Robert Marc

80487972

CS 2302

MW 1:30-3:00

Lab 5 Report

Introduction

Lab 5 required us to use Binary Search Trees and hash tables to store information read from a text file and then use query results to compare similarities between words.

Proposed Solutions

The program begins by requesting which type of data structure the user wants to use. Based on that selection, it either calls the main Query function depending on if the choice is binary search trees (BSTQuery) or hash tables (HashQuery). Each of the Query functions read the glove file into a holder object. Right before the functions that build the data structure are called, the start time is logged. Because the first character of the glove file is an encoding related character, that first character is sliced out of the list. Then line by line, the Build functions work through the file, splitting the first element of the string array and storing it as the word, and then the remaining 50 elements of the string array are converted into float values and stored in the embd variable. Using the built in insert functions for both BST’s and Hash tables, the word and embd are sorted and stored in the appropriate location. The duration of the build time is calculated on return.

On return to the main Query function, statistics related to the data structure in use are printed. Next, the comparisons text file is read into a holder object, and the appropriate CompQuery function is called. This splits apart and trims the two words to be compared of encoding related characters, then finds the bucket or node containing the word and its embd list. The lists are then used to calculate the similarities between the words. Each word pair and its similarity are printed. The time for each comparison query is logged and stored to determine the total elapsed time of the query.

Experimental Results

Average Build Time for Binary Search Tree and Hash Table (10 tests)

* Binary Search Tree – 10.64 seconds
* Hash Table – 10.377 seconds

Average Comparison Query Time for N comparisons (10 tests each) in seconds

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| N = | 1,000 | 2,000 | 4,000 | 8,000 | 16,000 | 32,000 |
| BinarySearchTree | 0.00244 | 0.0025 | 0.0053 | 0.01 | 0.0215 | 0.0422 |
| Hash Table | 0.00135 | 0.0027 | 0.006 | 0.0123 | 0.0208 | 0.0421 |

Conclusions

Both Data Structures seem to have a similar growth rate to that of the problem size. When the number of comparisons doubled, so did the average time to process those comparisons. Reading the file was the most difficult part of the lab because of the extra characters that were in front of or at the end of words as they were parsed.

Academic Honesty Statement

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.”



Appendix

Lab5.py

"""

@Course: CS2302 MW 1:30-2:50 pm

@Author: Robert Marc, 80487972

@Assignment: Lab 5

@Instructor: Dr. Olac Fuentes

@TAs: Anindita Nath and Maliheh Zargaran

@Date of Last Modification: 4/2/19 @6:19PM

@Purpose: To explore efficiency between binary search trees and hash tables

"""

import BST

import HashString

import time

import math

def BSTQuery():

"""

Main method for Binary Search Tree runtime

Reads glove file and stores in file object

Starts timer and then calls BSTBuild to build the Binary Search Tree

Ends timer on return.

Prints simple statistics about constructed tree

Reads comparisons file and stores in hold object

Calls BSTCompQuery to start comparison queries

Prints elapsed query time

"""

print("Building binary search tree.",end="\n\n")

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

lines = glove.readlines()

glove.close()

start = time.time()

T = BSTBuild(lines)

elapsed = time.time() - start

print("Binary Search Tree Stats:")

print("Number of Nodes: ",BST.NumNodes(T))

print("Height: ",BST.Height(T))

print("Running time for binary search tree construction: ",round(elapsed,2)," seconds",end="\n\n")

print("Reading word file to determine similarities.",end="\n\n")

comp = open("comparisons.txt")

hold = comp.readlines()

comp.close()

print("Word similarities found:",end="\n\n")

elapsed = BSTCompQuery(T,hold)

print()

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

def BSTBuild(file):

"""

Receives a file object and returns a build Binary Search Tree

Slices off the first character of the file, as it is an encoding character

Splits each line of the file into a string array

stores the first element in word

stores the remaining elements of the string array into embd as floats

Uses permC as a character list of approved characters that a word can begin with

Inserts each line into a node in the binary search tree

returns the built tree

"""

permC = 'abcdefghijklmnopqrstuvwxyz'

file[0] = file[0][1:]

s = file[0].split()

word = s[0]

embd = [float(k) for k in s[1:]]

T = BST.BST([word,embd])

if word[0] in permC:

T = BST.Insert(T,[word,embd])

for i in file[1:]:

i = i.split()

word = i[0]

embd = [float(j) for j in i[1:]]

if word[0] in permC:

BST.Insert(T,[word,embd])

return T

def BSTCompQuery(T,file):

"""

Comparison query function for binary search tree

For each line in the file, it stores the elapsed time,

the two words to be compared (trimming non alphanumeric characters)

and prints the return similarity results

Returns total elapsed time not counting print functions

"""

elapsed = 0

permC = 'abcdefghijklmnopqrstuvwxyz'

for i in file:

start = time.time()

s = i.split(",")

if s[0][0] not in permC:

s[0] = s[0][1:]

if s[1][len(s[1])-1] not in permC:

s[1] = s[1][:-1]

elapsed += time.time()-start

print("Similarity [",s[0],",",s[1],"] = ",sim(BST.Find(T,s[0]),BST.Find(T,s[1])))

return elapsed

def HashQuery():

"""

Main method for hash table with chaining runtime

Reads glove file and stores in file object

Starts timer and then calls HashBuild to build the hash table

Ends timer on return.

Prints simple statistics about constructed tree

Reads comparisons file and stores in hold object

Calls HashCompQuery to start comparison queries

Prints elapsed query time

"""

print("Building hash table with chaining.",end="\n\n")

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

lines = glove.readlines()

glove.close()

start = time.time()

size = 1

H = HashBuild(lines,size)

elapsed = time.time() - start

print("Hash Table Stats:")

print("Initial Table Size: ",size)

print("Final Table Size: ",len(H.item))

print("Load factor: ",round(HashString.load(H),2))

print("Percentage of Empty Lists: ",round(HashString.empty(H),2),"%")

print("Standard deviation of the lengths of the lists: ",round(HashString.standDev(H),3))

print("Running time for hash table construction: ",round(elapsed,2)," seconds",end="\n\n")

print("Reading word file to determine similarities.",end="\n\n")

comp = open("comparisons.txt")

hold = comp.readlines()

comp.close()

print("Word similarities found:",end="\n\n")

elapsed = HashCompQuery(H,hold)

print()

print("Running time for hash table with chaining query processing: ",elapsed," seconds")

def HashBuild(file,size):

"""

Receives a file object and returns a built hash table

Slices off the first character of the file, as it is an encoding character

Splits each line of the file into a string array

stores the first element in word

stores the remaining elements of the string array into embd as floats

Uses permC as a character list of approved characters that a word can begin with

Inserts each line into the hash table

If the number of items in the table ever equals the current number of buckets,

rebuilds the hash table with a new size equal to (size\*3)+1

returns the hash table

"""

permC = 'abcdefghijklmnopqrstuvwxyz'

file[0] = file[0][1:]

s = file[0].split()

word = s[0]

embd = [float(k) for k in s[1:]]

H = HashString.HashTableC(size)

if word[0] in permC:

HashString.InsertC(H,word,embd)

for i in file[1:]:

i = i.split()

word = i[0]

embd = [float(j) for j in i[1:]]

if word[0] in permC:

HashString.InsertC(H,word,embd)

if H.numItems == size:

size = (size\*2)+1

H = HashRebuild(H,size)

return H

def HashRebuild(H,size):

"""

Takes the passed hash table and builds a new table of the given size

with all elements in the old table

Returns new table

"""

temp = HashString.HashTableC(size)

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

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

HashString.InsertC(temp,H.item[i][j][0],H.item[i][j][1])

return temp

def HashCompQuery(H,file):

"""

Comparison query function for hash table

For each line in the file, it stores the elapsed time,

the two words to be compared (trimming non alphanumeric characters)

and prints the return similarity results

Returns total elapsed time not counting print functions

"""

elapsed = 0

permC = 'abcdefghijklmnopqrstuvwxyz'

for i in file:

start = time.time()

s = i.split(",")

if s[0][0] not in permC:

s[0] = s[0][1:]

if s[1][len(s[1])-1] not in permC:

s[1] = s[1][:-1]

elapsed += time.time()-start

print("Similarity [",s[0],",",s[1],"] = ",sim(HashString.FindC(H,s[0]),HashString.FindC(H,s[1])))

return elapsed

def sim(e0,e1):

dot = 0.0

uMag = 0.0

vMag = 0.0

for i in range(len(e0)):

dot += (e0[i] \* e1[i])

uMag += e0[i]\*\*2

vMag += e1[i]\*\*2

uMag = math.sqrt(uMag)

vMag = math.sqrt(vMag)

return round(dot/(uMag \* vMag),4)

print("Choose table implementation:")

choice = input("Type 1 for binary search tree or 2 for hash table with chaining:")

print("Choice: ",choice,end="\n\n")

if choice== "1":

BSTQuery()

elif choice == "2":

HashQuery()

else:

print("Input not recognized, please try again.")

HashString.py

# Implementation of hash tables with chaining using strings

import numpy as np

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.numItems = 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])

H.numItems += 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 H.item[b][i][1]

return b, -1, -1

def h(s,n):

"""

Modified formula to r = (r\*50 + ord(c))% n from r = (r\*n + ord(c))% n

"""

r = 0

for c in s:

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

return r

def load(H):

"""

Calculates load of hash table

"""

return H.numItems/len(H.item)

def empty(H):

"""

Counts the number of empty buckets

Returns percentage of empty buckets in hash table

"""

e = 0

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

if len(H.item[i]) == 0:

e += 1

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

def standDev(H):

"""

Calculates standard deviation of the length of lists

"""

lengths = []

for b in H.item:

lengths.append(len(b))

return np.std(lengths)

BST.py

# Code to implement a binary search tree

# Programmed by Olac Fuentes

# Last modified February 27, 2019

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[0] > newItem[0]:

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

else:

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

return T

def InOrder(T):

# Prints items in BST in ascending order

if T is not None:

InOrder(T.left)

print(T.item[0],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[0])

InOrderD(T.left,space+' ')

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.item[1]

if T.item[0] < k:

return Find(T.right,k)

return Find(T.left,k)

def NumNodes(T):

"""

Counts number of nodes

"""

if T is None:

return 0

else:

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

def Height(T):

"""

Finds height of tree

"""

if T is None:

return 0

else:

hLeft = Height(T.left)

hRight = Height(T.right)

if hLeft > hRight:

return hLeft + 1

else:

return hRight + 1