CS 2302 Data Structures
Fall 2019

Lab Report #5

Due: November 1st, 2019
Professor: Olac Fuentes
TA: Anindita Nath

Introduction

For this lab, we were asked to compare the running times of a hash table with chaining (HTC) and a hash table with linear probing (HTLP), designed to hold an organized list of lists, each of those inner lists containing objects. The objects in question for this lab are those of the WordEmbedding class, in which a word and its vector description, or embedding, is stored. To implement these data structures, we will read a set of 400,000 words and their embeddings from a file and construct instances of these hash tables using the classes and methods defined previously in class and written in previous assignments. The main objective of this lab is to recognize how the nature of each data structure influences its respective running time, as well as to ensure an understanding of how to access any particular element of these data structures.

Proposed Solution Design and Implementation

To preface, everything is exactly the same as Lab 4 in terms of the initial user prompt, the construction of the desired data structures. All methods necessary for the two classes were already provided, save the *h(k)* method that makes up Part 1.

Part 1(*h(k)*):

Of note this time around was that since the program was more or less written up in the last lab, the only compelling part of the story in designing this program was building the *h(k)* method that would determine the hash function by which each variable was stored. Even then, it was simply a matter of writing down the variables as was instructed in the handout.

To elaborate briefly, there were six hash functions to write, all ended with "% the length of the hash table:" The first was simply the length of the inputted word; The second was the ascii value of the word's first character; The third was the product of the ascii values of the word's first and last characters; The fourth was the sum of the ascii values of all the word's letters; the fifth was a recursive function that took the ascii value of the first letter and added the ascii value of each following letter multiplied by 255; and lastly, since the sixth hash function was mine to code, I decided to follow in the footsteps of the third and fourth by taking the first letter's ascii value % the last letter's ascii value.

I did also essentially copy-paste these to the *HashTableLP()* class's *h(k)* method, making sure to change out references to *self.bucket* with *self.item*, but ultimately I was never able to fully construct a HashTableLP because it took so much runtime. I'm not sure if I implemented the class incorrectly, if that's intentional, given part 2, but either way, it was highly impractical to wait and record those runtimes. Suffice it to say, they were much longer and much more inefficient than even HTC(1).

Experimental Results

Part 1:

h(1)

```
Choose a hash table implementation.
Type 1 for chaining or 2 for linear probing.

Choice: 1
Choose a hash function corresponding to Part 1 (1-6):

Choice: 1

Hash Table Chain stats:
Running time for hash table chain construction: 517.0052

Running time for hash table chain query processing: 0.8505
```

h(2)

```
Choose a hash table implementation.
Type 1 for chaining or 2 for linear probing.

Choice: 1
Choose a hash function corresponding to Part 1 (1-6):

Choice: 2

Hash Table Chain stats:
Running time for hash table chain construction: 234.8463

Running time for hash table chain query processing: 0.3634
```

h(3)

```
Choose a hash table implementation.
Type 1 for chaining or 2 for linear probing.

Choice: 1
Choose a hash function corresponding to Part 1 (1-6):

Choice: 3

Hash Table Chain stats:
Running time for hash table chain construction: 76.6083

Running time for hash table chain query processing: 0.0834
```

h(4)

```
Choose a hash table implementation.
Type 1 for chaining or 2 for linear probing.

Choice: 1
Choose a hash function corresponding to Part 1 (1-6):

Choice: 4

Hash Table Chain stats:
Running time for hash table chain construction: 35.4035

Running time for hash table chain query processing: 0.0555
```

h(5)

```
Choose a hash table implementation.
Type 1 for chaining or 2 for linear probing.

Choice: 1
Choose a hash function corresponding to Part 1 (1-6):

Choice: 5

Hash Table Chain stats:
Running time for hash table chain construction: 224.4353

Running time for hash table chain query processing: 0.3721
```

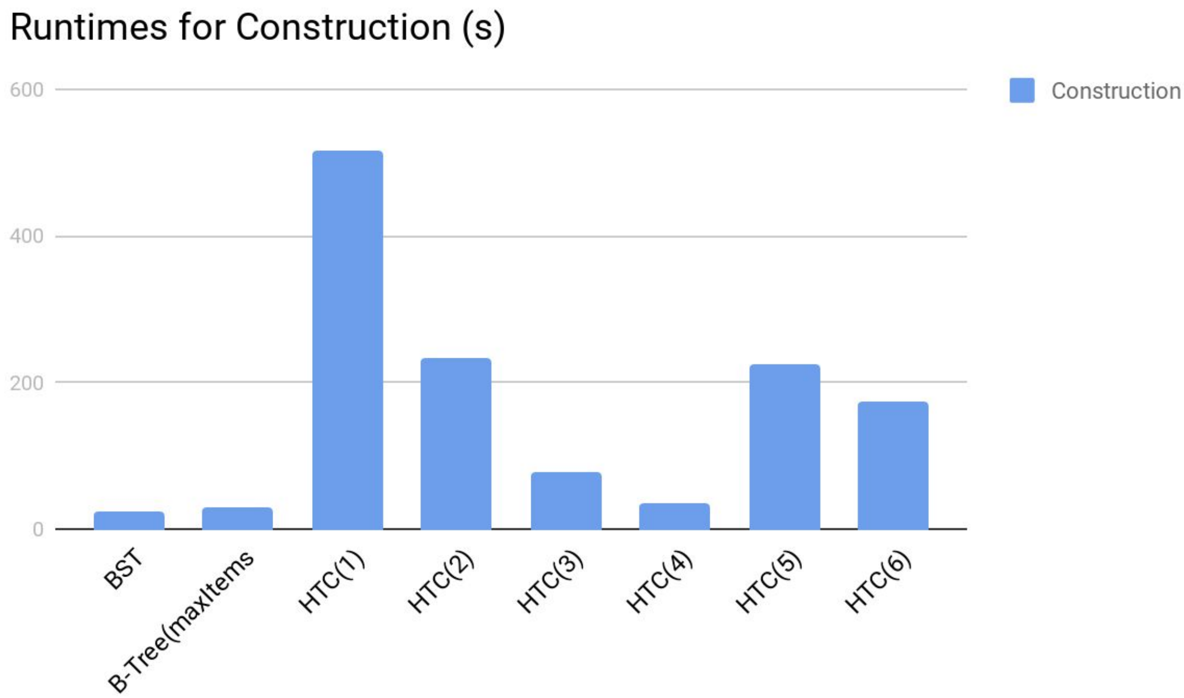
h(6)

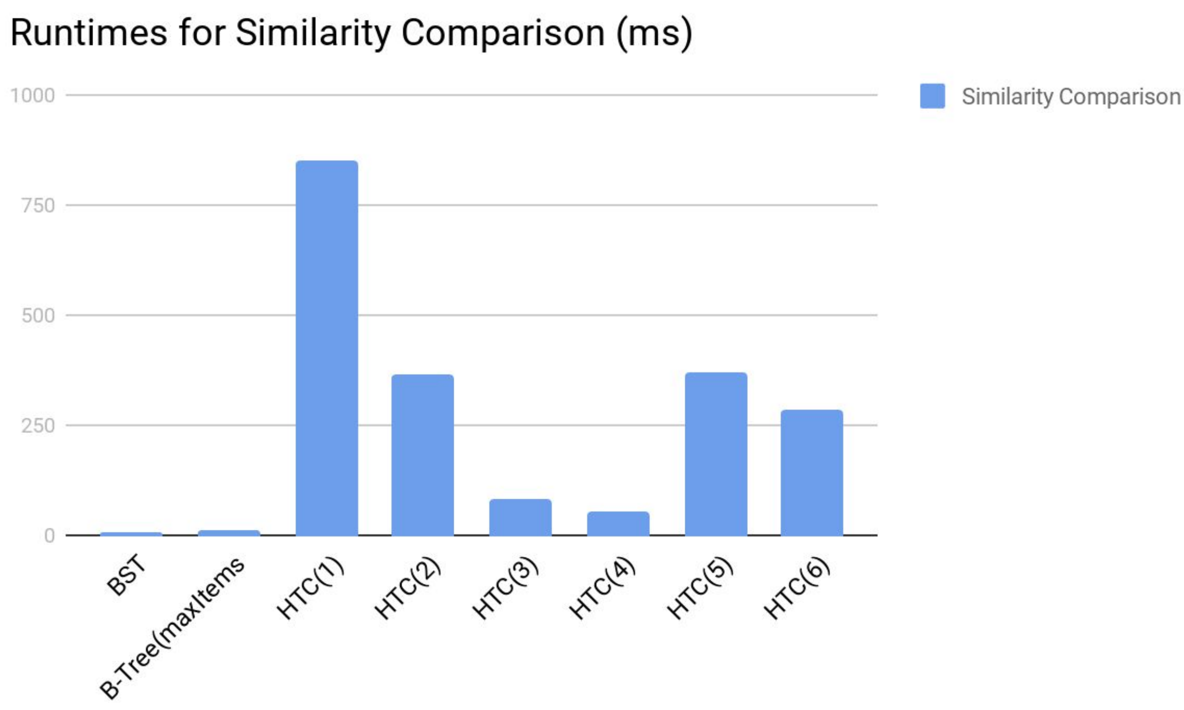
```
Choose a hash table implementation.
Type 1 for chaining or 2 for linear probing.

Choice: 1
Choose a hash function corresponding to Part 1 (1-6):

Choice: 6

Hash Table Chain stats:
Running time for hash table chain construction: 174.3625

Running time for hash table chain query processing: 0.2849
```

Tables:

Running Times (s)

Tree/Hash Table	Construction	Similarity Comparison
BST	24.4069	0.0053
B-Tree(maxItems = 5)	29.5578	0.0133
HTC(1)	517.0052	0.8505
HTC(2)	234.8462	0.36343
HTC(3)	76.6083	0.0834
HTC(4)	35.0435	0.0555
HTC(5)	224.4353	0.3721
HTC(6)	174.3625	0.2849





As the results indicate, every one of the HTC's hash functions failed to live up to either of the trees. HTC(4), the one in which the sum of the ascii values of the letters in a word was used to sort its insertion, was the only one that came remotely close. Surprisingly, even the recursive function wasn't up to par. Though I'm not exactly what you'd call proud, I am glad at the very least the my own hash function did slightly better than the three worst functions.

Conclusion

From this lab, I found only trouble regarding the topics. What seemed like an easy enough task at first, as my relative ease with implementing the HTC will demonstrate, quickly become much more difficult to comprehend, as I felt I didn't really understand how to effectively implement the HTLP class. If its poor speed was intentional, I'm still not sold on its viability as a data structure at all, especially when an HTC makes much more sense logically. Perhaps HTLPs are best used with smaller quantities? More than likely, I just failed to properly understand it. That said, I am definitely going to have to study this issue on my own time before too long.

Appendix

```python
# Course: CS 2302 Data Structures
# Date of last modification: December 5
# Assignment: Lab 5 Hash Table w/ Chaining/Linear Probing
# TA: Anindita Nath
# Professor: Olac Fuentes
# Purpose: The purpose of this lab is to implement hash tables with chaining and with linear probing in place of
#          BSTs and B-trees to compare the running times of all said data structures.

import numpy as np
import time

#WordEmbedding, HashTableChain, and HashTableLP classes provided by Dr. Fuentes
class WordEmbedding(object):
    def __init__(self,word,embedding):
        # word must be a string, embedding can be a list or and array of ints or floats
        self.word = word
        self.emb = np.array(embedding, dtype=np.float32)

class HashTableChain(object):
    # Builds a hash table of size 'size'
    # Item is a list of (initially empty) lists
    # Constructor
    def __init__(self,size, c = 0):
        self.bucket = [[] for i in range(size)]
        self.c = c

    def h(self,k):
        #Part 1
        #The length of the string % n
        if self.c == 1:
            return len(k.word) % len(self.bucket)
        #The ascii value (ord(c)) of the first character in the string % n
        elif self.c == 2:
            return ord(k.word[0]) % len(self.bucket)
        #The product of the ascii values of the first and last characters in the string % n
        elif self.c == 3:
            return (ord(k.word[0]) * ord(k.word[-1])) % len(self.bucket)
        #The sum of the ascii values of the characters in the string % n
        elif self.c == 4:
            ordAll = 0
            for l in range(len(k.word)):
                ordAll += ord(k.word[l])
            return ordAll % len(self.bucket)
        #The recursive formulation h(”,n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n
        elif self.c == 5:
            if k.word == '':
                return 1
            else:
                s = WordEmbedding(k.word[1:], 0)
                return (ord(k.word[0]) + 255 * self.h(s)) % len(self.bucket)
        #Another function of your choice(The product of ascii values of characters in string % n)
        else:
            return (ord(k.word[0]) % ord(k.word[-1])) % len(self.bucket)
```

```
                                 #to sort WordEmbedding objects alphabetically by ASCII value

    def insert(self,k):
        # Inserts k in appropriate bucket (list)
        # Does nothing if k is already in the table
        b = self.h(k)
        if not k.word in self.bucket[b]:#Modified "k" to "k.word" for proper detection
            self.bucket[b].append(k)         #Insert new item at the end

    def find(self,k):
        # Returns bucket (b) and index (i)
        # If k is not in table, i == -1
        b = self.h(k)
        try:
            i = self.bucket[b].index(k)
        except:
            i = -1
        return b, i

    def print_table(self):
        print('Table contents:')
        for b in self.bucket:
            print(b)

    def delete(self,k):
        # Returns k from appropriate list
        # Does nothing if k is not in the table
        # Returns 1 in case of a successful deletion, -1 otherwise
        b = self.h(k)
        try:
            self.bucket[b].remove(k)
            return 1
        except:
            return -1

class HashTableLP(object):
    # Builds a hash table of size 'size', initilizes items to -1 (which means empty)
    # Constructor
    def __init__(self,size,c = 0):
        self.item = np.zeros(size,dtype=np.int)-1
        self.c = c

    def insert(self,k):
        # Inserts k in table unless table is full
        # Returns the position of k in self, or -1 if k could not be inserted
        for i in range(len(self.item)): #Despite for loop, running time should be constant for table with low load factor
            pos = self.h(k)+i
            numK = []
            for i in range(len(k.emb)):
                numK = ''.join(str(k.emb[i]))
            numK = float(numK)
            if self.item[pos] < 0:
                self.item[pos] = numK
                return pos
```

```python
            return -1

    def find(self,k):
        # Returns the position of k in table, or -1 if k is not in the table
        for i in range(len(self.item)):
            pos = self.h(k)+i
            if self.item[pos] == k:
                return pos
            if self.item[pos] == -1:
                return -1
        return -1

    def delete(self,k):
        # Deletes k from table. It returns the position where k was, or -1 if k was not in the table
        # Sets table item where k was to -2 (which means deleted)
        f = self.find(k)
        if f >=0:
            self.item[f] = -2
        return f

    def h(self,k):
        #Part 1
        #The length of the string % n
        if self.c == 1:
            return len(k.word) % len(self.item)
        #The ascii value (ord(c)) of the first character in the string % n
        elif self.c == 2:
            return ord(k.word[0]) % len(self.item)
        #The product of the ascii values of the first and last characters in the string % n
        elif self.c == 3:
            return (ord(k.word[0]) * ord(k.word[-1])) % len(self.item)
        #The sum of the ascii values of the characters in the string % n
        elif self.c == 4:
            ordAll = 0
            for l in range(len(k.word)):
                ordAll += ord(k.word[l])
            return ordAll % len(self.item)
        #The recursive formulation h(",n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n
        elif self.c == 5:
            if k.word == '':
                return 1
            else:
                s = WordEmbedding(k.word[1:], 0)
                return (ord(k.word[0]) + 255 * self.h(s)) % len(self.item)
        #Another function of your choice
        else:
            return #Modified "k%len(self.bucket)" to "ord(k.word[0])"
                                 #to sort WordEmbedding objects alphabetically by ASCII value


    def print_table(self):
        print('Table contents:')
        for b in self.bucket:
            print(b)
```

```python

#My Code
if __name__ == "__main__":
    #Read file and create table based on tableChoice
    file = open("/Users/johanncampos369/Downloads/glove.6B/glove.6B.50d.txt", "r")

    while(True):
        print("Choose a hash table implementation.\nType 1 for chaining or 2 for linear probing.")
        tableChoice = int(input("Choice: "))
        print("Choose a hash function corresponding to Part 1 (1-6): ")
        c = int(input("Choice: "))
        #Store each WordEmbedding object in appropriate type of table
        if tableChoice == 1:
            start = time.time()
            htc = HashTableChain(255, c)#Set size to 255 to get buckets corresponding to ASCII values
                                        #Set c to determine which hash function to use.
            for f in file:
                words = f.split(" ")
                htc.insert(WordEmbedding(words[0], words[1:]))#Insert new object with word and emb list
            end = time.time()
            #Print stats of resulting
            print("\nHash Table Chain stats: ")
            print("Running time for hash table chain construction: %.4f" % (end - start))
            break
        elif tableChoice == 2:
            #size = int(input(("Enter size of htlp: ")))
            print("\nBuilding Hash Table w/ Linear Probing.")
            start = time.time()
            htlp = HashTableLP(400001, c)
            for f in file:
                words = f.split(" ")#Split each line into list of word and emb
                htlp.insert(WordEmbedding(words[0], words[1:]))#Insert new object with word and emb list
            end = time.time()
            #Print stats of resulting B-tree
            print("\nHash Table LP Stats: ")
            print("Running time for hash table LP construction: %.4f" % (end - start))
            break
        else:
            print("Invalid entry. Please try again.")
    #Read second file containing pairs of words
    print("\nReading word file to determine similarities")
    file2 = open("/Users/johanncampos369/Downloads/glove.6B/wordpairs.txt", "r")
    #Compare words in file2 line by line, finding them in tree made from first file
    print("\nWord similarities found: ")
    total = 0.0
    lines = file2.read().splitlines()#Make list of lines without \n
    for i in range(len(lines)):
        start = time.time()
        words = lines[i].split(" ")#Makes list of two words for given line
        if tableChoice == 1:#htc
            word1b, word1i = htc.find(WordEmbedding(words[0], 0))#find returns bucket and index of WordEmbedding objects
            word2b, word2i = htc.find(WordEmbedding(words[1], 0))
            #Take necessary numbers to calculate similarity based on emb by accessing htc elements via bucket and index
            top = np.dot(htc.bucket[word1b][word1i].emb, htc.bucket[word2b][word2i].emb)
            bottom = np.dot(np.linalg.norm(htc.bucket[word1b][word1i].emb), np.linalg.norm(htc.bucket[word2b][word2i].emb))
        else:#Previous try-except allows for tableChoice to only be either 1 or 2, thus making this htlp
            word1 = htlp.find(WordEmbedding(words[0],0))#find returns index of WordEmbedding object that contains sought for word
            word2 = htlp.find(WordEmbedding(words[1],0))
            #Take necessary numbers to calculate similarity based on emb by accessing htlp elements via index
            top = np.dot(htlp.item[word1], htlp.item[word2])
            bottom = np.dot(np.linalg.norm(htlp.item[word1]), np.linalg.norm(htlp.item[word2]))
        #Calculate similarity based on emb from objects of either table
        sim = top / bottom
        end = time.time()
        total += end - start#Accumulates time taken for every similarity comparison
        print("Similarity ", words, " = %.4f" % sim)
    if tableChoice == 1:
        print("Running time for hash table chain query processing: %.4f" % total)
    else:
        print("Running time for hash table LP query processing: %.4f" % total)
```

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.