Justin Choi (jc8mc)
March 15th 2017
CS 2150 – 106, Lab 6: Hashing

postlab6.pdf

For reading the dictionary file and inserting the words that were found from the word puzzle to the hash table is w, which is for words (linear). Then, the reading the grid file is r * c, for rows and columns (two linear multiplied). For l, the word length, it is omitted in this instance since it is a constant. Thus, the big-theta running time of the word puzzle program is r * c * w.

For the times, I ran my program in my laptop (*Macbook Air macOS Sierra Version 10.12.3 Mid 2013 1.3 GHz Intel Core i5, 4GB 1600 MHz DDR3, Intel HD Graphics 5000 1536 MB*) Prior to optimizing my word puzzle and hash table implementation program, for 5 runs, the un-optimized program produced an average time of 1.772 seconds when compiled with the –O2 flag for the 300x300.grid.txt file using words2.txt as the dictionary file.

In the original hash table program that I implemented in the prelab, I implemented the hash table using separate chaining in order to resolve the collision problem by hash tables. I used the list from the STL for when there is collision and stored the values that do not collide in a vector from the STL.

I picked a new hash function designed to make performance worse in terms of time by just deleting all the code in my hash function and just returning 0. If I return 0, it means that it would not have any unique key-value pairs. Thus, it will just create a long list of chains of strings found in the word puzzle through separate chaining by the list that I used from the STL. I did this since it would have more collisions. As more collisions occur due to simply returning 0 in my hash function, it increases the probability of collisions, which will take more time. The running time for this was 86.407 seconds, which makes it incredibly slower.

To make the running time worse by changing the table size, I changed the constructor of my hash table. As I set my table in my hashTable constructor, I originally multiplied it by 2 in order to take care of some collision that could appear. However, when I took out the 2, it gave me a slower running time. This is because the decrease in table size would increase the probability of collisions, and it gave me a running time of 1.914 seconds.

In order to optimize the running time of the word puzzle and my hash table implementation, so that I can improve and make it faster, I tried to decrease the load factor first. Since my original implementation had a “successful load factor” as covered in lecture, in which I add one node plus the average length of the string (1 + length / 2). Then I went through every other letter in the string, rather than going through each string. Then in a loop, I went and multiplied each two iterations, skipping a letter, to the counter and added the string at the ith position (ASCII value). During the prelab, I first thought I could code in the load factor when I was in my hash_func, which is my hash function. However, when I asked a TA, she said I couldn’t do this in my hash function since the load factor is not related to the hash function and it is only related for finding.

Since I did not take care of this at my prelab, for the optimized code, I went through each letter in the string. Then in the loop, I went and added the value of the string at the ith position. The program gave me a running time of 1.587 seconds. I was confused at first why this happened at first since I expected at first that I was increasing the load factor which would be more unsuccessful, increasing the running time. However, as I cannot modify the load factor in the hash function I speculate that the processor might have taken more time due to the unnecessary multiplication and addition that I implemented in the prelab. Thus, the postlab, which only goes through one addition, might have taken less time. Thus, the speedup is 1.1165 (1.772 / 1.587).

Since I was not so sure why this produced a faster time, I decided to try more optimization hash functions. When I increased the table size, it gave me a faster running time. I did this since increasing the table size would reduce the probability of collisions. And it gave me a faster running time. The running time for this was 1.682. And the speedup is 1.053 (1.772 / 1.682).

Also, for my original hash table, I was just adding character values. In order to optimize my running time, when I multiplied each by a large number and the position in the character array, it gave me a faster running time. This was covered in lecture, which is the third hash function that the professor covered in lecture. It was a big improvement since the running time of this implementation gave me a running time of 1.698, and the speedup is 1.043 (1.772 / 1.698).