Project 3: !thesaurizethis

Jake Deeks – Jad0420

Project Description

In this project, I was assigned to make a system that would initially read in a list of words and synonyms and do some computations with a created adjacently list made with the read in words and synonyms. There will be two different type of processes needed in this project that uses the created adjacently list. One of the processes is random synonym replacement which is where you would search a given paragraph till you find a word that is in the synonym list. Then you would proceed to replace the synonym with its synonyms for a number of times designated by the user and it will end with printing the new paragraph out to the user. The other function will take two paragraphs given by the user and perform a paragraph comparison analysis on the two paragraphs. It will compare each word in each paragraph against each other at that position in the paragraph till it finds a situation where the two words are dissimilar. Once it finds when the words are dissimilar, it will proceed to use a searching algorithm to see if the two words can be synonyms of each other through connected words. If they are connected synonyms, continue with the paragraph comparison but if they are not then stop the paragraph analysis and print out to the user that the paragraphs are different.

Data Structures

The main data structure that is used for both of the processes required is a undirected graph which was made into a adjacently list through the use of vectors. I made the adjacently list a 2-dimensional vector (Vector<Vector<string>>) which is used to hold all the root words and their synonyms. I decided to use an adjacently list instead of a adjacently matrix to hold the root words and synonyms graph due to it being more manipulatable and won't waste space like a adjacently matrix would. This 2-dimensional vector is stored within a menu like class that holds the data structure and all the functions that need to manipulate the data that is stored within the adjacently list.

This data structure is used in both the synonym replacement and paragraph comparison analysis. There are multiple functions throughout my program that have many different functions but a common occurrence between them is that there is vectors within each function. The vectors are used to either temporarily hold the synonym list before being put into the 2d vector or the vector is the temporary vector that is finally pushed into the 2d vector to create a root word with its synonyms. A Vector can be also used in the breadth first search to hold the visited nodes.

System Functionality

I started the project with prompting the user to input the file that will contain the thesaurus that will be used throughout the program. I proceeded in making a menu to allow the user to conveniently choose which type of process the user wanted to perform with it either being the random synonym replacement or the paragraph comparison analysis. Once the user chooses which process to use, I have a switch statement to call the functions needed to run that certain process. This is where the code were will split between the two processes and will be independent of each other.

While making the adjacently list, there were many conditions that were needed to take caution of. First thing to do is to grab the first word in the line of the file as the node word and grab the rest as a comma delimited list which will be split up into a vector of strings. This two will then be put into a temporary vector of strings to be pushed into the 2-dimensional vector. There are four different cases that one must be aware of when making the root nodes with it synonyms. First one is just to normally create the root word first followed by its synonyms. Second, a synonym becomes the root word with the original root word becoming a synonym along with the other original synonyms. Third, if the root word has an already created root node in which the synonyms would just get added on the already existing synonyms. The fourth and final case is when a synonym already has a root node in where you would follow the same procedure as condition three with adding on the root word and every synonym except the one you tested and found it had a match. Using these 4 conditions, the adjacently list was created to be able to use for the two processes.

For the first process of random synonym replacement, the user Is prompted for the paragraph that the user wants to use for the synonym replacement. After verifying the file, I read in word by word from the paragraph file while also simultaneously checking if the word has special cases like punctuation or capitalization. In case of capitalization, the word be transformed into all lower case while also flagging a Boolean to state that this word had a capital letter so it can be reverted to its original state later on after the replacement succeeded. In the case punctuation, a Boolean will be flagged stating that is contains punctuation and that certain punctuation will be held in a holder character to put back after replacement. For the replacement part, there is a loop calling the function “find(word)” that will use random number generated to randomly select a synonym to replace the word given and each iteration of the loop is a single hop so it will run the amount of hops the user designated. The punctuation and capitalization will be restored if their Booleans were flagged true. The function ends with printing out the new changed line of words.

For the second process of paragraph comparison analysis, the user is prompted for the two paragraphs that the user wants to compare. There will be a check to see if those files are valid to be opened or not. After opening the files, the program will proceed to read in 1 word at a time from each file simultaneously to compare against each other. Though first, the program has to proceed with removing all of the punctuation and capitalization on both words to make sure that there is no interference for those words. After normalizing the words, it will begin to check to see if the words are the same word and if they aren’t, it will proceed with calling the function “BFS(word1,word2)” which is the breadth first search that will search through the entire adjacently list starting with the first word given till it either doesn’t find the target word or finds the target word. After checking if your adjacently to see if the two words are connected through synonyms, it will return a Boolean value that states whether or not one word could possibly turn into the other word. If the words cannot be connected it will end the paragraph comparison and print out to the user the failure of the paragraph comparison while the opposite will continue looking through the entire paragraph of each file. At the end, if there is no dissimilarity between the paragraphs it will tell the user that the paragraphs are similar.