Source Code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Data;
using System.IO;
using ExcelDataReader;
using System.Text.RegularExpressions;
using System.ComponentModel;
using OfficeOpenXml;
using System.Collections;
using OfficeOpenXml.Style;
using OfficeOpenXml.Table;
using System.Diagnostics;

namespace PlagiarismValidation
{
    class Program
    {
        static void Main(string[] args)
        {
            Stopwatch Program_stopwatch = new Stopwatch();
            Program_stopwatch.Start();

            Dictionary<string,string> edge_with_its_hyper_link = new
Dictionary<string,string>();
            Tuple<string, string, int, int, int>[] edges = ReadFromExcelFile(ref
edge_with_its_hyper_link);
            Dictionary<KeyValuePair<string, string>, Tuple<int, int>> allEdges =
new Dictionary<KeyValuePair<string, string>, Tuple<int, int>>();
            Dictionary<string, List<Tuple<string, int, int, int>>> elements = new
Dictionary<string, List<Tuple<string, int, int, int>>>(); // edges with two values
            Dictionary<string, int> colored_vertices = new Dictionary<string, int>
();//for BFS
            Dictionary<string, List<string>> componentsLst = new Dictionary<string,
List<string>>(); //groups
            // statistics
            Dictionary<string, Tuple<float,int>> firstVandAvg = new
Dictionary<string, Tuple<float, int>>();
            List<Dictionary<KeyValuePair<string, string>, Tuple<int, int>>>
```

```
Components = new List<Dictionary<KeyValuePair<string, string>, Tuple<int, int>>>();
            List<Dictionary<KeyValuePair<string, string>, Tuple<int, int>>>
refinedGroups = new List<Dictionary<KeyValuePair<string, string>, Tuple<int, int>>>
();

            ConstructingTheGraph(edges, elements, colored_vertices, ref allEdges);

            int numberOfEdges = 0;
            float componentAVG = 0;
            int list_index = 0;
            Stopwatch bfs_stopwatch = new Stopwatch();
            bfs_stopwatch.Start();
            foreach (var vertex in elements) // V
            {
                componentAVG = 0;

                if (colored_vertices[vertex.Key] == 0)
                {
                    List<string> component = new List<string>();
                    Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
edges_of_components = new Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
();
                    BFS(vertex.Key, ref elements, ref colored_vertices, ref
component, ref numberOfEdges, ref componentAVG, ref edges_of_components); // V / 2
+ E
                    componentsLst.Add(vertex.Key, component);
                    Components.Add(edges_of_components);
                    Tuple<float, int> tuple = new Tuple<float, int>(componentAVG,
list_index);
                    firstVandAvg.Add(vertex.Key, tuple);
                    list_index++;
                }
            }
            bfs_stopwatch.Stop();
            TimeSpan ts = bfs_stopwatch.Elapsed;
            double totalSeconds = (ts.TotalHours * 60 * 60) + (ts.TotalMinutes *
60) + ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
            Console.WriteLine($"Elapsed Time for BFS: {totalSeconds} seconds");

            firstVandAvg = firstVandAvg.OrderByDescending(pair =>
pair.Value.Item1).ToDictionary(pair => pair.Key, pair => pair.Value);
            Stopwatch Kruskal_stopwatch = new Stopwatch();
            Kruskal_stopwatch.Start();
            foreach (string firstString in firstVandAvg.Keys)
            {
                Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
```

```
refinedcompnent = new Dictionary<KeyValuePair<string, string>, Tuple<int, int>>();
                Kruskal(Components[firstVandAvg[firstString].Item2], ref
refinedcompnent);
                refinedGroups.Add(refinedcompnent);
            }
            Kruskal_stopwatch.Stop();
            TimeSpan kts = Kruskal_stopwatch.Elapsed;
            double totalSecondsk = (kts.TotalHours * 60 * 60) + (kts.TotalMinutes *
60) + kts.TotalSeconds + (kts.TotalMilliseconds / 1000);
            Console.WriteLine($"Elapsed Time for Kruskal Algorithm: {totalSecondsk}
seconds");

            OutPut_Of_Stat(ref firstVandAvg, ref componentsLst , totalSeconds);
            OutPut_Of_MST(refinedGroups, ref allEdges , ref
edge_with_its_hyper_link , totalSecondsk);

            Program_stopwatch.Stop();
            TimeSpan tts = Program_stopwatch.Elapsed;
            double totalSecondst = (tts.TotalHours * 60 * 60) + (tts.TotalMinutes *
60) + tts.TotalSeconds + (tts.TotalMilliseconds / 1000);
            Console.WriteLine($"Elapsed Time for the whole program: {totalSecondst}
seconds");
        }

        public static void BFS(string vertex, ref Dictionary<string,
List<Tuple<string, int, int, int>>> graphDictionary, ref Dictionary<string, int>
colored_vertices, ref List<string> component, ref int numberOfEdges, ref float
componentAVG, ref Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
edges_of_components)
        {

            colored_vertices[vertex] = 1;
            Queue<string> bfsQueue = new Queue<string>();

            float avgScore = 0;
            bfsQueue.Enqueue(vertex);
            component.Add(vertex);

            numberOfEdges = 0;
            while (bfsQueue.Count != 0)
            {
                string newVertex = bfsQueue.Dequeue();
                List<Tuple<string, int, int, int>> adjacencyList =
graphDictionary[newVertex];
                foreach (var vertexTuple in adjacencyList)
                {
```

```
                    numberOfEdges++;
                    if (colored_vertices[vertexTuple.Item1] == 0)  // White
                    {

                        colored_vertices[vertexTuple.Item1] = 1; // Gray
                        component.Add(vertexTuple.Item1);
                        KeyValuePair<string, string> edge = new
KeyValuePair<string, string>(newVertex, vertexTuple.Item1);
                        if (vertexTuple.Item2 > vertexTuple.Item3)
                        {
                            // Item3 --> Min
                            Tuple<int, int> tuple = new Tuple<int, int>
(vertexTuple.Item2, vertexTuple.Item4);
                            edges_of_components[edge] = tuple;
                        }
                        else
                        {
                            // Item2 --> Min
                            Tuple<int, int> tuple = new Tuple<int, int>
(vertexTuple.Item3, vertexTuple.Item4);
                            edges_of_components[edge] = tuple;
                        }
                        avgScore += vertexTuple.Item2 + vertexTuple.Item3;
                        bfsQueue.Enqueue(vertexTuple.Item1);

                    }

                    else if (colored_vertices[vertexTuple.Item1] == 1)
                    {
                        KeyValuePair<string, string> edge = new
KeyValuePair<string, string>(newVertex, vertexTuple.Item1);
                        if (vertexTuple.Item2 > vertexTuple.Item3)
                        {
                            // Item3 --> Min
                            Tuple<int, int> tuple = new Tuple<int, int>
(vertexTuple.Item2, vertexTuple.Item4);
                            edges_of_components[edge] = tuple;
                        }
                        else
                        {
                            // Item2 --> Min
                            Tuple<int, int> tuple = new Tuple<int, int>
(vertexTuple.Item3, vertexTuple.Item4);
                            edges_of_components[edge] = tuple;
                        }
                        avgScore += vertexTuple.Item2 + vertexTuple.Item3;
```

```
                    }
                }

                colored_vertices[newVertex] = 2; // Black

            }
            componentAVG = avgScore / numberOfEdges;
            componentAVG = (float)Math.Round(componentAVG, 1);



        }
        public static Tuple<string, string, int, int, int>[] ReadFromExcelFile(ref
Dictionary<string, string> edge_with_its_hyper_link)
        {
            Stopwatch read_excel_stopwatch = new Stopwatch();
            read_excel_stopwatch.Start();
            //string inputfilePath = "D:\\Uni
Related\\Algorithms\\Project\\MATERIALS\\[3] Plagiarism Validation\\Algorithm-
Project\\PlagiarismValidation\\Test Cases\\Sample\\6-Input.xlsx";
            string inputfilePath = "F:\\Year 3 2nd term\\Analysis and Design of
Algorithm\\Project\\Algorithm-Project\\PlagiarismValidation\\Test
Cases\\Complete\\Hard\\1-Input.xlsx";
            int numberOfEdges;
            Tuple<string, string, int, int, int>[] edges;

            using (var stream = File.Open(inputfilePath, FileMode.Open,
FileAccess.Read))
            {
                IExcelDataReader reader = null;

                reader = ExcelReaderFactory.CreateReader(stream);

                DataSet resultDataSet = reader.AsDataSet();

                DataTable table = resultDataSet.Tables[0];

                string column1 = "";
                string column2 = "";
                string column3 = "";

                int linesMatched;

                numberOfEdges = table.Rows.Count - 1;
                //Console.WriteLine(numberOfEdges);
                edges = new Tuple<string, string, int, int, int>[numberOfEdges];
```

```
                for (int i = 1; i < table.Rows.Count; i++)
                {

                    DataRow row = table.Rows[i];

                    column1 = row[0].ToString();

                    // Retrieving the similarity percentage of each document in
column 1
                    int indexOfbrac1 = column1.LastIndexOf('(');
                    string firstPath = column1.Substring(0, indexOfbrac1);
                    int firstPercntage = int.Parse(column1.Substring(indexOfbrac1 +
1, column1.Length - (indexOfbrac1 + 3)));

                    column2 = row[1].ToString();

                    // Retrieving the similarity percentage of each document in
column 2
                    int indexOfbrac2 = column2.LastIndexOf('(');
                    string secondPath = column2.Substring(0, indexOfbrac2);
                    int secondPercntage = int.Parse(column2.Substring(indexOfbrac2
+ 1, column2.Length - (indexOfbrac2 + 3)));

                    // Retrieving the Lines Matched in column 3
                    column3 = row[2].ToString();
                    linesMatched = Convert.ToInt32(column3);

                    edges[i - 1] = new Tuple<string, string, int, int, int>
(firstPath, secondPath, firstPercntage, secondPercntage, linesMatched);

                }

                if (reader != null)
                {
                    reader.Close();
                    reader.Dispose();
                }

            }
            using (var package = new ExcelPackage(new FileInfo(inputfilePath)))
            {
                var worksheet = package.Workbook.Worksheets[0];

                for (int i = worksheet.Dimension.Start.Row; i <=
worksheet.Dimension.End.Row; i++)
                {
```

```
                    var cell1 = worksheet.Cells[i, 1];
                    var cell2 = worksheet.Cells[i, 2];
                    if (cell1.Hyperlink != null)
                    {
                        string hyperlink = cell1.Hyperlink.AbsoluteUri;
                        edge_with_its_hyper_link[cell1.Text] = hyperlink;
                    }
                    if (cell2.Hyperlink != null)
                    {
                        string hyperlink = cell2.Hyperlink.AbsoluteUri;
                        edge_with_its_hyper_link[cell2.Text] = hyperlink;

                    }
                }
            }
            read_excel_stopwatch.Stop();
            TimeSpan ts = read_excel_stopwatch.Elapsed;
            double totalSeconds = (ts.TotalHours * 60 * 60) + (ts.TotalMinutes *
60) + ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
            Console.WriteLine($"Elapsed Time for Reading Excel file: {totalSeconds}
seconds");
            return edges;
        }
        public static void ConstructingTheGraph(Tuple<string, string, int, int,
int>[] edges, Dictionary<string, List<Tuple<string, int, int, int>>> elements,
Dictionary<string, int> colored_vertices, ref Dictionary<KeyValuePair<string,
string>, Tuple<int, int>> allEdges)
        {
            Stopwatch Constructing_stopwatch = new Stopwatch();
            Constructing_stopwatch.Start();
            int maximum = 0;
            //the first float number is for percentage of doc 1 to doc 2 (form the
first vertex to the second vertex) (edge item 3)
            //the second float number is for percentage of doc 2 to doc 1 (form the
second vertex to the first vertex) (edge item 4)
            foreach (var edge in edges)
            {
                maximum = Math.Max(edge.Item3, edge.Item4);
                if (elements.ContainsKey(edge.Item1))
                {
                    KeyValuePair<string, string> newEdgeToBeAdded = new
KeyValuePair<string, string>(edge.Item1, edge.Item2);
                    elements[edge.Item1].Add(Tuple.Create(edge.Item2, edge.Item3,
edge.Item4, edge.Item5));
                    allEdges[newEdgeToBeAdded] = (Tuple.Create(edge.Item3,
edge.Item4));
```

```
                }
                else
                {
                    KeyValuePair<string, string> newEdgeToBeAdded = new
KeyValuePair<string, string>(edge.Item1, edge.Item2);
                    elements[edge.Item1] = new List<Tuple<string, int, int, int>>()
{ Tuple.Create(edge.Item2, edge.Item3, edge.Item4, edge.Item5) };
                    colored_vertices[edge.Item1] = 0;
                    allEdges[newEdgeToBeAdded] = (Tuple.Create(edge.Item3,
edge.Item4));
                }
                if (elements.ContainsKey(edge.Item2))
                {

                    elements[edge.Item2].Add(Tuple.Create(edge.Item1, edge.Item3,
edge.Item4, edge.Item5));
                }
                else
                {
                    elements[edge.Item2] = new List<Tuple<string, int, int, int>>()
{ Tuple.Create(edge.Item1, edge.Item3, edge.Item4, edge.Item5) };
                    colored_vertices[edge.Item2] = 0;
                }

            }
            Constructing_stopwatch.Stop();
            TimeSpan ts = Constructing_stopwatch.Elapsed;
            double totalSeconds = (ts.TotalHours * 60 * 60) + (ts.TotalMinutes *
60) + ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
            Console.WriteLine($"Elapsed Time for constructing the graph:
{totalSeconds} seconds");
        }
        public static void Kruskal(Dictionary<KeyValuePair<string, string>,
Tuple<int, int>> component, ref Dictionary<KeyValuePair<string, string>, Tuple<int,
int>> refinedGroups)
        {
            Dictionary<string, int> enumForVertices = new Dictionary<string, int>
();
            int count = 0;
            foreach (KeyValuePair<string, string> edge in component.Keys)
            {
                if (!enumForVertices.ContainsKey(edge.Key))
                {
                    enumForVertices.Add(edge.Key, count);
                    count++;
                }
```

```
                if (!enumForVertices.ContainsKey(edge.Value))
                {
                    enumForVertices[edge.Value] = count;
                    count++;
                }
            }
            SetsWithArray set_for_Kruskal = new SetsWithArray(count);
            for (int i = 0; i < count; i++)
            {
                set_for_Kruskal.Make_Set(i);
            }
            Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
sortedcomponent = component.OrderByDescending(pair => pair.Value).ToDictionary(pair
=> pair.Key, pair => pair.Value);
            foreach (var edge in sortedcomponent)
            {
                if (set_for_Kruskal.Find_Set(enumForVertices[edge.Key.Key]) !=
set_for_Kruskal.Find_Set(enumForVertices[edge.Key.Value]))
                {

                    refinedGroups[edge.Key] = sortedcomponent[edge.Key];
                    set_for_Kruskal.Union_Set(enumForVertices[edge.Key.Key],
enumForVertices[edge.Key.Value]);
                }
            }
        }
        public static void OutPut_Of_MST(List<Dictionary<KeyValuePair<string,
string>, Tuple<int, int>>> refinedGroups, ref Dictionary<KeyValuePair<string,
string>, Tuple<int, int>> allEdges , ref Dictionary<string, string>
edge_with_its_hyper_link , double timeKruskal)
        {
            Stopwatch mst_file_stopwatch = new Stopwatch();
            mst_file_stopwatch.Start();
            ExcelPackage.LicenseContext =
OfficeOpenXml.LicenseContext.NonCommercial;
            ExcelPackage excelPackage = new ExcelPackage();

            ExcelWorksheet mstSheet = excelPackage.Workbook.Worksheets.Add("MST
1");

            mstSheet.Cells[1,1].Value = "File 1";
            mstSheet.Cells[1,2].Value = "File 2";
            mstSheet.Cells[1,3].Value = "Line Matches";


            int i = 1;
```

```
            foreach (Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
group in refinedGroups)// number of components
            {
                //mstSheet.Cells[i + 1, 1].Value = group;
                Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
sorted_group = group.OrderByDescending(pair => pair.Value.Item2).ToDictionary(pair
=> pair.Key, pair => pair.Value);

                foreach (KeyValuePair<string, string> kvp in sorted_group.Keys) //
E of each refined component
                {
                    KeyValuePair<string, string> kvp2 = new KeyValuePair<string,
string>(kvp.Value, kvp.Key);
                    if (allEdges.ContainsKey(kvp))
                    {
                        string filePath1 = kvp.Key + '(' + allEdges[kvp].Item1 +
"%)";
                        string filePath2 = kvp.Value + '(' + allEdges[kvp].Item2 +
"%)";

                        mstSheet.Cells[i + 1, 1].Value = filePath1;
                        if (edge_with_its_hyper_link.ContainsKey(filePath1))
                        {
                            mstSheet.Cells[i + 1, 1].Hyperlink = new
Uri(edge_with_its_hyper_link[filePath1]);
                        }
                        mstSheet.Cells[i + 1, 2].Value = filePath2;
                        if (edge_with_its_hyper_link.ContainsKey(filePath2))
                        {
                            mstSheet.Cells[i + 1, 2].Hyperlink = new
Uri(edge_with_its_hyper_link[filePath2]);
                        }
                        mstSheet.Cells[i + 1, 3].Value = sorted_group[kvp].Item2;
                    }
                    else if (allEdges.ContainsKey(kvp2))
                    {
                        string filePath1 = kvp2.Key + '(' + allEdges[kvp2].Item1 +
"%)";
                        string filePath2 = kvp2.Value + '(' + allEdges[kvp2].Item2
+ "%)";

                        mstSheet.Cells[i + 1, 1].Value = filePath1;
                        if (edge_with_its_hyper_link.ContainsKey(filePath1))
                        {
                            mstSheet.Cells[i + 1, 1].Hyperlink = new
Uri(edge_with_its_hyper_link[filePath1]);
```

```
                        }
                        mstSheet.Cells[i + 1, 2].Value = filePath2;
                        if (edge_with_its_hyper_link.ContainsKey(filePath2))
                        {
                            mstSheet.Cells[i + 1, 2].Hyperlink = new
Uri(edge_with_its_hyper_link[filePath2]);
                        }
                        mstSheet.Cells[i + 1, 3].Value = sorted_group[kvp].Item2;
                    }
                    i++;
                }

            }

            string outputFilePath = @"F:\Year 3 2nd term\Analysis and Design of
Algorithm\Project\Algorithm-Project\PlagiarismValidation\Output\File2.xlsx";
            excelPackage.SaveAs(new System.IO.FileInfo(outputFilePath));

            mst_file_stopwatch.Stop();
            TimeSpan ts = mst_file_stopwatch.Elapsed;
            double totalSeconds = timeKruskal + (ts.TotalHours * 60 * 60) +
(ts.TotalMinutes * 60) + ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
            Console.WriteLine($"Elapsed Time for calculating and saving MST  file:
{totalSeconds} seconds");
        }
        public static void OutPut_Of_Stat(ref Dictionary<string, Tuple<float, int>>
firstVandAvg, ref Dictionary<string, List<string>> componentsLst , double time_BFS)
        {
            Stopwatch stat_file_stopwatch = new Stopwatch();
            stat_file_stopwatch.Start();
            ExcelPackage.LicenseContext =
OfficeOpenXml.LicenseContext.NonCommercial;
            ExcelPackage excelPackage = new ExcelPackage();

            ExcelWorksheet statisticsSheet =
excelPackage.Workbook.Worksheets.Add("Statistics 1");

            statisticsSheet.Cells[1,1].Value = "Component Index";
            statisticsSheet.Cells[1,2].Value = "Vertices";
            statisticsSheet.Cells[1,3].Value = "Average Similarity";
            statisticsSheet.Cells[1,4].Value = "Component Count";

            //Dictionary<string, float> sortedFirstVandAvg =
firstVandAvg.OrderByDescending(pair => pair.Value).ToDictionary(pair => pair.Key,
pair => pair.Value);
```

```
            int i = 1;
            int counter = 1;
            // O(Components log(Components) + componentItems Log(componentItems))

            foreach (var vertex in firstVandAvg) // --> no of component --> worst
case V/2 -- Best Case --> 1 time
            {
                statisticsSheet.Cells[i + 1, 1].Value = counter;
                statisticsSheet.Cells[i + 1, 3].Value = vertex.Value.Item1;
                List<string> component = componentsLst[vertex.Key];

                //component.Sort(); // O(vlogv)
                // +d
                List<int> componentItemsList = new List<int>();
                Regex digitsRegex = new Regex("\\d+");
                string componentItems = "";
                // matchPercentage = percentageRegex.Match(column1);

                foreach (var item in component) // O(V)
                {
                    Match digitsRegexMatch = digitsRegex.Match(item);

componentItemsList.Add(Convert.ToInt32(digitsRegexMatch.Value));
                }
                componentItemsList.Sort(); // O(vlogv)
                foreach (var item in componentItemsList) // O(V)
                {
                    componentItems = componentItems + item.ToString() + ",";
                }
                componentItems = componentItems.Remove(componentItems.Length - 1);
                statisticsSheet.Cells[i + 1, 2].Value = componentItems;
                statisticsSheet.Cells[i + 1, 4].Value = component.Count;
                i++;
                counter++;
            }

            //string outputFilePath = @"D:\Uni
Related\Algorithms\Project\MATERIALS\[3] Plagiarism Validation\Algorithm-
Project\PlagiarismValidation\Output\File.xlsx";
            string outputFilePath = @"F:\Year 3 2nd term\Analysis and Design of
Algorithm\Project\Algorithm-Project\PlagiarismValidation\Output\File.xlsx";
            excelPackage.SaveAs(new System.IO.FileInfo(outputFilePath));

            stat_file_stopwatch.Stop();
            TimeSpan ts = stat_file_stopwatch.Elapsed;
            double totalSeconds = time_BFS + (ts.TotalHours * 60 * 60) +
```

```
(ts.TotalMinutes * 60) + ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
            Console.WriteLine($"Elapsed Time for calculating and saving statistics
file: {totalSeconds} seconds");
        }
    }
}
```

First Defining Data Structures:

```
Stopwatch Program_stopwatch = new Stopwatch();
Program_stopwatch.Start();

Dictionary<string,string> edge_with_its_hyper_link = new Dictionary<string,string>
();
Tuple<string, string, int, int, int>[] edges = ReadFromExcelFile(ref
edge_with_its_hyper_link);
Dictionary<KeyValuePair<string, string>, Tuple<int, int>> allEdges = new
Dictionary<KeyValuePair<string, string>, Tuple<int, int>>();
Dictionary<string, List<Tuple<string, int, int, int>>> elements = new
Dictionary<string, List<Tuple<string, int, int, int>>>(); // edges with two values
Dictionary<string, int> colored_vertices = new Dictionary<string, int>();//for BFS
Dictionary<string, List<string>> componentsLst = new Dictionary<string,
List<string>>(); //groups
// statistics
Dictionary<string, Tuple<float,int>> firstVandAvg = new Dictionary<string,
Tuple<float, int>>();
List<Dictionary<KeyValuePair<string, string>, Tuple<int, int>>> Components = new
List<Dictionary<KeyValuePair<string, string>, Tuple<int, int>>>();
List<Dictionary<KeyValuePair<string, string>, Tuple<int, int>>> refinedGroups = new
List<Dictionary<KeyValuePair<string, string>, Tuple<int, int>>>();
```

Second Reading Excel File Function:

```
 public static Tuple<string, string, int, int, int>[] ReadFromExcelFile(ref
Dictionary<string, string> edge_with_its_hyper_link)
 {
     Stopwatch read_excel_stopwatch = new Stopwatch();
     read_excel_stopwatch.Start();
     //string inputfilePath = "D:\\Uni Related\\Algorithms\\Project\\MATERIALS\\[3]
Plagiarism Validation\\Algorithm-Project\\PlagiarismValidation\\Test
Cases\\Sample\\6-Input.xlsx";
     string inputfilePath = "F:\\Year 3 2nd term\\Analysis and Design of
Algorithm\\Project\\Algorithm-Project\\PlagiarismValidation\\Test
Cases\\Complete\\Hard\\1-Input.xlsx";
```

```
    int numberOfEdges;
    Tuple<string, string, int, int, int>[] edges;

    using (var stream = File.Open(inputfilePath, FileMode.Open, FileAccess.Read))
    {
        IExcelDataReader reader = null;

        reader = ExcelReaderFactory.CreateReader(stream);

        DataSet resultDataSet = reader.AsDataSet();

        DataTable table = resultDataSet.Tables[0];

        string column1 = "";
        string column2 = "";
        string column3 = "";

        int linesMatched;

        numberOfEdges = table.Rows.Count - 1;
        //Console.WriteLine(numberOfEdges);
        edges = new Tuple<string, string, int, int, int>[numberOfEdges];

        for (int i = 1; i < table.Rows.Count; i++)
        {

            DataRow row = table.Rows[i];

            column1 = row[0].ToString();

            // Retrieving the similarity percentage of each document in column 1
            int indexOfbrac1 = column1.LastIndexOf('(');
            string firstPath = column1.Substring(0, indexOfbrac1);
            int firstPercntage = int.Parse(column1.Substring(indexOfbrac1 + 1,
column1.Length - (indexOfbrac1 + 3)));

            column2 = row[1].ToString();

            // Retrieving the similarity percentage of each document in column 2
            int indexOfbrac2 = column2.LastIndexOf('(');
            string secondPath = column2.Substring(0, indexOfbrac2);
            int secondPercntage = int.Parse(column2.Substring(indexOfbrac2 + 1,
column2.Length - (indexOfbrac2 + 3)));

            // Retrieving the Lines Matched in column 3
            column3 = row[2].ToString();
```

```
            linesMatched = Convert.ToInt32(column3);

            edges[i - 1] = new Tuple<string, string, int, int, int>(firstPath,
secondPath, firstPercntage, secondPercntage, linesMatched);

        }

        if (reader != null)
        {
            reader.Close();
            reader.Dispose();
        }

    }
    using (var package = new ExcelPackage(new FileInfo(inputfilePath)))
    {
        var worksheet = package.Workbook.Worksheets[0];

        for (int i = worksheet.Dimension.Start.Row; i <=
worksheet.Dimension.End.Row; i++)
        {
            var cell1 = worksheet.Cells[i, 1];
            var cell2 = worksheet.Cells[i, 2];
            if (cell1.Hyperlink != null)
            {
                string hyperlink = cell1.Hyperlink.AbsoluteUri;
                edge_with_its_hyper_link[cell1.Text] = hyperlink;
            }
            if (cell2.Hyperlink != null)
            {
                string hyperlink = cell2.Hyperlink.AbsoluteUri;
                edge_with_its_hyper_link[cell2.Text] = hyperlink;

            }
        }
    }
    read_excel_stopwatch.Stop();
    TimeSpan ts = read_excel_stopwatch.Elapsed;
    double totalSeconds = (ts.TotalHours * 60 * 60) + (ts.TotalMinutes * 60) +
ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
    Console.WriteLine($"Elapsed Time for Reading Excel file: {totalSeconds}
seconds");
    return edges;
 }
```

Third Constructing the Graph Function:

First the function call in the main:

```
ConstructingTheGraph(edges, elements, colored_vertices, ref allEdges);
```

Second The function:

```
public static void ConstructingTheGraph(Tuple<string, string, int, int, int>[]
edges, Dictionary<string, List<Tuple<string, int, int, int>>> elements,
Dictionary<string, int> colored_vertices, ref Dictionary<KeyValuePair<string,
string>, Tuple<int, int>> allEdges)
{
    Stopwatch Constructing_stopwatch = new Stopwatch();
    Constructing_stopwatch.Start();
    int maximum = 0;
    //the first float number is for percentage of doc 1 to doc 2 (form the first
vertex to the second vertex) (edge item 3)
    //the second float number is for percentage of doc 2 to doc 1 (form the second
vertex to the first vertex) (edge item 4)
    foreach (var edge in edges)
    {
        maximum = Math.Max(edge.Item3, edge.Item4);
        if (elements.ContainsKey(edge.Item1))
        {
            KeyValuePair<string, string> newEdgeToBeAdded = new
KeyValuePair<string, string>(edge.Item1, edge.Item2);
            elements[edge.Item1].Add(Tuple.Create(edge.Item2, edge.Item3,
edge.Item4, edge.Item5));
            allEdges[newEdgeToBeAdded] = (Tuple.Create(edge.Item3, edge.Item4));
        }
        else
        {
            KeyValuePair<string, string> newEdgeToBeAdded = new
KeyValuePair<string, string>(edge.Item1, edge.Item2);
            elements[edge.Item1] = new List<Tuple<string, int, int, int>>() {
Tuple.Create(edge.Item2, edge.Item3, edge.Item4, edge.Item5) };
            colored_vertices[edge.Item1] = 0;
            allEdges[newEdgeToBeAdded] = (Tuple.Create(edge.Item3, edge.Item4));
        }
        if (elements.ContainsKey(edge.Item2))
        {

            elements[edge.Item2].Add(Tuple.Create(edge.Item1, edge.Item3,
```

```
edge.Item4, edge.Item5));
        }
        else
        {
            elements[edge.Item2] = new List<Tuple<string, int, int, int>>() {
Tuple.Create(edge.Item1, edge.Item3, edge.Item4, edge.Item5) };
            colored_vertices[edge.Item2] = 0;
        }

    }
    Constructing_stopwatch.Stop();
    TimeSpan ts = Constructing_stopwatch.Elapsed;
    double totalSeconds = (ts.TotalHours * 60 * 60) + (ts.TotalMinutes * 60) +
ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
    Console.WriteLine($"Elapsed Time for constructing the graph: {totalSeconds}
seconds");
}
```

Fourth BFS Function:

First the function call in main:

```
int numberOfEdges = 0;
float componentAVG = 0;
int list_index = 0;
Stopwatch bfs_stopwatch = new Stopwatch();
bfs_stopwatch.Start();
foreach (var vertex in elements) // V
{
    componentAVG = 0;

    if (colored_vertices[vertex.Key] == 0)
    {
        List<string> component = new List<string>();
        Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
edges_of_components = new Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
();
        BFS(vertex.Key, ref elements, ref colored_vertices, ref component, ref
numberOfEdges, ref componentAVG, ref edges_of_components); // V / 2 + E
        componentsLst.Add(vertex.Key, component);
        Components.Add(edges_of_components);
        Tuple<float, int> tuple = new Tuple<float, int>(componentAVG, list_index);
        firstVandAvg.Add(vertex.Key, tuple);
        list_index++;
    }
```

```
}
bfs_stopwatch.Stop();
TimeSpan ts = bfs_stopwatch.Elapsed;
double totalSeconds = (ts.TotalHours * 60 * 60) + (ts.TotalMinutes * 60) +
ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
Console.WriteLine($"Elapsed Time for BFS: {totalSeconds} seconds");
```

Second the function:

```
public static void BFS(string vertex, ref Dictionary<string, List<Tuple<string,
int, int, int>>> graphDictionary, ref Dictionary<string, int> colored_vertices, ref
List<string> component, ref int numberOfEdges, ref float componentAVG, ref
Dictionary<KeyValuePair<string, string>, Tuple<int, int>> edges_of_components)
{

    colored_vertices[vertex] = 1;
    Queue<string> bfsQueue = new Queue<string>();

    float avgScore = 0;
    bfsQueue.Enqueue(vertex);
    component.Add(vertex);

    numberOfEdges = 0;
    while (bfsQueue.Count != 0)
    {
        string newVertex = bfsQueue.Dequeue();
        List<Tuple<string, int, int, int>> adjacencyList =
graphDictionary[newVertex];
        foreach (var vertexTuple in adjacencyList)
        {
            numberOfEdges++;
            if (colored_vertices[vertexTuple.Item1] == 0)  // White
            {

                colored_vertices[vertexTuple.Item1] = 1; // Gray
                component.Add(vertexTuple.Item1);
                KeyValuePair<string, string> edge = new KeyValuePair<string,
string>(newVertex, vertexTuple.Item1);
                if (vertexTuple.Item2 > vertexTuple.Item3)
                {
                    // Item3 --> Min
                    Tuple<int, int> tuple = new Tuple<int, int>(vertexTuple.Item2,
vertexTuple.Item4);
                    edges_of_components[edge] = tuple;
                }
```

```
                else
                {
                    // Item2 --> Min
                    Tuple<int, int> tuple = new Tuple<int, int>(vertexTuple.Item3,
vertexTuple.Item4);
                    edges_of_components[edge] = tuple;
                }
                avgScore += vertexTuple.Item2 + vertexTuple.Item3;
                bfsQueue.Enqueue(vertexTuple.Item1);

            }

            else if (colored_vertices[vertexTuple.Item1] == 1)
            {
                KeyValuePair<string, string> edge = new KeyValuePair<string,
string>(newVertex, vertexTuple.Item1);
                if (vertexTuple.Item2 > vertexTuple.Item3)
                {
                    // Item3 --> Min
                    Tuple<int, int> tuple = new Tuple<int, int>(vertexTuple.Item2,
vertexTuple.Item4);
                    edges_of_components[edge] = tuple;
                }
                else
                {
                    // Item2 --> Min
                    Tuple<int, int> tuple = new Tuple<int, int>(vertexTuple.Item3,
vertexTuple.Item4);
                    edges_of_components[edge] = tuple;
                }
                avgScore += vertexTuple.Item2 + vertexTuple.Item3;
            }
        }

        colored_vertices[newVertex] = 2; // Black

    }
    componentAVG = avgScore / numberOfEdges;
    componentAVG = (float)Math.Round(componentAVG, 1);


}
```

Fifth the Kruskal Algorithm:

First the function call in main:

```

```

Second the function:

```
public static void Kruskal(Dictionary<KeyValuePair<string, string>, Tuple<int,
int>> component, ref Dictionary<KeyValuePair<string, string>, Tuple<int, int>>
refinedGroups)
{
    Dictionary<string, int> enumForVertices = new Dictionary<string, int>();
    int count = 0;
    foreach (KeyValuePair<string, string> edge in component.Keys)
    {
        if (!enumForVertices.ContainsKey(edge.Key))
        {
            enumForVertices.Add(edge.Key, count);
            count++;
        }
        if (!enumForVertices.ContainsKey(edge.Value))
        {
            enumForVertices[edge.Value] = count;
            count++;
        }
    }
    SetsWithArray set_for_Kruskal = new SetsWithArray(count);
    for (int i = 0; i < count; i++)
    {
        set_for_Kruskal.Make_Set(i);
    }
    Dictionary<KeyValuePair<string, string>, Tuple<int, int>> sortedcomponent =
component.OrderByDescending(pair => pair.Value).ToDictionary(pair => pair.Key, pair
=> pair.Value);
    foreach (var edge in sortedcomponent)
    {
        if (set_for_Kruskal.Find_Set(enumForVertices[edge.Key.Key]) !=
set_for_Kruskal.Find_Set(enumForVertices[edge.Key.Value]))
        {

            refinedGroups[edge.Key] = sortedcomponent[edge.Key];
            set_for_Kruskal.Union_Set(enumForVertices[edge.Key.Key],
enumForVertices[edge.Key.Value]);
        }
```

```
    }
}
```

Third the class set used in Kruskal:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Web;

namespace PlagiarismValidation
{
    internal class SetsWithArray
    {
        private int[] members;
        public SetsWithArray(int count)
        {
            this.members = new int[count];
        }

        public void Make_Set(int vertex_index)
        {
            members[vertex_index] = vertex_index;
        }

        public int Find_Set(int vertex_index)
        {
            return members[vertex_index];
        }

        public void Union_Set(int u, int v)
        {
            int cluster = this.members[u];
            for (int i = 0; i < this.members.Length; i++)
            {
                if (this.members[i] == cluster)
                {
                    this.members[i] = this.members[v];
                }
            }
        }
    }
```

```
}
```

Fifth the Statistics File output Function:

First the function call in main

```
OutPut_Of_Stat(ref firstVandAvg, ref componentsLst , totalSeconds);
```

Second the Function:

```
public static void OutPut_Of_Stat(ref Dictionary<string, Tuple<float, int>>
firstVandAvg, ref Dictionary<string, List<string>> componentsLst , double time_BFS)
{
    Stopwatch stat_file_stopwatch = new Stopwatch();
    stat_file_stopwatch.Start();
    ExcelPackage.LicenseContext = OfficeOpenXml.LicenseContext.NonCommercial;
    ExcelPackage excelPackage = new ExcelPackage();

    ExcelWorksheet statisticsSheet =
excelPackage.Workbook.Worksheets.Add("Statistics 1");

    statisticsSheet.Cells[1,1].Value = "Component Index";
    statisticsSheet.Cells[1,2].Value = "Vertices";
    statisticsSheet.Cells[1,3].Value = "Average Similarity";
    statisticsSheet.Cells[1,4].Value = "Component Count";

    //Dictionary<string, float> sortedFirstVandAvg =
firstVandAvg.OrderByDescending(pair => pair.Value).ToDictionary(pair => pair.Key,
pair => pair.Value);

    int i = 1;
    int counter = 1;
    // O(Components log(Components) + componentItems Log(componentItems))

    foreach (var vertex in firstVandAvg) // --> no of component --> worst case V/2
-- Best Case --> 1 time
    {
        statisticsSheet.Cells[i + 1, 1].Value = counter;
        statisticsSheet.Cells[i + 1, 3].Value = vertex.Value.Item1;
        List<string> component = componentsLst[vertex.Key];

        //component.Sort(); // O(vlogv)
        // +d
```

```
        List<int> componentItemsList = new List<int>();
        Regex digitsRegex = new Regex("\\d+");
        string componentItems = "";
        // matchPercentage = percentageRegex.Match(column1);

        foreach (var item in component) // O(V)
        {
            Match digitsRegexMatch = digitsRegex.Match(item);
            componentItemsList.Add(Convert.ToInt32(digitsRegexMatch.Value));
        }
        componentItemsList.Sort(); // O(vlogv)
        foreach (var item in componentItemsList) // O(V)
        {
            componentItems = componentItems + item.ToString() + ",";
        }
        componentItems = componentItems.Remove(componentItems.Length - 1);
        statisticsSheet.Cells[i + 1, 2].Value = componentItems;
        statisticsSheet.Cells[i + 1, 4].Value = component.Count;
        i++;
        counter++;
    }

    //string outputFilePath = @"D:\Uni Related\Algorithms\Project\MATERIALS\[3]
Plagiarism Validation\Algorithm-Project\PlagiarismValidation\Output\File.xlsx";
    string outputFilePath = @"F:\Year 3 2nd term\Analysis and Design of
Algorithm\Project\Algorithm-Project\PlagiarismValidation\Output\File.xlsx";
    excelPackage.SaveAs(new System.IO.FileInfo(outputFilePath));

    stat_file_stopwatch.Stop();
    TimeSpan ts = stat_file_stopwatch.Elapsed;
    double totalSeconds = time_BFS + (ts.TotalHours * 60 * 60) + (ts.TotalMinutes *
60) + ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
    Console.WriteLine($"Elapsed Time for calculating and saving statistics file:
{totalSeconds} seconds");
}
```

Sixth the MST File output:

First the function call:

```
OutPut_Of_MST(refinedGroups, ref allEdges , ref edge_with_its_hyper_link ,
totalSecondsk);
```

Second the Function:

```
public static void OutPut_Of_MST(List<Dictionary<KeyValuePair<string, string>,
Tuple<int, int>>> refinedGroups, ref Dictionary<KeyValuePair<string, string>,
Tuple<int, int>> allEdges , ref Dictionary<string, string> edge_with_its_hyper_link
, double timeKruskal)
{
    Stopwatch mst_file_stopwatch = new Stopwatch();
    mst_file_stopwatch.Start();
    ExcelPackage.LicenseContext = OfficeOpenXml.LicenseContext.NonCommercial;
    ExcelPackage excelPackage = new ExcelPackage();

    ExcelWorksheet mstSheet = excelPackage.Workbook.Worksheets.Add("MST 1");

    mstSheet.Cells[1,1].Value = "File 1";
    mstSheet.Cells[1,2].Value = "File 2";
    mstSheet.Cells[1,3].Value = "Line Matches";



    int i = 1;
    foreach (Dictionary<KeyValuePair<string, string>, Tuple<int, int>> group in
refinedGroups)// number of components
    {
        //mstSheet.Cells[i + 1, 1].Value = group;
        Dictionary<KeyValuePair<string, string>, Tuple<int, int>> sorted_group =
group.OrderByDescending(pair => pair.Value.Item2).ToDictionary(pair => pair.Key,
pair => pair.Value);

        foreach (KeyValuePair<string, string> kvp in sorted_group.Keys) // E of
each refined component
        {
            KeyValuePair<string, string> kvp2 = new KeyValuePair<string, string>
(kvp.Value, kvp.Key);
            if (allEdges.ContainsKey(kvp))
            {
                string filePath1 = kvp.Key + '(' + allEdges[kvp].Item1 + "%)";
                string filePath2 = kvp.Value + '(' + allEdges[kvp].Item2 + "%)";

                mstSheet.Cells[i + 1, 1].Value = filePath1;
                if (edge_with_its_hyper_link.ContainsKey(filePath1))
                {
                    mstSheet.Cells[i + 1, 1].Hyperlink = new
Uri(edge_with_its_hyper_link[filePath1]);
                }
                mstSheet.Cells[i + 1, 2].Value = filePath2;
                if (edge_with_its_hyper_link.ContainsKey(filePath2))
                {
                    mstSheet.Cells[i + 1, 2].Hyperlink = new
```

```
Uri(edge_with_its_hyper_link[filePath2]);
                }
                mstSheet.Cells[i + 1, 3].Value = sorted_group[kvp].Item2;
            }
            else if (allEdges.ContainsKey(kvp2))
            {
                string filePath1 = kvp2.Key + '(' + allEdges[kvp2].Item1 + "%)";
                string filePath2 = kvp2.Value + '(' + allEdges[kvp2].Item2 + "%)";

                mstSheet.Cells[i + 1, 1].Value = filePath1;
                if (edge_with_its_hyper_link.ContainsKey(filePath1))
                {
                    mstSheet.Cells[i + 1, 1].Hyperlink = new
Uri(edge_with_its_hyper_link[filePath1]);
                }
                mstSheet.Cells[i + 1, 2].Value = filePath2;
                if (edge_with_its_hyper_link.ContainsKey(filePath2))
                {
                    mstSheet.Cells[i + 1, 2].Hyperlink = new
Uri(edge_with_its_hyper_link[filePath2]);
                }
                mstSheet.Cells[i + 1, 3].Value = sorted_group[kvp].Item2;
            }
            i++;
        }

    }

    string outputFilePath = @"F:\Year 3 2nd term\Analysis and Design of
Algorithm\Project\Algorithm-Project\PlagiarismValidation\Output\File2.xlsx";
    excelPackage.SaveAs(new System.IO.FileInfo(outputFilePath));

    mst_file_stopwatch.Stop();
    TimeSpan ts = mst_file_stopwatch.Elapsed;
    double totalSeconds = timeKruskal + (ts.TotalHours * 60 * 60) +
(ts.TotalMinutes * 60) + ts.TotalSeconds + (ts.TotalMilliseconds / 1000);
    Console.WriteLine($"Elapsed Time for calculating and saving MST  file:
{totalSeconds} seconds");
}
```

Finally calculating the time elapsed for the entire program:

```
Program_stopwatch.Stop();
TimeSpan tts = Program_stopwatch.Elapsed;
```

```
double totalSecondst = (tts.TotalHours * 60 * 60) + (tts.TotalMinutes * 60) +
tts.TotalSeconds + (tts.TotalMilliseconds / 1000);
Console.WriteLine($"Elapsed Time for the whole program: {totalSecondst} seconds");
```