bit of problems implementing this. After continuous searching of how I would store all my outputs at the end, I found using stringstream to be the best option. This allows you to read from string as if it were cin statements. So I made a variable called stringstream output, and then made my 2 different cout statements (one for the double word directions (ex. SE), and one for the single word directions (ex. S)) using output instead of cout. I then assigned a variable to store my output, in which I used str() on my output variable to get and set my string objects that were in my "stream." Lastly, I printed out that variable after the timer ended. Initially, when I ran my two input files, the number of words found and the faster speed made me think that my buffering worked properly. However, when I went to go compare my output to the provided output, it stopped matching. It was most likely due to my 2 different cout statements that made it confused. After calling over a TA and not being able to figure out what was wrong, we decided to not do stringstream and just do the concatenation of strings. We initialized a boolean to true that would be false whenever a double-worded direction was checked. Then we just made an if and else statement that would check which format to print depending on the Boolean condition, stored it in a variable, and printed it out after the timer stopped. When I compiled and ran it again though, my format was all messed up; it was not in vertical format. After about 10 minutes, I embarrassingly found out that all I had to do was add "\n" to put each word found on a new line.

That took care of my two different outputs and allowed for no difference to be found between my output and the test output. My speed went from **4.68586** seconds to **2.09093** seconds. Although I was hoping for it to improve my performance more than just by 2 seconds since I was not continuously printing over and over again, it still made a difference.

FAILED OPTIMIZATION:

I just want to quickly mention how I tried to make a separate hash table to store my prefixes to immensely improve my run time performance but it failed. I created a for loop to add the substrings of the word being passed in from the dictionary to my separate prefix hash table in my insert function. I then went into my quad loop to check if a prefix in my prefix hash table was found, and if so, print the word without looking further. It all made sense in my head, but was quite tricky to implement throughout my code. I was getting too many errors, and I honestly do not know where or what I did wrong. Sadly, it did not work, but I tried for a very long time.

OVERALL SPEEDUP:

In conclusion, before my optimizations, my program ran in **9.82653** seconds, but after my optimizations, it ran at **2.09093**. Thus, the overall speedup of my optimizations was 4.6995977866, so nearly 5.