**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**