**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #4**

Due: October 23th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab we were tasked with creating either a binary search tree or a B-tree depending on what the user whishes to use. After the type of tree was specified, we needed to gather all words without special characters from the file “glove.6B.50b.txt” and create Word-Embedding object to then add into the trees. Once the tree was completed it would output statistics of the tree then we could see how long the tree takes to display the vector differences of two different words as well as the running time needed to do so.

**Proposed Solution Design and Implementation**

**Part 1:**

First, I created a prompt to ask the user what tree they would like to use as well as ask for the max number of items to store per node if they chose a B-tree.

**Part 2:**

Next, I created two more files called “**BTree.py**” and “**BSTree.py**” to create the tree chosen. Within each file there was a class objects that imitated the tree type as well as stored data such as children for a B-tree and left/right for a BST, as well as multiple methods needed to do certain things such as find the height of the tree or the number of nodes within the tree. Then I went line by line in the file “**glove.6B.50b.txt**”. I would split the file into two items using the method **lineSplit()**. The first item being returned was a string of the word called “word” and the second was call “emb” which was a float list that contained the words embeddings. After that the word would be checked with the “**.isalpha()**” method to see if it is free of any special characters and if so would be pushed into a Word-Embedding object and placed into the assigned tree type. Once all three hundred thousand plus items were added into the tree it would print out the tree’s statistics such as the number of nodes in the tree and the height of the tree. Along with the time taken to create the tree.

**Part 3:**

Then, after the tree was created and filled with all the Word-Embedding objects the program would read a new file containing two words per line to compare and print out their similarity. It would split the line into a list of the two words and search through the tree created to find the words embeddings. Next, I would push the two word embedding objects of the words through a method called “**Similarity(),**” which would them computer the two words dot product and divided it by the words vector magnitude. It would then print out the two words and their computed similarity. It would repeat this until all line in the file were compared.

**Experimental Results**

For both tree’s I ran the program with four test files. The first comparing 15 words, the second comparing 60 words, the third comparing 120 words, and the last comparing 240 words. The first 15 words are the words given to use from our lab instructions and the rest were chosen using a random word generator.

**Binary Search Tree:**

Time Taken to create the Tree: On average 16.1203 Seconds

* 

**Test 1:**

List Length: 15 words

Time Taken to Compute Similarity’s:

* 

**Test 2:**

List Length: 60 words

Time Taken to Compute Similarity’s:

* 

**Test 3:**

List Length: 120 words

Time Taken to Compute Similarity’s:

* 

**Test 4:**

List Length: 240 words

Time Taken to Compute Similarity’:

* 

**B-Tree: (Max\_items = 5)**

Time Taken to create the Tree: On average 18.6536 Seconds

* 

**Test 1:**

List Length: 15 words

Time Taken to Compute Similarity’s:

* 

**Test 2:**

List Length: 60 words

Time Taken to Compute Similarity’s:

* 

**Test 3:**

List Length: 120 words

Time Taken to Compute Similarity’s:

* 

**Test 4:**

List Length: 240 words

Time Taken to Compute Similarity’s:

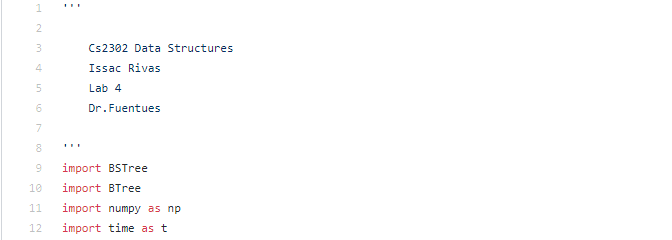
* 

The data collected helps show that the binary search tree always computes data faster then a b-tree with max inputs of 5. But if the Number of items being compared is moderately low and the max items is high then it has a chance of betting the binary search time as well as the b-tree with max input of 5. This is displayed by the two lines for the b-tree with max items 10, and 20.

**Conclusion**

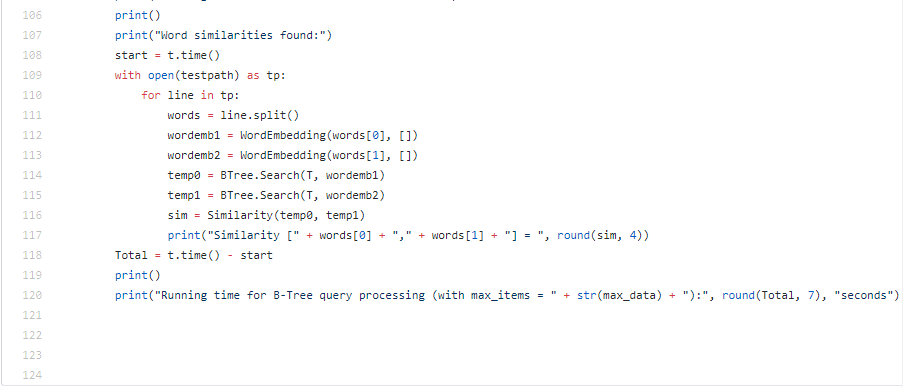
In conclusion, I was able to see how in certain instances of searching through trees that b-trees could be faster than binary search trees depending on factors such as the max number of items per node and the number of items being searched for. However, for a more reliable time of completion the binary search tree was the fastest and most consistent.

**Appendix**

 **Lab4.py:**





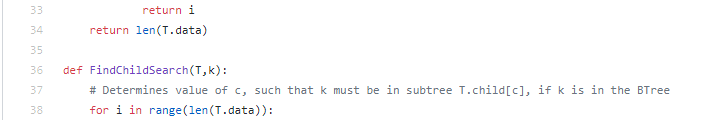


**BSTree.py:**





**BTree.py:**









I Issac Rivas, certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, preformed 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.