Lab 5 Report – CS 2302

This 5th lab consisted of handling large amounts of data into hash tables and binary search trees, in order to compare their efficiency in both building and accessing the structure. The steps to solve the problem were reading the data from a text file, storing that data into the data structure, and with another text file containing two words, compare their similarity using a provided formula.

The text file consisted of about 400,000 lines. Each line beginning with a word, followed by 50 numbers (embeddings) used in the computations to find similarities with other words.

Strings by themselves cannot be stored in either structure, so a unique value had to be assigned. This could have been done by simply adding up the ascii values of each character, but then two strings having the same letters in different order would be assigned the same value. Instead, the following formula was applied to every character in every string to be stored in the hash table: r = (r\*n + ord(c))% n, where r is the unique cumulative value, n the length of the string, and ord(c) the ascii value of the character. For the BST, each ascii value of each character was multiplied by 26 to the *i-th* power (i being the index of the character in the string).

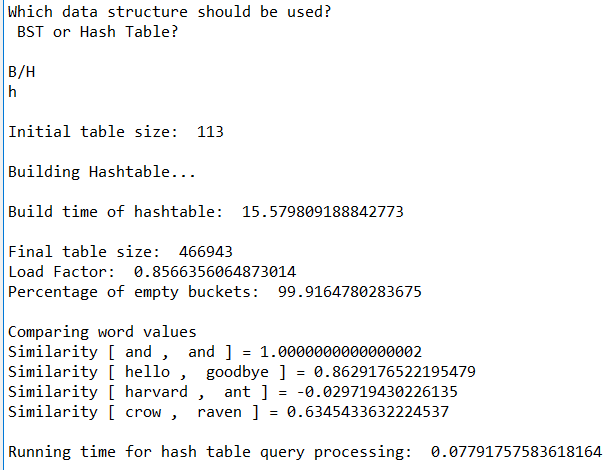
Next is the capturing of the embeddings into a numpy array. This was done by storing each line of the text file into a temporary variable, stripping it of it’s the next line character, and splitting it by the spaces into a list. That list was transferred into a numpy array of size 50. Finally, the value of the string was placed into the first index of a list of size 2, and the numpy array as the second element of said list. To store that list into the hash table, the insert method simply calculated the index by using the value at position 0 of the list of size 2; however, the BST needed to have a string in the first position of the list, so in order to compare the value, the string to decimal method was used.

Once having the data stored in each data structure, computing the similarity of the vectors was a matter of accessing the information and using the provided formula. For the BST, accessing the information was done by modifying the Find method so it searches for the decimal value of the string.

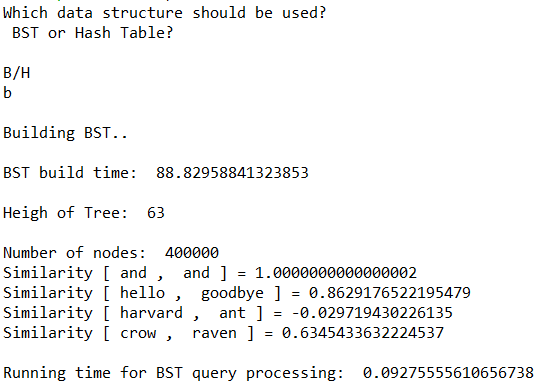
**Experimental Results**

The same input files were used for both tests. The glove file with the raw data, and the second text file with 4 lines (2 words per line) to be compared.

This is the output when selecting hash table as the data structure to be used.



And the BST.



**Notes:**

Both structures have their advantages and disadvantages. A hash table has faster build time but will also have empty buckets and must resize every time the load factor is greater than one. The BST will only create the number of nodes it needs, but the insertion takes longer than the hash table. Both structures had a very similar query processing time, but once the hash table is constructed (O(n)), the accessing time is constant while the BST accessing time will be as large as its height (log(n)).

**Academic Honesty Statement**

“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.”

-Andres Silva.

**Appendix**

# -\*- coding: utf-8 -\*-

'''

Created on Wed Mar 25th 20:55:43 2019

CS 2302 - Andres Silva

> Teacher: Olac Fuentes

> TAs: Anindita Nath & Maliheh Zargaran

> Lab #5

> The purpose of this lab is compare Hashtables vs BSTs in data handling and accessing.

> LAST MODIFIED: APRIL 3rd, 2019

'''

import numpy as np

from hash\_table\_chain\_strings import \*

from bst import \*

import math

import time

def StringToDec(word): #This method gives a unique decimal value to a string

value = 0

for i in range(len(word)):

value += ord(word[i]) \* (26\*\*i) #Add up all the ascii values of each character + 26^index - for a unique value

return value

"""

Hash Table Methods ############################################################

"""

def ReadAndStoreInH(H): #Reads text file with data and stores in dynamic hashtable.

with open("glove.6B.50d.txt" , 'r', encoding="utf8") as f: #Read from file,

for line in f: #For each line,

line = line.strip('\n') #Remove next line character

splitLine = line.split(' ') #Split line into an array

npArray = np.zeros(50) #Initialize numpy array

for i in range(1,50): #Insert every element of line into np array

npArray[i-1] = splitLine[i] #i-0 to start at 0 in the np array and at 1 in the line.

InsertC(H, splitLine[0], npArray) #Insert to H the previous the string and the array,

if H.num\_size > len(H.item): #If more items in hash table than slots, there is a bucket.

newHash = EnlargeC(H) #Make bigger hashtable with same elements.

H.item = newHash.item #Point H to new table.

H.num\_size = newHash.num\_size #Update H's number of items

def DotProductC(H,w0,w1): #Return the dot product of two strings.

product = 0

for i in range(50):

product += FindC(H,w0)[2][i] \* FindC(H,w1)[2][i] #FindC returns a bucket with the npArray at position 2, we access that array at i for each individual element.

return product

def MagnitudeC(H,word): #Return Magnitude of the embeddings of a word.

magnitude = 0

SumOfEmbeddings = 0

for i in range(50):

SumOfEmbeddings += FindC(H,word)[2][i] \*\* 2

magnitude = math.sqrt(SumOfEmbeddings)

return magnitude

def CompareC(H,word0,word1): #Returns similarity of two words by computing the distance between two vectors.

similarity = DotProductC(H,word0,word1) / (MagnitudeC(H,word0) \* MagnitudeC(H,word1))

return similarity

"""

BST Methods ###################################################################

"""

def ReadAndStoreB(B): #Reads big data from text file and stores it in binary search tree.

with open("glove.6B.50d.txt" , 'r', encoding="utf8") as f:#Read file.

for line in f: #For every line in file:

line = line.strip('\n') #Remove next line character.

splitLine = line.split(' ') #Split line into an array.

npArray = np.zeros(50) #Initialize numpy array.

for i in range(1,50): #Store all numerical data of line into np array.

npArray[i-1] = splitLine[i] #i-0 to start at 0 in the np array and at 1 in the line.

tempList = [splitLine[0], npArray] #Create list with string and npArray.

B = InsertB(B,tempList, StringToDec(tempList[0])) #Insert list in BST using numerical value of string.

return B

def DotProductB(B,s0,s1): #Returns dot product of two strings in a BST.

product = 0

w0 = StringToDec(s0) #Get numerical value of words

w1 = StringToDec(s1)

for i in range(50): #FindB returns a list with the npArray at the first index.

product += FindB(B,s0,w0).item[1][i] \* FindB(B,s1,w1).item[1][i] #Multiply and add all of the products of the same entries in the npArrays.

return product

def MagnitudeB(B,word): #Returns the magnitude of the embeddings of a string.

wordVal = StringToDec(word)

magnitude = 0

SumOfEmbeddings = 0

for i in range(50):

SumOfEmbeddings += FindB(B,word,wordVal).item[1][i] \*\* 2

magnitude = math.sqrt(SumOfEmbeddings)

return magnitude

def CompareB(B,word0,word1): #Returns the similarity of two strings in a BST.

similarity = DotProductB(B,word0,word1) / (MagnitudeB(B,word0) \* MagnitudeB(B,word1))

return similarity

"""

Printing and comparing methods

"""

def OptionHash(): #Used when Hashtable is selected as a data structure.

H = HashTableC(113)

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

print("\nBuilding Hashtable...")

start = time.time()

ReadAndStoreInH(H)

end = time.time()

print("\nBuild time of hashtable: ", end - start)

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

print("Load Factor: ", LoadFactor(H))

print("Percentage of empty buckets: ", PercentageOfEmpty(H))

print("\nComparing word values")

start = time.time()

with open("test.txt" , 'r', encoding="utf8") as f: #Read from test file.

for line in f: #For every line,

line = line.strip('\n') #remove next line character,

splitLine = line.split(' ') #Split line into individual strings.

print("Similarity [", splitLine[0], ", ",splitLine[1], "] =", CompareC(H, splitLine[0],splitLine[1])) #Compare the two words used.

end = time.time()

print("\nRunning time for hash table query processing: ", end - start)

def OptionBST(): #Used when BST is selected as a data structure.

B = None

print("\nBuilding BST..")

start = time.time()

B = ReadAndStoreB(B)

end = time.time()

print("\nBST build time: ", end - start)

print("\nHeigh of Tree: ", GetHeight(B))

print("\nNumber of nodes: ", NumberNodes(B))

start = time.time()

with open("test.txt" , 'r', encoding="utf8") as f:

for line in f:

line = line.strip('\n')

splitLine = line.split(' ')

print("Similarity [", splitLine[0], ", ",splitLine[1], "] =", CompareB(B, splitLine[0],splitLine[1]))

end = time.time()

print("\nRunning time for BST query processing: ", end - start)

"""

Ask for input #################################################################

"""

print("Which data structure should be used?\n BST or Hash Table?")

choice = input("B/H\n")

if choice.lower() == 'b':

OptionBST()

elif choice.lower() == 'h':

OptionHash()

else:

print("Invalid Input, please retry.")

# Code to implement a binary search tree with nodes as [string, npArray]

class BST(object):

# Constructor

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

self.item = item

self.left = left

self.right = right

def StringToDec(word): #Calculates a unique numerical value for a given string.

value = 0

for i in range(len(word)):

value += ord(word[i]) \* (26\*\*i) #Add up all the ascii values of each character + 26^index - for a unique value

return value

def InsertB(T,newItem, wordVal): #Modified to compare value of node based on their assigned value by StringToDec.

if T == None:

T = BST(newItem)

elif StringToDec(T.item[0]) > wordVal:

T.left = InsertB(T.left,newItem, wordVal)

else:

T.right = InsertB(T.right,newItem, wordVal)

return T

def FindB(T,k, wordVal): #Modified to compare value of nodes

# Returns the address of k in BST, or None if k is not in the tree on their assigned value by StringToDec.

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

if T != None:

return T

else:

print("Item not in tree.")

if StringToDec(T.item[0]) < wordVal:

return FindB(T.right,k, wordVal)

return FindB(T.left,k, wordVal)

def GetHeight(T):

if T == None: #At a none node, add 0 to our sum.

return 0

else:

left = GetHeight(T.left) #Get height of left and right subtree.

right = GetHeight(T.right)

if right > left: #add 1 to the largest subtree and return int.

return 1 + right

else:

return 1 + left

def NumberNodes(T): # Find number of nodes in the Tree

if T == None:

return 0

else:

left = NumberNodes(T.left)

right = NumberNodes(T.right)

if left == None:

left = 0

if right == None:

right = 0

if T.left is not None:

return 1 + left

if T.right is not None:

return 1 + right

def NumberNodes(T): # Find number of nodes in the Tree

if T != None:

c = 1

if T.left != None:

c += NumberNodes(T.left)

if T.right != None:

c += NumberNodes(T.right)

return c

# 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 = []

for i in range(size):

self.item.append([])

self.num\_size = 0

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])

H.num\_size += 1

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

def EnlargeC(H):

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

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

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

for k in range(len(H.item[i][j]) -1):

InsertC(newHash, H.item[i][j][0], H.item[i][j][1])

return newHash

def LoadFactor(H):

if H.num\_size == 0:

return 0

else:

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

def PercentageOfEmpty(H):

EmptyCounter = 0

if len(H.item) == 0:

return 100

else:

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

if len(H.item[i]) == 0:

EmptyCounter += 1

return (EmptyCounter \* 100) / len(H.item)