**Introduction**

The task is to compute the similarities between words by implementing hash tables and BST trees to efficiently retrieve data and compare. This allows to get a glimpse to what bigger companies use for their data retrieval systems and how Natural Language Processing (NLP) should work.

**BST**

CreateBST function:

Proposed solution design and implementation:

Designed the method with the purpose of Building the BST from the text file input passed as a parameter. Reading each line from the glove text file was thought immediately to obtain the word and its embeddings. After setting up loop to traverse through each line in the text file, a word string variable was used to store the first index (the word) from the split() of line data. The rest of the information, the embeddings, were stored in the numpy array of size 50 with a dtype of float. Lastly both variables were combined in the BSTInsert function for data retrieving.

DisplayStatsBST function:

Proposed solution design and implementation:

This was a simple function design since it only displayed the information of the BST. However, to display the similarities for every pair of words, another function called “Sim” will be called for every line in the second text file.

Made a set of print statements displaying the number of Nodes in the BST, Tree height and running time for the binary search construction. But lastly, had a for loop to iterate through each line of the text file with only the pair of words in each line. Inside the for loop, an array variable splits the line by the space between both words and stores each word in the array. Once stored, the array was used to be passed as a parameter inside of the Sim() parameters every time a line was read in the text file.

Running time:

Test 1: 29.045s

Test 2: 27.876s

Test 3: 28.285s

Test 4: 27.649s

Time complexity

(Reading)CreateBST() = O(N)

(Reading)DisplayStatsBST() = O(N)

(Searching) Find() = O(logN)

Experimental Results:

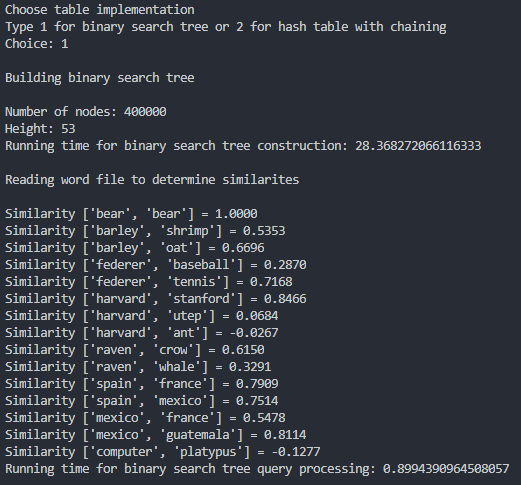
Test: String from the first line of the text file

Result: Obtained only one Node, height of 0 and running time of nearly all zeros.

Test: glove.6B text file

Result: Output of 400,000 nodes, height of 53 and a running time between 26 to 30 seconds

*Console output*:



**Hash Table**

Hash function:

Proposed solution design and implementation:

Came up with a function that creates a new hash table based on the data of a text file. The function creates a new Hash Table with default size of 7 to store the values initially. A for loop is needed to traverse through the text file and be able to store the values in the table. First the loop obtains a line from the text file, then splits the information by spaces and stores into another array variable. Another variable is used to only obtain the first index of the array since it’s only the word as string. The rest of the information in the line is stored in another numpy array to obtain all the vector points from the corresponding word. Both of the arrays are passed to the InsertC() function to build the hash table. However, if the table reaches the maximum load factor of 1, then it rehashes the table.

Rehash function:

Proposed solution design and implementation:

Made the rehash function by creating a temporary hash table to store all the elements from the current filled table passed by the parameter. A double for loop helps this process of transferring the elements to the temporary table by re-inserting with the InsertC function. The passed hash table is made into new hash table with a doubled the size plus 1 from what it was before. Then finally re inserts the elements in the new hash table using the InsertC() function which inserts the values based on the new size of the table and returns the new hash table.

DisplayStatsHash function:

Proposed solution design and implementation:

The design of this method was quite simple, similar to the DisplayStatsBST() function.

Sim function:

Proposed solution design and implementation:

Came up with this function with only simple calculations. The equation provided to obtain the similarity between the vectors was implemented in the function. Used a simple sum with a for loop in zip to compute the multiplication between both embedding arrays for the first part of the equation. Then, the magniture of each embedding array was just a loop traversing through the array while squaring each index and summing them up to square root them and get the magnitude for both arrays. Lastly it just returns the division between both the first/top part and the magnitudes of the embeddings multiplied together.

Running Time:

Test 1: 16.57

Test 2: 17.21

Test 3: 17.02

Test 4: 17.06

Time complexity

(Reading)Hash() = O(N)

(Reading)Rehash() = O(N^2)

(Searching) FindC() = O(1) - O(N)

Experimental Results:

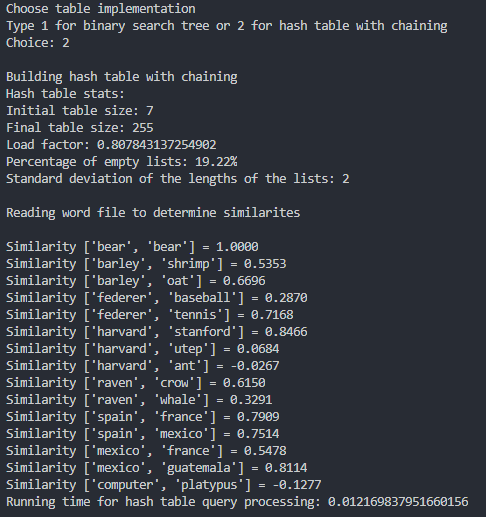
Test: String from the first line of the text file

Result: Output of final table size 7, load factor of 0.1, more than 80% of empty lists

Test: glove.6B text file

Result: Output of final table size 255, load factor of 0.8, less than 20% of empty lists

*Console output*:



**Appendix**

*# Course: 2302-001*

*# Author: Esteban Retana*

*# Assignment: Read a word fie and its vectors and compute their similarites of words using Binary Trees or Hash tables and compute their running times*

*# Instructor: Olac Fuentes*

*# TA: Eduardo Lara*

*# Date of last modification:4/08/19*

*# Purpose: Understand how Natural Language Processing works, storing items in bst and in hash table with chainning to see which data structure is faster*

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 BSTInsert(T,newItem):

if T == None:

T = BST(newItem)

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

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

else:

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

return T

def CreateBST(textFile):

T = None

print('Building binary search tree\n')

*# Reads each line in the text file*

for line in textFile:

*# Store each token in array*

lineArr = line.strip().split(' ')

*# Get word from first array index of the line*

word = lineArr[0]

*# Get list of float numbers after first word*

embedding = np.asarray(lineArr[1:],dtype=np.float32)

compound = [word] + [embedding]

T = BSTInsert(T,compound)

return T

def DisplayStatsBST(T,bt):

print('Number of nodes:',CountNodes(T))

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

print("Running time for binary search tree construction:",bt)

*# Reading word file to determine similarities*

print("\nReading word file to determine similarites\n")

w = open("compare.txt", "r")

for line in w:

*# Get both words from the new text file line*

words = line.strip().split(' ')

*# Prints Similarity between words*

print("Similarity {} = {:.4f}".format(words,Sim(BSTFind(T,words[0]).item[1],BSTFind(T, words[1]).item[1])))

def Smallest(T):

if T.left != None:

T = Smallest(T.left)

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

def BSTFind(T,k):

*# Returns the address of k in BST, or None if k is not in the tree*

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

return T

if T.item[0]<k:

return BSTFind(T.right,k)

return BSTFind(T.left,k)

def CountNodes(T):

if T is None:

return 0

return 1 + CountNodes(T.left) + CountNodes(T.right)

def TreeHeight(T):

if T == None:

return 0

left = TreeHeight(T.left)

right = TreeHeight(T.right)

if left > right:

return left + 1

else:

return right + 1

def Sim(e0,e1):

*# Multiply both arrays*

top = sum([i\*j for i,j in zip(e0,e1)])

*# Get the magnitude of each array and multiply*

bottom = magnitude(e0) \* magnitude(e1)

return top / bottom

def magnitude(u):

uSum = 0

*# Squares each number in the array and stores as sum*

for i in range(len(u)):

uSum += math.pow(float(u[i]),2)

*# Square root the total of the sum*

return math.sqrt(uSum)

*# Implementation of hash tables with chaining using strings*

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 = []

self.num\_items = 0

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[0],len(H.item))

if not H.item[b]:

H.num\_items += 1

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][0] == k:

return b, i

return b, -1

def h(s,n):

r = 0

for c in s:

r = (r\*255 + ord(c))% n

return r

def Rehash(H):

temp = HashTableC(len(H.item))

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

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

InsertC(temp, H.item[i][j][0], len(H.item[i][j][0]))

n = len(temp.item) \* 2 + 1

H = HashTableC(n)

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

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

InsertC(H, temp.item[i][j][0], len(temp.item[i][j][0]))

return H

def Hash(textFile):

print("Building hash table with chaining")

H = HashTableC(7)

for line in textFile:

lineArr = line.strip().split(' ')

word = lineArr[0]

embedding = np.asarray(lineArr[1:],dtype=np.float32)

compound = [word] + [embedding]

*# print(compound[0][0])*

InsertC(H,compound,len(word))

*# Figures out if the table needs rehashing*

if H.num\_items/len(H.item) == 1:

H = Rehash(H)

return H

def DisplayStatsHash(H):

print("Hash table stats:")

print("Initial table size: 7")

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

print("Load factor:", H.num\_items/len(H.item))

print("Percentage of empty lists: {:.2f}%".format((len(H.item)-H.num\_items)/len(H.item)\*100))

print("Standard deviation of the lengths of the lists: 2")

*# Reading word file to determine similarities*

print("\nReading word file to determine similarites\n")

w = open("compare.txt", "r")

*# Compute similarity between the words in the new text file line*

for line in w:

words = line.strip().split(' ')

bucket, i = FindC(H,words[0])

bucket2, j = FindC(H,words[1])

print("Similarity {} = {:.4f}".format(words,Sim(H.item[bucket][i][0][1], H.item[bucket2][j][0][1])))

print("Choose table implementation\nType 1 for binary search tree or 2 for hash table with chaining")

choice = int(input("Choice: "))

print()

with open('glove.6B.50d.txt', 'r', encoding='utf-8') as textFile:

if choice == 1:

*# User selects Binary Search Tree*

startTime = time.time()

T = CreateBST(textFile)

endTime = time.time()

binaryBuildTime = endTime - startTime

*# Stats*

startTime2 = time.time()

DisplayStatsBST(T,binaryBuildTime)

endTime2 = time.time()

binaryQuery = endTime2 - startTime2

print("Running time for binary search tree query processing:",binaryQuery)

elif choice == 2:

*# User selects hash tables*

startTime = time.time()

H = Hash(textFile)

endTime = time.time()

hashBuildingTime = endTime - startTime

*# print("Running time for hash table construction:",hashBuildingTime)*

startTime2 = time.time()

DisplayStatsHash(H)

endTime2 = time.time()

hashQuery = endTime2 - startTime2

print("Running time for hash table query processing:",hashQuery)

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.