Nathan Aun

7/24/19

Lab #4 – Hash Tables

CS2302 – Data Structures

Summer 2019

Introduction

The problem posed for this lab is to implement 6 different hash functions and use them to create a hash table of words. Using data gathered from an NLP (Natural language processing) program, we also need to have the embeddings in the hash table. Once the hash table is constructed we also need to compare how similar two words are based on the cosines difference formula. We also need to print runtime statistics of the hash function, from how long it took to hash everything as well as things like the longest chain, the average chain length, etc. for the function.

Proposed Solution

The actual implementation for each hash function was almost exactly how we were told to do it, so I will not go into them here, except for hash function 6. Hash function 6 was one of our choice, so the one I used multiplied every ascii value in the word by each other. The reason this is better than the sum algorithm is because it will spread the data out a lot more, whereas just summing them all together ends up with very long chains by the end

         The way other methods were implemented was fairly simple. Reading from the file will read all the words and return a python list that contains a list for each word, with those lists consisting of the word itself and all 50 embeddings. There is a for loop in the main method that will take the data from this list and use it to make a list of words, an object made specifically for this lab. The word object has two attributes, the word that it represents and an array of all the embeddings. Once this is done it will prompt the user for the hash function they would like to use. It then builds the hash table according to the hashNum that was inserted.

The insert method will resize itself if the load factor becomes greater than .5, meaning that more than half of the table is populated. This is important because the more empty the table is the higher chance we have to have a better runtime, so we want our load factor to be as low as possible. The way resize works is it will create a new temporary hash table of the larger size (determined by doing the original size times two plus one) and then from there it populates the new table with all the elements from the original one. Then it will set the original one to be the temporary one.

After the insert method is done it will print the stats. Time is taken before and after the table is built and we do after - before to find the time elapsed. The size of the table is easily retrieved by calling the length of it. The rest of the stats required their own methods to implement.

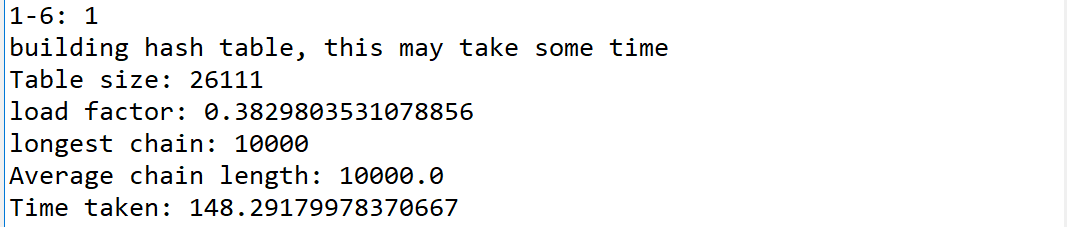
Load factor takes the number of elements in the table and divides it by the size of the table. Longest chain will find the longest chain of consecutive words in the table. Average chain length will take a list of all the chains in the table and average them all together.

The method sim will read the given file, and turn it into a list. Once the list has been created, it will iterate through it using a find method to find the word in the hash table. If the word  is not in the table it will notify the user, else it will run through the cosines difference formula and print how similar they are to one another.

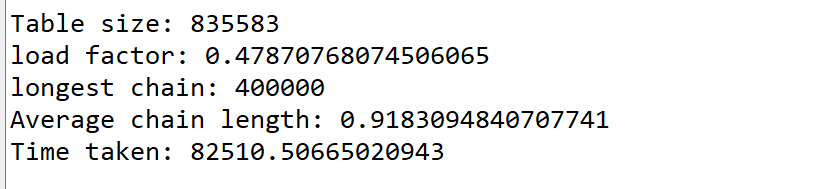
Experimental Results

The first few hash functions take a very long time to run, so instead of using the full amount of data with them I will instead be using a small portion of the words, around the size of 10,000. The exception to this is hash function 2, which I did let run to its entirety, to which it took 82,510 seconds, or approximately 22.9 hours.

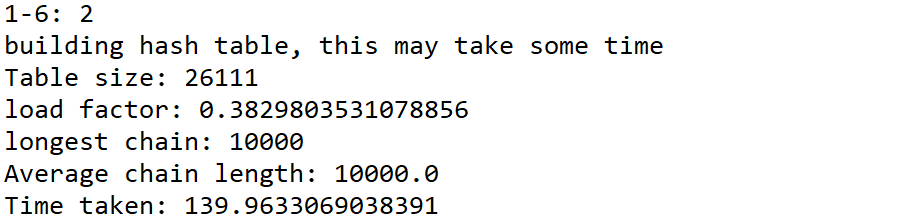
These are the following statistics for hash function 1(for 10,000 words)



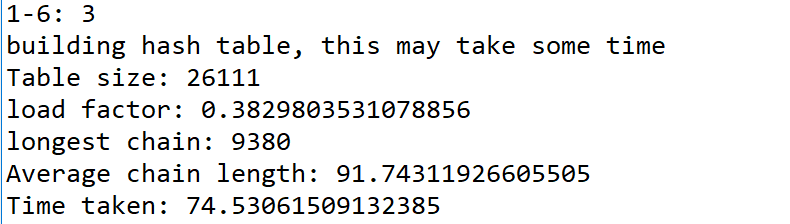
The output of hash function 2 is as follows (please note that the “average chain length” method was not working at the time of this run and as such it gives an inaccurate number. In reality this number should be 400,000)



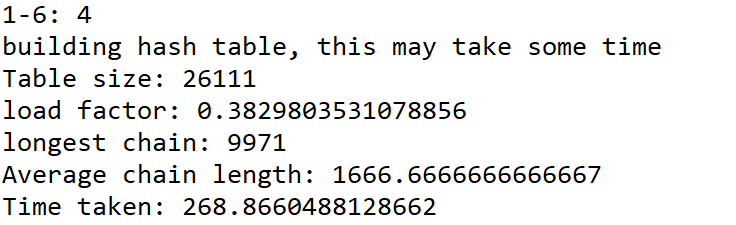
For sake of comparison, however, here are the statistics for hash function 2 with 10,000 words:



These are the following statistics for hash function 3 (for 10,000 words)

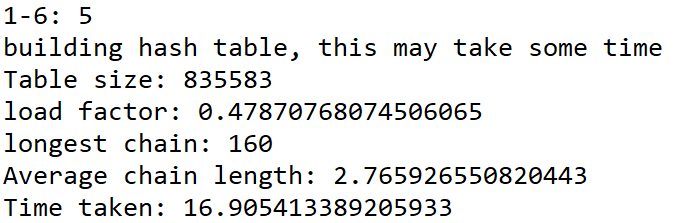


These are the following statistics for hash function 4 (for 10,000 words)

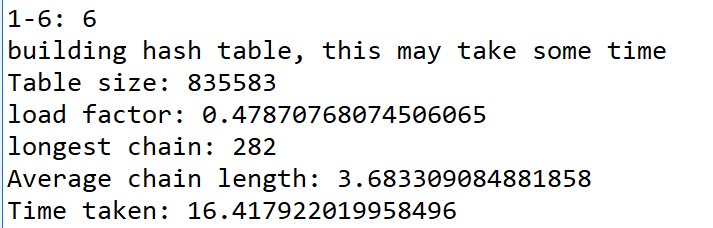


The final two hash functions did work a lot better than the first four, and as such I will be using the amount of words from the file (all 400,000)

         Here are the statistics for hash function 5:



Lastly, here are the statistics for hash function 6



Another important stat is how the tables are at comparing the words to one another. The following table has all the stats put together for easy comparison

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Hash Num | Num Words | TableSize | Load Factor | LongestChain | Avg Chain-length | Time Taken to make table | Time taken to compare words |
| 1 | 10000 | 26111 | 0.38298 | 10000 | 10000 | 148.29 | 0.15 |
| 2 | 10000 | 26111 | 0.38298 | 10000 | 10000 | 139.96 | 0.106 |
| 3 | 10000 | 26111 | 0.38298 | 9380 | 91.743 | 74.53 | 0.096 |
| 4 | 10000 | 26111 | 0.38298 | 9971 | 1666.667 | 268.86 | 0.0633 |
| 5 | 400000 | 835583 | 0.478707 | 160 | 2.765 | 16.905 | 0.003 |
| 6 | 400000 | 835583 | 0.478707 | 282 | 3.683 | 16.417 | 0.015 |

Lastly, here is the data for words being compared to other words 

Conclusions

In general, it seems that functions 1-4 really would not be good to use on functions even as large as 10,000 data sets. Even at that relatively small amount of words , it takes upwards of a minute for hash function 3 and upwards of 2 minutes for the others to build the table, which is less than ideal. Interestingly, hash function 6 was slightly faster for making a hash table than 5 was, but 5 is much faster at actually accessing the table and comparing items. This is because hash function 5 has a much better spread than any of the other functions that were used. Since the data is more spread out, it means there were fewer collisions and, as a result, faster times to actually find the words we’re looking for. Just generally, a good hash function will spread data out as much as possible.

As far as the comparisons themselves are concerned, they are close to what they should be but still slightly off and I’m not entirely sure why. It’s likely that there is some small error in my sim method, but I was unable to find it.

Appendix

Source code is as follows:

'''

Report link: https://docs.google.com/document/d/1diC-XctjkF1c8L9ORTI1NxkPMXp7iwcwUL0QJIbUmTg/edit?usp=sharing

Course: CS2302 Data Structures

Author: Nathan Aun

Assignment: Lab 4 - Hash tables

Instructor: Dr. Fuentes

TA: Ismael Villanueva-Miranda

Date of Last Modification: 7/24/2019

Purpose of Program: To implement and demonstrate 6 different hash functions to compare the similarity of two words using NLP(Natural language processing)

 '''

import numpy as np

import time

import math

#word class allows for our hash table to be more organized

class Word(object):

    def \_\_init\_\_(self, L):

        self.word = L[0]

        embedding = np.zeros(len(L)-1, dtype = np.double)

        for i in range(1,len(embedding)):

            embedding[i] = L[i+1]

        self.embedding = embedding

#Our hash table stores word objects in it and hashes by the word

class HashTable(object):

    def \_\_init\_\_(self, size = 101, numItems = 0):

        self.item = np.empty(shape=size, dtype=np.object)

        self.numItems = numItems

#read file will read all the lines in the file and return a list, where every element is the word and all 50 embeddings

def readFile(fileName):

    f = open(fileName,"r", encoding ="Latin-1")

    fullFile = f.readlines()

    L = []

    for i in fullFile:

        L.append(i.split(' '))

#        if len(L) == 1000:

#            break

    return L

#given a hash table will find the longest chain in it.

#Longest chain is our worst case insert

def longestChain(H):

    neg\_ones=np.argwhere(H.item == None)

    if len(neg\_ones)==0:

        return len(H.item)

    neg\_ones=np.append(neg\_ones, neg\_ones[0]+len(H.item))

    chain\_lens = neg\_ones[1:]-neg\_ones[:-1]-1

    return np.max(chain\_lens), chain\_lens

#Average chain length will find every chain that exists in the hash table and average them together

def AverageChainLength(L):

    total = 0

    numChains = 0

    for i in L:

        if i != 0:

            total += i

            numChains += 1

    return total / numChains

#Determines the load factor of the table, used for resizing purposes

def LoadFactor(H):

    return H.numItems / len(H.item)

#this is a general hash function that will call the correct hash function based on what

#the user input. i is used in the case of collisions

def h(H, word, i, hashNum):

    if hashNum == 1:

        return h1(H,word, i)

    if hashNum == 2:

        return h2(H,word, i)

    if hashNum == 3:

        return h3(H, word, i)

    if hashNum == 4:

        return h4(H, word, i)

    if hashNum == 5:

        return h5(H, word, i)

    if hashNum == 6:

        return h6(H, word, i)

#hash function 1 works off word length

def h1(H,word, i):

    return (len(word) + i) % len(H.item)

#hash function 2 works off ascii values of the first character

def h2(H, word, i):

    return (ord(word[0]) + i) % len(H.item)

#hash function 3 works off ascii values of the first character multiplied by the last character

def h3(H, word, i):

    return (ord(word[0]) \* ord(word[-1]) + i) % len(H.item)

#hash function 4 works off the sum of ascii values of all characters in the word

def h4(H, word, i):

    temp = 0

    for j in range(len(word)):

        temp += ord(word[j])

    return (i + temp) % len(H.item)

#hash 5 works by adding the first character, followed by adding every other character multiplied by 255

def h5(H, word, i):

    if word == '':

        return 1

    return (i + (ord(word[0]) + 255\*h5(H,word[1:],0))) % len(H.item)

#hash 6 works by mutliplying ascii values of every character in the word

def h6(H, word, i):

    hVal = 1

    for x in word:

        hVal \*= ord(x)

    return (i + hVal) % len(H.item)

#this function is called whenever the load factor of the table reaches >.5

#it creates a temp new hash table of a bigger size and sets the original one to it

def resize(H, hashNum):

    newSize = len(H.item) \* 2 + 1

    H2 = HashTable(newSize)

    for i in H.item:

        if i is not None:

            Insert(H2,i,hashNum)

    H.item = H2.item

    H.numItems = H2.numItems

#insert's an item into a hash table

def Insert(H,word,hashNum):

    if LoadFactor(H) > .5:

        resize(H, hashNum)

    for i in range(len(H.item)):

        pos = h(H, word.word, i, hashNum)

        if H.item[pos] == word:

            break

        if H.item[pos] == None:

            H.item[pos] = word

            H.numItems += 1

            return pos

    return -1

#find will determine whether or not a word is in the hash table

#if the word is, it will return the index of it in the hahs table

def find(H, word, hashNum):

    Hval = h(H,word, 0,hashNum)

    while H.item[Hval] is not None:

        if H.item[Hval].word == word:

            return Hval

        Hval += 1

    return None

#sim reads a file and compares two words and checks how similar they are

def sim(H, hashNum,fileName):

    f = open(fileName,"r")

    fullFile = f.readlines()

    L = []

    for i in fullFile:

        L.append(i.split(' '))

    for i in L:

        i[0] = i[0].strip()

        i[1] = i[1].strip()

        index1 = find(H,i[0],hashNum)

        index2 = find(H,i[1],hashNum)

        if index1 is None or index2 is None:

            print("one or both words are not in the data set, cannot compare")

        else:

            dotProduct = 0

            magnitude1 = 0

            magnitude2 = 0

            for j in range(50):

                embedding1 = H.item[index1].embedding[j]

                embedding2 = H.item[index2].embedding[j]

                magnitude1 += math.pow(embedding1, 2)

                magnitude2 += math.pow(embedding2, 2)

                dotProduct +=  embedding1 \* embedding2

            similarity = dotProduct / (math.sqrt(magnitude1) \* math.sqrt(magnitude2))

            print("Similarity", i, "=", similarity)

if \_\_name\_\_ == "\_\_main\_\_":

    print("loading, please wait")

    L = readFile("glove.6B.50d.txt")

    wordList = []

    for i in L:

        wordList.append(Word(i))

#        if len(wordList) == 1000:

#            break

    print("done!")

    print("What hash function would you like to use? (note, 1-4 take a very long time to run)")

    userIn = input("1-6: ")

    hashNum = int(userIn)

    if hashNum < 1 or hashNum > 6:

        print("Invalid hash argument, program cannot work")

        quit()

    H = HashTable()

    print("building hash table, this may take some time")

    start = time.time()

    for i in wordList:

        Insert(H,i,hashNum)

    timeTaken = time.time() - start

    print("Table size:", len(H.item))

    print("load factor:", LoadFactor(H))

    longest\_chain, chain\_lens = longestChain(H)

    print("longest chain:", longest\_chain)

    print("Average chain length:", AverageChainLength(chain\_lens))

    print("Time taken:", timeTaken)

    start = time.time()

    sim(H, hashNum, "testFile.txt")

    timeTaken = time.time() - start

    print("time taken for comparing words:", timeTaken)

Statement of Academic Honesty

“I certify that this project is entirely my own work, I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_