Holt Crews, jhc6we, 10/19/17, postlab6.pdf

Everything in this postlab was done on my personal laptop and used the wordPuzzle.cpp, hashTable.cpp, primenumber.cpp, getWordInGrid.cpp, and timer.cpp files, along with any necessary header files.

Big Theta:

The big theta of my word search algorithm is r*c. The algorithm for searching for words includes a double nested for loop dependent on the number of rows and the number of columns of the input grid. The algorithm has to go through each individual to (x, y) coordinate of the grid and then search in each of the 8 directions and up to a certain number of letters. The 8 directions and the number of letters are two constants so they do not affect the Big Theta of the algorithm. The hash table itself does the searching in constant time because of the nature of the hash function and the collision strategy. The limiting factor here is the size of the grid and having to go through every possible coordinate value.

Timing Information:

All of these tests were done with the words2.txt and the 300x300.grid.txt files. The original application, after a great deal of optimization, ran in .742426 seconds. To make the application slower, I changed the hash function to just the sum of the ascii values of each letter in the the string to be hashed. By doing this, I am creating a lot more collisions and thus the hash table has to do much more work to search for a word. The final runtime for this implementation was 46.208722 seconds. Next, I changed the load factor of the hash table from .5 to .8. This increased the runtime of the program by a significant amount but not by nearly as much as my alteration to the hash function. Changing this load factor creates more collisions and more subsequent needs for the linear probing that I implemented as my collision strategy. The final runtime for this implementation was 1.765956 seconds, still a fast time but significantly slower than the .5 implementation.

Optimization:

All of this was on my Dell XPS13 with an i5 processor. The first collision strategy I used was open addressing with a linear probing collision strategy. I chose this because it was relatively easy to implement with an array and did a pretty good job with conserving space. In one result, this resulted in a runtime of 3.955805 seconds when printing to the console. I tried with a quadratic probing strategy and got a result of around 5.337265 seconds so for some reason, in this scenario, linear probing was faster. I began with a load factor of .5 because less than that would sacrifice valuable memory, and when I dropped it to .1, I only saw an improvement of .2 seconds. So, I decided to stick with the original load factor and look for other ways in which to optimize the word search algorithm. I decided to now leave my hashTable implementation alone and focus on the way in which I searched through the grid of letters and find ways to optimize that. I had also already tweaked the getWordInGrid code a little bit so that returned a non-real word when it began to search for words out of the bounds of the original grid. Here I found very little improvement and the speedup was about 1 or less than one in the case of quadratic probing.

I decided not to edit my hash function as I was already getting decent results with the one I was currently using and figured that I would have gotten much worse results with an even a tiny bit worse hash function as that affects the speed so much. I decided to continue on and look toward optimizations within the physical word search of the problem.

I had already put some restrictions on the the for loop that iterates through lengths of searches in order to cut down on the run time. I made sure that the loop did not look for words outside of the grid boundaries and not search for words longer than the maximum length of a word in the dictionaries. I decided to now look into stopping the word search in a direction in which there was sure not to be another word. So, to get an optimal time while totally disregarding memory usage, I created a second hashTable that contains all of the 5 letter prefixes to words that are longer than 5 letters. I chose 5 rather arbitrarily but partly because there are so many more 3, 4, and 5 letter words than any greater size. I was then able to stop a search in a given direction if a prefix in the prefix hashTable was not found in the first five letters. This greatly improved my time. My most recent runtime was only .742426 seconds for the 300x300 grid with words2.txt as the dictionary. At the end of all of my optimization, my speedup factor was 5.3282145291.