**Analysis of Algorithms**

Spring 2020

**Members Details**

| Group ID | *CS311-G36* |
| --- | --- |
| Registration Number of Group Members | *2018-CS-16*  *2018-CS-50* |
| Section | *A* |

**Project Details**

|  |  |
| --- | --- |
| ***Project*** |  |
| Project Title | FILE COMPRESSION TOOL |
| Executive Summary | This Project involved the design and implementing of a File Compression Tool which is basically a tool that compresses and decompresses files. The core goal of this tool was to design and build such a tool that compresses files to reduce size of files and then decompresses files that file to produce the original file. So, that files/data manipulation becomes more efficient and it also reduces files system resources.  For this purpose Huffman coding algorithm is used which is a lossless file compression algorithm. It basically counts the frequencies of the characters from file then apply Huffman coding to get a binary code against each character. Characters with large frequency will have the smallest code to reduce the size of the output file. Output is written bit by bit in the compressed file with the file header in which Huffman tree information is stored. That tree information will help in decompression.  For decompression, compressed file is used. First of all tree information is extracted from it and a Huffman tree is build. Then reading each bit from file, tree is traversed whenever the leaf node is reached that character is written in the output file.  The tool is implemented using high-level language and as it is required to make a UI for application so a desktop application is built. It gives the users a responsive layout. So, it is implemented using language C# and WPF (Windows Presentation Foundation) .NET Framework.  C# is used to implement the backend of the project and WPF.NET Framework is used to support the UI of the File Compression Tool. Numerous modules or unit functions are also used to get the desired outcomes from the tool.  This project successfully, compressed the files having extensions **.pdf, .docx, .cpp, .csv and .txt** and also decompressed the files having same extensions that are mentioned. Users can easily interact with the tool by simply uploading a file that is require to compress, then tool will compress the file and users can save the compressed file with the same extension as it was uploaded for compression using any desired name for it. It also benefits users by providing, option for decompressing the files and users get their original files.  Hence, File Compression Tool meets the user’s needs of compressing and decompressing the files having multiple extensions. It minimizes the file size and helps to save space or minimize memory requirements without losing file contents. |
| ***Business Case*** |  |
| Outline the business need for the project | The business needs for File Compression Tool are reductions in storage hardware, data transmission time and saves transmission costs. Compressed files need significantly minimum storage capacity than uncompressed files, meaning a significant decrease in expenses for storage. |
| End user of the product | The end-users of this project are those people who use a computer to deal with files. Users can also be students who need to compress the files for their assignment submissions as well it is helpful in data/file sharing. |
| Motivation for Project | The motivation behind selecting this project was to get the hands on practice how the file size reduces and saves storage capacity and how data transmission becomes efficient. |
| Description of the project objective(s) | The key objective of the project is to compress and decompress the files uploaded as input as per user requirements. |
| State the level of impact expected should the project proceed and implications of not proceeding | At operational level our project compresses and decompresses the files having extensions .txt, .cpp,.c,.cs,.pdf,.docx based on user requirements.  While at strategic level, we want to add more file extensions for the purpose of compression and decompression and also it is a hope that we work for minimizing the compression and decompression time as more as possible. |
| Functional Requirements | Our project gives the service of compressing and decompressing a file. It takes a file as input compress it and saves the compressed file. Then it takes compressed file as input and decompress it. |
| ***Benefits*** |  |
| What benefits are expected/ anticipated? | The following is the list of benefits expected or anticipated from the project:   * The main benefit of Huffman encoding is to leave the rigid 8-bits-per character requirement and use different-length binary encodings for different characters to minimize storage requirements. * Data transmission increases and data transmission cost decreases. * Storage expense decreases. |
| ***Implementation Details*** |  |
| Link to GitHub Repository | <https://github.com/MARIA-AZRAR/CS311S20PID36> |
| Total Number of commits in repository before 5th August, 2020 | 62 commits |
| Exact contribution of each member |  |
| ***Commits in GitHub repository by each member*** | |
| |  |  | | --- | --- | | **Member Registration No.** | **Total Commits** | | 2018-CS-16 | 18 commits | | 2018-CS-50 | 42 commits | |  |  | | |
| **Details of commits** | |
| |  |  |  |  | | --- | --- | --- | --- | | **Sr. No.** | **Details of commit** | **Date** | **Member Reg No.** | | **1.** | Reading bit by bit from file was implemented and decompressing Algorithm to compress the text files for the project was created. | 13th July, 2020 | 2018-CS-16 | | **2.** | Characters from file were successfully read and their frequencies were calculated. Nodes were created and added in queue. Huffman Encoding Algorithm successfully implemented and codes were calculated. | 13th July, 2020 | 2018-CS-50 | | **3.** | Writing into files bit by bit was implemented and then .txt, .cpp, .c, .pdf files were successfully compressed. | 14th July, 2020 | 2018-CS-50 | | **4.** | UI for the project was successfully created in WPF C# .Net Framework. | 25th July, 2020 | 2018-CS-16 | | **5.** | A working project completely integrated with the UI was created. | 27th July, 2020 | 2018-CS-50 | | **6.** | Decompressed class and read bit by bit was updated. | 28th July, 2020 | 2028-CS-16 | | **7.** | .docx file compression was successfully implemented. As word files behave differently than the text files so different measures were taken. A library was used to read the entire content from the .docx file and then DocX package was used to create and write in the word file. | 4th August, 2020 | 2018-CS-50 | | **8.** | Project configuration document was created. More over compression speed which was very slow was 80% increased and UI was updated with some instruction written on the home page. | 5th August, 2020 | 2018-CS-50 | | **9.** | Application was successfully deployed using squirrel Tool. | 11th August, 2020 | 2018-CS-50 | | |
| Have you used built in algorithms or you have implemented yourself? | No, Algorithm has been implemented personally. No C# built in algorithms are used. |
| Formats of input | Application accepts input in the form of a file either for compression or decompression.  File extensions which are allowed are **.pdf, .docx, .txt, .cpp, .cs** |
| Validations | Application accepts the files of all extensions but if the extensions are not supported message is shown. |
| Format of output | Output is also in the form of file and the output file should be saved with the same extension as of input file. |
| Deployment | Squirrel Tool is used for deployment of File Compression Tool.  Reason: Deploying desktop application requires installers that is about to setup exe files or msi files that are basically complex installer files that require different dependencies as well. Contrary to this, Squirrel is a tool provides easy installation of an app, just run and launches the app and whenever update to an application is required, just check for updates and if there just download and install that new version behind the scenes.  The following steps will deploy Desktop Software.  First step is, on your application’s name click on references, then right click on it and from here install squirrel.windows by GitHub.      Second step is, go to properties of your application and then click on  assemblyinfo.cs file and change the assembly version here as 1.0.0 which is basically version number of our application also change the file version to 1.0.0, when the update of application occurs just increment the updated version by 1. Hit ok.    Now, create a NuGet package that squirrel uses for updates. For this go to Microsoft Store and install NuGet Package Explorer.    Next step is to add the Package metadata. Edit the details like id, title with the name of application and add some description as well that shows to the user.  Create a new folder in Package contents with a name lib and in lib folder make another new folder with name net46 which refers to (.Net framework version= v4.6).    Then from Release folder add all files in this folder except .pdb files as it gives access to source code to the user, don’t add these files for data integrity and security. Hit ok and save this NuGet Package Explorer file at the root of your application, where the solution of app is located.  Finally run the command in Package Manager Console and hit Enter. This will generate an extra folder name as Release, where the solution of an application is located.    Finally, user simply runs the Setup.exe file hence the software is install and ready to use. |
| ***Details of algorithms*** | |
| **Huffman Encoding Pseudocode**  **Procedure HuffmanEncoding** (PQ) //PQ is the priority Queue with Letters  S = PQ.Size // and Frequencies set it will be custom built  **while** S is not **equal** **to** 1 **do**  N = new Node ( )  N.left = PQ.pop  N.right = PQ.pop  N.frequency = N.left.frequency + N.right.frequency  PQ.Insert (N)  S = PQ.Size  **end while**  **return** PQ.Top  **Description:**  Huffman Encoding algorithm was proposed by David A. Huffman for compressing text data for the purpose to make a file occupy smaller number of bytes. The idea of Huffman encoding is to leave the rigid 8-bits-per character requirement and use different-length binary encodings called Huffman Codes for different characters.  This algorithm is based on the frequency of the characters that appears in a file and that’s why it is called greedy algorithm based on the Huffman Codes, having least frequent codes are given the longest paths in the tree. The tradeoff is that some characters may need to use encodings that are longer than 8 bits, but this is reserved for characters that occur infrequently, so the extra cost is worth it.  The following are the steps involved in Huffman Encoding to convert a given file (input) into a destination compressed file (output):  The steps involved in Huffman encoding a given text source file into a destination compressed file are:   1. **Count frequencies**: First step is to inspect a source file's contents and then count the number of occurrences of every character. 2. **Build encoding tree**: Then, build a binary tree with a specific structure, where each node is a character representation and its count of occurrences/frequencies in the file. A **priority** queue, data structure is used to help build the tree along the way. 3. Considering the first two nodes having minimum frequency.  * Create a new internal node. * The frequency of this node is the sum of frequency of those two nodes. * Make the first node as a left child and the other node as a right child of the newly created node. * Keep repeating Step-02 and Step-03 until all the nodes form a single tree. * The tree finally obtained is the desired Huffman Tree.  1. **Build encoding map**: Now, traverse the binary tree to explore the binary encodings of each character. 2. **Encode data**: Lastly, re-examine the source file's contents, and for each character, output the encoded binary code of that character to the compressed file. Data in compressed file should be written as bits by bits. For example usually we write a byte which is 8 bits in file so the size is fixed but when variable codes are used more frequently occurring words will have code smaller than 8 bits so size of compressed file is reduced.   **Huffman Decoding Pseudocode**  **Procedure HuffmanDecoding** (fileName, out) **//**out is output file, fileName is Compressed file  Bit = readBitByBit(filename ) //opening compressed file to read bit by bit  Root = ReadHeader(bit) //reading tree from the compressed file  Top = Root  **while** true **do**  **If** Top.left **equal to** NULL **OR** Top.right **equal to** NULL  leaf = Top.value  **If** leaf equal to PSEUDO\_EOF //reached at the end of file  out.Close  **break**  **Else**  Out.Write(leaf)  Top = Root //again start from the tree start  ReturnBit = Bit.bitRead()  **If** ReturnBit is **equal to** 0  Top = Top.left  **If** ReturnBit is **equal to** 1  Top = Top.right  **end while**  **Description:**  In Huffman Decoding, Huffman tree is used to decode text that was previously encoded with its binary patterns or Huffman Codes. Basically, the decoding algorithm is to read each bit from the file, one at a time, and use this bit to traverse the Huffman tree. If the bit is 0, move left in the tree and if the bit is 1, move right. And keep doing this until it hit a leaf node. Leaf nodes represent characters, so once reach there, a leaf, output that character. And then again goes to root. In this algorithm there are different steps which are as follows   * Decoding Procedure is passed compressed file name and output file. * Then compressed file is opened to read bit by bit from it. * At the start of compressed file, Huffman tree is stored as file header. Now tree is stored like whenever a non-leaf node comes 0 is written and when leaf node comes 1 is written followed by the character stored at leaf. * Huffman Tree is retrieved from the compressed file and Root indicates the root of the tree. Then root is temporarily stored in Top. * A while loop runs until a PSEUDO\_EOF is found. * If top left is null and right is also null it means we are at leaf node. Now there are two possibilities one if the character at leaf is PSEUDO\_EOF, it means we have read all characters. So code breaks out of the loop. If not PSEUDO\_EOF the character is written in the output file and Top is again set equal to Root. * A bit from compressed file is read if 0 top becomes top left and if 1 top becomes top right. * When this is done file is decompressed successfully.   **Correctness of Huffman Coding**  Huffman Code uses a greedy approach to generate prefix code T that minimizes the expected length to encode a string. Every code in Huffman Algorithm is unique and not prefix of others. This prefix property is evident by the fact code words are the leaves of binary tree. To put it simply Huffman Algorithm generate optimum prefix codes.  The cost of any encoding tree T is  =  Here is the number of bits required to encode a file. is the frequency of each character x in alphabet C coming in file and is the depth of character x in the Huffman Tree. According to Huffman Coding we get the two characters with smallest frequencies and combine them and so on. We propose that this method will give us which will be smallest than the given by any other methods and we will prove that there is no other way to get smaller than Huffman Coding.  To prove the correctness we will show that any coding tree constructed by some other method can be converted into Huffman Coding Tree without increasing its cost.  **The Claim 1**  Consider two characters and with smallest frequencies then there is an optimal code tree where these two characters are siblings at the maximum depth in tree.  **Proof:**  To prove this we will consider a tree **T** which is not a Huffman Coding Tree. Where characters and are with smallest frequencies but are not at deepest level of three and characters and are at the maximum depth.  C:\Users\probook 430\AppData\Local\Microsoft\Windows\INetCache\Content.Word\1st.png  **T**  Since and are at deepest level so we know that  **≥** &  **≥**  Here represents depth of tree so depth of is greater or equal to depth of and depth of is greater or equal to depth of. Now we can assume that  **- ≥ 0**  **&**  **- ≥ 0**  Thus we can say that  **( - ) . ( -) ≥ 0 ……………… EQU 1**  Now if we **swap** the position of and  **T**  3  We will get a new tree    **3**  If we calculate the cost ofthen  Adding and removing the cost of particular character before and after swap.  **= -**   **+ - +**  By taking common  **= +**   **- - -**  Now we can see that our cost is less than the previous tree.  **= - ( - ) . ( -) ≤**  Because from EQU 1 **( -. ( -) ≥ 0**  Now we clearly see that the cost of new tree will be less than the cost of ***T*** which means that our Tree  **is an optimal tree.**  This cost is reduced by swapping one element we can do same for the ***y*** and    **4**  We will get a new Tree by swapping with      **5**  If we calculate by same method we will get  **= - ( - ) . ( -) ≤**  So is again less thanwhich shows that out tree is very much optimal.  Our cost has also been reduced a lot while we did nothing too much complex just simple swapping and good thing is that we have no need to do this in Huffman coding as it always gives  Optimal Coding Tree.  **Conclusion:**  By looking at the diagram of tree it is proved that our Tree is optimal and and are smallest frequency siblings at the maximum depth of**.**  **The Claim 2**  The Tree for optimal prefix code must be **full** which means that each nodes have exactly two children.  **Proof:**  It is very easy to prove as the answer lies within our encoding method. When we do Huffman encoding we always take two nodes with smallest frequencies and make a new node their parent therefore each node will have exactly two children.  Moreover consider a tree where an internal node does not have exactly two children.    7  What we can do is we can simply replace it with its unique children as it will not affect our tree.  8  **Conclusion:**  We got a full binary tree as it can be seen above. The best thing about Huffman coding is that we would not need to do this replacement as it will always generate full binary tree.  **The Claim 3:**  Huffman algorithm Gives Optimal Prefix Code Tree.  **Proof:**  We need to prove that and the first step that is to combine two smallest frequencies which Huffman algorithm uses is proper to perform. The first step of Huffman algorithm is a **greedy** approach as we choose the smallest frequencies at start and hope for the globally optimal solution which is having Optimal Prefix Code Tree.  We will prove this by **Induction** on **n** where n is total number of characters.   * For the **base case** if **n = 2** then there are two nodes there tree will be   1st-Page-2  Which is already optimal as we cannot get better prefix codes than **0** and **1.**   * What we want to show that it is true for exactly **n** characters.   Page-1  If we have n characters then using previous **claim 1** which states that Consider two characters and with smallest frequencies then there is an optimal code tree where these two characters are siblings at the maximum depth in tree.   * Now Remove and and replace them with a new character**.**     C:\Users\probook 430\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Page-2.png  Frequency **z** will be the combination of characters and frequencies.  =  **+**  Thus **n – 1** character remains. Consider tree with n - 1 characters as**.**   * Replace **z** again with and and consider this tree of **n** charactersas**.** Here we have undone the previous step. * The **Cost** of Tree will be   We are removing **z** node from **n -1** and replacing it with and**.** Depth is because andare the child of **z** so are at one level deep.  **= -**   **+ +**  **= -**  **= -**  **= -**   * The cost of final Tree changes but as we can see that there is no depth in equation of so the change doesn’t depend upon the structure of tree ***T*** ( Tree for n – 1 characters ) * So to minimize this cost of final tree ***T’*** we need to build tree ***T*** on **n-1** characters optimally. * By induction this is exactly what our Huffman algorithm does so the final tree is optimal.   **Conclusion:**  By induction we proved that our final tree will be optimal. Time Complexity Analysis **Huffman Encoding PseudoCode**  **Procedure HuffmanEncoding** (PQ) //PQ is the priority Queue with Letters  1. S = PQ.Size // and Frequencies set it will be custom built  2. **while** S is not **equal** **to** 1 **do**  3. N = new Node ( )  4. N.left = PQ.pop  5. N.right = PQ.pop  6. N.frequency = N.left.frequency + N.right.frequency  7. PQ.Insert (N)  8. S = PQ.Size  9. **end while**  10. **return** PQ.Top    To analyze the running time of Huffman’s algorithm, first: it is noted that algorithm implemented using priority queue.   * 1. In this step, there is a need of calculations of the frequencies for each character. For this step we need to read the entire data source once which will take time of **O(n)**.   2. Using priority queue, an encoded binary tree is constructed. Insertion and deletion operations are being implemented.   3. Using priority queue, an encoded binary tree is constructed. Insertion and deletion operations are being implemented. * Insertion requires **O (lg n)** as it requires to traverse the tree (**max height from root to the leaf node**). Hence, it requires height of the tree cost, which is **lg(n)**. * Similarly, in deletion operation it is required to traverse the tree. Hence, cost of deletion is the height of tree as well which is **lg(n)**.   Overall, each iteration on priority queue requires time **O (lg n)**.  So, by this tree corresponding to a prefix code, the number of bits required to encode a file can easily be computed. For each character say **x** in the alphabet **C,** consider attribute **x.freq** denote the frequency of **x** in the file and let **d┬(x)** denote the depth of x’s leaf in the tree. Note that is also the length of the codeword for character **x**. Thus, the number of bits required to encode a file is thus;  **B(T) =** ∑𝒙∊𝑪 **x.freq. d┬(x)**  which is defined as the cost of the tree T, where **x** belongs **C**.  Keeping in view, all this underlying model its time complexity can be calculated as follows from the above algorithm.  **Step 1:**  **S** is initialized with Priority Queue, taking **O (lg n)** cost in its operations (**insertion and deletion**).  **Step 2:**  In the lines **2** to **9**, the **while** loop executes **n-1** times. Plus, as each priority queue operation requires **O (lg n)**, the loop contributes **O (n lg n)** to the running time. There are **n** iterations, one for each data point so the overall running time complexity **O (n log n)**.  **Summarization:**  In our algorithm the first file for compression take **O (n log n)** running time but for the subsequent files there will no need of calculating frequencies also not a need to maintain the priority queue. Thus, each character will be read from the data source and converted into binary sequences. This will take **linear time** to code the data.    **Space Complexity:**  The conventional algorithm requires space to maintain the priority queue for each data file. In our case the space is fixed and stored priority queue does not change. | |
| ***Bonus Task*** | |
| No extra file is used to store the codes instead Huffman Tree is stored in the compressed file as a header. Due to this reason file to compress shouldn’t be very small and it should be greater than at least 251 bytes large. It is done in 2 steps  **Store Tree**  The idea is basically to traverse tree from top and write 0 in compressed file whenever a non-leaf node is encountered. When a leaf node is hit 1 is stored followed by the letter stored at the leaf node until all the nodes are traversed. This is done recursively.  **Procedure printHeaderTree**(Qtop, Bit) //Qtop is top of queue and Bit is the class to write bit by bit  **if** top **equal** NULL  **RETURN**             if top.leftZero equal NULL AND  top.rightOne equal  NULL                 bit.ByteWrite('1')                 bit.ByteWrite(top.value)  **else**                 bit.ByteWrite('0')                 printHeaderTree(top.leftZero, bit)                 printHeaderTree(top.rightOne, bit)  **Read Tree**  When decompressing the compressed file 1st tree is read from it. The main idea here is that a character is read from file if it is 1 then a new node is created and again a character is read from the file which becomes equal to node value then that node is returned.  If not 1 then 1st left child call the function then right child and then a new node is created with left and write child.  **Procedure ReadTreeHeader**(bit) //bit is read Bit By Bit             c = bit.ByteRead()             node = new PQueue.cNode()  **if** c **equal** '1'                  node = new PQueue.cNode()                  node.value = bit.ByteRead()                  return node  **else**                  leftChild = ReadTreeHeader(bit)                  rightChild = ReadTreeHeader(bit)                  node = new PQueue.cNode ('0', leftChild, rightChild)  **return node** | |
| ***Interfaces for your project*** | |
| This the first UI that shows to the users when they interact with this File Compression Tool.  COMPRESS  HOME  CLOSE BUTTON  MENU TOGGLE BUTTON  DECOMPRESS    When a user clicks/hovers over a toggle menu button then the following UI appears to a user. And the menu list shows to a user. So, when a user clicks HOME icon then the backend hides the current UI screen and WELCOME UI shows to the user.    When a user clicks COMPRESS icon then the following UI shows to the user.  UPLOAD BUTTON    START COMPRESSION BUTTON  SAVE COMPRESSED FILE BUTTON  When a user clicks Upload Button then the following UI appears backend brings user to the system assets or system resources and allows user to select a file that he wants to compress. After selecting file user can upload it for compression.  After uploading file, user hits the start compression button and the uploaded file starts compression.  Finally, when compression is done, a dialogue box appears and success/done message is shown to the user.    After successful compression, user can click a Save Compressed File Button and saves compressed the file with the same extension as it was uploaded for compression and user saves a compressed file with a desired name.    When a user clicks DECOMPRESS icon, the following UI shows to the user.  When a user clicks Upload Button then backend brings user to the system assets/system resources and allows user to select a file that he wants to decompress. After selecting file user can upload it for decompression.  After Uploading file, decompression starts and when decompression is done a success dialogue box shows to the user.  After successful decompression, user can click a Save Decompressed File Button and saves compressed the file with the same extension as it was uploaded for decompression and user saves a decompressed file with a desired name.  SAVE DECOMPRESSED FILE BUTTON  A close button closes the Application.  UPLOAD BUUTON | |
| ***Integration*** | |
| 1. **Garbage values at the end decompressed file**   As in compressed file to write bit by bit we first fill up bits in a byte then write it off. While writing last character it was found that when our byte is not full and end of file comes then some garbage is written at the end.  **Solution**  To solve this problem a variable PSEUDO\_EOF of type integer is defined. Its value is set equal to ASCII 254. It is written at the end of the compressed file so while decompressing when PSEDUO\_EOF is found we stop reading from file in this way no garbage is read from the file.   1. **Very Slow File Compression**   Initially a single character from file were being read at a single time and then to calculate the frequency of letters and store them in dictionary we used **try and catch**, which was used to check that the frequency of character once added should only be incremented and should not be added again in the dictionary. It then threw a lot of exceptions as one character comes many time in a file. So the speed of compression was extremely slow.  **Solution**  To solve this problem we used **if and else** to check whether the character exists in dictionary or not. In this case no more exceptions were thrown so problem was solved and compression became very fast. | |
| ***Change Requests*** | |
| No changes in Huffman Encoding Algorithm were made.  But Huffman Decoding was changed a bit according to the project which are as follows   * Instead of passing the tree to the decoding algorithm, tree was calculated inside. * A check for PSEUDO\_EOF was created, which indicates the end of file to avoid garbage values. | |
| ***Testing*** | |
| In the project guidelines document, it was mentioned that CS311-G50 with reg no 2018-CS-82 is going to perform our project’s testing but she had some personal issues, therefore she couldn’t perform our project’s testing. We mentioned all the details in the mail, sent at [samyan.uet@gmail.com](mailto:samyan.uet@gmail.com), dated August 6th, 2020, you can check it out as well.    As we didn’t get any response till the last date of testing submission. So, we decided to share our problem with Miss Komal. She told us that we have to do our project’s testing by ourselves, when there were just 2 hours left in testing submission. So, we did it and below are the issues that we found and their solutions as well.  **Issue # 1:** desktop application could be deployed or at least do some setup reported by **2018-CS-16**.  **Issue resolved #1:** Application has been successfully deployed. As it is a desktop Application so its windows setup has been created with the help of Squirrel Tool.  **Issue # 2:** .csv file not displaying properly reported by **2018-CS-50**.  **Issue resolved #2:** This issue was caused by using ReadLine, EndOfStream instead of Read and checking condition with -1. it has been solved.  Issue # 3: In case of .docx file if there is this 0x1D hexadecimal value, which is invalid for XML, is in file, then error “ ' ', hexadecimal value 0x1D, is an invalid character” was found and application crashes reported by **2018-CS-16**.  **Issue resolved #3:** Error was coming due to the Usage of Package DocumentFormat.OpenXml which is not best solution to extract the content of word document as it has limitations so instead of it another library Microsoft.Office.Interop.Word was used. | |
| ***Technology*** |  |
| Programming Language | C# |
| Platform | Desktop Application using WPF.NET Framework |