**CS2302 - Data Structures**

**Spring 2019**

**Lab Report 5**

April 5, 2019

Sebastian Gomez

**Introduction:**

Write a simple version of Natural Language Processing (NLP) comparing two words and display the similarities between them. In order to do so, using a binary search tree or a hash table, depending on the choice of the user, store the information of the file provided by the research of Stanford in the choice of implementation to access a word and its embeddings, which are the description of the word. By reading another file containing pairs of words, using the embeddings compare the pairs and output the similarity between them, for example if they are identical words it should output one otherwise it should display the similarity between them.

**Implementation:**

First, I used a while loop to make the user choose between the binary search tree and the hash table. Once the user made its choice by typing 1, for binary search tree, or 2, hash table, I opened the file with the embeddings and the file made by me containing the pairs of words that I wanted to compare. If the user typed 1, it would call the method I created to do the binary search tree and it had as parameter the name of the file I gave to the embeddings and the pairs of words which was f and pairsOfWords. If the user typed 2, then it would call the method for creating a hash table having as parameters both files. If the user type anything else it ended the program.

To build the binary search tree, I had to use the methods given on the webpage plus some extra methods like height, number of nodes and search. To do the height method I checked if the tree was empty and if it was none it returned zero, if not I used the max function to determine which side, left or right, was bigger and used that as the height. For the number of nodes, same base case, if none it returned zero, if not it traversed the left and right tree and added one. For the search method if the tree was empty it returned none, if not, I used a temp variable to keep the tree intact and then did a while loop to search for k. Inside the loop it checked if k was equal to temp’s item at index zero, because that was the word, and if it did it would return item at index one because that is where the embeddings were located. If k was greater than temp’s item then it would traverse the right tree, if less than then it would go the left subtree. If the element was not found, it would exit the for loop because the tree was empty and it would return none. After making and having these methods, I started to do the method for building the binary search tree and do the similarities with the two files. I did a for loop that traversed the file and split it by the spaces it had, then it would insert what it read into the binary search tree using the insert method given. Inside the insert method, when putting the elements into the binary search tree, it put an array containing the word at index zero and at the rest of them it would put the embeddings which were float numbers describing each word. Then I called my methods NumOfNodes and Height to know what the numbers were after insertion. After the tree had been built, which took float running time between 25 to 31, it would be a tree with 400,000 nodes and a height of 53. With the tree done, then I read the file which contained the pairs of words and looked for each word with the search method and its value was stored in e0 and e1. After that it printed both words and the similarity between them by accessing the embeddings, which was possible due to the formula given. To round things up, I used the round method that needed one parameter, which was the number and another one that was optional which was the number of digits. This method made it possible to display the similarity in 4 digits just as the sample run given. For this implementation I had a problem printing all the pairs of words because for some reason it read the file and only kept the last pair and then it computed the similarity, but when trying to do every pair it gave me a float and nonetype error. This complicated things because it would only do the similarity of one pair so the running time for doing the binary search query process was very low.

For the hash table, I used the given code for handling strings. It came with an h method, which returned the index to make insertion possible. I added three methods which was to double the hash table, to know the number of empty lists and the deviation of length. For the doubling size method, I just multiplied the length of the hash table by two to double, after that I just traversed the hash and append the elements into it and adding the number of items every time it appended. For the empty lists’ method, I just traversed the hash using a for loop and checked if there was an empty list, if yes, it would increment the variable I made to keep track of the number of empty lists. For the deviation method I used the load factor and a for loop traversing the hash to increment the deviation by adding the product of the subtraction of the length of each item minus the load factor, then when the for loop ended, I returned the deviation obtained divided by the length of the hash to know what was the deviation of lengths. After having these methods, I started to do the actual method to build the hash table with the embeddings and using it to do the similarities. As I did with the binary search tree, I read the file line by line and split the information by the space it had and then it would store in the hash table. Then I printed the final size, which was 417,791, with a load factor of 0.96, a percentage of empty lists of 99.91, and a float running time between 21 and 23, making it faster than the binary search construction. After doing this, I read the file containing the pairs of words and used the find function to assign to e0 and e1, then printing the words and performing the similarity operation which was the same formula as for the binary search tree. In this case, it did printed out every pair of word and computed the similarities and had a 0.008665 running time for processing.

At first, one issue was for reading the file but once I looked for the problem, I knew that by adding the encoding, the possibility of reading the file was possible. Another of the issues I had when printing the pairs of words was that it printed the second word with \n and that made the similarity result different from what it should be, so in order to remove that problem, I used the function rstrip when reading each line and splitting it. This made things possible for the similarity result to be correct.

**Conclusion:**

This was a confusing and complicated lab so this gave an idea on how Siri, Bixby or Alexa are very complex. I learned how a hash table worked and noticed that the binary search tree can implement anything. I hope I will learn more and improve my work to get better grades, but more importantly to learn what I am supposed to by doing these labs.

**I, Sebastian Gomez, certify that this project is entirely my own work, I wrote, debugged, and tested the code being presented, performed 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.**

**Appendix:**

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

"""

@author: Sebastian Gomez

Course: Data Structure 2302

Assignment: Lab 5

Instructor: Olac Fuentes

T.A: Anindita Nath and Maliheh Zargaran

Purpose: Binary search tree and hash implemenatation of reading a file to compute similarities of a given file

"""

import time

import numpy as np

import math

global start

global end

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 Height(Tr):

if Tr is None:

return 0

return max(Height(Tr.left), Height(Tr.right)) + 1#using max to know which side is longer

def NumOfNodes(T):

if T is None:

return 0

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

def Search(T,k):

if T is None:

return T

temp = T

while temp is not None:

if temp.item[0] == k:

return temp.item[1]

elif temp.item[0] < k:

temp = temp.right

else:

temp = temp.left

return None

def BuildBST(f,pairsOfWords):

print('Building binary search tree')

print()

T = None

start = float(time.time())

for line in f:

info = line.split(' ')

T = Insert(T,[info[0],np.array(info[1:]).astype(np.float)])

end = float(time.time())

print('Binary Search Tree stats:')

print()

print('Number of Nodes: ',NumOfNodes(T))

print('Height: ',Height(T))

print('Running time for constructing binary search tree: ',round((end-start),2))

print()

print('Reading word file to determine similarities')

print()

start = float(time.time())

print('Word Similarities Found:')

for Line in pairsOfWords:

infor = Line.split(' ')

e0 = Search(T,infor[0])

e1 = Search(T,infor[1])

print('Similarity',infor[0:2], '= ', round(np.sum(e0 \*e1) / (math.sqrt(np.sum(e0\*e0)) \* math.sqrt(np.sum(e1\*e1))),4))

end = float(time.time())

print()

print('Running time for binary search tree query processing: ',round((end-start),2))

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 = []

self.num\_items = 0

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

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

H = double(H)

b = h(k[0],len(H.item))

H.item[b].append([k[0],np.array(k[1:]).astype(np.float)])

H.num\_items+=1

return H

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

return -1

def double(H):

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

DoubleH = HashTableC(size)

for i in H.item:

if i != []:

for j in i:

DoubleH.item[h(j[0],len(DoubleH.item))].append([j[0],j[1]])

DoubleH.num\_items += 1

return DoubleH

def emptyList(H):

numOfEmptyLists = 0

for h in H.item:

if h == []:

numOfEmptyLists +=1

return numOfEmptyLists

def deviation(H):

loadFactor = H.num\_items / len(H.item)

deviation = 0

for i in H.item:

deviation += len(i) - loadFactor

return deviation / len(H.item)

def h(s,n):

r = 0

for c in s:

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

return r

def BuildHashTable(f,pairsOfWords):

print('Building hash table with chaining')

print()

print('Hash Table Stats:')

H = HashTableC(50)

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

start = float(time.time())

for line in f:

info = line.split(' ')

H = InsertC(H,info,1)

end = float(time.time())

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

print('Load factor: ',H.num\_items/len(H.item))

e = emptyList(H)

d = deviation(H)

print('Percentage of empty lists: ',(e\*100 / len(H.item)))

print('Standard deviation of the lenghts of the lists: ',d)

print('Running time for hash table construction: ',round((end-start),2))

print()

print('Reading word file to determine similarities')

print()

start = float(time.time())

print('Word Similarities Found:')

for Line in pairsOfWords:

infor = Line.rstrip('\n').split(' ')

e0 = FindC(H,infor[0])

e1 = FindC(H,infor[1])

print('Similarity',infor[0:2], '= ', round(np.sum(e0 \*e1) / (math.sqrt(np.sum(e0\*e0)) \* math.sqrt(np.sum(e1\*e1))),4))

end = float(time.time())

print()

print('Running time for hash table query processing: ',round((end-start),6))

c = '1'

while c == '1' or c == '2':

c = input('Choose table implementation Type 1 for binary search tree or 2 for hash table with chaining ')

print('Choice:', c)

print()

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

pairsOfWords = open('pairs.txt',encoding = 'utf-8',)

if c == '1':

BuildBST(f,pairsOfWords)

elif c == '2':

BuildHashTable(f,pairsOfWords)

f.close()

pairsOfWords.close()