David Amparan

Instructor: Dr. Fuentes, Olac

T.A: Anindita Nath, Maliheh Zargaran

Lab 5

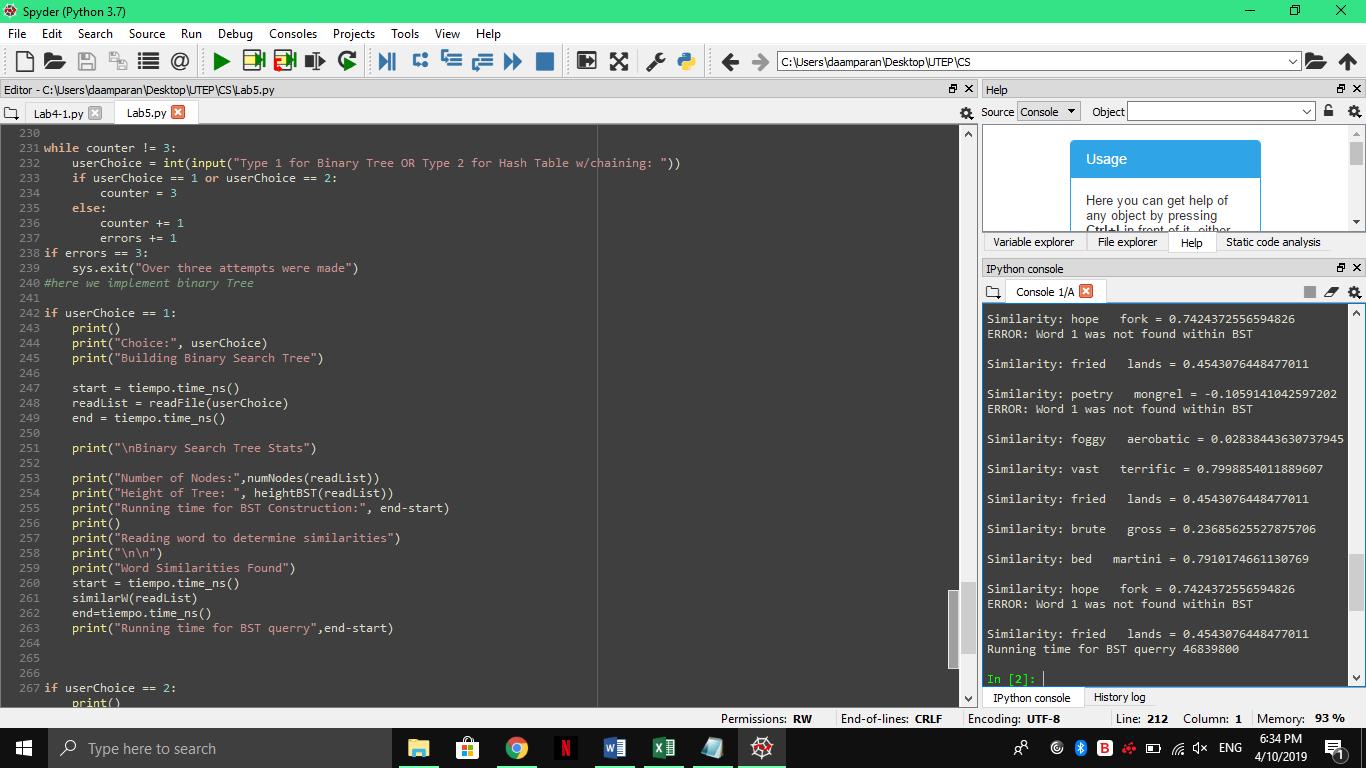
**Introduction**

For our fifth laboratory assignment, we were responsible for utilizing binary search trees as well as hash tables to implement a work of Natural Processing Language. NPL is a sub-field AI and in short these algorithms, methods, and systems understand human language and writing. Such as Alexa and Siri or any other AI assistant. By using the cosine distance between two words we can analyze how similar these two words in turn are.

Moving on, for the lab we must read .txt file which contains words along with 50 numbers, these are the embedding’s of the corresponding word. After reading the file we must construct a hash table or binary search tree with this information, read a separate file with pairs of words, find those in the data type we use, and finally calculate the cosine difference or similarities.

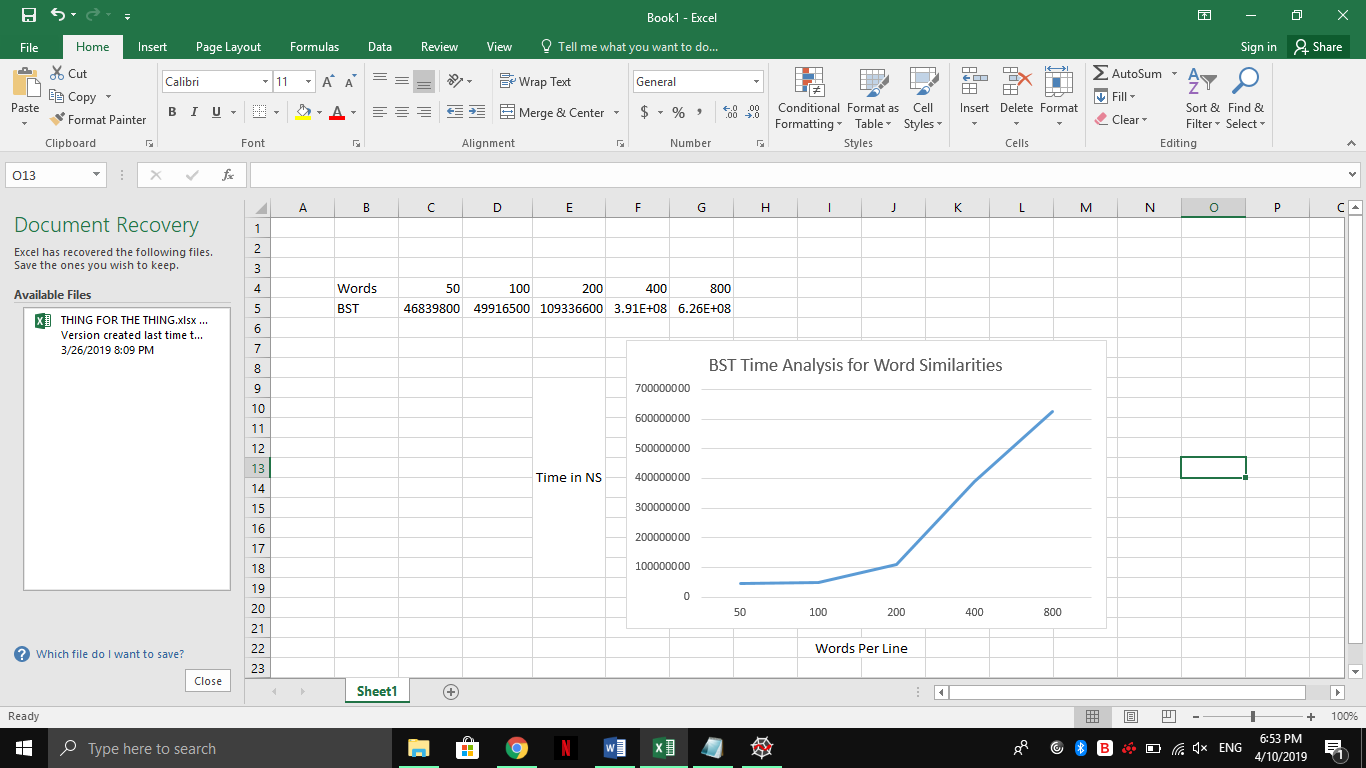
**Proposed Solutions**

First we will ask the user to choose which ADT to use, either the hash table or the binary search tree. After selecting the build, we will send this choice to a method which will build our data type by reading the file, splitting it, parsing (if needed), and finally inserting the data. After attaining our data type we will use it in a separate method, which will read the second file and find the pair of words in the second file within the data type. If this word is found, we then return the embedding of that word and calculate the cosine difference and print the results. Finally, we compare both data types with one another to understand their behavior and compare time complexity.

Each step was broken down into separate pieces, each with its own method to avoid confusion and improve legibility within the “main”. I also created two user defined types for the sake of improving readability, these two being wordPair and wordEmbed. wordEmbed contains the word and embedded array from the first file that was read while word pair simply contains the pair of words from the second file read.

**Experimental**

Due to not being able to test the BST and hash table head to head, I will only analyze the running time of the BST. The time is taken in nanoseconds and the file will increase in the same amount of words each time.



**Conclusion**

In conclusion, this lab showed me the benefits from both ADT’s from a design stand point only since I could not make the hash function work correctly. In theory, a hash table has a constant running time for both inserting and finding an object within it. While I could not successfully build my hash table I did construct the BST and this even with its log(n) time complexity took a while due to the sheer size of the file that was read. Searching within this ADT was also a bit time consuming since there were cases in which the entire tree was traversed. Therefore, a hash table in a much preferred ADT although it was complex to build.

**Appendix**

Source Code:

|  |
| --- |
| # -\*- coding: utf-8 -\*- |
|  | """ |
|  | Author: David Amparan |
|  | Last Date Modified: March 28, 2019 |
|  | TA: |
|  | Instructor: Fuentes, Olac |
|  | Purpose: |
|  | """ |
|  | import numpy as np |
|  | import sys |
|  | import time as tiempo |
|  | import math |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | class wordEmbed(object): |
|  | def \_\_init\_\_(self, word, arr): |
|  | self.word = word |
|  | self.arr = arr |
|  |  |
|  | class wordPair(object): |
|  | def \_\_init\_\_(self,w0,w1): |
|  | self.w0=w0 |
|  | self.w1=w1 |
|  |  |
|  |  |
|  | #ADT Implementations |
|  | class HashTableC(object): |
|  | # Builds a hash table of size 'size' |
|  | # Item is a list of (initially empty) lists |
|  | # Constructor |
|  | def \_\_init\_\_(self,size, num\_items=0): |
|  | self.item = [] |
|  | for i in range(size): |
|  | self.item.append([]) |
|  | self.num\_items = num\_items |
|  |  |
|  | def InsertC(H,data): |
|  | currentItems = H.num\_items |
|  | if currentItems == len(H.item): |
|  | H = doubleHash(H) |
|  | # Inserts k in appropriate bucket (list) |
|  | # Does nothing if k is already in the table |
|  | b = h(data.word , len(H.item)) |
|  | H.item[b].append(data) |
|  | H.num\_items+=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 b, i, H.item[b][i][1] |
|  | return b, -1, -1 |
|  |  |
|  | def h(s,n): |
|  | r = 0 |
|  | c = s[-1] |
|  | r = (r\*255 + ord(c))% n |
|  | return r |
|  |  |
|  |  |
|  | #Binary Tree |
|  | class BST(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self, item, left=None, right=None): |
|  | self.item = item |
|  | self.left = left |
|  | self.right = right |
|  |  |
|  | def InsertTree(T,newItem): |
|  | if T == None: |
|  | T = BST(newItem) |
|  | elif T.item[0] > newItem[0]: |
|  | T.left = InsertTree(T.left, newItem) |
|  | else: |
|  | T.right = InsertTree(T.right, newItem) |
|  | return T |
|  |  |
|  | def findItem(T,k): |
|  | if T is None: |
|  | return None |
|  | if T.item[0]==k: |
|  | return T.item[1] |
|  | if T.item[0]>k: |
|  | return findItem(T.left,k) |
|  | else: |
|  | return findItem(T.right,k) |
|  |  |
|  | """ |
|  | Method Name: heightBST | Parameters: T |
|  | Functionality: Will calculate the height of the tree |
|  | """ |
|  | def heightBST(T): |
|  | if T is None: |
|  | return 0 |
|  | h1 = 1 + heightBST(T.left) |
|  | h2 = 1 + heightBST(T.right) |
|  |  |
|  | if h1>h2: |
|  | return h1 |
|  | return h2 |
|  |  |
|  |  |
|  | """ |
|  | Method Name: numNodes | Parameter: T |
|  | Functionality: Will traverse the tree and keep count and return the count |
|  | this count will then represent the number of nodes found across the tree |
|  | """ |
|  | def numNodes(T): |
|  | if T is None: |
|  | return 0 |
|  | return 1 + numNodes(T.left) + numNodes(T.right) |
|  |  |
|  |  |
|  | """ |
|  | Method Name: doubleHash | Parameters: Hash |
|  | Functionality: The double hash function will double the size of the given hash |
|  | and insert the items as well |
|  | """ |
|  | def doubleHash(H): |
|  | newSize = len(H.item)\*2 |
|  | newHash = HashTableC(newSize+1) |
|  | #now we must append the new hash |
|  | for i in range(len(H.item)): |
|  | for k in range(len(H.item[i])): |
|  | b=h(H.item[i][k].word, len(H.item[i][k].word)) |
|  | H.item[b].append(H.item[i][k]) |
|  | newHash.num\_items = H.num\_items |
|  | return newHash |
|  |  |
|  |  |
|  | """ |
|  | Method Name: checkLoad Parameters: H |
|  | Functionality: The method check load simply checks the load factor |
|  | within a hash table and returns that value |
|  | """ |
|  | def checkLoad(H): |
|  | count = 0 |
|  | for i in H: |
|  | if i is not None: |
|  | count+=1 |
|  | return count/len(H) |
|  |  |
|  |  |
|  | """ |
|  | Method Name: readFile | Parameters: build |
|  | Functionality: e will read a file and return a list of of strings to then be able |
|  | to create the BST from or the hash table depending on the value of build |
|  |  |
|  | """ |
|  | def readFile(build): |
|  | f = open('glove.6B.50d.txt', encoding='utf-8') |
|  | #here we must parse our line |
|  | dataType = None |
|  |  |
|  | if build == 1: #we create a bst |
|  | for line in f: |
|  | fLine=line.split(" ") |
|  | #here we add the items of the line as we go onto the BST |
|  | if fLine[0].isalpha(): |
|  | dataType = InsertTree(dataType,[fLine[0], np.array(fLine[1:], dtype=float)]) |
|  |  |
|  | #same approach but with HASH |
|  | else: |
|  | size=19 |
|  | dataType = HashTableC(size) |
|  | print("Initial Table Size", size) |
|  | for line in f: |
|  | fLine=line.split(" ") |
|  | #here we add onto our hashtable |
|  | toInsert = None |
|  | toInsert = wordEmbed(fLine[0], np.array(fLine[1:], dtype=float)) |
|  | InsertC(dataType, toInsert) |
|  |  |
|  | f.close() |
|  | return dataType |
|  |  |
|  |  |
|  | """ |
|  | Method Name: similarW | Parameters: Reference, either BST or HASH |
|  | Functionality: Will read the file and determine the pairs, after reading the pairs |
|  | we will search for them within the ADT and calculate its cosine distance |
|  | """ |
|  | def similarW(dataType): |
|  | f=open('2Words.txt', encoding='utf-8') |
|  | #here we wanna read and compare |
|  | for line in f: |
|  | readL=line.split(' ') |
|  | dot=0 |
|  | mag0=0 |
|  | mag1=0 |
|  |  |
|  | pairs = wordPair(readL[0],readL[1]) |
|  | w0Embed=findItem(dataType, pairs.w0) |
|  | w1Embed=findItem(dataType,pairs.w1) |
|  |  |
|  | if w0Embed is None: #K was not found within the BST |
|  | print("ERROR: Word 0 was not found within the BST") |
|  | elif w1Embed is None: #k was not found within the BST |
|  | print("ERROR: Word 1 was not found within BST") |
|  | else: |
|  | #both words must have been found now we can compute the stats |
|  | for i in range(len(w0Embed)): |
|  | dot=dot+(w0Embed[i]\*w1Embed[i]) |
|  | mag0=mag0+math.pow(w0Embed[i],2) |
|  | mag1=mag1+math.pow(w1Embed[i],2) |
|  | totalMag=math.sqrt(mag0+mag1) |
|  | print("\nSimilarity:",pairs.w0," ",pairs.w1,"=",dot/totalMag) |
|  |  |
|  |  |
|  |  |
|  |  |
|  | #User Instructions |
|  | print("-------------------OVERVIEW---------------------") |
|  | print("Natural Processing Language is a sub-field of AI") |
|  | print("which can understand written and spoken language") |
|  | print("This program here is a small sample of that field") |
|  | print("-------------------------------------------------") |
|  | print() |
|  |  |
|  | print("Choose implementation") |
|  | #allow the user to pick the correct choice total of 3 times |
|  | userChoice = 0 |
|  | counter = 0 |
|  | errors = 0 |
|  |  |
|  | while counter != 3: |
|  | userChoice = int(input("Type 1 for Binary Tree OR Type 2 for Hash Table w/chaining: ")) |
|  | if userChoice == 1 or userChoice == 2: |
|  | counter = 3 |
|  | else: |
|  | counter += 1 |
|  | errors += 1 |
|  | if errors == 3: |
|  | sys.exit("Over three attempts were made") |
|  | #here we implement binary Tree |
|  |  |
|  | if userChoice == 1: |
|  | print() |
|  | print("Choice:", userChoice) |
|  | print("Building Binary Search Tree") |
|  |  |
|  | start = tiempo.time\_ns() |
|  | readList = readFile(userChoice) |
|  | end = tiempo.time\_ns() |
|  |  |
|  | print("\nBinary Search Tree Stats") |
|  |  |
|  | print("Number of Nodes:",numNodes(readList)) |
|  | print("Height of Tree: ", heightBST(readList)) |
|  | print("Running time for BST Construction:", end-start) |
|  | print() |
|  | print("Reading word to determine similarities") |
|  | print("\n\n") |
|  | print("Word Similarities Found") |
|  | start = tiempo.time\_ns() |
|  | similarW(readList) |
|  | end=tiempo.time\_ns() |
|  | print("Running time for BST querry",end-start) |
|  |  |
|  |  |
|  |  |
|  | if userChoice == 2: |
|  | print() |
|  | print("Choice:", userChoice) |
|  | print("Building Hash Table w/Chaining") |
|  | print("\nHash Table Stats") |
|  | print() |
|  |  |
|  | start = tiempo.time\_ns() |
|  | readList = readFile(userChoice) |
|  | end = tiempo.time\_ns() |

Academic Honesty Statement

I, David Amparan, guarantee this source code was done by myself with no copying and/or online sources. Therefore I take full responsibility if any act of academic dishonesty is found and accused of.

David Amparan April 4th 2019