CS 2302 Data Structures

Fall 2019

Lab Report #5

Due: November 5th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

Introduction

For this lab we were tasked with creating a Hash Table using Linear Probing or Chaining depending on what the user would like to use. Then we would give them six choices on how they would like the data to hash in the list. After the file "glove.6B.50d.txt" has been sorted within the chosen list using the chosen hash the lists stats are displayed then all words in the test file are compared for similarity's and displayed properly.

Proposed Solution Design and Implementation

Part 1:

First, I reused most of the code from lab 4 main.py file and imported two new files "HashTables_LP.py" and "HashTables_Chaining.py". These two files contain the classes of both Hash tables. Once the Program loads the chosen method of hashing would be added to the creation of the table along with an appropriate size to create the initial table.

Part 2:

For Hash Tables with Linear Probing I first created a while loop to figure out how many lines where in the glove.6B.50d file and used that number plus one to be the length of the hash table. Next it would go through the file line by line and split each line into a word and a list for the 50-word embeddings. It would then check if, "word.isalpha" to check for any special characters in the sting. If the word had no special characters, it would be added to a word embedding class then that would be added to the Hash Table and if not, the method would move on to the next line. Once all the word-embedding classes are added the tables stats are displayed. Then the similarities file is used to find the words embeddings within the hash table and compared using dot product of the two words embeddings divided by the magnitudes of the two words embeddings. They are then displayed along with the running time to fine all of them.

Part 3:

For Hash Tables with Chaining I first created a while loop to figure out how many lines where in the glove.6B.50d file and used that number divided by twenty to be the length of the hash table. Next it would go through the file line by line and split each line into a word and a list for the 50-word embeddings. It would then check if, "word.isalpha" to check for any special characters in the sting. If the word had no special characters, it would be added to a word embedding class then that would be added to the Hash Table and if not, the method would move on to the next line. Once all the word-embedding classes are added the tables stats are displayed. Then the similarities file is used to find the words embeddings within the hash table and compared using dot product of the two words embeddings divided by the magnitudes of the two words embeddings. They are then displayed along with the running time to fine all of them.

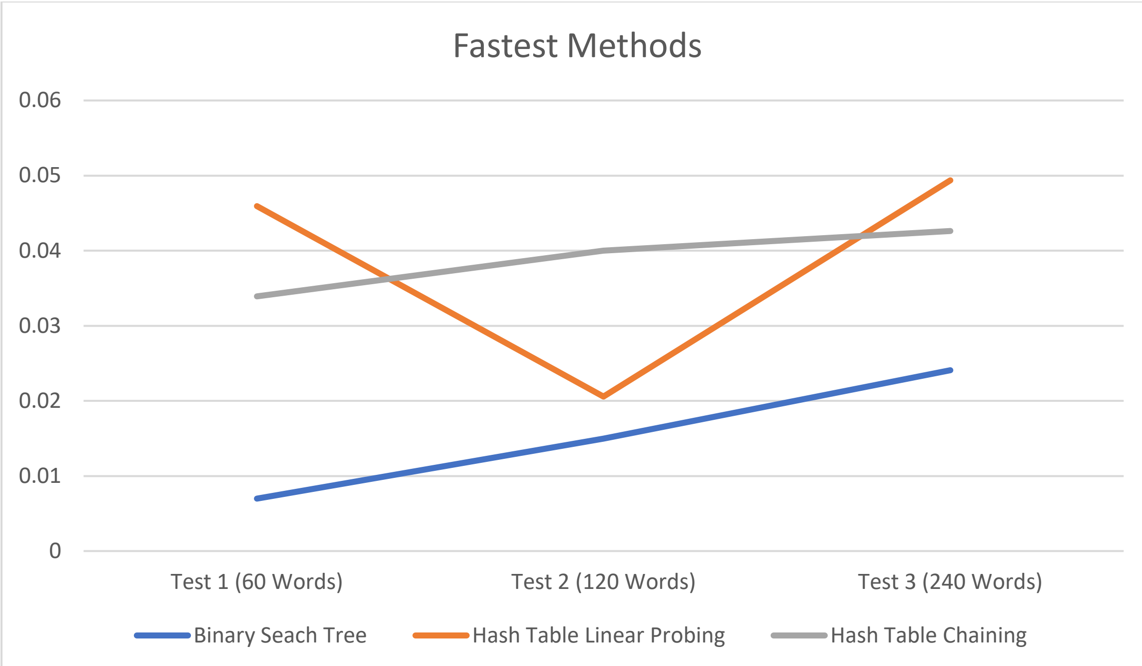
Experimental Results

Hash Tables with Linear Probing:

Hashing Methods	Test 1 (60 words)	Test 2 (120 words)	Test 3 (240 words)
1. The length of the string % n (Average time to create table: 6 Hours)	839.46397 Seconds	842.14224 Seconds	876.45831 Seconds
2. The ascii value (ord(c)) of the first character in the string % n (Average time to create table: 2 Hours)	653.198631 Seconds	673.416398 Seconds	690.23659 Seconds
3. The product of the ascii values of the first and last characters in the string % n (Average time to create table: 1.5 Hour)	371.123642 Seconds	421.123475 Seconds	424.123541 Seconds
4. The sum of the ascii values of the characters in the string % n (Average time to create table: 45 Min)	0.2932154 Seconds	0.3247586 Seconds	0.3842327 Seconds
5. The recursive formulation h(”,n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n (Average time to create table: 32 Sec)	0.0385463 Seconds	0.0540861 Seconds	0.0493659 Seconds
6. The ascii value of each letter of the string times 2302 % n (Average time to create table: 1.5 Min)	0.2349631 Seconds	0.2314586 Seconds	0.2546923 Seconds

Hash Tables with Chaining:

Hashing Methods	Test 1 (60 words)	Test 2 (120 words)	Test 3 (240 words)
1. The length of the string % n (Average time to create table: 4.5 Min)		8.0543294 Seconds	7.9071414 Seconds
2. The ascii value (ord(c)) of the first character in the string % n (Average time to create table: 2 Min)	4.1035479 Seconds	4.0026483 Seconds	4.3181658 Seconds
3. The product of the ascii values of the first and last characters in the string % n (Average time to create table: 1 Min)	2.1264592 Seconds	2.4851369 Seconds	2.3752146 Seconds
4. The sum of the ascii values of the characters in the string % n (Average time to create table: 18 Sec)	0.1547962 Seconds	0.1746324 Seconds	0.189647 Seconds
5. The recursive formulation h(”,n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n (Average time to create table: 13 Sec)	0.0155804 Seconds	0.0237725 Seconds	0.048619 Seconds
6. The ascii value of each letter of the string times 2302 % n (Average time to create table: 18 Sec)	0.1296547 Seconds	0.1322591 Seconds	0.1913447 Seconds



Conclusion

In Conclusion the both methods were able to find the similarities in a relatively fast time compared to Btrees but where longer faster than Binary search trees. However, the time taken to create a hash Table of this size could take a massive amount of time if the hash method isn't good. As with the case of using the length of the string divided by n the program took about 6 hours to create the table as compared to the hash using the recursive formulation which created it in 30 seconds. Also the implementation of a very sophisticated hash function could result in the best case of a data search of O(1) so theoretically of applied well a hash function has the potential to beat the run time of a binary search tree.

Appendix

Lab5.py:

```
'''
    Cs2302 Data Structures
    Issac Rivas
    Lab 4
    Dr.Fuentues
'''
import Binary_Search_Tree as BSTree
import BTree
import HashTables_Chaining as HTC
import HashTables_LP as HTLP
import numpy as np
import time as t
```

```python

#Creates an object that holds the word as well as 50 element embedding list
class WordEmbedding(object):
    def __init__(self, word, embedding):
        self.word = word
        self.emb = np.array(embedding, dtype=np.float32)

#Splis a text file line into a word and a float list of the remainig 50 elements
def lineSplit(line):
    emb = line.split()
    word = emb[0]
    del emb[0]
    for x in range(0, len(emb)):
        emb[x] = float(emb[x])
    return word, emb

#Calculates for the Similartiy of two words
def Similarity(word0, word1):
    dotProduct = np.dot(word0.emb,word1.emb)
    magnitude = np.linalg.norm(word0.emb) * np.linalg.norm(word1.emb)
    return dotProduct/magnitude

if __name__ == "__main__":
    print("Choose table implementation:")
    print("Enter the number '1' for a Binary Search Tree")
    print("'2' for a B-tree")
    print("'3' for a Hash Table with Linear Probing")
    print("'4' for a Hash Table with Chaining")
    ans = int(input("Input: "))
    while(ans != 1 and ans != 2 and ans != 3 and ans != 4):
        print()
        print("Incorrect input entered please Try again.")
        print("Type 1 for binary search or 2 for B-tree")
        ans = int(input("Input: "))

    filepath = 'C:/Users/Issac/Desktop/Lab_5/glove.6B.50d.txt'
    testpath = 'C:/Users/Issac/Desktop/Lab_5/test3.txt'

    #If user wants to create a BST
    if ans == 1:
        start = t.time()
        T = None
        with open(filepath,  encoding="utf8") as fp:
            for line in fp:
                word, emb = lineSplit(line)
                if word.isalpha():
                    temp = WordEmbedding(word, emb)
                    T = BSTree.Insert(T, temp)
        Total = t.time() - start
        print()
        print("Building Binary Search Tree")
        print()
        print("Binary Tree Stats:")
        print("Number of Nodes: ", BSTree.numOfNodes(T))
        print("Tree Height is: ", BSTree.treeHeight(T))
        print("Running time for Binary Search Tree construction:", round(Total, 4), " seconds")
        print()
        print("Reading word file to determine similarities")
        print()
        print("Word similarities found:")
        start = t.time()
        with open(testpath) as tp:
            for line in tp:
                words = line.split()
                temp0 = BSTree.Search(T, words[0])
                temp1 = BSTree.Search(T, words[1])
                sim = Similarity(temp0.data, temp1.data)
                print("Similarity [" + words[0] + "," + words[1] + "] = ", round(sim, 4))
        Total = t.time() - start
        print()
        print("Running time for Binary Search Tree query processing: ", round(Total, 7))
```

```
        print("Running time for Binary Search Tree query processing: ", round(Total, 7))

    #If user wants to create a B-tree
    elif ans == 2:
        max_data = int(input("Please enter the maximum number of items per node: "))
        while (max_data < 1):
            print("Error please insert a number equal to or greater then 1.")
            max_data = int(input("Please enter the maximum number of items per node: "))
        start = t.time()
        T = BTree.BTree([], max_data)
        with open(filepath,  encoding="utf8") as fp:
            for line in fp:
                word, emb = lineSplit(line)
                if word.isalpha():
                    temp = WordEmbedding(word, emb)
                    BTree.Insert(T, temp)
        print()
        print("Building B-Tree")
        print()
        print("B-Tree Stats:")
        print("Number of Nodes: ", BTree.NumItems(T))
        print("Tree Height is: ", BTree.Height(T))
        Total = t.time() - start
        print("Running time for B-tree construction (with max_items = " + str(max_data) + "):", round(Total, 4), "seconds")
        print()
        print("Reading word file to determine similarities")
        print()
        print("Word similarities found:")
        start = t.time()
        with open(testpath) as tp:
            for line in tp:
                words = line.split()
                wordemb1 = WordEmbedding(words[0], [])
                wordemb2 = WordEmbedding(words[1], [])
                temp0 = BTree.Search(T, wordemb1)
                temp1 = BTree.Search(T, wordemb2)
                sim = Similarity(temp0, temp1)
                print("Similarity [" + words[0] + "," + words[1] + "] = ", round(sim, 4))
        Total = t.time() - start
        print()
        print("Running time for B-Tree query processing (with max_items = " + str(max_data) + "):", round(Total, 7), "seconds")

    #If the user want to create a Hash Table with Linear Probing
    else:
        print("Choose Hash Function you would like to use:")
        print("Enter the number")
        print("'1' for The length of the string % n")
        print("'2' for The ascii value (ord(c)) of the first character in the string % n")
        print("'3' for The product of the ascii values of the first and last characters in the string % n")
        print("'4' for The sum of the ascii values of the characters in the string % n")
        print("'5' for The recursive formulation h(”,n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n")
        print("'6' for The ascii value of each letter of the string times 2302 % n")
        ans2 = int(input("Input: "))
        while(ans2 != 1 and ans2 != 2 and ans2 != 3 and ans2 != 4 and ans2 != 5 and ans2 != 6):
            print()
            print("Enter the number")
```

```
            print("'1' for The length of the string % n")
            print("'2' for The ascii value (ord(c)) of the first character in the string % n")
            print("'3' for The product of the ascii values of the first and last characters in the string % n")
            print("'4' for The sum of the ascii values of the characters in the string % n")
            print("'5' for The recursive formulation h(“,n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n")
            print("'6' for The ascii value of each letter of the string times 2302 % n")
            ans2 = int(input("Input: "))
        if ans == 3:
            start = t.time()
            count = 0
            with open(filepath,  encoding="utf8") as fp:
                for line in fp:
                    count += 1
            T = HTLP.HashTableLP(int(count+1), ans2)
            counter = 0
            with open(filepath,  encoding="utf8") as fp:
                for line in fp:
                    word, emb = lineSplit(line)
                    if word.isalpha():
                        temp = WordEmbedding(word, emb)

                        #If the used buckests/Unused is grteater then 0.8 precent resize the bucket
                        if (counter / len(T.item)) >= 0.95:
                            T = HTLP.resize(T)
                        HTLP.insert(T, temp)
                        counter += 1
            Total = t.time() - start
            print()
            print("Building Hash Table with Linear Probing")
            print()
            print("Hash Table Stats:")
            print("Length of the Hash Table with Linear Probing: ", len(T.item))
            print("Number of items in the Hash Table with Linear Probing: ", counter)
            print("Running time for Hash Table with Linera Probing construction:", round(Total, 4), " seconds")
            print()
            print("Reading word file to determine similarities")
            print()
            print("Word similarities found:")
            start = t.time()
            with open(testpath) as tp:
                for line in tp:
                    words = line.split()
                    W1 = WordEmbedding(words[0], [])
                    W2 = WordEmbedding(words[1], [])
                    temp0 = HTLP.find(T, W1)
                    temp1 = HTLP.find(T, W2)
                    sim = Similarity(T.item[temp0], T.item[temp1])
                    print("Similarity [" + words[0] + "," + words[1] + "] = ", round(sim, 4))
            Total = t.time() - start
            print()
            print("Running time for Hash Table with Linear Probing query processing: ", round(Total, 7))

        #If the user want to create a Hash Table with Chaining
        elif ans == 4:
            start = t.time()
            count = 0
```

```
            with open(filepath,  encoding="utf8") as fp:
                for line in fp:
                    count += 1
            T = HTC.HashTableChain(int(count/20), ans2)
            counter = 0
            with open(filepath,  encoding="utf8") as fp:
                for line in fp:
                    word, emb = lineSplit(line)
                    if word.isalpha():
                        temp = WordEmbedding(word, emb)
                        HTC.insert(T, temp)
                        counter += 1
            Total = t.time() - start
            print()
            print("Building Hash Table with Chaining")
            print()
            print("Hash Table Stats:")
            print("Length of the Hash Table with Chaining: ", len(T.bucket))
            print("Number of items in the Hash Table with Chaining: ", counter)
            print("Running time for Hash Table with Chaining construction:", round(Total, 4), " seconds")
            print()
            print("Reading word file to determine similarities")
            print()
            print("Word similarities found:")
            start = t.time()
            with open(testpath) as tp:
                for line in tp:
                    words = line.split()
                    W1 = WordEmbedding(words[0], [])
                    W2 = WordEmbedding(words[1], [])
                    temp0, ind0 = HTC.find(T, W1)
                    temp1, ind1 = HTC.find(T, W2)
                    sim = Similarity(T.bucket[temp0][ind0], T.bucket[temp1][ind1])
                    print("Similarity [" + words[0] + "," + words[1] + "] = ", round(sim, 4))
            Total = t.time() - start
            print()
            print("Running time for Hash Table with Chaining query processing: ", round(Total, 7))
```

HastTables_LP:

```
import numpy as np

class HashTableLP(object):
    def __init__(self,size, hashNum):
        self.item = np.zeros(size,dtype=np.object)-1
        self.hashNum = hashNum

def insert(self,k):
    for i in range(len(self.item)):
        pos = h(self, k)+i
        if pos >= len(self.item):
            pos = pos - len(self.item)
        if self.item[pos] == -1:
            self.item[pos] = k
            return pos
    return -1
```

```python

def find(self,k):
    for i in range(len(self.item)):
        pos = h(self, k)+i
        if pos >= len(self.item):
            pos = pos - len(self.item)
        if self.item[pos] == -1:
            return -1
        if self.item[pos].word == k.word:
            return pos
    return -1

def delete(self,k):
    f = self.find(k)
    if f >=0:
        self.item[f] = -2
    return f

def h(self,k):
    if self.hashNum == 1:
        return ((len(k.word)-1)) % len(self.item)
    elif self.hashNum == 2:
        return ord(k.word[0]) % len(self.item)
    elif self.hashNum == 3:
        return (ord(k.word[0])+ord(k.word[-1])) % len(self.item)
    elif self.hashNum == 4:
        sum = 0
        for x in range(len(k.word)):
            sum = sum + ord(k.word[x])
        if sum > len(self.item):
            sum = sum - len(self.item)
        return (sum % len(self.item))
    elif self.hashNum == 5:
        return recursiveForm(self, len(self.item), k.word)
    else:
        mx = 0
        for x in range(len(k.word)):
            temp = ord(k.word[x])
            mx += temp * 2302
        return mx % len(self.item)

def recursiveForm(self, n, s):
    if s == '':
        return 1
    else:
        return ((ord(s[0]) + 255*recursiveForm(self, n, s[1:])) % n)

def print_table(self):
    print('Table contents:')
    print(self.item)

def n(self, k):
    temp = []
    hash = self.h(k)
    for b in self.item:
        if self.h(b) == hash:
            temp.append(b)
    if len(temp) < 1:
        return -1
    else:
        temp.sort()
        return temp[len(temp)-1]

def resize(H):
    new = HashTableLP(len(H.item)*2, H.hashNum)
    for x in range(len(H.item)):
        if H.item[x] != -1:
            insert(new, H.item[x])
    return new
```

HashTables_Chaining:

```
class HashTableChain(object):
    def __init__(self,size, hashNum):
        self.bucket = [[] for i in range(size)]
        self.hashNum = hashNum

def insert(T,k):
    b = h(T, k)
    if len(T.bucket[b]) >= len(T.bucket)*3:
        T = resize(T)
    if not k in T.bucket[b]:
        T.bucket[b].append(k)


def find(T,k):
    b = h(T, k)
    i = -1
    for x in range(len(T.bucket[b])):
        if T.bucket[b][x].word == k.word:
            i = x
    return b, i

def print_table(T):
    print('Table contents:')
    for b in T.bucket:
        print(b)

def delete(T,k):
    b = T.h(k)
    try:
        T.bucket[b].remove(k)
        return 1
    except:
        return -1

def resize(H):
    new = HashTableChain(len(H.bucket)*2, H.hashNum)
    for x in range(len(H.bucket)):
      if H.bucket[x] != None:
        for y in range(len(H.bucket[x])):
            insert(new, H.bucket[x][y])
    return new

def h(self,k):
    if self.hashNum == 1:
        return ((len(k.word)-1)) % len(self.bucket)
    elif self.hashNum == 2:
        return ord(k.word[0]) % len(self.bucket)
    elif self.hashNum == 3:
        return (ord(k.word[0])+ord(k.word[-1])) % len(self.bucket)
    elif self.hashNum == 4:
        sum = 0
        for x in range(len(k.word)):
            sum = sum + ord(k.word[x])
        return sum % len(self.bucket)
    elif self.hashNum == 5:
        return recursiveForm(self, len(self.bucket), k.word)
    else:
        mx = 0
        for x in range(len(k.word)):
             temp = ord(k.word[x])
```

```
            mx += temp * 2302
        return mx % len(self.bucket)

def recursiveForm(self, n, s):
    if s == '':
        return 1
    else:
        return (ord(s[0]) + 255*recursiveForm(self, n, s[1:])) % n
```

I Issac Rivas, certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, preformed 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.