Course: CS 2302

Author: Kimberly Morales

Assignment: Lab 5

Instructor: Olac Fuentes

TA(s): Anindita Nath, Maliheh Zargaran

Date: 4/2/2019

# Introduction

To create a basic foundation of a natural language processing(nlp) program that finds the similarity between two words. This is done by calculating the cosine similarity by having the dot product of the embeddings divided by the magnitude of the two embeddings. Then, to either build a binary search tree(bst) or hash table based on the user's choice. After the data structure is built, stats are shown along with runtimes of building it and querying for the words within it. To run the program with multiple words, another word doc called words.txt is used to contains pairs.

# Proposed Solution Design and Implementation

I ended up trying to find the methods of the binary search tree that I did for lab 3 and repurposed it for this lab. The hash table implementation was like the class webpage’s version except for slight differences which is described more in the hash table section. Is spent majority of my time trying to understand the cosine similarity and reading documentation of several python modules to see if I could lessen the amount of work done for computation and instead focus on the data structures. My overall approach to the hash table was to insert into the hash table by looping through the 2d list and appending the word and embedding. For the bst, I wanted to make it balanced but had issues inserting the 2d array. This ended up just being an issue of just inputting the wrong index. I reused my sortlist to bst method from lab 3.

**Hash Table (h)**

I reused the hash table chaining class that uses the values of a string instead of integers to put the elements within the table. It originally came with the size and empty list, but another parameter called num items is used to keep track of new items added so that the load factor is not always calculated. This method came with the find and insert methods, so I just used those without much modification. I did change the hash function slightly so that n is not always multiplied to the hash. For the initial table size, I used 11 since it is prime and is not too small to use. When querying the h, I used the find function to get the value and used c\_sim to get the similarity.

Resize\_h

This method takes in only the hash table and appends an empty list to the table. This is done with a for loop that iterates twice the length of h and adds 1. H is returned at the end.

Loadfactor

The load factor of a hash table with chaining is the number of elements divided over the length of the h. So, to count the number of elements, a loop is used to count only non-empty lists within each slot. Afterwards, the load factor method divides this count over the length of the hash table.

Per\_emp\_l

The percentage of empty lists is just how much empty lists are in the table. Like load factor, a count is used within the loop except it only counts the empty list. Then the method returns the count over the number of slots. This number is also multiplied by 100 since we need a percentage.

Std\_dev

I’m not too familiar to standard deviation so I ended up using the sample formula. To start off, I use a new list to acquire the number of lists in each slot and append this to a. Then I used the python statistics module’s function of standard deviation which utilizes sample. I used the sample version because I’m not sure about the differences between this version and population and sample focuses on a smaller range.

**Binary Search Tree (bst)**

I reused my version of the bst from lab3 and kept the insert and find methods of the class. Before adding any elements into the tree, the initial list of the file is first sorted with the python default sort method. Sort can take in a key to specify what guidelines to sort and I used the python operator module. Operator is a module designed to expand on the logic and abstract math’s of python. I used itemgetter from operator so that I only sort based on the first index of each column in the 2d list. This sorted list is then used with slist\_to\_bst. I did not used the typical way of insertion mostly because I wanted the runtime to be efficient and to create a balanced tree.

Height

The height is calculated by finding the max height of the right and left subtrees and adding 1. This is done by continuously recursing. This method is reused from lab 3.

Num\_nodes

To count the number of nodes, recursion is used to count each traversal in the left and right subtrees. While like height, we are looking for how many times we hit a node rather than which subtree has the bigger height.

Slist\_to\_bst

Reused from lab 3 and is used to create a balanced subtree so that the words do not make the height unnecessarily big. It’s also used this way since it is in alphabetical order. This method takes in a sorted list as input and uses the mid elements in the list to make the root. Then the elements from the beginning until the mid-element is added to the left subtree. Similarity, elements from the middle to the end of the list are added to the right subtree.

**Utilities**

C\_sim

To calculate the cosine similarity, I need to have the magnitudes of the embeddings and the dot product of the embeddings. When looking into the numpy documentation, I found a convenient method to use the dot product that can take in a list as a parameter. Since I saw this method, I went to see if there was something similar with finding the magnitude. I ended up finding another method called linalg.norm that also can take in a list to find the magnitude of a vector. I did used these functions in the end since it did compute correct results compared to my results that I did by hand.

File\_reader

File reader takes in two parameters: the filename and an option. The option parameter is used to make the file reader reusable since I needed to make slight changes on how a file is read with glove and the text file of the words. Option g reads for the glove file and t is used for the words text file. Instead of a for loop, a with loop is used since python needs to open and close the file and glove has a specific encoding so encoding with utf8 is used. When option is g, the line of each file is split into a string list and if the first element (the word) is not punctuation then the first element and list(embedding) is put into a list and appended to a new list. For option t, the word file is read and is split to a string list. The first element (the first word) and second element (second word) is put into a list and appended to a new list. This new list is returned at the end once the file is closed.

**Main I/O**

This main function deals with the input and the overall menu display. Not all displays are shown here since each data structure has their specified stats to show so debug\_bst and debug\_h deals with the stats. This was done mostly for debug purposes and to not clutter the main area. Main contains the filenames and reads in the file before any menu choice. Then the menu is printed, and the user can pick 1 or 2, anything else causes an error. Choice 1, the binary search tree, sorts the list and puts the parameters of the two words to be compared and the sorted list into debug\_bst. Choice 2, the hash table, just inputs the two words to be compared into the hash table. Each of these choices has the words text file read in order to get the two words to be compared.

# Experimental Results

Binary Search Tree

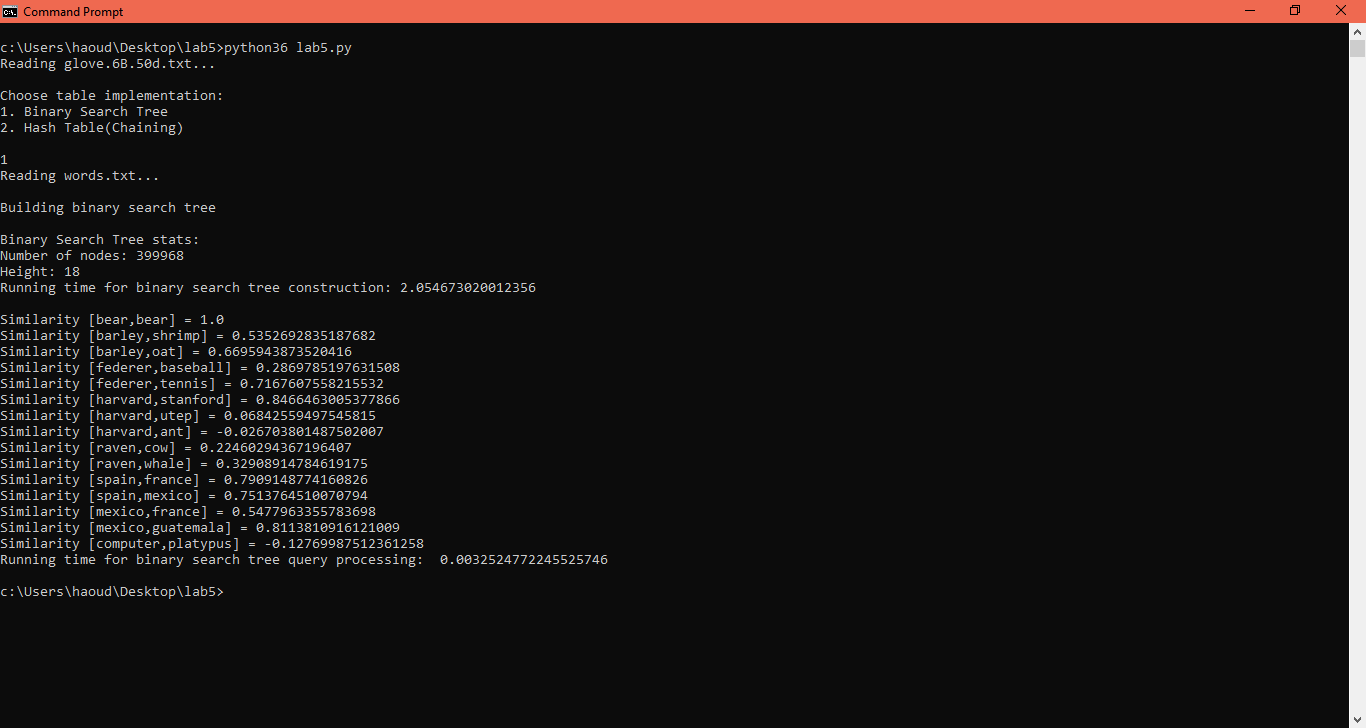
|  |  |  |
| --- | --- | --- |
| Input | Build Run Time | Query Run Time |
| 15 | 2.05 | .0033 |
| 25 | 2.02 | .0050 |
| 35 | 2.21 | .0104 |
| 45 | 2.05 | .0286 |
| 55 | 2.04 | .0287 |
| 65 | 2.20 | .0440 |

Hash Table

|  |  |
| --- | --- |
| Input | Query Run Time |
| 15 | .0028 |
| 25 | .0039 |
| 35 | .0297 |
| 45 | .0146 |
| 55 | .0328 |
| 65 | .0382 |

BST Screenshots

N = 15



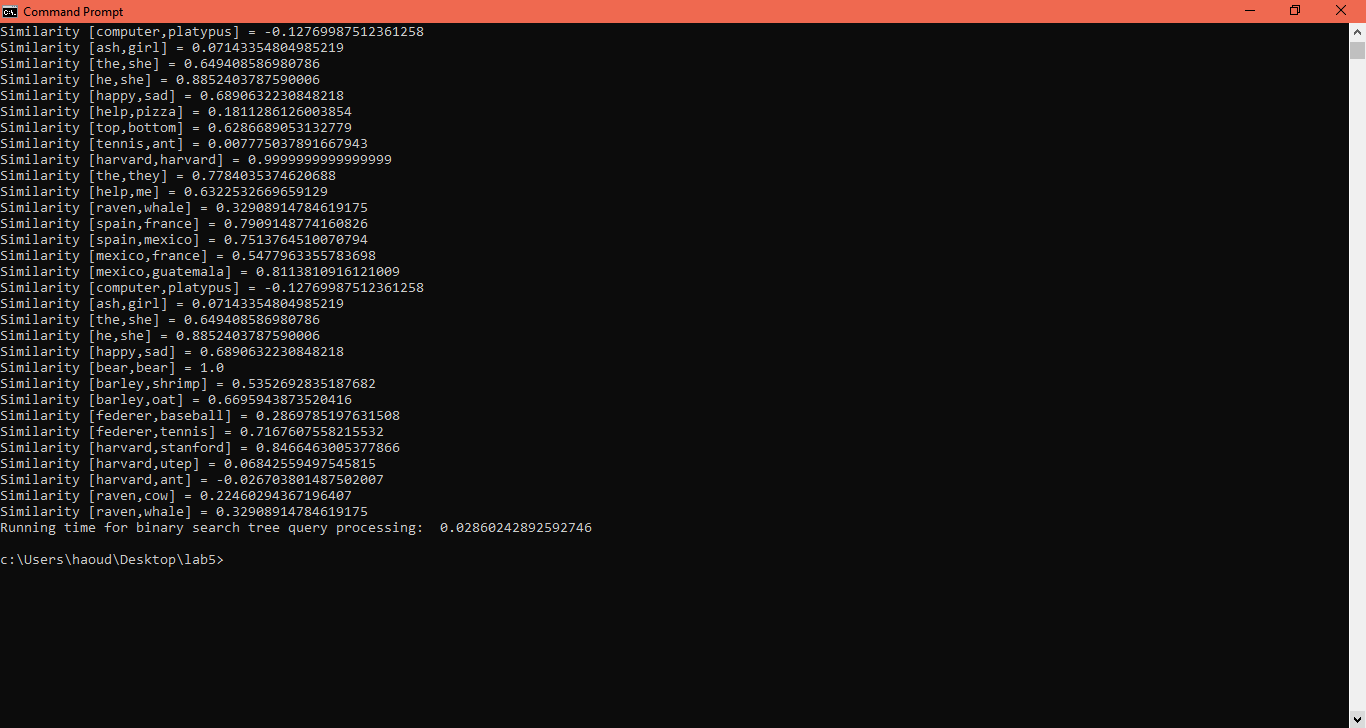
N = 25

# 

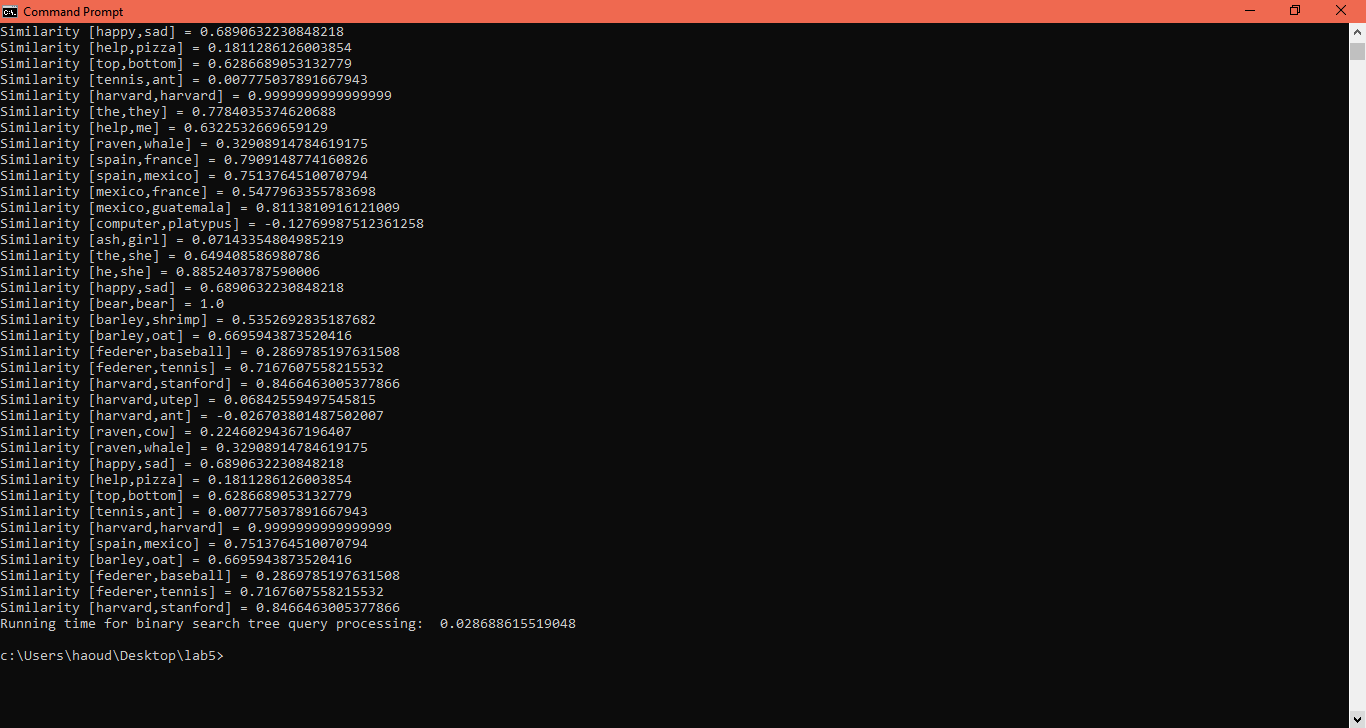
N = 35

# 

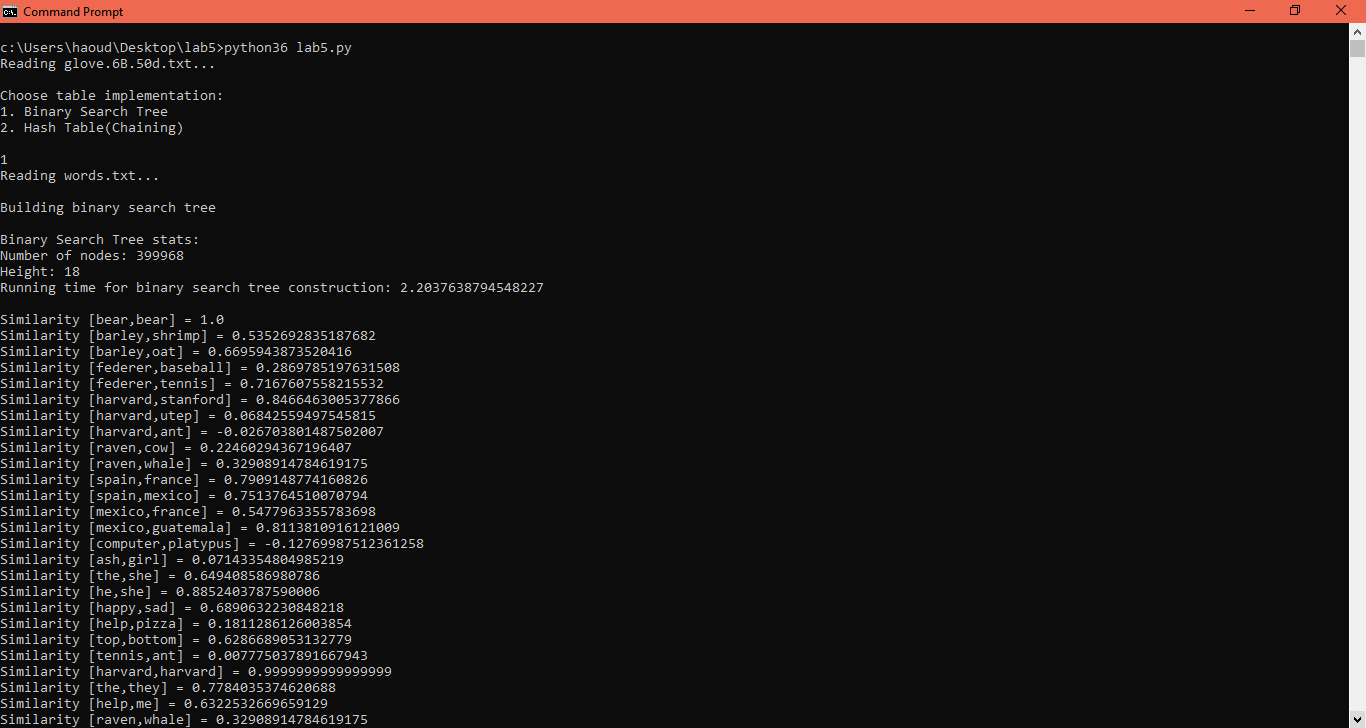
N = 45



N = 55

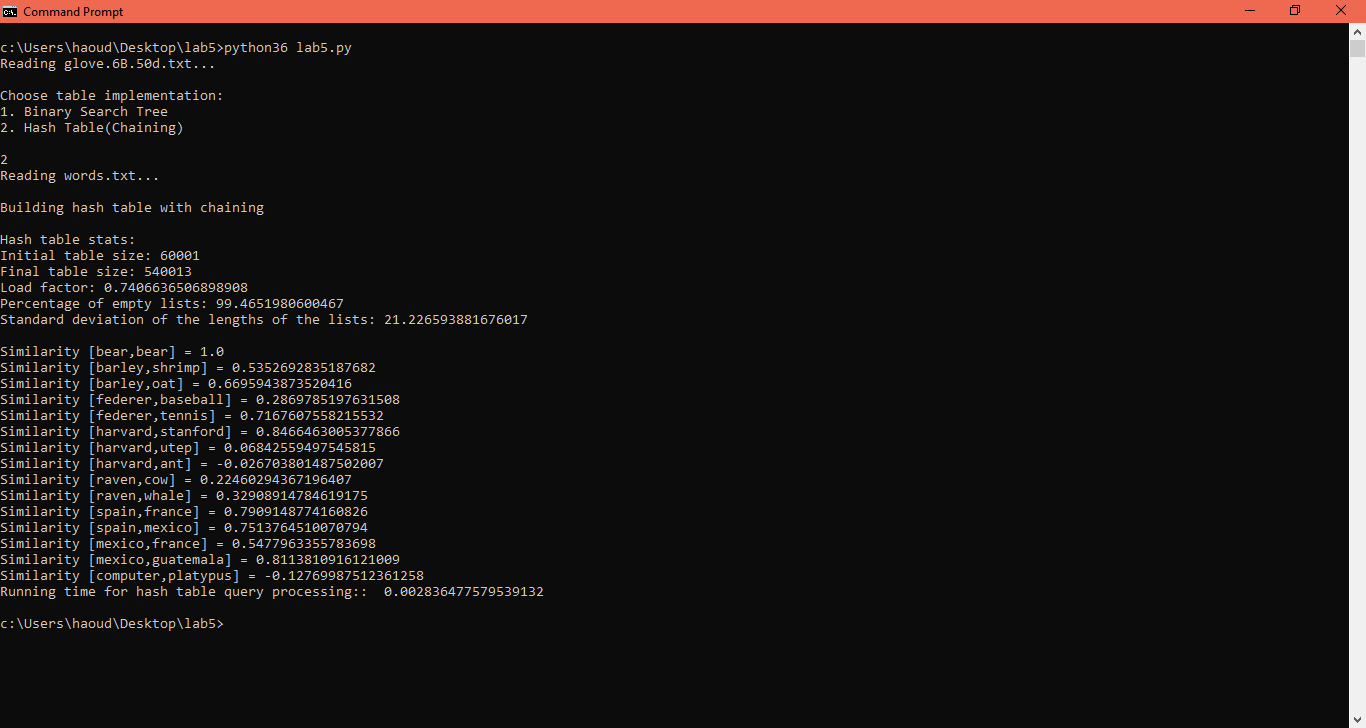


N = 65

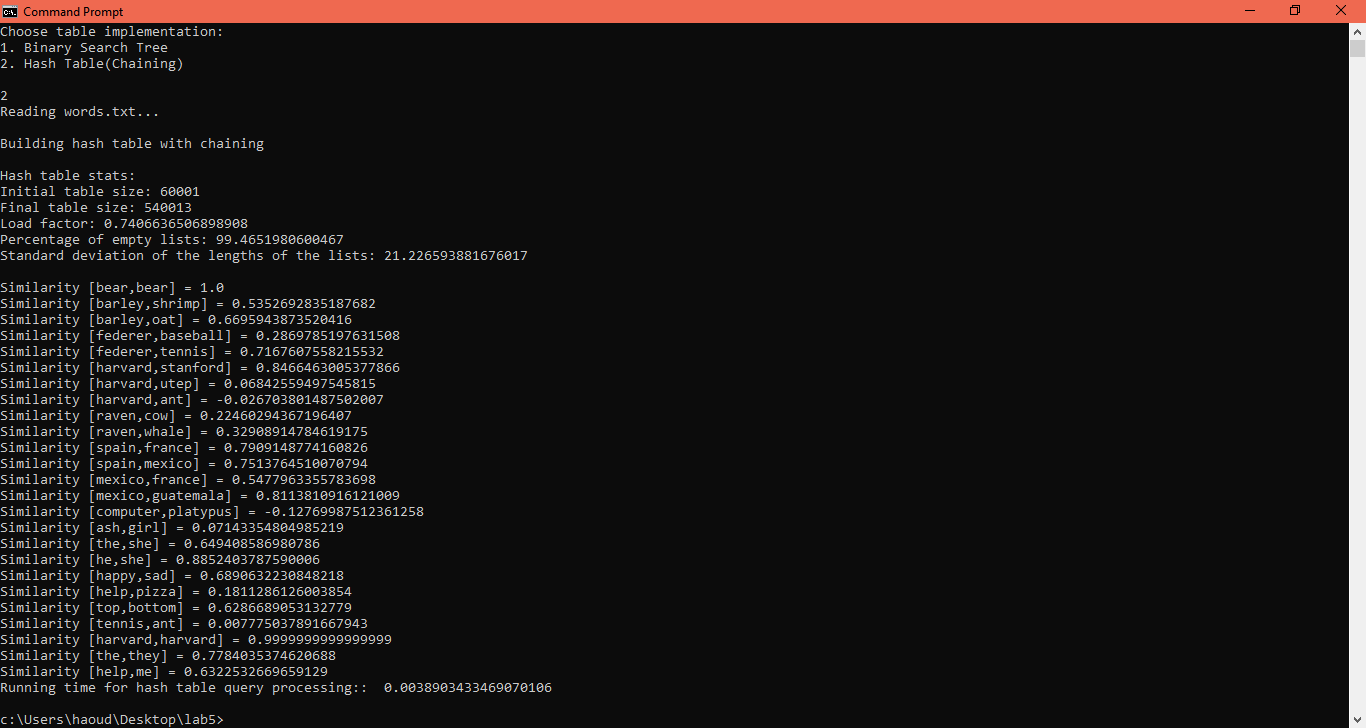


H Screenshots

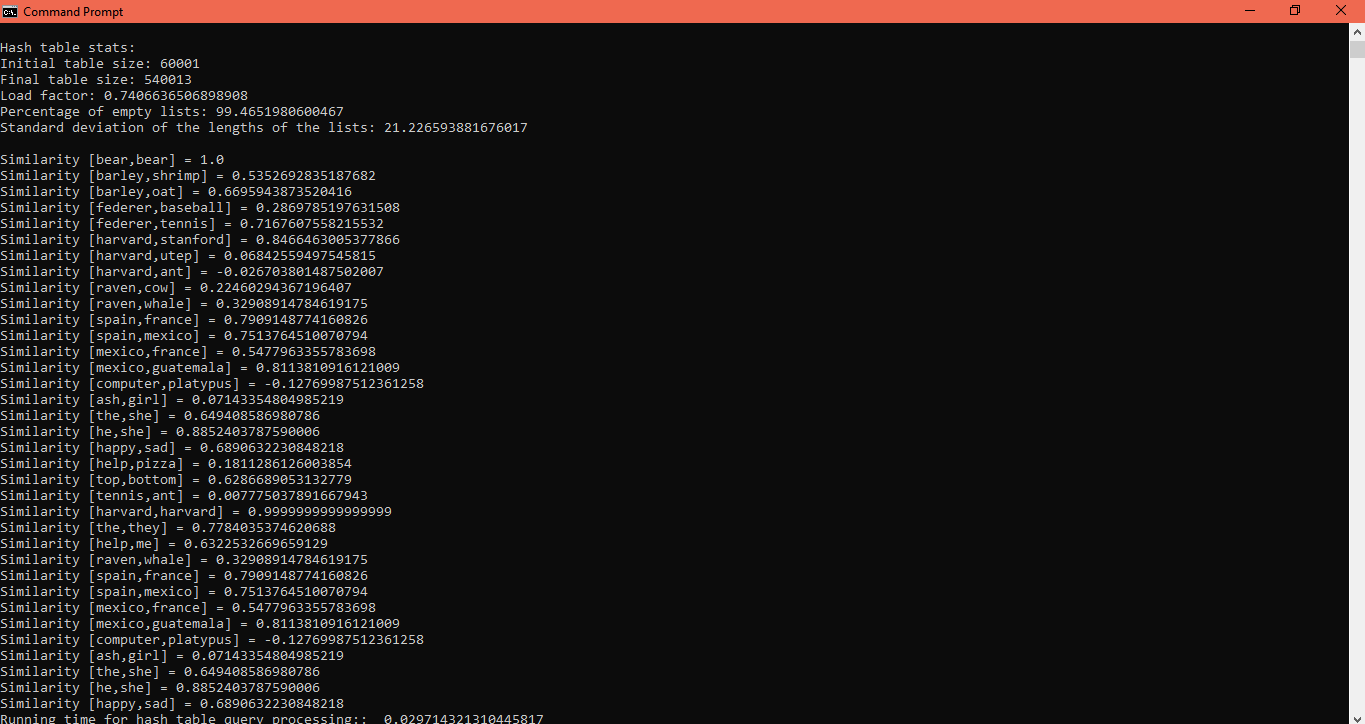
N =15



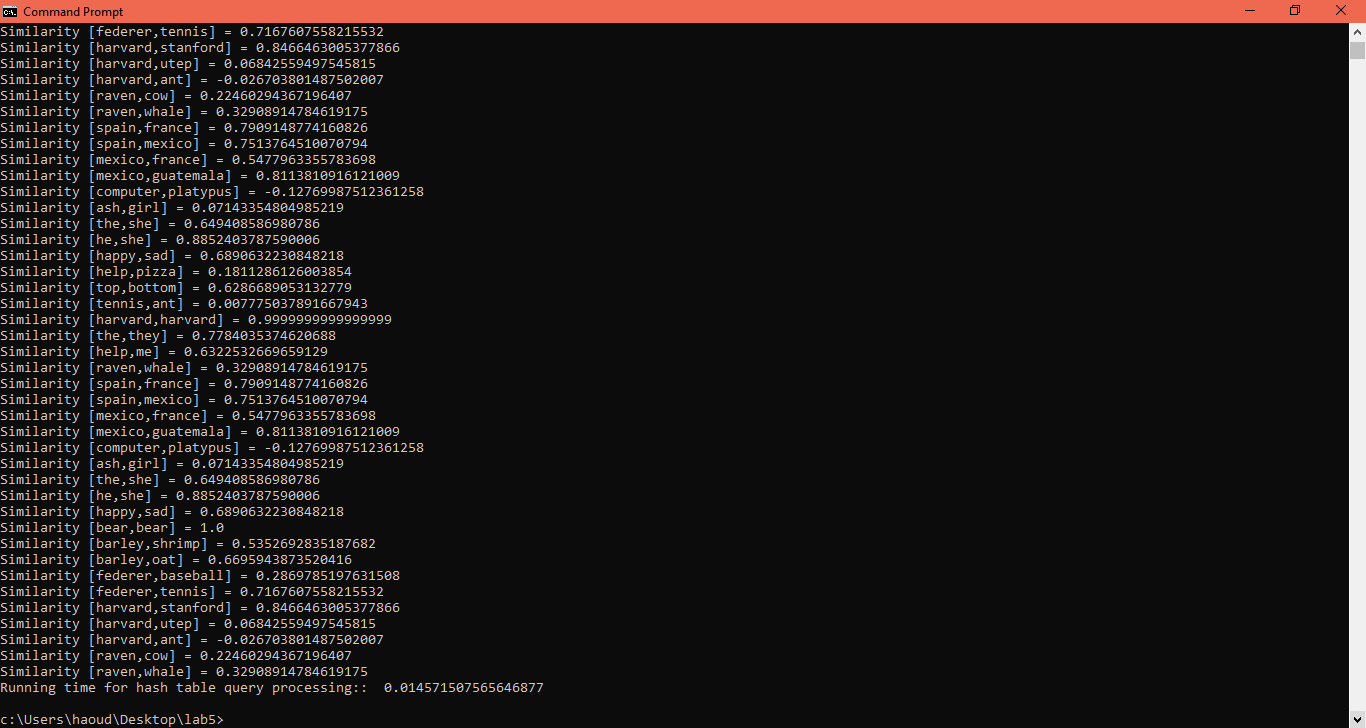
N = 25



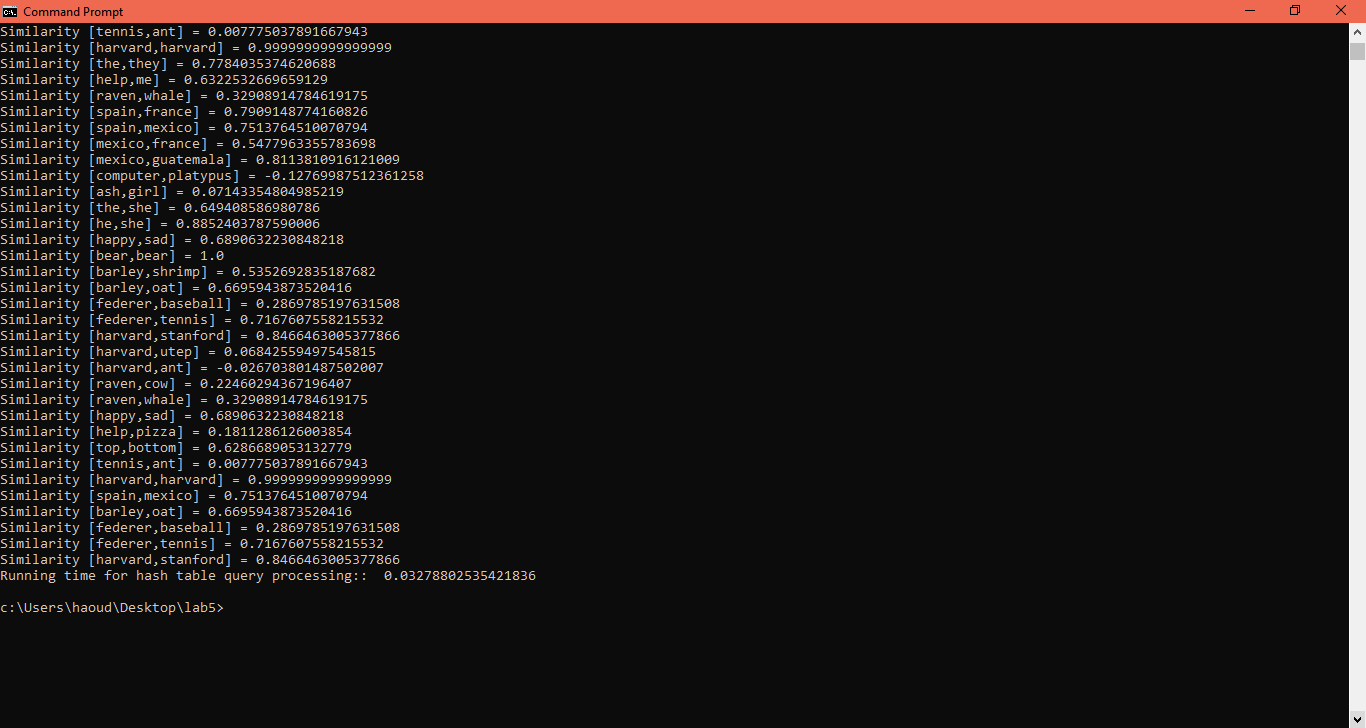
N = 35



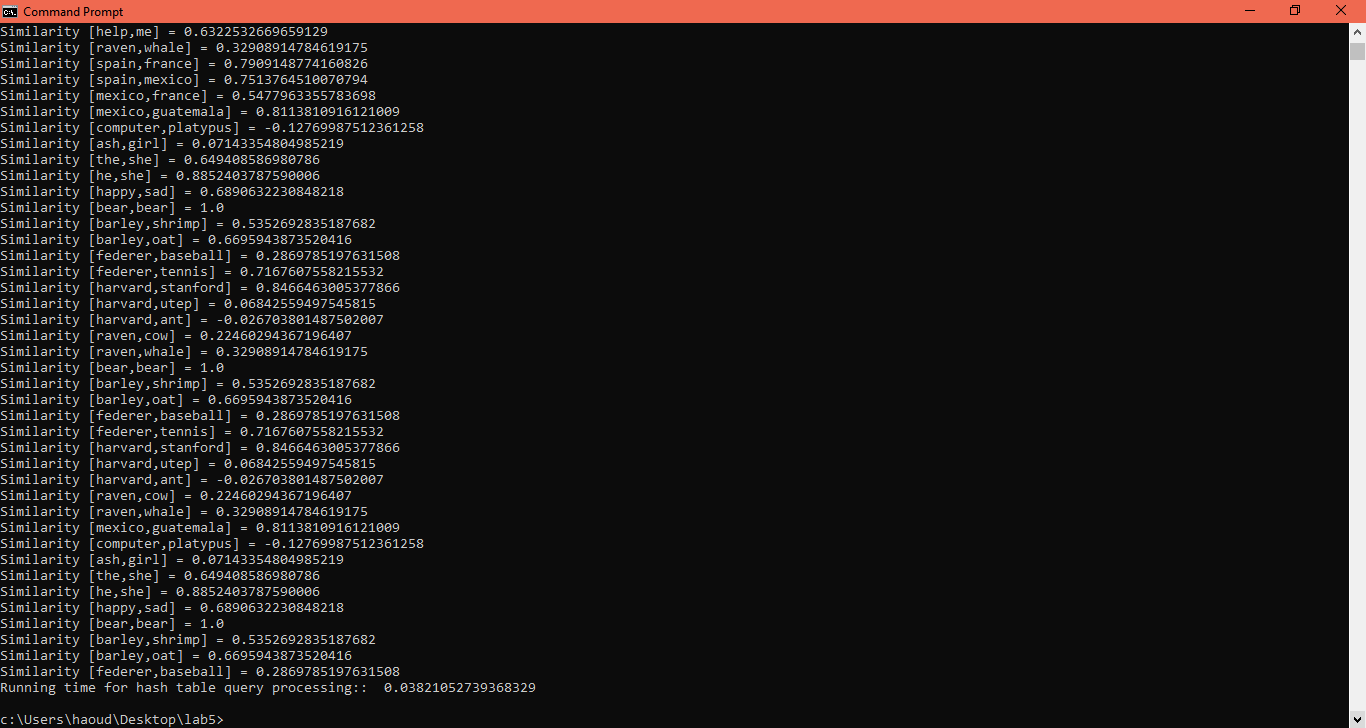
N = 45



N = 55



N = 65



# Conclusions

Overall, I learned a lot about nlp more than I intended since the lab initially looked threatening to me. I ended up learning a lot about python documentation for this project and learned the intricacies of the python native list. Reading a file with python is also interesting since Java requires a lot of syntax to read a text file that often needs exceptions and to close the file. In python, I could just use with and not worry about closing the file.

# Appendix

"""

Course: CS 2302 [MW 1:30-2:50]

Author: Kimberly Morales

Assignment: Lab 5

Instructor: Olac Fuentes

TA(s): Anindita Nath , Maliheh Zargaran

Date: 4/1/2019

Date of last modification: 4/1/2019

Purpose of program:

To create a basic foundation of a natural language processing(nlp) program that finds the similarity between two words.

This is done by calculating the cosine simularity by having the dot product of the embeddings divided by the magnitude of the two embeddings.

Then, to either build a binary search tree(bst) or hashtable based on the user's choice.

After the data structure is built, stats are shown along with runtimes of building it and querying for the words within it.

In order to run the program with multiple words, another word doc called words.txt is used to contains pairs.

"""

import numpy as np

import matplotlib.pyplot as plt

import math

import random

import timeit

from operator import itemgetter

import statistics

import string

#################################################################################################

#HASHTABLE (CHAINING) CLASS

#################################################################################################

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 #num\_items: Counts new item so to not recalculate load factor

for i in range(size):

self.item.append([])

#resize\_h: Resizes the table if the loadfactor is 1 or over

#Increases table by making it twice as big plus 1

def resize\_h(H):

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

#Traverse through list and add to new list

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

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

InsertC(H2,((H.item[i])[j])[0], ((H.item[i])[j])[1]) #Insert items from old list to new list with new hash key

return H2

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

return r

def loadfactor(H):

count = 0

for i in range(len(H.item)): #If the list is not empty then add all elements in the count

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

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

return count / len(H.item) #Return num of elements over length of list since it is chaining

def per\_emp\_l(H):

count = 0

#If the list is empty then add one to count

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

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

count += 1

#Return num of empty lists over table size times 100 to get percentage

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

def std\_dev(H):

a = []

#Get length of each slot in the table and append to list s

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

a.append(len(H.item[i]))

#Python statistics method (sample) to get stddev

return statistics.stdev(a)

def debug\_h(A, w):

orig\_size = 60001

H = HashTableC(orig\_size)

print("Building hash table with chaining \n")

for i in range(len(A)):

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

#print(H.item)

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

H=resize\_h(H)

print("Hash table stats: \n"

+ "Initial table size: " + str(orig\_size) + "\n"

+ "Final table size: " + str(len(H.item)) + "\n"

+ "Load factor: " + str(loadfactor(H)) + "\n"

+ "Percentage of empty lists: " + str(per\_emp\_l(H)) + "\n"

+ "Standard deviation of the lengths of the lists: " + str(std\_dev(H)) + "\n"

)

#Processes words.txt to run many similarities

start = timeit.default\_timer()

for i in range(len(w)):

e0 = FindC(H, w[i][0])[2]

e1 = FindC(H, w[i][1])[2]

#print(FindC(H, w[i][0]))

#print(FindC(H, w[i][1]))

if e0 == -1 or e1 == -1:

print("No word found")

else:

print("Similarity [" + str(w[i][0]) + "," + str(w[i][1]) + "] = " + str(c\_sim(e0,e1)))

stop = timeit.default\_timer()

print('Running time for hash table query processing:: ', stop - start)

#################################################################################################

#BINARY SEARCH TREE CLASS

#################################################################################################

class BST(object):

# Constructor

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

self.item = item

self.left = left

self.right = right

def Insert(T,newItem):

if T == None:

T = BST(newItem)

elif T.item > newItem:

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

else:

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

return T

def Find(T,k):

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

while T is not None:

if T.item[0] == k:

return T.item

elif T.item[0] < k:

T = T.right

else:

T = T.left

return -1

def height(T):

if T is None:

return -1

return max(height(T.left) , height(T.right)) + 1

def num\_nodes(T):

if T is None:

return 0

else:

return num\_nodes(T.left) + num\_nodes(T.right) + 1

def slist\_to\_bst(L):

if not L:

return None

#If there is only 1 element then the bst is already balanced

if len(L) == 1:

T = BST(L[0])

return T

mid = (len(L)) // 2 #mid: Middle element/median of sorted list

T = BST(L[mid]) #Make mid the root

#From start until middle of the list, add the middle element for the left subtree

T.left = slist\_to\_bst(L[:mid])

#From mid plus until end of the list, add the middle element for the righ subtree

T.right = slist\_to\_bst(L[(mid+1):])

return T

def debug\_bst(A,w):

# Code to test the functions above

start = timeit.default\_timer()

print("Building binary search tree \n")

T = slist\_to\_bst(A)

stop = timeit.default\_timer()

print("Binary Search Tree stats: \n"

+ "Number of nodes: " + str(num\_nodes(T)) + "\n"

+ "Height: " + str(height(T)) + "\n"

+ "Running time for binary search tree construction: " + str(stop-start) + "\n"

)

#Processes words.txt to run many similarities

start = timeit.default\_timer()

for i in range(len(w)):

e0 = Find(T,w[i][0])

e1 = Find(T, w[i][1])

if e0 == -1 or e1 == -1:

print ("No words found")

else:

print("Similarity [" + str(w[i][0]) + "," + str(w[i][1]) + "] = " + str(c\_sim(e0[1],e1[1])))

stop = timeit.default\_timer()

print('Running time for binary search tree query processing: ', stop - start)

#################################################################################################

#LAB METHODS

#################################################################################################

def c\_sim(e0, e1):

#numpy function dot does the dot product of the embeddings (2 lists)

if e0 == None or e1 == None:

return 0.0

dp = np.dot(e0, e1)

#linalg.norm function is a numpy function that gets the magnitude of the embeddings(a vector)

me0 = np.linalg.norm(e0)

me1 = np.linalg.norm(e1)

return dp / (me0 \* me1)

def file\_reader(filename,option):

lfile = []

#with is used to open and close the file once it is done reading

#encoding is utf8 due to errors being generated if it is the wrong filetype

with open(filename,"r",encoding="utf8") as f:

for line in f:

#Appends the string and a list(the embedding)

#Uses map to convert string to float numbers and to only put in the numbers(excludes the string)

if option == 'g':

if line.split()[0] not in string.punctuation: #Skips line if its punctuation

lfile.append([line.split()[0], list(map(float,line.split()[1:]))] )

if option == 't':

l = line.split(',')

lfile.append([l[0], l[1]])

return lfile

def print\_sim(w0,w1):

print("Similarity [" + str(w0[0]) + "," + str(w1[0]) + "] = " + str(c\_sim(w0[1],w1[1])))

#################################################################################################

#MAIN

#################################################################################################

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

fnt = "words.txt"

print ("Reading " + fng + "...\n")

l = file\_reader(fng,'g')

#l = [['b',[1]],['a',[2]],['c',[3]],['d',[4]],['e',[5]],['f',[6]],['g',[7]],['h',[8]],['i',[9]],['j',[10]], ['k',[11]], ['l',[12]]]

#l = l[:100]

try:

print("Choose table implementation: \n"

+ "1. Binary Search Tree\n"

+ "2. Hash Table(Chaining)\n")

choice = int(input())

print ("Reading " + fnt + "...\n")

w = file\_reader(fnt,'t')

#w = [['e','a']]

#Binary Search Tree

if choice == 1:

#Sorts 2d based on alpha order so that it can be put into bst

l.sort(key=itemgetter(0)) #Sort will only sort with first index of each column(a string) as the guideline

debug\_bst(l,w)

#Hash table

elif choice == 2:

debug\_h(l,w)

else:

print("Incorrect Choice")

except ValueError:

print ("Incorrect input")

# 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.”

Name: Kimberly Morales



Signature: