Alex Yang, ahy3nz

11AM lab 104

Postlab 6

All the program were run on my personal computer.

The big-theta running time of the program is theta (r*c*w) where r is the number of rows, c is the number of columns, and w is the number of words. We must first loop through each character of the table, which is r*c. Afterwards, we need to search for every possible word w in the 8 directions. Reading in the dictionary is only linear with respect to the number of words because reading in the dictionary is only composed of scanning in line by line, which is just one word at a time. Compared to searching for words in the grid, which is theta(r*c*w), scanning in the dictionary has little effect on running time. Similarly, reading in the grid is theta(r*c) since we are just scanning in every element, and there are r*c elements. Still, however, this is smaller than searching for words, theta(r*c*w). Prefixing (mentioned later) reduced the amount of words w that had to be searched for. In doing so, running time was a considerable factor faster because w was cut drastically by comparing prefixes first. Printing each ward is simply constant time, but printing is lumped into the loops with searching for words, which, as we have already concluded, is theta(r*c*w).

One method I used to optimize my hashtable was my hash function. By multiplying by 101, I received times of about 412 milliseconds (140x70.grid.txt) and 3023 milliseconds (300x300.grid.txt). By multiplying by powers of 37, it took 8009 milliseconds(140x70.grid.txt) and 66769 milliseconds(300x300.grid.txt). Using this new hash function of powers of 37, I multiplied each character in the word by a power of 37, and the resulting sum was the modded by the table size to get the hash value. Performance was worse because I had to repeatedly calculate powers of 37 for each character of every word. I did not store the powers of 37, which meant the program had to recalculate the powers each time, which greatly slowed down the running speed. By simply multiplying the current sum by 101 and adding another value, computation time was diminished. There were no extra

calculations necessary like in calculating the powers of 37. As a result, the faster hash function had a speed up of 20.

To make table performance worse by changing the table size, I created the hash table to be some needlessly large size (the prime number after 100,000). This resulted in the hash table having a lot of empty space. Processing 300x300.grid.txt ended up taking 3380 milliseconds (compared to 3023 milliseconds of an appropriately sized hash table). Given my collision resolution strategy of separate chaining, the size of the hash table had only a minor effect on running time since there was no need to probe and search the table for open spots. However, there was still a marked slowness due to the large size of the table. The speed up from appropriate table sizing was 1.11.

Another method I used to optimize my hashtable was prefixing. Without the implementation of prefixes, it took 703 milliseconds to resolve 140x70.grid.txt. With the implementation of prefixes, it took 412 milliseconds to resolve 140x70.grid.txt. As a result of prefixing, my code almost ran twice as fast. Using 300x300.grid.txt without prefixing, the entire program took 6881 milliseconds. With prefixing, 300x300.grid.txt took 3023 milliseconds. Again, prefixing roughly doubled the running speed of the program. By prefixing, I reduced the number of times the program had to search for words in a given direction. Rather than loop through various lengths of words for every possible word, the program was able to take a shortcut and look at the first couple of characters in a given direction and use that information to conclude whether to continue searching for words in that direction or not. This greatly reduced the number of word searches the program had to perform. The speed up from prefixing was 1.71.

Another optimization was the load factor. Using a load factor of .5, 300x300.grid.txt took 3036 milliseconds. For the same grid file but with a load factor of .75, running time was 2967 milliseconds. This was only a slight optimization, but still a noticeable difference. Because I was using separate chaining, the load factor had less bearing on performance than with other collision resolution strategies (i.e. some probing methods where the program visits subsequent spots in the hash table in order to look

for an open spot to place the key).