Emmanuel Alvarez

80567137

**Introduction**

This is a report of the lab #5. The objective of this lab is to know the similarity among two random words using a text file that has a lot of words with 50 floating numbers that represent its vectors. To accomplish it, I used two different data structures: binary search tree and hash table with chaining.

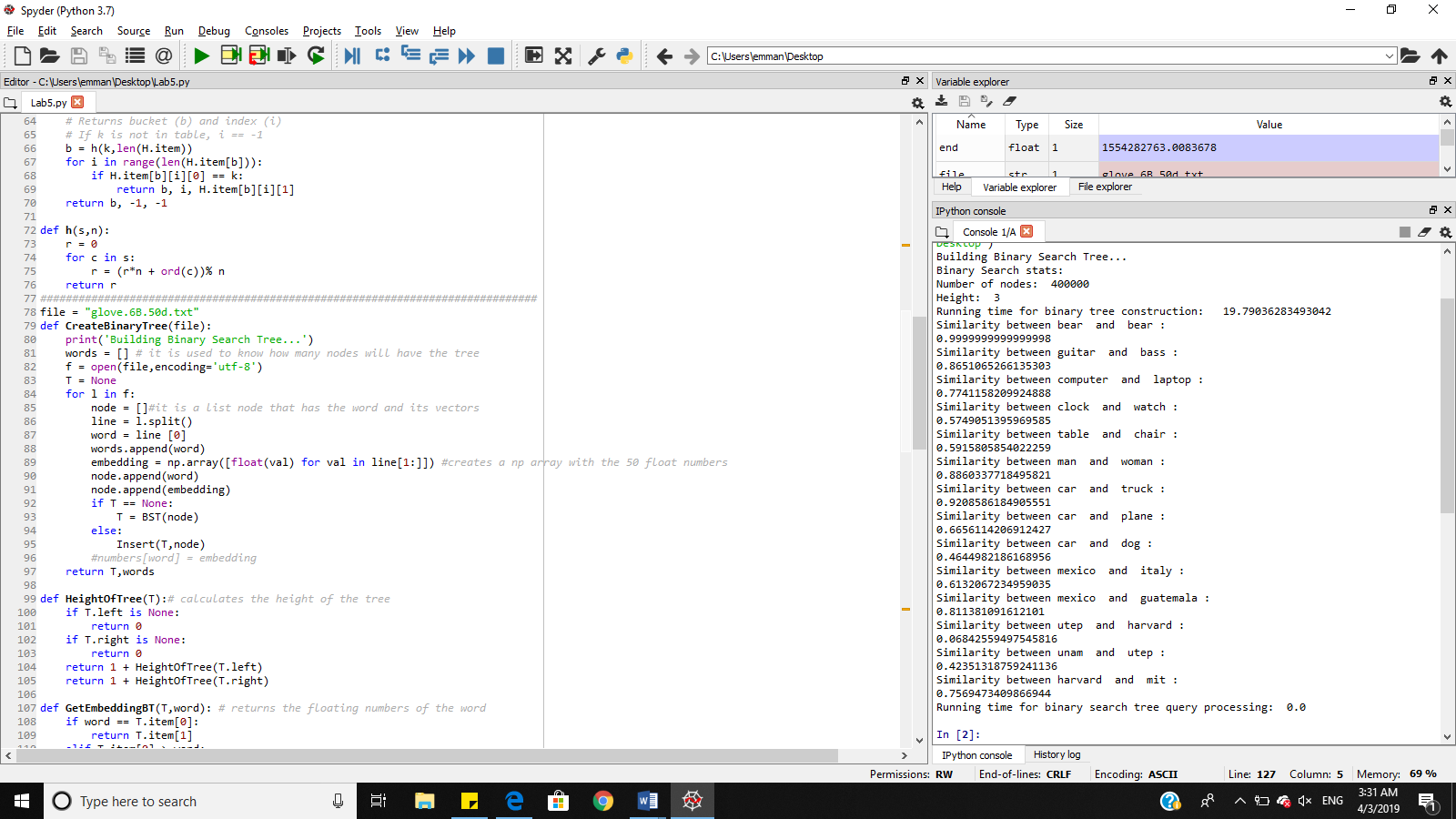
**Proposed solution and design implementation**

The first step that I did was to create the binary search tree reading each line of the text file and adding the word with its 50 floating numbers as a one node of the tree. Then, to get the similarity among two words a created another method that calculates it using the formula sim(word0,word1) = e0\*e1/|e0||e1|. What I did was to create a method that finds the word desired and returns its vectors and then, I just applied the formula. Then, I printed out the running time for creating the binary search tree (an average was from 19 to 22 seconds), the number of nodes (400,000), the height of the tree(3), and finally the running time for query processing(it was to little, in average was from 0 to 1 seconds).

On the other hand, I created a hash table with chaining in order to organize the words and its vectors. To accomplish it, I was adding each word to its correct place in the hash table, but before adding it, I had to check if the load factor was equal or greater than 1. If this condition was true, I duplicated the size of the hash table and reinserted all the words and its vectors to the new hash table. Same as binary search tree, I created a different method that finds the word desired and returns its vectors in order to calculate the similarity of both words.

**Experimental results**

I created a text file with the words that I wanted to compare and, in order to know if this code was working appropriately, I used pair of words that has a lot of similarity and pair of words that do not have any similarity. The results were the following:



**Conclusion**

The hash tables are faster than the binary search tree since they can have a constant running time complexity.

Appendix

"""

Author: Emmanuel Alvarez

Instructor: Olac Fuentes

Last Day of modification: 04-03-2019

This code reads a text file with words and its vectors in order to order them in a binary search tree and a hash table.

Then, using formulas, it calculates the similarity among two random words and the running time of both algorithms

"""

import numpy as np

import math

import time

class BST(object):

# Constructor

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

self.item = item

self.left = left

self.right = right

def Insert(T,newItem):

if T == None:

T = BST(newItem)

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

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

else:

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

return T

def Delete(T,del\_item):

if T is not None:

if del\_item < T.item:

T.left = Delete(T.left,del\_item)

elif del\_item > T.item:

T.right = Delete(T.right,del\_item)

else: # del\_item == T.item

if T.left is None and T.right is None: # T is a leaf, just remove it

T = None

elif T.left is None: # T has one child, replace it by existing child

T = T.right

elif T.right is None:

T = T.left

else: # T has two chldren. Replace T by its successor, delete successor

m = Smallest(T.right)

T.item = m.item

T.right = Delete(T.right,m.item)

return T

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

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([])

def InsertC(H,k,l):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

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

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

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

def h(s,n):

r = 0

for c in s:

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

return r

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

file = "glove.6B.50d.txt"

def CreateBinaryTree(file):

print('Building Binary Search Tree...')

words = [] # it is used to know how many nodes will have the tree

f = open(file,encoding='utf-8')

T = None

for l in f:

node = []#it is a list node that has the word and its vectors

line = l.split()

word = line [0]

words.append(word)

embedding = np.array([float(val) for val in line[1:]]) #creates a np array with the 50 float numbers

node.append(word)

node.append(embedding)

if T == None:

T = BST(node)

else:

Insert(T,node)

#numbers[word] = embedding

return T,words

def HeightOfTree(T):# calculates the height of the tree

if T.left is None:

return 0

if T.right is None:

return 0

return 1 + HeightOfTree(T.left)

return 1 + HeightOfTree(T.right)

def GetEmbeddingBT(T,word): # returns the floating numbers of the word

if word == T.item[0]:

return T.item[1]

elif T.item[0] > word:

return GetEmbeddingBT(T.left,word)

else:

return GetEmbeddingBT(T.right,word)

file2 = "Words.txt"

def ReadCompareWords(T,file2):

f = open(file2)

for l in f:

words = [] # saves the words to be compare on each line of the text file

line = l.split()

words.append(line[0])

words.append(line[1])

print('Similarity between',words[0],' and ', words[1],': ')

WordSimilarity(T,words[0],words[1])#calculates the similarity among the two words

def WordSimilarity(T,word1,word2):

embeddingWord1 = GetEmbeddingBT(T,word1)#gets the floating numbers of the first word

embeddingWord2 = GetEmbeddingBT(T,word2)

product = embeddingWord1 \* embeddingWord2#multiplies all the floating numbers of the first word with the floating numbers of the second word

dotProduct = 0

magnitude1 = 0

magnitude2 = 0

sum1 = 0

sum2 = 0

magnitude1 = 0

magnitude2 = 0

for i in range(len(product)):

dotProduct += product[i]

sum1 += embeddingWord1 [i] \* embeddingWord1 [i]

sum2 += embeddingWord2 [i] \* embeddingWord2 [i]

magnitude1 = math.sqrt(sum1)

magnitude2 = math.sqrt(sum2)

similarity = dotProduct / (magnitude1 \* magnitude2)

#return similarity

print(similarity)

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

def CreateHashTable(file):

print('Building hash table with chaning... ')

f = open(file,encoding = 'utf-8')

H = HashTableC(89)

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

numItems = 0

for l in f :

node = []

line = l.split()

word = line[0]

embedding = np.array(float(val) for val in line[1:])

if numItems == len(H.item):# checks if the load factor is equal to 1

H = ReSize(H,embedding)#duplicates the size of the old hash table

InsertC(H,word,embedding)

numItems += 1

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

print('Load factor:',numItems/len(H.item))

return H

def ReSize(oldHash,embedding):

newHash = HashTableC(len(oldHash.item)\*2+1)#dublicates the size of the old hash table

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

for j in range(len(oldHash.item[i])):

InsertC(newHash,oldHash.item[i][j][0],embedding)#copies all the elements to the new hash table in the correct order

return newHash

def GetEmbeddingHash(H,word):#returns the floating numbers of the words in the hash table

bucket,index,embedding = FindC(H,word)

return H.item[bucket][index][0]

#H = CreateHashTable(file)

#print(GetEmbeddingHash(H,'the'))

start = time.time()

T,nodes = CreateBinaryTree(file)

end = time.time()

totalTime = end - start

print('Binary Search stats:')

print('Number of nodes: ', len(nodes))

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

print('Running time for binary tree construction: ', totalTime)

start = time.time()

ReadCompareWords(T,file2)

end = time.time()

totalTime= end - start

print('Running time for binary search tree query processing: ', totalTime)

“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