CS2302 Data Structures

Fall 2019
Lab Report Number 4

Author: Bryan Ramos
Due: October 21st, 2019
Professor: Dr. Olac Fuentes
TA: Anindita Nath

Introduction

The purpose of this lab assignment was to provide additional practice of the topics we have covered in class by actually modeling the application of binary search trees (BSTs) and B-Trees, which we have covered in previous lectures and in-class quizzes, in a probable real-world scenario. Data structures allow computer scientists to store and manipulate data for different cases, in the real world that means dealing with large amounts of data. Here, we will build BSTs and B-Trees by reading data from a file, in this case from Stanford's GloVe project, to determine similarities between words. The analysis of the running time for building each of the trees as well as performing several operations to manipulate or retrieve data from each of these structures is crucial to determine the advantages of each in further real-world applications.

Proposed Solution Design and Implementation

Part 1 asks that the program ask the user to choose a tree implementation. I implemented a menu expecting valid user input. If the user provides an invalid integer value the menu is displayed until a valid choice is given, if the user enters a non-integer value, the program terminates. Using if conditions, I split the code so that building each implementation occurs with what the user chose. In addition, if the B-Tree option is chosen the user is prompted to enter the maximum number of items to store in a node.

Next, to create the functions to either construct a BST or a B-Tree, the BST and B-Tree classes provided in the class webpage had to be modified for the purposes of the lab. I imported the code provided for the WordEmbedding class provided in the lab instructions. Both tree classes were modified to accept WordEmbedding objects data. The operations within each of these classes were modified to allow access to the new object data.

For **part 2**, to build either tree, first it was necessary to check whether the required file found on the Glove project website

(https://nlp.stanford.edu/projects/glove/) existed, if not, a tree was not to be built. I initially had trouble opening the file, so I did some research on why this might be occurring and found out it had to deal with encoding, the file uses UTF-8 encoding. In either tree implementation, each line of the file is read, being tokenized into a list of strings, for the values found on that line. Now, looking at the data from the glove file, each of the words in the file is to be stored as a word in a WordEmbedding object, and the embeddings that followed as a list in the WordEmbedding object. Well, the word is always the first string in the line. Therefore, my implementation checks to make sure that the first string in the list of tokens for a line begins with a letter. If that's the case, the WordEmbedding object is inserted into the BST. In both implementations, once the file is traversed completely, the file is closed and the built tree implementation is returned to be used further.

In either tree implementation case, it is required that some stats be calculated for the tree such as the number of nodes, tree's height, and running time for the construction. In order to find the number of nodes and the height of the tree some additional methods were added to both the BST and the B-Tree classes, utilizing code that was written during the lectures with Dr. Fuentes, using recursion. Using formatting, the calculated stats are displayed. I created a function called treeType that fills a part of the following output:

```
Running time for {treeType} construction: {running_time}
```

The tree type function checks for the type of tree data structure returning a tree for its classification. I utilized the treeType again later in my design.

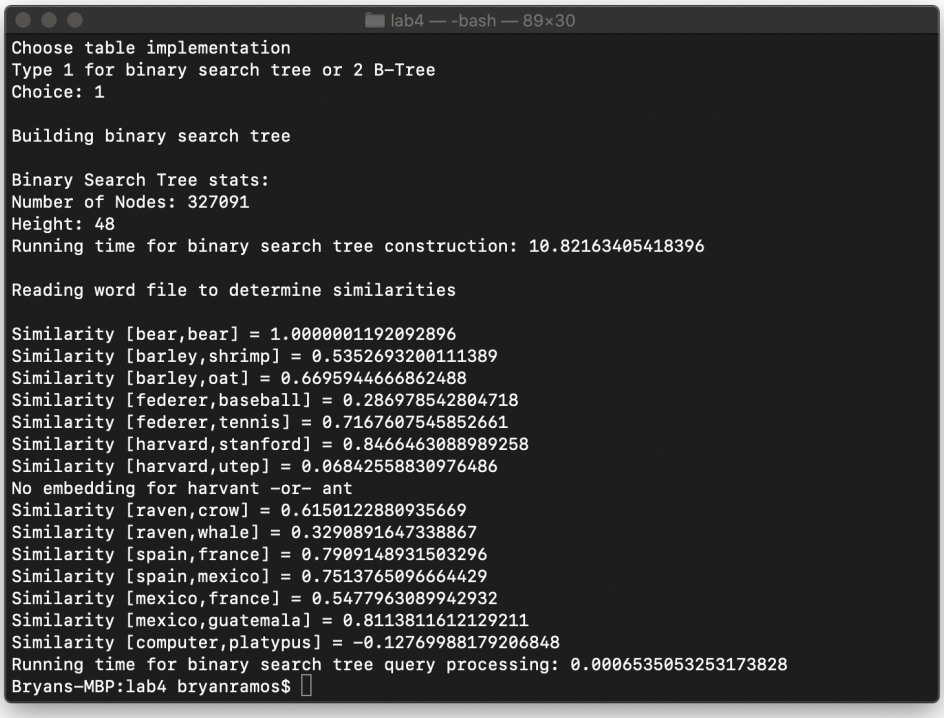
Part 3 asks to compute the "similarity" of words. To do this, it is necessary to read another file containing pairs of words (two words per line), and for each pair of words, find and display the similarity of the words. I utilized an already existing file containing a large amount of English words and manipulated that, saving the changes to a new file. I've written data to a file utilizing other programming languages like Java and C-Sharp but never in Python, so I utilized some guides found on StackOverflow and Geeks for Geeks. In the new file, my program writes two words separated by a comma per line, just like the instructions ask for. Back in the main method, I check to see if the similarities file exists, if not, then the function to write the

similarities the file with two words is called and the similarities file will be created at that point. I imported the OS class in Python to do this.

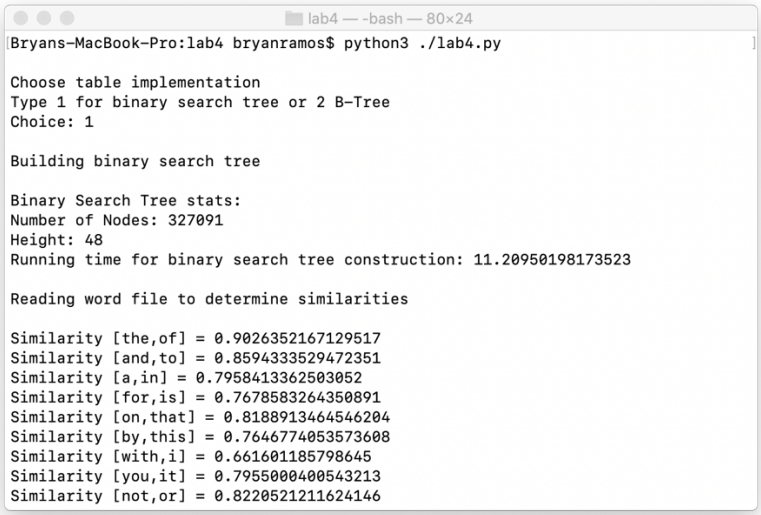
Next, to calculate the similarities I created another method. This method accepts the tree and the file of similarities as parameters. Next, to find words common in the tree and in the similarities file, the similarities file is traversed, reading line by line, the data of each line being converted to strings by splitting so that the strings can actually be utilized. Next, using the little trick I used earlier with looking at the type of the data structure that a variable is, depending on whether the tree was a BST or a B-Tree, I searched for the two words in the tree. If either one of the words is not found, the strings are appended to the list, otherwise a list is appended to the embeddings list. This will be useful when the list is returned and it is then necessary to determine which were similarities and which were not. After each line is read, the runtime is calculated and then added to the total runtime. At the end when all the embeddings are gathered, the runtime is returned along with the list of embeddings. The runtime for the query processing for finding the similarities is outputted to the user.

Experimental Results

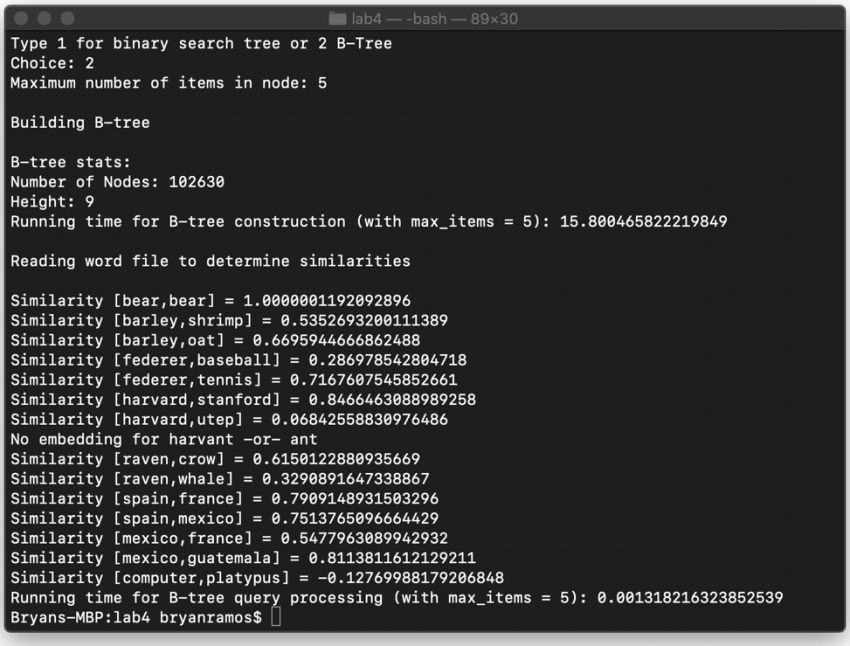
BST using same data as lab instructions:



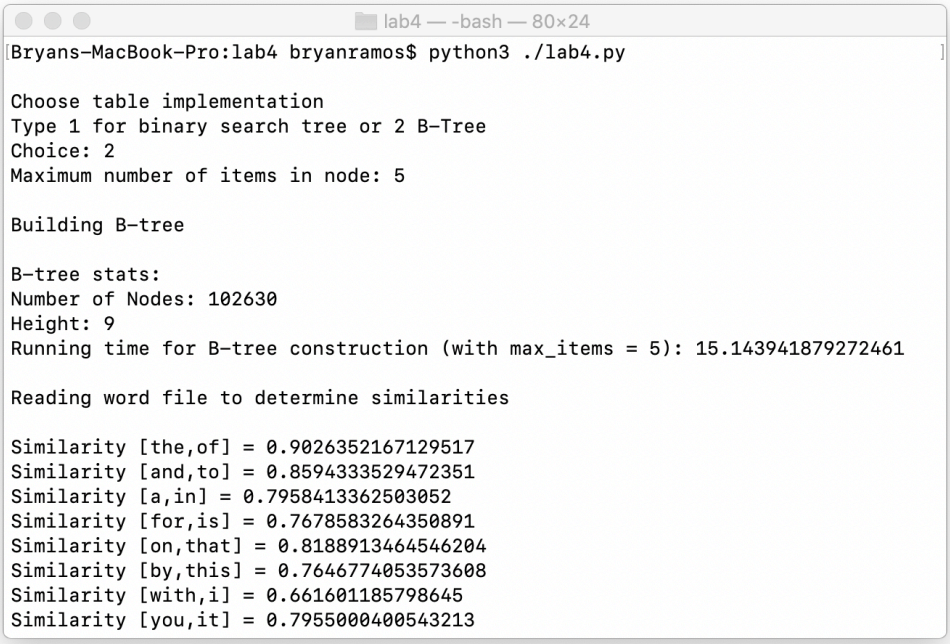
BST with 10,000 English words:



B-Tree using same data as lab instructions:

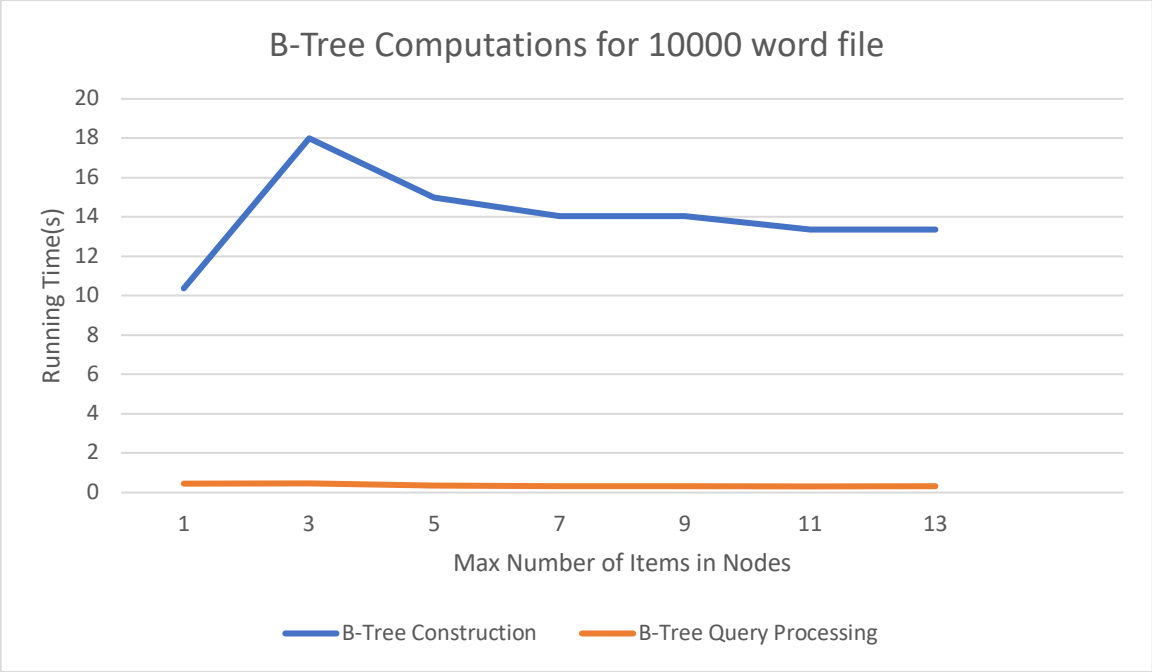


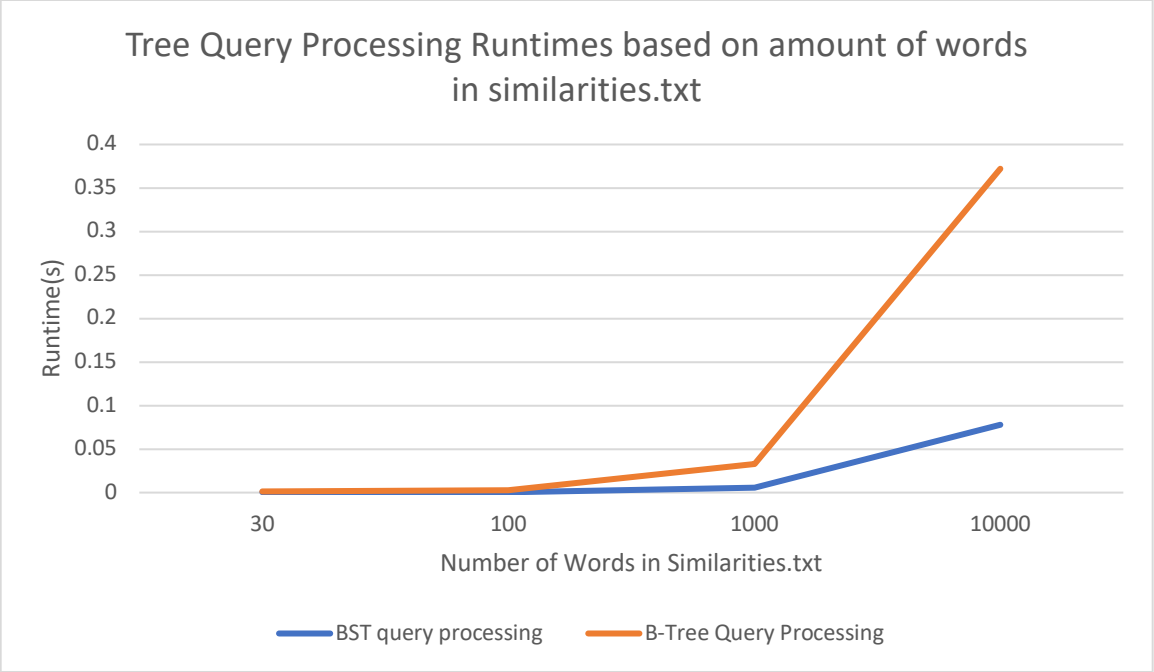
B-Tree with 10000 words:



BST Runtimes for BST Construction

Trial	BST Construction	BST Query Processing
1	10.20391297340393	0.07804036140441895
2	10.15591025352478	0.0892629623413086
3	10.503875017166138	0.08148908615112305
4	10.238296031951904	0.0857858657836914





In order to test the efficiencies of querying through either implementation, a different number of words was added to the similarities.txt. By looking at this table, it's evident that as the number of words in the similarities file increases, the query processing runtime for B-Trees increases more significantly than the query processing for BSTs. As the amount of data grows, B-Trees are generally slower than BSTs. Albeit, both are very fast since their runtimes are fractions of a second, even though we are dealing with somewhat big data.

Conclusion

Generally, the binary search tree implementation outperformed the B-Tree implementation as I expected, evident from looking at the table and graphs. While the B-tree implementation generally has less nodes than the binary search tree implementation (See beginning of Experimental results for screenshots of output), the BST implementation has better efficiency when it comes to building the tree, populating it with word embeddings. Remember that, the data each node can store up to max_data values. As the max number of items that can be stored per node in a B-Tree increases, the runtimes begin to even out and are the same as evidenced by the table. While there is a difference between the running times for building either

tree and query processing through the data for similarities, both implementations are very fast, especially when dealing with the file containing 10000 words, that's a lot of data.

Appendix

```
'''
        Course: CS2302 Data Structures Fall 2019
        Author: Bryan Ramos [88760110]
        Assignment: Lab 4 BSTs and BTrees
        Instructor: Dr. Olac Fuentes
        TA: Anindita Nath
        Last Modified: October 21st 2019
        Purpose: Using both binary search trees and b-trees as deliberate
    practice
        to read data from a file and store in either type of tree and then
    access
        the data from the tree using operations to find the number of nodes,
    similarities
        between words, etc.
    '''

    import time
    import bst
    import btree
    import os
    import numpy as np
    from wordEmbedding import WordEmbedding

    # returns a built BST from the file
    # if file not found, throw exception
    def buildBST(filename):
        try:
            T = None
            # file uses utf8 encoding
            f = open(filename, "r", encoding="utf8")

            for line in f:
                # tokenize each line into a list of strings
                tokens = line.split(" ")
```

```
            # if the value stored at the index begins with an alphabetical
letter
            if tokens[0].isalpha():
                T = bst.Insert(T, tokens[0], tokens[1:])

        f.close() # close the file to save memory
        return T # return bst
    except IOError:
        print("File", filename, "not found!\n")

# returns a built B-tree from the file
# if file not found, throw exception
def buildBTree(max, filename):
    try:
        # set the max value of b tree to the value passed as max parameter
        T = btree.BTree([], max_data = max)
        f = open(filename, "r", encoding="utf8")
        for line in f:
            # tokenize each line into a list of strings
            tokens = line.split(" ")
            # if the value stored at the index begins with an alphabetical
letter
            if tokens[0].isalpha():
                btree.Insert(T, WordEmbedding(tokens[0], tokens[1:]))

        f.close() # close the file to save memory
        return T # return btree
    except IOError:
        print("File", filename, "not found!\n")

def getEmbeddings(T,filename):
    totaltime = 0
    f = open(filename, "r")
    embeddings = []

    for line in f:
        # first remove new line char from line
        if "\n" in line:
            line = line[:-1]
        # split by commas
        words = line.split(",")

        # if its of type binary search tree
```

```
        if type(T) == bst.BST:
            start = time.time()
            first = bst.Search(T, words[0])
            second = bst.Search(T, words[1])

            if first == None or second == None: #if word is not in tree,
just appends the string words
                embeddings.append(words)
            else:
                embeddings.append([first, second])

            totaltime = totaltime + (time.time() - start)
        # if its of type Btree - similar code as the one for BST above
        if type(T) == btree.BTree:
            start = time.time()
            first = btree.Search(T, words[0])
            second = btree.Search(T, words[1])

            if first == None or second == None:
                embeddings.append(words)
            else:
                embeddings.append([first, second])

            totaltime = totaltime + (time.time() - start)
    # return the runtime of either BST or BTree operation and the
embeddings list
    return totaltime, embeddings

def treeType(T):
    if type(T) == bst.BST:
        return "binary search tree"
    if type(T) == btree.BTree:
        return "B-tree"

# get a file of words, create a new file, and write to
# the file having two words per line in the new file
def writeToSimilaritiesFile(filename):
    try:

        # open file to read words from
        # create new file to write to
        f = open(filename)
        new = open("similarities.txt", "w")
```

```
            word = 0
            for line in f:
                if "\n" in line:
                    line = line[:-1]
                if word == 1:
                    new.write(line + "\n")
                    word = 0
                else:
                    new.write(line + ",")
                    word = word + 1

            f.close() # close the files to save memory
            new.close()
        except IOError:
            print("File", filename, "not found!\n")

# part 3
def similarities(e1, e2):
       return np.dot(e1.emb, e2.emb)/(np.linalg.norm(e1.emb) *
np.linalg.norm(e2.emb))

if __name__ == "__main__":

    # txt file from nlp.stanford.edu
    filename = "glove.6B.50d.txt"

    # check if similarities text file exists
    # if not - open a file containing English words and write to a new
file
    # two words per row to be used to find similarities in part 3
    if not os.path.exists("similarities.txt"):
        writeToSimilaritiesFile("english-words.txt")

    # part 1
    while (True):
        c = 0
        try:
            while (c < 1 or c > 2):
                print("\nChoose table implementation\nType 1 for binary
search tree or 2 B-Tree") # menu
                c = int(input("Choice: "))
        except ValueError:
```

```
            print("Invalid input! Provide an integer value.")
            break

        # build bst
        if c == 1:
            print("\nBuilding binary search tree\n")
            start = time.time()
            T = buildBST(filename)
            end = time.time()

            # stats
            print("Binary Search Tree stats:\nNumber of Nodes:
{}".format(bst.NumberOfNodes(T)))
            print("Height: {}".format(bst.Height(T)))
            print("Running time for {} construction:
{}".format(treeType(T), end-start))

        # build b-tree
        if c == 2:
            max = int(input("Maximum number of items in node: "))
            print("\nBuilding B-tree\n")
            start = time.time()
            T = buildBTree(max, filename)
            end = time.time()

            print("B-tree stats:\nNumber of Nodes:
{}".format(btree.NumberOfNodes(T)))
            print("Height: {}".format(btree.Height(T)))
            print("Running time for {} construction (with max_items = {}):
{}".format(treeType(T), max, end-start))

        print("\nReading word file to determine similarities\n")

        totalTime, embeddings = getEmbeddings(T, "similarities.txt" )

        for element in embeddings:
            if any(isinstance(words, str) for words in element):
                print("No embedding for {} -or- {}".format(element[0],
element[1]))
            else:
                print("Similarity [{},{}] = {}".format(element[0].word,
element[1].word, similarities(element[0], element[1])))
```

```
        # final running time output message based on type of tree
        if type(T) == bst.BST:
            print("Running time for {} query processing:
{}".format(treeType(T), totalTime))

        if type(T) == btree.BTree:
            print("Running time for {} query processing (with max_items =
{}): {}".format(treeType(T), max, totalTime))

        break;
```

I, Bryan Ramos, 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.