Ethan Trinh

Lab 107

10/20/16

Post Lab 6

Big Theta

The big theta of the application remained the same even after the optimizations. Worse case scenario, the running time will be r*c*w. There is a chance, if there is a poor hash function, that all of the words hash to the same bucket, and therefor the quadratic probing would have to search through every previously added word, creating a linear runtime. The reason it stays the same after optimizations is because while the optimized program runs faster for the same n, the runtimes will both grow at the same rate. This would remain true for any sort of optimizations and any sort of hashtable, including double hashing, linear probing, and separate chaining with different buckets. While on average some may perform better than one another, all of them have the worst case scenario, and that is if they all hash to the same bucket.

Timing

For my original program, which used separate chaining with lists as buckets, the runtime on my laptop for the 250x250 grid was 26 seconds, and for the 300x300 grid it was about 28 seconds. My optimized program runs on average at 4.822 seconds for the 300x300 grid with words2.txt as the dictionary, and 9.482 seconds for the 250x250 grid with words.txt as the dictionary. For a worse hash function, I chose to use pow(str.at(i)*7,i+1), which is essentially the ascii of the character times 7 to the power of the location of the character in the string plus 1. This is then added to the current hash, which starts at 0. The average runtime for this with the words2.txt for the dictionary and the 300x300 grid is 18.611 seconds and for the words.txt and 250x250 grid the runtime is 13.149 seconds. The reason that this is so much slower is because of the calculation time. Every time the hash function is called, it takes more time to calculate the hash and therefore longer to find the word you are searching for. More complicated functions, while reducing collisions, come at a cost because of the computing time for the hashes. To test a worse hash table size, I used my optimized code but increased my load factor from .25 to .75. The runtime for the 250x250 grid is 14.5 seconds and for the 300x300 grid 6.65 seconds. This hash table size is essentially 4 times smaller than for my original load factor. While this is more memory efficient, the performance is worse. Smaller hash tables tend to have more collisions due to the fact that there are less buckets and therefore the mod is smaller. Each hash is more likely to hash to the same hash as a previous word, and therefore they will have to move on to a different bucket.

Optimizations

My original application used separate chaining with lists, had a load factor of .75, and used the hash function mentioned above that was used as an example for worse performance. The run times for the 250x250 and 300x300 grid were on average 26 seconds and 28 seconds respectively. While not the worst results, there was definitely room for improvement. My final run times on laptop, with all the optimizations, were 4.708 and 9.615 seconds for the 300x300 and 250x250 grids respectively. The application was sped up 5.5 times and 2.91 times respectively.

The first step I took to optimize my code was to change my application from separate chaining to linear probing. I initially ran into several errors, most of which were segfault errors due to the vector trying to access elements outside of its range when trying to increase the hash by 1 to prob the next bucket for both the find and insert functions. I fixed this issue by modding the resulting hash every single time to ensure that it was not greater than the size of the vector -1. Linear probing actually greatly worsened the performance of the search. It increased the run times on average by 20-30 seconds from my original program. The run time for the 250x250 grid was about 53 seconds. This was most likely due to the fact that linear probing is susceptible to clustering. Many of the words must have hashed to values that were the same or close in value and quickly many buckets were filled up that were next to each other.

To improve my search time, I switched to quadratic probing instead. This brought my search times down to around 10 seconds for the 300x300 grid, without any further optimizations. Quadratic probing was more effective because clustering is less likely for it. If a bucket is filled, it will check the next bucket, and if that is filled, it will skip the next 3 buckets and check the 4th bucket, 9th bucket, and so on until it finds an available bucket. This reduces the chance that buckets that are next to each other are filled, therefore reducing the collisions, and reducing runtime.

The next step that I took was to modify my hash function. I found that the more complicated the hash function, the longer the run time was for the searches. The cause for this is most likely because more complicated hash functions take a longer time to calculate, and each time the hash function is used in the find and insert functions it reduces the performance of the search. Because of this odd result, I decided to check the collisions of my hash functions in order to confirm that the functions were doing what I wanted them to do. The more complicated hash functions which used the pow function for exponents and had more multiplication against prime numbers and such had fewer collisions, but still performed worse. These functions had collisions at around 60 per find. The simpler hash functions, such as the current one that I used to optimize my code, had 100+ collisions per find but had a quicker run time. Therefore, to optimize my code, despite what I previously believed about collisions and more thorough hash functions, I used a simpler hash function with more collisions but that took less time to execute. This reduced my run time for the 300x300 grid to about 6 seconds on average.

The next step I took was to change the method of outputting to the screen. At the moment, I had my program print the found word and its information in the for loop. Therefore, the search time was worsened from this placement of the code. To improve my program, I decided to store the information for the strings in a list. I encountered a few issues with this initially. I tried using the to_string method for this, but realized that c++ 11 was required for this. Instead, I used sprintf to convert the ints to strings. When I printed the elements after the for loops and after the timer ended, it was only printing half of the elements that I wanted it to. I realized the reason for this was because the for loop used the condition that i was less than the size of the list, but every time I popped an element off the list in order to access the next front element, the size of the list decreased, therefore only printing half of the list elements. I fixed this issue by assigning a variable to the size of the list and using that in the condition. This way, the size of the list used in the for loop was kept constant. This optimization improved my runtime by about .5 seconds on average.

The final optimization that I applied was changing the load factor. The original load factor that I used was .75 because it was recommended in lecture. While this ideal when we are considering the memory usage, the only that mattered in this case was the search time, not the size of the hash table. Therefore, I reduced the load factor to .25, increasing the size of the hash table by 4 times. Reducing the load factor increases the size of the hash table, and therefore decreases the chance of collisions in the hash table. This optimization improved my run times to my final run times.