**LAB 4**

**DANIEL CRUZ**

**Introduction**

The purpose of this lab is to model an application of a binary search tree and b-tree. Natural language processing is an emerging field of artificial intelligence that requires data structures with quick look up for large data sets. In relation to modeling the application of these data structures, this lab also allows us to compare the runtimes of both building and searching between the two tree types.

**Proposed Solution Design and Implementation**

In both trees, I tried to incorporate the idea of abstraction in objects. I added a definition to the less than operator in word embedding to compare the words alphabetically from the object itself. I also allowed this operator to be compared with two WordEmbeddings as well as a WordEmbedding and a string. From here, I slightly modified the BST and btree codes from the class website to allow the insertion of wordembedding objects. I also modified the search functions to allow the user to input a string, and return the specific WordEmbedding that contains the given word.

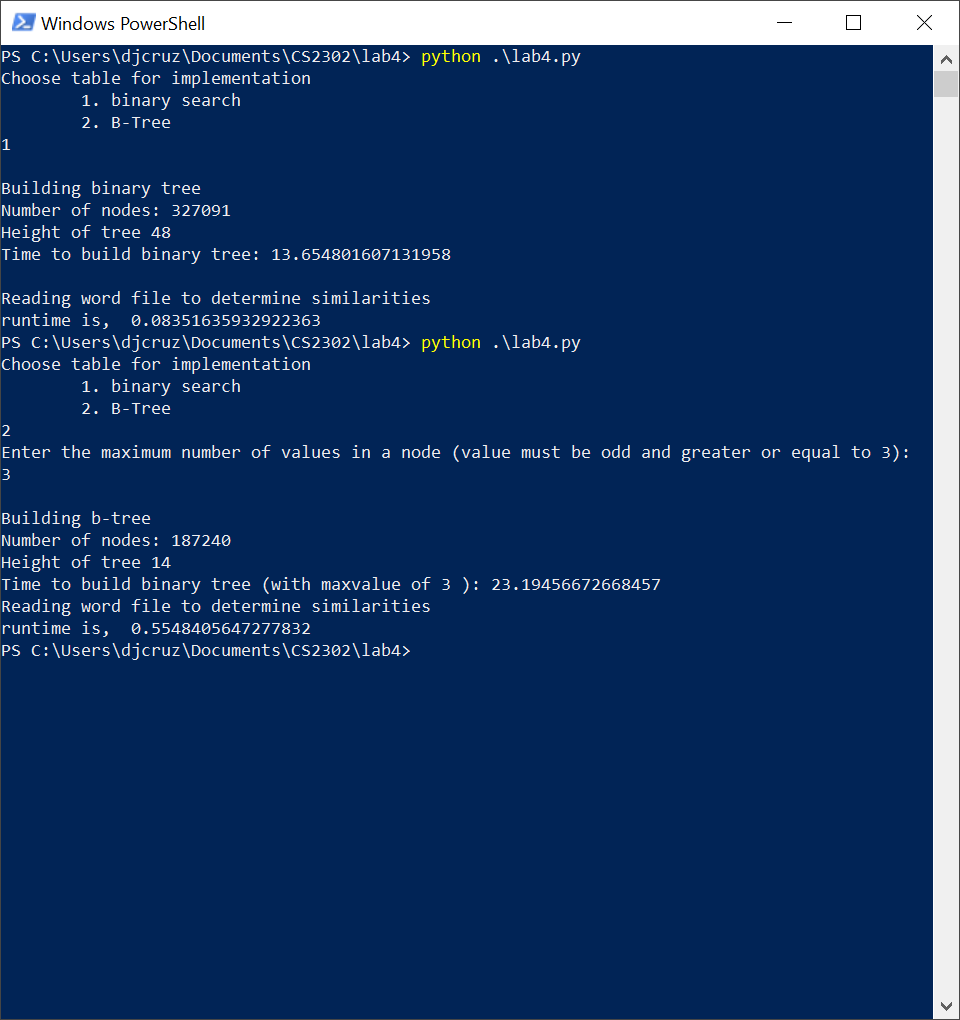
In terms of functions to interface with these objects, I chose to split the functions based on purpose. Main contains the menu system that the user will interface with. After checking for user input, the menu breaks off into either forming a binary search tree or b-tree from their own functions and printing the relative stats of forming the trees. After this, another function takes in the built tree (it doesn’t matter which one was chosen) and uses it to search words from a file. These words are either calculated for their similarities if they were found in the tree or reports that no embeddings were found if they were not found in the tree. These similarities are printed and then the runtime to search these words is printed.

Building either the binary tree or b-tree takes on a similar structure in either case. The file is opened and each line is split into a space separated list. The list contains the word as the first element and the embedding values as the rest of the elements. For every word that begins with an alpha character, it is inserted into the tree structure. After the end of file is reached, the tree structure is returned.

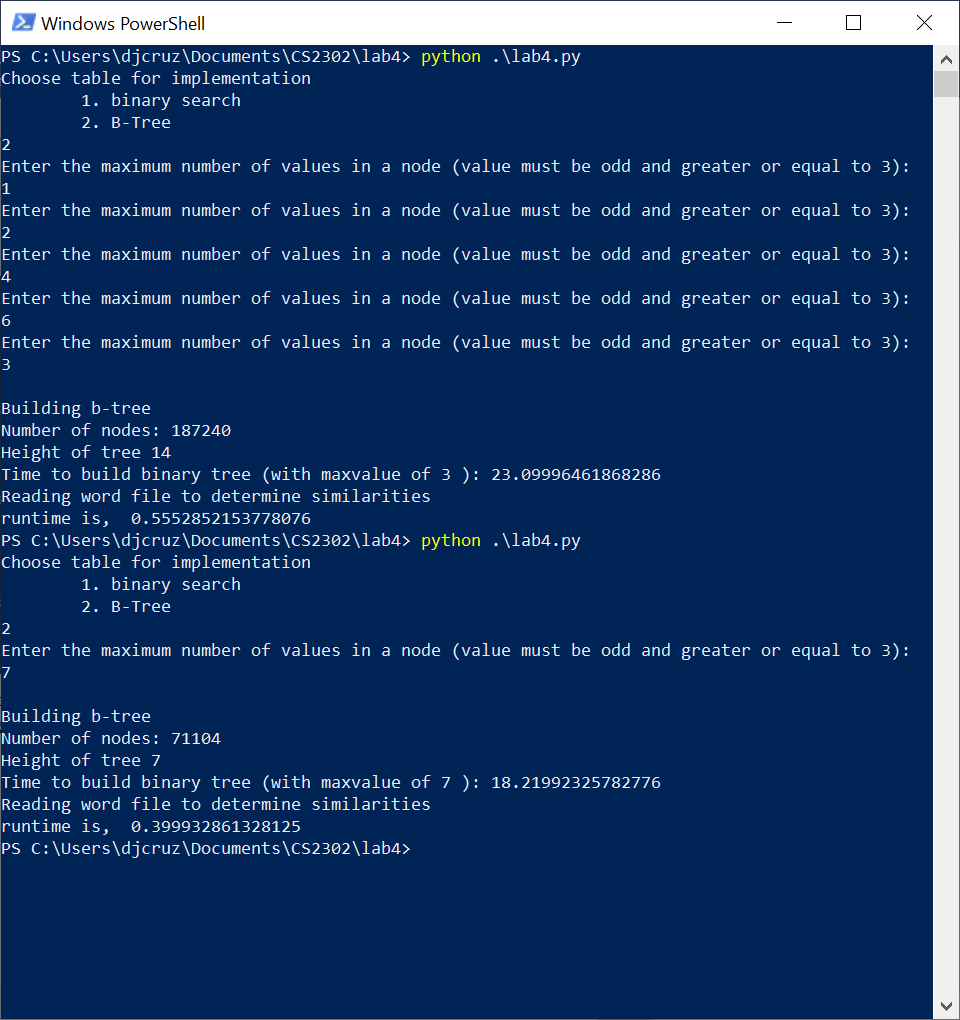
Another function was created to search for words from a file from either tree. A helper function was created to convert a file with words separated by newline characters to the required format for this function. This function requires that the two words to be compared be on the same line separated by a comma. Thus, this function reads in a line from the formatted file, splits the two words into a list, searches both independently based on the tree given, and then appends the two returning wordEmbeddings to a 2-D list. If either word returns None from the search, then they were not found in the list and just the strings are appended to the list. A summing runtime for each search is then returned along with the 2D list.

**Experimental Results**

This first output is just building a tree. Here, the file directed from the lab is read and a tree is built using this data.

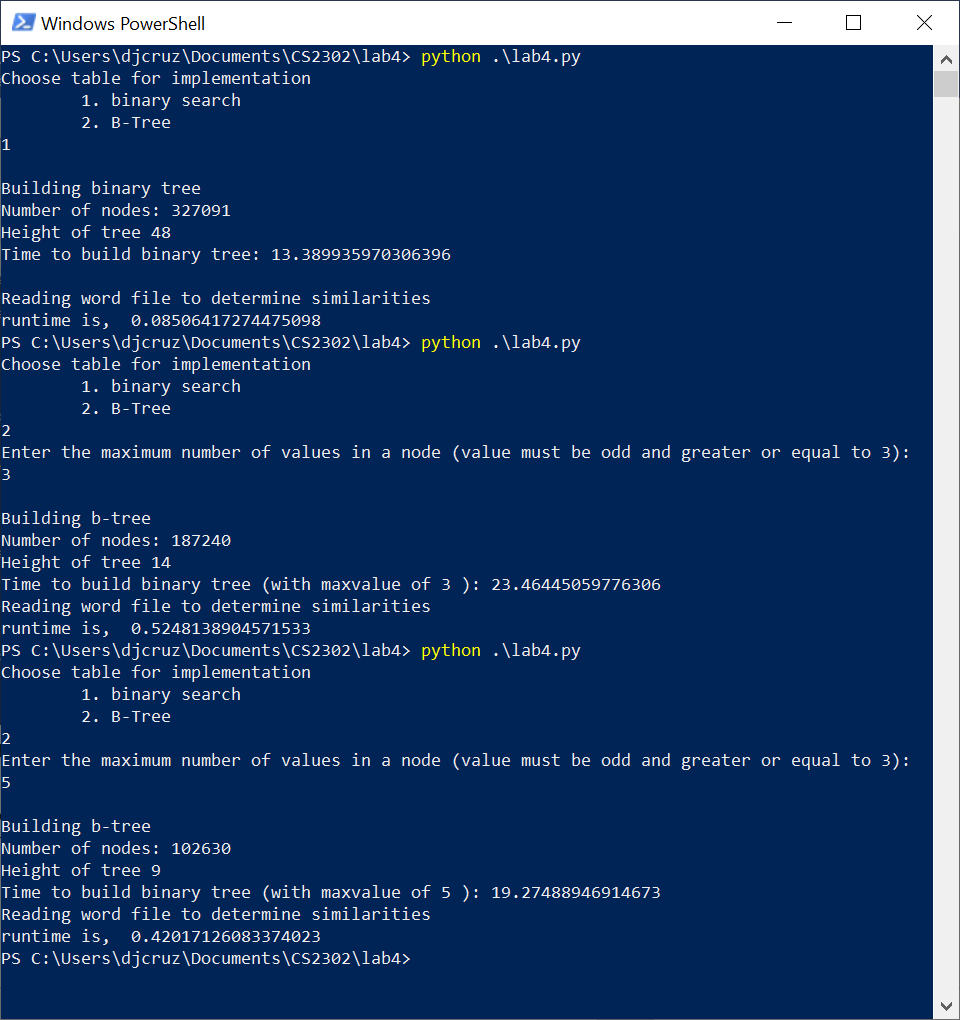


Both trees are being built and able to search words. I will come back to discuss the runtimes in a little bit. I will now test the ability to change the number of objects in a single node for the b-tree:



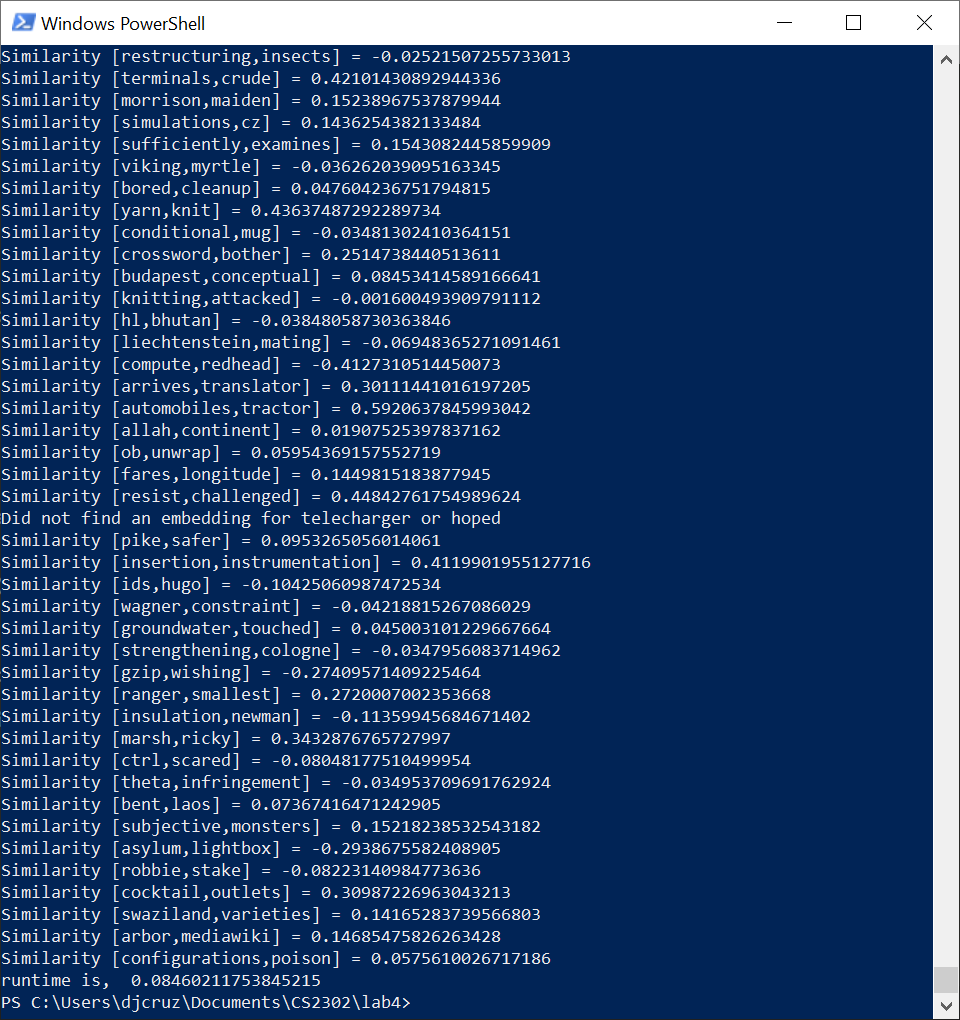
The menu is able to catch improper values for the number of elements in the node and able to apply the correct values to the tree when building.

The runtimes of building the trees will now be considered. I am using the text file from the lab instructions, only using words that begin with an alpha character.

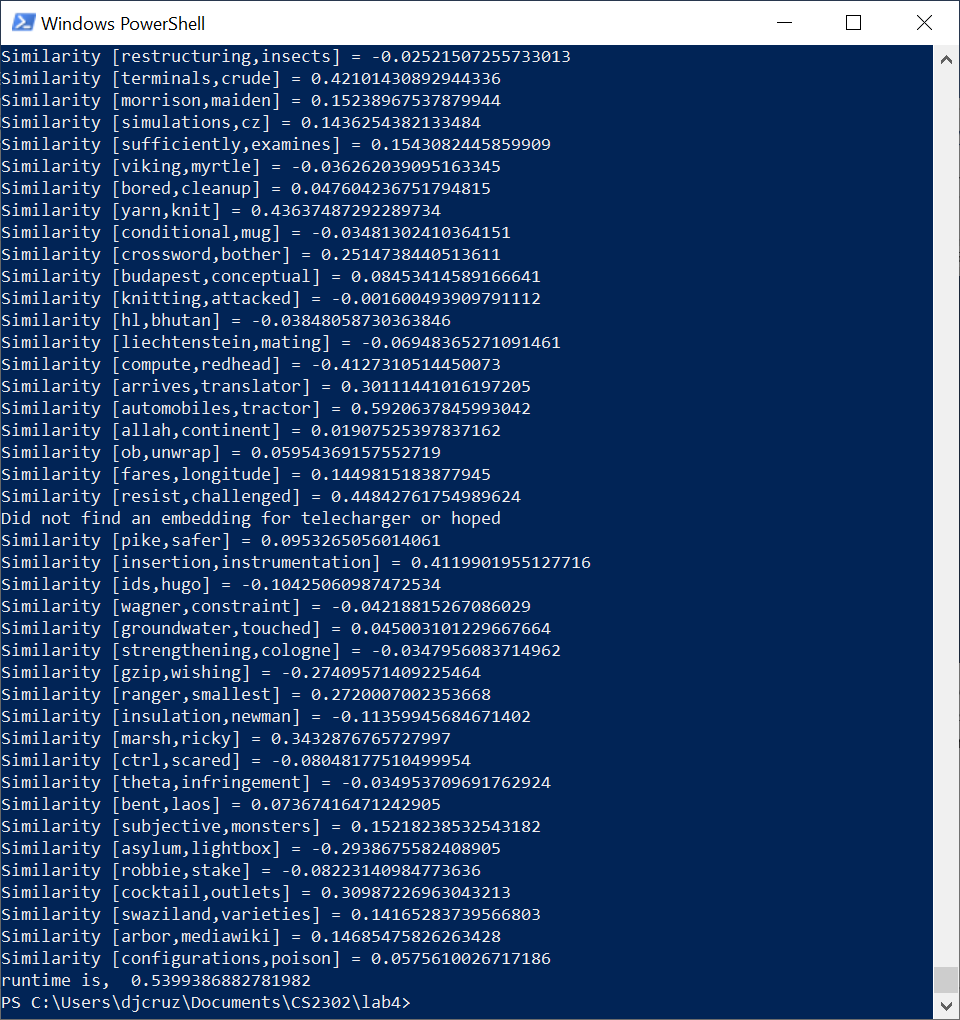


Binary trees are quicker to build while a b-trees with the minimum elements in a node. As shown, the time to build the b-tree does go down with the more elements allowed in a node as this would require more split operations performed on the tree. The greater the number of elements in a node, the less time it takes to build. This trend evens out after 9 elements are allowed in which the time to build is around 17 seconds.

For the comparisons between two words, I am using a file of 9,895 words obtained from the github: [google-10000-english](https://github.com/first20hours/google-10000-english). All the words start with an alpha character. However, there are some words found in this list that is not present in the word embeddings list. This is helpful to test the runtime even when given values that are not present in the tree. In the previous outputs, I suppressed the output of the similarities. They will now be printed for a binary search tree.



Here the last lines of similarities are shown. For word pairs where it could not find one of the words, such as telecharger and hoped, it does not print a similarity rating. These misses were very few and far between. The time it took to find all values was .085 seconds. Here are the b-tree similarities at a max element of 3:



The b-tree shows the same misses and similarities for the same word pairs as the binary search tree. While b-tree is still extremely fast, it is still slower compared to that of the binary search tree. Given different numbers of elements in a node, the runtime never reached below 0.4 seconds. Moreover, time began to rise up slowly after 11 elements.

**Conclusion**

I expected binary search tree to be quicker in building but slower in searching than a b-tree with optimized number of elements in a node. However, binary search tree outperformed b-tree in both cases. Even so, the optimal number of elements in a node in a b-tree is about 9 or 11 as this produced both the quickest build and search. It was rather difficult obtaining the runtime for searching as it required a large amount of words to derive any meaningful runtime. This shows the testament of how fast both trees are given the large database they hold. Word embeddings is an interesting way to represent words and is fairly accurate in finding how similar words are. I can see word embeddings being useful in auto-correct functionality in word processors.

**Appendix**

from time import time

from wordEmbedding import WordEmbedding

import numpy as np

import binarySearchTree as BST

import btree

def buildBinaryTree():

try:

file = open("glove.6B.50d.txt", encoding="utf8")

BT = None

for line in file:

dataList = line.split(" ") #first element is word, the rest is the float array

if dataList[0].isalpha(): #if word starts with an alpha character

BT = BST.Insert(BT, dataList[0], dataList[1:])

return BT

except Exception as e:

print(e)

print(line)

raise e

#inputs the max number of elements allowed in a node

def buildBTree(maxValue):

try:

file = open("glove.6B.50d.txt", encoding="utf8")

B\_T = btree.BTree([],max\_data=maxValue)

for line in file:

dataList = line.split(" ")

if dataList[0].isalpha():

btree.Insert(B\_T, WordEmbedding(dataList[0], dataList[1:]))

return B\_T

except Exception as e:

print(e)

raise e

#reads words from a file and returns a 2-d list of wordEmbedding and

def getEmbeddingsFromFile(T,fileName):

file = open(fileName)

runtime = 0

embeddingsList = []

for line in file:

#sliced to remove the newline character, then split by the comma

if "\n" in line:

line = line[:-1]

seachWords = line.split(",")

#determine type of tree is inputted

if type(T) == BST.BST:

startTime = time()

output1 = BST.Search(T, seachWords[0])

output2 = BST.Search(T, seachWords[1])

if output1 is None or output2 is None: #if word is not in tree, just appends the string words

embeddingsList.append(seachWords)

else:

embeddingsList.append([output1, output2])

runtime = runtime + (time() - startTime)

elif type(T) == btree.BTree:

startTime = time()

output1 = btree.Search(T, seachWords[0])

output2 = btree.Search(T, seachWords[1])

if output1 is None or output2 is None:

embeddingsList.append(seachWords)

else:

embeddingsList.append([output1, output2])

runtime = runtime + (time() - startTime)

return embeddingsList, runtime

#Does vector calculation to determine how similar words are

def similarities(embed1, embed2):

return np.dot(embed1.emb, embed2.emb)/(np.linalg.norm(embed1.emb) \* np.linalg.norm(embed2.emb))

def main():

fileName = ""

inp = 0

while(int(inp) < 1 or int(inp) > 2):

print("Choose table for implementation")

print("\t1. binary search")

print("\t2. B-Tree")

inp = input()

#Build binary search tree

if int(inp) == 1:

print("\nBuilding binary tree")

startTime = time()

table = buildBinaryTree()

endTime = time()

print("Number of nodes:",BST.Size(table))

print("Height of tree",BST.Height(table))

print("Time to build binary tree:",str(endTime-startTime))

print()

embeddingsList = getEmbeddingsFromFile(table,"wordSimilarities.txt")

#Build b-tree

if int(inp) == 2:

maxValue = 0

while maxValue < 3 or maxValue%2 == 0:

print("Enter the maximum number of values in a node (value must be odd and greater or equal to 3): ", end = " ")

maxValue = int(input())

print("\nBuilding b-tree")

startTime = time()

table = buildBTree(maxValue)

endTime = time()

print("Number of nodes:",btree.Size(table))

print("Height of tree",btree.Height(table))

print("Time to build binary tree (with maxvalue of", maxValue, "):",str(endTime-startTime))

#find similarities

print("Reading word file to determine similarities")

embeddingsList,runtime = getEmbeddingsFromFile(table,"wordSimilarities1.txt")

for embed in embeddingsList:

#pass #used to supress output of similarities

if any(isinstance(words,str) for words in embed):

print("Did not find an embedding for {} or {}".format(embed[0], embed[1]))

else:

print("Similarity [{},{}] = {}".format(embed[0].word,embed[1].word,similarities(embed[0],embed[1])))

print("runtime is, ",runtime)

#used to convert a file with words on each line to a file of two words on each line. This is the format for searching two words.

def convertToCSV(fileName):

inputFile = open(fileName)

outputFile = open("wordSimilarities1.txt","w")

word = 0

for line in inputFile:

if "\n" in line:

line = line[:-1]

if(word == 1):

outputFile.write(line + "\n")

word = 0

else:

outputFile.write(line + ",")

word = word + 1

#used to build csv for graph in report

def buildRunner():

fp = open("btBuild.csv","w")

for maxnum in range(3,21,2):

startTime = time()

table = buildBTree(maxnum)

endTime = time()

fp.write(str(maxnum) + "," + str(endTime - startTime) + "\n")

fp.close()

#used to build csv for graph in report

def searchRunner():

fp = open("btSearch.csv","w")

for maxnum in range(3,21,2):

table = buildBTree(maxnum)

embeddingsList,runtime = getEmbeddingsFromFile(table,"wordSimilarities1.txt")

fp.write(str(maxnum) + "," + str(runtime) + "\n")

fp.close()

if \_\_name\_\_ == '\_\_main\_\_':

#convertToCSV("google-10000-english-no-swears.txt")

main()

#bulidRunner()

#searchRunner()

I [Daniel Cruz] 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