CS2302 - Data Structures

Spring 2019

Lab No.5 Report

Introduction

This lab will cover the use of Hash Tables and Binary Search Trees. Using these data structures to see how applications such as Alexa and Siri work when searching for word embeddings. Doing the same in simpler manner.

Reading a file that has a string in each line followed by 50 embeddings for the string. Then inserting these items into a BST or hash table respectively.

Design & Implementation

Reading file insert into BST:

After the user has chosen to use a BST implementation we start by reading a file. Each item of the BST will contain a list of size 2 having the word string at *item[0]* and the embedding numpy array at *item[1].*

We first start by reading the file line by line, each line will be a string. We also need to remove the *"\n"* character from our last number, this will be done using the ***strip('\n')*** function.

$$line = line.strip('\n')$$

After this is done we will separate the line string by spaces, giving us a list of strings that has 51 items in it, the word and the 50 embedding numbers. Then we perform a check to see if the given word starts with a letter, if it does the method will insert it into the tree.

$$if\ lineList[0][0]\ in\ string.ascii\_lowercase$$

After the past check is passed we then insert the first item of the line list into an item list. After the word is removed from the line list we will have left a list of 50 string numbers, we convert them into float and then insert them into our item list in a form of a numpy array.

$$lineList = list(map(float, lineList))$$

$$item.append(np.array(lineList))$$

Finally, this list will be inserted into our BST using a modified insertion method.

Modified Insertion:

To insert something into our BST we need to give it a numeric value in order to know where it will be placed.

For this we have another method that will give a numerical value to our word string.

```
def wordVal(s):
    val = 0
    y = 0
    for char in s:
        val += ord(char) * 26 ** y
        y += 1
    return val
```

Method will multiply the ASCII value of each character in the word by 26 and elevate it depending on its relative position in the word.

Return the word value obtained, after each character has been evaluated.

Once we have a word value we then call our *Insert* method.

$$BSTree = Insert(BSTree, item, wordVal(item[0]))$$

The parameters are the BST into which we want to insert an item, the list containing the word string and numpy array, and at the end the word value of the item we want to insert. The method receives the word value to not recalculate it each time there is a comparison.

$if\ T\ ==\ None$:

$T\ =\ BST(newItem)$

$elif\ wordVal(T.item[0])\ >\ itemVal$:

Compare the value of the item to be inserted with current node item value.

$T.left\ =\ Insert(T.left, newItem, itemVal)$

$else$:

$T.right\ =\ Insert(T.right, newItem, itemVal)$

$return\ T$

Reading file insert into Hash Table:

Same as when trying to insert into a BST but in here we will consider our load factor and we will not pass the integer value of the word since the hash table doesn't make multiple recursive calls.

Modified InsertionC:

Our hash table insertion has some slight adjustments considering we are trying to insert a list. The hash table constructor also has a new variable named *num_items* that will augment by 1 each time a new item is inserted, this variable will be updated in the insertion method.

Same as with BST we will need to convert the word into an integer value and then obtain the hash index of it.

Obtain index where the new item will be inserted.

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

Update num_items variable.

$H.num\_items\ +=\ 1$

Add item to hash table.

$H.item[b].append(k)$

Once we have inserted a new item into our hash table we will perform a check to see if the load factor has reached 1. This will simply be performed by checking if our *num_items* is equal or greater than the length of *H.item.*

```
if H.num_items >= len(H.item):
    H = reHash(H)
```

If the load factor reaches 1 then the time to look for an item in the hash table will become linear instead of constant. For this we will rehash our hash table.

To rehash we will first initialize a new hash table with size twice of that our current one plus one. Then since the length of this hash table is different, we need to insert the items from the old table to this new one. For them to be in the correct index.

```
def reHash(H):
    n = len(H.item)
    n = (n * 2) + 1
    newH = HashTableC(n)
    for i in range(len(H.item)):
        for j in range(len(H.item[i])):
            InsertC(newH, H.item[i][j])
    return newH
```

Initialize new table with double the size plus 1.

Insert the items from previous table to the new table.

Words to be found and symmetry of them:

The words to be found are taken from a text file and are taken in an identical way from the previous files read. The Similarity method is called in order to obtain the words to be found and then a second method Check is called.

There is a Check method for the BST and another one for the hash table but both follow similar steps.

First finding the word in the appropriate data structure, once this is done the numpy array of the word is then obtained. This numpy array is then used in order to obtain the dot product and magnitude of it. These are obtained using the python numpy functions.

The dot product is then divided by the magnitude obtained and finally this similarity value is returned and printed.

Conclusion

This lab furthered my knowledge in Hash Tables and Binary Search Trees, seeing their differences in how they are built. Hash tables are commonly faster than Binary Search Tree, with their construction time being faster as also the look up of data. Unfortunately my algorithm to obtain the items table indexes is not optimized fully giving me a hash table that has many empty buckets. This making the searching of an item not constant time. Binary Search Trees on the other hand will always have a run time of O(logn), giving us a more slower but also more reliable way of obtaining data.

Outputs & Runtimes

```
Choose table implementation
Type 1 for binary search tree or 2 for hash table with chaining
Choice: 1

Building BST

BST stats:
Number of nodes: 355704
Height: 58
Running time for BST construction: 46.9079

Word similarities found:
Similarity ['bear', 'bear'] = 1.0
Similarity ['barley', 'shrimp'] = 0.5353
Similarity ['barley', 'oat'] = 0.6696
Similarity ['federer', 'baseball'] = 0.287
Similarity ['federer', 'tennis'] = 0.7168
Similarity ['harvard', 'stanford'] = 0.8466
Similarity ['harvard', 'utep'] = 0.0684
Similarity ['harvard', 'ant'] = -0.0267
Similarity ['raven', 'crow'] = 0.615
Similarity ['raven', 'whale'] = 0.3291
Similarity ['spain', 'france'] = 0.7909
Similarity ['spain', 'mexico'] = 0.7514
Similarity ['mexico', 'france'] = 0.5478
Similarity ['mexico', 'guatemala'] = 0.8114
Similarity ['computer', 'platypus'] = -0.1277
Similarity ['argentina', 'cellphone'] = -0.1039
Similarity ['laptop', 'climate'] = -0.117
Similarity ['rainy', 'cloud'] = 0.3599
Similarity ['chair', 'hair'] = 0.4752
Similarity ['horse', 'morse'] = 0.1041
Similarity ['pc', 'hot'] = 0.3417
Similarity ['island', 'bread'] = 0.0921
Similarity ['wheat', 'cup'] = 0.3208
Similarity ['football', 'videogame'] = 0.1741
Similarity ['read', 'news'] = 0.624
Similarity ['date', 'friends'] = 0.3949
Similarity ['suck', 'hate'] = 0.4003
Similarity ['summer', 'new'] = 0.6693
Similarity ['old', 'prank'] = 0.1743
Similarity ['corporate', 'spotify'] = -0.0738
Similarity ['disco', 'stage'] = 0.442

Running time for BST query processing: 0.0115
```

```
Type 1 for binary search tree or 2 for hash table with chaining
Choice: 2

Building hash table with chaining

Hash table stats:
Initial table size: 3
Final table size: 524287
Load factor: 0.6785
Percentage of empty lists: 50.8575
Standard deviation of the lengths of the lists: 0.8261
Time to build hash table: 8.920934617002786

Word similarities found:
Similarity ['bear', 'bear'] = 1.0
Similarity ['barley', 'shrimp'] = 0.5353
Similarity ['barley', 'oat'] = 0.6696
Similarity ['federer', 'baseball'] = 0.287
Similarity ['federer', 'tennis'] = 0.7168
Similarity ['harvard', 'stanford'] = 0.8466
Similarity ['harvard', 'utep'] = 0.0684
Similarity ['harvard', 'ant'] = -0.0267
Similarity ['raven', 'crow'] = 0.615
Similarity ['raven', 'whale'] = 0.3291
Similarity ['spain', 'france'] = 0.7909
Similarity ['spain', 'mexico'] = 0.7514
Similarity ['mexico', 'france'] = 0.5478
Similarity ['mexico', 'guatemala'] = 0.8114
Similarity ['computer', 'platypus'] = -0.1277
Similarity ['argentina', 'cellphone'] = -0.1039
Similarity ['laptop', 'climate'] = -0.117
Similarity ['rainy', 'cloud'] = 0.3599
Similarity ['chair', 'hair'] = 0.4752
Similarity ['horse', 'morse'] = 0.1041
Similarity ['pc', 'hot'] = 0.3417
Similarity ['island', 'bread'] = 0.0921
Similarity ['wheat', 'cup'] = 0.3208
Similarity ['football', 'videogame'] = 0.1741
Similarity ['read', 'news'] = 0.624
Similarity ['date', 'friends'] = 0.3949
Similarity ['suck', 'hate'] = 0.4003
Similarity ['summer', 'new'] = 0.6693
Similarity ['old', 'prank'] = 0.1743
Similarity ['corporate', 'spotify'] = -0.0738
Similarity ['disco', 'stage'] = 0.442

Running time for hash table query processing: 0.0054
```

Appendix

```
import numpy as np
import string
import timeit
import statistics

"""
BTS --------------------------------------------------------------------------
"""
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,itemVal): # Edited to receive string value as a parameter
    if T == None:
        T =  BST(newItem)
    # Compare val of item to be inserted against current node string val
    elif wordVal(T.item[0]) > itemVal:
        T.left = Insert(T.left,newItem,itemVal)
    else:
        T.right = Insert(T.right,newItem,itemVal)
    return T

def Find(T,k,kVal): # Edited to receive string value as a parameter
    # 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:
        if T == None:
            print(k,"not found in BST")
        return T
    # Compare val of item to be inserted against current node string val
    if wordVal(T.item[0]) < kVal:
        return Find(T.right,k,kVal)
    return Find(T.left,k,kVal)

def InOrderD(T,space):
    # Prints items and structure of BST
    if T is not None:
        InOrderD(T.right,space+'   ')
        print(space,T.item[0])
        InOrderD(T.left,space+'   ')

def Height(T): # Obtain Tree height
    if T is None:
        return 0 ;
    else :
        lDepth = Height(T.left)
        rDepth = Height(T.right)
        if (lDepth > rDepth):
            return lDepth+1
        else:
            return rDepth+1

def wordVal(s): # Obtain an integer value for a given string
```

```
    val = 0
    y = 0
    for char in s:
        val += ord(char)*26**y
        y += 1
    return val

def numNodes(T): # Find number of nodes in the Tree
    if T is not None:
        count = 1
        if T.left is not None:
            count += numNodes(T.left)
        if T.right is not None:
            count += numNodes(T.right)
    return count

"""
Hash-Table ----------------------------------------------------------------
"""
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 # Added num
        for i in range(size):
            self.item.append([])
```

```
def InsertC(H,k):
    # Inserts k in appropriate bucket (list)
    # Does nothing if k is already in the table
    b = h(k[0],len(H.item))
    H.num_items += 1
    H.item[b].append(k)

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
    print(k,"not found in hash table")
    return b, -1

def h(s,n):
    r = 0
    for c in s:
        r = (r*26+ord(c))%n
    return r

def reHash(H): # Duplicate table size + 1 and insert all items in new table
    n = len(H.item)
    n = (n*2)+1
    newH = HashTableC(n)
```

```
    for i in range(len(H.item)):
        for j in range(len(H.item[i])):
            InsertC(newH,H.item[i][j])
    return newH

def findEmpty(H): # Find number of empty slots in the hash table
    count = 0
    for i in range(len(H.item)):
        if len(H.item[i]) == 0:
            count += 1
    return count

def stdevSample(H): # Obtain list of different hash table lists sizes
    L = []
    for i in range(len(H.item)):
        L.append(len(H.item[i]))
    return L

"""
File to BST -----------------------------------------------------------------
Read the text file line by line, convert the string line into a list of strings
separating each element when a space is found. Insert the word of the string
into an item list and also the numpy array. Insert the item list into the BST.
"""
def File2BST():
    file = open("glove.6B.50d.txt", encoding='utf-8')
    BSTree = None
    for line in file:
```

```
        item = []
        line = line.strip('\n')
        lineList = line.split(" ") # Convert string into a list
        if lineList[0][0] in string.ascii_lowercase:
            item.append(lineList.pop(0)) # Append the word to the list
            lineList = list(map(float,lineList)) # Convert string array into float array
            item.append(np.array(lineList)) # Append the embedding of the word
            BSTree = Insert(BSTree,item,wordVal(item[0])) # Insert list into tree
    return BSTree

"""
File to Hash-Table ----------------------------------------------------------
Same as with File to BST but this method instead inserts the item list into a
hash table.
"""
def File2H(H):
    file = open("glove.6B.50d.txt", encoding='utf-8')
    for line in file:
        item = []
        line = line.strip('\n')
        lineList = line.split(" ")
        if lineList[0][0] in string.ascii_lowercase:
            item.append(lineList.pop(0))
            lineList = list(map(float,lineList))
            item.append(np.array(lineList))
            InsertC(H,item)
            if H.num_items >= len(H.item): # If load factor becomes 1 double table size
                H = reHash(H)
```

```
    return H

"""
Similarity Check BST ------------------------------------------------------
Obtain words to be looked up in BST. Search each node and traverse the BST
depending on the value of each string.
"""
def Similarity(T):
    file = open("test.txt") # Words to be looked up
    for line in file:
        line = line.strip('\n')
        words = line.split(" ")
        similarity = CheckT(T,words[0],words[1]) # Call check method
        if similarity != None: # Check to see if word was not found
            print("Similarity",words,"=",similarity)

def CheckT(T,s1,s2):
    s1Node = Find(T,s1,wordVal(s1)) # Obtain node of frist word
    s2Node = Find(T,s2,wordVal(s2)) # Obtain node of second word
    if s1Node == None or s2Node == None: # Word was not found return None
        return
    product = np.dot(s1Node.item[1],s2Node.item[1])
    magnitude = np.linalg.norm(s1Node.item[1])*np.linalg.norm(s2Node.item[1])
    return round(product/(magnitude),4) # Return similarity rounded up

"""
Similarity Check Table ----------------------------------------------------
Obtain words to be looked up in hash table. Traverse the hash table depenging
```

```
on the value of each string.
"""
def SimilarityC(H):
    file = open("test.txt") # Words to be looked up
    for line in file:
        line = line.strip('\n')
        words = line.split(" ")
        similarity = CheckC(H,words[0],words[1])
        if similarity != None: # Check to see if word was not found
            print("Similarity",words,"=",similarity)

def CheckC(H,s1,s2):
    b1,p1 = FindC(H,s1) # Obtain bucket and position of first word
    b2,p2 = FindC(H,s2) # Obtain bucket and position of second word
    if p1 == -1 or p2 == -1: # Word was not found return None
        return
    s1Array = H.item[b1][p1][1] # Numpy array of first word
    s2Array = H.item[b2][p2][1] # Numpy array of second word
    product = np.dot(s1Array,s2Array)
    magnitude = np.linalg.norm(s1Array)*np.linalg.norm(s2Array)
    return round((product/magnitude),4)

"""
Main ------------------------------------------------------------------------
"""
print("Choose table implementation")
print("Type 1 for binary search tree or 2 for hash table with chaining",end='')
choice = 0
```

```
while 1: # Continue until 1 or 2 is input
    try:
        choice = int(input("Choice: "))
        if choice == 1 or choice == 2:
            break
        else:
            print("Choose 1 or 2")
    except:
        print("Choose 1 or 2")

if choice == 1:
    print("\nBuilding BST")
    start_time = timeit.default_timer()
    BSTree = File2BST()
    end_time = timeit.default_timer()
    print("\nBST stats:")
    print("Number of nodes:",numNodes(BSTree))
    print("Height:",Height(BSTree))
    print("Running time for BST construction:",round((end_time - start_time),4))
    print("\nWord similarities found:")
    start_time = timeit.default_timer()
    Similarity(BSTree)
    end_time = timeit.default_timer()
    print("\nRunning time for BST query processing:",round((end_time - start_time),4))

if choice == 2:
    initialSize = 3
    H = HashTableC(initialSize)
```

```
    print("\nBuilding hash table with chaining")
    start_time = timeit.default_timer()
    H = File2H(H)
    end_time = timeit.default_timer()
    print("\nHash table stats:")
    print("Initial table size:",initialSize)
    print("Final table size:",len(H.item))
    print("Load factor:",round((H.num_items/len(H.item)),4))
    empty = findEmpty(H)
    print("Percentage of empty lists:",round(((empty*100)/len(H.item)),4))
    sample = stdevSample(H)
    print("Standard deviation of the lengths of the lists:",round(statistics.stdev(sample),4))
    print("Time to build hash table:", end_time - start_time)
    print("\nWord similarities found:")
    start_time = timeit.default_timer()
    SimilarityC(H)
    end_time = timeit.default_timer()
    print("\nRunning time for hash table query processing:",round((end_time -
start_time),4))
```

I certify that this essay is original work prepared by me, Jesus A Hernandez.