Lab 5 Report

Nicole Favela

CS2302

April 3, 2019

**Introduction**

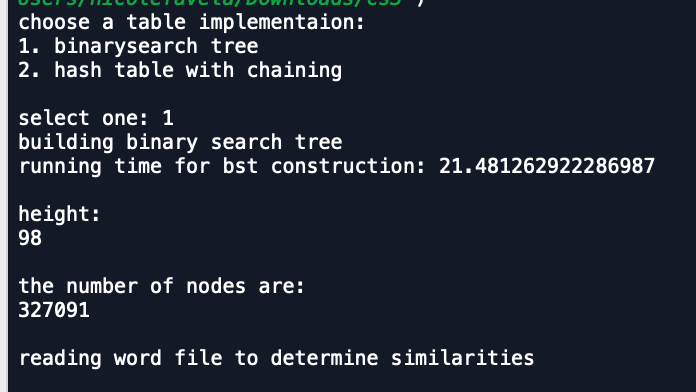
The problem I am trying to solve is to compare create 2 different tables to store a file containing an embedding and words provided by Stanford. Given this file and corresponding equation of similarities, I was tasked with looking up words and determining if they are similar. The first table implementation is a hash table with chaining. The second table implementation is a BST with nodes containing the word and corresponding embedding. This program compares an embedding of words found in the file and returns a floating-point value. Very similar words will produce a value of 1 and smaller similarity value means words are not very closely related at all. This program gives the user a choice of selecting 1 for building a BST and 2 for selecting a hash table with chaining for searching purposes.

# Proposed solution and implementation:

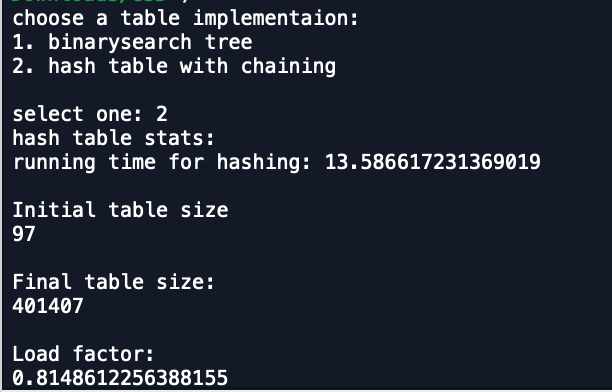
My approach to choice 1, BST, was to store the word part of the file at position 0 in ever line in a list of size 1. I stored the embedding in another list so I could quickly compute the dot product for use in the similarities equation. This was given by the magnitudes of corresponding vectors. I computed the magnitudes of the vectors by squaring corresponding elements, adding them and taking the square root. Since my BST nodes can store the node objects already, I stored those lists depending on their dictionary order by comparing the strings. For my find method in the BST implementation, I modified it to compare the strings instead of numbers. The number of nodes in the tree and the height were already completed in lab 3 which was utilized in this lab as well. For every implementation, I used a timer to record the time taken to build the BST and hash tables. From there, I could look up the nodes in log(n) time to obtain the embedding for the pairs of words. My approach to the hash table was to do a similar operation to building the BST in terms of reading the file. The difference is the way in which I stored the string values. The given function ﻿h(s,n) takes a string s and the length of the list n to choose a bucket to hash the values into. This function determines the bucket to hash into based on the ASCII value. When the load factor of the hash table exceeds 1 (computed by the ﻿number of items/len(H.item)) I used ﻿doubleSize(H) to increase the size of the table. My function doubleSize(H) multiples the hashtable size by 2 and adds 1 and rehashes the values into new buckets. My initial table size was 97. My percentage of empty lists ﻿percentageOfEmptyLists(H) adds to counter whenever a list of size 1 is encountered and then divides it by the length of H. I multiplied the result by 100 to show the percentage. The computation of similarities was computed by the dot product divided by the magnitudes. Had I finished the standard deviation, it would have been measure of how spread out the lengths of the lists in the hash tables are.

**Experimental results:**

I used the given file of ﻿glove.6B.50d.txt to produce the following output when option 1 is selected:



If the find operation would have been implemented for similarities of pairs of words, it would have operated in O(nlogn) time. If the hashtable implementation of find would have been performed on the pairs of words w0 and w1 it would have found the words in O(1) and in the worst case O(n) where n is the length of the list associated with the bucket. The sample output for the hash table implementation (option 2) is as shown:





The results of the time were taken by starting time.time() and then stopping after the method calls in both implementation.

For testing purposes, I used a file I copied with length 4 (from given file) called “﻿lab5testfile.txt.”

testfile:

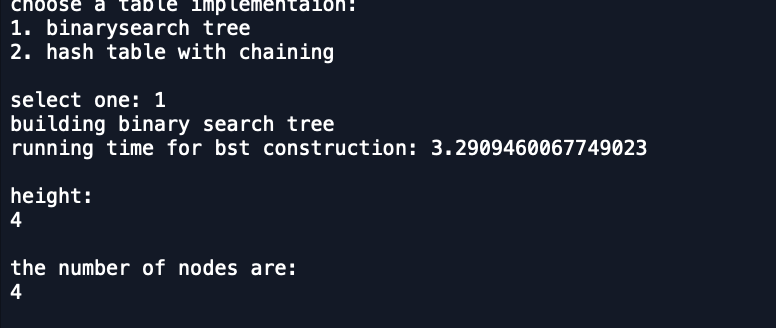
now 0.40991 0.081369 0.29962 -0.2535 0.30073 -0.23569 -0.89244 -0.232 -0.24374 -0.14191 0.026541 -0.093562 -0.51718 -0.36123 0.60513 0.66266 0.30449 0.44857 -0.092027 -0.069213 -0.1058 0.10899 0.1033 0.34372 -0.04186 -1.9213 -0.33361 0.16119 0.084769 -0.28993 3.5443 0.26021 -0.023478 -0.39401 0.036919 -0.32256 -0.15009 0.48977 0.16311 -0.44252 -0.44062 -0.043155 0.23357 0.2823 -0.4349 0.40295 -0.47853 -0.332 -0.3991 0.13063

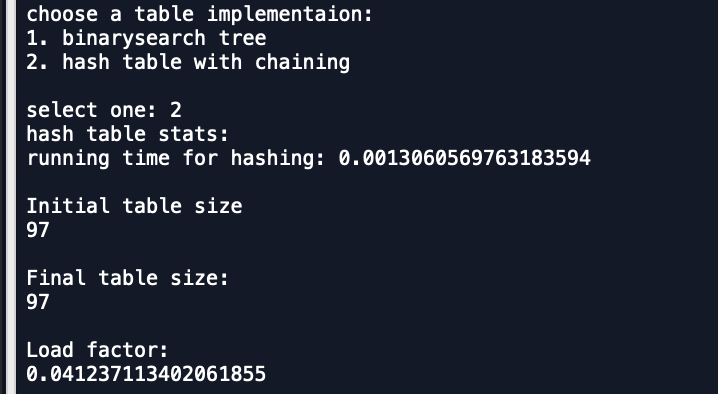
city 0.43945 0.43268 -0.36654 0.27781 0.062932 -0.80201 -0.93041 0.016388 -0.5503 -0.16278 -0.40351 -1.3975 0.32079 -0.88954 -0.18853 0.11516 0.045326 0.83002 -0.87595 0.77653 0.55948 0.074721 -0.84672 0.4098 -0.59774 -2.062 -0.15894 0.57979 0.2827 -1.0213 3.2488 0.50029 0.11559 -1.1707 0.19016 0.36888 -0.042018 0.028234 0.54125 0.8489 -0.66711 0.60799 0.23787 -0.65378 -0.7055 0.5165 -1.078 -0.71524 0.48402 -0.3256

made 0.14205 0.0046063 -0.40052 0.076568 0.27081 0.61955 -0.84904 -0.30413 -0.58594 0.092473 -0.034618 0.45746 -0.42953 -0.044566 0.38408 -0.096784 -0.048204 -0.26517 -0.50191 -0.51384 0.19384 0.15066 0.43169 -0.86243 0.059659 -1.5264 -0.11795 0.1915 0.10062 -0.49343 3.388 0.047361 -0.2738 0.03688 0.43812 0.28114 0.40339 0.87699 -0.40557 -0.15664 0.085523 0.36692 -0.081265 -0.32905 -0.069627 0.10247 -0.61574 0.27769 0.033304 -0.24918

like 0.36808 0.20834 -0.22319 0.046283 0.20098 0.27515 -0.77127 -0.76804 -0.34861 0.5062 -0.24401 0.71775 -0.33348 0.37554 0.44756 0.36698 0.43533 0.4757 -0.056113 -0.93531 -0.27591 0.3161 0.22116 0.36304 0.10757 -1.7638 -1.2624 0.30284 0.56286 -1.0214 3.2353 0.48483 0.027953 0.036082 -0.078554 0.18761 -0.52573 0.0372 0.27579 -0.07736 -0.27955 0.79752 0.0016028 0.45479 0.88382 0.43893 -0.19263 -0.67236 -0.39709 0.25183

That produced the following results:







I was unable to finish the comparisons and embedding part of this due to time constraints and errors when reading the file. The size of the file also made things difficult and often caused my computer to crash.

**Conclusion:**

I learned more about how NLP (natural language programming works) to compare how words. It is an interesting sub-field in artificial intelligence. I learned how to utilize 2 different table implementations to store a list of items and their embedding or vectors. I can see how this would be useful in real life since large amounts of data or objects can be quickly positioned and retrieved in worse case O(n) for hash tables with chaining. The math behind how these word similarities are computed is very interesting. The embedding of the words can create vector points from which the magnitudes and cosine angles can be computed. Both of these implementations provide a quick lookup of items or objects that can be useful in many applications.

**Appendix (source code):**

﻿

﻿

##CS2302

#Nicole Favela

#last modified: April 3, 2019

#Lab5

#purpose: to compare runtimes of 2 implementaions of tables and compute similarities of embeddings

#and to practice working with hash tables and binary search trees

#instructor: Olac Fuentes

#TAs: Anindita Nath and Maliheh Zargaran

#import numpy as np

import math

import time

#BST constructor

class BST(object):

# Constructor

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

self.item = item

self.left = left

self.right = right

#inserts new items to bst

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

#prints structure of bst

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+' ')

#BST

def bstImplementation():

try:

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

line=f.readline()

T=None

while line:

lists=[]

lists=line.split(" ")

if lists[0].isalpha():

embedding=((lists[1:]))

# embedding=[float(i) for i in embedding]

T=Insert(T,(lists[0],embedding))

line=f.readline()

return T

except:

print("File not found")

# 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([])

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])

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

#hashes based on ascii

def h(s,n):

r = 0

for c in s:

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

return r

#counts lines in file to test

def countlines():

count = 0

filename = "glove.6B.50d.txt"

with open(filename,encoding='utf-8') as f:

for line in f:

count+=1

return count

#gets dot product for e0 and e1

def dotProduct(emb):

mag = vectorMagnitudes(emb,emb)

total = 0

for i in range (len(emb)):

total+= (mag)\*(mag)

return total

#computes the magnitudes of the items

def vectorMagnitudes(e0,e1):

sum = 0

mag = 0

for i in e0:

sum += ((e0[i])\*\*2) + ((e1[i])\*\*2)

mag = math.sqrt(sum)

return mag

#get similarities

def computeSimilaritiesForHash(w0,w1):

#FindC(H,k)

sim = 0

w0 = Find(H,w0)

w1 = Find(H,w1)

sim = dotProduct(w0,w1)/(vectorMagnitudes(w0,w1))

return sim

#gets similities for bst

def computeSimilaritiesForBST(w0,w1):

sim = 0

w0 = Find(T,w0) #finds w0 in tree

w1 = Find(T,w1)

sim = dotProduct(w0,w1)/(vectorMagnitudes(w0,w1)) #computes the dot product

return sim

#reads file of pairs for comparison and compares similarities

def CompareWithBST(T):

try:

with open("similarities.txt",encoding='utf-8') as f:

line= f.readline()

while line:

lists=[]

lists=line.split(" ")

#finds words in tree

w0 = Find(T, lists[0])

w1 =Find(T,lists[1])

# sim = computeSimilaritiesForBST(w0,w1)

# print('similarites are: ',sim)

line=f.readline()

except:

print('file not found')

finally:

f.close()

#doubles size of hash table

def doubleSize(H):

H2=HashTableC(2\*len(H.item)+1)

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

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

#rehashes values

InsertC(H2,((H.item[i])[j])[0], ((H.item[i])[j])[1])

return H2

#computes load factor for hash

def loadfactor(H):

if H is None:

return -1

counter = 0

for i in range(len(H.item)): #goes through len of hash and adds to counter

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

return counter/len(H.item)

#counts all nodes in bst

def countAllNodes(T):

if T is None:

return 0

else: #count nodes of L and nodes of R

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

#gets height of bst

def DepthCounter(T):

if T is None:

return 0

CountL=1

CountR=1

CountL+=DepthCounter(T.left) #gets count of depth in L tree

CountR+=DepthCounter(T.right) #gets count of depth in R tree

if CountL>CountR:

return 1+CountL #returns larger count

return 1+CountR

#find for bst

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

#compares order to determine location

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

#checks right

return Find(T.right,k)

return Find(T.left,k)

#gets percentage empty of hash

def percentageOfEmptyLists(H):

empty=0

for i in H.item:

if len(i)==0: # if empty add to counter

empty+=1

empty=empty/2 #list double size needed

return (empty/len(H.item))\*100 # multiply by 100 to get percentage

#hash table implementation

def HashImplementation():

try:

# reads file

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

line= f.readline()

word=[]

embedding=[]

while line:

lists=[]

lists=line.split(" ")

if lists[0].isalpha():

#stores words

word.append(lists[0])

#stores embeddings

embedding = lists[1:]

#inserts word followed by embedding in hash

InsertC(H,lists[0],embedding)

line=f.readline()

return H

except IOError:

print('file not found')

finally:

f.close()

##menu

print('choose a table implementation:')

#prompt user

print('1. binarysearch tree ')

print('2. hash table with chaining ')

#get choice input

choice = int(input('select one: '))

if choice == 1:

print('building binary search tree')

start = time.time()

T = bstImplementation()

end= time.time()

print('running time for bst construction:',end - start)

print()

print('height:')

print(DepthCounter(T))

print()

print('the number of nodes are:')

print(countAllNodes(T))

print()

print('reading word file to determine similarities')

# print(CompareWithBST(T))

#reads file

if choice == 2:

H = HashTableC(97)

print('hash table stats:')

start = time.time()

H=HashImplementation()

end= time.time()

print('running time for hashing:',end - start)

print()

print('Initial table size')

print(len(H.item))

print()

print('Final table size:')

while loadfactor(H)>=1:

H=doubleSize(H)

print(len(H.item))

print()

print('Load factor:')

print(loadfactor(H))

print()

print('percentage of empty lists:',percentageOfEmptyLists(H))

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.”

-Nicole Favela