Hyun Suk Ryoo (Max Ryoo)
Hr2ee
CS2150
Post-Lab 6.pdf

Big Theta Running Time

The big theta runtime for my program would be rows * cols * words for the entire word puzzle. The length of the words would not be accounted for. The rows, cols, and words matter because it is in the quad nested for loop and have a great impact. The big theta running time of the dictionary insertion would be linear because each lines is read and inserted into the hash table and is dependent upon the number of lines in the file.

Optimizations (all times recorded by 2017 MacBook pro 2.3 GHz i5 8GB) **all data shown in table

The method I originally took on was linear probing. I would increase the index by 1 and check the next index if that index was filled in the array. This resulted in a really slow running time because of the way I implemented it. I would get results such as 205.4938seconds for the words.txt 50x50.grid.txt. In order to change this problem, I changed the x++ to x = x*2 to make the search for a new open index easier and faster. I also tried to mod the hash function by modular 10 instead of modulating the asci value of the string by the number of indices in the array, which resulted in a really bad time probably due to many clashing and trashing. The difference of time is shown in the **data collected table of the report. The N/A is shown that the program would crash or I couldn't detect a reasonable time nor could I wait so long.

I tried making the size of the array bigger, but didn't make a huge difference in run time. However, when I made the array size half of what I originally had before it became significantly slow as shown in the data collected table in the report. Since the original size seemed to fit decently I decided to keep that implementation.

**Data collected:

./a.out	Original time it took	After -O2 optimization	After optimization	After -O2 after optimization	Sped up factor	Bad implementation of smaller hash table after optimizations	Bad implementation with a bad has function of mod 10.
/a.out words2.txt 300x300.grid.txt	188.242485	19.977297	8.885325	2.17414	x86	40.391156	N/A
./a.out words.txt	205.493815	20.099578	2.295957	0.509679	X403	48.518788	N/A

50x50.grid.txt							

I believe the biggest optimization I made was for making the x++ to a double probing method. This would decrease the time significantly. The most important thing however was fixing my insert and find function. In my originally implementation I would not insert the words correctly because I would do x++ ++ instead of just x++, which resulted in the difference of words found. The main thing to look for to make the implementation of hash table a good hash table is to be careful of the clashes that happen and also the size of the array that was used. The worst case that I experienced was when my hash function would make a lot of collisions when my array was small. I fixed this by making my hash function not have much collision by modulating it by the array number and also making the load factor below 0.5. I would also not linear probing it x++, but would rather do x= x*2, which resulted in a fairly fantastic implementation.