Jose Escobar

80536060

CS2302 Lab 3 Report

Instructor: Diego Aguirre

**Introduction**

The lab 3 required to implement a method that would read two different text files containing word embeddings, give the user the options to choose either AVL or Red-Black trees to find the similarities between two words and show complete results from the trees like their height and nodes created.

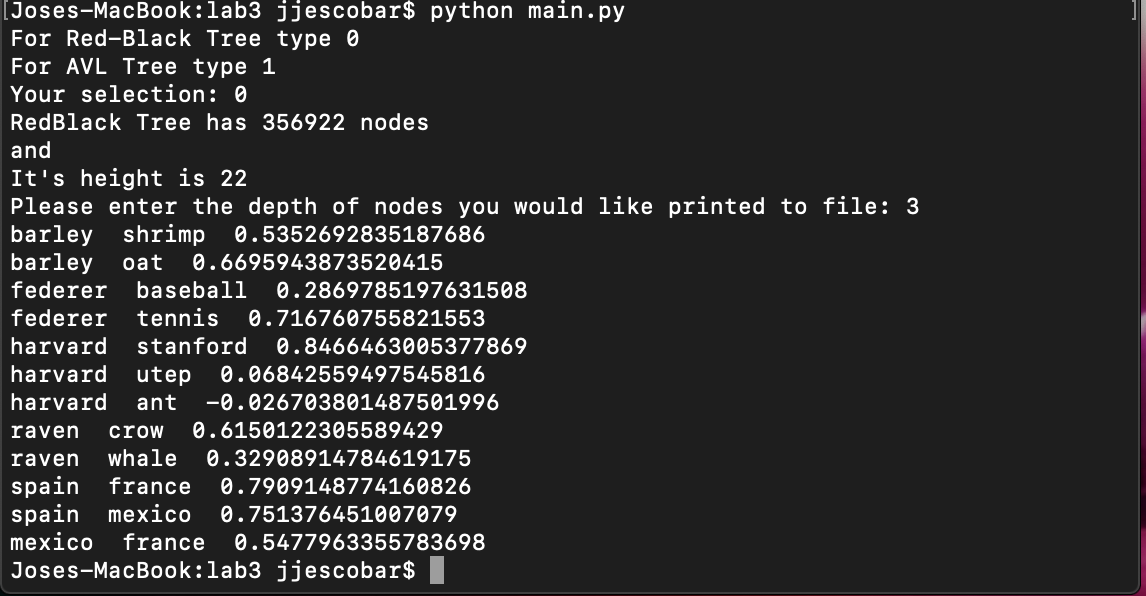
**Proposed Solution**

For this lab since it will have too many methods and implementations, I chose to crate 3 different files which they are still long but for me it was the better way to implement. The first file ‘AVLTree’ contains nodes and methods required to use AVL tree. The second file ‘RedBlackTree’ contains nodes and all methods required to use Red-Black trees. The last file ‘main’ imports the other two files and use them to get the expected results. In addition, we had a text files containing sets of two word embeddings for comparing purposes.

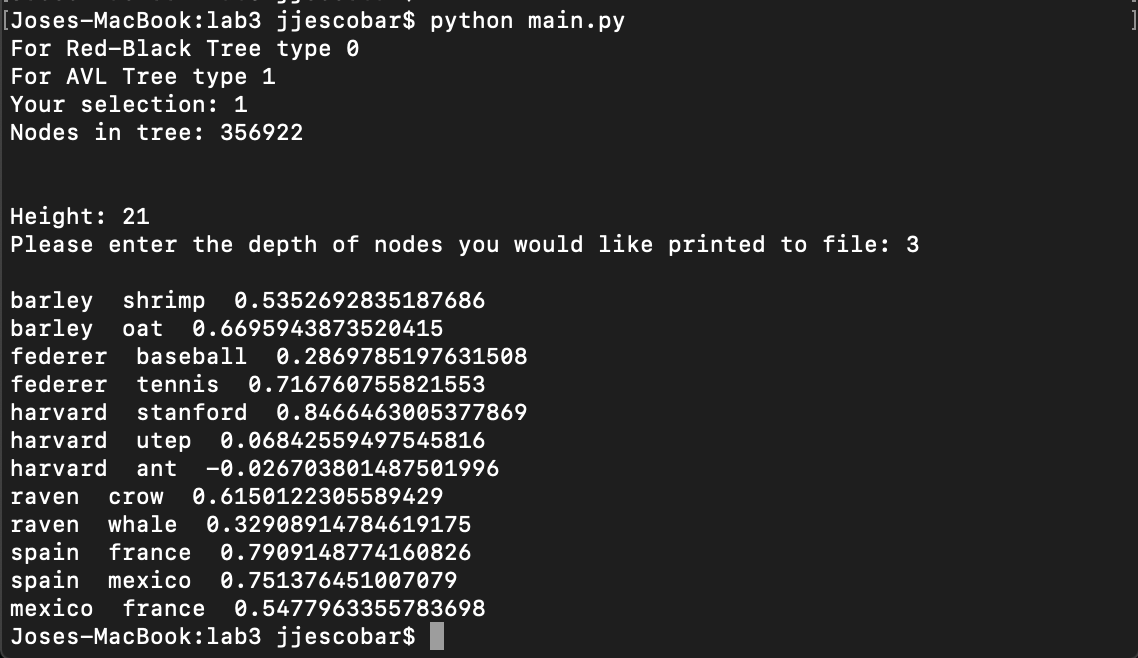
**Experimental Results**

Even though we were able to implement zybooks methods, this lab turned into a hard and stressful one. Many debugging process and changes had to occur for the lab to work as expected.

Using Red-Black Tree



Using AVL Tree



**Conclusion**

This lab challenging but at the end it helped a lot to clear up and give a better understanding of the implementation for AVL and Red-Black trees, which are important because they are often one of the better options in what a running times. Also it made me to clear up the differeces between the two.

\

**Appendix**

**AVLTree file:**

'''

Jose Escobar

UTEP ID 80536060

CS3 Lab 3: Red-Black and AVL Tree implementation

'''

class Node: #AVL Tree Node

def \_\_init\_\_(self, key, embedding):

self.key = key

self.embedding = embedding

self.parent = None

self.left = None

self.right = None

self.height = 0

def get\_balance(self):

left\_height = -1

if self.left is not None:

left\_height = self.left.height

right\_height = -1

if self.right is not None:

right\_height = self.right.height

return left\_height - right\_height

def update\_height(self):

left\_height = -1

if self.left is not None:

left\_height = self.left.height

right\_height = -1

if self.right is not None:

right\_height = self.right.height

self.height = max(left\_height, right\_height) + 1

def set\_child(self, which\_child, child):

if which\_child != "left" and which\_child != "right":

return False

if which\_child == "left":

self.left = child

else:

self.right = child

if child is not None:

child.parent = self

self.update\_height()

return True

def replace\_child(self, current\_child, new\_child):

if self.left is current\_child:

return self.set\_child("left", new\_child)

elif self.right is current\_child:

return self.set\_child("right", new\_child)

return False

def get\_embedding(self):

if self.embedding is not None:

return self.embedding

def set\_embedding(self, array):

self.embedding = array

class AVLTree: ##Implementation of methods to modify/research AVL Tree

def \_\_init\_\_(self):

self.root = None

def rotate\_left(self, node):

right\_left\_child = node.right.left

if node.parent is not None:

node.parent.replace\_child(node, node.right)

else:

self.root = node.right

self.root.parent = None

node.right.set\_child('left', node)

node.set\_child('right', right\_left\_child)

return node.parent

def rotate\_right(self, node):

left\_right\_child = node.left.right

if node.parent is not None:

node.parent.replace\_child(node, node.left)

else:

self.root = node.left

self.root.parent = None

node.left.set\_child('right', node)

node.set\_child('left', left\_right\_child)

return node.parent

def rebalance(self, node):

node.update\_height()

if node.get\_balance() == -2:

if node.right.get\_balance() == 1:

self.rotate\_right(node.right)

return self.rotate\_left(node)

elif node.get\_balance() == 2:

if node.left.get\_balance() == -1:

self.rotate\_left(node.left)

return self.rotate\_right(node)

return node

def insert(self, node):

if self.root is None:

self.root = node

node.parent = None

else:

current\_node = self.root

while current\_node is not None:

if node.key < current\_node.key:

if current\_node.left is None:

current\_node.left = node

node.parent = current\_node

current\_node = None

else:

current\_node = current\_node.left

else:

if current\_node.right is None:

current\_node.right = node

node.parent = current\_node

current\_node = None

else:

current\_node = current\_node.right

node = node.parent

while node is not None:

self.rebalance(node)

node = node.parent

def \_depth(self, k):

\_dep = self.\_depth\_total(self.root, k)

f=open("AVL\_depth.txt", "a+")

for i in range (len(\_dep)):

f.write(str(\_dep[i]+" \n"))

f.close()

return None

def \_depth\_total(self, node, k):

arr = []

if node is None:

return

if k==0:

arr.append(node.key)

else:

arr = arr + self.\_depth\_total(node.left, k-1)

arr = arr + self.\_depth\_total(node.right, k-1)

return arr

def \_write(self):

\_ascend = self.\_write\_afile(self.root)

f=open("AVL\_tree.txt", "a+", encoding="utf-8")

for i in range (len(\_ascend)):

f.write(str(\_ascend[i])+" \n")

f.close()

return None

def \_write\_afile(self, node):

arr = []

if node:

arr = self.\_write\_afile(node.left)

arr.append(node.key)

arr = arr + self.\_write\_afile(node.right)

return arr

def \_height(self):

return self.\_height\_total(self.root)

def \_height\_total(self, node):

if node is None:

return -1

left\_height = self.\_height\_total(node.left)

right\_height = self.\_height\_total(node.right)

return 1 + max(left\_height, right\_height)

def \_size(self):

return self.\_size\_total(self.root)

def \_size\_total(self, node):

if node is None:

return 0

else:

return self.\_size\_total(node.left) + 1 + self.\_size\_total(node.right)

def search(self, key):

current\_node = self.root

while current\_node is not None:

if current\_node.key == key: return current\_node

elif current\_node.key < key: current\_node = current\_node.right

else: current\_node = current\_node.left

return None

def remove\_key(self, key):

node = self.search(key)

if node is None:

return False

else:

return self.remove\_node(node)

def remove\_node(self, node):

if node is None:

return False

parent = node.parent

# Case 1: Internal node with 2 children

if node.left is not None and node.right is not None:

successor\_node = node.right

while successor\_node.left != None:

successor\_node = successor\_node.left

node.key = successor\_node.key

self.remove\_node(successor\_node)

return True

# Case 2: Root node (with 1 or 0 children)

elif node is self.root:

if node.left is not None:

self.root = node.left

else:

self.root = node.right

if self.root is not None:

self.root.parent = None

return True

# Case 3: Internal with left child only

elif node.left is not None:

parent.replace\_child(node, node.left)

# Case 4: Internal with right child only OR leaf

else:

parent.replace\_child(node, node.right)

node = parent

while node is not None:

self.rebalance(node)

node = node.parent

return True

**RedBlackTree file**

'''

Jose Escobar

UTEP ID 80536060

CS3 Lab 3: Red-Black and AVL Tree implementation

'''

class RBTNode: #Red-Black Tree Node

def \_\_init\_\_(self, key, embedding, parent, is\_red = False, left = None, right = None):

self.key = key

self.embedding = embedding

self.left = left

self.right = right

self.parent = parent

if is\_red:

self.color = "red"

else:

self.color = "black"

def are\_both\_children\_black(self):

if self.left != None and self.left.is\_red():

return False

if self.right != None and self.right.is\_red():

return False

return True

def count(self):

count = 1

if self.left != None:

count = count + self.left.count()

if self.right != None:

count = count + self.right.count()

return count

def get\_grandparent(self):

if self.parent is None:

return None

return self.parent.parent

def get\_predecessor(self):

node = self.left

while node.right is not None:

node = node.right

return node

def get\_sibling(self):

if self.parent is not None:

if self is self.parent.left:

return self.parent.right

return self.parent.left

return None

def get\_uncle(self):

grandparent = self.get\_grandparent()

if grandparent is None:

return None

if grandparent.left is self.parent:

return grandparent.right

return grandparent.left

def is\_black(self):

return self.color == "black"

def is\_red(self):

return self.color == "red"

def replace\_child(self, current\_child, new\_child):

if self.left is current\_child:

return self.set\_child("left", new\_child)

elif self.right is current\_child:

return self.set\_child("right", new\_child)

return False

def set\_child(self, which\_child, child):

if which\_child != "left" and which\_child != "right":

return False

if which\_child == "left":

self.left = child

else:

self.right = child

if child != None:

child.parent = self

return True

def get\_embedding(self):

if self.embedding is not None:

return self.embedding

def set\_embedding(self, array):

self.embedding = array

class RedBlackTree: #Implementation of methods to modify/research Red-Black Tree

def \_\_init\_\_(self):

self.root = None

def \_\_len\_\_(self):

if self.root is None:

return 0

return self.root.count()

def \_bst\_remove(self, key):

node = self.search(key)

self.\_bst\_remove\_node(node)

def \_bst\_remove\_node(self, node):

if node is None:

return

if node.left is not None and node.right is not None:

successor\_node = node.right

while successor\_node.left is not None:

successor\_node = successor\_node.left

successor\_key = successor\_node.key

self.\_bst\_remove\_node(successor\_node)

node.key = successor\_key

elif node is self.root:

if node.left is not None:

self.root = node.left

else:

self.root = node.right

if self.root is not None:

self.root.parent = None

elif node.left is not None:

node.parent.replace\_child(node, node.left)

else:

node.parent.replace\_child(node, node.right)

def \_depth(self, k):

\_dep = self.\_depth\_total(self.root, k)

f=open("RB\_depth.txt", "a+")

for i in range (len(\_dep)):

f.write(str(\_dep[i]+" \n"))

f.close()

return None

def \_depth\_total(self, node, k):

arr = []

if node is None:

return

if k==0:

arr.append(node.key)

else:

arr = arr + self.\_depth\_total(node.left, k-1)

arr = arr + self.\_depth\_total(node.right, k-1)

return arr

def \_height(self):

return self.\_height\_total(self.root)

def \_height\_total(self, node):

if node is None:

return -1

left\_height = self.\_height\_total(node.left)

right\_height = self.\_height\_total(node.right)

return 1 + max(left\_height, right\_height)

def \_write(self):

\_ascend = self.\_write\_afile(self.root)

f=open("RedBlack\_tree.txt", "a+", encoding="utf-8")

for i in range (len(\_ascend)):

f.write(str(\_ascend[i])+" \n")

f.close()

return None

def \_write\_afile(self, node):

arr = []

if node:

arr = self.\_write\_afile(node.left)

arr.append(node.key)

arr = arr + self.\_write\_afile(node.right)

return arr

def insert(self, key, embedding):

new\_node = RBTNode(key, embedding, None, True, None, None)

self.insert\_node(new\_node)

def insert\_node(self, node):

if self.root is None:

self.root = node

else:

current\_node = self.root

while current\_node is not None:

if node.key < current\_node.key:

if current\_node.left is None:

current\_node.set\_child("left", node)

break

else:

current\_node = current\_node.left

else:

if current\_node.right is None:

current\_node.set\_child("right", node)

break

else:

current\_node = current\_node.right

node.color = "red"

self.insertion\_balance(node)

def insertion\_balance(self, node):

if node.parent is None:

node.color = "black"

return

if node.parent.is\_black():

return

parent = node.parent

grandparent = node.get\_grandparent()

uncle = node.get\_uncle()

if uncle is not None and uncle.is\_red():

parent.color = uncle.color = "black"

grandparent.color = "red"

self.insertion\_balance(grandparent)

return

if node is parent.right and parent is grandparent.left:

self.rotate\_left(parent)

node = parent

parent = node.parent

elif node is parent.left and parent is grandparent.right:

self.rotate\_right(parent)

node = parent

parent = node.parent

parent.color = "black"

grandparent.color = "red"

if node is parent.left:

self.rotate\_right(grandparent)

else:

self.rotate\_left(grandparent)

def in\_order(self, visitor\_function):

self.in\_order\_recursive(visitor\_function, self.root)

def in\_order\_recursive(self, visitor\_function, node):

if node is None:

return

self.in\_order\_recursive(visitor\_function, node.left)

visitor\_function(node)

self.in\_order\_recursive(visitor\_function, node.right)

def is\_none\_or\_black(self, node):

if node is None:

return True

return node.is\_black()

def is\_not\_none\_and\_red(self, node):

if node is None:

return False

return node.is\_red()

def prepare\_for\_removal(self, node):

if self.try\_case1(node):

return

sibling = node.get\_sibling()

if self.try\_case2(node, sibling):

sibling = node.get\_sibling()

if self.try\_case3(node, sibling):

return

if self.try\_case4(node, sibling):

return

if self.try\_case5(node, sibling):

sibling = node.get\_sibling()

if self.try\_case6(node, sibling):

sibling = node.get\_sibling()

sibling.color = node.parent.color

node.parent.color = "black"

if node is node.parent.left:

sibling.right.color = "black"

self.rotate\_left(node.parent)

else:

sibling.left.color = "black"

self.rotate\_right(node.parent)

def remove(self, key):

node = self.search(key)

if node is not None:

self.remove\_node(node)

return True

return False

def remove\_node(self, node):

if node.left is not None and node.right is not None:

predecessor\_node = node.get\_predecessor()

predecessor\_key = predecessor\_node.key

self.remove\_node(predecessor\_node)

node.key = predecessor\_key

return

if node.is\_black():

self.prepare\_for\_removal(node)

self.\_bst\_remove(node.key)

if self.root is not None and self.root.is\_red():

self.root.color = "black"

def rotate\_left(self, node):

right\_left\_child = node.right.left

if node.parent != None:

node.parent.replace\_child(node, node.right)

else: # node is root

self.root = node.right

self.root.parent = None

node.right.set\_child("left", node)

node.set\_child("right", right\_left\_child)

def rotate\_right(self, node):

left\_right\_child = node.left.right

if node.parent != None:

node.parent.replace\_child(node, node.left)

else: # node is root

self.root = node.left

self.root.parent = None

node.left.set\_child("right", node)

node.set\_child("left", left\_right\_child)

def search(self, key):

current\_node = self.root

while current\_node is not None:

if current\_node.key == key:

return current\_node

elif key < current\_node.key:

current\_node = current\_node.left

else:

current\_node = current\_node.right

return None\

def try\_case1(self, node):

if node.is\_red() or node.parent is None:

return True

return False # node case 1

def try\_case2(self, node, sibling):

if sibling.is\_red():

node.parent.color = "red"

sibling.color = "black"

if node is node.parent.left:

self.rotate\_left(node.parent)

else:

self.rotate\_right(node.parent)

return True

return False # not case 2

def try\_case3(self, node, sibling):

if node.parent.is\_black() and sibling.are\_both\_children\_black():

sibling.color = "red"

self.prepare\_for\_removal(node.parent)

return True

return False # not case 3

def try\_case4(self, node, sibling):

if node.parent.is\_red() and sibling.are\_both\_children\_black():

node.parent.color = "black"

sibling.color = "red"

return True

return False # not case 4

def try\_case5(self, node, sibling):

if self.is\_not\_none\_and\_red(sibling.left) and self.is\_none\_or\_black(sibling.right) and node is node.parent.left:

sibling.color = "red"

sibling.left.color = "black"

self.rotate\_right(sibling)

return True

return False # not case 5

def try\_case6(self, node, sibling):

if self.is\_none\_or\_black(sibling.left) and self.is\_not\_none\_and\_red(sibling.right) and node is node.parent.right:

sibling.color = "red"

sibling.right.color = "black"

self.rotate\_left(sibling)

return True

return False # not case 6

**Main file**

'''

Jose Escobar

UTEP ID 80536060

CS3 Lab 3: Red-Black and AVL Tree implementation

'''

import AVLTree

import RedBlackTree

import math

from AVLTree import AVLTree

from RedBlackTree import RedBlackTree

from AVLTree import Node

def read\_file(): #Creates a Tree from the given file

f = open('glove.6B.50d.txt', encoding="utf-8")

line = f.readline()

while line:

\_line = line.split(" ")

word = \_line[0]

if word[0].isalpha():

embedding\_array = []

for j in range(1,len(\_line)):

embedding\_array.append(float(\_line[j]))

node = Node(word, embedding\_array)

try:

tree.insert(node)

except:

tree.insert(word,embedding\_array)

line = f.readline()

f.close()

#Main method stars here. User is asked the type of Tree to use and methods are called for implementation

while True:

\_input = input("For Red-Black Tree type 0" + "\n" + "For AVL Tree type 1" + "\n" + "Your selection: ")

if \_input is not '0' and \_input is not '1':

print("Invalid, type 0 or 1" )

continue

else:

break

if \_input is "0": #Red-Black Tree

tree = RedBlackTree()

read\_file()

print("RedBlack Tree has "+ str(len(tree)) + ' nodes')

print('and')

print("It's height is " + str(tree.\_height()))

output\_file = open("RedBlack\_tree.txt", "w+", encoding = 'utf-8')

tree.\_write()

output\_file.close()

while True:

\_inputuser = input("Please enter the depth of nodes you would like printed to file: ")

# Checks if the input is valid for the tree.

if int(\_inputuser) >= int(tree.\_height()) or int(\_inputuser) < 0:

print("Depth is not valid, please choose another depth size" )

continue

else:

break

#Depth of tree is read

k=int(\_inputuser)

depth\_file = open("RB\_depth.txt", "w+", encoding="utf-8")

tree.\_depth(k)

depth\_file.close()

if \_input is "1": #AVL Tree

tree = AVLTree()

read\_file()

print("Nodes in tree: "+ str(tree.\_size()))

print("\n")

print("Height: " + str(tree.\_height()))

output\_file = open("AVL\_tree.txt", "w+", encoding = 'utf-8')

tree.\_write()

output\_file.close()

while True:

\_inputuser = input("Please enter the depth of nodes you would like printed to file: ")

print()

if int(\_inputuser) >= int(tree.\_height()) or int(\_inputuser) < 0:

print("Depth is not valid, please choose another depth size: ")

continue

else:

break

#Depth of tree is read

k=int(\_inputuser)

depth\_file = open("AVL\_depth.txt", "w+", encoding = 'utf-8')

tree.\_depth(k)

depth\_file.close()

f = open('twoWord.txt')

line = f.readline()

while line:

\_line = line.split(" ")

w0 = tree.search(\_line[0])

w1 = tree.search(\_line[1])

if w0 is None or w1 is None:

print('no comparison is found')

else:

dot\_prod = 0

magnitude\_0 = 0

magnitude\_1 = 0

e0 = w0.get\_embedding()

e1 = w1.get\_embedding()

for i in range (len(e0)):

dot\_prod+= e0[i]\*e1[i]

magnitude\_0 += e0[i]\*e0[i]

magnitude\_1 += e1[i]\*e1[i]

magnitude\_0 = math.sqrt(magnitude\_0)

magnitude\_1 = math.sqrt(magnitude\_1)

magnitude\_0 = magnitude\_0 \* magnitude\_1

cosine\_similarity = dot\_prod/magnitude\_0

print(\_line[0],"",\_line[1],"", cosine\_similarity)

line = f.readline()

**Academic Honest 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 provided inappropriate assistance to any student in the class.

Jose Escobar