**Lab #5**

CS 2302

Ana Luisa Mata Sánchez

# Introduction

This lab reads a file with a string and 50 vector values. After reading it will place value sin either a hash table or a binary search tree depending on what the user decides. It will also read a file with multiple pairs of words, look them up in the selected data structure, and calculate the similarity between those two words.

# Proposed solution design and implementation

**Module 1 – H**

This module calculates the hash for each item. It multiplies the variable “r” that is changed each iteration by a constant and adds the ordinal value of each character in the string. It returns the value of r modded by the length of the table.

**Module 2 – CreateHashFromArray**

This method creates a hash table from an array of words and numpy arrays. If the Load Factor of the table is less than one it keeps inserting the items, if it gets larger it calls the doubleTheHash method that doubles the hash table.

**Module 3 – doubleTheHash**

This method creates a new table that is twice the size plus one of the old one. After that it reinserts all of the items in the old one and returns the new one.

**Module 4 – LoadFact**

This method returns the number of items divided by the size of the table.

**Module 5 – EmptyListPercent**

This method calculates the percentage of empty lists on the hash table. It goes through every bucket and if the bucket is empty it adds one to the counter. In the end it divides the number of empty buckets by the length of the item and multiplies by 100 to get the percentage.

**Module 6 – StandardDev**

This method goes through each bucket and saves each length in an array. After that is uses the stdev function in the statistics library to calculate the standard deviation.

**Module 7 – PrintSimilarityC**

This method prints the similarity between two strings. It uses the FindC method to find each string in the hash table. If one them is not found it just returns. To calculate the similarity it needs the dot product of both vectors and the magnitude of each. To calculate these it uses np.dot and np.linalg.norm. The similarity isi the dot product divided by the multiplication of both magnitudes. It will then print both strings and their similarity. In order to calculate the query times without printing I just comment the print line.

**Module 8 – Depth**

If the tree is none or if it reaches a null node it returns 0. To iterate through the tree it takes the max depth between the right and left side of the tree and adds one for each node it visits..

**Module 9 – NumNodes**

If the tree is none or if it reaches a null node it returns 0. To iterate through the tree it goes to each right and left node and adds one for each node it visits.

**Module 10 – PrintSimilarityBST**

This method prints the similarity between two strings. It uses the Find method to find each string in the BST. If the node is found it calculates the similarity. To calculate the similarity it needs the dot product of both vectors and the magnitude of each. To calculate these it uses np.dot and np.linalg.norm. The similarity is the dot product divided by the multiplication of both magnitudes. It will then print both strings and their similarity. In order to calculate the query times without printing I just comment the print line. If it doesn’t find either string it will print “not found” and return.

**Module 11 – readFile**

This method read the initial word and vector file. It opens the file and for each line in the file it stores each word in an array. The first item of the array is always the word and the rest are the vectors. If the word is alphanumeric it stores all the numerical vector values in an array. After this it appends the word and the converted vector array into the final array.

**Module 11 – ReadSimilFile**

This method opens the file that contains the pairs of words that are going to be compared. For each line in the text file it each word to each index in the array. After that it inserts only the first two words (in case there are more words in the file) into the final array.

# Experimental results

**\*File used for examples: glove.6B.50d**

**\*File used for similarity**

**TypeA:**

bear bear

barley shrimp

barley oat

federer baseball

federer tennis

harvard stanford

harvard utep

harvard ant

raven crow

raven whale

spain france

spain mexico

mexico france

mexico guatemala

computer platypus

**TypeB:**

barley shrimp

echo voice

barley oat

federer baseball

federer tennis

harvard stanford

harvard utep

harvard ant

taco burrito

meat fish

light pen

raven crow

raven whale

spain france

spain mexico

wolf water

food bread

sweet bottle

phone computer

moist eye

mexico france

keyboard glasses

apple dark

car school

colossal metal

half truck

temper division

pathetic oafish

current drab

first canvas

petite vast

tasteful bow

absurd slow

afraid nutritious

blade frightened

mexico guatemala

computer platypus

|  |  |
| --- | --- |
| **Method call** | **Output** |
| var = 2  ComparisonFileName = **TypeA** | Hash table stats:  Initial table size: 7  Final table size: 262143  Load factor: 0.28242981883933577  Percentage of empty lists: 31.549574087425565 %  Standard deviation of the lengths of the lists: 1.2929636047121746  Running time for hash table construction: 4.324402093887329  Enter name of file to compare similarity  wordstocompare.txt  Reading word file to determine similarities  Running time for hash table query processing: 0.0010006427764892578 |
| var = 1  ComparisonFileName = **TypeA** | Binary Search Tree stats:  Number of nodes: 336158  Height: 49  Running time for binary search tree construction: 3.9649698734283447  Enter name of file to compare similarity  wordstocompare.txt  Reading word file to determine similarities  Running time for binary search tree query processing: 0.0 |
| var = 2  ComparisonFileName = **TypeB** | Hash table stats:  Initial table size: 7  Final table size: 262143  Load factor: 0.28242981883933577  Percentage of empty lists: 31.549574087425565 %  Standard deviation of the lengths of the lists: 1.2929636047121746  Running time for hash table construction: 4.6027467250823975  Enter name of file to compare similarity  wordstocompare.txt  Reading word file to determine similarities  Running time for hash table query processing: 0.0010044574737548828 |
| var = 1  ComparisonFileName = **TypeB** | Binary Search Tree stats:  Number of nodes: 336158  Height: 49  Running time for binary search tree construction: 4.364353656768799  Enter name of file to compare similarity  wordstocompare.txt  Reading word file to determine similarities  Running time for binary search tree query processing: 0.0009975433349609375 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BSTConstruction | BSTQuery (Type A) |  | HashConstruction | HashQuery (Type A) |
| 4.097040892 | 0.001028061 |  | 5.371596813 | 0.000997066 |
| 4.636634111 | 0.001024485 |  | 4.954748392 | 0.000996828 |
| 4.344379187 | 0.001024723 |  | 5.208598614 | 0.000964403 |
| 4.150864124 | 0.000997782 |  | 5.402582884 | 0.000977039 |
| 4.45112586 | 0.000996828 |  | 4.946736813 | 0.00099802 |
| 5.273866892 | 0.000997543 |  | 5.176156521 | 0.000995398 |
| 4.588728428 | 0.001967669 |  | 5.195103168 | 0.000997782 |
| 4.299527168 | 0.000963688 |  | 5.092381716 | 0.0010252 |
| 4.554849148 | 0.000993729 |  | 5.186145544 | 0.00099802 |

# Conclusion

The hash table is faster at generating queries than the BST but the Hash table takes longer to construct than the BST.

**“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.”**

-Ana Luisa Mata Sánchez

# Appendix

|  |
| --- |
| # Author: Ana Luisa Mata Sanchez |
|  | # Course: CS2302 |
|  | # Assignment: Lab #5 |
|  | # Instructor: Olac Fuentes |
|  | # Description: Hash table and BST implementation |
|  | # T.A.: Anindita Nath, Malilheh Zargaran |
|  | # Last modified: 04/03/2019 |
|  | # Purpose: Calculate the similarity between two strings using hash tables and binary search tree to store words |
|  | # and vectors |
|  |  |
|  |  |
|  | import numpy as np |
|  | import statistics |
|  | import time |
|  | ############################################### HASH SECTION ############################################### |
|  |  |
|  | 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 |
|  | 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 |
|  |  |
|  | ############################################### MY CODE ############################################### |
|  |  |
|  | #calculates index for new item |
|  | def h(s,n): |
|  | r = 0 |
|  | for c in s: |
|  | k = int(ord(c)/5)+1 |
|  | #in an attempt to do a better job, the key changes each iteration |
|  | r += (r\*k + ord(c)) |
|  | #Mods so that it is within range |
|  | return r% n |
|  |  |
|  | #Creates the hash table using an array |
|  | def CreateHashFromArray(A,initsize): |
|  | H = HashTableC(initsize) |
|  | for i in range(len(A)): |
|  | #check if load factor is less than one, if it is continue insertin |
|  | if LoadFact(H)<1 : |
|  | InsertC(H,A[i][0],A[i][1]) |
|  | H.num\_items+=1 |
|  | #table is too packed, double the size |
|  | else: |
|  | NH = doubleTheHash(H) |
|  | H = None |
|  | H = NH |
|  | InsertC(H,A[i][0],A[i][1]) |
|  | H.num\_items+=1 |
|  | return H |
|  |  |
|  | #Doubles a hash table |
|  | def doubleTheHash(H): |
|  | B = HashTableC((len(H.item)\*2)+1) |
|  | for k in range(len(H.item)): |
|  | for m in range(len(H.item[k])): |
|  | #reinsert each item in old hash to calculate a new hash index |
|  | InsertC(B,H.item[k][m][0],H.item[k][m][1]) |
|  | return B |
|  |  |
|  | #calculates the load factor |
|  | def LoadFact(H): |
|  | return H.num\_items/len(H.item) |
|  |  |
|  | #returns the percent of emptyt lists |
|  | def EmptyListPercent(H): |
|  | emptyls = 1 |
|  | for i in range(len(H.item)): |
|  | if not H.item[i]: |
|  | #everytime a list is empty, the counter is incrementes |
|  | emptyls+=1 |
|  | #empty lists divided by the total lengths to get the proportion |
|  | return (emptyls\*100)/len(H.item) |
|  |  |
|  | #calculates the standar deviation |
|  | def StandardDev(H): |
|  | data = [] |
|  | for i in range(len(H.item)): |
|  | #append each length of each bucket to an array |
|  | data.append(len(H.item[i])) |
|  | #returns the standard deviation of the length of each list |
|  | return statistics.stdev(data) |
|  |  |
|  | #prints the similarity between two strings |
|  | def PrintSimilarityC(H, s1, s2): |
|  | #Finds each string within the hash table |
|  | b1,i1,data1 = FindC(H, s1) |
|  | b2,i2,data2 = FindC(H, s2) |
|  | #If it did not find it, it does not do anything |
|  | if i1==-1 or i2==-1: |
|  | if i1 == -1: |
|  | print(s1, "not found") |
|  | if i2 == -1: |
|  | print(s1, "not found") |
|  | return |
|  | #Calculates the dot product and magnitudes using the vector |
|  | DotProduct = np.dot(data1,data2) |
|  | magnitude1 = np.linalg.norm(data1) |
|  | magnitude2 = np.linalg.norm(data2) |
|  | similarity = DotProduct/(magnitude1\*magnitude2) |
|  |  |
|  | sim = "Similarity [" + s1 + "," + s2 + "] =" |
|  | print(sim, round(similarity,4)) |
|  |  |
|  | ############################################### BST SECTION ############################################### |
|  |  |
|  | 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 Delete(T,del\_item): |
|  | if T is not None: |
|  | if del\_item < T.item: |
|  | T.left = Delete(T.left,del\_item) |
|  | elif del\_item > T.item: |
|  | T.right = Delete(T.right,del\_item) |
|  | else: # del\_item == T.item |
|  | if T.left is None and T.right is None: # T is a leaf, just remove it |
|  | T = None |
|  | elif T.left is None: # T has one child, replace it by existing child |
|  | T = T.right |
|  | elif T.right is None: |
|  | T = T.left |
|  | else: # T has two chldren. Replace T by its successor, delete successor |
|  | m = Smallest(T.right) |
|  | T.item = m.item |
|  | T.right = Delete(T.right,m.item) |
|  | return T |
|  |  |
|  | def InOrder(T): |
|  | # Prints items in BST in ascending order |
|  | if T is not None: |
|  | InOrder(T.left) |
|  | print(T.item,end = ' ') |
|  | InOrder(T.right) |
|  |  |
|  | 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+' ') |
|  |  |
|  | def SmallestL(T): |
|  | # Returns smallest item in BST. Returns None if T is None |
|  | if T is None: |
|  | return None |
|  | while T.left is not None: |
|  | T = T.left |
|  | return T |
|  |  |
|  | def Smallest(T): |
|  | # Returns smallest item in BST. Error if T is None |
|  | if T.left is None: |
|  | return T |
|  | else: |
|  | return Smallest(T.left) |
|  |  |
|  | def Largest(T): |
|  | if T.right is None: |
|  | return T |
|  | else: |
|  | return Largest(T.right) |
|  |  |
|  | 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 T.item[0] == k: |
|  | return T |
|  | if T.item[0]<k: |
|  | return Find(T.right,k) |
|  | return Find(T.left,k) |
|  |  |
|  | def FindAndPrint(T,k): |
|  | f = Find(T,k) |
|  | if f is not None: |
|  | print(f.item,'found') |
|  | else: |
|  | print(k,'not found') |
|  |  |
|  | ############################################### MY CODE ############################################### |
|  |  |
|  | #Calculates the depth of the tree |
|  | def Depth(T): |
|  | if T is None: |
|  | #if it reaches the end or is a null tree, depth = 0 |
|  | return 0 |
|  | else: |
|  | #takes the larger value from either the left or right side and adds one for each level traversed |
|  | return max(Depth(T.left), Depth(T.right)) + 1 |
|  |  |
|  | #Calculates number of nodes |
|  | def NumNodes(T): |
|  | if T==None: |
|  | #if it reaches the end or is a null tree, num nodes = 0 |
|  | return 0 |
|  |  |
|  | #adds one for each right and/or left node |
|  | return NumNodes(T.right) + NumNodes(T.left) + 1 |
|  |  |
|  | #Prints similarity between two strings |
|  | def PrintSimilarityBST(T, s1, s2): |
|  | #Finds the string within the BST |
|  | T1 = Find(T,s1) |
|  | T2 = Find(T,s2) |
|  | #Checks if it was found, if it was it makes the data variables equal to their respective vectors |
|  | if T1 != None or T2 != None: |
|  | data1 = T1.item[1] |
|  | data2 = T2.item[1] |
|  | else: |
|  | if T1 == None: |
|  | print(s1, "not found") |
|  | if T2 == None: |
|  | print(s2, "not found") |
|  | return |
|  | #Calculates the similarity using the vectors |
|  | DotProduct = np.dot(data1,data2) |
|  | magnitude1 = np.linalg.norm(data1) |
|  | magnitude2 = np.linalg.norm(data2) |
|  | similarity = DotProduct/(magnitude1\*magnitude2) |
|  |  |
|  | #prints the similarity, creating the string is optional and its only purpose is cosmetic |
|  | sim = "Similarity [" + s1 + "," + s2 + "] =" |
|  | print(sim, round(similarity,4)) |
|  | ############################################### Others ############################################### |
|  | #reads the initial file and parses it into an array |
|  | def readFile(filename): |
|  | NodeArray = [] |
|  |  |
|  | with open(filename, encoding="utf8") as f: |
|  | for line in f: |
|  | Vec = [] |
|  | #reads line by line, stores that line in an array |
|  | lines = line.split() |
|  | #The first thing in the line array is the word |
|  | Word = lines[0] |
|  |  |
|  | if Word.isalnum(): |
|  | #After that it parses each number and appends it to a list |
|  | for i in range(1,len(lines)): |
|  | Vec.append(float(lines[i])) |
|  | #make that list into a numpy array and insert with word to the final array |
|  | NodeArray.append([Word,np.array(Vec)]) |
|  |  |
|  | return NodeArray |
|  |  |
|  | #Reads each pair of words in the similarity file |
|  | def ReadSimilFile(filename): |
|  | with open(filename, encoding="utf8") as f: |
|  | words = [] |
|  | for line in f: |
|  | #reads line by line, stores that line in an array |
|  | lines = line.split() |
|  | #the first word is stored at index 0, the secocnd at 1 |
|  | words.append([lines[0],lines[1]]) |
|  | return words |
|  |  |
|  | ############################################### Run the code ############################################### |
|  |  |
|  | var = input('Choose a table implementation \nType 1 for binary search tree \nor 2 for hash table with chaining \n') |
|  | try: |
|  | A = readFile('test.txt') |
|  | #glove.6B.50d |
|  |  |
|  | #if user selects hash table |
|  | if var == "2": |
|  | size = 7 |
|  | #creates the hash table |
|  | H = CreateHashFromArray(A,size) |
|  | print('\nHash table stats:') |
|  | print('Initial table size: ', size) |
|  | print('Final table size:', len(H.item)) |
|  | print('Load factor:', LoadFact(H)) |
|  | print('Percentage of empty lists:', EmptyListPercent(H), "%") |
|  | print('Standard deviation of the lengths of the lists:', StandardDev(H)) |
|  |  |
|  | ComparisonFileName = input('Enter name of file to compare similarity\n') |
|  | #wordstocompare.txt |
|  | #Reads the compare similarity file into an array |
|  | ComparisonArray = ReadSimilFile(ComparisonFileName) |
|  |  |
|  | print('\nReading word file to determine similarities\n') |
|  |  |
|  | #calculates time that query takes |
|  | iquerytime = time.time() |
|  | for i in range(len(ComparisonArray)): |
|  | PrintSimilarityC(H, ComparisonArray[i][0], ComparisonArray[i][1]) |
|  | fquerytime = time.time() |
|  |  |
|  | querytime = fquerytime - iquerytime |
|  | print("Running time for hash table query processing:",querytime) |
|  |  |
|  | elif var == "1": |
|  | T = None |
|  | #Calculates the time that it takes to create the binary tree |
|  | iconstructTime = time.time() |
|  | for a in A: |
|  | T = Insert(T,a) |
|  | fconstructTime = time.time() |
|  | constructTime = fconstructTime -iconstructTime |
|  |  |
|  | print('\nBinary Search Tree stats:') |
|  | print("Number of nodes:",NumNodes(T)) |
|  | print("Height:", Depth(T)) |
|  | print("Running time for binary search tree construction:", constructTime) |
|  |  |
|  | ComparisonFileName = input('Enter name of file to compare similarity\n') |
|  | #Reads the compare similarity file into an array |
|  | ComparisonArray = ReadSimilFile(ComparisonFileName) |
|  |  |
|  | print('\nReading word file to determine similarities\n') |
|  |  |
|  | #calculates time that query takes |
|  | iquerytime = time.time() |
|  | for i in range(len(ComparisonArray)): |
|  | PrintSimilarityBST(T, ComparisonArray[i][0], ComparisonArray[i][1]) |
|  | fquerytime = time.time() |
|  |  |
|  | querytime = fquerytime - iquerytime |
|  |  |
|  | print("\nRunning time for binary search tree query processing:",querytime) |
|  |  |
|  | except FileNotFoundError: |
|  | print('File not found') |
|  | else: |
|  | print("Incorrect input") |