Justin Gou (jyg2qhc)
3/5/2020
postlab6.pdf

Postlab 6 Report

The big-theta run time of the word search program Θ(r*c*w). This is because the program loops through every position in the grid, which is r*c total positions. For each position on the grid, we test every direction and every possible word length to see if the words produced from them are words in our dictionary. The number of directions and the possible word lengths are both small constants (8 and 22 respectively), so we can ignore them in our big-theta analysis. However, to see if the given word is in our dictionary, we use the hash table. While finding an element in a hashtable is on average O(1), the worst case is O(n), where n is the number of elements. This is because if all the elements happen to hash to the same bucket, you would still need to perform a linear search. This means for every position in the grid, you would need to linearly search all the possible words, making the big-theta: Θ(r*c*w).

My program after optimizations run on words2.txt and the 300x300.grid.txt takes on average 0.13 seconds. Using a hash function that was designed to be slower, the runtime of the program was 3.54 seconds. The hash function I used was simply summing the integer values of all the characters in the string, then modding that by the table size. The reason this was slower is because it does not uniformly distribute the hashes across the table, meaning there are many collisions, meaning the program needs to iterate over a longer list in every bucket, which takes longer. I know it does not uniformly distribute the hashes because the order of the characters in the string does not matter, so any permutation of the same letters, such as “act” and “cat”, produce the same hash value, meaning they would collide. To slow down the program based on table size, I simply decreased the table size by a factor of 10. This slowed the runtime to 2.64 seconds. This is because decreasing the size of the table increases the load factor; in this case, the load factor was 5 because the number of elements was n and I set the table size to 0.2*n. By increasing the load factor, the number of collisions naturally increases. In this case, since the load factor is greater than one, by pigeonhole principle, there must be a certain number of collisions. However, if the load factor were less than one, pigeonhole principle says that there is no guarantee that there is a collision. This means the program needs to search through longer lists for each bucket when the size of the table is decreased, meaning the runtime is slower. The computer I ran my program on was a 2015 MacBook Pro with a 2.2GHz Quad-Core Intel i7 processor.

My code initially ran with an average of 14.30 seconds using words2.txt on the 300x300.grid.txt. I used this as a standard and ran my optimizations on the same sample. The first optimization I did was compiling my code with the -O2 flag, as instructed to do so in the inlab. This significantly decreased my runtime to an average of 4.32 seconds. The next thing I did was change my hash function. I found out that multiplication is a rather slow operation so instead I decided to use bit shifting. Bit shifting to the left by one bit doubles the number. This

means we can easily make the hash function $\sum_{i=0}^{n} s_i \cdot 2^i$ by simply shifting by the index for each character in the string. This decreased my runtime by a decent amount, running on average in 2.87 seconds. Next, following the suggestion from the postlab instructions, I implemented a prefix table using another hash table. This was done by looping through each word while creating the dictionary and adding all the prefixes to a table. Then, when searching the grid, if the word of shorter length was not in the prefix table, we know it is not a prefix of another word, so we can break that search length and move to the next direction. This drastically decreased my runtime to an average of 1.38 seconds. Next, I tried changing the size of the hashtables to change the load factor. However, I did not see a noticeable difference except when significantly decreasing the size of the prefix table, which only made it slower. I believe my original implementation had a small enough load factor (around 0.5) to maintain a small number of collisions. Finally, I decided to test different hash functions. I ended up finding a couple of algorithms online and implemented those. The first one I tried was called a Jenkin's one-at-a-time hash. This dropped my runtime significantly, bringing it down to 0.14 seconds. I also tried another hash called the FNV hash, which was very easy to implement but worked very well, though only slightly faster than Jenkin's hash. It brought my program's runtime down to an average of 0.13 seconds.

The overall speedup of my program run with words2.txt on the 300x300 grid is 14.30/0.13, which is 110.