Pat Wongwiset

nw9ca

10/18/17

postlab6.pdf

Big-theta running time:

The word searching has r*c***w** big-theta time complexity such that the program uses quad loops such that r and c are big enough to effect running time when the other two loops has the small number of repetitions compared to r and c. A hash table using quadratic probing and load factor = 0.1 optimize the searching method to be constant when finding each word. The program prevents *vector<string> hTable* to resize automatically and words overflowing in *hTable* by increment the dictionary file once, and then specifies the size of *hTable* to be 10 times larger than the number of total words in dictionary file. Thus, the bigger size and quadratic probing can reduce collisions when inserting and finding elements in *hTable.* Also, we need to run at least w times to find all words in the grid table, so the running time would be r*c*w.

Timing Information:

Using MacBook Pro (Mid 2012) to run words2.txt and 300x300.grid.txt

1. Original Application (quadratic probing): 2.439 s
2. New hash function (hashFunct = Σ(i=0 to i = length(s) - 1) s[i]*boostPower(101,1)): 4.112 s

If the hash function is powering 101 by i instead of multiplying the number, the program will take longer time to calculatate the power in the hash function.

3. New hash table size (size = sizeDict*2): 3.832 s

The load factor was changed from 0.1 to 0.5, so the hash table had smaller size. The small size of the hash table increases the number of collisions, so the time implementation is longer when inserting and searching.

Optimizations

step	**Optimization**	**time (s)**
1	hashFunct = Σ(i=0 to i = length(s) - 1) s[i] (linear probing)	16403.88

2	hashFunct = Σ(i=0 to i = length(s) - 1) s[i]*pow(103,i)
	hashFunct2 = 101 - (Σ(i=0 to i = length(s) - 1) s[i]% 101)
	(double hashing)
		10.362

3	Including -O2 flag in Makefile	9.997
4	Changing load factor from 0.5 to 0.1	9.854
5	Storing all result in a stack and then printing	9.445
6	Storing all result in a string and then printing	9.115
7	Using boostPower() instead of pow()	4.475
8	Changing len in getWordInGrid() to 22 instead of 255	4.296
9	hashFunct = Σ(i=0 to i = length(s) - 1) s[i]*boostPower(103,i) (quadratic probling)	4.112
10	hashFunct = Σ(i=0 to i = length(s) - 1) s[i]*103 (quadratic probling)	2.658

1. The hash function caused the table to have lots of collisions, so the insertion method only ran in specific range of the table.
2. The hash function was changed so that elements could be spread out through all blanks spaces in the table more. Also, I created double hashing in order to speed up the program and reduce clustering.

3. Even though the program compiled slower, the executable part ran a little faster with -O2 flag.
4. The smaller load factor can decrease collisions. The program can run faster since the number of overflown key is smaller when the size of a hash table is larger.
5. The printed statement in four loops causes the program to run slower since the program needs to run and print out every time when conditions meet. In stead, I added all elements in a stack, and then printed it out after finishing word searching.
6. I used the other print implementation, and then compared to that of stack implementation. Storing data as a string can boost the program to run a bit faster.
7. pow() method from <math.h> is slow, so I created new boostPower() method in order to accelerate the implementation on that part.
8. I adjusted the given getWordInGrid() method to declare a length of a character array from 256 to 22 because we knew that the program has the word length up to 22 characters. Thus, declaring a large array every time when the function is called uses lots of unnecessary space, and then decelerates the speed of the program.
9. The second hash function was removed and then the program was tested with quadratic probing. The program ran faster because calling the second hash function may use extra time to implement for every words.
10. Since multiplying 103 to a current index of a given word is efficient and can spread out the element through a hash table as the previous hash function. The program will run faster since the current hash function is simpler to implement.

Problems and Solutions

My code had a lot of ‘segmentation fault: 11’ when tested, so it is useful to work in a little piece of code instead of checking the whole code at one time because it is easier to find the wrong section in the code. Otherwise, printing test to check the whole code is another way to do that, but it would be more difficult if there were the large number of lines and files to run a program.

Additional Opinions

I think the lab should direct students to try on both main hashing implementations, chaining and linear probing, by simplifying the structure of this lab to be easier. Then, students can learn how to use both implementations, so they can learn and experience advantages and disadvantages of them.

After optimizing the program, it speeds up (16403.88/2.658) = 6171.51x.