Javier Soto

Lab 5 report

UTEP

## Introduction

The purpose of lab 5 is to compare the run times of two tables used in information retrieval namely hash tables and binary trees. In these tables we need to read from a file containing a large number of strings with their associated embedding’s and store this information in an efficient manner.

## Proposed Solution

My code starts by prompting the user to select the method they would like me to store the information. I have written a catch to make sure only the two options can be chosen from any other action will be rejected. If the user chooses to implement a binary search I start the method by creating variable “numWords” That I will increment by one every time I insert a new item into the tree. Since the binary search tree is an object I can also achieve the same results by adding this attribute to the class itself however I do not see my self needing that attribute in the future.

To start the creating of my binary search tree I was though I needed to add an item to the tree when initializing it. I figured I would create a fake node I called foo node. I use this node to start the creation of the tree then I delete it afterward to have the correct tree. I remembered from our lab I can use a ordered list to crate a balanced binary search tree iteratively, this was my next option. I know that this aproache would double my memory complexity and have a much longer time complexity of at very least O(N^2) to create the list then traverser the list then O(log^n) to insert into the tree. Later while going over my BST class I discovered that I am able to initialize an empty binary tree. Once the tree is initialized, I read line per line in the file. Each line is passed to my “wordnode” object.

My word node class has two attributes one attribute called text that will store the word itself. The second attribute, called embedding is a numpy array of size 50 that stores the embeddings of the word. When I line is passed form the file I use python native list method “Pop”. To return the first index of the of the line, that I know from observation to be the word. Since the word is now removed form line all that remain are the 50 embedding that I now insert into the numpy array specifying that they are to be type casted to floats from strings. I am not sure how accurate this cast is but they seem to be accurate enough for the purpose of this lab. Now that we have created a word node from this line we can add it to the tree.

For insertion of a word into a tree I recycled code from our previous labs and made very minor modifications for compatibility. Previously we use the bst class to store integers, for this lab it will be storing the word nodes and for that reason we cannot compare based on the item attribute of the tree. For the purpose of this lab I compare the text attribute of my word class. The result is a tree that will traverse to the left child if the current roots word is alphabetically after the current comapree, will traverse to right child otherwise. I will revisit this subject later when discussing the find implementation for binary trees of word nodes.

Once the word node is crated and inserted into the tree I then travers to the next line of the file and repeat for all lines in the fine not ignoring any line regardless if it starts with a word or a special character. Immediately before starting to read the file I collected the current system time, once the file is done being read, meaning the tree is already created, I again collect the current system time so that I can recorded the time it takes to create the binary search tree. Now that the tree has been created I am ready to retrieve and display all necessary statistics.

The get the height of the tree I implement a function that will take the root of a tree and compare the length of the left and right subtrees of that root. The larger of the subtrees will be returned recursively until the traversal of the tree is exhausted. For this particular lab I ended up with a tree height of 52 nodes. The number of words contained in this file will be equal to the number of line in the file thus using the numWords variable mentioned earlier I am able to calculate that there are 400k words in this file, again to include special character words. With a time, complexity of O(n) my algorithm was able to reliable build the tree in an average of 12 seconds.

Following the binary search tree case, the next goal is to read a second file of words, search the tree in order to have the embeddings of these words so that we can calculate the cosine similarity between two words. The specification for the second file to read from make for an easy traversal of the file and here is why. Much like the creation of the tree I read line by line doing all required work before moving to the next line. I first read the line then split the line into two sperate strings. With each string I make a call to find the word in the binary search tree. Much like the insert function, the find function took very little modifications to make compatible with the new word node.

In the find function we take advantage of the tree structure by not having to iterate through all 400k nodes. When comparing the searched to the current root we have three options, the current root is the searchee, the searchee is alphabetically before or alphabetically after with this information we traverse the tree accordingly. When the node containing the searchee Is found, the entire sub tree is returned and stored in a temporary variable.

With the two trees stored locally I make a call to function cosine distance that take a parameter of two-word nodes. I choose to make this function as generic as possible so that I will not have to write an additional function to do the same job later when I deal with hash tables. To compute the cosine distance, I use numpys dot product function to get the dot product of the two arrays containing the embedding of the two words that are being compared. Next, I call function magnitude to that uses numpy function to return the magnitude of each words embeddings. Multiplying the two magnitudes and dividing the dot product by that product I can now return and print the cosine distance of the two words.

For the hashtable implementation of my program I use the same algorithm as my tree implementation with the needed modification. I first create a hash table of a large primary number; my initial number is a table size of 9587. I choose this number because I already knew from testing the tree that the file contains 400k words. If I started with a smaller table size, then I would have to resize more times. Alternatively, an excessively large table size would lead to a greater standard deviation and a larger percentage empty. As before when a line is read, a temporary word node is created in the same manor. When inserting into the hash table I learned that to achieve maximum optimization I need to be especially selective with my hash value.

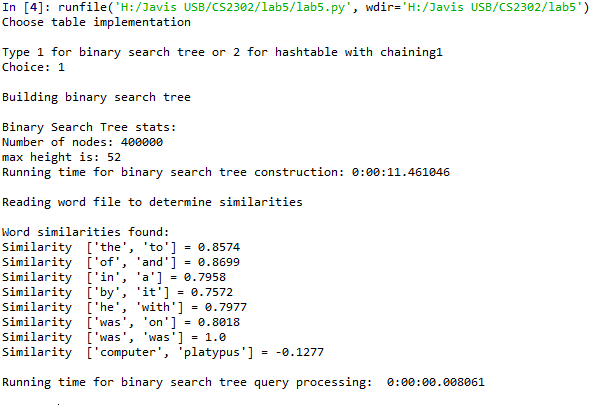
After researching different hash algorithms, I decided the one I want to implement is the one we went over in class that take the position of letters into account when computing the integer hash code. I learned if the position wasn’t taken into consideration there would be more frequent collisions in words that hash the same such as act, cat and tac. I then remember from coding in java when dealing with objects to assert equal I had to overload the equals method and override the hash method which made me curios as to python’s native hash function. I found that python does indeed have a function that takes a string and returns the integer hash value so my insert method takes advantage of this function. To demonstrate my understanding of the function, a string is iterated character by character raising the integer value of the character to the power of the index value of that character in the string then multiplied by the same formula of the next character. The modules of the hash value and the size of the table returns a number that I use and an index of where to store the current word node, all collisions are handled with chaining.

Once the table is created I print the initial size prior to resizing the table. For the table resizing I implement a loop to execute while the load factor of the table remains above 1. To achieve the resizing I create a second table twice the size of the original, I then iterate the entire original table adding each node to the new table as I visit them since they will need to be rehashed with the new table size. I lastly return the new table. It is wise to not resize the table or recheck the load factor after every insertion because after 400k insertions the run time of checking and resizing will be exponential.

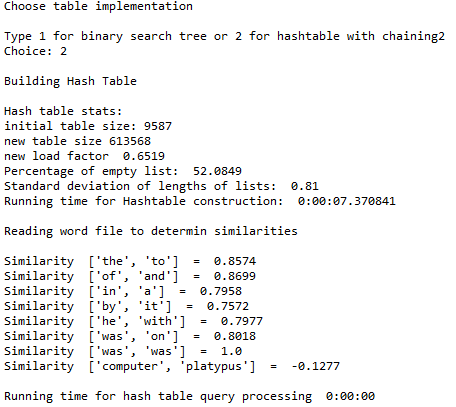
To compute the load factor, I divide the number of elements in the list by the length of the list to get a rough estimate since we will see later that our final table is half empty. To compute the percentage empty, I count the number of buckets that are empty divide that by the total number of buckets in the table and multiply that by 100 to return the empty percentage. Next, I compute the standard deviation. To find the standard deviation I create a list where each index will store the number of nodes in a bucket at the corresponding index, once this list is complete I call numpy standard deviation function with the list and return that number to be printed.

My last obstacle with hash table was to read the second file and compute the cosine similarity between the two-word pairs. When finding the word in the hash table the issue that took me the longest and that I am still not completely clear on was returning the word node. I understand that each bucket will have a list of word nodes which sounds easy enough to access. I made multiple attempts to try and return the word node however I kept returning items of different data types that were not desired. When I was finally able to return the desired word node I was able to use my already created functions to finish the lab.

## Experimental Results

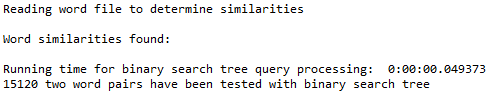


Displaying the output of the binary tree implementation.

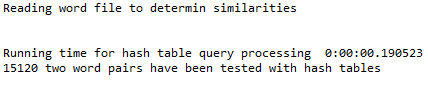


Results of Hash table implementation.

After adding more words to the second file I found that my implementation of the find in the hash table was slower than my implementation of find for my binary search tree. There can be a number of reason why my hastable is not the faster of the two, to include my hash value calculations can be optimized and also my resizing can be optimized. In the future I would to prefer to not double the table every time I need to resize I would rather create an algorithm to resize my table at a slower right when dealing with large table size in order to have a less than 50 percent empty rate.



Binary tree computing cosine similarities of 15k pairs



Hash table computing cosine similarities of 15k items.

At input size of 11 million words my hash table has crashed however prior to crashing we are able to see for the results that the hash table was taking longer than the binary search tree to find the cosine similarities.

## Appendix

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

"""

Created on Thu Mar 28 10:04:14 2019

@author: Javier Soto

Professor Olac fuentes

TAs:

-Anindita Nath

-Maliheh Zaragaran

IA

-Eduardo Lara

Peer Leader

-Erick Macik

"""

import hash\_table\_chain\_strings as HTC

import bst

import Wordnode as WN

from datetime import datetime as dt

import numpy as np

def startUp():

#will prompt user to select data type and call metheod ac

try:

print("Choose table implementation")

answer = int(input("Type 1 for binary search tree or 2 for hashtable with chaining"))

print("Choice: ", end ="")

print(str(answer))

if answer == 1:

print()

print("Building binary search tree")

buildBTree()

elif int(answer) == 2:

print()

print("Building Hash Table")

buildHTable()

else:

print("invalid input")

print()

startUp()

except ValueError:

print("exception reached")

print()

#startUp()

def buildHTable():

#will read from file and create a hash table of words and there embeddings

numWords =0

initSize =9587

myTable = HTC.HashTableC(initSize)

buildStart = dt.now()

theFile = open('H:\Javis USB\CS2302\lab5\glove.6B.50d.txt', encoding = 'utf-8')

#theFile = open('G:\Javis USB\CS2302\lab5\glove.6B.50d.txt', encoding = 'utf-8')

#theFile = open('G:\Javis USB\CS2302\lab5\reduced', encoding = 'utf-8')

#while reading each line, crate a word node and add that new node to the table

for line in theFile:

numWords += 1

tempNode = WN.Word(line)

HTC.InsertC(myTable,tempNode)

buildEnd = dt.now() - buildStart

theFile.close()

print()

print("Hash table stats:")

print("initial table size: " + str(initSize))

while HTC.loadFactor(myTable)>1:

myTable = HTC.resize(myTable)

print("new table size " + str(len(myTable.item)))

print("new load factor ", round(HTC.loadFactor(myTable), 4))

print("Percentage of empty list: ", round(HTC.percentEmpty(myTable), 4))

print("Standard deviation of lengths of lists: ", round(HTC.TableStandardDeviation(myTable),2))

print('Running time for Hashtable construction: ' , buildEnd)

print()

print('Reading word file to determin similarities')

print()

read\_test\_file\_hash(myTable)

def buildBTree():

#will read from file and create a binary tree of words and there embeddings

numWords =0

#foo item used to create the tree outside the loop

#item will be deleted once the tree is created

fooNode = WN.Word("fooItem 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17")

myTree = bst.BST(fooNode)

buildStart = dt.now()

theFile = open('H:\Javis USB\CS2302\lab5\glove.6B.50d.txt', encoding = 'utf-8')

#theFile = open('G:\Javis USB\CS2302\lab5\glove.6B.50d.txt', encoding = 'utf-8')

#theFile = open('reduced.txt', encoding = 'utf-8')

for line in theFile:

numWords += 1

tempNode = WN.Word(line)

bst.Insert(myTree,tempNode)

bst.Delete(myTree,fooNode)

buildEnd = dt.now() - buildStart

theFile.close()

myTreeHeight = bst.MaxHeight(myTree)

print()

print('Binary Search Tree stats:')

print('Number of nodes: ' + str(numWords) )

print('max height is: ' + str(myTreeHeight))

print('Running time for binary search tree construction: ' + str(buildEnd))

print()

print('Reading word file to determine similarities')

print()

similarites\_bst(myTree)

def similarites\_bst(T):

#will take a tree as input

#will read a file of paired words search the tree for the embeddings of these words

#will print the cosine distance of the two words

print("Word similarities found:")

st = dt.now()

theFile2 = open('test.words.txt')

for line in theFile2:

words = line.split()

#will be of type bst

word\_one\_node = bst.Find(T,words[0])

word\_two\_node = bst.Find(T,words[1])

#call to cosineDistance with two word nodes and rounding the retuned value to the 4th decimal

print("Similarity ", words, '=',round(cosineDistance(word\_one\_node.item,word\_two\_node.item), 4))

endt = dt.now()-st

print()

print("Running time for binary search tree query processing: ", endt)

def read\_test\_file\_hash(H):

filePath = 'G:\Javis USB\CS2302\lab5\test.words.txt'

#theFile2 = open(filePath)

htqueryStart = dt.now()

theFile2 = open('test.words.txt')

for line in theFile2:

words = line.split()

word\_one\_node = HTC.FindWordNode(H,words[0])

word\_two\_node = HTC.FindWordNode(H,words[1])

#if word\_one\_node == -1 or word\_two\_node == -1:

# print('item not in list')

# return

dist = cosineDistance(word\_one\_node,word\_two\_node)

print('Similarity ', words ,' = ', round(dist,4))

endtime = dt.now() -htqueryStart

print()

print('Running time for hash table query processing ', endtime)

def cosineDistance(w1,w2):

#will take 2 word nodes and return the cosine distance between those two nodes

dotP = np.dot(w1.embedding,w2.embedding)

magW1= abs(magnitude(w1))

magW2= abs(magnitude(w2))

return dotP/(magW1\*magW2)

def magnitude(item):

#will take a word node and return the magnitude of the assosiated embedding

return np.linalg.norm(item.embedding)

startUp()

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

"""

Created on Thu Mar 28 11:01:53 2019

@author: yatha

"""

import numpy as np

class Word(object):

def \_\_init\_\_(self,wordAndEm=None):

#will take a list use the forst item in the list as the text and the

#remaining 50 items as the embedding

#creates a node (Word) to be used in hastables and bst

items = wordAndEm.split()

self.text = items.pop(0)

self.embedding = np.array(items, dtype=np.float)

def printWord(W):

print(W.text)

Academic dishonesty

I, Javier Soto, certify that this script and lab report are of my own unless otherwise documented above.

