Rakshith Raghu(rr5de) 20/25/2018

Postlab

First, a clarification. I printed the results separately from the search portion of the application. In full details, I stored the discovered words in the correct format in a string vector. This vector was then printed using “cout” after the search and cleared. Now, the big O run time for my method would be O(r*c*w). However, my algorithm would on average be more efficient than this which I will explain later in this report. This is because there are 4 nested for loops in the implementation. 1 of the for loops is constant 8 everytime and so doesn’t matter. The last for loop, which looks at the length of the word, is of length between 3 to 22. It is dynamic in length but small, so disregarded. The rows and columns are the middle for loops and it runs through all of the columns and rows. Now my dictionary is of the size is the size of the amount of words in the list converted to the next prime. In the worst case, every word would produce the same key, which would see every word stored in the same bucket (which I used linked lists for). This meant that the search for whether the string matches a word in the dictionary could take linear time in the worst case. However, in most cases, it took much less than that. My comparison was to take a string, get a value of its elements, and use that value as the key. If the string was in that look up bucket, than it was “found.” In most cases, this would be of max 1 or 2 searches rather than a linear search of the whole list. My method could be optimized further by using a balanced tree structure. This would decrease w to log w which is how much time it takes to travel down the list.

In general, I believe my initial implementation was badly done. The first action that probably extended my time in the initial set up was the look up of words in the dictionary. This took linear time as I used linear collision to deal with conflicts. In the inlab tests, my method thus took: 4588.91 seconds to run (nearly an hour).I also printed out the result each time I found a word. The method also overestimated the amount of words found. It found 33k words in the 250x250 grid and the initial words txt. This is because the edges of the grid would still check for words between sizes 3 and 22 and produce the same string for those various sizes when the check went past the grid. The initial code finally did not create a hashtable of a prime size, instead creating a hashtable the size of the list of words.

I fixed the overestimation by adding a switch inside the 3rd for loop that checked edges and changed the max potential size of a string based on closeness to edge. I changed the collision strategy from linear collision (which would likely place numerous words out of order) to buckets. The bucket of choice a linked list, which might be less optimal than a balanced binary search tree. This meant that looking a word no longer required a linear search in most cases. I could just take a string, convert it into a number, and use that number as a key (well the modulus of that number). If the key showed that exact same string in the dictionary, then it was found. If not, then that string was rejected. This served to rapidly increase the speed of my code. Finally, I changed my output to instead store found words in a string vector which would output its elements at the end. For the makefile, I added the -02 flag.

Using the 250x250 grid and the words txt file, this produced 27574 words in both times I ran the file. In an incredible increase from the 4588.91 seconds , it ran in 4.5 seconds. I further optimized my code by changing the hashtable size to prime numbers. If the size of the txt of words was not prime, it would find the next prime number. This decreased the time it ran to 4.33 seconds. Dividing 4588.9 by 4.33 produced a decrease by a factor of 1000 times. I believe the main 2 increases in efficiency came from the change from linear collision to linked-list buckets, and the change in output from on the spot, to all at the end. If I had more time, I would attempt to further optimize my code by changing from linked list to balanced binary trees and by changing the load factor of the hashtable.