Carlos’ Technical Report

This technical report outlines the various files in the hangman game. This project aims to create a hangman game in the terminal. It has two modes, random, and player. Random mode picks a random word from a txt file and plays hangman. Player mode allows the user to input a word uses a selected algorithm to validate the user-selected word. The major difference here is that instead of implementing a regular binary search tree it uses an AVL tree. The overall function of the game is similar to a regular hangman game. A wrong guess results in a body part. Once all parts are shown the game is over. To win the game all letters must be selected. The three search algorithms are brute force, decrease and conquer, and AVL search tree. Brute force has a worse case of O(n). Overall it doesn’t run too slow. The list is not long enough to cause it to run slow. The decrease and conquer algorithm, binary search, has a worst case of O(logn). This also runs well. The tree search runs an O(n^2). The main issue with this algorithm is that it requires the data to be in a tree. This takes a long time in Python and slows down the overall project.

1. Hangman.py

Hangman.py contains the game class. This class contains all the necessary components to run the game. It starts off with the variable maxGuess.

* maxGuess- This holds the maxim guess a player can have before they lose. It is used later in the class to check if the game is over or not.
* Hangman[]- this is an array containing the images of hangman. This array is iterated through to display the player’s progress
* \_\_init\_\_(self)- this is the constructor. This sets up the necessary variables for the game to run flawlessly. This is where the word is selected. It can later be overwritten with a later function.
* random\_word(self)- Here we open the dictionary file and using the Python random library choose a random word.
* get\_progress(self)- This function is used later to print the results of the ongoing game. This function in specific displays the progress on the word. It uses “\_” to denote the letters remaining and in which spots.
* already\_guessed(self, guess)- As the name implies this checks to see if the user’s guess has already been guessed. It is used later to validate the user’s guess.
* guess\_letter(self, guess)- if the guess is valid it will determine whether the letter is added to the correct letter array or the wrong letter array.
* status(self)- In this function, the game determines if you win or lose. A loss means the number of guesses is greater than the maxGuess. A win is determined if the correct letter array is the same as the chosen word. If the game is ongoing it prints out nothing
* \_\_str\_\_(self)- This prints out all the progress of the game. It starts off by making a line so the play can distinguish each game block. It then prints out the correct index of the hangman array by using the length of the wrong guesses array. It then uses get\_progress to print out the words progress. Next, it prints out the wrongGuesses array. Finally, it prints out its status.
* setWord(self, word, tree, root)- this function uses an algorithm to validate the user-chosen word. The validation takes place in the algo.py file. This function will overwrite the random word with the provided one if it is valid.
* validGuess- this function checks the guess and determines if it is valid. If quit is provided it quits the game. If the guess is not a letter it tells the user to provide a letter. It also checks if the guess was already guessed.

1. Algo.py

Algo.py contains all the algorithms used in the player mode. It is only accessed when the player types “player” when prompted for mode.

* bruteForce(word)- this function is used when the player selects brute force. It opens the dictionary file and uses the toArr function which was created in project 1 and repurposed for this one. It then iterates through the array checking each element and comparing it to a modified version of the user’s word. This is because the list’s words all contain the escape sequence \n at the end. So we add that same escape sequence to our newWord variable. This method is repeated for the other search algorithms. If the word is not found it is sent to the autocorrect function. Otherwise, it returns the user-selected word. This searching worst case is O(n)
* doDecreasebyConstant(word)- this function does not actually contain the algorithm itself but instead prepares the variables to send it over. Similar to the Bruteforce function we create an array for the dictionary and modify our word. It then sends our word and array into the binarySearch function. If found it returns the word. Otherwise, it goes to the autocorrect function.
* binarySearch(x, arr)- This function uses a binary algorithm to locate the word. If found returns true. Otherwise, return false. It worse case is O(logn)
* avl(word, tree, root)- again we create an array for dictionary and modified word. We then use the search function form the avl class to validate the word. If it is found the user is asked to delete or use the word. If deleted the word is removed from the tree, not the dictionary. Once deleted the user is given the default word “mango” to play. If not found the user is asked if they would like to add it to the tree. If added the game begins. If not it is sent to the autocorrect function.
* autocorrect(word, arr)- autocorrect works by importing Python’s difflib library. It then uses difflib’s function, “get\_close\_matchs” to rearrange an array from most similar words to least. The closest three are printed to be selected form. The user-selected word is sent back to the function it came from and then validated again. This is to ensure that the new word is also valid.

1. avl.py

Avl.py contains the avl class. It is the class that allows items to be added to the avl tree. It also contains the tree node class which is used to structure the tree.

* Treenode:\_\_init\_\_(object)- this represents a node in the tree. It hold a root to the left and right.
* insert(self, root, key)- this function inserts a node into the tree. The first step determines what side of the tree it belongs in. it then updates the height of the tree. It then checks the balance of the tree. Depending on its value will rotate the tree left or right inorder to balance the tree.
* leftRotate/rightRotate- Both functions are somewhat similar. They both rotate the items in the tree update the height and then return the root.
* getHeight- return the height of tree
* getBalance- return the balance of the tree
* search(root, key)- this search algorithm first checks to see if the root is null. Then checks if the root’s value is the same as the key. If they are it returns true. If root’s value is less than the key it compares the right node. Else it compares the left side of the tree. This happens recursively so it only ends if the root is found or is not.
* delete(root, key)- This function works similarly to the insertion function. If the word is found it is deleted from the tree. The function then rebalances the tree.

1. Tester.py

Tester serves as the main file for the program. Here is where all the other files are brought together and plays.

* The first thing we do is create our avl tree. This is done so we don’t have to do it later because it takes so long. We open our dictionary.txt and add its elements to a tree.
* We then create a while loop to keep playing. The first thing created is a game class. This is done on every loop to reset the progress and change the random word.
* The user then selects the mode. Any other input besides player will go to random.
* If player is selected the user will ask for a word. They will then be prompted to choose their algorithm. Once the word is validated the game will begin.
* The next loop allows the user to keep making guesses. In this loop, the user is asked to guess. After each guess the progress is shown.
* Once the game is over the user is then required to either play again or end. Any other response other than “n” will continue the game.

Overall the Python file runs as expected. The most disappointing aspect is how long the avl takes. A simple game like hangman should not take as long as it does. It makes it less enjoyable having to wait for a game that isn’t really fun in the first place. If I were to enhance this project I would change the langue. Many of my peers did this in Java and ran into way fewer issues. Python takes way to long with large data sets. I knew about this from my last project but did it again in Python. I would also want to figure out how to implement a BST. For some reason, it simply did not worst and I resorted to using an AVL. With more time it might be possible to figure it out but to submit I wanted to make the deadline. Another improvement would be to use something like Tkinter to create a GUI. This would make the game more interesting as it wouldn’t be stuck to just the terminal. In all, the project was very fun and interesting to work with.