postlab6.pdf
Patrick Anderson psa5dg
Lab 105

Big theta: (rcw)^8

Reading in input files takes constant time, as well as initializing the variables. Adding the words into the hash table takes linear time, since I'm going one-by-one through the file. Same with counting how many words are in the file. Checking to see if a combination of characters is where the meat of the program is. It has to go through all the rows and all the columns, and has to potentially go through all of the words to find the given word in the worst case. Furthermore, it has to do this in all 8 directions.

Original Application

All on my MacBook Air	250x250	300x300
Original Application	1293634 ms	136272 ms
Optimized Application	5151 ms	6842 ms
Worse Hash Function	1040060 ms	46706 ms
Worse Hash Table Size	1207530 ms	36705 ms

With my new hash function, I multiplied each character in the string by a larger prime number than before. I did this really just by trial and error. I tried various prime numbers and got much better results with smaller numbers. I assume that my performance was worse because of more collisions resulting from the hash function, or that the numbers were so big that the modulo caused them to group together. The table size I chose that was worse was a table size that was just big enough to fit all the words. This is terrible, because there are going to be lots of collisions.

The times that resulted are shown in the table above for the optimized application. For 250x250, I sped it up to a factor of roughly 251.1, and for the 300x300, 19.9. I used many of the suggested optimizations.

The first thing I realized was that my table size was, in fact, too small. It was just big enough to fit all the values. I proceeded to make the table size 5 times bigger, since the extra memory really wasn't a problem.

Next, I modified my linear probing, turning it into quadratic probing. I assumed that this might help reduce clusters of filled spots that linear probing would create. I also modified my find function, which was atrociously slow, to stop searching once it hit a vacant spot, as opposed to searching the entire list, as I had been doing before. These seriously increased the speed, more so than most of the other optimizations.

I also changed my hash function a little bit. I changed the prime number the ASCII code was being multiplied by from 37 to 7, which seemed to also help reduce the time.

Finally, I learned a new tool in C++, the sstream. What I did was instead of immediately printing to the screen, I stored the string into a vector using sstream. Then, when the timer was stopped, I printed out the results. This didn't help too much, but it certainly helped shave off a couple seconds.

Working with sstream was difficult at first, but after some trial and error I got the hang of it. I also had to make sure to utilize the modulo to prevent my quadratic probing to not go out of the bounds of the hash table, something that I had not done before. I really liked the idea of quadratic probing over linear probing, since I believed that it would cause less collisions, which it seemed to definitely do. I thought that maybe a larger prime number in my hash function would actually make my function better, but the opposite was the case, which surprised me when trying out multiple scenarios.

One big takeaway for me is that less lines of code doesn't always yield more efficient results. Normally, I try to write as simplistic code as possible, because for the most part of these labs, less code does tend to point out that I understand the logic better conceptually. However in this case, where time efficiency really mattered, I had to add a few more lines of code that made it a bit more complicated in order to really make my program run fast.