**Lab 5**

This code compares the efficiency of storing words and their embeddings in Hash Tables to Binary Search Trees. The BST code just had to be modified to accept the new objects to properly implement its new function and now includes the method to count the number of nodes in the tree. The HashC class has added and modified methods, like load\_fact() that computes the load factor of the hash table, FindC1() which is a modified version of FindC() that returns only the embedding of the word, percentageH() finds the percentage o empty lists in the Hash Table. Standard(H) calculates the standard deviation of the length of the lists in the hash table, SimilarityT(T) compares the similarity of two words in the tree, similarly SimilarityH(H) uses the hash table to compare the words in the table.

**Experimental Results**

|  |  |  |
| --- | --- | --- |
| Input size | Outputs | |
| 50 | **BST** | **Number of Nodes: 163600**  **Height: 50**  **Tree Construction: 2.81048**  **Query processing:** **0.0097** |
| **Hash Table** | **Init size: 11**  **Final size: 193607**  **Load Factor: 1.5256**  **Percentage: 43.502%**  **Standard Deviation: .9128**  **Hash Construction:2.61305**  **Query processing:0.00695** |
| 100 | **BST** | **Number of Nodes: 163600**  **Height: 50**  **Tree Construction: 1.50498**  **Query processing:**  **0.** **01293** |
| **Hash Table** | **Init size: 11**  **Final size: 193607**  **Load Factor: 1.5256**  **Percentage: 43.502**  **Standard Deviation: 0.9128**  **Hash Construction:2.** **01277**  **Query processing: 0.00798** |
| 200 | **BST** | **Number of Nodes: 163600**  **Height: 50**  **Tree Construction: 1.50498**  **Query processing:**  **0.01895** |
| **Hash Table** | **Init size: 11**  **Final size: 193607**  **Load Factor: 1.5256**  **Percentage: 43.502%**  **Standard Deviation: .918**  **Hash Construction: 2.01277**  **Query processing: 0.01193** |

**Time complexity**

load\_fact() = O(n^2)

FindC1() = O(n)

percentageH() = O(n)

Standard(H) = O(n)

SimilarityT(T) = O(1)

SimilarityH(H) = O(1)

**Conclusion**

Using a Hash Table with a proper hashing function can, in this case reduce the running times for comparing words by at least 1/3 when compared to the binary trees formation, making it a more efficient function for comparing words’ embeddings. Using embeddings to compare words is a comparatively simple process that enables machines to “understand” human language.

**Source Code**

import math

import time

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):

self.item = []

for i in range(size):

self.item.append([])

self.num\_items = num\_items

def InsertC(H,k,l):

# Inserts k in appropriate bucket (list)

# Does nothing if k is already in the table

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

temp = HashTableC((len(H.item)\*2)+ 1,0)

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

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

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

H.item = temp.item

H.num\_items = temp.num\_items;

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

H.num\_items +=1

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

def load\_fact(H):

average = 0

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

if H.item[i] != []:

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

average += len(H.item[i])

return average/len(H.item)

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 FindC1(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 H.item[b][i][1]

return b, -1, -1

def h(s,n): #hashing function

r = 0

for c in s:

r = (r\*27 + ord(c))% n

return r

def percentageH(H):

empty = 0

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

if H.item[i] == []:

empty+=1

percentage = (empty \* 100)/len(H.item)

return percentage

#-----------------------------------------------------------------------------------------------------------

class BST(object):

# Constructor

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

self.item = item

self.left = left

self.right = right

def InsertO(T,newItem):

if T == None:

T = BST(newItem)

elif T.item.word > newItem.word:

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

else:

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

return T

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

def InOrder(T):

# Prints items in BST in ascending order

if T is not None:

InOrder(T.left)

print(T.item,end = ' ')

InOrder(T.right)

def InOrderD(T,space):

# Prints items and structure of BST

if T is not None:

InOrderD(T.right,space+' ')

print(space,T.item)

InOrderD(T.left,space+' ')

def SmallestL(T):

# Returns smallest item in BST. Returns None if T is None

if T is None:

return None

while T.left is not None:

T = T.left

return T

def Smallest(T):

# Returns smallest item in BST. Error if T is None

if T.left is None:

return T

else:

return Smallest(T.left)

def Largest(T):

if T.right is None:

return T

else:

return Largest(T.right)

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 T.item[0] == k:

return T

if T.item[0]<k:

return Find(T.right,k)

return Find(T.left,k)

def FindAndPrint(T,k):

f = Find(T,k)

if f is not None:

print(f.item,'found')

else:

print(k,'not found')

def height(T):

if T is None:

return 0

else:

ldepth= 1 + height(T.left)

rdepth = 1 + height(T.right)

if ldepth < rdepth:

return rdepth

else:

return ldepth

def NumNodes(T):

if T is None:

return 0

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

#----------------------------------------------------------------------------------------------

def Standard(H):

total\_len= 0

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

total\_len += len(H.item[i])

mean = total\_len/len(H.item)

total\_sq = 0

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

x= len(H.item[i]) - mean

total\_sq += (x \* x)

data = total\_sq/len(H.item)

return math.sqrt(data)

def SimilarityT(ob1, ob2):

dot = 0

for i in range(50):

dot += ob1.item[1][i] \* ob2.item[1][i]

mag1 = 0

for i in range(50):

mag1 += ob1.item[1][i] \* ob1.item[1][i]

mag1 = math.sqrt(mag1)

mag2 = 0

for i in range(50):

mag2 += ob2.item[1][i] \* ob2.item[1][i]

mag2 = math.sqrt(mag2)

return dot/(mag1 \* mag2)

def SimilarityH(ob1, ob2):

dot = 0

for i in range(len(ob2)):

dot += ob1[i] \* ob2[i]

mag1 = 0

for i in range(len(ob2)):

mag1 += ob1[i] \* ob1[i]

mag1 = math.sqrt(mag1)

mag2 = 0

for i in range(len(ob2)):

mag2 += ob2[i] \* ob2[i]

mag2 = math.sqrt(mag2)

t = dot/(mag1 \* mag2)

return t

#---------------------------------------------------------------------------------------------

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

arr = []

for line in f:

string = f.readline()

strsplit = string.split()

y = strsplit[0]

x = strsplit[1:]

t = []

for i in range(50):

t.append(float(x[i]))

if y.isalpha():

arr.append([y,t])

choice = input('Choose 1 for Binary Search Tree, 2 for Hash Table:')#, or 3 to exit:')

if choice is '1':

T = None

print('\nOption 1 selected, building binary search tree.')

start = time.time()

for i in range(len(arr)):

T = Insert(T,arr[i])

end = time.time()

print('\nNumber of nodes:', NumNodes(T))

print('\nHeight:', height(T))

print('\nRunning time for binary search tree construction:', round((end-start),5), ' seconds')

print('\nReading word file to determine similarities...')

star = time.time()

t = open('test.txt', encoding='utf-8')

for line in t:

string = t.readline()

strsplit = string.split()

if len(strsplit) == 2:

O1 = Find(T,strsplit[0])

O2 = Find(T,strsplit[1])

if O1 is not None and O2 is not None:

print('Similarity ', strsplit,' = ',SimilarityT(O1, O2))

en = time.time()

print('\nRunning time for hash table query processing:', round((en-star),5), ' seconds')

#--------------------------------------------------------------------------------------------

if choice is '2':

initSize = 11

H = HashTableC(initSize,0)

print('\nOption 2 selected, building hash table with chaining.')

start = time.time()

for i in range(len(arr)):

InsertC(H,arr[i][0], arr[i][1])

end = time.time()

print('\nHash table stats:')

print('\nInitial table size:', initSize)

print('\nFinal Table size:', len(H.item))

print('\nLoad Factor:', round(load\_fact(H),4))

print('\nPercentage of empty lists:', round(percentageH(H),4), '%')

print('\nStandard deviation of the length of lists:', round(Standard(H),4))

print('\nRunning time for hash table construction:', round((end-start),5), ' seconds')

print('\nReading word file to determine similarities...')

start = time.time()

t = open('test.txt', 'r')

for line in t:

string = t.readline()

strsplit = string.split()

if len(strsplit) == 2:

O1 = FindC1(H,strsplit[0])

O2 = FindC1(H,strsplit[1])

if O1 is not None and O2 is not None and len(O2) == 50 and len(O1) == 50:

print('Similarity ', strsplit,' = ',SimilarityH(O1, O2))

end = time.time()

print('\nRunning time for hash table query processing:', round((end-start),5), ' seconds')

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.

\_\_\_\_\_\_\_\_\_\_\_\_Hugo Chavez\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_