Najera Ericka

8 April 2019

CS 2302

Dr. Fuentes

Lab 5

The lab consisted of using knowledge not only from the data structures of binary search tree and hash tables but also the basic knowledge of reading files and storing them into the binary structures. This lab was more of something you would do in the real career world where you can be asked to read a file of information and storing correctly. This lab was quite challenging but feel accomplished for completing it.

The instructions for this lab was to read two files. One was the globe text file provided and one separately which was a text file of a list of two words. From those words we had to make the appropriate calculations to get the standard deviation of the those words and see if they are similar of not. At the beginning I honestly had no idea how to start and how I would even read a file in python. To help with that I created a flow chart to see which steps I needed to do to get the indicated results. First I decided I should work separately the binary tree and the hash table to focus on one and then the other. So from there I decided I had to read the globe file. From there I would store the info into separating the word and the embedding. Since the hash table was already created to store the word and embedding I just need to create a list that stored the word and embedding separately. As for the binary tree I would have to change the bst class which I could make it receive a list and that would be the item. From there I would make store them into the appropriate data structure which was hash tables or binary search tree. From there I would read the other file with words and save them into the two separate words. From there I needed to search for that specific word in the globe file. Then I would have to calculate the those two embeddings to get that standard deviation. After that I just need to print the results, add a time function to get the running time, and create different modules to get the report of the tree and hash table.

In order to read the file I didn’t know how to do it python so I had to research how to do it. From there I had to create a for loop to read each line and separate it by spaces or commas depending on the structure of the file. After that I save the first word of the file which would be the word and then I would do a for loop to save each embedding number into a np array. From there I had to insert that line either in the hash table or the binary. The hash table I would simply insert the word and the embed separately and the hash insert function would get the value of the word by itself. Since the instructions said we had to check the load factor and double the hash table. I created a double size and a load factor. As for the binary I made the item into a list and from there I would insert a list where it contains the value of the word, the word, and the embedding. Then I created that function into a build hash and build tree and returning the hash table or tree respectively.

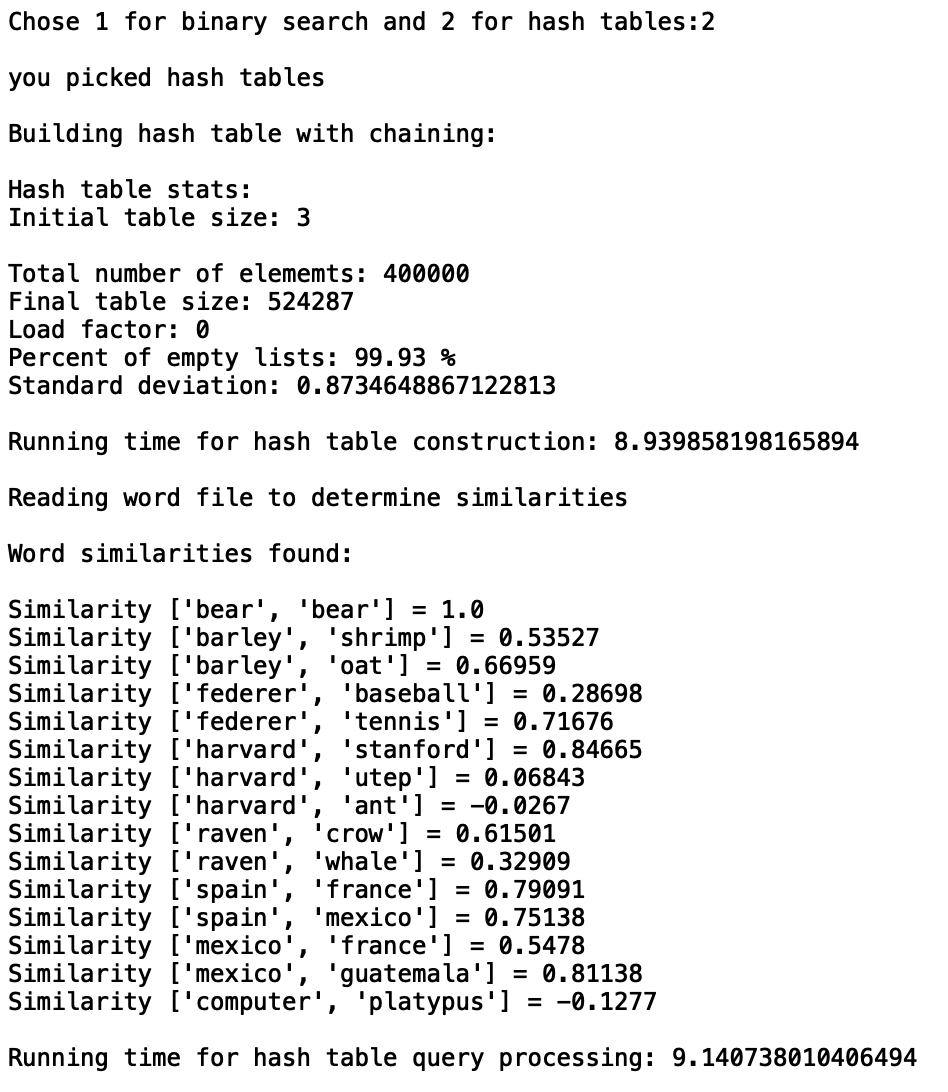
For the double the size I created a hash table with double the length plus 1. Then I would traverse the old hash table and reinsert everything into the new hash table. The load factor was just returning the number of items divided the length of the hash the table. For the value of the word for binary I used a similar technique as for the h method in the hash class. I receive the word which will make a list of each letters ascii number. Then I got that list and added the numbers from there I got the value of the word.

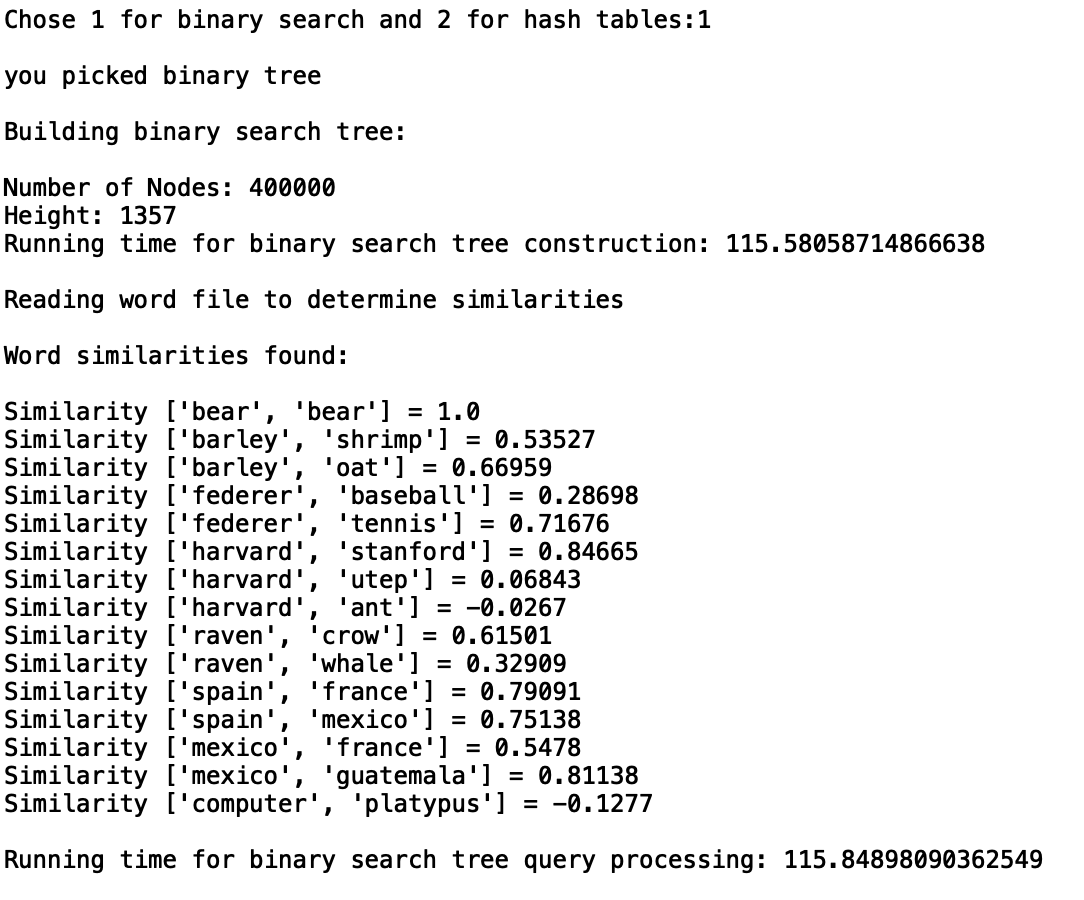
From there the next step would be to save the file of words to compare. To do that I saved read that file and saved the words into two separate variables. After I printed the words I realized that the last word was printing ‘/n’ at the end so I did the string replace function to delete that. Then after saving each word I would have to look for it in the hash tree of binary.

Starting with the hash table the function of findc was already implemented. So I just made sure to save the result in another variable. For the binary tree I had to create the function myself. I created a temp variable to traverse the tree. From there I would compare the first number and the value of the word we were looking for. If the value was smaller then, temp would become the left tree or if it was bigger it would be the right tree. Towards the end I realized I was getting the wrong calculations and after trouble shooting almost everything I realized I was returning the wrong word but had the same value. So I fixed the method by not only checking the value but also checking the word in the same if statement.

After I found the word in the array and the hash table I would need to get the standard deviation of the two embeddings. I thought about creating a for loop to read each value but I was pretty sure there could be a shortcut to adding and multiplying a np array and there was np.sum. The top of the fraction would be multiplying both sums of the eo and e1 or word1 and word2 embeddings. The bottom is the magnitudes which was obtained by squaring each embedding then multiplying each embedding by the same placement in the other embedding the the sum of each product followed the square root of everything at the end. At first I would get the value to be really big in decimal places so then I add the round function to round to 5 places. Since that was done for each pair of words it would print all the similarities of the words inside the for loop.

To print a report of the hash table I needed to create a function that calculated the empty lists and the deviation. For the binary search tree I just need to create height function for the tree. The empty lists I created it by creating a count then traversing the hash table. If it came across an empty list then the count would increment, returning the count of the empty lists in the hash table. For the deviation I created a list to store the lengths of all the lengths of the lists. Then I added all the lengths together then I got the mean. From there I subtracted each length in the list by the mean and then squared it. Then I got the sum of the those then I got the mean again and I square root it generating the standard deviation of the lengths. In order to get the height of the tree I thought about doing it with a while loop one for each tree. Then I realized you cant travers the whole tree by that. So I decided to do recursion. I traverse the tree whie it is not none and I count I everytime it goes down towards the leafs. But since one side can be larger than the other I had to get the max of the right and left to avoid getting a wrong number like I initially did.





Running times:

Build a hash tables:

﻿8.94 seconds

Build a binary tree:

﻿115.58 seconds

Similarities:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 50 | 100 | 500 | 1000 |
| hash | ﻿﻿13239000 | ﻿36963000 | ﻿332230000 | 539539000 |
| binary tree | ﻿19382000 | ﻿31679000 | ﻿383134000 | 547247000 |

﻿

When it comes to storing the word with embeddings, it is much faster to do it in a hash table. When it comes to actually finding a word, a binary search tree is much easier because it is O(logn) time where in the hash table you don’t know how long a list will be in that specific index.

Appendix:

﻿import numpy as np

import math

import time

class HashTableC(object):

# Builds a hash table of size 'size'

# Item is a list of (initially empty) lists

# Constructor

def \_\_init\_\_(self,size):

self.item = []

for i in range(size):

self.item.append([])

self.num\_items=0

#doubles the size of the hash table

def doubleSize(H):

#create a new hash table with twice the length +1

H2 = HashTableC((len(H.item)\*2)+1)

#traverse the old hash table

for i in range(len(H.item)):

for a in H.item[i]:

#for every index traverse the lists and insert into new hash table

InsertC(H2,a[0],a[1])

return H2

#counts number of empty lists in hash table

def emptyLists(H):

count =0

#traverse hash table and if that is empty increment count

for h in H.item:

if h == []:

count+=1

return count

#deviation of the lengths of the lists

def deviation(H):

lengths =[]

#get lengths of each index in hash table

for h in range(len(H.item)):

lengths.append(len(H.item[h]))

#get the sum of the lengths

sum = 0

for l in lengths:

sum +=l

#get average of the sum

mean = sum/len(lengths)

#subtract the mean of each length and sqaure it

for x in lengths:

x= x - mean

x = x \* x

#get sum again of the new lengths

sum = 0

for a in lengths:

sum +=a

#get mean again and square root

mean = sum/len(lengths)

mean = math.sqrt(mean)

return mean

def InsertC(H,k,l):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

if loadFactor(H)==1:

H = doubleSize(H)

b = h(k,len(H.item))

H.item[b].append([k,l])

H.num\_items+=1

return H

def FindC(H,k):

# Returns bucket (b) and index (i)

# If k is not in table, i == -1

b = h(k,len(H.item))

for i in range(len(H.item[b])):

if H.item[b][i][0] == k:

return b, i, H.item[b][i][1]

return b, -1, -1

#find the vakue of the word in order to get a value to store into hash table

def h(s,n):

r=0

for c in s:

r = (r\*n+ord(c))%n

return r

#computes the load factor

def loadFactor(H):

return H.num\_items//len(H.item)

def buildhash(f):

#creates a hash table

H=HashTableC(3)

print('Hash table stats:')

print('Initial table size:',len(H.item))

#read the globe file

for x in f:

#save the line and split after space

line = x

text = line.split(" ")

#save the word seperately

word = text[0]

#traverse the line and create a np array

i = 0

embed = np.empty([50], dtype=float)

#save the values of the line after the first word

for j in range(1,len(text)):

embed[i]=text[j]

i +=1

#insert the word and embedding into the hash table

H=InsertC(H,word,embed)

return H

#prints the similarities of the two words in file

def similarities(H,file):

#traverse the file with words

print('Reading word file to determine similarities')

print()

print('Word similarities found:')

print()

for x in file:

#build list with two words

words = x.split(',')

#gets rid of the /n

words[1] = words[1].replace('\n','')

#find the two words seperatly in the hash table

word1 = FindC(H,words[0])

word2 = FindC(H,words[1])

#e0 \*e1 is the sum of the the profuct of u and v values

top= np.sum(word1[2]\*word2[2])

#|e0||e1| is the square root of the sum of the product u and v squared

bottom = (math.sqrt(np.sum(word1[2]\*word1[2]))\*math.sqrt(np.sum(word2[2]\*word2[2])))

#print the similarity and reduces the float to only 5 digits

print('Similarity',words,'=',round((top/bottom),5))

print()

##############################################################################

class BST(object):

# Constructor

#item becomes an array

def \_\_init\_\_(self,item=[],left=None, right=None):

self.item = item

self.left = left

self.right = right

#reads file and creates a binary search tree

def buildtree(file):

T = None

#creates count of how many nodes in the tree

num\_Nodes=0

#reads globe file

for x in file:

line = x

#creates a list of each line

text = line.split(" ")

word = text[0]

#gets the word and get the value of that word

val = value(word)

i = 0

#create an np.array of length 50

embed = np.empty([50], dtype=float)

#traverse the line and input the floats after the word

for j in range(1,len(text)):

embed[i]=text[j]

i +=1

#insert them into the tree with the item being a list including the value, word and embedding

T=Insert(T,[val,word,embed])

#increase number of nodes after each insert

num\_Nodes+=1

print('Number of Nodes:',num\_Nodes)

return T

#read file of words and print the similarties

def similarT(T,file):

#read the file of words

for x in file:

#build list with two words

words = x.split(',')

#gets rid of the /n

words[1] = words[1].replace('\n','')

#find the two words seperatly in the hash table

word1 = find\_word(T,words[0])

word2 = find\_word(T,words[1])

#e0 \*e1 is the sum of the the profuct of u and v values

top= np.sum(word1[2]\*word2[2])

#|e0||e1| is the square root of the sum of the product u and v squared

bottom = (math.sqrt(np.sum(word1[2]\*word1[2]))\*math.sqrt(np.sum(word2[2]\*word2[2])))

#print the similarity and reduces the float to only 5 digits

print('Similarity',words,'=',round((top/bottom),5))

#insert item into the tree using the value of the word

def Insert(T,newItem):

if T == None:

T = BST(newItem)

#compares the value of the word in order to sort them

elif T.item[0] > newItem[0]:

T.left = Insert(T.left,newItem)

else:

T.right = Insert(T.right,newItem)

return T

#returns height of the tree

def height(T):

if T is not None:

#recursive calls both left and right tree

#keeps the biggest side and adds one each time

return 1 + max([height(T.left),height(T.right)])

return -1

#find the words and return that node

def find\_word(T,k):

#create a temp variable

temp = T

#traverse the tree

while temp is not None:

#if the the value of the word equal the value of k and the word equals the word return that node

if temp.item[0]==value(k) and temp.item[1] == k :

return temp.item

#if not check the value and send the left or right tree

elif temp.item[0]>value(k):

temp = temp.left

else:

temp = temp.right

return None

#returns the value of the word

def value(word):

#gets the ascii code for each letter

num =[ord(c) for c in word]

count =0

#adds all the numbers for each other and returns the sum

for a in num:

count +=a

return count

#####################################################################

file = open("glove.6B.50d.txt", "r")

words = open('words.txt','r')

chosen = input('Chose 1 for binary search and 2 for hash tables:')

print()

type(chosen)

if chosen=='1':

start = time.time()

print('you picked binary tree')

print()

print('Building binary search tree:')

print()

startbuild = time.time()

T=buildtree(file)

endbuild = time.time()

print('Height:', height(T))

print('Running time for binary search tree construction:',endbuild-startbuild)

print()

print('Reading word file to determine similarities')

print()

print('Word similarities found:')

print()

similarT(T,words)

end = time.time()

print()

print('Running time for binary search tree query processing:',end - start)

elif chosen =='2':

start = time.time()

print('you picked hash tables')

print()

print('Building hash table with chaining:')

print()

startbuild = time.time()

H=buildhash(file)

endbuild = time.time()

print()

print('Total number of elememts:',H.num\_items)

print('Final table size:',len(H.item))

print('Load factor:',loadFactor(H))

dev= deviation(H)

empty =emptyLists(H)

print('Percent of empty lists:',round((empty/len(H.item)\*100),2),'%')

print('Standard deviation:',dev)

print()

print('Running time for hash table construction:',endbuild- startbuild)

print()

similarities(H,words)

end = time.time()

print('Running time for hash table query processing:',end -start)

else:

print(' you did not choose 1 or 2')

file.close()

words.close()

I Ericka Najera certify that this project is entirely my own work. I wrote, debugged, and teste 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.

