Christian Kinzer

Cfk5ax

10/25/18

9:30 AM Lab

Postlab6.pdf

Big Theta Runtime

The big theta runtime of finding all of the words in a word puzzle was rows * columns * 8 directions * length of the longest word in the dictionary file. This reduces to rows * columns, as the other factors were small constants. This is because of the four nested for loops, with the first iterating over the number of rows, the second by the number of columns, the third over the 8 directions, and the fourth over the length of the longest word in the dictionary. The big theta runtime is completely independent of the number of words to be found in the word puzzle, as the program checks every possible string in the puzzle that could be a word. Thus, the big theta runtime is mainly dependent on the size of the grid file. While the big theta runtime of adding the dictionary to the hashTable was linear, since the program goes through every line of the dictionary file once, the big theta runtime of actually solving the word puzzle was rows * columns.

Timing Information

The following times were collected using averagetime.sh on virtual box on a Lenovo thinkpad.

	250x250 words.txt	300x300 words2.txt
Original program	5.161 seconds	3.418 seconds
Bad hash	12.571 seconds	8.624 seconds
Bad table size	15.634 seconds	11.295 seconds

The bad hash function I used created a hash value by summing all the ascii values for the characters in a string. This increased collisions dramatically, as strings of the same length were likely to give similar hashes. For the times under bad table size, I decided to remove the prime number part of resizing the hash table. This reduced the size of the table quite significantly and resulted in a very small table with a high number of collisions. After seeing the practical effects of a bad hash function and table, I now have a better understanding of why it's important to have good ones and what makes a good hash value or table.

Optimizations

While starting this postlab, I began by looking through my code from the prelab and I noticed that I had hard coded the for loop that iterates over the lengths of potential words. I had originally set this loop to just run from 0 to 22, and I realized that words2.txt has a different longest string than words.txt. I realized I was wasting iterations, and though this was only a constant for loop, I decided to change it to iterate from 0 to the length of the longest word in the hash table. I did this by creating a public field for the hash table, an int called longest. In my insert function, if the length of the string being inserted was greater than longest, longest would then update to the length of the string inserted. Then, in my for loop, I iterated from 0 to table.longest. As I had assumed, this had no major effect on runtimes with words.txt, however, on 300x300 with words2.txt, my average speed increased from 6.919 to 3.237 seconds, for a speedup of 2.13.

When I was using averagetime.sh to test my program, I noticed that my speeds were consistently lower than my speeds given by using a.out. I decided that this was probably due to the fact that averagetime.sh doesn't print much of anything to the screen, whereas using a.out prints the grid and every single word found in the grid. So, instead of having a cout line for each word found, I changed my code to hold a string called out, and whenever a word was found, I would concatenate that string with the output line and the new line operator, \n. Then, after the for loops had run, I would cout and print the whole string to the screen at once, as opposed to calling cout over and over and over again. On 300x300 using words.txt, my speeds increased from 11.484 to 7.924, for a total speedup of 1.45.

At a certain point, I wanted to see the effect of the -O2 flag on my final code. I remember that on my first semi-working build, -O2 helped cut my speed in half on 50x50 with words.txt. So, in order to see what the final speedup was, I removed the -O2 flag from the makefile, recompiled with make, and recorded a time of 7.486 seconds on 300x300 with words.txt. Then, I added the -O2 flag back into the makefile, recompiled, and recorded a time of 7.416 seconds. This was kind of surprising to me, I had definitely expected a similar speedup to that of what I remembered from earlier, only to find the speedup to be 1.01. This result makes me wonder what exactly the -O2 flag does, and how much the speedup depends on the input size. I imagine that with a smaller input size, such as the 50x50 mentioned earlier, the -O2 flag has a much more significant speedup.

I definitely encountered many problems with optimizing my code. Firstly, I tried a multitude of different hash functions to speed up my program. I had known in the prelab not to use the math.pow() function, so I had an array filed with powers of 37. I would then multiply the ascii value of a character by 37 to the power of its index, then sum over all the characters, then modulo by the table size. This was the first working hash function I had tried. Despite trying many others after, nothing ran any faster than my first hash function, which made me feel lucky to have a good first hash function and also made me feel that I wasted a good amount of time trying to find a better one. When I decided to concatenate an output string instead of printing to the screen over and over, I ran into the issue of concatenating and int, such as the row or the column of the first letter of a word, to the output string. There is no good way to do this in C++, which is frankly surprising. I had hoped that it would've just worked with +, but instead I had to use to_string() and change my CXXFLAGS in the makefile to -Wall -O2 -std=c++11. This took me way too long to figure out, and was honestly a little embarrassing. Overall, I don't think this lab was necessarily as hard as everyone anticipates it being. Once you actually get a working build of the prelab, everything else is fairly straightforward. I definitely feel as though I have a much better understanding of hash tables and hash functions now that I have completed this lab.