Postlab6 Report

Alan Zheng (az4xfp)

CS 2150, Section 104, March 6 2020

1 Big Theta

The big-theta running time of the word-search portion of my program should be $\Theta(r \cdot c \cdot w)$. The actual code is several nested for loops. It loops through each row individually, it loops through each column individually, and each lookup of a word has worst case $O(w)$ (using separate chaining with linked lists) if the hash function or the table size is really bad and all the elements fall under one cell. This could theoretically be improved by making the buckets AVL trees instead, which would speed the worst case lookup to $O(\log w)$ but slow down the creation of the hash table. But for my implementation, the big-theta runtime is $\Theta(r \cdot c \cdot w)$. We can ignore the for loop for word length since we are assuming that to be a small constant.

2 Timing Results

2.1 Post-Optimization

After all of my optimizations, the best 5-trial average I got was **31 milliseconds (0.031 seconds)**.

2.2 Slower Hash Function

The slower hash function that I chose, like many of my hashes, goes through the string character by character. I start with an initial hash value of 1, and then for each character, I multiply the hash value by $((c * 37)\%128)$ and do one left bit shift. Using this hash function, the 5-trial average was **45 milliseconds (0.045 seconds)**. This hash function was slower because it created more collisions and had more operations than my more optimized hash.

2.3 Slower Table Size

The slower table size that I chose was 2041, which gives a load factor of about 75%. Using this table, the 5-trial average was **48 milliseconds (0.048 seconds)**. This table size was slower because it had more collisions since there

were less cells to fill. I am using separate chaining with a linked list to handle collisions, so that indeed matters.

3 Computer

The type of computer I am using is a 2018 MacBook Pro. It has a 2.3 GHz Quad-Core Intel Core i5 processor and 8 GB of 2133 MHz LPDDR3 memory.

4 Optimizations

Before any of these optimizations, the 5-run average was **7454 milliseconds (7.454 seconds)**

- O2 flag
  - I tried this optimization because you guys told us to - this flag just lets the compiler optimize my code.
  - Time improvement: **7454 ms to 1578 ms**
  - Speedup: **4.72370089**
- Breaking when the retrieved word size was less than the looped variable len instead of simply continuing
  - I tried this optimization it would allow my code to skip a lot of unnecessary repeated lookups of the exact same word in the exact same place.
  - Time improvement: **1578 ms to 1517 ms**
  - Speedup: **1.04021094**
- Storing the found word "coordinates" as one number to be processed after all words were found
  - Instead of using the to_string method over and over, which is slow, leave that to postprocessing.
  - I was having trouble figuring how to store the information, and I decided to condense the row, column, and direction into a 3 digit base-301 number, since 300 is the max row/column.
  - Time improvement: **1517 ms to 1490 ms**
  - Speedup: **1.01812081**
- Keeping track of the last hash when searching in one direction
  - Instead of recalculating the hash from the beginning each time, since my hash is accumulated character by character, I can keep track of the last hash so not as many operations are needed.

  - I had to redefine the parameters of the hash function to implement this optimization.
  - Time improvement: **1490 ms to 1460 ms**
  - Speedup: **1.02054795**

- Using FVN alternate hash
  - This hash is fast and also has not as many collisions.
  - Time improvement: **1460 ms to 1398 ms**
  - Speedup: **1.04434907**

- Creating prefix tables for every word length during preprocessing
  - Using prefixes makes sure that we aren't looking in directions where we know won't yield any valid words, so many of the lookups are pruned.
  - The one problem that doing this is that it makes reading the dictionary and building the hash tables much slower. It is not too noticeable for smaller dictionary sizes, but it is for larger dictionaries. This implementation is very not space efficient and makes the preprocessing take much longer, but does make the solving much faster.
  - Time improvement: **1398 ms to 50 ms**
  - Speedup: **27.96**

- Removing sanitizers during compilation
  - Now that I know that my code doesn't have any leaks, I can remove the sanitizers for speed purposes.
  - Time improvement: **50 ms to 31 ms**
  - Speedup: **1.61290323**