

# Image quantization project

## Algorithms course

### Classes:

class edge:

```
1. public class edge: IComparable<edge>
```

A class which holds each edge weight, start point and end point.

```
1. public double distance = 0;  
2. public int[] points = new int[2];
```

and implements function CompareTo to enable sorting according to the distances

```
1. public int CompareTo(edge other) => other.distance.CompareTo(this.distance) * -1;
```

class ImageOperations:

A class to hold all functions to perform distinct colors extracting, MST constructing, clustering, pallet generating, DisplayImage and Quantizing the image

*Functions description and complexity:*

### OpenImage:

```
1. public static RGBPixel[,] OpenImage(string imagePath, ref int[, ] mapper, ref List<RGBPixel> colorSet)
```

parameters:

**ImagePath** => which takes the path of the image to open.

**Mapper** => As reference 3D Boolean array to map every color to a position according to red, green, blue values to create a simple set which adds the color if mapper[red, green, blue] = false which means the color is not present in the list "colorSet" and changes the value to true to avoid adding the color again to the list.

**colorSet** => As reference List which initially holds no colors and after the execution of the function it holds all distinct colors from the image to pass it to the other functions later.

**Complexity:** while reading the image it checks for every color " $O(N^2)$ " in this snippet

```
1. for (y = 0; y < Height; y++) //O(N)  
2.   for (x = 0; x < Width; x++) //O(N)
```

if it is present in the mapper or not " $O(1)$ " in this snippet

```
1. if (colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] == false) // O(1)  
2. {  
3.   colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] = true; //O(1)  
4.   colorSet.Add(Buffer[y, x]); //O(1)  
5. }
```

Which means over all " $O(N^2)$ " complexity for extracting the distinct colors.

# CalculateMST:

```
1. public static List<edge> CalculateMST(int numberOfDistinctColors, List<RGBPixel> colorSet)
```

parameters:

numberOfDistinctColors: an integer value which holds the number of distinct colors.

colorSet: the list which holds the distinct colors itself.

Attributes:

```
1. bool[] partnersOfTheTree = new bool[numberOfDistinctColors];
2. int least = 0, localLeast = 0;
3. double min, dist;
4. edge tmp;
5. List<edge> MST = new List<edge>();
6.
```

partnersOfTheTree: a Boolean array which maps every color from colorSet depending on the index of each color to check if the color is already in the tree or not

least, localLeast: an integer value which holds the index of the starting point of distance calculations which is considered the point which optimal edge found (initially set the first color in index 0)

min: a double value to hold the minimum edge weight.

dist: a double value to hold the current edge weight to compare it to min and add it to the MST(if selected).

tmp: a pointer to hold an edge from the MST to avoid re calculating in every access.

MST: a list to hold the selected edges for creating the minimum spanning tree and to be returned as a result for the function.

## Complexity:

The looping is separated into 2 phases

phase 1:

Looping from 0 to all other vertices and calculating the distances for every edge

```
1. dist = Math.Sqrt(Math.Pow(colorSet[localLeast].red - colorSet[j].red, 2) +
2. Math.Pow(colorSet[localLeast].green - colorSet[j].green, 2) +
3. Math.Pow(colorSet[localLeast].blue - colorSet[j].blue, 2));
```

Then creates the initial MST which holds all edges from color 0 to all other colors ... which makes the first feasible solution and not the optimal solution (No cycle but not the best cost) and has a size of (V-1) then chooses the least distance to point to the next edge to loop from (similar to Dijkstra)

```
1. int k = 1;
2. while (k < numberOfDistinctColors)
3. {
4.
5.     dist = Math.Sqrt(Math.Pow(colorSet[localLeast].red - colorSet[k].red, 2) +
6.         Math.Pow(colorSet[localLeast].green - colorSet[k].green, 2) +
7.         Math.Pow(colorSet[localLeast].blue - colorSet[k].blue, 2));
8.     MST.Add(new edge(dist, 0, k));
9.     if (dist < min)
10.    {
```

```

11.         min = dist;
12.         least = k;
13.     }
14.     k++;
15. }

```

Which is done in  $O(N)$  as it loops only for all other vertices.

Phase 2:

Looping from all vertices except of the first one to every other vertex to compare with the initial MST to check if better way(distance) to reach a specific vertex and replace it.

It is done in  $O(1)$  as comparing with the pervious best wat is done with indexing the MST as it is kind of sorted and the order is not changing in the entire code.

```

1.  while (k < numberOfDistinctColors-1) //O(N)
2.      {
3.          min = 99999;
4.          partnersOfTheTree[least] = true; //O(1)
5.          localLeast = least; //O(1)
6.          for (int j = 1; j < numberOfDistinctColors; j++) //O(N)
7.              {
8.                  if (partnersOfTheTree[j] == false) //O(1)
9.                      {
10.                         dist = Math.Sqrt(Math.Pow(colorSet[localLeast].red - colorSet[j].red, 2) +
11.                             Math.Pow(colorSet[localLeast].green - colorSet[j].green, 2) +
12.                             Math.Pow(colorSet[localLeast].blue - colorSet[j].blue, 2)); //O(1)
13.                         tmp = MST[j]; //O(1)
14.                         if (dist < tmp.distance) //O(1)
15.                             {
16.                                 MST[j].distance = dist; //O(1)
17.                                 MST[j].points[0] = localLeast; //O(1)
18.                             }
19.                         if (tmp.distance < min) //O(1)
20.                             {
21.                                 min = tmp.distance; //O(1)
22.                                 least = tmp.points[1]; //O(1)
23.                             }
24.                         }
25.                  }
26.          k++;
27.      }
28.

```

Which means over all “ $O(N^2)$ ” complexity for calculating the distance from every vertex to all other vertices and extracting the MST edges.

## clustering:

```

1.  public static List<List<int>>> clustering (int numberOfDistinctColors, List<edge> MST, int k)

```

parameters:

numberOfDistinctColors => an integer value which holds the number of distinct colors.

MST => the list that holds the edges which creates the minimum spanning tree.

K => the number of desired clusters for this image

```

1.  List<List<int>>> clusters = new List<List<int>>>();
2.  int[] indexer = new int[numberOfDistinctColors];

```

```
3. int cluster = 1, unClusterd = numberOfDistinctColors, numberOfClusters = 0 , p1 , p2 , I1, I2;  
4. edge tmp;
```

clusters => list which holds a list of clusters that is extracted from the function and to be returned.

Indexer => an integer array to map every color from colorSet to it is cluster Number

Cluster => defines the number of the current cluster to be added and initially set to 1

unClusterd => the number of not yet clustered vertices initially set to numberOfDistinctColors and decrements

numberOfClusters => the number of clustered vertices initially set to 0 and increments.

p1 , p2 , I1, I2 => pointers to point for every wanted index from indexer or color to avoid re calculating

tmp: a pointer to hold an edge from the MST to avoid re calculating in every access.

### Complexity:

first step is to sort the MST List to access the least distance ascendingly

```
1. MST.Sort(); // O(NlogN)
```

Clustering is done by accessing the MST from the least distance ascendingly and combining every to points for an edge according to some conditions in 2 phases

First phase: has 4 conditions

First condition is if the 2 vertices are clustered to different clusters, then we will combine the 2 clusters

```
1. if (I1 != 0 && I2 != 0 && I1 != I2) // O(1)  
2. {  
3.     int keep = I2;  
4.     clusters[keep - 1].ForEach(l => indexer[l] = I1); // O(D)  
5.     clusters[I1 - 1].AddRange(clusters[keep - 1]); // O(D)  
6.     clusters[keep - 1].Clear(); // O(1)  
7.     numberOfClusters--; // O(1)  
8. }
```

The second and third conditions are the same to check if only one from those points is clustered then we add the not clustered one to the other's cluster

```
1. else if (I1 != 0)  
2. {  
3.     indexer[p2] = I1;  
4.     clusters[I1 - 1].Add(p2);  
5.     unClusterd--;  
6.  
7. }  
8. else if (I2 != 0)  
9. {  
10.    indexer[p1] = I2;  
11.    clusters[I2 - 1].Add(p1);  
12.    unClusterd--;  
13. }
```

The fourth condition is if the 2 points are not clustered yet then we add the 2 points to a new cluster

```
1. else  
2. {  
3.     List<int> tempCluster = new List<int>();
```

```

4. tempCluster.Add(tmp.points[0]);
5. tempCluster.Add(tmp.points[1]);
6. clusters.Add(tempCluster);
7. indexer[p1] = cluster;
8. indexer[p2] = cluster;
9. cluster++;
10. numberOfClusters++;
11. unClusterd -= 2;
12. }

```

The second phase has 3 conditions and is started by the condition:

```

1. if (numberOfClusters + unClusterd == k)

```

and the 2 main conditions is checking for the remaining edges if any 1 of 2 points is not clustered yet

```

1. if (I1 == 0)//O(1)
2. {
3.     List<int> tempCluster = new List<int>();//O(1)
4.     tempCluster.Add(p1); //O(1)
5.     clusters.Add(tempCluster); //O(1)
6.     indexer[p1] = cluster; //O(1)
7.     cluster++; //O(1)
8.     numberOfClusters++; //O(1)
9.     unClusterd--; //O(1)
10. }
11. if (I2 == 0)
12. {
13.     List<int> tempCluster = new List<int>();//O(1)
14.     tempCluster.Add(p2); //O(1)
15.     clusters.Add(tempCluster); //O(1)
16.     indexer[p2] = cluster; //O(1)
17.     cluster++; //O(1)
18.     numberOfClusters++; //O(1)
19.     unClusterd--; //O(1)
20. }
21. continue;
22.

```

and the third condition is if all vertices are clustered then break the loop

```

1. if (unClusterd == 0) break;

```

Which means over all “ $O(D^3)$ ” complexity for clustering the colors

## generatePalette:

```

1. public static List<RGBPixel> generatePalette(List<List<int>> clusters, List<RGBPixel> colorSet, ref int[, ] mapper)

```

parameters:

clusters: the list of clusters that holds all vertices with all clusters

colorSet: the list of distinct colors

mapper: an integer array that maps between every color and its cluster representative color

attributes:

```

1. List<RGBPixel> palletaa = new List<RGBPixel>();

```

```

2. int clusterCounter = 0, count = clusters.Count;
3. double red = 0, green = 0, blue = 0;
4. int numberOfClusterElements
5. RGBPixel current;

```

palletaa: is the list that holds every cluster representative color

clusterCounter: an integer to hold clusters ID to assign to the mapper

current : a pointer to hold the color to add to pallet later.

### Complexity:

we loop on clusters to get the list of specific cluster

```

1. for (int i = 0; i < count; i++) //O(K)

```

then we loop on those colors in this cluster to calculate avg for every red , green , blue

```

1. foreach (var j in clusters[i]) //O(D)
2. {
3.     red = red + colorSet[j].red; // O(1)
4.     green = green + colorSet[j].green; // O(1)
5.     blue = blue + colorSet[j].blue; // O(1)
6.     mapper[colorSet[j].red, colorSet[j].green, colorSet[j].blue] = clusterCounter; // O(1)
7. }

```

Then we assign this color to pallet and assign the same id to every color in this cluster to help with mapping every color to its cluster representative color

```

1. current.red = (byte)Math.Ceiling(red / numberOfClusterElements);
2. current.green = (byte)Math.Ceiling(green / numberOfClusterElements);
3. current.blue = (byte)Math.Ceiling(blue / numberOfClusterElements);
4. palletaa.Add(current);
5. clusterCounter++;

```

as the number of loops will be bounded by the number of distinct colors then the over all “O (D)” complexity for generating pallet

### Quantize:

```

1. public static RGBPixel[,] Quantize(RGBPixel[,] ImageMatrix, List<RGBPixel> p, int[, ] mapper)

```

ImageMatrix: the image which we want to perform quantization on

P: the color pallet

Mapper: the array that maps every color to its representative color

### Complexity:

We loop on the image matrix to get every pixel and we check it on the mapper to get it's cluster id and check it in the pallet list to change the color to its representative color

```

1. for (int i = 0; i < Height; i++)// O(N)
2. for (int j = 0; j < Width; j++)// O(N)
3. {
4.     RGBPixel quantized = new RGBPixel(); // O(1)

```

```

5. quantized = p[mapper[ImageMatrix[i, j].red, ImageMatrix[i, j].green, ImageMatrix[i, j].blue]]; //O(1)
6. Filtered[i, j].red = quantized.red; // O(1)
7. Filtered[i, j].green = quantized.green; // O(1)
8. Filtered[i, j].blue = quantized.blue; // O(1)
9. }
10.

```

Then we return the quantized image

Which means over all “ $O(N^2)$ ” complexity for mapping each color to its representative color

The Code of image operations:

```

1. using System;
2. using System.Collections.Generic;
3. using System.Drawing;
4. using System.Windows.Forms;
5. using System.Drawing.Imaging;
6. using System.Linq;
7. using System.Diagnostics;
8. ///Algorithms Project
9. ///Intelligent Scissors
10. ///
11. namespace ImageQuantization
12. {
13.     public class edge : IComparable<edge>
14.     {
15.         public double distance = 0;
16.         public int[] points = new int[2];
17.         public edge(double distance, int point1, int point2)
18.         {
19.             this.distance = distance;
20.             this.points[0] = point1;
21.             this.points[1] = point2;
22.         }
23.         public int CompareTo(edge other) => other.distance.CompareTo(this.distance) * -1;
24.     }
25.     public struct color
26.     {
27.         public byte red, green, blue;
28.     }
29.     /// <summary>
30.     /// Holds the pixel color in 3 byte values: red, green and blue
31.     /// </summary>
32.     public struct RGBPixel
33.     {
34.         public byte red, green, blue;
35.     }
36.     /// <summary>
37.     /// Library of static functions that deal with images
38.     /// </summary>
39.     public class ImageOperations
40.     {
41.         public static Stopwatch overAllTime = new Stopwatch();
42.         public static double phase1Time;
43.         /// <summary>
44.         /// Open an image and load it into 2D array of colors (size: Height x Width)
45.         /// </summary>
46.         /// <param name="ImagePath">Image file path</param>
47.         /// <returns>2D array of colors</returns>
48.         public static RGBPixel[,] OpenImage(string ImagePath, ref int[, ] mapper, ref List<RGBPixel>
colorSet)
49.         {
50.             //List<edge> edges = new List<edge>();
51.             colorSet = new List<RGBPixel>();

```

```

52.         bool[, ,] colorsBoolArray = new bool[256, 256, 256];
53.         Bitmap original_bm = new Bitmap(ImagePath);
54.         int Height = original_bm.Height;
55.         int Width = original_bm.Width;
56.         RGBPixel[,] Buffer = new RGBPixel[Height, Width];
57.         Stopwatch DistinctTime = new Stopwatch();
58.         DistinctTime.Start();
59.         overAllTime.Start();
60.         unsafe
61.         {
62.
63.             BitmapData bmd = original_bm.LockBits(new Rectangle(0, 0, Width, Height),
ImageLockMode.ReadWrite, original_bm.PixelFormat);
64.             int x, y;
65.             int nWidth = 0;
66.             bool Format32 = false;
67.             bool Format24 = false;
68.             bool Format8 = false;
69.             if (original_bm.PixelFormat == PixelFormat.Format24bppRgb)
70.             {
71.                 Format24 = true;
72.                 nWidth = Width * 3;
73.             }
74.             else if (original_bm.PixelFormat == PixelFormat.Format32bppArgb ||
original_bm.PixelFormat == PixelFormat.Format32bppRgb || original_bm.PixelFormat ==
PixelFormat.Format32bppPArgb)
75.             {
76.                 Format32 = true;
77.                 nWidth = Width * 4;
78.             }
79.             else if (original_bm.PixelFormat == PixelFormat.Format8bppIndexed)
80.             {
81.                 Format8 = true;
82.                 nWidth = Width;
83.             }
84.             int nOffset = bmd.Stride - nWidth;
85.             byte* p = (byte*)bmd.Scan0;
86.             for (y = 0; y < Height; y++)
87.             {
88.                 for (x = 0; x < Width; x++)
89.                 {
90.                     if (Format8)
91.                     {
92.                         Buffer[y, x].red = Buffer[y, x].green = Buffer[y, x].blue = p[0];
93.                         p++;
94.                     }
95.                     else
96.                     {
97.                         Buffer[y, x].red = p[2];
98.                         Buffer[y, x].green = p[1];
99.                         Buffer[y, x].blue = p[0];
100.                        if (Format24) p += 3;
101.                        else if (Format32) p += 4;
102.                    }
103.                    if (colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue]
== false)
104.                    {
105.                        colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue]
= true;
106.                        colorSet.Add(Buffer[y, x]);
107.                    }
108.                }
109.                p += nOffset;
110.            }
111.            original_bm.UnlockBits(bmd);
112.        }
113.
114.
115.        //Console.WriteLine(colorSet.Count + " Distinct colors ");
116.        DistinctTime.Stop();

```



```

117.         overAllTime.Stop();
118.         MessageBox.Show(colorSet.Count.ToString() + " Number of distinct colors in time " +
DistinctTime.ElapsedMilliseconds + " ms");
119.
120.         return Buffer;
121.     }
122.     /// <summary>
123.     /// Get the height of the image
124.     /// </summary>
125.     /// <param name="ImageMatrix">2D array that contains the image</param>
126.     /// <returns>Image Height</returns>
127.     public static int GetHeight(GBPixel[,] ImageMatrix)
128.     {
129.         return ImageMatrix.GetLength(0);
130.     }
131.     /// <summary>
132.     /// Get the width of the image
133.     /// </summary>
134.     /// <param name="ImageMatrix">2D array that contains the image</param>
135.     /// <returns>Image Width</returns>
136.     public static int GetWidth(GBPixel[,] ImageMatrix)
137.     {
138.         return ImageMatrix.GetLength(1);
139.     }
140.     /// <summary>
141.     /// Display the given image on the given PictureBox object
142.     /// </summary>
143.     /// <param name="ImageMatrix">2D array that contains the image</param>
144.     /// <param name="PictureBox">PictureBox object to display the image on it</param>
145.     public static void DisplayImage(GBPixel[,] ImageMatrix, PictureBox PictureBox)
146.     {
147.         // Create Image:
148.         //=====
149.         int Height = ImageMatrix.GetLength(0);
150.         int Width = ImageMatrix.GetLength(1);
151.         Bitmap ImageBMP = new Bitmap(Width, Height, PixelFormat.Format24bppRgb);
152.         unsafe
153.         {
154.             BitmapData bmd = ImageBMP.LockBits(new Rectangle(0, 0, Width, Height),
ImageLockMode.ReadWrite, ImageMatrix.PixelFormat);
155.             int nWidth = 0;
156.             nWidth = Width * 3;
157.             int nOffset = bmd.Stride - nWidth;
158.             byte* p = (byte*)bmd.Scan0;
159.             for (int i = 0; i < Height; i++)
160.             {
161.                 for (int j = 0; j < Width; j++)
162.                 {
163.                     p[2] = ImageMatrix[i, j].red;
164.                     p[1] = ImageMatrix[i, j].green;
165.                     p[0] = ImageMatrix[i, j].blue;
166.                     p += 3;
167.                 }
168.                 p += nOffset;
169.             }
170.             ImageBMP.UnlockBits(bmd);
171.         }
172.         SaveFileDialog dialog = new SaveFileDialog();
173.         ImageBMP.Save("myfile.png", ImageFormat.Png);
174.         PictureBox.Image = ImageBMP;
175.     }
176.     /// <summary>
177.     /// Apply Gaussian smoothing filter to enhance the edge detection
178.     /// </summary>
179.     /// <param name="ImageMatrix">Colored image matrix</param>
180.     /// <param name="filterSize">Gaussian mask size</param>
181.     /// <param name="sigma">Gaussian sigma</param>
182.     /// <returns>smoothed color image</returns>
183.     public static List<edge> CalculateMST(int numberOfDistinctColors, List<GBPixel> colorSet)
184.

```

```

185.     {
186.         Stopwatch MSTTime = new Stopwatch();
187.         MSTTime.Start();
188.         overAllTime.Start();
189.         bool[] partnersOfTheTree = new bool[numberOfDistinctColors];
190.         int least = 0, localLeast = 0;
191.         double min = 99999, dist;
192.         edge tmp;
193.         List<edge> MST = new List<edge>();
194.         MST.Add(new edge(0, 0, 0));
195.         int k = 1;
196.         while (k < numberOfDistinctColors)
197.         {
198.
199.             dist = Math.Sqrt(Math.Pow(colorSet[localLeast].red - colorSet[k].red, 2) +
200.                               Math.Pow(colorSet[localLeast].green - colorSet[k].green, 2) +
201.                               Math.Pow(colorSet[localLeast].blue - colorSet[k].blue, 2));
202.             MST.Add(new edge(dist, 0, k));
203.             if (dist < min)
204.             {
205.                 min = dist;
206.                 least = k;
207.             }
208.             k++;
209.         }
210.         k = 1;
211.         while (k < numberOfDistinctColors)
212.         {
213.             min = 99999;
214.             partnersOfTheTree[least] = true;
215.             localLeast = least;
216.             for (int j = 1; j < numberOfDistinctColors; j++)
217.             {
218.                 if (partnersOfTheTree[j] == false)
219.                 {
220.                     dist = Math.Sqrt(Math.Pow(colorSet[localLeast].red - colorSet[j].red, 2) +
221.                                       Math.Pow(colorSet[localLeast].green - colorSet[j].green, 2) +
222.                                       Math.Pow(colorSet[localLeast].blue - colorSet[j].blue, 2));
223.                     tmp = MST[j];
224.                     if (dist < tmp.distance)
225.                     {
226.                         MST[j].distance = dist;
227.                         MST[j].points[0] = localLeast;
228.                     }
229.                     if (tmp.distance < min)
230.                     {
231.                         min = tmp.distance;
232.                         least = tmp.points[1];
233.                     }
234.                 }
235.             }
236.             k++;
237.         }
238.         double MST_SUM = 0;
239.         for (int i = 0; i < MST.Count; i++) MST_SUM = MST_SUM + MST[i].distance;
240.         MSTTime.Stop();
241.         overAllTime.Stop();
242.         phase1Time = overAllTime.ElapsedMilliseconds;
243.         //Console.WriteLine(MST_SUM + " MST SUM");
244.         //int MSTCOUNT = MST.Count();
245.         MessageBox.Show(MST_SUM.ToString() + " MST SUM IN TIME " + MSTTime.ElapsedMilliseconds +
246. " ms");
247.         //Console.WriteLine(MSTCOUNT - 1 + " MST COUNT");
248.         //double MEAN = MST_SUM / MST.Count() - 1;
249.         //Console.WriteLine(MEAN + " MST MEAN");
250.         //List<double> std = new List<double>();
251.         //double stdSUM = 0;
252.         //for (int i = 1; i < MSTCOUNT; i++) stdSUM += Math.Pow((MST[i].distance - MEAN), 2);
253.         //stdSUM /= MSTCOUNT - 1;

```

```

253.         //stdSUM = Math.Sqrt(stdSUM);
254.         //Console.WriteLine(stdSUM + " MST STD");
255.         //Console.WriteLine("-----");
256.         return MST;
257.     }
258.     public static List<List<int>> clustering(int numberOfDistinctColors, List<edge> MST, int k)
259.     {
260.
261.         Stopwatch clusteringTime = new Stopwatch();
262.         clusteringTime.Start();
263.         overAllTime.Restart();
264.         List<List<int>> clusters = new List<List<int>>();
265.         int[] indexer = new int[numberOfDistinctColors];
266.         int cluster = 1, unClusterd = numberOfDistinctColors, numberOfClusters = 0, p1, p2, I1,
I2;
267.         edge tmp;
268.         MST.Sort(); // O(NlogN)
269.         for (int i = 1; i < numberOfDistinctColors; i++)
270.         {
271.             tmp = MST[i];
272.             p1 = tmp.points[0]; p2 = tmp.points[1]; I1 = indexer[p1]; I2 = indexer[p2];
273.             if (numberOfClusters + unClusterd == k)
274.             {
275.                 if (unClusterd == 0) break;
276.                 if (I1 == 0)
277.                 {
278.                     List<int> tempCluster = new List<int>();
279.                     tempCluster.Add(p1);
280.                     clusters.Add(tempCluster);
281.                     indexer[p1] = cluster;
282.                     cluster++;
283.                     numberOfClusters++;
284.                     unClusterd--;
285.                 }
286.                 if (I2 == 0)
287.                 {
288.                     List<int> tempCluster = new List<int>();
289.                     tempCluster.Add(p2);
290.                     clusters.Add(tempCluster);
291.                     indexer[p2] = cluster;
292.                     cluster++;
293.                     numberOfClusters++;
294.                     unClusterd--;
295.                 }
296.                 continue;
297.             }
298.             if (I1 != 0 && I2 != 0 && I1 != I2)
299.             {
300.                 int keep = I2;
301.                 clusters[keep - 1].ForEach(l => indexer[l] = I1);
302.                 clusters[I1 - 1].AddRange(clusters[keep - 1]);
303.                 clusters[keep - 1].Clear();
304.                 numberOfClusters--;
305.             }
306.             else if (I1 != 0)
307.             {
308.                 indexer[p2] = I1;
309.                 clusters[I1 - 1].Add(p2);
310.                 unClusterd--;
311.             }
312.             else if (I2 != 0)
313.             {
314.                 indexer[p1] = I2;
315.                 clusters[I2 - 1].Add(p1);
316.                 unClusterd--;
317.             }
318.             else
319.             {
320.                 List<int> tempCluster = new List<int>();
321.

```

```

322.         tempCluster.Add(tmp.points[0]);
323.         tempCluster.Add(tmp.points[1]);
324.         clusters.Add(tempCluster);
325.         indexer[p1] = cluster;
326.         indexer[p2] = cluster;
327.         cluster++;
328.         numberOfClusters++;
329.         unClusterd -= 2;
330.     }
331. }
332. clusteringTime.Stop();
333. overAllTime.Stop();
334. MessageBox.Show("the number of clusters " + k + " in time " +
clusteringTime.ElapsedMilliseconds + " ms");
335.     return clusters;
336. }
337. public static List<RGBPixel> generatePallette(List<List<int>> clusters, List<RGBPixel>
colorSet, ref int[, ] mapper)
338. {
339.     overAllTime.Start();
340.     Stopwatch generatePalletteTime = new Stopwatch();
341.     generatePalletteTime.Start();
342.     mapper = new int[256, 256, 256];
343.     List<RGBPixel> palletaa = new List<RGBPixel>();
344.     int clusterCounter = 0, count = clusters.Count;
345.     for (int i = 0; i < count; i++)
346.     {
347.         RGBPixel current;
348.         double red = 0, green = 0, blue = 0;
349.         int numberOfClusterElements = clusters[i].Count();
350.         foreach (var j in clusters[i])
351.         {
352.             red = red + colorSet[j].red;
353.             green = green + colorSet[j].green;
354.             blue = blue + colorSet[j].blue;
355.             mapper[colorSet[j].red, colorSet[j].green, colorSet[j].blue] = clusterCounter;
356.         }
357.         if (clusters[i].Count == 0) continue;
358.         current.red = (byte)Math.Round(red / numberOfClusterElements);
359.         current.green = (byte)Math.Round(green / numberOfClusterElements);
360.         current.blue = (byte)Math.Round(blue / numberOfClusterElements);
361.         palletaa.Add(current);
362.         clusterCounter++;
363.     }
364.     generatePalletteTime.Stop();
365.     overAllTime.Stop();
366.     MessageBox.Show("generating Pallet done in time " +
generatePalletteTime.ElapsedMilliseconds + " ms");
367.     return palletaa;
368. }
369. public static RGBPixel[, ] Quantize(RGBPixel[, ] ImageMatrix, List<RGBPixel> p, int[, ]
mapper)
370. {
371.     Stopwatch QuantizeTime = new Stopwatch();
372.     QuantizeTime.Start();
373.     overAllTime.Start();
374.     int Height = GetHeight(ImageMatrix);
375.     int Width = GetWidth(ImageMatrix);
376.     RGBPixel[, ] Filtered = new RGBPixel[Height, Width];
377.     for (int i = 0; i < Height; i++)
378.         for (int j = 0; j < Width; j++)
379.         {
380.             RGBPixel quantized = new RGBPixel();
381.             quantized = p[mapper[ImageMatrix[i, j].red, ImageMatrix[i, j].green,
ImageMatrix[i, j].blue]];
382.             Filtered[i, j].red = quantized.red;
383.             Filtered[i, j].green = quantized.green;
384.             Filtered[i, j].blue = quantized.blue;
385.         }
386.     QuantizeTime.Stop();

```

```

387.         overAllTime.Stop();
388.         MessageBox.Show("image mapping colors done in time " + QuantizeTime.ElapsedMilliseconds
+ " ms");
389.         MessageBox.Show("over all time is " + (overAllTime.ElapsedMilliseconds + phase1Time) +
" ms");
390.         MessageBox.Show("over all time is " + ((overAllTime.ElapsedMilliseconds + phase1Time) /
1000) + " S");
391.         MessageBox.Show("over all time is " + ((overAllTime.ElapsedMilliseconds + phase1Time) /
1000 / 60) + " M");
392.         return Filtered;
393.     }
394. }
395. }

```

The code of MainForm.cs:

```

1.  using System;
2.  using System.Collections.Generic;
3.  using System.ComponentModel;
4.  using System.Data;
5.  using System.Drawing;
6.  using System.Text;
7.  using System.Windows.Forms;
8.
9.  namespace ImageQuantization
10. {
11.     public partial class MainForm : Form
12.     {
13.         public MainForm()
14.         {
15.             InitializeComponent();
16.         }
17.
18.         RGBPixel[,] ImageMatrixOriginal, ImageMatrixQuantized;
19.         List<RGBPixel> p = new List<RGBPixel>(), colorSet = new List<RGBPixel>();
20.         List<List<int>> clustersList = new List<List<int>>();
21.         List<edge> MST;
22.         int[,] mapper = new int[256, 256];
23.         private void btnOpen_Click(object sender, EventArgs e)
24.         {
25.             OpenFileDialog openFileDialog1 = new OpenFileDialog();
26.             if (openFileDialog1.ShowDialog() == DialogResult.OK)
27.             {
28.                 //Open the browsed image and display it
29.                 string OpenedFilePath = openFileDialog1.FileName;
30.                 ImageMatrixOriginal = ImageOperations.OpenImage(OpenedFilePath, ref mapper, ref colorSet);
31.                 ImageOperations.DisplayImage(ImageMatrixOriginal, pictureBox1);
32.                 MST = ImageOperations.CalculateMST(colorSet.Count, colorSet);
33.             }
34.             txtWidth.Text = ImageOperations.GetWidth(ImageMatrixOriginal).ToString();
35.             txtHeight.Text = ImageOperations.GetHeight(ImageMatrixOriginal).ToString();
36.         }
37.
38.         private void btnGaussSmooth_Click(object sender, EventArgs e)
39.         {
40.             int clusters = (int)nudMaskSize.Value;
41.             clustersList = ImageOperations.clustering(colorSet.Count, MST, clusters);
42.             p = ImageOperations.generatePallete(clustersList, colorSet, ref mapper);
43.             ImageMatrixQuantized = ImageOperations.Quantize(ImageMatrixOriginal, p, mapper);
44.             ImageOperations.DisplayImage(ImageMatrixQuantized, pictureBox2);
45.         }
46.
47.         private void pictureBox1_Click(object sender, EventArgs e)
48.         {
49.
50.         }
51.

```

```
52.     private void pictureBox2_Click(object sender, EventArgs e)
53.     {
54.
55.     }
56.
57.     private void label1_Click(object sender, EventArgs e)
58.     {
59.
60.     }
61.
62.     private void label2_Click(object sender, EventArgs e)
63.     {
64.
65.     }
66.
67.     private void label3_Click(object sender, EventArgs e)
68.     {
69.
70.     }
71.
72.     private void label4_Click(object sender, EventArgs e)
73.     {
74.
75.     }
76.
77.     private void txtHeight_TextChanged(object sender, EventArgs e)
78.     {
79.
80.     }
81.
82.     private void nudMaskSize_ValueChanged(object sender, EventArgs e)
83.     {
84.
85.     }
86.
87.     private void txtWidth_TextChanged(object sender, EventArgs e)
88.     {
89.
90.     }
91.
92.     private void label5_Click(object sender, EventArgs e)
93.     {
94.
95.     }
96.
97.     private void label6_Click(object sender, EventArgs e)
98.     {
99.
100.    }
101.
102.    private void txtGaussSigma_TextChanged(object sender, EventArgs e)
103.    {
104.
105.    }
106.
107.    private void panel1_Paint(object sender, PaintEventArgs e)
108.    {
109.
110.    }
111.
112.    private void panel2_Paint(object sender, PaintEventArgs e)
113.    {
114.
115.    }
116. }
117. }
```