Project Description:

The project aims to perform certain algorithms on a dataset loaded from an Excel file, specifically focusing on identifying connected components, calculating average weights, creating a maximum spanning tree, and exporting the results to Excel files.

Project Functionality:

1. ### Loading Data (LOAD_DATA):

   Reads input data from an Excel file containing file paths and similarity measures.
   Constructs a dictionary representing the similarity graph between files.

2. ### Creating Connected Components (CreateConnectedComponents):

   Utilizes depth-first search (DFS) to identify connected components within the similarity graph.
   Returns a list of lists, each containing nodes belonging to a connected component.

3. ### Calculating Average Weight (CalculateAverageWeight):

   Computes the average similarity weight within each connected component.
   Iterates through relevant edges in the graph and calculates the average weight.

4. ### Converting Graph to List of Graphs (GetEdgesInComponent):

   Segregates the edges of the similarity graph based on connected components.
   Returns a list of dictionaries, each containing edges specific to a component.

5. ### Finding Maximum Spanning Tree (FindMaxSpanningTree):

   Constructs the maximum spanning tree (MST) for each connected component.
   Utilizes a modified Kruskal's algorithm to find the MST with respect to similarity weights.

6. ### Exporting Results to Excel (ConvertDictionaryToExcel, ConvertListToExcel):

   Converts the computed results to Excel format for further analysis and visualization.
   Handles the export of component data and MST edges along with relevant metadata.

Average Run Time on hard test

functionality	Best performance time
Read a excel file and Construct the Graph.	972ms
Create connected components.	11ms
Calculate average weight for each component	127ms
Prepare graph and Find Max Spanning Tree	250ms
Convert Group statistics output to excel file	157ms
Convert matching pairs output to excel file	1007ms
State file generation & save	295ms
MST Generation & save include Kruskal	1757ms

Best Performance time for Total time: 3750ms

Time Complexity Analysis:

1. Load data:

```
1 reference
private static Dictionary<Tuple<string, string>, Tuple<double, double, double>> LOAD_DATA()
{
    string path = @"D:\GAM3A\ALGOOO\PROJECT\[3] Plagiarism Validation\Test Cases\complete\Hard\1-Input.xlsx";
    var matchingPairs = new Dictionary<Tuple<string, string>, Tuple<double, double, double>>();

    FileInfo fileInfo = new FileInfo(path);
    using (var package = new ExcelPackage(fileInfo))
    {
        ExcelWorksheet worksheet = package.Workbook.Worksheets[0]; // First worksheet
        int rowCount = worksheet.Dimension.Rows;

        for (int row = 2; row <= rowCount; row++)
        {
            if (worksheet.Cells[row, 1].Value == null || worksheet.Cells[row, 2].Value == null)
                break;
            string file1Path = worksheet.Cells[row, 1].Text;
            string file2Path = worksheet.Cells[row, 2].Text;
            string Linesmatch = worksheet.Cells[row, 3].Text;
            string[] parts1 = file1Path.Split('/');
            string[] parts2 = file2Path.Split('/');
            string similarityPart1 = Regex.Replace(parts1[parts1.Length - 1], "[^0-9]", "");
            string similarityPart2 = Regex.Replace(parts2[parts2.Length - 1], "[^0-9]", "");
            string idPart1 = Regex.Replace(parts1[parts1.Length - 2], "[^0-9]", "");
            string idPart2 = Regex.Replace(parts2[parts2.Length - 2], "[^0-9]", "");

            string hashing1 = idPart1 + Linesmatch;
            string hashing2 = idPart2 + Linesmatch;
            string hashing_unique1 = idPart1+idPart2+Linesmatch;

            HASHH[hashing1] = file1Path;
            HASHH[hashing2] = file2Path;
            Tuple<string,string> rawan = Tuple.Create(file1Path,file2Path);
            HASHH_unique[hashing_unique1] = rawan;

            double weightOne = double.Parse(similarityPart1);
            double weightTwo = double.Parse(similarityPart2);

            Tuple<string, string> pairKey = Tuple.Create(idPart1, idPart2);
            Tuple<double, double, double> weights = Tuple.Create(weightOne, weightTwo, double.Parse(Linesmatch));
            matchingPairs.Add(pairKey, weights);
        }
    }
    return matchingPairs;
}
```

Load Data (LOAD_DATA):

This function is used to load the input data from the given files that will be converted in a specific way and returned as a dictionary of two tuples <Tuple, Tuple>

• The code consists of loop that iterates throw all edges(pairs), so the complexity of this loop is O(E)

• The body of the loop consists of assignments, object creation, and basic file system interactions (amortized) so the complexity of the body of the loop is constant time (O(1))

• Overall complexity: complexity of loop * complexity of the body = O(E)*O(1) = O(E)

```
internal class Program
{
    static Dictionary<string, string> HASHH = new Dictionary<string, string>();
    static Dictionary<string, string> HASHH_mst = new Dictionary<string, string>();
    static Dictionary<string, Tuple<string,string>> HASHH_unique = new Dictionary<string,Tuple<string,string>>();
```

In the Load Data section, HASHH and HASHH_unique dictionaries are filled to swiftly access file paths, enhancing subsequent data parsing efficiency.

2. Create Connected Components:

```
public static List<List<string>> CreateConnectedComponents(Dictionary<Tuple<string, string>, Tuple<double, double, double>> edges)
{
    Dictionary<string, List<string>> adjacencyList = new Dictionary<string, List<string>>();
    HashSet<string> visitedNodes = new HashSet<string>();
    List<List<string>> components = new List<List<string>>();

    // Build adjacency list
    foreach (var edge in edges)
    {
        string node1 = edge.Key.Item1;
        string node2 = edge.Key.Item2;

        if (!adjacencyList.ContainsKey(node1))
        {
            adjacencyList[node1] = new List<string>();
        }
        adjacencyList[node1].Add(node2);

        if (!adjacencyList.ContainsKey(node2))
        {
            adjacencyList[node2] = new List<string>();
        }
        adjacencyList[node2].Add(node1);
    }

    // Depth-First Search to find and mark all connected components
    void DFS(string node, List<string> component)
    {
        Stack<string> stack = new Stack<string>();
        stack.Push(node);
        visitedNodes.Add(node);

        while (stack.Count > 0)
        {
            string current = stack.Pop();
            component.Add(current);

            foreach (string neighbor in adjacencyList[current])
            {
                if (!visitedNodes.Contains(neighbor))
                {
                    visitedNodes.Add(neighbor);
                    stack.Push(neighbor);
                }
            }
        }
    }

    // Initialize DFS for components not yet visited
    foreach (string node in adjacencyList.Keys)
    {
        if (!visitedNodes.Contains(node))
        {
            List<string> component = new List<string>();
            DFS(node, component);
            components.Add(component);
        }
    }

    return components;
}
```

This function takes a dictionary representing a graph (edges) and identifies its connected components.

Building Adjacency List (Outer Loop):

- Iterates through all edges in edges (up to E times).
- Performs constant time operations for key lookups and adding elements to lists within the loop.
- Overall time complexity: O(E)

Iterating over Unvisited Nodes (Outer Loop):

- Iterates through all nodes present as keys in the adjacency list (up to V times in the worst case).
- Inside the loop, a BFS call is made for unvisited nodes.

DFS Calls:

- In the worst case, the DFS function might be called for each node (V times).
- The BFS function itself has a time complexity of O(E_v + V_v), where:
    - E_v: Number of edges connected to the current node (bounded by the node's degree)
    - V_v: Number of nodes reachable from the current node (depends on the graph structure)
- In the worst-case scenario, a DFS call could explore all edges and nodes in the graph. However, this typically only happens for the first DFS call on a large connected component. Subsequent calls on disconnected components will explore fewer edges and nodes

Combining Complexities:

- The outer loop iterates at most V times.
- Within each iteration, the DFS call has a worst-case complexity of O(E_v + V_v).
- Since the total number of edges (E) and nodes (V) considered across all DFS calls is bounded by the original graph, the overall complexity becomes:
    - O(V * (E_v + V_v))
    - This simplifies to O(VE + V^2) due to constant factors.
- In practice, for most graphs, the average value of E_v (number of edges per node) is less than V (total number of nodes). This leads to a dominant term of O(VE).
- Additionally, the outer loop might terminate early if all connected components are found before iterating through all nodes.

Therefore, considering both theoretical and practical aspects, the overall time complexity of CreateConnectedComponents is considered to be **O(E + V)**.

Top Run Time in Hard Test: 11ms.

3. Calculate Average Weight:

```
1 reference
static (double, int) CalculateAverageWeight(List<string> component, Dictionary<Tuple<string, string>, Tuple<double, double, double>> graph)
{
    // Convert the component list to a set for faster lookups
    HashSet<string> componentNodes = new HashSet<string>(component);// O(1)
    double totalWeight = 0.0;// O(1)
    int edgeCount = 0;// O(1)

    // Iterate only over the subset of the graph that is relevant to the component
    foreach (var edge in graph) //O(E)
    {
        string node1 = edge.Key.Item1;// O(1)
        string node2 = edge.Key.Item2;// O(1)

        // Check if both nodes are in the component
        if (componentNodes.Contains(node1) && componentNodes.Contains(node2))// O(1)
        {
            totalWeight += (edge.Value.Item1 + edge.Value.Item2);// O(1)
            edgeCount++;// O(1)
        }
    }

    // Calculate the average, accounting for the double counting of weight in bidirectional graphs
    return edgeCount > 0 ? (totalWeight / (edgeCount * 2), component.Count) : (0.0, 0);// O(1)
}
```

Description:

In this function we iterate over all edges in graph and check if it found in the current connected component so this function aims to calculate the average weight for the current connected component.

Overall Complexity:

Since the dominant factor is the loop iterating over relevant edges (O(E_c)), and other operations are constant time, the overall complexity of the function is O(E_c). This reflects the fact that the function's execution time depends primarily on the number of edges within the specific connected component being analyzed.

Where **E_c**: Number of edges in the subgraph induced by the connected component.

Top Run Time in Hard Test: 127ms, this time with iterate on all connected components in main().

4. Convert The Graph to List of Dictionary

```
1 reference
static List<Dictionary<Tuple<string, string>, Tuple<double, double, double>>> GetEdgesInComponent(
List<List<string>> connectedComponents, Dictionary<Tuple<string, string>, Tuple<double, double, double>> edges)//overall O(V + E)
{
    // Initialize a list of dictionaries to store edges for each component
    List<Dictionary<Tuple<string, string>, Tuple<double, double, double>>> componentEdgesList =
        new List<Dictionary<Tuple<string, string>, Tuple<double, double, double>>>(connectedComponents.Count);// O(1)

    // Create a dictionary to map each node to its component index
    Dictionary<string, int> nodeComponentIndex = new Dictionary<string, int>(); // O(1)

    // Fill the nodeComponentIndex dictionary with the component index of each node

    // O(V)
    for (int i = 0; i < connectedComponents.Count; i++)
    {
        foreach (var node in connectedComponents[i])
        {
            nodeComponentIndex[node] = i; //O(1)
        }
        // Initialize an empty dictionary for the current component
        componentEdgesList.Add(new Dictionary<Tuple<string, string>, Tuple<double, double, double>>()); //O(1)
    }

    // Iterate over each edge and add it to the corresponding component's dictionary
    foreach (var edge in edges) //O(E)
    {
        string node1 = edge.Key.Item1; //O(1)
        string node2 = edge.Key.Item2; //O(1)

        // Check if both nodes of the edge belong to the same component
        if (nodeComponentIndex.TryGetValue(node1, out int componentIndex1) &&
            nodeComponentIndex.TryGetValue(node2, out int componentIndex2) &&
            componentIndex1 == componentIndex2) //O(1)
        {
            // Add the edge to the dictionary of the corresponding component
            componentEdgesList[componentIndex1][edge.Key] = edge.Value; //O(1)
        }
    }

    return componentEdgesList; //O(1)
}
```

Description:

In this block of code, we loop on all vertices in each connected component and map each vertex to its component index, then loop on all edges to get all edges in each connected component.

Overall Complexity:

`GetEdgesInComponent` function efficiently processes edges and assigns them to their respective connected component dictionaries with a time complexity of O(V + E). This reflects its scalability for handling graphs with a moderate number of nodes and edges.

Top Run Time in Hard Test: 6ms.

5. Convert The Graph to List of Dictionary

```
3 references
public class DisjointSet<T>
{
    // Stores the parent of each element for efficient path compression (amortized O(1))
    private Dictionary<T, T> parent;
    // Stores the rank of each element for union by rank optimization (O(1))
    private Dictionary<T, int> rank;
    1 reference
    public DisjointSet(IEnumerable<T> elements)
    {
        parent = new Dictionary<T, T>();
        rank = new Dictionary<T, int>();
        foreach (var element in elements)
        {
            parent[element] = element;  // O(1) amortized for insertion to dictionary
            rank[element] = 0;          // O(1) amortized for insertion to dictionary
        }
        // Overall complexity: O(V) where V is the number of elements in connected componet
    }
    5 references
    public T Find(T element)
    {
        if (!parent.ContainsKey(element))
        {
            throw new KeyNotFoundException("Element not found in the disjoint set.");
        }
        if (!element.Equals(parent[element]))
            parent[element] = Find(parent[element]);
        return parent[element];
    }
    // Overall complexity: O(1) amortized due to path compression in worst case it will be O(V)
    1 reference
    public void Union(T element1, T element2)
    {
        var root1 = Find(element1); // O(1) amortized due to path compression
        var root2 = Find(element2); // O(1) amortized due to path compression
        if (!root1.Equals(root2))
        {
            if (rank[root1] < rank[root2])
            {
                parent[root1] = root2; // O(1) amortized for dictionary update
            }
            else if (rank[root1] > rank[root2])
            {
                parent[root2] = root1; // O(1) amortized for dictionary update
            }
            else
            {
                parent[root2] = root1;   // O(1) amortized for dictionary update
                rank[root1]++;           // O(1) amortized for dictionary update
            }
        }
        // Overall complexity: O(1) amortized due to constant time operations and path compression
    }
}
```

This class implements a Disjoint-Set data structure, also known as a Union-Find data structure.

1. **Constructor (DisjointSet(IEnumerable<T> elements)):**
    - Initializes parent and rank dictionaries (O(1) for initialization).
    - Iterates through elements and initializes each element as its own parent (root) with rank 0. Dictionary insertions have an amortized time complexity of O(1).
    - **Overall complexity:** O(N) where N is the number of elements.

2. **Find(T element):**
    - Checks if element exists in the parent dictionary. If not, throws an exception.
    - Implements path compression:
        - If element is not its own parent, recursively calls Find on its parent. This compresses the path to the root by setting the parent of each element along the path to the root itself.
    - Returns the root element of the set containing element.

    Overall complexity:

    O(1) amortized due to path compression. In the worst case (deeply nested tree), it could be O(V) where V is the number of elements in the connected component.

3. **Union(T element1, T element2):**
    - Finds the roots of the sets containing element1 and element2 using Find.
    - If the roots are not the same, performs union by rank:
        - If rank[root1] < rank[root2], sets the parent of root1 to root2.
        - If rank[root1] > rank[root2], sets the parent of root2 to root1.
        - If rank[root1] == rank[root2], sets the parent of root2 to root1 and increments rank[root1] to favor shallower trees.
    - Dictionary updates for parent and rank have an amortized time complexity of O(1).
    - **Overall complexity:** O(1) amortized due to constant time operations and path compression.

Complexity Analysis:

- The constructor takes O(N) time to initialize the data structure for N elements.
- Find has an amortized complexity of O(1) due to path compression, but in the worst case (deeply nested tree), it could be O(V).
- Union has an amortized complexity of O(1) due to constant time operations and path compression.

6. Find MST For each component

```
1 reference
public static List<Dictionary<Tuple<string, string>, double>> FindMaxSpanningTree(
List<List<string>> connectedComponents,
List<Dictionary<Tuple<string, string>, Tuple<double, double, double>>> connectedNodes)
{
    // Initialize an empty list to store MST edges (O(1))
    List<Dictionary<Tuple<string, string>, double>> mstEdges = new List<Dictionary<Tuple<string, string>, double>>();

    // Iterate through each connected component (O(N) where N is the number of components)
    foreach (var component in connectedComponents)
    {
        // Find the dictionary of edges for the current component (O(M) on average, worst case O(N))
        Dictionary<Tuple<string, string>, Tuple<double, double, double>> edges =
          connectedNodes.FirstOrDefault(nodes => nodes.Keys.Select(k => k.Item1).Intersect(component).Any());

        if (edges == null)
        {
            continue; // O(1)
        }

        // Create a list of edges from the component's dictionary (O(M) where M is the number of edges in the component)
        List<Tuple<Tuple<string, string>, Tuple<double, double, double>>> edgeList = edges.Select(kvp => Tuple.Create(kvp.Key, kvp.Value)).ToList();

        // Sort the edges in descending order of weights (O(M * log M) - average and worst case for most sorting algorithms)
        edgeList.Sort((x, y) =>
        {
            int comparison = Math.Max(y.Item2.Item1, y.Item2.Item2).CompareTo(Math.Max(x.Item2.Item1, x.Item2.Item2));
            if (comparison == 0)
            {
                comparison = y.Item2.Item3.CompareTo(x.Item2.Item3);
            }
            return comparison;
        });

        // Initialize a DisjointSet for the component (O(M) where M is the number of nodes in the component)
        DisjointSet<string> disjointSet = new DisjointSet<string>(component);

        // Create a dictionary to store MST edges for the current component (O(1))
        Dictionary<Tuple<string, string>, double> mstComponentEdges = new Dictionary<Tuple<string, string>, double>();

        // Iterate through sorted edges (O(M) in total)
        foreach (var edge in edgeList)
        {
            string node1 = edge.Item1.Item1;
            string node2 = edge.Item1.Item2;

            // Find representatives for both nodes (O(1) amortized)
            if (disjointSet.Find(node1) != disjointSet.Find(node2))
            {
                // Add the edge to the MST component (O(1) amortized)
                mstComponentEdges[edge.Item1] = edge.Item2.Item3;
                // These lines (HASHH_mst) seem like potential unused code and are commented out
                // HASHH_mst[node1 + edge.Item2.Item3] = HASHH[node1 + edge.Item2.Item3];
                // HASHH_mst[node2 + edge.Item2.Item3] = HASHH[node2 + edge.Item2.Item3];

                // Union the sets (O(1) amortized)
                disjointSet.Union(node1, node2);
            }
        }

        // Add the MST component edges to the overall MST edges (O(1))
        mstEdges.Add(mstComponentEdges);
    }

    return mstEdges; // O(1)
}
```

here's the complexity analysis of the FindMaxSpanningTree function:

Loop Over Connected Components: The function iterates over each connected component. Let's denote the number of connected components as C. This contributes O(C) to the complexity.

Finding Edges for Each Component: For each connected component, there's a search operation to find the associated edges. If the number of connected nodes is N so the complexity is O(N).

Sorting Edges: After retrieving the edges for each component, they are sorted based on their weights. Sorting takes O(ElogE) where E is the number of edges in the component.

Disjoint Set Operations: Within each connected component, disjoint set operations (Find and Union) are performed for each edge. These operations usually have an amortized complexity of nearly O(1) , but across all edges, it becomes O(Eα(V)), where V is the number of vertices.

Considering all these steps, the overall complexity of the function is: $O(C \cdot (N+ElogE+E\alpha(V)))$.

Top Run Time in Hard Test: 244ms.

7. Convert data to Excel File

```
1 reference
public static void ConvertDictionaryToExcel(List<Tuple<Dictionary<Tuple<string, string>, double>, double>> data)
{
    // Check for empty data (O(1))
    if (data.Count == 0)
    {
        Console.WriteLine("No data to write to Excel.");
        return;
    }

    // Define file path (O(1))
    string filePath = @"D:\Output1000.xlsx";
    FileInfo fileInfo = new FileInfo(filePath);

    // Delete existing file (O(1) amortized)
    if (fileInfo.Exists)
    {
        fileInfo.Delete();
    }

    // Create Excel package (O(1))
    using (var package = new ExcelPackage(fileInfo))
    {
        // Create worksheet (O(1))
        var worksheet = package.Workbook.Worksheets.Add("Data");

        // Setting up headers (O(1)) - Three constant time cell assignments
        worksheet.Cells["A1"].Value = "First String";
        worksheet.Cells["B1"].Value = "Second String";
        worksheet.Cells["C1"].Value = "Double Value";
```

Description:

This code block performs necessary setup tasks before writing data to an Excel file.

Overall Complexity:

The dominant complexity is constant time (O(1)) due to the nature of variable assignments, object creation, and basic file system interactions (amortized). These operations are generally fast and independent of the data size.

Top Run Time in Hard Test: 1149ms with hyperlink, without it run in 423ms.

```
        //---> over all complexity in this two loops is O(E) => E is number of edges

        // Loop through data items (O(N)) - N is the number of items in 'data'--> connected commponents
        int row = 2;
        foreach (var item in data)
        {
            // Loop through key-value pairs (O(M)) - M is the maximum key-node per item
            foreach (var kvp in item.Item1)
            {
                // Construct key string (O(1) assumed) - Assuming reasonable key length
                string key1 = kvp.Key.Item1 + kvp.Key.Item2 + kvp.Value.ToString();

                // Check for key existence (O(1) amortized) - Assuming hash table for HASHH_unique
                if (!HASHH_unique.ContainsKey(key1))
                {
                    continue;
                }

                // Prepare row data (O(1)) - Three constant time assignments
                string value1 = HASHH_unique[key1].Item1;
                string value2 = HASHH_unique[key1].Item2;
                double value3 = kvp.Value;

                // Add row data to worksheet (O(1)) - Three constant time cell assignments
                worksheet.Cells[row, 1].Value = value1;
                worksheet.Cells[row, 2].Value = value2;
                worksheet.Cells[row, 3].Value = value3;

                // Check and add hyperlink (O(1) on success, O(N) on failure per value)
                // - N is the string length for Uri.TryCreate
                Uri uri1, uri2;
                if (Uri.TryCreate(value1, UriKind.Absolute, out uri1))
                {
                    worksheet.Cells[row, 1].Hyperlink = uri1;
                    worksheet.Cells[row, 1].Style.Font.UnderLine = true;
                    worksheet.Cells[row, 1].Style.Font.Color.SetColor(System.Drawing.Color.Blue);
                }

                if (Uri.TryCreate(value2, UriKind.Absolute, out uri2))
                {
                    worksheet.Cells[row, 2].Hyperlink = uri2;
                    worksheet.Cells[row, 2].Style.Font.UnderLine = true;
                    worksheet.Cells[row, 2].Style.Font.Color.SetColor(System.Drawing.Color.Blue);
                }

                // Increment row (O(1))
                row++;
            }
        }

        // Save the changes (O(1) amortized)
        package.Save();
        Console.WriteLine("Excel file created or overwritten at: " + filePath);
    }
}
```

block takes a list of items (data) and writes the data to an Excel file ("Output1000.xlsx").

1. Outer Loop:
    - It iterates through each item in the connected components list.
2. Inner Loop:
    - For each edge in component, it iterates through it.

- So, this part iterates over all edges to convert it to excel file, and it is complexity is O(E).

3. Hyperlink Addition (O(1) on success, O(N) on failure per value)):
    - It attempts to convert each extracted value (value1 and value2) into a valid Uri object using Uri.TryCreate. This is a constant time operation (O(1)) if successful.
    - If successful, it adds a hyperlink to the corresponding cell and formats the font for better presentation.

Overall Complexity:

The dominant complexity of this code block is O(E) due to the nested loops, because the if I combine the input and convert it to one dictionary this will give me same number of the original dictionary of edges.

Top Run Time in Hard Test: 1149ms with hyperlink, without it run in 423ms.

8. Convert data to excel file.

```
1 reference
public static void ConvertListToExcel(List<Tuple<List<string>, double, int>> data)
{
    string filePath = @"D:\Output1.xlsx";
    FileInfo fileInfo = new FileInfo(filePath);

    // If the file exists and we want to overwrite it, we can delete it beforehand
    if (fileInfo.Exists)
    {
        fileInfo.Delete(); // Delete the existing file to ensure a fresh workbook
    }

    // Using the package with a new FileInfo object, which will create a new file
    using (var package = new ExcelPackage(fileInfo))
    {
        // Create a worksheet
        string sheetName = "Data";
        var worksheet = package.Workbook.Worksheets.Add(sheetName);

        // Setting up headers

        worksheet.Cells["A1"].Value = "Counter";
        worksheet.Cells["B1"].Value = "Component";
        worksheet.Cells["C1"].Value = "Average Similarity";
        worksheet.Cells["D1"].Value = "Count";

        int row = 2;
        int i = 1;
        foreach (var item in data)
        {
            List<string> component = item.Item1;
            double averageSimilarity = item.Item2;
            int count = item.Item3;

            // Assign values to cells
            worksheet.Cells[row, 2].Value = string.Join(", ", component);
            worksheet.Cells[row, 3].Value = Math.Round(averageSimilarity, 1);
            worksheet.Cells[row, 4].Value = count;
            worksheet.Cells[row, 1].Value = i;
            row++;
            i++;
        }

        // Save the changes to a new file
        package.Save();
        Console.WriteLine("Excel file created or overwritten at: " + filePath);
    }
}
```

- **File Handling:**
  - FileInfo creation and potential deletion: O(1)
  - Creating new ExcelPackage: O(1) (assuming constant time for initialization)
- **Worksheet Creation:**
  - Adding a worksheet: O(1)
- **Header Setup:**
  - Setting values for 4 header cells: O(1)
- **Looping through Data:**

- Iterating through N elements in the data list: O(N)
- Within the loop:
  - Accessing list elements: O(1) (assuming constant time access)
  - String concatenation for component: O(M) where M is the average number of strings in a component (depends on the data). However, this is dominated by the loop itself.
  - Math operations: O(1)
  - Assigning values to cells: O(1) per cell (assuming constant time for writing to cells)

- **Saving Changes:**
  - package.Save(): O(1) (assuming constant time for saving)

Overall:

The complexity is dominated by the loop iterating through the data list (O(N)) and the potential string concatenation within the loop (which can be considered O(M) in the worst case, but is likely much smaller than N in most cases). Therefore, the overall complexity can be simplified to **O(N)**.

9. Program Main

```
//------------------------------------------------------------------------------------------------------------------------
//-----------------------------------------------------Load Data----------------------------------------------------------
Stopwatch loadDataWatch = new Stopwatch();
loadDataWatch.Start();
Dictionary<Tuple<string, string>, Tuple<double, double, double>> matchingPairs = LOAD_DATA();
loadDataWatch.Stop();
Console.WriteLine($"LOAD_DATA executed in {loadDataWatch.ElapsedMilliseconds} ms");
//------------------------------------------------------------------------------------------------------------------------
//-------------------------------------------------Create Connected Components--------------------------------------------
Stopwatch createComponentsWatch = new Stopwatch();
createComponentsWatch.Start();
List<List<string>> components = CreateConnectedComponents(matchingPairs);
createComponentsWatch.Stop();
Console.WriteLine($"CreateConnectedComponents executed in {createComponentsWatch.ElapsedMilliseconds} ms");
//------------------------------------------------------------------------------------------------------------------------
```

```
//------------------------------------------------------------------------------------------------------------------------
//-----------------------------------------------Calculate Average Weight and Count---------------------------------------
Stopwatch calculateWeightWatch = new Stopwatch();
calculateWeightWatch.Start();
List<Tuple<List<string>, double, int>> Connected_Components_with_Average_Weight =
new List<Tuple<List<string>, double, int>>();
foreach (List<string> component in components)
{
    int count;
    double averageWeight;
    (averageWeight, count) = CalculateAverageWeight(component, matchingPairs);
    Tuple<List<string>, double, int> Connected_ = Tuple.Create(component, averageWeight, count);
    Connected_Components_with_Average_Weight.Add(Connected_);
}
calculateWeightWatch.Stop();
Console.WriteLine($"CalculateAverageWeight executed in {calculateWeightWatch.ElapsedMilliseconds} ms");
//------------------------------------------------------------------------------------------------------------------------
```

- Loop on all component and calculate average weight

```
//-------------------------------------------------------------------------------------------------------------------------------------------
//-----------------------------------------Sorting First OutPut-----------------------------------------------------------------------------
List<Tuple<List<string>, double, int>> sortedList = Connected_Components_with_Average_Weight.OrderByDescending(t => t.Item2).ToList();
List<Tuple<List<string>, double, int>> Listt = new List<Tuple<List<string>, double, int>>();
foreach (var list in sortedList)
{
    List<string> hi = list.Item1;
    List<string> sortedHi = hi.OrderBy(x => int.Parse(x)).ToList();
    Tuple<List<string>, double, int> Connected_ = Tuple.Create(sortedHi, list.Item2, list.Item3);
    Listt.Add(Connected_);
}
//-------------------------------------------------------------------------------------------------------------------------------------------
```

- Sort the components in the statistics file by their average similarity in descending order.
- Sort elements within each component in ascending order based on the numeric part in the path.

Complexity analysis:

Sorting `sortedList based on its average similarity` (O(C log C)).

Looping through `sortedList` (O(C)), and each time order the inner list based on its numerical id O(V_c).

The dominant factor in the complexity is the nested loops. While there's a sort within the inner loop, the outer loop iterates C times, making the overall complexity **O(C)**. In most cases, C (number of outer elements) will be significantly larger than V_c (number of elements in the inner list).

```
//-------------------------------------------------To Excel Sheet-----------------------------------------------------------------------------
Stopwatch ListToExcel = new Stopwatch();
ListToExcel.Start();
ConvertListToExcel(Listt);
ListToExcel.Stop();
Console.WriteLine($"Convert List To Excel executed in {ListToExcel.ElapsedMilliseconds} ms");
//-------------------------------------------------------------------------------------------------------------------------------------------
//-----------------------------------------Convert Data To Use In MST-----------------------------------------------------------------------
Stopwatch calculate = new Stopwatch();
calculate.Start();
List<Dictionary<Tuple<string, string>, Tuple<double, double, double>>> componentEdges =
GetEdgesInComponent(components, matchingPairs);
calculate.Stop();
Console.WriteLine($"CalculateAverageWeight executed in {calculate.ElapsedMilliseconds} ms");
//-------------------------------------------------------------------------------------------------------------------------------------------
//-----------------------------------------------------------MST-----------------------------------------------------------------------------
Stopwatch findMSTWatch = new Stopwatch();
findMSTWatch.Start();
List<Dictionary<Tuple<string, string>, double>> data1;
data1 = FindMaxSpanningTree(components, componentEdges);
findMSTWatch.Stop();
Console.WriteLine($"FindMaxSpanningTree executed in {findMSTWatch.ElapsedMilliseconds} ms");
//-------------------------------------------------------------------------------------------------------------------------------------------
```

- Here converts the first output to excel file.
- Convert data to use it in MST function and put it in componentEdges.
- Then use MST function and put the output in data1.

```
//-----------------------------------------------------Sorrt Second Output----------------------
List<Tuple<Dictionary<Tuple<string, string>, double>, double>> dataaa =
new List<Tuple<Dictionary<Tuple<string, string>, double>, double>>(data1.Count);
//---------sort el data mn gwa--------
for (int i = 0; i < data1.Count; i++)
{
    var dictionary = data1[i];
    var sortedKeyValuePairs = dictionary.OrderByDescending(kvp => kvp.Value).ToList();
    Dictionary<Tuple<string, string>, double> sortedDictionary =
    sortedKeyValuePairs.ToDictionary(kvp => kvp.Key, kvp => kvp.Value);
}
```

Here I loop on all connected components and sort all edges within it based on its average similarity.

Loop complexity is ➔ O(Connected components).

To sort its take ➔ O(E_C Log(E_C)). Assume that E_C is number of edges with in component.

```
static Dictionary<string, string> HASHH_mst = new Dictionary<string, string>();
```

- We make this dictionary to add on it all nodes after make MST Function.

```
// Find representatives for both nodes (O(1) amortized)
if (disjointSet.Find(node1) != disjointSet.Find(node2))
{
    // Add the edge to the MST component (O(1) amortized)
    mstComponentEdges[edge.Item1] = edge.Item2.Item3;
    HASHH_mst[node1 + edge.Item2.Item3] = HASHH[node1 + edge.Item2.Item3];
    HASHH_mst[node2 + edge.Item2.Item3] = HASHH[node2 + edge.Item2.Item3];

    // Union the sets (O(1) amortized)
    disjointSet.Union(node1, node2);
}
```

- Here we fill the dictionary by values in MST function.

```
//----------sort el componants nfsha--------------
for (int i = 0; i < componentEdges.Count; i++)
{
    var dictionary = componentEdges[i];
    var sortedKeyValuePairs = dictionary.OrderByDescending(kvp => kvp.Value.Item3).ToList();
    Dictionary<Tuple<string, string>, Tuple<double, double, double>> sortedDictionary =
        sortedKeyValuePairs.ToDictionary(kvp => kvp.Key, kvp => kvp.Value);
    //-------------------------------------------------------------------------

    double sum = 0;
    double average = 0;
    int counter = 0;
    string[] parts1, parts2;
    List<Tuple<string, string>> keysToRemove = new List<Tuple<string, string>>();
```

- now we loop on each connected component and sort the components internally based on its Lines match, but this sort for edges before make MST.

- And then initialize some important variables.
- Loop complexity is ➔ O(Connected components).
- To sort its take ➔ O(E_C Log(E_C)). Assume that E_C is number of edges with in component.

```
foreach (var kvp in sortedDictionary)
{
    string hashing1 = kvp.Key.Item1 + kvp.Value.Item3.ToString();
    string hashing2 = kvp.Key.Item2 + kvp.Value.Item3.ToString();
    string node1 = "";
    string node2 = "";
    bool flag = false;

    try
    {
        string node5 = HASHH_mst[hashing1];
        string node6 = HASHH_mst[hashing2];
        flag = true;
    }
    catch
    {
        flag = false;
    }
```

- Then we loop internally on sorted dictionary and check if this edge found after making MST or not, and have Boolean flag to recognize that.
- Which is take time complexity O(V_C).

```
node1 = HASHH[hashing1];
node2 = HASHH[hashing2];

parts1 = node1.Split('/');
parts2 = node2.Split('/');

// Take the second-to-last part, which should contain the ID
string similarityPart1 = Regex.Replace(parts1[parts1.Length - 1], "[^0-9]", "");
string similarityPart2 = Regex.Replace(parts2[parts2.Length - 1], "[^0-9]", "");
double weightOne = double.Parse(similarityPart1);
double weightTwo = double.Parse(similarityPart2);
counter += 2;
sum += weightOne + weightTwo;

if (!flag)
{
    keysToRemove.Add(kvp.Key);
}
```

- Then we get the node and parse it to get the weight One and Weight Two.
- And increment counter by 2, and get the sum.
- if this edge not found after making MST add it to KeysToRemove List.
- For each statement here have O(1) Complexity.

```
    // Remove keys from sortedDictionary
    foreach (var keyToRemove in keysToRemove)
    {
        sortedDictionary.Remove(keyToRemove);
    }

    average = sum / counter;

    Dictionary<Tuple<string, string>, double> khello = new Dictionary<Tuple<string, string>, double>();
    foreach (var x in sortedDictionary)
    {
        khello[x.Key] = x.Value.Item3;
    }

    Tuple<Dictionary<Tuple<string, string>, double>, double> componentData = Tuple.Create(khello, average);
    dataaa.Add(componentData);
}
dataaa = dataaa.OrderByDescending(item => item.Item2).ToList();
```

- Remove all keys not found in After MST.
- And calculate the average similarity of current component.
- Add all to dataaa in each time.

  After that sort all component based on its average similarity.

```
//-----------------------------------------------To Excel Sheet2-----------------------------------------------------------------------
Stopwatch convertExcelWatch = new Stopwatch();
convertExcelWatch.Start();
ConvertDictionaryToExcel(dataaa);
convertExcelWatch.Stop();
Console.WriteLine($"ConvertDictionaryToExcel executed in {convertExcelWatch.ElapsedMilliseconds} ms");
//-------------------------------------------------------------------------------------------------------------------------------------
```

- Then call this function to convert output to Excel file.