**LAB 5**

**DANIEL CRUZ**

**Introduction**

The purpose of this lab is to model a use case for hash tables. In this lab, we are implementing a hash table in order to store objects that contain a string and its embedding (a vector that can be used to determine how similar words are) for quick look up. This lab will involve comparing different hashing algorithms and how they compare in the runtime of both building and searching the data structure. Also, linear probing and chaining will also be compared for efficiencies in runtime.

**Proposed Solution Design and Implementation**

Both linear probing and chaining codes provided from the class website were used as a basis for developing the data structures. Both codes were restructured to allow for insertion of wordEmbedding and searching using strings. The wordEmbedding object itself was defined to allow for comparing for equality and less than with strings. In both hash table codes, I constructed 6 different versions of insertion and find functions. Each performs a different algorithm for finding the hash key has directed in the lab instructions.

For the 6th algorithm, I implemented an algorithm that focused on uniqueness of each word. For a given string, I made an algorithm that will bit shift the ASCII value of a character 8 bits to the left and OR it with the amount of characters from that character to the end of the string. I sought to focus on creating a unique hash key based on the characters in a string and the position they are in the string.

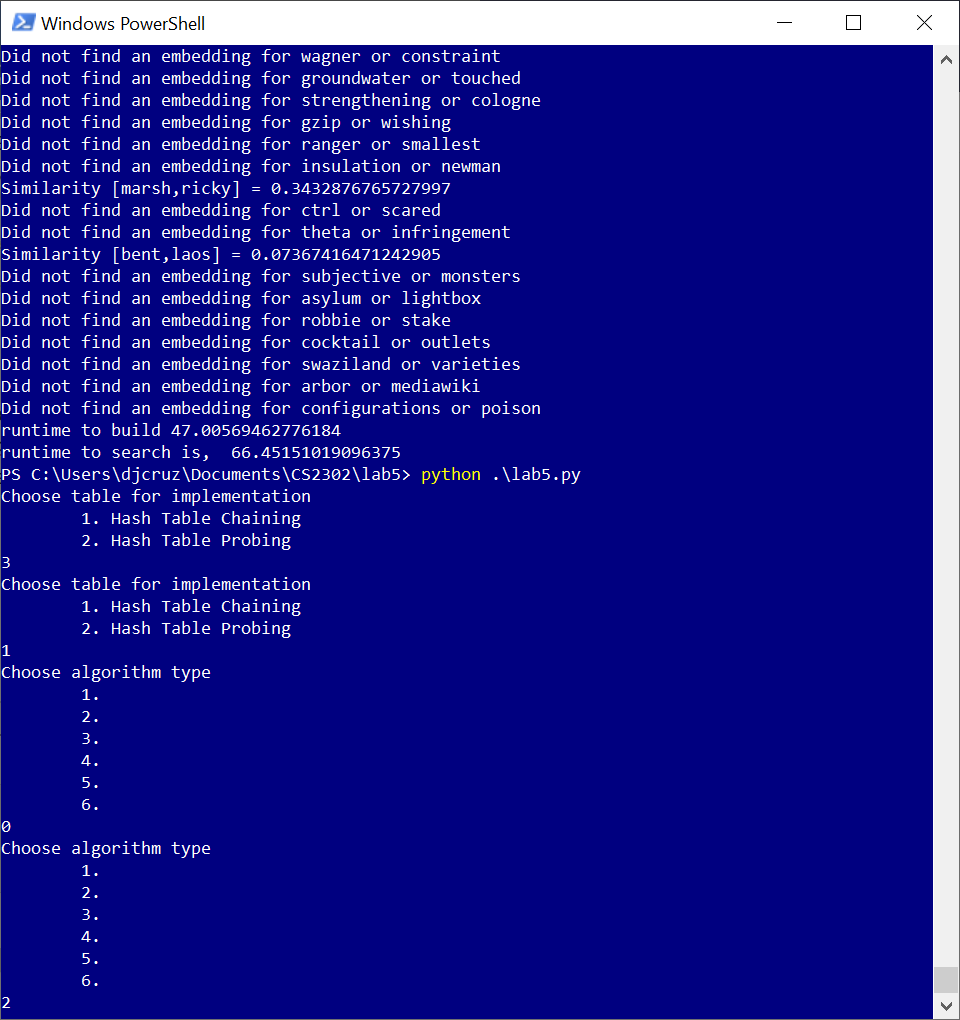
In the main program, I broke the functions up by purpose. The man function contains the main menu system that will ask for user input and calls functions that build the hash table implementation with the specified hash key algorithm specified by the user. It then calls a separate function that will search for words given from a file. Main will then calculate the similarities between the two words and prints this value or prints if it could not find a word in the hash table.

Both functions to create the hash tables are very similar save for the type of hash table created. In both functions, the hash table is initialized with a size of 400,009. The file with the word embeddings has a total of 400,000 words. 4000009 is the nearest prime number larger than this value. This is especially important to do for linear probing since the size of the table is fixed and must be large enough to hold all the words. Depending on the algorithm used, much of this space could be wasted when utilizing chaining. However, I chose to keep this static size to try achieve best optimization with each algorithm. With a size that can theoretically hold every word in a separate bucket, the only thing to slow down searches is the robustness of the key algorithms. An optional parameter can be used to specify how many words to build the hash table with. Depending on the algorithm chosen, the insert function for that algorithm will be called and the table will be built and returned along with its runtime.

The function that will search words specified in a file takes in a file name, the table and number representing the algorithm type. It will then read a line of the file. In each line, two words are expected separated by a space. These two words will be searched in the hash table based on the algorithm type. If the words are found, their WordEmbedding objects are pushed into a 2D list for calculating similarities. If the find function returns None, then that word was not found in the table and the strings themselves are put into the 2D list. Both the 2D list and the runtime of searching all the words is returned.

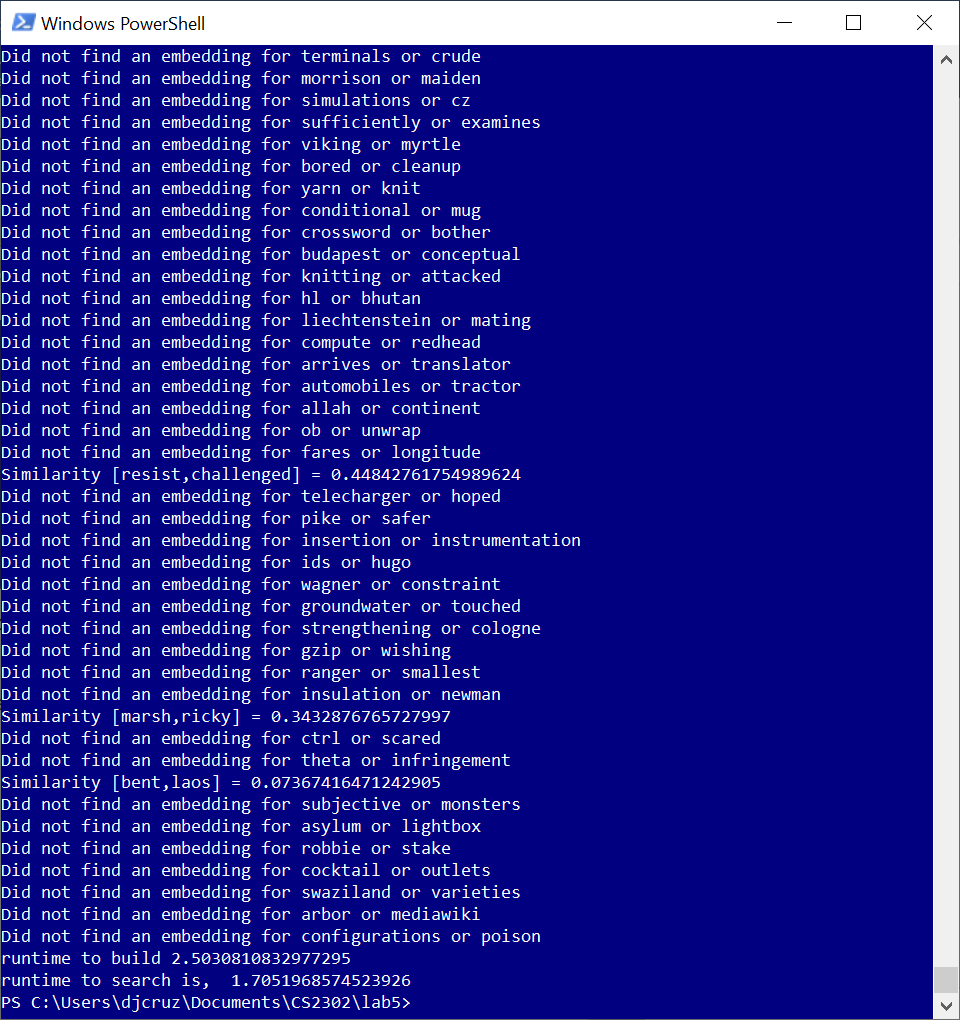
**Experimental Results**

The first test that will be done is testing the menu system. It will be tested to ensure values not in the ranges are not allowed.



The menu works in both allowing the user to pick which implementation to use and which algorithm type to use in the implementation.

Now a single run through of the program will be performed. This is using hash tables with chaining using algorithm 1:

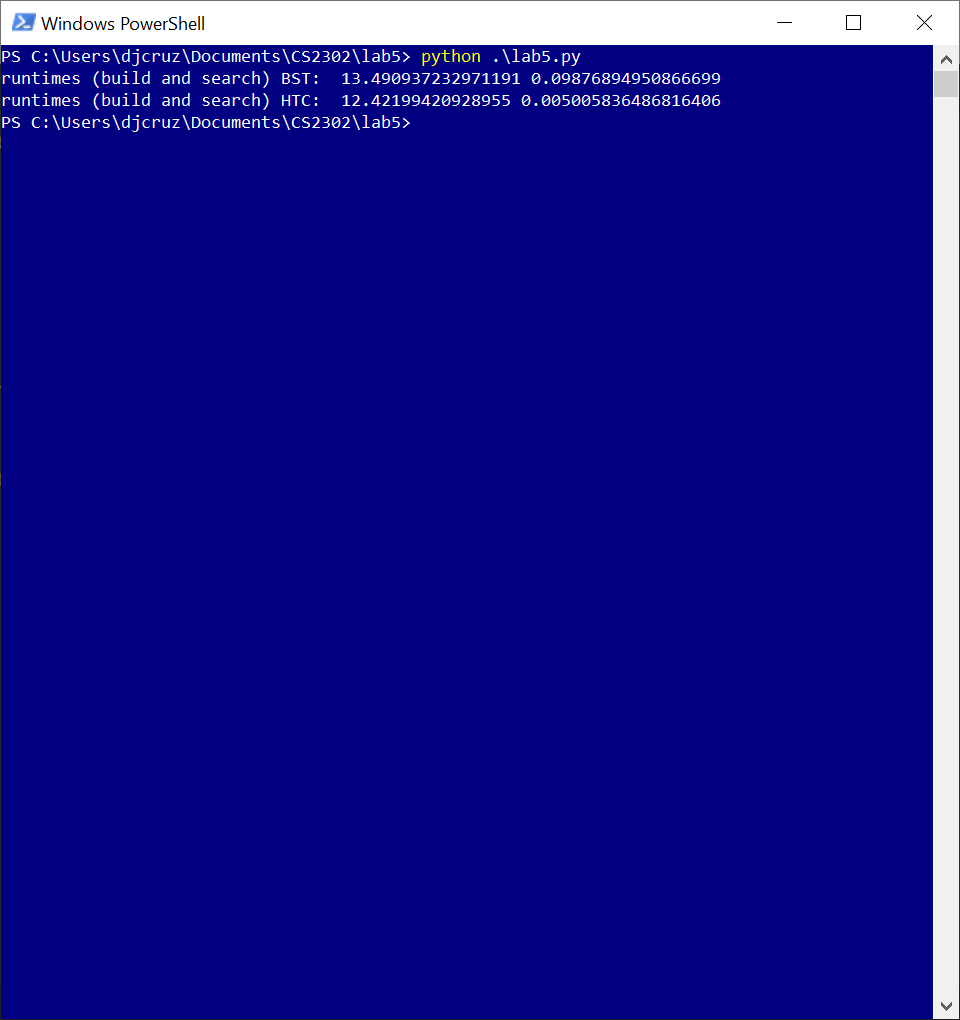


It runs to completion searching in a list of words, displaying the similarity coefficient, and then displaying the runtimes for both building and searching.

The next test will be to compare the runtimes for each for algorithm in both probing and chaining configurations. For this benchmark, I am creating a hash table with the first 10,000 of words in the glove file. I am not running the total 400,000 as some of the algorithms in linear probing take an upwards of hours to build. As such, the words to get similarities for has also changed. I am going to find similarities for the first 12,000 words in the glove file. I chose to search for slightly more words than what is being built with to take into account the runtime of searching for words not in the hash table. Below are the runtimes for building and searching:

The results show some interesting characteristics. Linear probing in most algorithms have a longer search time than build time. Difference between the worst runtime and best runtime are far greater than the chaining algorithms. The best algorithm is algorithm 5, followed closely by algorithm 6. Hash tables with chaining almost always outperformed probing. The search and built times from the worst algorithm are much smaller than those in linear probing. The runtimes for algorithm 5 in both methods are extremely close, yet chaining debatably comes out superior. I believe the penalty for searching for a word not in the hash table greatly lengthened the search runtimes in linear probing. This penalty will only increase the more filled the hash table is.

The most efficient hash table algorithm is algorithm 5 in a chaining configuration. This will be compared with that of a binary tree, best tree algorithm from the previous lab. I will build both data structures with all embedding words and search using 10,000-word list used in the previous lab for searching.



Building in both cases are very close, with hash tables slightly out performing binary search tree. However, in terms of searching, hash table is far superior in runtime.

**Conclusion**

Hash tables are only as good as its hash key algorithm. Given an ideal algorithm in which no conflicts occur, building and searching will have a runtime of O(1). However, with a hash key algorithm that contains conflicts, the worst-case runtime of O(n). Hash table with chaining has the ability to provide a less detrimental upper limit in its runtime as opposed to hash table with linear probing which might possibly have to search through every element inserted. Given a good hash key algorithm, hash tables can outperform binary search trees. This makes sense as hash tables approach O(1) and binary search trees approach O(ln(n)).

**Appendix**

from time import time

import numpy as np

from wordEmbedding import WordEmbedding

from HashTableChaining import HashTableChainWord

from HashTableProbing import HashTableLP

#wordLimit, amount of words to build from glove. -1 builds entire file

def buildHTChaining(algor = 1, wordLimit = -1):

try:

runTime = 0

#total size of glove file is 400,000 words. 400,009 is the next largest prime number.

HTC = HashTableChainWord(400009)

file = open("glove.6B.50d.txt", encoding="utf8")

lineCount = 0

for line in file:

dataList = line.split(" ") #first element is word, the rest is the float array

if dataList[0].isalpha(): #only insert if word starts with an alpha character

if algor == 1:

startTime = time()

HTC.insert1(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 2:

startTime = time()

HTC.insert2(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 3:

startTime = time()

HTC.insert3(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 4:

startTime = time()

HTC.insert4(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 5:

startTime = time()

HTC.insert5(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 6:

startTime = time()

HTC.insert6(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

lineCount += 1

if lineCount == wordLimit:

return HTC, runTime

print(lineCount,end = "\r")

return HTC,runTime

except Exception as e:

print(e)

HTC.print\_table()

raise e

#wordLimit, amount of words to build from glove. -1 builds entire file

def buildHTProbing(algor = 1, wordLimit = -1):

try:

runTime = 0

#total size of glove file is 400,000 words. 400,009 is the next largest prime number.

HTP = HashTableLP(400009)

file = open("glove.6B.50d.txt", encoding="utf8")

lineCount = 0

for line in file:

dataList = line.split(" ") #first element is word, the rest is the float array

if dataList[0].isalpha(): #only insert if word starts with an alpha character

if algor == 1:

startTime = time()

HTP.insert1(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 2:

startTime = time()

HTP.insert2(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 3:

startTime = time()

HTP.insert3(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 4:

startTime = time()

HTP.insert4(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 5:

startTime = time()

HTP.insert5(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

if algor == 6:

startTime = time()

HTP.insert6(WordEmbedding(dataList[0],dataList[1:]))

runTime += time()-startTime

lineCount += 1

if lineCount == wordLimit:

return HTP, runTime

print(lineCount,end = "\r")

return HTP,runTime

except Exception as e:

print(e)

HTP.print\_table()

raise e

#Does vector calculation to determine how similar words are

def similarities(embed1, embed2):

return np.dot(embed1.emb, embed2.emb)/(np.linalg.norm(embed1.emb) \* np.linalg.norm(embed2.emb))

#reads words from a file and returns a 2-d list of wordEmbedding and

def getEmbeddingsFromFile(T,fileName, algor = 1):

file = open(fileName)

runtime = 0

embeddingsList = []

for line in file:

#sliced to remove the newline character, then split by the comma

if "\n" in line:

line = line[:-1]

seachWords = line.split(",")

#determine algorithm to use

if algor == 1:

startTime = time()

output1 = T.find1(seachWords[0])

output2 = T.find1(seachWords[1])

if output1 is None or output2 is None: #if word is not in tree, just appends the string words

embeddingsList.append(seachWords)

else:

embeddingsList.append([output1, output2])

runtime = runtime + (time() - startTime)

if algor == 2:

startTime = time()

output1 = T.find2(seachWords[0])

output2 = T.find2(seachWords[1])

if output1 is None or output2 is None: #if word is not in tree, just appends the string words

embeddingsList.append(seachWords)

else:

embeddingsList.append([output1, output2])

runtime = runtime + (time() - startTime)

if algor == 3:

startTime = time()

output1 = T.find3(seachWords[0])

output2 = T.find3(seachWords[1])

if output1 is None or output2 is None: #if word is not in tree, just appends the string words

embeddingsList.append(seachWords)

else:

embeddingsList.append([output1, output2])

runtime = runtime + (time() - startTime)

if algor == 4:

startTime = time()

output1 = T.find4(seachWords[0])

output2 = T.find4(seachWords[1])

if output1 is None or output2 is None: #if word is not in tree, just appends the string words

embeddingsList.append(seachWords)

else:

embeddingsList.append([output1, output2])

runtime = runtime + (time() - startTime)

if algor == 5:

startTime = time()

output1 = T.find5(seachWords[0])

output2 = T.find5(seachWords[1])

if output1 is None or output2 is None: #if word is not in tree, just appends the string words

embeddingsList.append(seachWords)

else:

embeddingsList.append([output1, output2])

runtime = runtime + (time() - startTime)

if algor == 6:

startTime = time()

output1 = T.find6(seachWords[0])

output2 = T.find6(seachWords[1])

if output1 is None or output2 is None: #if word is not in tree, just appends the string words

embeddingsList.append(seachWords)

else:

embeddingsList.append([output1, output2])

runtime = runtime + (time() - startTime)

return embeddingsList, runtime

def main():

htType = 0

buildRunTime = 0

while(int(htType) < 1 or int(htType) > 2):

print("Choose table for implementation")

print("\t1. Hash Table Chaining")

print("\t2. Hash Table Probing")

htType = input()

algType = -1

while(int(algType) < 1 or int(algType) > 6):

print("Choose algorithm type")

print("\t1. ")

print("\t2. ")

print("\t3. ")

print("\t4. ")

print("\t5. ")

print("\t6. ")

algType = input()

hashT = None

if int(htType) == 1:

hashT,buildRunTime = buildHTChaining(int(algType), 10000)

else:

hashT,buildRunTime = buildHTProbing(int(algType), 10000)

#find similarities

print("Reading word file to determine similarities")

embeddingsList,runtime = getEmbeddingsFromFile(hashT,"wordSimilarities2.txt",int(algType))

for embed in embeddingsList:

#pass #used to supress output of similarities

if any(isinstance(words,str) for words in embed):

print("Did not find an embedding for {} or {}".format(embed[0], embed[1]))

else:

print("Similarity [{},{}] = {}".format(embed[0].word,embed[1].word,similarities(embed[0],embed[1])))

print("runtime to build",buildRunTime)

print("runtime to search is, ",runtime)

#used to convert words in glove to format for searching for word similarities.

def convertToCSV(fileName):

inputFile = open(fileName, encoding="utf8")

outputFile = open("wordSimilarities2.txt","w")

word = 0

lineCount = 0

for line in inputFile:

if "\n" in line:

line = line[:-1]

line = line.split(" ")[0]

if line.isalpha():

if(word == 1):

outputFile.write(line + "\n")

word = 0

else:

outputFile.write(line + ",")

word = word + 1

lineCount += 1

if lineCount == 12000:

inputFile.close()

outputFile.close()

return

#used to build csv for graph in report

def timeRunner():

fp = open("htRuntime.csv","w")

for i in range(1,7):

hashT,buildRunTime = buildHTChaining(int(i), 10000)

embeddingsList,runtime = getEmbeddingsFromFile(hashT,"wordSimilarities2.txt",int(i))

fp.write("algorithm " + str(i) + "," + str(buildRunTime) + "," + str(runtime) + "\n")

for i in range(1,7):

hashT,buildRunTime = buildHTProbing(int(i), 10000)

embeddingsList,runtime = getEmbeddingsFromFile(hashT,"wordSimilarities2.txt",int(i))

fp.write("algorithm " + str(i) + "," + str(buildRunTime) + "," + str(runtime) + "\n")

fp.close()

def compareWithBST():

import lab4

#binary search tree

bstBuildTime = 0

start = time()

BST = lab4.buildBinaryTree()

bstBuildTime = time() - start

bstEmbeddingsList,bstSearchTime = lab4.getEmbeddingsFromFile(BST,"wordSimilarities1.txt")

#hash table with chaining

HTC, htcBuildTime= buildHTChaining(5)

htcEmbeddingsListHTC,htcSearchTime = getEmbeddingsFromFile(HTC,"wordSimilarities1.txt")

print("runtimes (build and search) BST: ",bstBuildTime,bstSearchTime)

print("runtimes (build and search) HTC: ",htcBuildTime,htcSearchTime)

if \_\_name\_\_ == '\_\_main\_\_':

main()

#convertToCSV("glove.6B.50d.txt")

#timeRunner()

#compareWithBST()

I [Daniel Cruz] 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