Lab5 asked of us to read a file named ‘glove.6B.50d.txt’ and save them into a user chosen implementation. Save into a BST or a hash table that utilizes chaining. If they chose a hash table then we were asked to resize the table when the load factor reaches 1. After saving the data from the file we were then asked to display statistics describing the table or the tree, time to compile, number of items in the structure, etc.. Afterwards we are to read another file containing pairs of words and compute the similarity between said words using the embeddings provided in the glove file.

I began my code by starting with the hash table as I believed it to be the harder of the 2 implementations. The very first thing that I did was begin with the code to read the glove file. Once opening the file I read all the lines at once using the f.readlines() function. I went through each line splitting the values at the spaces and saving the embedding (last 50 items in the 51 items per line). I would then check if the string(the very first item in the line) was a valid string to save. In order to be a valid string It had to be made up of only alphabetical characters and no symbols like \*, &, etc. so if it was a valid string it would be inserted along with it’s embedding into the table, other wise it would be skipped, then I would check if the number of items in the table was equal to the length of the list to see if the load factor would be one. If the load factor was one then I would resize the table, after going through every line I would close the file and return the newly created table.

The next function I had to create was the function ResizeTable which would get a table h and change the size to 2\*len(H.item) + 1. Once that was done it would re hash all items in the original table into the new table and return the address of the new table.

Afterwards I went to computing the similarity of the words. The Function would open a file I names Pairs.txt, save the lines and look for address of the words in each line. Once the words were located in the hash table it would then compare the similarity of the values using a separate function that calculated the dot product, and magnitude of each embedding vector and divide the dot product by the magnitude of the first embedding times the magnitude of the second embedding, resulting in a value in between -1 and 1 that would show the ‘similarity’ of the words.

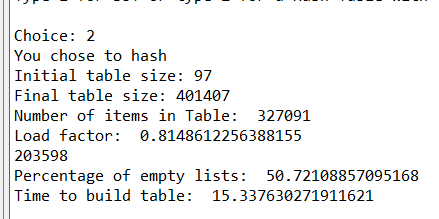
The BST went about the same as the Hash, read the file, insert into Tree if it was a valid item. The real difference was then in the insertion process of the BST where they were sorted by the string and not a number. This resulted in a tree that was sorted alphabetically in insertion process.

The process for finding the similarity went the same as well, finding the words in the structure, then saving their address and calculating the dot product and magnitude and return the value of the previously stated formula.

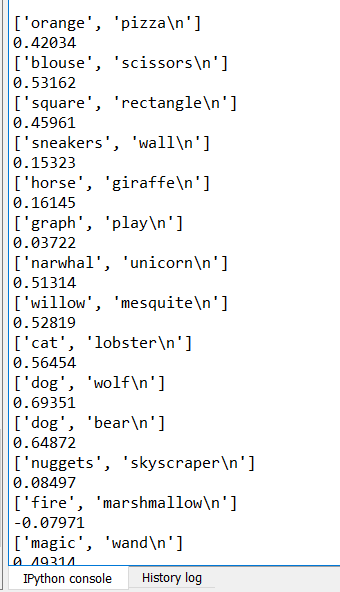
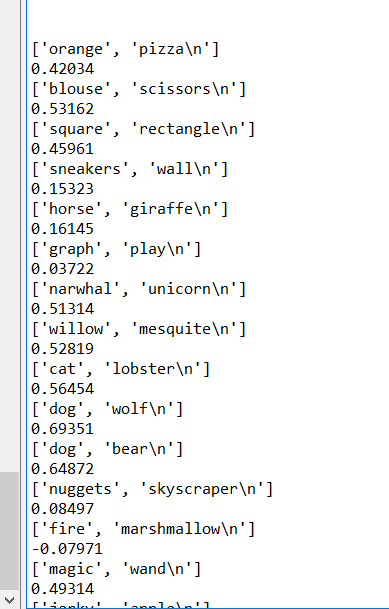
Testing is where this lab became difficult. The first issue I was having was in the comparison of words, I didn’t realize that the \n-new line indicator- was seen as a character in the file reading process

And thus, the proper words were not correctly located when trying to find them in the hash table. After printing out the data being read, I saw that the items inside of the file had a \n in the 2nd word resulting in the improper hashing process, to solve this it was as simple as removing the end character in the string when hashing and then comparing values. When comparing the values of the embedding I was also having trouble as it was difficult to imagine at first that the program was getting an address for the table, then getting an address that was for the item containing the word, and then using address [1] to access the embedding for the word and go through the embedding using another [] address indicator, seeing so many brackets visually confused me and made me question if I was doing this correctly. After using various inputs of similar words I saw it made some sense and was working as intended.

When computing the time it takes to create the table and tree I used the time function and found that the BST would be created on average 18 seconds and the Hash table would be made in about 20 seconds on average, on my laptop. Taking into consideration that the resulting data structures contained 321097 items I figured that the functions were inserting data at a rate of roughly 60 thousand items per second and found the items and would find the items in the comparison process in less than one thousandth of a second consistently leading me to conclude that the comparison process was done in constant time and the insertion process was done in linear time, as each table recreation took more and more time to create as the tables got longer, I also checked the time it took to create the table after resizing. The BST didn’t have to resize at all which is why it took a bit less time on average to create.

The final issue I had was the percentage of empty lists. My initial table would insert based up the final character in the word which is what led to my issue, after staring at the hash code for hours and a hint from the professor I realized it was only looking at the final character and not all the characters and saving based upon the sum of all characters. I reworked this by then adding them all and instead of multiplying r by n – the original code- I changed n to the length of the string which better spread my data across the table to then only having about half of the listings in the table be empty.

The above image is an example of the hash table output and the bellow is an example of the word comparisons hash on the left and bst on the right



Appendix=

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

"""

Course 2302(Data Structures)

Instructor: Olac Fuentes

Teaching Assistant: Anithdita Nath, Mali Zargaran

Lab 5

Last Edited on 4.1.2019

@author: Seth

"""

# Implementation of hash tables with chaining using strings

import numpy as np

import time

import sys

import math

class HashTableC(object):

# Builds a hash table of size 'size'

# Item is a list of (initially empty) lists

# Constructor

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

self.item = []

self.num\_items = num\_items

for i in range(size):

self.item.append([])

def InsertC(H,k,l):

# l must be the following np list of data following k

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

b = Hash(k,len(H.item))#only str in used in hashing process, before hashing check if a non letter char is in the str

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

H.num\_items += 1

def FindC(H,k):

# Returns bucket (b) and index (i)

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

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

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

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

return b, i, H.item[b][i][1]#Returns bucket b, index i, then embedding

return b, -1, -1

def Hash(s,n):

r = 0

#must type cast s to str otherwise errors ocurr......idk why

for c in str(s):

r += (r\*len(str(s)) + ord(c))

return r%n

def ResizeTable(H):

#resizes old table into a new larger table

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

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

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

#H.item[i] is the inital index of where an item is in the list, [j] is for each item in that subsequent list, [0] is to specifically acess the string portion of that list

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

return newTable

def CheckStr(a):

#checks if str a is a valid string to insert

if a.isalpha():

return True

return False

def Array\_to\_Float(array):

#converts array of str into a float array

embed = []

for i in array:

embed.append(float(i))

return embed

def similarityH(H,b0,b1, w0, w1):#H table, B is index of list containing word 0 and 1, w0 w1 are index of the words

DotProduct = []

MagE0 = 0

MagE1 = 0

DP = 0

for i in range(49):#we know that the index is 50 long

DotProduct.append((((H.item[b0])[w0])[1])[i] \* (((H.item[b1])[w1])[1])[i])

for i in range(len(DotProduct)):

DP += DotProduct[i]

# ^^ calculates dotproduct

for i in range(len((((H.item[b0])[w0])[1]))-1):

MagE0 += (((H.item[b0])[w0])[1])[i] \* (((H.item[b0])[w0])[1])[i]

for i in range(len((((H.item[b1])[w1])[1]))-1):

MagE1 += (((H.item[b1])[w1])[1])[i] \* (((H.item[b1])[w1])[1])[i]

MagE0 = math.sqrt(MagE0)

MagE1 = math.sqrt(MagE1)

# ^^ calculates magnitude of E0 and E1

return DP / (MagE0\*MagE1)#formula for similarity

def Compare\_Words(H):

try:

f = open('Pairs.txt')

Lines = f.readlines()

for i in range(len(Lines)):

temp = Lines[i].split(' ')

print(temp)

b0, w0, e0 = FindC(H, str(temp[0]))

if i < len(Lines) -1 :#if else statement does the if except the last line where it does the else

b1, w1, e1 = FindC(H, str(temp[1])[:-1])#[-2] removes the last character in str the \n

else:

b1, w1, e1 = FindC(H, str(temp[1]))

# print(b1, ' : ', w1)

# print(((H.item[b0])[w0])[0], end = ' ')

# print(((H.item[b1])[w1])[0])

print("%.5f" % similarityH(H, b0,b1,w0,w1))

except:

print('word pairs not found')

f.close()

def PercentageEmpty(H):

numEmpty = 0

for i in H.item:

if len(i) == 0:

numEmpty+=1

print(numEmpty)

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

def HashFile(H):

try:

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

Lines = f.readlines()

for i in range(len(Lines)):

temp = Lines[i].split(' ')#splits lines of text into arrays, [0] index will be the str, everything else will be the encoding numbers

embedding = Array\_to\_Float(temp[1:])#converts rest of line into a array

# print(embedding)

# print(len(embedding))

valid = CheckStr(temp[0])

if valid:

InsertC(H, str(temp[0]), embedding)

#if the str if valid then insert str and it's embedding into hash table

if H.num\_items == len(H.item):#LF becomes 1

# print('Resizing from ', len(H.item))

H = ResizeTable(H)

# H = Re\_Hash\_All(H)

f.close()

return H

except:

print("ERROR 404 file is ded")

class BST(object):

# Constructor

def \_\_init\_\_(self, item,numNodes = 0, left=None, right=None):

self.item = item

self.left = left

self.right = right

self.numNodes = numNodes

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 BSTFile():

try:

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

Lines = f.readlines()

for i in range(len(Lines)):

temp = Lines[i].split(' ')

embedding = Array\_To\_Float(temp[1:])

if i == 0:

T = BST([str(temp[0]), embedding])

T.numNodes+=1

if str(temp[0]).isalpha():

Insert(T, [str(temp[0]), embedding])

T.numNodes+=1

f.close()

return T

except:

print('file not found')

sys.exit()

def Array\_To\_Float(array):

#converts array of str into a float array

embed = []

for i in array:

embed.append(float(i))

return embed

def Compare\_WordsBST(T):

try:

f = open('Pairs.txt')

Lines = f.readlines()

for i in range(len(Lines)):

temp = Lines[i].split(' ')

print(temp)

W0 = Find(T, (temp[0]))

if i < len(Lines)-1 :#if else statement does the if except the last line where it does the else

W1 = Find(T, (temp[1])[:-1])#[-1] removes the last character in str the \n

else:

W1 = Find(T, (temp[1]))

# print(W0.item)

# print(W1.item)

print("%.5f" % similarityBST(W0, W1))

except:

print('file not found')

f.close()

def similarityBST(W0, W1):#W0, W1 adresses of W0 and W1 in BST

DotProduct = []

MagE0 = 0

MagE1 = 0

DP = 0

for i in range(49):#we know that the index is 50 long

DotProduct.append((W0.item[1])[i] \* (W1.item[1])[i])

for i in range(len(DotProduct)):

DP += DotProduct[i]

# ^^ calculates dotproduct

for i in range(len(W0.item[1])-1):

MagE0 += math.pow((W0.item[1])[i], 2)

MagE1 += math.pow((W1.item[1])[i], 2)

MagE0 = math.sqrt(MagE0)

MagE1 = math.sqrt(MagE1)

# ^^ calculates magnitude of E0 and E1

return DP / (MagE0\*MagE1)#formula for similarity

def Find(T,k):

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

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

return T

if str(T.item[0])<str(k):

return Find(T.right,k)

return Find(T.left,k)

def FindDepth(T):

if T is None:

return 0

Ltree = 1

Rtree = 1

#initialized to 1 to include their depth

Ltree+=FindDepth(T.left)

Rtree+=FindDepth(T.right)

if Ltree < Rtree:

return 1+Rtree#add root when returning

else:

return 1+Ltree

print('Type 1 for BST or type 2 for a Hash Table with Chaining')

choice = input("Choice: ")

if int(choice) == 1:

print('creating BST')

start = time.time()

T = BSTFile()

end = time.time()

print('Number of nodes: ', T.numNodes)

print('Time needed to create tree:',end-start )

print('Height of tree: ', FindDepth(T))

print('\n\nREADING FILE TO COMPARE WORDS.....\n\n')

# print(T.left.left.item)

# print(T.right.item)

# print(T.left.right.item)

Compare\_WordsBST(T)

#sys.exit('END OF PROGRAM')

if int(choice) == 2:

print('You chose to hash')

H = HashTableC(97)#we start with 97 as it's the largest prime number from 0 -100 and we will need space

start = time.time()

H = HashFile(H)

end = time.time()

LF = H.num\_items / len(H.item)#load factor

#print((H.item[97])[3])

#print((H.item[97])[0])

print('Initial table size: 97')

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

print('Number of items in Table: ', H.num\_items)

print('Load factor: ', LF)

print('Percentage of empty lists: ', PercentageEmpty(H))

print('Time to build table: ', end - start)

print('\n\nReading file to compare words.......\n\n')

start = time.time()

Compare\_Words(H)

end = time.time()

print('Time to find similarities: ', end - start)

# sys.exit('END OF PROGRAM')

“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.”

- Seth Abel Flores