CS 2302 Data Structures

Fall 2019

Lab Report #4

Due: October 23th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

Introduction

For this lab we were tasked with creating either a binary search tree or a B-tree depending on what the user whishes to use. After the type of tree was specified, we needed to gather all words without special characters from the file "glove.6B.50b.txt" and create Word-Embedding object to then add into the trees. Once the tree was completed it would output statistics of the tree then we could see how long the tree takes to display the vector differences of two different words as well as the running time needed to do so.

Proposed Solution Design and Implementation

Part 1:

First, I created a prompt to ask the user what tree they would like to use as well as ask for the max number of items to store per node if they chose a B-tree.

Part 2:

Next, I created two more files called "**BTree.py**" and "**BSTree.py**" to create the tree chosen. Within each file there was a class objects that imitated the tree type as well as stored data such as children for a B-tree and left/right for a BST, as well as multiple methods needed to do certain things such as find the height of the tree or the number of nodes within the tree. Then I went line by line in the file "**glove.6B.50b.txt**". I would split the file into two items using the method **lineSplit()**. The first item being returned was a string of the word called "word" and the second was call "emb" which was a float list that contained the words embeddings. After that the word would be checked with the "**.isalpha()**" method to see if it is free of any special characters and if so would be pushed into a Word-Embedding object and placed into the assigned tree type. Once all three hundred thousand plus items were added into the tree it would print out the tree's statistics such as the number of nodes in the tree and the height of the tree. Along with the time taken to create the tree.

Part 3:

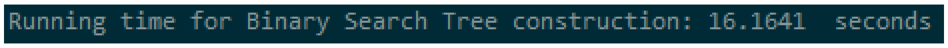
Then, after the tree was created and filled with all the Word-Embedding objects the program would read a new file containing two words per line to compare and print out their similarity. It would split the line into a list of the two words and search through the tree created to find the words embeddings. Next, I would push the two word embedding objects of the words through a method called "**Similarity(),**" which would them computer the two words dot product and divided it by the words vector magnitude. It would then print out the two words and their computed similarity. It would repeat this until all line in the file were compared.

Experimental Results

For both tree’s I ran the program with four test files. The first comparing 15 words, the second comparing 60 words, the third comparing 120 words, and the last comparing 240 words. The first 15 words are the words given to use from our lab instructions and the rest were chosen using a random word generator.

Binary Search Tree:

Time Taken to create the Tree: On average 16.1203 Seconds

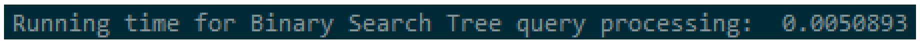
- 

```
Running time for Binary Search Tree construction: 16.1641  seconds
```

Test 1:

List Length: 15 words

Time Taken to Compute Similarity’s:

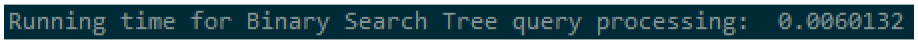
- 

```
Running time for Binary Search Tree query processing:  0.0050893
```

Test 2:

List Length: 60 words

Time Taken to Compute Similarity’s:

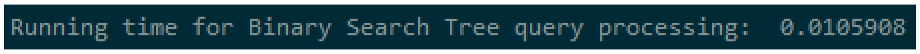
- 

```
Running time for Binary Search Tree query processing:  0.0060132
```

Test 3:

List Length: 120 words

Time Taken to Compute Similarity’s:

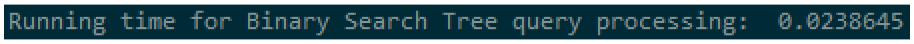
- 

```
Running time for Binary Search Tree query processing:  0.0105908
```

Test 4:

List Length: 240 words

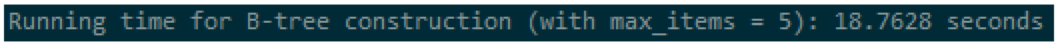
Time Taken to Compute Similarity’:

- 

```
Running time for Binary Search Tree query processing:  0.0238645
```

B-Tree: (Max_items = 5)

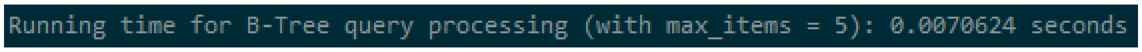
Time Taken to create the Tree: On average 18.6536 Seconds

- 
```
Running time for B-tree construction (with max_items = 5): 18.7628 seconds
```

Test 1:

List Length: 15 words

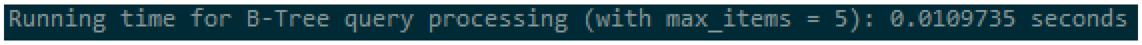
Time Taken to Compute Similarity's:

- 
```
Running time for B-Tree query processing (with max_items = 5): 0.0070624 seconds
```

Test 2:

List Length: 60 words

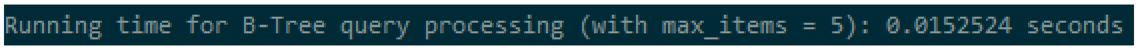
Time Taken to Compute Similarity's:

- 
```
Running time for B-Tree query processing (with max_items = 5): 0.0109735 seconds
```

Test 3:

List Length: 120 words

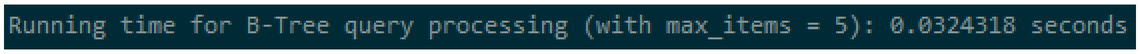
Time Taken to Compute Similarity's:

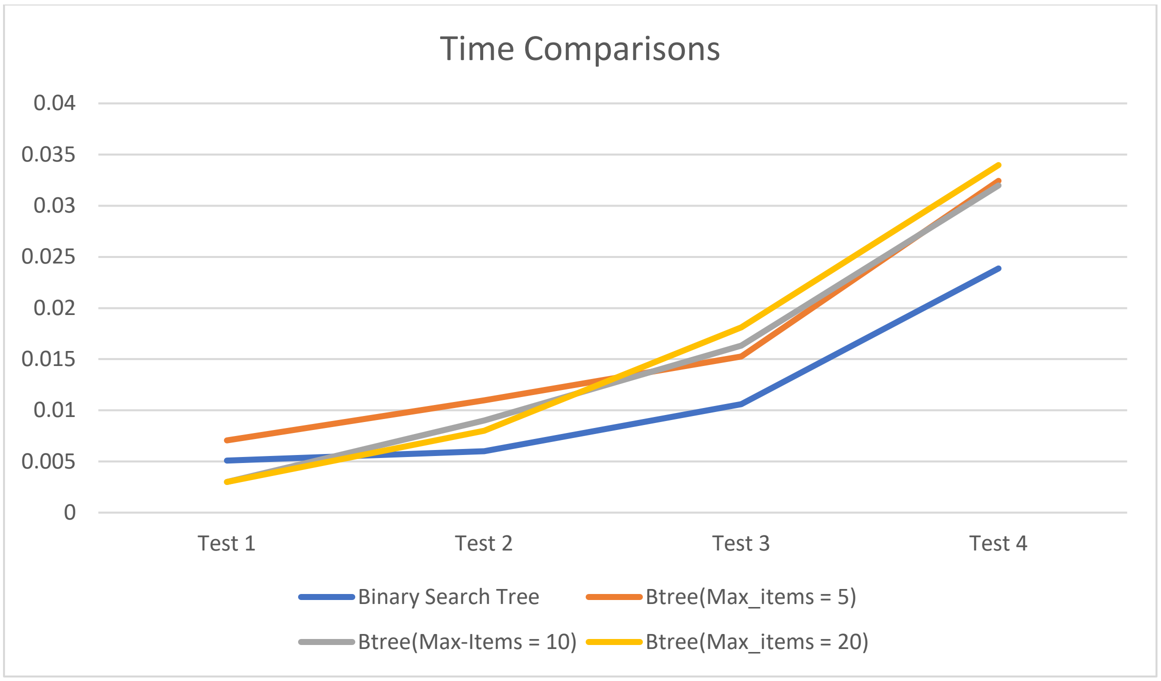
- 
```
Running time for B-Tree query processing (with max_items = 5): 0.0152524 seconds
```

Test 4:

List Length: 240 words

Time Taken to Compute Similarity's:

- 
```
Running time for B-Tree query processing (with max_items = 5): 0.0324318 seconds
```



The data collected helps show that the binary search tree always computes data faster then a b-tree with max inputs of 5. But if the Number of items being compared is moderately low and the max items is high then it has a chance of betting the binary search time as well as the b-tree with max input of 5. This is displayed by the two lines for the b-tree with max items 10, and 20.

Conclusion

In conclusion, I was able to see how in certain instances of searching through trees that b-trees could be faster than binary search trees depending on factors such as the max number of items per node and the number of items being searched for. However, for a more reliable time of completion the binary search tree was the fastest and most consistent.

Appendix

Lab4.py:

```
'''

    Cs2302 Data Structures
    Issac Rivas
    Lab 4
    Dr.Fuentues

'''
import BSTree
import BTree
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 '1' for binary search or '2' for B-tree")
    ans = int(input("Input: "))
    while(ans != 1 and ans != 2):
        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_4/glove.6B.50d.txt'
    testpath = 'C:/Users/Issac/Desktop/Lab_4/test.txt'
    start = t.time()

    #If user wants to create a BST
    if(ans == 1):
        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()
        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))

    #If user wants to create a B-tree
    else:
        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)
        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,
```

BSTree.py:

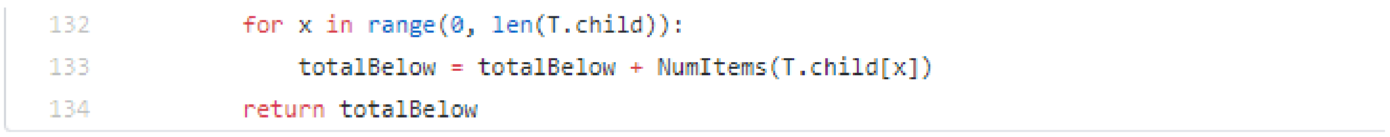
```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 22 19:35:08 2019

@author: Issac
"""
import matplotlib.pyplot as plt
import numpy as np

class BST(object):
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

def Insert(T,newItem):
    if T == None:
        T =  BST(newItem)
    elif T.data.word >= newItem.word:
        T.left = Insert(T.left,newItem)
    else:
        T.right = Insert(T.right,newItem)
    return T

```

```python
def numOfNodes(T):
    if T is None:
        return 0
    return numOfNodes(T.left) + 1 + numOfNodes(T.right)

def treeHeight(T):
    if T != None:
        left = 1 + treeHeight(T.left)
        right = 1 + treeHeight(T.right)
        if left > right:
            return 1 + left
        else:
            return 1 + right
    else:
        return -1

def Search(T, k):
    if T == None:
        return T
    if T.data.word == k:
        return T
    if T.data.word > k:
        return Search(T.left, k)
    else:
        return Search(T.right, k)


def DrawBST_(T, x0, x1, y, y_inc,ax):
    if T is not None:
        xm = (x0+x1)/2
        yn = y-y_inc
        if T.left is not None:
            p=np.array([[xm,y], [(x0+xm)/2,yn]])
            ax.plot(p[:,0],p[:,1],linewidth=1,color='k')
            DrawBST_(T.left,x0,xm,yn, y_inc,ax)
        if T.right is not None:
            p=np.array([[xm,y], [(x1+xm)/2,yn]])
            ax.plot(p[:,0],p[:,1],linewidth=1,color='k')
            DrawBST_(T.right,xm,x1,yn, y_inc,ax)
        ax.text(xm,y, str(T.data.word), size=10,ha="center", va="center",
            bbox=dict(facecolor='w',boxstyle="circle"))

def DrawBST(T):
    fig, ax = plt.subplots()
    DrawBST_(T, 0, 200, 400, 20, ax)
    ax.set_aspect(1.0)
    ax.axis('off')
    plt.show()
```

BTree.py:

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 22 19:18:15 2019

@author: Issac
"""
class BTree(object):
    def __init__(self,data,max_data,child=[],isLeaf=True):
        self.data = data
        self.child = child
        self.isLeaf = isLeaf
        if max_data <3: #max_data must be odd and greater or equal to 3
            max_data = 3
        if max_data%2 == 0: #max_data must be odd and greater or equal to 3
            max_data +=1
        self.max_data = max_data

def Insert(T,i):
    if not IsFull(T):
        InsertInternal(T,i)
    else:
        m, l, r = Split(T)
        T.data =[m]
        T.child = [l,r]
        T.isLeaf = False
        k = FindChild(T,i)
        InsertInternal(T.child[k],i)

def FindChild(T,k):
    # Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree
    for i in range(len(T.data)):
        if k.word < T.data[i].word:
            return i
    return len(T.data)

def FindChildSearch(T,k):
    # Determines value of c, such that k must be in subtree T.child[c], if k is in the BTree
    for i in range(len(T.data)):
```

```python
        if k.word < T.data[i].word:
            return i
    return len(T.data)

def InsertInternal(T,i):
    # T cannot be Full
    if T.isLeaf:
        InsertLeaf(T,i)
    else:
        k = FindChild(T,i)
        if IsFull(T.child[k]):
            m, l, r = Split(T.child[k])
            T.data.insert(k,m)
            T.child[k] = l
            T.child.insert(k+1,r)
            k = FindChild(T,i)
        InsertInternal(T.child[k],i)

def Split(T):
    #print('Splitting')
    #PrintNode(T)
    mid = T.max_data//2
    if T.isLeaf:
        leftChild = BTree(T.data[:mid],max_data=T.max_data)
        rightChild = BTree(T.data[mid+1:],max_data=T.max_data)
    else:
        leftChild = BTree(T.data[:mid],T.max_data,T.child[:mid+1],T.isLeaf)
        rightChild = BTree(T.data[mid+1:],T.max_data,T.child[mid+1:],T.isLeaf)
    return T.data[mid], leftChild,  rightChild

def InsertLeaf(T,i):
    temp = []
    for x in range(0, len(T.data)):
        temp.append(T.data[x].word)
    temp.append(i.word)
    temp.sort()
    for y in range(0,len(temp)):
        if temp[y] == i.word:
            T.data.insert(y, i)
            return

def IsFull(T):
    return len(T.data) >= T.max_data

def Leaves(T):
    # Returns the leaves in a b-tree
    if T.isLeaf:
        return [T.data]
    s = []
    for c in T.child:
        s = s + Leaves(c)
    return s

def Height(T):
    if T.isLeaf:
        return 0
    return 1 + Height(T.child[0])

def PrintD(T,space):
    # Prints data and structure of B-tree
    if T.isLeaf:
        for i in range(len(T.data)-1,-1,-1):
            print(space,T.data[i].word)
    else:
        PrintD(T.child[len(T.data)],space+'   ')
        for i in range(len(T.data)-1,-1,-1):
            print(space,T.data[i].word)
            PrintD(T.child[i],space+'   ')

def Search(T,k):
    # Returns node where k is, or None if k is not in the tree
    for x in range(0, len(T.data)):
        if k.word == T.data[x].word:
            return T.data[x]
    if T.isLeaf:
        return None
    return Search(T.child[FindChildSearch(T,k)],k)

def Set_x(T,Dx):
    # Finds x-coordinate to display each node in the tree
    if T.isLeaf:
        return
    else:
        for c in T.child:
            Set_x(c,Dx)
        d = (Dx[T.child[0].data[0]] + Dx[T.child[-1].data[0]] + 10*len(T.child[-1].data))/2
        Dx[T.data[0]] = d - 10*len(T.data)/2

def NumItems(T):
    if T.isLeaf:
        return len(T.data)
    else:
```



```
        for x in range(0, len(T.child)):
            totalBelow = totalBelow + NumItems(T.child[x])
        return totalBelow
```

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.