Laboratory 4 – Trees

CS 2302 – Data structures Fall 2019

Carlos cardenas

INSTRUCTOR: OLAC FUENTES

Contents

[Introduction 2](#_Toc15996822)

[Implementation 2](#_Toc15996823)

[Experimental results 3](#_Toc15996824)

[Conclusion 7](#_Toc15996825)

[Appendix 7](#_Toc15996826)

[Honesty Certification 14](#_Toc15996827)

# Introduction

This lab makes use of Binary Search Trees and Binary Trees. A type of data structure that stores items in memory, allowing fast lookup, addition and removal of items. In contrast to Binary Trees, these trees have a condition that the items on the left will be smaller than the current one and items in the right will be greater.

Unlike BST, B Trees must have the following properties:

1) All leaves are at same level.

2) A B-Tree is defined by the term minimum degree ‘t’. The value of t depends upon disk block size.

3) Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.

4) All nodes (including root) may contain at most 2t – 1 keys.

5) Number of children of a node is equal to the number of keys in it plus 1.

6) All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.

7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

# Implementation

To implement this lab, both a BST and B Tree must be initialized, so I started by using the given classes to implement the two trees, along with some of their most basic functions that may help along the way.

For both implementations, a recursive height function that adds 1 for every level was implemented. Also, a count to determine the number of elements in each tree, also done recursively.

The next step was to implement the word embeddings class given to us to help with the word search. A main function was used to be used as a menu for the user to work with the program. First, we must ask what kind of implementation they would like to use, BST or B Tree. Should they choose a B Tree, the program would then ask how many keys per node would they want. We then read the desired files containing the word embeddings and our chosen words to compare using simple read functions. We use a split function to separate each word from the file and then call the given Word Embedding constructor to create an object with the given properties. We then use this newly created objects and store them in our desired implementation to create the tree.

After the Tree has been created, our program reads our file with the words we want to compare and stores them in a list. Then, it uses the search function to look for the words in the tree. Finally, it uses the cosine distance formula to determine the similarities between the words.

# Experimental Results

To really test my program, I replicated the words used in the example given to us to see if the output matched what the example had. If my formula or something else was wrong in my code, my program’s output would be different from the example.

The following is the output of my code:

A screenshot of a cell phone

Description automatically generated

This output matched the one in the example perfectly so I knew the code was correct.

Now I tried to compare both implementations to see the different results and try to determine which one was best.

Since the BST implementation would always be the same, I experimented with the max values of keys in the B Tree to see the differences in ouput:

With 1 key per node:

A screenshot of a cell phone

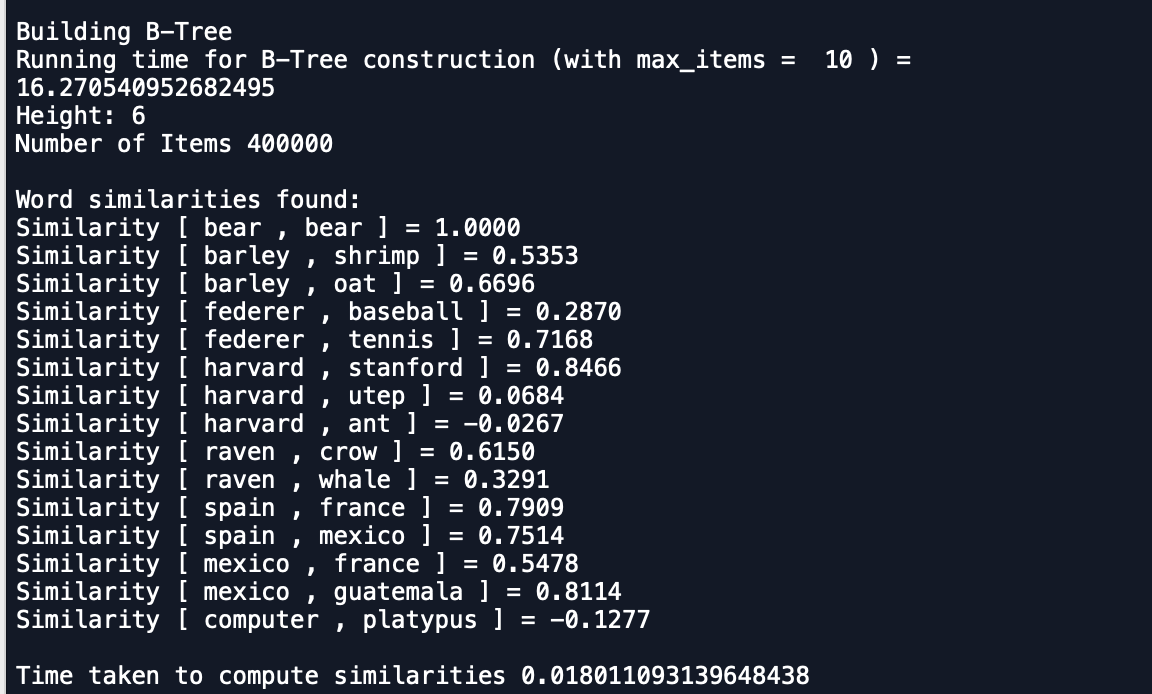
Description automatically generated

With 5 keys per node:

A screenshot of a cell phone

Description automatically generated

With 11 keys per node:



With 101 keys per node:

A screenshot of a cell phone

Description automatically generated

Comparing the outputs above, we can see that the more keys a B Tree has, the faster it takes to compute the similarities. However, with a greater number of keys, the tree takes longer to build than a BST.

# 

# Conclusion

By doing this lab, I learned how to read files and stores them better. As well as implementing BST and B Trees more successfully. The most challenging part of the lab was to implement the cosine distance formula. I must say that this was a challenging lab, but it was satisfying to complete

I also say that I would prefer to implement a B Tree for this lab, given that it has faster comparisons. Even though it may take a little longer to build the tree. In the real word, thousands of comparisons will be made, so ultimately a B Tree would work best.

# Appendix

﻿

import numpy as np

import time

# B Tree

class BTree(object):

# Constructor

def \_\_init\_\_(self,data,child=[],isLeaf=True,max\_data=5):

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 FindChild(T,k):

for i in range(len(T.data)):

if k.word < T.data[i].word:

return i

return len(T.data)

def Split(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.child[:mid+1],T.isLeaf,max\_data=T.max\_data)

rightChild = BTree(T.data[mid+1:],T.child[mid+1:],T.isLeaf,max\_data=T.max\_data)

return T.data[mid], leftChild, rightChild

def InsertLeaf(T,i):

T.data.append(i)

T.data.sort(key = lambda x: x.word)

def IsFull(T):

return len(T.data) >= T.max\_data

def InsertInternal(T,word\_object):

if T.isLeaf:

InsertLeaf(T,word\_object)

else:

k = FindChild(T,word\_object)

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,word\_object)

InsertInternal(T.child[k],word\_object)

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 PrintD(T,space):

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 NumItems(T):

sum = len(T.data)

for i in T.child:

sum+=NumItems(i)

return sum

def Search(T,k):

for i in range(len(T.data)):

if k.word == T.data[i].word:

return T.data[i]

if T.isLeaf:

return None

return Search(T.child[FindChild(T,k)],k)

def Height(T):

if T.isLeaf:

return 0

return 1 + Height(T.child[0])

# Binary Search Tree

class BST(object):

def \_\_init\_\_(self, data, left=None, right=None):

self.data = data

self.left = left

self.right = right

def InsertBinary(T,newItem):

if T == None:

T = BST(newItem)

elif T.data.word > newItem.word:

T.left = InsertBinary(T.left,newItem)

else:

T.right = InsertBinary(T.right,newItem)

return T

def height(T):

if T == None:

return -1

l = height(T.left)

r = height(T.right)

return 1 + max([l,r])

def items(T):

if T == None:

return 0

leftNum = items(T.left)

rightNum = items(T.right)

return 1 + sum([leftNum,rightNum])

def SearchBST(T,k):

if T == None or T.data.word == k:

return T

elif k < T.data.word:

return SearchBST(T.left,k)

else:

return SearchBST(T.right,k)

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)

if \_\_name\_\_ == "\_\_main\_\_":

menu = 0

while menu <= 2:

print("1. Binary Search Tree")

print("2. B-Tree")

print("3. Exit")

menu = int(input("Choose table implementation\n"))

print()

# B Tree

if menu == 1:

print("Building Binary Search Tree")

BinaryST = None

with open("glove.6B.50d.txt", 'r', encoding='utf-8') as file:

start = time.time()

for line in file:

binary\_list = line.split(" ")

word\_object = WordEmbedding(binary\_list[0],binary\_list[1:])

BinaryST = InsertBinary(BinaryST,word\_object)

end = time.time()

print("Binary Search Tree stats:")

print("Number of nodes",items(BinaryST),"\n")

print("Height:", height(BinaryST))

print("Running time for Binary Search Tree construction:", end-start)

print("Reading word file to determine similarities")

with open("wordpairs.txt","r") as file2:

start2 = time.time()

print("Word similarities found:")

for line2 in file2:

#splits lines

listBST = line2.split(" ")

#removes extra space from words

listBST[1]=listBST[1].strip()

word1 = SearchBST(BinaryST,listBST[0])

word2 = SearchBST(BinaryST,listBST[1])

#cosine distance calculation

cosine\_distance = np.dot(word1.data.emb,word2.data.emb)/(abs(np.linalg.norm(word1.data.emb))\*abs(np.linalg.norm(word2.data.emb)))

print("Similarity [", word1.data.word,",", word2.data.word, "] =",'%.4f'%cosine\_distance)

end2 = time.time()

print("\nTime taken to compute similarities", end2-start2,"\n")

# Binary Search Tree

if menu == 2:

user\_max\_keys = int(input("Enter max keys in B Tree:\n "))

print()

print("Building B-Tree")

T = BTree([], max\_data = user\_max\_keys)

with open("glove.6B.50d.txt", 'r', encoding='utf-8') as file:

start = time.time()

for line in file:

list1 = line.split(" ")

word\_object = WordEmbedding(list1[0],list1[1:])

Insert(T,word\_object)

end = time.time()

print("Running time for B-Tree construction (with max\_items = ",user\_max\_keys, ") = " , end-start)

print("Height:", Height(T))

print("Number of Items",NumItems(T),"\n")

with open("wordpairs.txt","r") as file2:

start2 = time.time()

print("Word similarities found:")

for line in file2:

line = line.strip().split(" ")

firstW = WordEmbedding(line[0])

secondW = WordEmbedding(line[1])

word1 = Search(T,firstW)

word2 = Search(T,secondW)

cosine\_distance = np.dot(word1.emb,word2.emb)/(abs(np.linalg.norm(word1.emb))\*abs(np.linalg.norm(word2.emb)))

print("Similarity [",word1.word,",",word2.word, "] =",'%.4f'%cosine\_distance)

end2 = time.time()

print("\nTime taken to compute similarities", end2-start2,"\n")

if menu > 2:

print("Bye!")

# 

# 

# Honesty Certification

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 provide inappropriate assistance to any student in the class.

11/ 01/ 2019

Carlos Cardenas Date