**Analysis of Algorithms**

Spring 2020

**Members Details**

| Group ID | CS311-G12 |
| --- | --- |
| Registration Number of Group Members | 2018-CS-103  2018-CS-133 |
| Section | C |

**Project Details**

|  |  |
| --- | --- |
| ***Project*** |  |
| Project Title | Plagiarism Checking Tool |
| Executive Summary | This project is about checking plagiarism (similar content) between two as well as number of input files provided. The input will be given in the form of files (or folder in second task) and output will be shown in a new window (form). A number of tasks have been implemented in our project. First task implemented, checks the same content between two textual files (.txt extension) and returns the similar text between the files. The output (content) that would be shown will contain both matching words as well as matching sentences. Similarly, the second task checks the plagiarism among number of files (having .txt extension) kept in a zip folder form (.zip) inputted to our checker tool. While in case of third task (bonus) i.e. checking the altered/changed content between two C++ coded files, showing that content and also telling the total number of changes that had been performed in those C++ files — has not been implemented in the project yet.  The coding has been performed in Java. Desktop application was made in C# (only forms/interfaces) at first, but due to not being able to connect it with our Java code, we switched to Java (Eclipse integrated development environment) for making the desktop application (GUI), and successfully connected it with back end Java code. Moreover, the approach for implementation was naïve based at first, but due high time complexity, it was changed and new pseudocode with relatively less time complexity was committed on Github. Final algorithm implemented in the project was “Modified longest common subsequence” (LCS) telling both about similar words as well as sentences.  The time complexity of algorithm (modified LCS) used, has also been committed. The final graphical user interface has been made in Java which is synchronized with the “drawing of interfaces” (on professional tool), submitted at the initial phase of term project i.e. having same buttons, features etc. |
| ***Business Case*** |  |
| Outline the business need for the project | Plagiarism checking tool is needed in both educational and organizational set up for monitoring the students and employees’ tasks/assignments respectively. While going through each and every task can be a hectic job for an evaluator, this tool solves the problem efficiently by doing the job on its own and helps assessing people’s work. |
| End user of the product | In this project, we are basically targeting teachers, checking similar content among number of files of the students, evaluating if there exists a ditto copy in assignments. |
| Motivation for Project | If we could able to make such a tool checking plagiarism so efficiently, it would be a great help for teachers checking students’ assignments. |
| Description of the project objective(s) | * This automation of checking plagiarism can reduce labor costs of evaluators. * It would be more efficient for assessments as there may be chances of missing out something manually. * While manually assessing the work of each and every person would be time consuming and hectic, it would be doing the job on its own. Thus, saving time of evaluators. * It would definitely save the energy costs of evaluators. * It would be helpful in taking decisions/compiling results more fast. * By setting a proper/acceptable format of input (e.g. .txt extension), it can clearly approve or reject the provided work. * Once implemented successfully, it may enhance the performance of students/observees doing their assignments. * It may help to reform the organizational culture of not doing plagiarism at all. * It can help teachers getting quality work by the doers. |
| State the level of impact expected should the project proceed and implications of not proceeding | The implementation would certainly have an effect on checking the tasks of observees. Checking the plagiarism through it would be time saving for evaluators.  While it can save the time of evaluators, it might take more time of the tool when a folder containing many files is inputted. Also, the correctness in the assignments cannot be evaluated by this project. |
| Functional Requirements | Features that have been implemented are:   * It can easily check the similar content between two textual files. * It can check the similar content among number of files in a folder, inputted to the tool. * It enables user to upload any file (.txt), placed anywhere in the computer by permitting him to browse through his computer. * It can check the validation of input data/file e.g. if the extension is set .txt in the algorithm, user cannot input any file other than this extension.   We are trying more features of telling about the readability of input files and telling the percentage of plagiarism among files, if implemented successfully, would be updated in the second report after testing phase. |
| ***Benefits*** |  |
| What benefits are expected/ anticipated? | * Evaluators, specifically teachers need not to be spending time on checking whether there exits plagiarism in the assignments or not. So, it would be time saving for them. * Teachers may simply upload the folder containing students’ files and the tool will do the job on its own. Thus automation is achieved. * Once implemented successfully, it may elevate the performance of the students, pave the way for producing more quality work without copying from any other student. |
| ***Implementation Details*** |  |
| Link to Github Repository | <https://github.com/fahadmurtaza133/CS311S20PID12> |
| Total Number of commits in repository before 5 August 2020 | 29 commits |
| Exact contribution of each member | Member with Registration number 2018-CS-103 performed the following tasks:  Decision of algorithm to be implemented, drawing of simple interfaces on a tool, initial Pseudocode, decision of new algorithm (LCS) to be implemented, new Pseudocode, time complexity analysis,  Synchronization of interfaces in C# with initial document of drawing of interfaces (modification), tried finding about connection of Java code with desktop application in C#, suggestion of new feature of readability (grade) of files and started working on it.  Member with Registration number 2018-CS-133 performed the following tasks:  Implementation of initial algorithm, time calculation (in nanoseconds) of the code, suggestion on modification of LCS, implementation of modified LCS algorithm, making of interfaces on C#, tried finding about connection of Java code with desktop application in C#, made desktop application on Java Eclipse IDE and connected with code, suggestion of new feature of telling percentage of plagiarism and started working on it.  Both members coordinated with each other throughout the project. Modification (editing) of other member’s task was performed on github if necessary. |
| ***Commits in github repository by each member*** | |
| |  |  | | --- | --- | | **Member Registration No.** | **Total Commits** | | 2018-CS-103 | 11 | | 2018-CS-133 | 18 | | |
| **Details of commits** | |
| |  |  |  |  | | --- | --- | --- | --- | | **Sr. No.** | **Details of commit** | **Date** | **Member Reg No.** | | 1. | Initial algorithm used and drawing of interfaces on a tool. | July 3, 2020 | 2018-CS-103 | | 2. | Pseudocode of initial algorithm | July 4, 2020 | 2018-CS-103 | | 3. | Implementation of initial algorithm in Java | July 4, 2020 | 2018-CS-133 | | 4. | Finding of time in nanoseconds by the algorithm | July 8, 2020 | 2018-CS-133 | | 5. | Decision of new algorithm and its Pseudocode | July 9, 2020 | 2018-CS-103 | | 6. | New pseudocode complexity analysis | July 9, 2020 | 2018-CS-103 | | 7. | Implementation of new algorithm in Java | July 13, 2020 | 2018-CS-133 | | 8. | Interfaces (only forms) in C# | July 20, 2020 | 2018-CS-133 | | 9. | Implementation of desktop application in Java Eclipse IDE. | August 4, 2020 | 2018-CS-133 | | 10. | Suggestion of new feature of readability (grade) of files | August 4, 2020 | 2018-CS-103 | | |
| Have you used built in algorithms or you have implemented yourself? | First we implemented our own algorithm which was naïve based having high time complexity. Therefore, we switched to LCS algorithm. In LCS algorithm, we did a little bit of changes. We introduced one more function in the implementation and finally achieved the desired results i.e. matching words plus matching sentences. |
| Formats of input | Input can be given to the system in two ways:   1. One can upload two textual files (.txt) one by one by clicking on the first button “Plagiarism between two files” on the application. 2. A zip folder containing many files, having same extension (.txt), can also be uploaded by using the second button “Plagiarism among number of files” on application. |
| Validations | Input will be valid in the following ways:   1. When first button “Plagiarism between two files” in the application will be clicked, user will able to upload two files one by one to the application. Those files will be valid if they both have .txt extension, otherwise not valid. 2. When second button “Plagiarism among number of files” will be clicked, user will be able to upload a zip folder containing many files. It would be valid if the extension is .zip, otherwise not valid. 3. If third button “Plagiarism between C++ coded files” will be clicked, user will be able to upload two files one by one to the application. Those files will be valid if they both have .cpp extension, otherwise not valid. |
| Format of output | When two textual files are uploaded with the help of first button, the output is shown in a new jframe containing the similar content between two files. Also in case of uploading the folder using second button, the output shown to the user will have a jframe having similar lines and words among the files provided in the folder. |
| Deployment | We made desktop application. We have not deployed anything so far. |
| ***Details of algorithms*** | |
| First of all, we made our own algorithm which was naïve based and had high time complexity. Its pseudocode is given below:  Let s1[1…n] and s2[1…n] are two input strings.  FIND\_PLAGIARISM (s1, s2)  1. newf1[] <- SUBSTRING (s1)  2. newf2[] <- SUBSTRING (s2)  3. **FOR** i <-1 **TO** length[newf1]  4. **FOR** j <- 1 **TO** length[newf2]  5. **IF** newf1[i]==newf2[j]  6. **THEN** *highlight that portion*  7. **ELSE** *don’t highlight anything*  Definition of SUBSTRING () procedure:  Let s[1…n] be a string given to our procedure.  SUBSTRING (s)  1. temp1 <- length[s]  2. s1[] <- split[s] *//splitting our string character by character*  4. fs1[] <- New\_string [] *//this string will be created at runtime for separating the whole*  *//sentences*  5. **FOR** k <-1 TO temp1  6. fs1[k] <- “ ” *//for ignoring NULL values*  7. j <- 1  8. i <- 1  9.  **WHILE** j != temp1 **AND** s1[j] != “.” *//because we want whole sentence to be highlighted*  10. fs1[i] += s1[j]  11. j <- j+1  12. i <- i+1  13. **return** fs1  At initial stage, the algorithm made was enabling user to input two strings s1 and s2. We found plagiarism between two strings by comparing each and every character of first string with the second one. If there had been whole sentence matching in both texts, then it would be printed. Otherwise, in case of a word matching, there would not have been printing anything. For example, if “s1=Hi Ali. You are a boy.” and “s2=Hi Ahmad. You are a boy.” Then “Hi” will not be printed, instead “You are a boy” will be printed.  From above pseudocode if s1 had length m, then combinations it could make was 2m and similarly s2 with length n had combinations of 2n. As the testing combinations whether or not s1 was present in s2 took O(n) time. Ultimately, the naive method could take O(n2m) time in its execution which was very high.  So, to reduce the time complexity of the initial algorithm discussed above, we switched to LCS algorithm having less time complexity comparatively. The pseudocode of our new algorithm is given below:  PRINT-LCS (X, Y)  1. m = X.length  2. n = Y.length  3. Z[m+1][n+1] *//It contains length of LCS of X[0..i-1] and Y[0..j-1]*  4. **FOR** i = 1 to m  5. **FOR** j = 1 to n  6.  **IF** i == 0 OR j == 0  7. Z[i][j] = 0  8.  **ELSE IF** X[i-1] == Y[j-1]  9. Z[i][j] = Z[i-1][j-1] + 1  10. **ELSE**  11. Z[i][j] = max(Z[i-1][j], Z[i][j-1])  12. index = Z[m][n] *//For printing out the LCS*  13. lcs[index+1] *//character array for storing same string*  14. lcs[index] = '\0'  15. i = m  15. j = n  16. **WHILE** i > 0 AND j > 0  17. **IF** X[i-1] == Y[j-1]  18. lcs[index-1] = X[i-1]  19. i –  20. j –  21. index –  22. **ELSE IF** Z[i-1][j] > Z[i][j-1]  23. i –  24. **ELSE**  25. j –  26. **RETURN** lcs  Definition of SUBSTRING () procedure:  Let X[1…n] be a string given to our procedure.  SUBSTRING (X)  1. m <- length[X]  2. Y[] <- split[X] *//splitting our string character by character*  4. fs1[] <- New\_string [] *//this string will be created at runtime for separating the whole*  *//sentences*  5. **FOR** k <-1 TO m  6. fs1[k] <- “ ” *//for ignoring NULL values*  7. j <- 1  8. i <- 1  9.  **WHILE** j != m **AND** Y[j] != “.” *//because we want whole sentence to be highlighted*  10. fs1[i] += Y[j]  11. j <- j+1  12. i <- i+1  13. **return** fs1  As seen from above, that we implemented Dynamic Programming in our Plagiarism Checking tool. Of course, it had some space complexity as well, but according to us it was a good compromise. Besides implementing merely LCS, we made one more function for finding the substring (the matching portion of two texts/files). For telling the time and space complexity of procedure PRINT-LCS, we can clearly see that there are two FOR loops for both the strings (line 4, 5), therefore the time complexity of finding the longest common subsequence using dynamic programming approach is **O(m\*n)** where m and n are the lengths of the strings X and Y respectively. Since this implementation involves only m rows and n columns for building LCX (X, Y), thus the space complexity would also be **O(m\*n)**.  Also the time complexity of procedure SUBSTRING () will be O (m + n) because the main loop of this procedure runs no more than m + n times. Therefore, the whole time complexity of our implemented code will be O (m\*n) + O (n + m). | |
| ***Interfaces for your project*** | |
| The first interface shown to the  user will look like as shown in  figure 1.1  It will have three buttons:  1. First button will show the  plagiarized content between  two textual files.  2. Second button will show the  plagiarized content among  number of given files.  3. Third button will show the  changes that were done  between C++ coded files.  Figure 1.1    When first button i.e. "Plagiarism between two textual files" will be clicked, screen (Figure 1.2) will be shown to user.    Figure 1.2  The first two buttons shown on Figure 1.2 will enable user to input his textual files while the third button will check plagiarism between those files. The "Back" button at the end will bring user back to first screen like shown in Figure 1.1.  When the button "Check Plagiarism" will be clicked, our system will show the similar content between the files. If there will be some similar content, a new screen/jframe will be opened showing the plagiarized content while if there will be no similar content, it will just output a message saying "No Plagiarism detected".  When second button in Figure 1.1 i.e "Plagiarism among number of files" will be clicked, a screen will be shown to user like given below:        Figure 1.3  User will have to upload a zip folder (.zip) having textual files of same extension through the button "Upload folder containing files". When "Check Plagiarism" button will be clicked, our tool will show the similar content among number of files and "Back" button will return to Figure 1.1.  When third button in Figure 1.1 i.e. "Plagiarism between C++ coded files" will be clicked, a screen will be shown to user like given below:    Figure 1.4  User will have to upload two C++ coded files individually through the first two buttons shown on Figure 1.4. When "Check Plagiarism" button will be clicked, our tool will show the changes that were done between two files. "Back" button will return to Figure 1.1.  This functionality (bonus task) has not been implemented in our project yet. | |
| ***Integration*** | |
| We first made our user interface in C#, but due to not being able to connect/integrate with our algorithm code written in Java, we switched to Java Eclipse IDE and achieved the desired results.  The difficulty/problem of connection with C# was we could not take any help from the internet or any other platform about connection. So, for not wasting more time, we used Eclipse IDE. | |
| ***Change Requests*** | |
| Initially, we proposed our own algorithm for finding plagiarism between two texts (naïve method). But due to its very high time complexity (exponential), we switched to LCS algorithm using dynamic programming. Instead of residing on LCS merely, we made changes a little bit in our implementation by introducing one more function called Substring () and gave the whole time complexity.  Also, in case of interfaces, our first choice was C# but due to not being able to connect it with implemented code, we made new (same) GUI or jframes on Eclipse IDE and achieved results. | |
| ***Testing*** | |
| *In this section, you are required to mention the issues report and solution proposed.* | |
| ***Technology*** |  |
| Programming Language | Java, C# |
| Platform | Desktop Application |