Gabriel Groover (gtg3vv)

CS2150 lab 101

The big theta can be calculated to be rows*columns. The timed portion of my code consists of a quad nested for loop that calls numerous functions in the innermost loop. The first loop must loop through all rows of the grid for any number of rows, r. The second loop must do the same for any number of columns. The inner two loops are approximately constant. The max word length is both finite and small, having no impact on the growth rate. The number of directions is a constant also not a factor. All of the functions called within the innermost loop are constant time operations, with the exception of find. Appending to list, hashing, getting word in table and inserting are all independent of number of words. For find however, in the best case, the growth rate is constant time with increased dictionary size. In the worst case it is linear. This means that big theta falls between rows*columns and rows*columns*words. None of my optimizations affected the find cases or the first two for loops and the big theta remained the same.

For this section, all values will be referring to the files words.txt and 300x300.grid.txt. The average time for my initial application was 6.903s. I performed this test on the virtual box on my personal machine, but achieved much better values on the in lab computers. I discuss hash functions effect on performance in detail below, but by way of example I tested an extremely simple hash function. My function was the value of the first letter cubed and modulus the size of the hash table. The average running time increased to 7.904s. This is because the hash is depended on only the first letter of the word, so there will be only 26 values of the hash table used. The amount of collisions causes the find method to take longer. For a hash function this poor, a smaller table would not greatly affect performance because there is so little diversity already. Using my original hash function (still not a good one) and a table size of 100, the average time increased to 8.850s. By decreasing the size of the table so much, we greatly increase the chance of collisions.

My average running time before any optimizations, including the –O2 flag, was 6.338. Simply adding the –O2 flag increased the average running time to 3.460s. (before any coding optimization) Note that these times are after running the code on words2.txt and 300x300.grid.txt. The times that I achieved seemed to be substantially faster than what others had as their initial times. I believe that this is because the data structure I used for my Hash table was among the more optimal. I used separate chaining to avoid having to linearly probe the list to input values. It was clear that the main issue limiting my running time was the hash function. In my original program, my hash function was designed to multiply the first couple characters in the word by the length of the string. The problem with this hash is that while it can produce large numbers, it does not produce a wide spread. I tried multiple times to vary the formula in order, but because the number of words was so large, I had 90% of my hash values below 1000. The vast majority of my values in the hash table were empty. This meant that as more values were inserted, there were many collisions and the growth rate of find was

close to linear. I used a more standard string hash function in an attempt to widen the distribution. For each value in a given word, I multiplied by a prime number and added the letter to it. This allows for very large hash values before the modulus is applied, but also extremely varied values. The difficulty I had here was finding a hash function complex enough to be diverse, but simple enough to not negatively affect the running time. After implementing the new hash function, my average running time was 1.890s. This was an improvement of 1.83x from the previous. The size of the chains was one of my largest speed problems.

In order to further increase the running time and decrease the size of the chains I decided to increase the size of the hash table. I had previously been using a load factor of .75 to determine a potential hash size, and then using the next highest prime as the final dimension of the table. I tried using different load factors to see how it affected performance. I used the same process as above with an initial load factor .5. This greatly increased the hash size and, while much less memory efficient, did increase the running time. My average running time with this change was 1.776s. This did not have quite as much of an impact as I was hoping for, but is still a 1.064x improvement. If I had attempted to make this change without changing the hash function as above, I do not believe my running time would have changed at all.

The final change that I attempted to make to improve running the time was printing out a cumulative statement as recommended in the lab. Calling cout over and over again can require a large amount of time. After completing each find in the hash table, I compiled a vector containing the string for that line. This allowed me to print out all of the strings from the known vector outside of the timer. The actual application of the hash table would, in theory require less time. I had a lot of trouble with this section of the code finding a way to combine all of my variables into a single string. I originally wanted to use to_string to convert my row and column values to a string, but do to a compiler bug in this version, I was forced to use a osstream to make the conversion. After making the conversion and adding several values to the string and putting it into the vector, the process seemed to be about as taxing as printing out the strings individually. I think this may have been due to my use of the osstream in particular. My final running time with this optimization was 1.766. The tiny difference between this one and the previous could just be due to small fluctuations. I am not confident the change made the program faster at all. My other changes were successful however, and overall I had a speed increase of 1.96x. (not including the change with –O2) If we do include the difference with –O2 we find an improvement of 3.59x.