//Robert Alexander, RMA3MZ, Post-Lab6.pdf

Post-Lab 6

Big Theta Running Time

The Big Theta running time of the dictionary portion of my code (inserting words) was linear, as it read each line and inserted the word into the hash table. The actual word search part of my program was dependent on the number of rows and columns. Specifically the complexity of this portion is: rows * cols * 8.

The time complexity for the entire word puzzle is: rows * cols * 8 * words

Optimizations

*Note: Speeds given with 300x300 grid w/either words.txt or words2.txt, All times taken from virtualbox on a MacBook Pro

I implemented multiple different optimization methods for improving the running time of my word puzzle app.

In terms of my hash function, I made a few optimizations in both the pre-lab and post-lab. In the pre-lab I was using the pow function, which proved to be extremely slow and not very stable. So, I decided to switch to a hash function that included a hash value and did some multiplication on the ascii value of the characters in the word. In the post-lab I modified the initial hash value until I saw minor improvement (ended up with about 5% improvement from this).

One of the major optimization improvements I made was storing the maximum word length in a given dictionary. By storing the maximum size, I was able to prevent the constant searching for words that are longer than the longest word that could possibly be found (the longest word in the given dictionary). For example, in words2.txt the maximum word length is 16. As opposed to running separate checks for all words between 16 and 22 characters, I was able to bypass this wasted time by implementing the length check. Before these improvements my program took 9.021 seconds to run on virtualbox (on a MacBook Pro), after I saw a 1.62 speedup to 5.586 seconds. This clearly is a major improvement for dictionaries that have fewer than 22 characters. Also, while it didn't make any improvements for the words.txt dictionary (because its max length was 22 words), it did not hamper performance.

The final major optimization I added to increase the speed of my program was the addition of a buffer. I ran into a few issues implementing the buffer, such as issues with clearing and figuring out how to actually implement it. In the end I just added a bunch of string streams to a vector and printed the contents of the vector. This proved to be a major improvement over just printing to the screen. When looking at words.txt (so I don't take into account improvements from the max length optimization) I saw an original speed of 13.74 and an improvement of 1.7 to 8.06 seconds. This proves to also be a major improvement.

Overall my optimizations improved the speed of the 300x300 search with words.txt by 1.8 from 13.7 to 7.7 and words2.txt by 2.03 from 7.95 to 3.92.

*Note: Virtualbox Macbook Pro, 140x70 words2.txt

Instance	**Time**
Original Time	.397
Bad Function	.967
Bad Size	1.164

Many of different bad functions I chose were too bad, such as basing it completely on the size of the word! The bad function I got to work without causing compete destruction is basing the hash value on the length of the string causes the program to take 3 times longer. Basing it on the length causes multiple collisions because many words have the same length. To cause problems with the table size I decided to remove the prime number check, which causes the table to be much smaller and increases collisions by quite a bit.