Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: April 2, 2019

Instructor: Olac Fuentes

Assignment: Lab 5 NLP

TA: Anindita Nath & Maliheh Zaragan

**Introduction:**

In python, you can read word embeddings representing words by vectors of floating-point numbers such that if two words are like each other, that particular embedding will also be similar. My task for this lab to use see the running times of two implantations of the tables to retrieve two words and determine the floating-point number of those words. The first implementation should use a binary search tree(BST) while the second one should use a hash table with chainings(HTC).

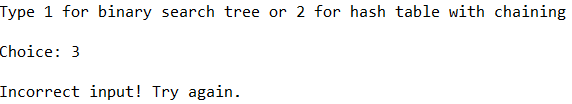
**Proposed solution design and implementation:**

1. Before I begin, I must find a way to ask the user to input either one or two to access the different runs for the implementation. This should not be hard at all, as I can just implement an integer as an input, if 1, it will take to the run for BST comparison, 2 will take the user to HTC comparison, and literally anything else will terminate the program with an error message.
2. BSTComparison: Before I even think about adding any functions to my program, I need to find a way to go through the given file 'glove.6B.50d.txt', and to be able to call it every time I open the program. Once I have that in place, I also need to implement some statistic information to be displayed to the user, such as the number of nodes, the height of the tree, as well the running time taken to build the BST. All three of these methods can be easily found using different methods found from the previous lab to include in this program. The next task is to compare a series of words using this text file so that it displays the float number, which my guess could be made by using a separate method to check the two words letter y letter until a value is calculated. Finally, I must print the running time it took for the entire operation to occur, which can be made by easily added by putting a start time at the beginning of the BST method, and ending once everything has been calculated.
3. HTCComparison: Before I even think about adding any functions to my program, I need to find a way to go through the given file 'glove.6B.50d.txt', and to be able to call it every time I open the program. Once I have that in place, I also need to implement some statistic information to be displayed to the user, such as the initial table size, the final table size, the load factor, the percentage of empty lists, the standard deviation of the lengths of the lists, and the running time taken to build the HTC. All of these methods can must be found using different methods found from the code given to us on the class webpage. The next task is to compare a series of words using this text file so that it displays the float number, which my guess could be made by using a separate method to check the two words letter y letter until a value is calculated. Finally, I must print the running time it took for the entire operation to occur, which can be made by easily added by putting a start time at the beginning of the HTC method, and ending once everything has been calculated.

**Experimental results**:

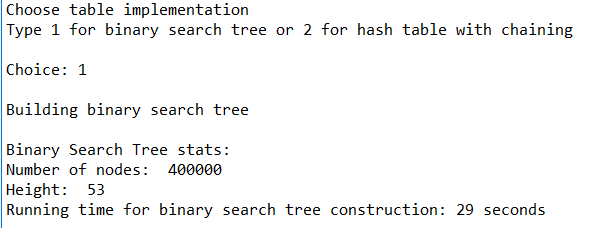
NOTE: For all of my methods, I used the words being compared given in the handout for the lab.

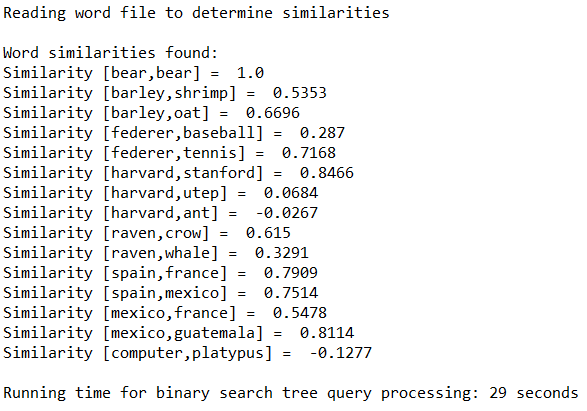
1. As mentioned above, I implemented a way to ask the user for two options for either BST or HTC comparisons, otherwise, literally anything will give an error message and terminate the program. An example of the error message is shown below.



1. BSTComparison: As mentioned before, the BST comparison will begin once the user selects 1 once prompted to do so. For finding the number of nodes, I used a method called CountNodesBST that will go through each node of the BST, and keep a counter as it adds them until the end. For the height, I used a method called maxDepthBST that will search through all of the branches of the tree until it reaches the farthest node from the root, while every time it searches for a new depth, it will add one until the bottom has been reached. For finding the similar words, I used a method called simulationBST that calculated the magnitude of the two given words as well as the dot product and found the floating-point number by dividing the dot product by the product of the magnitude of the two given words. For my running time of my method, I placed a starting point at the beginning of my method, and an end point at the end of the method, that way it will calculate the entirety of the time it took to calculate the running time.

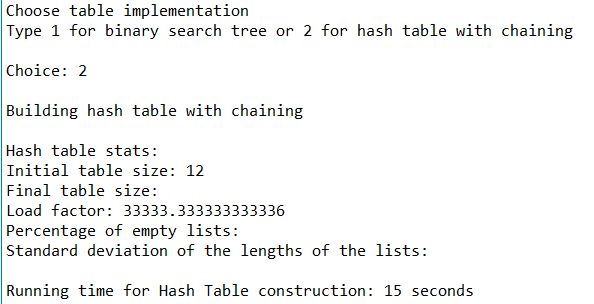
My results for the stats shown below:



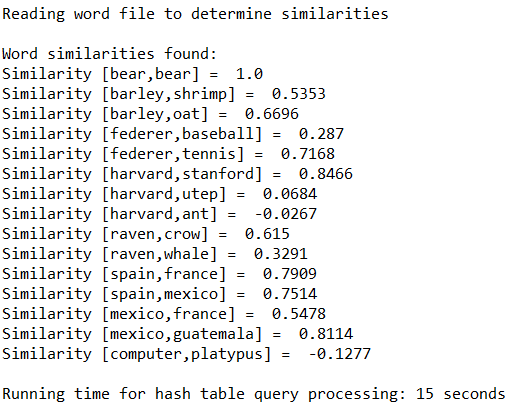
My results for the word comparison as well as the time it took to complete it shown below:

1. HTCComparison: As mentioned before, the HTC comparison will begin once the user selects 2 once prompted to do so. For finding the initial size of the hash table, I separated the initial number needed to create the table and printed it to the user. For the load factor, I used the same method found in the example code given by the professor and used it to find my number. Unfortunately, I wasn’t able to find the final table, the percentage of empty lists, or the standard deviation of the length of the lists at the time of this document. For finding the similar words, I used a method called simulationHTC that calculated the magnitude of the two given words as well as the dot product and found the floating-point number by dividing the dot product by the product of the magnitude of the two given words. For my running time of my method, I placed a starting point at the beginning of my method, and an end point at the end of the method, that way it will calculate the entirety of the time it took to calculate the running time.

My results for the stats shown below:



My results for the word comparison as well as the time it took to complete it shown below:



**Running times Table**

|  |  |  |
| --- | --- | --- |
| Attempt | BST | HTC |
| 1 | 28 sec | 16 sec |
| 2 | 29 sec | 15 sec |
| 3 | 27 sec | 17sec |

As seen above, the method with the best times overall was the method that compared the words with hash tables. This is not surprising as traversing through a hash table is easier than that of a binary tree, since a hash table has an easier way to check where objects are compared through traversing through various branches of the binary tree. It is also important to note that it was important to check the time complexity of these two methods to prove which one is better and faster to use overall.

**Conclusions**:

With this lab, I was able to learn to code better using the Python language, including using algorithms to find create and modify binary trees as well as hash tables with chaining to my advantage. I was also able to learn to solve different problems by using binary trees and hash tables throughout my lab, including to determine the value in which two words are either similar or not.

**Appendix :**

**NLP.py**

"""

Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: April 1, 2019

Instructor: Olac Fuentes

Assingment: Lab 5 Natural Language Processing

TA: Anindita Nath & Maliheh Zaragan

Purpose: to implement hash tables in order to compare its running times and

to succesfully find the comparison of two given words.

"""

#Imports various tools to help us calculate the hash tables to be used in this lab

import math

import time

# This class in the program is used to create objects of BTrees, or binary trees

class BST(object):

#Creates the Constructor

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

self.item = item

self.left = left

self.right = right

# Builds a hash table of a certain size

class HashTableC(object):

# Creates the Constructor

def \_\_init\_\_(self,size,num\_items=0):

self.item = []

self.num\_items = num\_items

if num\_items//size==1:

size = (size\*2)+1

for i in range(size):

self.item.append([])

#Method that caluclates the dot product, and returns it

def dotProduct(e0,e1):

#total starts at 0

total = 0

#For i in length of e0, it will add total to the prodcut of the current e0 and e1 values

for i in range(len(e0)):

total += e0[i]\*e1[i]

#Returns the total value

return total

#Method that returns the magnitude used in the simulation methods

def Magnitude(ex):

return math.sqrt(dotProduct(ex,ex))

#Method used in the BST used to insert new items into the tree

def InsertBST(T,newItem):

#If T is none, T will add a new item inside of it

if T == None:

T = BST(newItem)

#If the current value of T is greater than the value of the nrew item, it will insert on the left side of the tree

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

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

#If the current value of T is less than the value of the nrew item, it will insert on the right side of the tree

else:

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

#Returns the value of T

return T

#Method that counts the nodes of a BST

def CountNodesBST(T):

#Creates a new value called count that starts at 0

count = 0

#If T is not empty, it will add one value to count

if T is not None:

count += 1

#Otherwise it will add 0 to count

else:

return 0

#Returns the value of count with thw number of nodes in the right and left children

return count + CountNodesBST(T.right) + CountNodesBST(T.left)

#Method that counts the maximum depth of a certain BST

def maxDepthBST(T):

#If T is empty, it will return 0

if T is None:

return 0

#Otherwise it will compute the depth of each subtree, and use the largest one to find its depth

else :

leftDepth = maxDepthBST(T.left)

rightDepth = maxDepthBST(T.right)

if (leftDepth > rightDepth):

return leftDepth+1

else:

return rightDepth+1

#Method that returns the address of an item k, or None if not present in a BST

def FindBST(T,k):

#If T is none or the initial item is k, it returns the value of T.item[1]

if T is None or T.item[0][0] == k:

return T.item[1]

#If the current item is less than k, it will return the value found from the right side of the tree

if T.item[0][0]<k:

return FindBST(T.right,k)

#Otherwise, it will return the value found from the lest side of the tree

return FindBST(T.left,k)

#Method that finds the value of similarities between two words in the BST

def simulationBST(w0,w1,T):

#Creates the value e0, which will locate the first word in the BST

e0 = FindBST(T,w0)

#Creates the value e1, which will locate the second word in the BST

e1 = FindBST(T,w1)

#Returns the dot product form the two words, and returns it to an integer dot

dot = dotProduct(e0,e1)

#Finds the magnitude of the two e values

Me0 = Magnitude(e0)

Me1 = Magnitude(e1)

#Calculates the result value of the similarities betweeen the two words

result = (dot)/(Me0\*Me1)

#Returns the result to BSTComparison

return result

#Method that inserts k in the appropriate bucket other wise it does nothing if k is already in the table

def InsertHTC(H,k,l):

#Finds the slot for that item

a = hashTable(k[0][0],len(H.item))

#Adds that item to the HTC

H.item[a].append([k[0][0],l])

#Increases the value of the number of items in the HTC

H.num\_items += 1

#Method that returns the bucket and index of an item

def FindHTC(H,k):

#Finds the slot for that item

a = hashTable(k,len(H.item))

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

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

return H.item[a][i][1]

#Returns -1 if the item is not found

return a, -1, -1

#Method that calculates the placement of a particular item into a slot of an HTC, and returns that value

def hashTable(s,n):

value = 0

for c in s:

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

return value

#Method that calculates the load factor of the HTC

def LoadFactorHTC(H):

#Creates an empty decimal number called count

count = 0.0

#For every item in the hash table, count will add itself to the length of i

for i in H.item:

count +=len(i)

#Returns the value of count divided vy the length of the item in the hash table

return count/len(H.item)

#Method that finds the value of similarities between two words in the HTC

def simulationHTC(w0,w1,H):

#Creates the value e0, which will locate the first word in the HTC

e0 = FindHTC(H,w0)

#Creates the value e1, which will locate the second word in the HTC

e1 = FindHTC(H,w1)

#Returns the dot product form the two words, and returns it to an integer dot

dot = dotProduct(e0,e1)

#Finds the magnitude of the two e values

Me0 = Magnitude(e0)

Me1 = Magnitude(e1)

#Calculates the result value of the similarities betweeen the two words

result = (dot)/(Me0\*Me1)

#Returns the result to HTCComparison

return result

#The method that will create a Binary Search Tree and compare different words and determine how similar they are.

def BSTComparison():

#Creates an empty BST called T

T = None

#Opens the text file 'glove.6B.50d.txt', which willl be used to determine if the words being used are similar or not.

f = open('glove.6B.50d.txt',encoding='utf-8')

#Goes through the entire text file and inserts the items inside of a BST

for line in f:

lines = line.split()

name = [lines[0]]

nums = []

for i in range(len(lines)-1):

nums.append(float(lines[i+1]))

p = [name,nums]

T = InsertBST(T,p)

print()

#Prints the Binary Search Tree stats, including the number of nodes, the height, and the running time it took to create the binary search tree

print("Binary Search Tree stats:")

#Returns the number of nodes

print("Number of nodes: ",CountNodesBST(T))

#Returns the height of the Binary Search Tree

print("Height: ",maxDepthBST(T))

#Starts the timer for the constriuction of the BST

elapsed\_time\_CONSTRUCTION\_BST = time.time()-start

#Returns the time it took to create the BST

print("Running time for binary search tree construction:", round(elapsed\_time\_CONSTRUCTION\_BST),"seconds")

print()

#Alerts the user that it is detecting the similarities between the words

print("Reading word file to determine similarities")

print()

#Prints the words being compared, and the value of its comparison

print("Word similarities found:")

print("Similarity [bear,bear] = ",round(simulationBST('bear','bear',T),4))

print("Similarity [barley,shrimp] = ",round(simulationBST('barley','shrimp',T),4))

print("Similarity [barley,oat] = ",round(simulationBST('barley','oat',T),4))

print("Similarity [federer,baseball] = ",round(simulationBST('federer','baseball',T),4))

print("Similarity [federer,tennis] = ",round(simulationBST('federer','tennis',T),4))

print("Similarity [harvard,stanford] = ",round(simulationBST('harvard','stanford',T),4))

print("Similarity [harvard,utep] = ",round(simulationBST('harvard','utep',T),4))

print("Similarity [harvard,ant] = ",round(simulationBST('harvard','ant',T),4))

print("Similarity [raven,crow] = ",round(simulationBST('raven','crow',T),4))

print("Similarity [raven,whale] = ",round(simulationBST('raven','whale',T),4))

print("Similarity [spain,france] = ",round(simulationBST('spain','france',T),4))

print("Similarity [spain,mexico] = ",round(simulationBST('spain','mexico',T),4))

print("Similarity [mexico,france] = ",round(simulationBST('mexico','france',T),4))

print("Similarity [mexico,guatemala] = ",round(simulationBST('mexico','guatemala',T),4))

print("Similarity [computer,platypus] = ",round(simulationBST('computer','platypus',T),4))

print()

#Starts the timer for the query processing of the BST

elapsed\_time\_QUERY\_BST = time.time()-start

#Returns the timer for the query processing of the BST

print("Running time for binary search tree query processing:", round(elapsed\_time\_QUERY\_BST),"seconds")

#The method that will create a Hash Chaining Table and compare different words and determine how similar they are.

def HTCComparison():

#Creates an empty Hash Table with 12 items

IV = 12

H = HashTableC(IV)

#Opens the text file 'glove.6B.50d.txt', which willl be used to determine if the words being used are similar or not.

f = open('glove.6B.50d.txt',encoding='utf-8')

#Goes through the entire text file and inserts the items inside of a HTC

for line in f:

lines = line.split()

name = [lines[0]]

nums = []

for i in range(len(lines)-1):

nums.append(float(lines[i+1]))

p = [name,nums]

InsertHTC(H,p,p[1])

print()

#Prints the Hash Table stats, including the initial table size, the final table size, the load factor, percentage of empty lists, and standard deviation of the lengths of the lists

print("Hash table stats:")

print("Initial table size:",IV)

print("Final table size:")

print("Load factor:",LoadFactorHTC(H))

print("Percentage of empty lists:")

print("Standard deviation of the lengths of the lists:")

print()

#Starts the timer for the constriuction of the HTC

elapsed\_time\_CONSTRUCTION\_HTC = time.time()-start

#Returns the time it took to create the HTC

print("Running time for Hash Table construction:", round(elapsed\_time\_CONSTRUCTION\_HTC),"seconds")

print()

#Alerts the user that it is detecting the similarities between the words

print("Reading word file to determine similarities")

print()

#Prints the words being compared, and the value of its comparison

print("Word similarities found:")

print("Similarity [bear,bear] = ",round(simulationHTC('bear','bear',H),4))

print("Similarity [barley,shrimp] = ",round(simulationHTC('barley','shrimp',H),4))

print("Similarity [barley,oat] = ",round(simulationHTC('barley','oat',H),4))

print("Similarity [federer,baseball] = ",round(simulationHTC('federer','baseball',H),4))

print("Similarity [federer,tennis] = ",round(simulationHTC('federer','tennis',H),4))

print("Similarity [harvard,stanford] = ",round(simulationHTC('harvard','stanford',H),4))

print("Similarity [harvard,utep] = ",round(simulationHTC('harvard','utep',H),4))

print("Similarity [harvard,ant] = ",round(simulationHTC('harvard','ant',H),4))

print("Similarity [raven,crow] = ",round(simulationHTC('raven','crow',H),4))

print("Similarity [raven,whale] = ",round(simulationHTC('raven','whale',H),4))

print("Similarity [spain,france] = ",round(simulationHTC('spain','france',H),4))

print("Similarity [spain,mexico] = ",round(simulationHTC('spain','mexico',H),4))

print("Similarity [mexico,france] = ",round(simulationHTC('mexico','france',H),4))

print("Similarity [mexico,guatemala] = ",round(simulationHTC('mexico','guatemala',H),4))

print("Similarity [computer,platypus] = ",round(simulationHTC('computer','platypus',H),4))

print()

#Starts the timer for the query processing of the HTC

elapsed\_time\_END\_HTC = time.time()-start

#Returns the timer for the query processing of the HTC

print("Running time for hash table query processing:", round(elapsed\_time\_END\_HTC),"seconds")

#Starts the clock to print the runtime of the find method at the end

start = time.time()

#Asks the user to choose a binary search tree or hash table with chaining

print('Choose table implementation')

print('Type 1 for binary search tree or 2 for hash table with chaining')

#Sets a variable 'x' to be made for the input section of the program

x = int(input('Choice: '))

print()

#If the user selects 1, it will find the words using a binary search tree

if x == 1:

print('Building binary search tree')

BSTComparison()

#If the user selects 2, it will find the words using a hash table

elif x == 2:

print('Building hash table with chaining')

HTCComparison()

#If anything else is inputed, the program will end with an error message.

else:

print('Incorrect input! Try again.')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 provide inappropriate assistance to any student in the class.

