Lab 5 : Computing Similarities Trough Hash Tables and Binary Search Trees

Steven Robles *

University of Texas, El Paso, TX

April 4, 2019

This paper aims to demonstrate implementations of a binary tree and a hash table and compare the performances of both. The data being stored in both data structures is parsed from a huge file. From this information, the data needs to be requested and computed, all of which are timed to evaluate the algorithm's performance.

Contents

I. Introduction

Binary search trees and hash tables make up the most essential parts of data structures. They are both designed to sort data while they being stored within the structure. Of coarse, they performance will vary depending on the layout of the algorithm and the size of the file. The lab exits to demonstrate and evaluate both data structures while parsing, handling, and computing large amounts of data.

To deploy an accurate analysis of both structures, every function and tasks performed by each data structure is timed and recorded for later comparison. Additional statistics pertaining to the each individual structure such as tree height and load factor will be calculated in order to get a full grasp on its preforming. The computation of the similarities of two data points (words in this case) will also be displayed which shows how accurate the function is able to retrieve the information under a specific certain time.

*ID Number: 80678755

II. Design

This lab is based upon a study that calculated the similarities between two words based upon their numeric embeddings. The scale ranged form -1 to 1 (least similar to most similar). Since the initial file which contains dictionary of words utilized for this lab was large, an initial parsing operation must be preformed. This took some challenge as the file contains 400,000 words with 50 embeddings each. This operations took O(n^2) time.

A. Part 1: Binary Search Tree

With the file already already parsed and its contents stored in an array, a binary search tree (BST) ban be constructed to store the data for easy retrieval upon request. One of the challenges that appeared when implementing this structure is how the data is going to be stored within the tree. Since the program is based on python, the final solution was set on comparing the values of the word characters as strings. This process took O(log n) time.

Additional functions such as calculating the amount of nodes created and the height of the tree also increased the over all time preformance of the program. Each of them contributed *O(log n* time as well, although more efficient solutions could have been implemented in their place.

B. Parts 2: Hash Table

The hash table data structure was harder to implement compared to that of the binary search tree. The hast table initially stats at a certain size. In this lab, the size of the hash table start 991 *buckets*. It doubles in size every time the load factor reaches 1 but since the hash table is handling a large set of data, it is very inefficient to calculate the load factor of the table every time a new item is inserted. This problem was overcome by tracking how many items is inserted into the table. Clearly not every bucket might be occupied when the lists doubles in size, but it gives an almost an accurate solution which doesn't involve increasing the time execution.

The hash table receives an item, which consists of a words attached to its 50 embeddings, and stores the item in the hash table according the the value of its last character. The average time complexity for this function is *O(n)*, and may increase when it doubles in size. Additional challenges that appeared when implementing this structure was calculating its standard deviation according to its buckets. A formula needed to be created which was able to effectively count all of the times within a bucket and disregard those with no items at all.

III. Results

After the implementations of both data structure types were successfully, each functions were executed and recorded for time performance. The time values were used to evaluate each program and compares each data structure with one another. Below are their time executions and over all statistical results.

Figure 1: BST Statistics

```
Binary Search Tree Stats:
Number of Nodes:  400000
Height of Tree :  53
Running Time For Binary Search Tree Consturctoin :  7.377795627020532  Seconds
```

Figure 2: BST Time Performances

```
Average Runnig Time For Binary Search Tree Qeury Processing :  2.324043113427857
5e-05  Seconds
Total Time :  0.0006972129340283573  Seconds
```

Figure 3: Hasth Table Statistics

```
Hash Table Stats :
Initial Table Size : 991
Final Table Size :  507392
Load Factor :  0.7883451059535822
Percentage of Empty Lists :  21.165489404641775
Standard Deviation of the Lengths of the Lists:  18818.258843421572
Number of Items:  400000
```

Figure 4: Hash Table Time Performances

```
Average Runnig Time For Hash Table Qeury Processing :  2.1644867956638336e-05  S
econds
Total Time :  0.0006493460386991501  Seconds

Average Search Time :  0.00026499936357140543  Seconds
Total Time :  0.007949980907142162  Seconds
```

Figure 5: Similarity Computation Outputs From Both Programs

```
Similarity  ['bear', 'bear']  :  1.0
Similarity  ['barley', 'shrimp']  :  0.5353
Similarity  ['barley', 'oat']  :  0.6696
Similarity  ['federer', 'baseball']  :  0.287
Similarity  ['federer', 'tennis']  :  0.7168
Similarity  ['harvard', 'stanford']  :  0.8466
Similarity  ['harvard', 'utep']  :  0.0684
Similarity  ['harvard', 'ant']  :  -0.0267
Similarity  ['raven', 'crow']  :  0.615
Similarity  ['raven', 'whale']  :  0.3291
Similarity  ['spain', 'france']  :  0.7909
Similarity  ['spain', 'mexico']  :  0.7514
Similarity  ['mexico', 'france']  :  0.5478
Similarity  ['mexico', 'guatemala']  :  0.8114
Similarity  ['computer', 'platypus']  :  -0.1277
Similarity  ['computer', 'ant']  :  0.1001
```

IV. Discussion

Overall, implementing both a binary tree and a hash table was a success and was able to handle, store, and retrieve data from the large file provided. The hash table did out preform the binary search tree in terms of executing in the least time, but both implementations ended in the same exact results when computing similarities.

V. Source Code

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 04/01/2019
Discreption: Lab 5:
        The purpse of this program is to act as the main file of two other porgrams that build a hash table and
        a binary tree. Each function is called and excecuted upon prompt request

"""

from Hashtable import hashTable
from B_Tree import treeBuilder
from Parse import parseText

loop = True

#the following parses the file before hand
arr1, arr2 = parseText()

#the is the while loop which pompts the user.
while loop:

  print("Which would rather choose?")
  print("1. Binary Search Tree")
  print("2. Hash Table")
  number = input("3. Exit\n")

  #trys converting the input into an int. if it fails, the pormpt runs again
  try:
    choice = int(number)
  except:
    choice = -1

  #has table runs
  if choice == 1:
    treeBuilder(arr1, arr2)

  #binary tree runs
  elif choice == 2:
    hashTable(arr1, arr2)

  #program exits
  elif choice == 3:
    print("Good Bye!")
    loop = False
  else:
    print("Try Again")
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 04/01/2019
Discreption: Lab 5:
        The purpse of this program is to construct a binary search tree based on the parse file from
        the text. This binary tree will then search for key words anc compute its similarties. It's
        preformance will be times and compared to that of the hash table

"""

from Parse import parseText
```

```
import math
import timeit

class BST(object):
    # Constructor
    def __init__(self, item, LIn, left=None, right=None):
        self.item = item
        self.L = LIn
        self.left = left
        self.right = right

#inserss the item into the bst accoriding to alphebetical order
def Insert(T,newItem, listIn):
    if T == None:
        T =  BST(newItem, listIn)
    elif T.item > newItem:
        T.left = Insert(T.left,newItem, listIn)
    else:
        T.right = Insert(T.right,newItem, listIn)
    return T

#returns the number of items in a BST
def CountItems (T):
    if T is not None:
        count = 1
        count += CountItems(T.right)
        count += CountItems(T.left)
        return count
    return 0

#searches key word within the tree
def Search(T, k):
    while T is not None:
        if T.item == k:
            return T
        if T.item < k:
            T = T.right
        elif T.item > k :
            T = T.left
    return T

#computes the similarity between two words using its embeddings
def sim(w0, w1):
    dotSum, sumMag0, sumMag1 = 0.0, 0.0, 0.0
    for i in range(50):
        dotSum += w0.L[i] * w1.L[i]
        sumMag0 += w0.L[i] * w0.L[i]
        sumMag1 += w1.L[i] * w1.L[i]
    sumMag0 = math.sqrt(sumMag0)
    sumMag1 = math.sqrt(sumMag1)
    return round(dotSum/(sumMag0*sumMag1), 4)

#computes the height of the tree
def Height(T):
    if T is None:
        return 0

    else:
        leftH = Height(T.left)
        rightH = Height(T.right)
        if (leftH > rightH): #retuns only the biggest count form the left and right sub trees
            return leftH+1
        else:
            return rightH+1

#main function called from the program
def treeBuilder(arr, samples):
    T = None
    embed = []
```

```
    simCount = []
    time = []
    nodeCount = 0

    #constructs the tree by inserting nodes
    start = timeit.default_timer()
    for i in range(len(arr[0][0])):
        T = Insert(T, arr[0][0][i], arr[1][0][i])
        nodeCount +=1
    stop = timeit.default_timer()
    constructionT = stop - start

    #retrieves the embeddings of the key words by searching the binary tree
    for i in range(len(samples)):
        embed.append([Search(T, samples[i][0]), Search(T, samples[i][1])])

    #computes the similarity between the words by using its embedding information
    for i in range(len(embed)):
        start = timeit.default_timer()
        simCount.append(sim(embed[i][0], embed[i][1]))
        stop = timeit.default_timer()
        time.append(stop-start)

    #from here one, the results are printed
    print("Binary Search Tree Stats: ")
    print("Number of Nodes: ", nodeCount)
    print("Height of Tree : ", Height(T))
    print("Running Time For Binary Search Tree Consturctoin : ", constructionT, " Seconds")
    print()
    print("Reading Word Files to Determine Similarities")
    print()

    for i in range(len(simCount)):
        print("Similarity ",samples[i] ," : ", simCount[i])
    print()

    sums = 0
    for i in range(len(time)):
        sums += time[i]
    avg = sums/len(time)

    print("Average Runnig Time For Binary Search Tree Qeury Processing : ", avg, " Seconds")
    print("Total Time : ", sums , " Seconds")
    print()
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 04/01/2019
Discreption: Lab 5:
        The purpse of this program is to construct a hash table which stores the information of the
        parsed file. The size of the table doubles by size each time the laod factor reaches one.
        The preformance of this function is kept on record to compare to that of the binary search
        tree implemitation.

"""
from Parse import parseText
import math
import timeit

class HashTableC(object):
    def __init__(self,slots):
        self.item = []
        self.num_items = 0
        self.size = slots
```

```
        for i in range(slots):
            self.item.append([])

#resizes the hash table by doubling its current size if the load factor reaches 1
def reSize(H):
   if H.num_items == H.size:
        for i in range(H.size):
            H.item.append([])
        H.item.append([])
        H.size *=2

#insersts element onto hash table
def InsertC(H,k,l):
    reSize(H)
    b = h(k,len(H.item))
    H.item[b].append([k,l])
    H.num_items +=1

#finds element within the hash table and returns its embeddings
def FindC(H,k):
    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][:]
    return b, -1, -1

#computes the bucket location in which the item will be stored in
def h(s,n):
    r = 0
    for c in s:
        r = (r*n + ord(c))% n
    return r

#computes the similarity between two words based on their embeddingss
def sim(w0, w1):
    dotSum, sumMag0, sumMag1 = 0.0, 0.0, 0.0
    for i in range(50):
        dotSum += w0[1][i] * w1[1][i]
        sumMag0 += w0[1][i] * w0[1][i]
        sumMag1 += w1[1][i] * w1[1][i]
    sumMag0 = math.sqrt(sumMag0)
    sumMag1 = math.sqrt(sumMag1)
    return round(dotSum/(sumMag0*sumMag1), 4)

#computes the sandard deviation of the buckets of the hash table
def standardDev(H):
    mean = 0
    for i in range(len(H.item)):
        if H.item[i] is not None:
            mean += len(H.item[i])
    mean = mean /len(H.item)
    dist = 0
    for i in range(len(H.item)):
        if H.item[i] is not None:
            dist += abs(len(H.item[i]) - mean) * abs(len(H.item[i]) - mean)
    return dist/len(H.item)

#function that is called from the main program
def hashTable(arr, samples):

    embed = []
    simCount = []
    time = []
    searchTime = []
    H = HashTableC(991)

    #fills the hash table with the parsed items from the text file
    for i in range(len(arr[0][0])):
        InsertC(H, arr[0][0][i], arr[1][0][i])

```

```python
    #finds the embeddings of the key words that are going to be compared
    for i in range(len(samples)):
        start = timeit.default_timer()
        embed.append([FindC(H, samples[i][0]), FindC(H, samples[i][1])])
        stop = timeit.default_timer()
        searchTime.append(stop-start)

    #calculates the similarity and times the time it takes
    for i in range(len(embed)):
        start = timeit.default_timer()
        simCount.append(sim(embed[i][0], embed[i][1]))
        stop = timeit.default_timer()
        time.append(stop-start)

    #from here on out, the resutls are displayed
    print("Hash Table Stats : ")
    print("Initial Table Size : 991")
    print("Final Table Size : ", H.size)
    print("Load Factor : " , H.num_items/H.size)
    print("Percentage of Empty Lists : ", ((H.size -H.num_items)/H.size)*100)
    print("Standard Deviation of the Lengths of the Lists: ", standardDev(H))
    print()

    print("Reading Word File to Determine Similarities")
    print()

    for i in range(len(simCount)):
        print("Similarity ", samples[i] ," : ", simCount[i])
    print()

    sums1, sums2 = 0, 0
    for i in range(len(time)):
        sums1 += time[i]
    avg = sums1/len(time)

    for i in range(len(searchTime)):
        sums2 += searchTime[i]
    avg2 = sums2/len(searchTime)

    print("Average Runnig Time For Hash Table Qeury Processing : ", avg, " Seconds")
    print("Total Time : ", sums1 , " Seconds")
    print()
    print("Average Search Time : ", avg2, " Seconds")
    print("Total Time : ", sums2 , " Seconds")
    print()
```

```python
def parseText():
  f = open('glove.6B.50d.txt',encoding='utf-8')

  lines = f.readlines()
  for line in f:
    lines = f.readlines()

  glue = [[[]] for x in range(2)]

  for i in range(len(lines)):
    test = lines[i].split()
    glue[0][0].append(test[0])
    for j in range(50):
      test[j+1] = float(test[j+1])
    glue[1][0].append(test[1:])



  w = open('words.txt',encoding='utf-8')

  lines = w.readlines()
  for line in w:
    lines = w.readlines()

```

```python
samples = []

for i in range(len(lines)):
  test = lines[i].split()
  samples.append(test)

return glue, samples
```

VI. Academic Dishonesty

Scholastic Dishonesty

Any student who commits an act of scholastic dishonesty is subject to discipline. Scholastic dishonesty includes, but not limited to cheating, plagiarism, collusion, the submission for credit of any work or materials that are attributable to another person.

- **Cheating**
    - Copying form the test paper of another student
    - Communicating with another student during a test
    - Giving or seeking aid from another student during a test
    - Possession and/or use of unauthorized materials during tests (i.e. Crib notes, class notes, books, etc)
    - Substituting for another person to take a test
    - Falsifying research data, reports, academic work offered for credit
- **Plagiarism**
    - Using someone's work in your assignments without the proper citations
    - Submitting the same paper or assignment from a different course, without direct permission of instructors
- **Collusion**
    - Unauthorized collaboration with another person in preparing academic assignments

Sign: ____________________ **Date:** **04/04/2019**