**Author:**  *Nguyen, Abram*

**Assignment:** *Lab 5 Report*

**Course:** *CS 2302 - Data Structures*

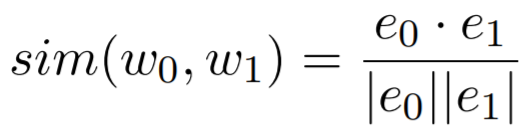
**Instructor:**  *Fuentes, Olac*

**T.A.:**  *Nath, Anindita*

Introduction:

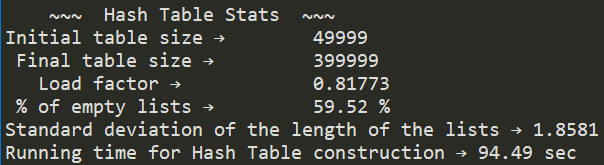
In this lab, I’m implementing and demonstrating the real life applications of hash tables. I have a hash table class that solves collisions through the method of chaining. Every element of the hash table should contain a list of size 2. I also have a binary search tree class that should hold a list of size 2 in each of its nodes. The lists of size 2 contain a word and its embedding. I’ll be comparing the running times between the binary search tree implementation and the hash table with chaining implementation. Both structures should hold the exact same information for the test run.

Proposed solution design and implementation:

1. For my hash table that solves collisions by chaining:
   1. I started out with a large hash table of size 49,999. I knew that I’d end up with less than 400,000 items in my final hash table, but I didn’t want to start with a large hash table. I think the biggest flaw with this set-up is that once my table reaches about 49,999 items, my algorithm will have to re-hash another table with a size of 99,999. Then again, when it reaches that number, it will do the same with a new table of size 199,999, then another table of size 399,999. I made a while loop that would keep re-hashing my values as long as it came out with a load factor of 1 or more. The load factor is computed by dividing the number of items in the list by the length of the hash table.
   2. After initializing my hash table, I have to take what’s in the given text file ‘glove.6b.50d.txt’ and place every word and array of values into a single list of size 2. For each line of the file, I use the ‘split()’ method to automatically place every item into a simple array. This array that I call ‘temp’ now contains a word as its first element and 50 numbers after that, which is the word’s embedding. I check to make sure that the first element of ‘temp’, the ‘word’, actually contains letters and isn’t just a symbol like a comma or period. If it’s a word, I take every element of ‘temp’, except the first ‘word’ and create a numpy array of float values. I use the given ‘InsertC’ function to insert the first element of ‘temp’, the word, and the numpy array of numbers, the embedding.
   3. The method that decides which bucket in the hash table a word is inserted into was a given method, but I ended up editing it a little. This method would actually only consider the last character in a string. So every string, for example, that ended in ‘y’ would be placed into one bucket. This would be the same for all other words that end in the same letter as well. The resulting hash table would end up only containing 26 buckets that contain any items. I ended up changing the equation used to decide where a word would be placed. ‘K’ is a constant that I chose randomly at the time. I think a better substitute for ‘k’ might be the length of the string being considered. That constant, essentially, helps to ‘spread out’ the words into more buckets so that I end up with significantly less empty buckets. This will significantly decrease the time it takes to find a word and its embedding in the hash table.
      1. r = (r\*n + ord(c))% n → r = (r\*k + ord(c))% n
2. For my binary search tree version:
   1. Because I’d been working with a tree and not a hash table, I didn’t have to waste a lot of space. The number of nodes needed to fit every word and embedding pair would be no more than necessary and no less than necessary. If I had 400,001 lists to insert as nodes, the number of nodes in the BST would be 400,001. I didn’t need to implement a load factor or calculate the number empty nodes because of this advantage.
   2. My first objective, like the hash table implementation, was to convert the file ‘glove.6b.50d.txt’ into a list of size 2. This list would contain a word and it’s embedding. The embedding is a numpy array of float values. My algorithm for this part is exactly the same as it is in my hash table implementation. I read each line of the file and store the line’s word as well as the embedding after it as a numpy array. I had a little trouble inserting the nodes of my binary search tree, however. The tree seemed to print out ok, but I couldn’t get the results of the similarities between words using the BST.
3. Calculating the similarity of two words using their embeddings:
   1. To calculate the similarity between two words, take the dot product of both their embeddings and divide them by the product of the embeddings’ magnitudes. This is shown to the right as the function ‘sim(w0, w1)’.
   2. To find the dot product of e0 and e1, I multiplied each corresponding element of the numpy array, then added those products. The first element of e0 is multiplied by the first element of e1, then e0 and e1’s second element, and so on. All those products are summed up to the dot product of e0 and e1. Equation shown to the right.
   3. The magnitude of each embedding is the square root of the sum of the squares of each element in the array. The first step is to square each element, then sum up all these values, and return the square root of that sum. Equation shown to the right.

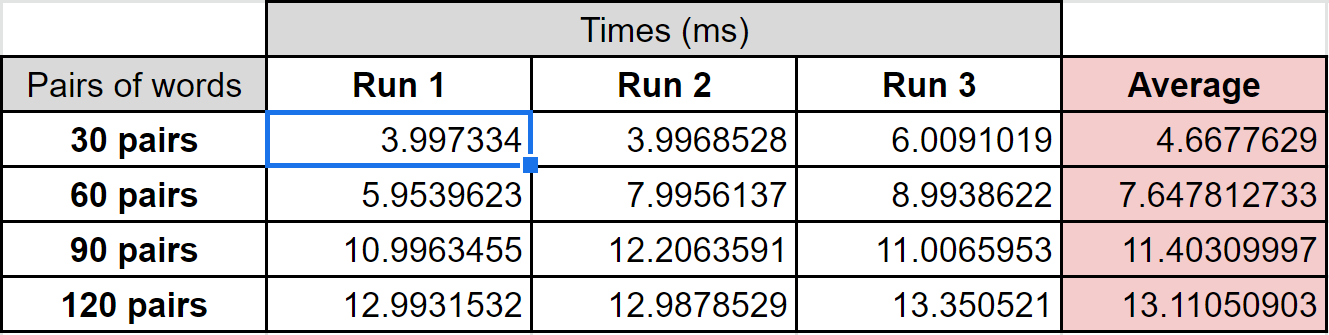
The main differences in calculating the embeddings in a hash table versus a binary tree are the run time and the method used to find those embeddings.

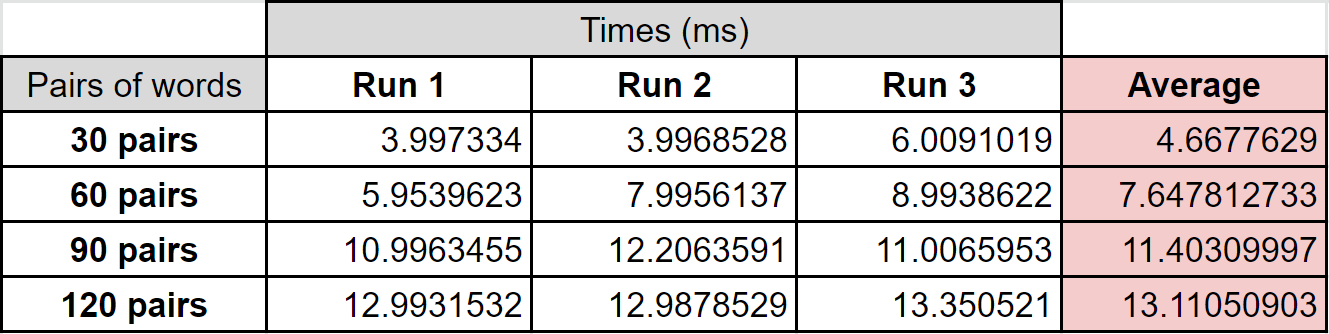
Experimental results:

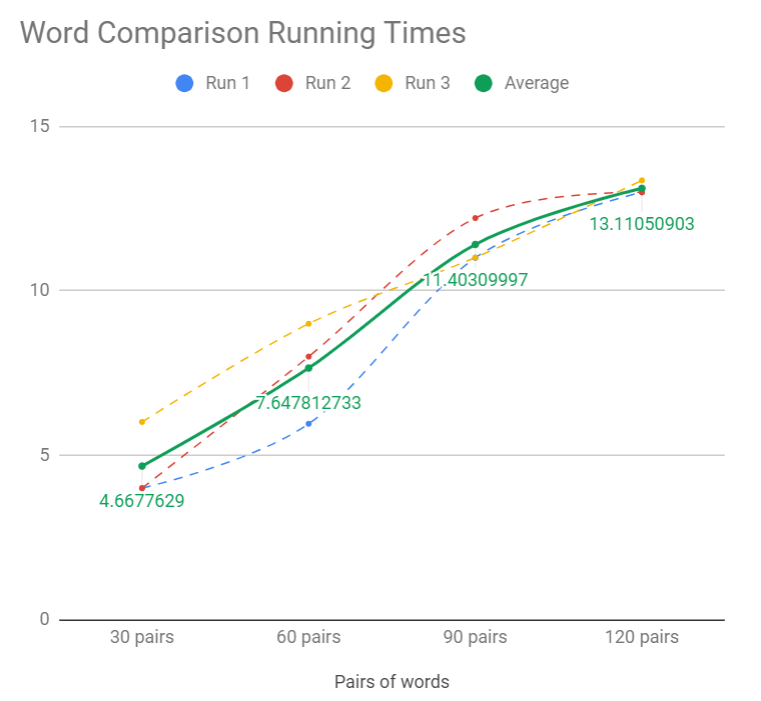
1. The average time it took to build a hash table came out to 92.398 seconds. These are the statistics of my hash table after the data from ‘glove.6b.50d.txt’ is inserted.

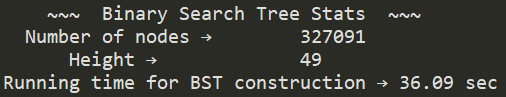
These are the **120 pairs of words** I used and their **results**:

|  |  |  |  |
| --- | --- | --- | --- |
| ['spain', 'spain'] → 1.0  ['spain', 'mexico'] → 0.75138  ['utep', 'yale'] → 0.18837  ['utep', 'harvard'] → 0.06843  ['ant', 'insect'] → 0.60197  ['game', 'fun'] → 0.6006  ['game', 'baseball'] → 0.75207  ['game', 'soccer'] → 0.64863  ['fun', 'baseball'] → 0.49233  ['fun', 'soccer'] → 0.42259  ['life', 'death'] → 0.72641  ['death', 'life'] → 0.72641  ['spider', 'fly'] → 0.27461  ['spider', 'man'] → 0.38416  ['rabbit', 'easter'] → 0.35356  ['dungeons', 'dragons'] → 0.74666  ['caves', 'carnivores'] → 0.27887  ['play', 'fun'] → 0.67225  ['abe', 'aria'] → -0.14498  ['test', 'quiz'] → 0.35592  ['man', 'woman'] → 0.88603  ['king', 'queen'] → 0.7839  ['mind', 'fang'] → 0.02399  ['crab', 'cockroach'] → 0.46241  ['lobster', 'cockroach'] → 0.47494  ['crab', 'spider'] → 0.54338  ['lobster', 'spider'] → 0.48443  ['obama', 'mama'] → 0.04372  ['fan', 'summer'] → 0.49679  ['fan', 'wind'] → 0.39521 | ['rice', 'fish'] → 0.54538  ['rice', 'meat'] → 0.64067  ['fish', 'meat'] → 0.79954  ['number', 'variable'] → 0.4505  ['wheel', 'circle'] → 0.48879  ['box', 'cube'] → 0.40577  ['giant', 'large'] → 0.61624  ['big', 'large'] → 0.6746  ['big', 'giant'] → 0.60982  ['small', 'large'] → 0.96846  ['small', 'big'] → 0.70084  ['small', 'giant'] → 0.61624  ['mother', 'father'] → 0.8909  ['easter', 'bunny'] → 0.38423  ['together', 'forever'] → 0.53017  ['time', 'space'] → 0.62761  ['program', 'code'] → 0.51266  ['twenty', 'two'] → 0.76009  ['pizza', 'pasta'] → 0.73526  ['pizza', 'dent'] → 0.02103  ['pizza', 'teeth'] → 0.18881  ['food', 'teeth'] → 0.31314  ['brush', 'teeth'] → 0.57616  ['zipper', 'jacket'] → 0.44195  ['cloth', 'clothing'] → 0.7221  ['flip', 'flop'] → 0.61829  ['ping', 'pong'] → 0.70016  ['friend', 'family'] → 0.7233  ['hair', 'hairs'] → 0.65853  ['hair', 'head'] → 0.52048 | ['synonym', 'antonym'] → 0.49979  ['place', 'location'] → 0.67433  ['google', 'facebook'] → 0.8087  ['google', 'youtube'] → 0.68407  ['xylophone', 'instrument'] → 0.48049  ['piano', 'instrument'] → 0.68017  ['guitar', 'instrument'] → 0.63524  ['trumpet', 'instrument'] → 0.63319  ['drum', 'instrument'] → 0.66089  ['rubber', 'band'] → 0.34126  ['cold', 'flu'] → 0.44598  ['cold', 'hot'] → 0.80105  ['fly', 'plane'] → 0.72628  ['drive', 'car'] → 0.69624  ['fly', 'fall'] → 0.38097  ['bend', 'snap'] → 0.22778  ['push', 'pop'] → 0.31114  ['jazz', 'funk'] → 0.71582  ['red', 'orange'] → 0.80429  ['red', 'yellow'] → 0.89955  ['red', 'green'] → 0.85619  ['red', 'blue'] → 0.89017  ['red', 'purple'] → 0.83232  ['music', 'band'] → 0.78066  ['nirvana', 'music'] → 0.4344  ['nirvana', 'religion'] → 0.19424  ['lonely', 'person'] → 0.54608  ['lonely', 'people'] → 0.42589  ['flag', 'country'] → 0.43455  ['flag', 'color'] → 0.46016 | ['destroy', 'halibut'] → -0.10126  ['fish', 'halibut'] → 0.60445  ['cone', 'ice'] → 0.45041  ['flame', 'ice'] → 0.52691  ['ball', 'game'] → 0.71063  ['ball', 'soccer'] → 0.38113  ['ball', 'ball'] → 1.0  ['flower', 'flour'] → 0.36023  ['zebra', 'animal'] → 0.37368  ['meat', 'cook'] → 0.61508  ['cook', 'cooking'] → 0.69573  ['hundred', 'thousand'] → 0.93101  ['box', 'boxes'] → 0.7835  ['box', 'boxing'] → 0.26734  ['bend', 'bent'] → 0.33395  ['destroy', 'destroyed'] → 0.66754  ['destroy', 'destroys'] → 0.71229  ['find', 'found'] → 0.71269  ['find', 'founded'] → 0.22422  ['found', 'founded'] → 0.36443  ['place', 'places'] → 0.82252  ['hide', 'hid'] → 0.74258  ['hit', 'hits'] → 0.87701  ['shot', 'shoot'] → 0.7674  ['kiss', 'kisses'] → 0.76317  ['kiss', 'kissed'] → 0.62184  ['kisses', 'kissed'] → 0.75739  ['run', 'ran'] → 0.82897  ['rub', 'rob'] → 0.01602  ['rib', 'rob'] → 0.03635 |







1. Not everything in my binary search tree implementation worked out the way I wanted it to, although it seemed easier to execute in my head. I mostly had trouble accessing the word and embeddings in the tree’s nodes. It was most likely just that, but the problem could have stemmed from the way I inserted each node. I didn’t get any data pertaining to similarities between words. I only included some ‘preliminary’ results relating to building my BST.

Conclusion:

Like most other algorithms, I have the choice of sacrificing the use of space for faster running time and vice versa. In the case of the binary search tree implementation, I sacrificed running time to save space. In the hash table implementation, I used excess space to come out with faster running time. Although I had trouble coming up with definite results from my binary search tree implementation, I understand that the hash table is faster when looking up values. A hash table, in theory, should take a constant O(1) time to find a value. A binary search tree takes O(log(n)) time to find a value.

Appendix:

|  |  |
| --- | --- |
| """ |  |
|  | Author: Nguyen, Abram |
|  | Assignment: Lab 5 |
|  | Course: CS 2302 - Data Structures |
|  | Instructor: Fuentes, Olac |
|  | T.A.: Nath, Anindita |
|  | Last modified: April 2, 2019 |
|  |  |
|  | Purpose of program: The purpose of this program is to demonstrate the real life |
|  | applications of hash tables and compare the running times |
|  | of hash tables and binary search trees given the same |
|  | information. |
|  | """ |
|  | import numpy as np |
|  | import math |
|  | import time |
|  | ############################################################################### |
|  | # BST |
|  | class BST(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self, item, left=None, right=None): |
|  | self.num\_items = 0 |
|  | self.item = item |
|  | self.left = left |
|  | self.right = right |
|  |  |
|  | def Insert(T,newItem): |
|  | if T == None: |
|  | T = BST(newItem[0]) |
|  | elif T.item > newItem[0]: |
|  | T.left = Insert(T.left,newItem) |
|  | else: |
|  | T.right = Insert(T.right,newItem) |
|  | T.num\_items += 1 |
|  | 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) |
|  | T.num\_items -= 1 |
|  | 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 == k: |
|  | return T |
|  | if T.item<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') |
|  |  |
|  | ############################################################################### |
|  | # Implementation of hash tables with chaining using strings |
|  |  |
|  | 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.num\_items += 1 |
|  | H.item[b] == None |
|  | 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 H.item[b][i][1] |
|  | return None |
|  |  |
|  | #r = (r\*k + ord(c))% n |
|  | def h(s,n): |
|  | r = 0 |
|  | for c in s: |
|  | #r = (r\*k + ord(c))% n |
|  | r = (r\*4 + ord(c))% n |
|  | return r |
|  |  |
|  | """ |
|  | ################################ |
|  | ################################ |
|  | ################################ |
|  | """ |
|  | ############################################################################### |
|  |  |
|  | def HashTableBST(T, filename): |
|  | f = open(filename, encoding="utf-8") |
|  | #for each line in the file: |
|  | for line in f: |
|  | str(line) |
|  | #create array of items including 'words' and values |
|  | temp = line.split(" ") |
|  | #ignore all non 'words' |
|  | if temp[0].isalpha(): |
|  | #create array of numbers from array of items |
|  | nums = np.array([]) |
|  | nums = np.append(nums, temp[1:]).astype(float) |
|  | #insert word and embedding into binary search tree |
|  | Insert(T, [temp[0], nums]) |
|  | #InOrder(T) |
|  | #InOrder(T, '') |
|  |  |
|  | # Find the similarity between two words, return running time in ms |
|  | def SimilaritiesBST(T): |
|  | print("Reading word file to determine similarities...\n") |
|  | file = open('pairs.txt') |
|  | c = 0 |
|  | start = time.time() |
|  | for line in file: #per line: |
|  | temp = line.split() |
|  | e0 = FindC(T,temp[0]) #embedding of first word |
|  | e1 = FindC(H,temp[1]) #embedding of second word |
|  | dot = np.sum(e0\*e1,dtype=float) #compute dot product of e0 and e1 |
|  | mag = math.sqrt(np.sum(e0\*e0,dtype=float))\*math.sqrt(np.sum(e1\*e1,dtype=float)) |
|  | print("Similarity",temp,"→",round(dot/mag, 5)) |
|  | c+=1 |
|  | end = time.time() |
|  | print("\nTime to compare", c, "pair(s) of words →", round((end-start)\*1000, 5), "ms") |
|  |  |
|  | # Return the number of nodes in tree 'T' |
|  | def Count(T): |
|  | c = 1 |
|  | if T.left != None: |
|  | c += Count(T.left) |
|  | if T.right != None: |
|  | c += Count(T.right) |
|  | return c |
|  |  |
|  | # Return the height of tree 'T' |
|  | def Height(T): |
|  | if T == None: |
|  | return 0 |
|  | L = Height(T.left) #left height |
|  | R = Height(T.right) #right height |
|  | if L < R: |
|  | return R+1 |
|  | else: |
|  | return L+1 |
|  | ############################################################################### |
|  |  |
|  | def HashTableChaining(H, filename): |
|  | #f = open('glove.6B.50d.txt', encoding="utf-8") |
|  | f = open(filename, encoding="utf-8") |
|  | #for each line in the file: |
|  | for line in f: |
|  | str(line) |
|  | #create array of items including 'words' and values |
|  | temp = line.split(" ") |
|  | #ignore all non 'words' |
|  | if temp[0].isalpha(): |
|  | #create array of numbers from array of items |
|  | nums = np.array([]) |
|  | nums = np.append(nums, temp[1:]).astype(float) |
|  | #insert word and embedding into hash table |
|  | InsertC(H, temp[0], nums) |
|  |  |
|  | # Compute the load factor of hash table 'H' |
|  | def LoadFactorChain(H): |
|  | #number of items / length of table |
|  | return float(H.num\_items/len(H.item)) |
|  |  |
|  | # Compute the percentage of empty lists in hash table 'H' |
|  | def EmptyPercent(H): |
|  | c = 0 |
|  | for i in range(len(H.item)): |
|  | #count number of empty lists |
|  | if len(H.item[i]) == 0: |
|  | c += 1 |
|  | #compute percentage of empty lists in hash table 'H' |
|  | return c/len(H.item)\*100 |
|  |  |
|  | # Compute the standard deviation of the lengths of the lists in hash table 'H' |
|  | def HashSD(H): |
|  | lengths = [] |
|  | for i in range(len(H.item)): |
|  | #record every list's length |
|  | lengths.append(len(H.item[i])) |
|  | #compute using numpy.std |
|  | return np.std(lengths) |
|  |  |
|  | # Find the similarity between two words, return running time in ms |
|  | def SimilaritiesHash(H): |
|  | print("Reading word file to determine similarities...\n") |
|  | print("\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ ") |
|  | file = open('pairs.txt') |
|  | c = 0 |
|  | start = time.time() |
|  | for line in file: #per line: |
|  | temp = line.split() |
|  | e0 = FindC(H,temp[0]) #embedding of first word |
|  | e1 = FindC(H,temp[1]) #embedding of second word |
|  | dot = np.sum(e0\*e1,dtype=float) #compute dot product of e0 and e1 |
|  | mag = math.sqrt(np.sum(e0\*e0,dtype=float))\*math.sqrt(np.sum(e1\*e1,dtype=float)) |
|  | print("Similarity",temp,"→",round(dot/mag, 5)) |
|  | c+=1 |
|  | end = time.time() |
|  | print("\nTime to compare", c, "pair(s) of words →", round((end-start)\*1000, 5), "ms") |
|  |  |
|  |  |
|  | """ |
|  | ################################ |
|  | ######### METHOD CALLS ######### |
|  | ################################ |
|  | """ |
|  |  |
|  | p = input("Please choose preferred implementation (enter number):\n[1] Binary search tree\n[2] Hash table w/ chaining\nChoice: ") |
|  | print() |
|  | filename = 'glove.6B.50d.txt' |
|  |  |
|  | if p == '1': #store values into BST |
|  | T = BST("") |
|  | print("Building binary search tree...\n") |
|  | start = time.time() |
|  | HashTableBST(T, filename) |
|  | end = time.time() |
|  | print(" ~~~ Binary Search Tree Stats ~~~ ") |
|  | print(" Number of nodes → ", Count(T)) |
|  | print(" Height → ", Height(T)) |
|  | print("Running time for BST construction →", round(end-start, 2), "sec") |
|  | #SimilaritiesBST(T) DOESN'T WORK CORRECTLY |
|  |  |
|  | if p == '2': #store values into hash table w/ chaining |
|  | start = time.time() |
|  | #initialize hash table |
|  | H = HashTableC(49999) |
|  | print("Building hash table with chaining...\n") |
|  | print(" ~~~ Hash Table Stats ~~~ ") |
|  | print("Initial table size → ", len(H.item)) |
|  | #fill hash table |
|  | HashTableChaining(H, filename) |
|  | while LoadFactorChain(H) >= 1: |
|  | H = HashTableC(2\*len(H.item) + 1) |
|  | HashTableChaining(H, filename) |
|  | end = time.time() |
|  | print(" Final table size → ", len(H.item)) |
|  | print(" Load factor → ", round(LoadFactorChain(H), 5)) |
|  | print(" % of empty lists → ", round(EmptyPercent(H), 2), "%") |
|  | print("Standard deviation of the length of the lists →", round(HashSD(H), 5)) |
|  | print("Running time for Hash Table construction →", round(end-start, 2), "sec\n") |
|  | SimilaritiesHash(H) |
|  | ############################################################################### |
|  | # End of program |

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.

* Abram Nguyen