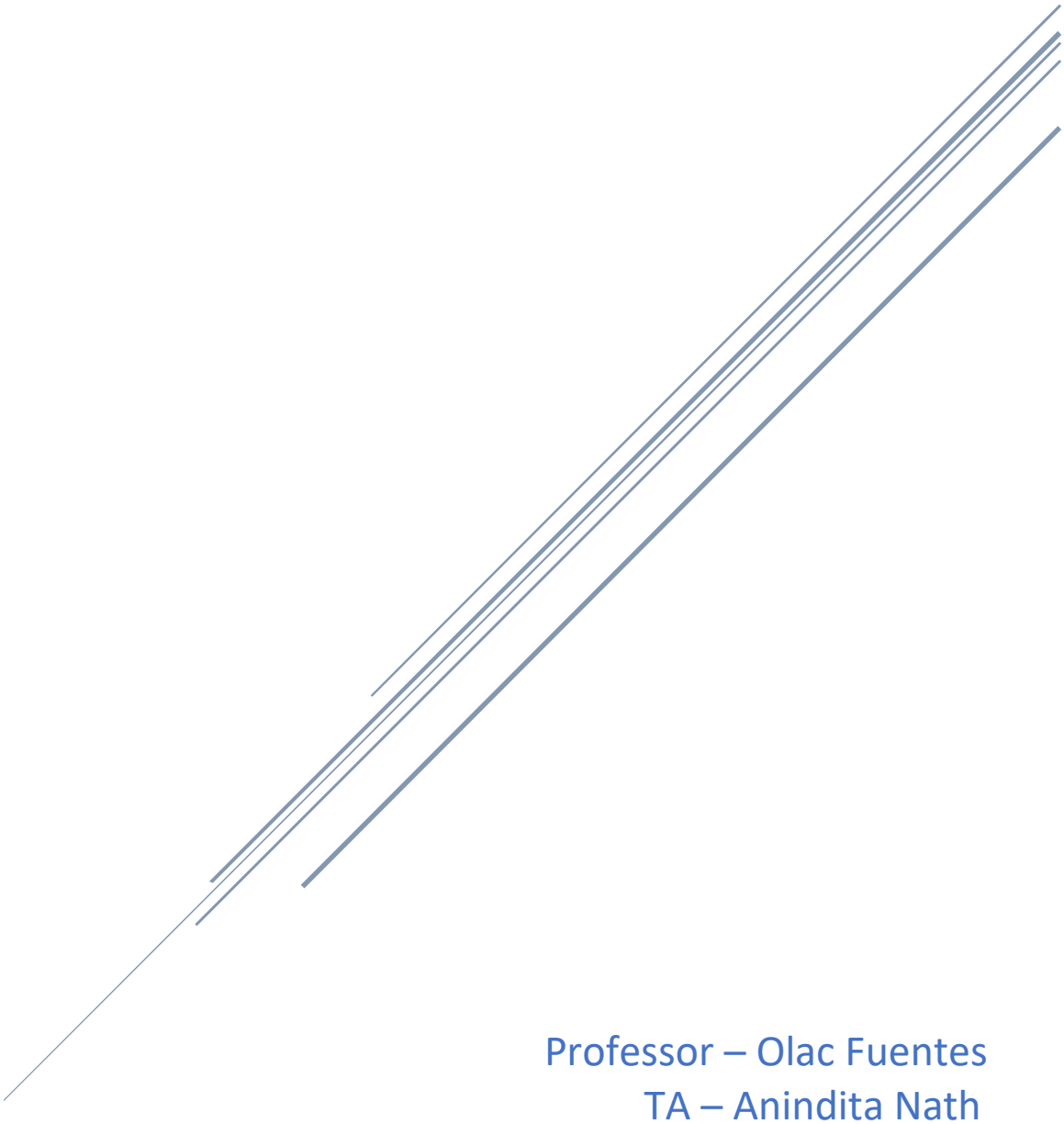
LAB REPORT #4

Sergio Ortiz
October 21, 2019



Professor – Olac Fuentes
TA – Anindita Nath
CS 2302 Data Structures – Fall 2019

Introduction

For this lab, we were asked to implement a simple Natural Language Processing algorithm to be able to compare the similarities between two words. In order to do this, we created a Binary Search Tree and a B – Tree that both contained most of the English words along with their embeddings. The program would use a formula to compute the similarity of the words using these embeddings. Our task was to implement these functions and compare their complexity.

Proposed Solution Design and Implementation

Question 1:

For question 1, I simply decided to create a method that would prompt the user to enter which tree they wanted to use. I decided to use a while loop to keep calling this loop while the user entered erroneous input.

Question 2:

First, I focused on the implementation of the binary search tree. I honestly feel like my implementation goes against the spirit of what the lab intended, but I cooked it up rather quickly and never bothered to go back and change it. I simply added the extra parameters of ‘word’ and ‘embedding’ to the node that the class website implementation already had. Then, I made any necessary changes to the rest of the program. Implementing the B – Tree was more complicated. I created a ‘WordEmbedding’ object and added it to every node instead of a regular int value. Again, this came with a lot of small different changes I had to do to all the program’s functions. Once I had both implementations, I began working on how to add all the words from the file to the trees. The first hurdle was how to separate the word from the embedding, and I decided to create a list of each line separated by the spaces and then simply extract the first element as the word and the rest of the elements as the embedding. This did the job.

Question 3:

The main thing I had to focus on here was how to extract the embeddings of a given word. Once I had this, I would be able to implement this part quite easily. The implementation of this algorithm for the Binary Search Tree was quite simple, simply going either left or right depending on the value of the word until we reach the desired node and then return the embedding of that node. For the B – Tree it was more complicated. I wanted to simply use the already made Search function, but it was used somewhere in the Insert function or something like that and it took the second value as a WordEmbedding object. What this means is that it was being used to search for a WordEmbedding and not for just a singularly word. Therefore, I began working on my own implementation. First, I created an index ‘i’ and initialized it at 0. Then, I added one to this index as long as it was less than the length of the data minus 1 and it was bigger than the value of the word at the current index. From here, I would check if the word at position i was the one we were looking for, returning it if so. Then, I would check if T was a leaf,

returning None if so. Finally, I would recursively call the same function again but this time with the child at position i. The only issue with this was that it would not go to the rightmost child if it had too, so I solved this via a brute force method and added an extra one to i if the value we were searching for was bigger than the value at the end of the list. Once I could extract the embeddings for any word I wanted, I simply applied the formula given in the instructions to compute the similarity.

Question 4:

For this part, I used python's time class and started it before building either tree and ended it after it finished. Then, I also started it before computing the similarities and ended it after it ended. Finally, I displaced the final times by subtracting initial minus final times.

Experimental Results

Question 1:

The easiest part of the lab, I simply created a method that would ask the user what table they wanted and return an integer. If this integer was not 1 or 2, the program would call the function again.

```
t = chooseTable() #to hold user's choice of table
T = None #the actual table
while t != 1 and t != 2:
    print("Please enter 1 or 2")
    t = chooseTable()
```

```
def chooseTable():
    print("Choose table implementation")
    return int(input("Type 1 for binary search tree or 2 B-Tree "))
```

Question 2:

For the binary search tree, I simply added the word embedding parameters to the already existing Node.

```
#I added the word, and embedding here, not sure if that is acceptable
def __init__(self, word, embedding, left=None, right=None):
    self.word = word
    self.emb = np.array(embedding, dtype=np.float32) # For Lab 4, len(e
    self.left = left
    self.right = right
```

Then, I had to make some changes to some of the program's functions such as the Insert function, where I had to add an extra parameter to accommodate for the embedding.

```
def Insert(T, new_word, embedding):
    if T == None:
        T =  BST(new_word, embedding)
    elif T.word > new_word:
        T.left = Insert(T.left,new_word, embedding)
    else:
        T.right = Insert(T.right,new_word, embedding)
    return T
```

For the B – Tree implementation, I created a WordEmbedding class and added it as the data to every node instead of a regular int.

```
class WordEmbedding(object):
    def __init__(self,word,embedding):
        # word must be a string, embedding can be a list or and array
        self.word = word
        self.emb = np.array(embedding, dtype=np.float32) # For Lab 4,
```

```
def Insert(T, word, embedding):
    a = WordEmbedding(word, embedding)
    Insert1(T, a)

#original Insert(), just renamed it
def Insert1(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)
```

As we can see, I created a new insert method that would take advantage of the insert we already had by first creating a WordEmbedding object with the word and embedding and then calling our original insert with the B- Tree and the WordEmbedding object. However, this would cause me problems later.

To add all the words to my trees, I decided to read a line off the file and make it into a list separated by the spaces. This looked something like this.

```
word_file = open("words.txt", "r")
line = word_file.readline().split(' ')
```

The variable line here contains a list containing all the values stored in the first line of the file "words.txt". Then, I would add the first element to either the Search Tree or the B – Tree by passing the first element of the line list as the word and the rest of the list as the embedding.

```
while line[0] != '':
    btree.Insert(T, line[0], np.asarray(line[1:-1]))
    line = f.readline().split(' ')
```

Here we can see a peculiarity with the while statement. I originally had it as simply 'while line', but the split function was returning a small list containing only two colons infinitely after finishing reading a file, so I changed the condition to keep going until it came upon that.

Question 3:

The main thing to complete this question was to create methods that would return an embedding when you imputed a certain string. I did not struggle to do this with the Search Tree as I simply used the already implemented Search function and then returned the embedding of this node.

```
def GetEmbedding(T, k):
    N = Search(T, k)
    if(N is None):
        return None
    return N.emb
```

I tried the exact same approach with the B – tree but I came across an unfortunate realization. Somewhere along the insert method, this search method was used, and the parameter k was taken in as a WordEmbedding object. This was unfortunate for me because I wanted the parameter k to be used as a string literal. Therefore, I had to implement my own different search method.

```
def Search1(T, k):
    i = 0
    n = len(T.data) - 1
    while(i < n and k > T.data[i].word):
        i += 1
    if T.data[i].word == k:
        return T
    if T.isLeaf:
        return None
    if(k > T.data[i].word):
        return Search1(T.child[i+1], k)
    return Search1(T.child[i], k)
```

The only strange thing here is that last if statement, which was placed there because the function would never go to the rightmost child if it had too. I tried messing with the comparative signs in the while loop for a short while but quickly decided on this brute force solution instead. From here, the algorithm to extract the embedding of a word was identical as with the Search Tree.

```
def GetEmbedding(T, k):
    N = Search1(T, k)
    if(N is None):
        return None
    for i in range(len(N.data)):
        if N.data[i].word == k:
            return N.data[i].emb
```

Also, note that I could not simply return the embedding of the node as previously because with the B-tree each node is a list, so I had to traverse this list until I found the proper element and then return the embedding. Once I had the embeddings, I simply applied the formula given in the lab handout.

```
word1 = line[0]
word2 = line[1].strip('\n')
a = btree.GetEmbedding(T, str(word1))
b = btree.GetEmbedding(T, str(word2))
numerator = np.dot(a, b)
den1 = np.linalg.norm(a)
den2 = np.linalg.norm(b)
sim = (numerator) / ((den1) * (den2))
```

Analysis:

I am not sure if this is simply my implementation, but for some reason the b-tree took longer both to build and to find the comparisons than the binary search tree.

```
Building binary search tree

Binary Search Tree Stats:
Number of Nodes:  400000
Height:  4
Running time for binary search tree construction:  22.497337102890015  seconds
```

```
Running time for binary search tree query processing:  0.005983829498291016
seconds
```

```
Building B - tree

B - tree stats:
Number of Nodes:  5
Height:  9
Running time for B-tree construction (with max_items =  5 ):
31.564850091934204  seconds
```

```
Running time for binary search tree query processing:
0.011967182159423828  seconds
```

I feel like perhaps intensive searching are not the main applications of b-trees as they seem sot clearly favor the search tree here. As for how the choice of maximum nodes in the b-tree affected it, less nodes seemed to make the construction take longer and the search only slightly less. When we make the maximum nodes bigger, we seem to get faster comparison times.

```
B - tree stats:
Number of Nodes:  3
Height:  14
Running time for B-tree construction (with max_items =  3 ):
37.9739944934845  seconds
```

```
Running time for binary search tree query processing:  0.010933637619018555
seconds
```

```
Building B - tree

B - tree stats:
Number of Nodes:  11
Height:  6
Running time for B-tree construction (with max_items =  10
): 31.83777379989624  seconds
```

```
Running time for binary search tree query processing:
0.0069820880889892 58  seconds
```

Conclusion

This lab was very fun but a bit challenging for me. I feel like perhaps I did not take the most efficient approach to implement this lab, but along the way I learnt much more about the structure of both data structures, particularly B – trees. I enjoyed finally debugging my code and seeing the data structure unfold before my eyes. It was fun because of the challenge, although I can now comfortably say that since I am finished. Finally, the approach of using embeddings to represent words is very interesting and even efficient. As I always think, if we can expresses any sort of value in terms of numbers, a computer will be able to work with it much faster and more efficiently.

Appendix

```
#CS 2302 Data Structures Fall 2019 MW 10:30
#Sergio Ortiz
#Assignment - Lab #4
#Instructor - Olac Fuentes
#Teaching Assistant - Anindita Nath
#October 21, 2019
#This program will compare the runtimes between
#binary search trees and B-trees
import bst #import the binary search tree implementation
import btree #import the btree implementation
import numpy as np
import time

#BuildBST() - builds the binary search tree
#-->input: None
#-->output: an implemented bst with all the words in the file
def BuildBST():
    T = None
    #I dont know what 'encoding' is but I was getting an error and that
    #fixed it
    f = open("glove.6B.50d.txt", encoding = "utf8")
```

```
    line = f.readline().split(' ') #creates a list, splits at spaces
    while line[0] != '': #for some reason when I reached the end of the
        #file it would keep creating lists with the empty ['']
        T = bst.Insert(T, line[0], np.asarray(line[1:-1]))
        line = f.readline().split(' ')
    f.close()
    return T

#BuildBTree() - builds the B-tree
#-->input: max_node, the maximum data in a node
#-->output: an implemented B-Tree with all the words in the file
def BuildBTree(max_node):
    T = None
    T = btree.BTree([],max_data=max_node)
    f = open("glove.6B.50d.txt", encoding = "utf8" )
    line = f.readline().split(' ')
    while line[0] != '':
        btree.Insert(T, line[0], np.asarray(line[1:-1]))
        line = f.readline().split(' ')
    f.close()
    return T
```

```
#chooseTable() - method which prompts the user to choose
#a table implementation
#-->inputs: None
#-->outputs: an int representing either a bst or a btree
def chooseTable():
    print("Choose table implementation")
    return int(input("Type 1 for binary search tree or 2 B-Tree "))

#main() - main method where everything happens
#-->inputs: None
#-->outputs: None
def main():
    t = chooseTable() #to hold user's choice of table
    T = None #the actual table
    while t != 1 and t != 2:
        print("Please enter 1 or 2")
        t = chooseTable()

    if t == 1:
        print("Choice: ", t)
        print()
        print("Building binary search tree")
```

```
66          print()
67          start = time.time()
68          T = BuildBST()
69          end = time.time()
70          elapsed = end - start
71          print("Binary Search Tree Stats:")
72          print("Number of Nodes: ", bst.GetNumberOfNodes(T))
73          print("Height: ", bst.GetHeight(T))
74          print("Running time for binary search tree construction: ", elapsed," seconds")
75          print()
76          print("Reading word file to determine similarities")
77          print()
78          print("Word Similarities found: ")
79          start = time.time()
80          SimilaritiesBST(T)
81          end = time.time()
82          elapsed = end - start
83          print()
84          print("Running time for binary search tree query processing: ", elapsed," seconds
85      else:
86          print("Choice: ", t)
87          max_node = int(input("What is the maximum number of items"
```

```
88                              " you want to store in a node? "))
89          print()
90          print("Building B - tree")
91          print()
92          start = time.time()
93          T = BuildBTree(max_node)
94          end = time.time()
95          elapsed = end - start
96          print("B - tree stats:")
97          print("Number of Nodes: ", T.nodes)
98          print("Height: ", btree.Height(T))
99          print("Running time for B-tree construction (with max_items = ",max_node,"):", elapsed,'
100         print()
101         print("Reading word file to determine similarities")
102         print()
103         print("Word Similarities found: ")
104         start = time.time()
105         SimilaritiesBTree(T)
106         end = time.time()
107         elapsed = end - start
108         print("Running time for binary search tree query processing: ", elapsed," seconds")
109
```

```
#SimilaritiesBST() - checks the similarities in a file using bst
#-->inputs: T, a bst containing a lot of english words and their embeddings
#-->outputs: None
def SimilaritiesBST(T):
    word_file = open("words.txt", "r")
    line = word_file.readline().split(' ')
    while line[0] != '':
        word1 = line[0]
        word2 = line[1].strip('\n')
        a = bst.GetEmbedding(T, str(word1))
        b = bst.GetEmbedding(T, str(word2))
        numerator = np.dot(a, b)
        den1 = np.linalg.norm(a)
        den2 = np.linalg.norm(b)
        sim = (numerator) / ((den1) * (den2))
        print("Similarity [",str(word1),",",str(word2),"] = ", str(sim))
        line = word_file.readline().split(' ')

#SimilaritiesBTree() - checks the similarities in a file using B - Tree
#-->inputs: T, a B-Tree containing a lot of english words and their embeddings
#-->outputs: None
def SimilaritiesBTree(T):
```

```
    print("Reading word file to determine similarities")
    word_file = open("words.txt", "r")
    line = word_file.readline().split(' ')
    while line[0] != '':
        word1 = line[0]
        word2 = line[1].strip('\n')
        a = btree.GetEmbedding(T, str(word1))
        b = btree.GetEmbedding(T, str(word2))
        numerator = np.dot(a, b)
        den1 = np.linalg.norm(a)
        den2 = np.linalg.norm(b)
        sim = (numerator) / ((den1) * (den2))
        print("Similarity [",str(word1),",",str(word2),"] = ", str(sim))
        line = word_file.readline().split(' ')

if __name__ == "__main__":
     main()
```

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.