Lab Assignment 5

ISMAEL VILLALOBOS

PROFESSOR FUENTES

TA: DITA NATH

11-6-19

Introduction

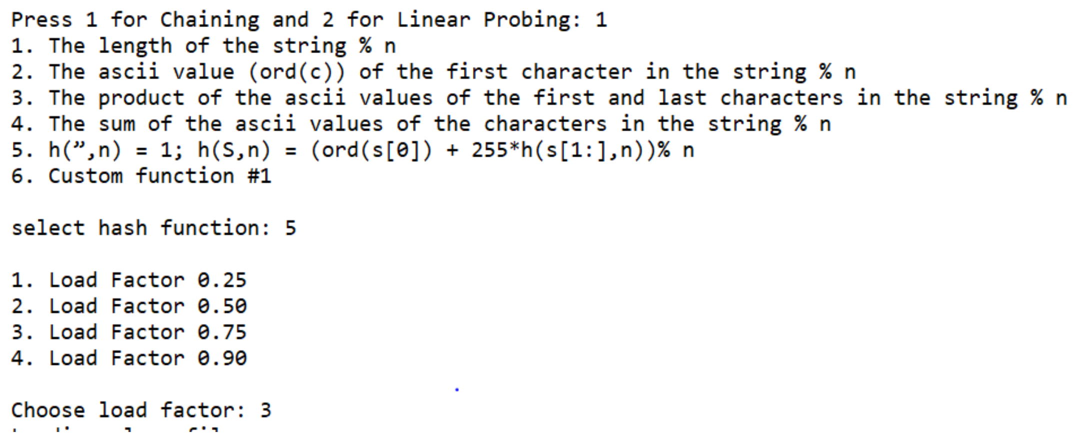
For Lab5, we were given the task to implement the same concept from the previous lab which was Natural Language Processing and finding similarities between word vectors. This time we used Hash Tables with Chaining as well as with Linear Probing, aside from using both types of hashing we used 6 different types of hash functions which included:

- The length of the string % n
- The ascii value (ord(c)) of the first character in the string % n
- The product of the ascii values of the first and last characters in the string % n
- The sum of the ascii values of the characters in the string % n
- The recursive formulation h(”,n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n
- Another function of your choice

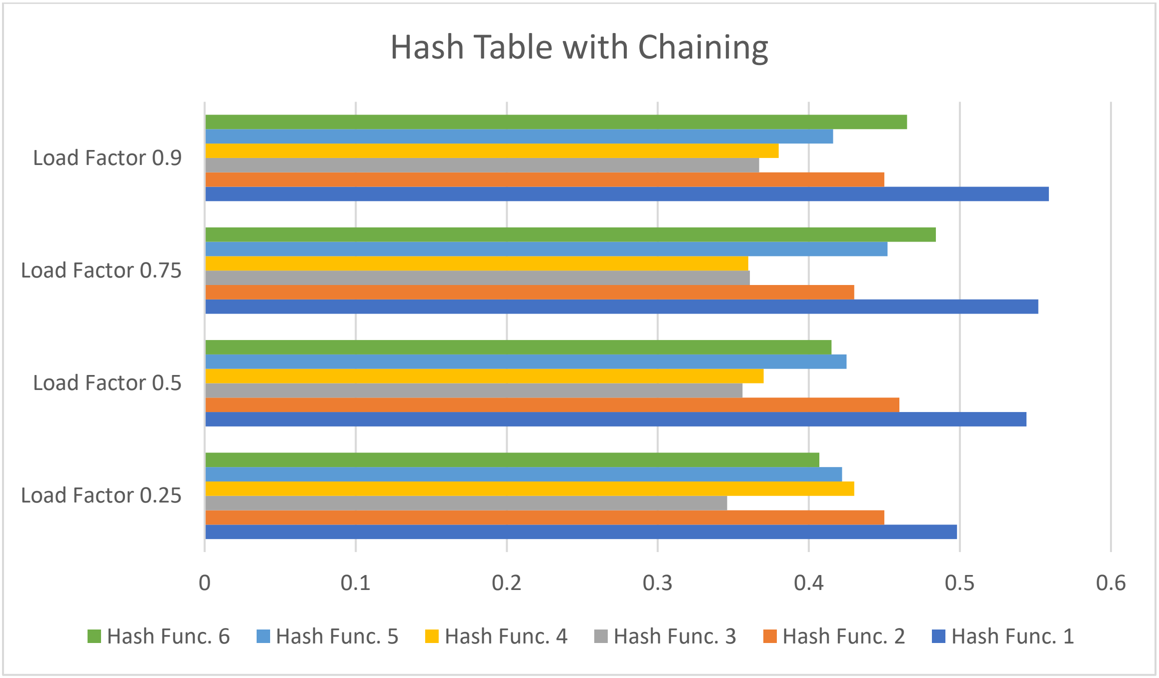
Proposed Solutions

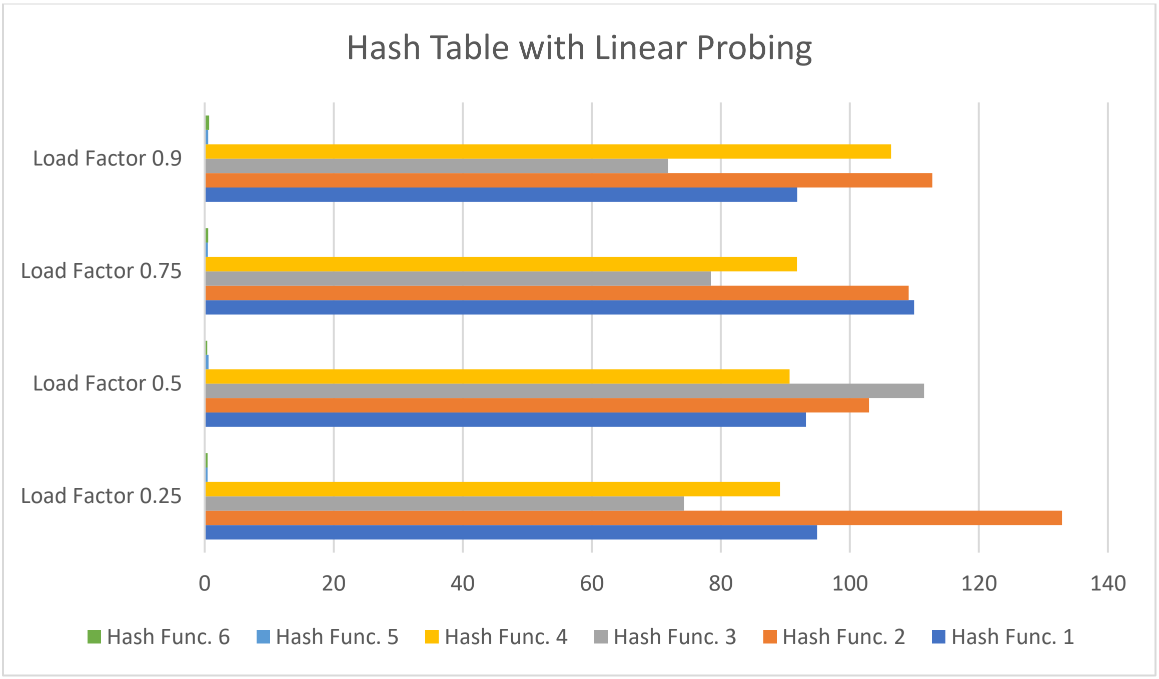
Similar to the approach used in Lab 4 we begin by opening the Glove File provided. When reading the file, I strip each line into a word and an embedding that will be inserted in the corresponding bucket in the hash table. In order to find the appropriate position in the hash table we must implement the functions that were provided in order to find the hash value. In addition to the hash functions as options, I decided to give the user options as to the load factor for the table in order to test different table sizes.

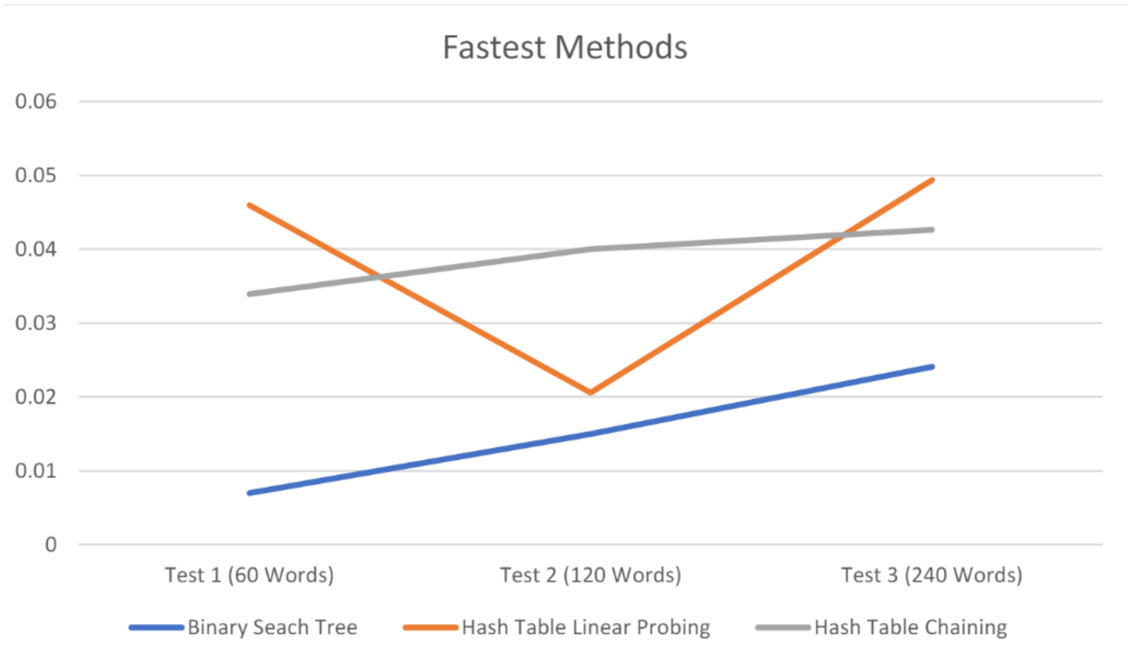
Experimental Results



```
Press 1 for Chaining and 2 for Linear Probing: 1
1. The length of the string % n
2. The ascii value (ord(c)) of the first character in the string % n
3. The product of the ascii values of the first and last characters in the string % n
4. The sum of the ascii values of the characters in the string % n
5. h(”,n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n
6. Custom function #1

select hash function: 5

1. Load Factor 0.25
2. Load Factor 0.50
3. Load Factor 0.75
4. Load Factor 0.90

Choose load factor: 3
Choose load factor: 3
Loading glove file...
Similarity [brown,brown] = 100.0
Similarity [brown,color] = 56.02
Similarity [brown,bear] = 56.3
Similarity [brown,university] = 33.9
Similarity [curry,food] = 42.1
Similarity [curry,mexico] = 15.62
Similarity [curry,india] = 15.59
Similarity [indian,native] = 56.2
Similarity [indian,india] = 86.49
Similarity [apple,fruit] = 59.18
Similarity [apple,computer] = 64.03
Similarity [apple,banana] = 56.08
Similarity [russia,putin] = 77.53
Similarity [russia,stalin] = 44.66


Running time for similarities: 0.001

Hash Table with Chaining stats:
Running time for construction: 0.440818
```







Big O Notation & Runtimes

The hash table with chaining was so much faster than both BST and B-Tree in terms of speed and efficiency. However for the search function the table with chaining is faster than a B-Tree but not faster than a BST.

Hash operations usually have O(1) time with at worst being O(n)

Tree operations are O(logn)

Conclusion

With my findings I found that hash tables are way faster than any tree structures and will take that into account for future projects. This really opened my eyes to the efficiency of the algorithms and their use for Natural Language Processing.

Academic Honesty

"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."

Name: Ismael Villalobos

Appendix:

```
'''
Ismael Villalobos
11-5-19
Lab Assignment 5
Professor Fuentes
TA-Dita Nath
Purpose:
'''
import re
import time
import numpy as np

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) # For Lab 4, len(embedding=50)
```

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

    # Hash function with length of string k % size of table
    def lenword_hash(self,k):
        if isinstance(k, WordEmbedding):
            k=k.word


        return len(k)%len(self.bucket)
    # Hash function with ASCII value of the first character of k % size of table
    def ascii_first_hash(self,k):
        if isinstance(k, WordEmbedding):
            k=k.word
        return ord(k[0])%len(self.bucket)

    # Hash function with product of ASCII values from first and last char % size of table
    def ascii_product_hash(self,k):
        if isinstance(k, WordEmbedding):
            k=k.word
        return (ord(k[0])*ord(k[-1]))%len(self.bucket)

    # Hash function with the sum of the ASCII values in k % size of table
    def ascii_sum_hash(self, k):
```

```
    if isinstance(k, WordEmbedding):
        k=k.word
    return sum(map(ord, k))%len(self.bucket)

# Recursive Hash function that multiplies the ASCII values of all the characters
# in k (plus 255 on each value) % size of table
def recursive_hash(self, k):
    if isinstance(k, WordEmbedding):
        k=k.word

    if len(k) == 0:
        return 1
    return (ord(k[0]) + 255 * self.recursive_hash(k[1:])) % len(self.bucket)

# Custom Hash function done with a loop to add all the ASCII
# values in k to the power of it's index % size of table
def custom_hash(self, k):
    if isinstance(k, WordEmbedding):
        k=k.word
    total=0
    for i in range(len(k)):
        total+=ord(k[i])**i

    return total % len(self.bucket)



# h function returns the selected hash function choice
def h(self,k,choice):
    if choice == 1:
        return self.lenword_hash(k)
    if choice == 2:
        return self.ascii_first_hash(k)
    if choice == 3:
        return self.ascii_product_hash(k)
    if choice == 4:
        return self.ascii_sum_hash(k)
    if choice == 5:
        return self.recursive_hash(k)
    if choice == 6:
        return self.custom_hash(k)



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

def find_emb(self,k,choice):
    # Returns bucket (b) and index (i)
    # If k is not in table, i == -1
    b = self.h(k,choice)

    for j in self.bucket[b]:
        if j.word==k:
            return j.emb
        return

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

    def load_factor(self):
        #number of elements/size
        num=0
        for i in self.bucket:
            num+=len(i)
        return num/len(self.bucket)

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

    def insert(self,k,choice):
        # initial position of k
        start = self.h(k.word,choice)
        for i in range(len(self.item)):
            # initial positon in the table to check
            pos = (start+i)%len(self.item)
        # check if current element is a WordEmbedding
        if isinstance(self.item[pos], WordEmbedding):
            # check if element to be inserted is already in the table
            if self.item[pos].word==k.word:
                return -1
        # if it is not a WordEmbedding, check if it's less than 0
        elif self.item[pos] < 0:
            # insert k if current element is less than 0
            self.item[pos]=k
            return pos

def find_emb(self,k,choice):
    # initial position of k
    if isinstance(k, WordEmbedding):
        k=k.word
    start=self.h(k,choice)
    for i in range(len(self.item)):
        # initial positon in the table to check
        pos = (start+i)%len(self.item)
        # if current element is in the table, return it's embedding
        try:
            if self.item[pos].word == k:
                return self.item[pos].emb
        # if it throws an error, k is not in the table, return None
        except:
            if self.item[pos]<0:
                return None
```

```
# Hash function with length of string k % size of table
def lenword_hash_LP(self,k):
    if isinstance(k, WordEmbedding):
        k=k.word
    return len(k)%len(self.item)

# Hash function with ASCII value of the first character of k % size of table
def ascii_first_hash_LP(self,k):
    if isinstance(k, WordEmbedding):
        k=k.word
    return ord(k[0])%len(self.item)

# Hash function with product of ASCII values from first and last char % size of table
def ascii_product_hash_LP(self,k):
    if isinstance(k, WordEmbedding):
        k=k.word
    return (ord(k[0])*ord(k[-1]))%len(self.item)

# Hash function with the sum of the ASCII values in k % size of table
def ascii_sum_hash_LP(self, k):
    if isinstance(k, WordEmbedding):
        k=k.word
    return sum(map(ord, k))%len(self.item)

# Recursive Hash function that multiplies the ASCII values of all the characters
# in k (plus 255 on each value) % size of table
def recursive_hash_LP(self, k):
    if isinstance(k, WordEmbedding):
            k=k.word

        if len(k) == 0:
            return 1
        return (ord(k[0]) + 255 * self.recursive_hash_LP(k[1:])) % len(self.item)

    # Custom Hash function done with a loop to add all the ASCII
    # values in k to the power of it's index % size of table
    def custom_hash_LP(self, k):
        if isinstance(k, WordEmbedding):
            k=k.word
        total=0
        for i in range(len(k)):
            total+=ord(k[i])**i

        return total % len(self.item)

    def h(self,k,choice):
        if choice == 1:
            return self.lenword_hash_LP(k)
        if choice == 2:
            return self.ascii_first_hash_LP(k)
        if choice == 3:
            return self.ascii_product_hash_LP(k)
        if choice == 4:
            return self.ascii_sum_hash_LP(k)
        if choice == 5:
            return self.recursive_hash_LP(k)
        if choice == 6:
            return self.custom_hash_LP(k)



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

    def load_factor(self):
        #number of elements/size
        num=0
        for i in self.item:
            if isinstance(i, WordEmbedding):
                num+=1
        return num/len(self.item)



def menu():
    print('1. The length of the string % n')
    print('2. The ascii value (ord(c)) of the first character in the string % n')
    print('3. The product of the ascii values of the first and last characters in the string % n')
    print('4. The sum of the ascii values of the characters in the string % n')
    print('5. h(”,n) = 1; h(S,n) = (ord(s[0]) + 255*h(s[1:],n))% n')
    print('6. Custom function #1')

    choice = int(input('select hash function: '))

    print('\n1. Load Factor 0.25')
    print('2. Load Factor 0.50')
    print('3. Load Factor 0.75')
    print('4. Load Factor 0.90')

# Load factor options for 14108 words with 6400000 lines from GLoVe file
    lf=int(input('Choose load factor: '))
    if lf==1:
        table_size=56432
    if lf==2:
        table_size=28216
    if lf==3:
        table_size=18810
    if lf==4:
        table_size=15505

    return choice, table_size

def HashTableC_Main():

    choice, table_size = menu()

    H = HashTableChain(table_size)
    # Pattern to be used to remove words with unwanted characters
    pattern=re.compile("[A-Za-z]+")
    print('Loading glove file...')
    # Open glove file
    file = open('glove.6B.50d.txt',encoding="utf-8")
    count=0
    # Start counter
    start = time.perf_counter()

    # readlines limited to a small sample of the GLoVe file to reduce times of certain
    # hash functions
    for line in file.readlines(6400000):
        row = line.strip().split(' ')
         # Check if word matches the pattern of characters
        if pattern.fullmatch(row[0]) is not None:
            # Insert into Hash Table with word and its embedding
            H.insert(WordEmbedding(row[0],[(i) for i in row[1:]]),choice)
            count+=1
    # Stop counter
    end = time.perf_counter()

    Similarity(H, choice, 'wordpairs.txt', 15)

    print('\nHash Table with Chaining stats:')
    print('Running time for construction: '+ str(round((end - start), 6))+'\n')
    print('Table size:', table_size)
    print('Load factor:',round(H.load_factor(),6))

    return H

def HashTableLP_Main():

    choice, table_size = menu()
```

```
H = HashTableLP(table_size)

# Pattern to be used to remove words with unwanted characters
pattern=re.compile("[A-Za-z]+")
print('Loading glove file...')
# Open glove file
file = open('glove.6B.50d.txt',encoding="utf-8")

# Start counter
start = time.perf_counter()
count=0
# readlines limited to a small sample of the GLoVe file to reduce times of certain
# hash functions
for line in file.readlines(6400000):
    row = line.strip().split(' ')
    # Check if word matches the pattern of characters
    if pattern.fullmatch(row[0]) is not None:
        # Insert into Hash Table with word and its embedding
        H.insert(WordEmbedding(row[0],[(i) for i in row[1:]]),choice)
        count+=1
# Stop counter
end = time.perf_counter()

Similarity(H, choice, 'wordpairs.txt',15)

print('\nHash Table with Linear Probing stats:')
print('Running time for construction: '+ str(round((end - start), 6))+'\n')
print('Table size:', table_size)
```

```
    print('Load factor:',round(H.load_factor(),6))



def Similarity(H,choice, file_choice, num_pairs):


    # Open and read pairs word file and insert into a list as list of pairs
    pairs=[line.strip().split(' ') for line in open(file_choice,'r')]

    # Assign timer to 0
    timer=0

    # Loop to iterate through list of pairs line by line
    for i in range(num_pairs):
        # Start timer
        start = time.perf_counter()

        # word1 gets the first column in each line from pairs list
        word1=pairs[i][0]
        # word2 gets the second column in each line from pairs list
        word2=pairs[i][1]

        #word1emb and word2emb gets the embedding that is found by
        word1emb=(H.find_emb(word1,choice))
        word2emb=(H.find_emb(word2,choice))

        # Check if word1emb or word2emb is not found
        if word1emb is None or word2emb is None:
            continue

        # Formula to find cosine distance between both word embeddings
        cosine_distance = np.dot(word1emb, word2emb)/(np.linalg.norm(word1emb)*
np.linalg.norm(word2emb))
        # Stop timer for every iteration
        end = time.perf_counter()

        # Add each timed iteration of finding similaties to "timer"
        timer += end - start

        print('Similarity ['+word1+','+word2+'] =',str(round(100*cosine_distance,2)))

    print('\n\nRunning time for similarities: '+ str(round(timer,4)))

if __name__=="__main__":
    select=int(input('Press 1 for Chaining and 2 for Linear Probing: '))
    if select == 1:
        H=HashTableC_Main()
    if select == 2:
        H=HashTableLP_Main()
```