Derek Johnson dej3tc

10/24/19

The Big Theta run time of my application is $\theta(n\text{^}3)$. When we are looking at the maximum run time we have to determine what would result in the worst possible performance from the application. In our case, this would be when we are searching the words starting at the character in the last row and last column. Additionally, since we are using separate chaining without a rehashing strategy, it is possible that the hash table will be one long list. In this case we must search every word in the list to find the match. This adds another degree to our time complexity which can now be represented as (n*r*w). The number of directions that we need to check and the word size do not need to be factored into the time complexity as they are both constants and are the same for every run.

Timing Information:

My original hashing function used a polynomial rolling hash. In this method, every character is converted to an int using c++'s typecasting. It is then multiplied by a prime number(in my case 37) and the degree of the prime number is increased by 1. This is a relatively strong hashing function and it does not require very much time to calculate. The average time using this function is 1.39 seconds.

2) The hash function that I designed to make performance worse used a similar step as before but with less collision safety. In my new hash function, I convert every character in the string to an int using typecasting and add those integers up. This does the job but it is not as good as the original function because it makes it easier for two different strings to output the same hash. This will cause more collisions and thus longer chains in the vector. These longer chains will slow down the process of finding matches or non matches when the search section of the program is running. The average time with the new hash function is 1.457 sec.

3) In my program, in order to determine how large to make the hash table, it first runs through the file to count the number or lines. The number of lines is then multiplied by a constant call the load factor (in my original method this was 2). This multiple of the number of lines is then put into a helper method to find the next highest prime number. Making the table size a prime number minimizes the chance for collisions when doing collision resolution. In my discussion below I list the performance with different load factors, but they all result in an increase in speed. To decrease the speed of the program I will reduce the size of the load factor. I reduced this constant to .5, reducing the size of the hashTable. This makes the chances of collisions much higher, and for the same reason as above slows down the search method. Using a load factor of .5 I got an average run time of 1.670.

Starting Speed

Dictionary File	Word Grid	RunTime(sec)
250x250	words	1.317
	Words2	.967
300x300	words	2.563
	Words2	1.563

Optimizations for code

- Choose a good load factor
    - For the implementation that I chose, I did not require any resizing because before hashing I go through the dictionary file to check the number of words. My load factor is then just a constant that I multiply by the number of words.
    - After looking at the chart I am going to use a load factor of 4 for all further trials

Load Factor	time
2	1.558
1.5	1.683
3	1.434
4	1.398
6	1.369

- I noticed that the average times I was getting when I ran the program without outputting to the terminal were much faster than those I got when I ran them with a.out. To cut down on buffer time, I used the flush command at the end of my cout statements and included the line
  cout.sync_with_stdio(false) to cut down on buffer time

  My new average time was 1.354 sec

- Better Hash Function: I had read that you can get a better hash function if you split the word into groups of bits, hash them and then add the result. I implemented this in a new hashFunction that I will call hashFunctionV2. What I found was that the new hash function performed better than the original hashFunction when the load factor was small (< 3) but worse for larger load factors. This probably means that it is more efficient but takes longer to compute.

HashFunctionV2	Load Factor	1.368
	4	1.368
	2	1.383
	3	1.358

  For the remainder of the trials I will be using this new hash function with a load factor of 3 as this seems to be the best mix of speed and memory efficiency.

- Data Structures for Separate Chaining: In my original implementation I used a list. Below are the speeds using different data structures

Separate Chaining Data Structure	time
Forward_list	1.389
deque	2.077

I was surprised by the much slower speed of the deque data structure. From what I could gather on cplusplus.com it seemed as if this data structure would work similarly to the list and forward_list since we are only inserting at the tail of the sequence. I think that the difference between the forward_list and the list is negligible. The forward list would be a better implementation for our uses

however as it is more memory efficient. Forward lists only store a pointer to the next element and thus are better suited to operations starting from the beginning of the list.

Overall speedup: 1.151