

# 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.   graph.Add(new List<int>());  
6. }
```

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 show 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 the separated to 2 phases

phase 1:

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

```
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 (like Dijkstra)

```
1. int k = 1;
2. // loops from 0 to all other vertices
3. while (k < numberOfDistinctColors)//O(N)
4. {
5.     // calculating the distance
6.     byte red1 = colorSet[localLeast].red;//O(1)
7.     byte green1 = colorSet[localLeast].green;//O(1)
8.     byte blue1 = colorSet[localLeast].blue;//O(1)
```

```

9.         byte red2 = colorSet[k].red;//O(1)
10.        byte green2 = colorSet[k].green;//O(1)
11.        byte blue2 = colorSet[k].blue;//O(1)
12.        double red = (red1 - red2) * (red1 - red2);//O(1)
13.        double green = (green1 - green2) * (green1 - green2);//O(1)
14.        double blue = (blue1 - blue2) * (blue1 - blue2);//O(1)
15.        dist = Math.Sqrt(red + green + blue);//O(1)
16.        // adding the edge to the initial MST
17.        MST.Add(new edge(dist, 0, k));//O(1)
18.        graph[localLeast].Add(k);//O(1)
19.        graph[k].Add(localLeast);//O(1)
20.        // checks if it is the minimum edge to start from its end
21.        if (dist < min)//O(1)
22.        {
23.            // change the minimum
24.            min = dist;//O(1)
25.            // save the index of the next start over
26.            least = k;//O(1)
27.        }
28.        k++;
29.    }

```

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.    k = 1;
2.        // looping from every vertex except of 0 to the others
3.        while (k < numberOfDistinctColors)//O(N)
4.        {
5.            //reset the min for every vertex
6.            min = 99999;
7.            //set the startPoint as optiml way found
8.            partnersOfTheTree[least] = true;
9.            // reserve the start point index to prevent fomr modifications
10.           localLeast = least;
11.           // check the distance to all others from the current start point
12.           for (int j = 1; j < numberOfDistinctColors; j++)//O(D)
13.           {
14.               // exclude the optimal points if found
15.               if (partnersOfTheTree[j] == false)//O(D)
16.               {
17.                   // calculating the distance
18.                   byte red1 = colorSet[localLeast].red;//O(1)
19.                   byte green1 = colorSet[localLeast].green;//O(1)
20.                   byte blue1 = colorSet[localLeast].blue;//O(1)
21.                   byte red2 = colorSet[j].red;//O(1)
22.                   byte green2 = colorSet[j].green;//O(1)
23.                   byte blue2 = colorSet[j].blue;//O(1)
24.                   double red = (red1 - red2) * (red1 - red2);//O(1)
25.                   double green = (green1 - green2) * (green1 - green2);//O(1)
26.                   double blue = (blue1 - blue2) * (blue1 - blue2);//O(1)
27.                   dist = Math.Sqrt(red + green + blue);//O(1)
28.                   // hold the edge to reduce access time
29.                   tmp = MST[j];//O(1)
30.                   //check if it is a better way to this vertex
31.                   if (dist < tmp.distance)//O(1)
32.                   {
33.                       graph[j].Remove(MST[j].points[0]);//O(N)
34.                       graph[MST[j].points[0]].Remove(j);//O(N)
35.                       //replace the distance
36.                       MST[j].distance = dist;//O(1)

```

```

37.         //replace the starting index
38.         MST[j].points[0] = localLeast;//O(1)
39.         graph[j].Add(MST[j].points[0]);//O(1)
40.         graph[MST[j].points[0]].Add(j);//O(1)    }
41.         //check if it is the best edge in the tree to set the next start point
42.         if (tmp.distance < min)//O(1)
43.         {
44.             //change the min
45.             min = tmp.distance;//O(1)
46.             // reserve the index
47.             least = tmp.points[1];//O(1)
48.         }
49.     }
50. }
51. k++;
52. }

```

Which means over all “ $O(D^2)$ ” 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(1 => indexer[I] = 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 “ $O(D^2)$ ” complexity for clustering the colors

## Clustering with bonus:

First we calculate the mean

```

1. int MSTCOUNT = MST.Count() - 1; //O(1)
2. double MEAN = MST_SUM / MSTCOUNT; //O(1)

```

then standard deviation

```

1. double stdSUM = 0; //O(1)
2. int STDcount = MSTCOUNT - 1; //O(1)
3. // calculate the standard deviation
4. for (int i = 0; i < MSTCOUNT; i++) //O(E) bouns loop
5.     stdSUM += (MST[i].distance - MEAN) * (MST[i].distance - MEAN);
6. stdSUM /= STDcount;
7. stdSUM = Math.Sqrt(stdSUM);

```

our stopping condition is when the reduction is exceedingly small value

```

1. while (Math.Abs(stdSUM - stdSUM_loop) >= 0.0001 || h == 0) //O(E) bouns loop // to check if stopping

```

which means we need to check all MST edges to check which makes the most reduction as a naive solution or in more efficiently we can say that the best value to exclude is the most far away from the mean

```

1. if (Math.Abs(MST[r].distance - MEAN) >= MST[h].distance - MEAN) // decide which makes more reduction

```

in a sorted array this means the first or the last value

then we break the chosen edge the same steps if it is the first or the last

```

1. // breaking the edge
2. graph[MST[r].points[0]].Remove(MST[r].points[1]); //O(N)
3. graph[MST[r].points[1]].Remove(MST[r].points[0]); //O(N)

```

then we recalculate the mean by excluding the chosen edge weight

```

1. //calculate new mean

```

```

2.  MEAN = MEAN * (STDcount + 1);
3.  MEAN = MEAN - MST[r].distance;
4.  MEAN = MEAN / STDcount;

```

Then skipping the edge and loop again to calculate the standard deviation

```

1.  r--; // skipping this edge
2.  //calculate new standard deviation
3.  for (int i = h; i < r; i++) // O(E) bounds loop
4.      stdSUM_loop += (MST[i].distance - MEAN) * (MST[i].distance - MEAN);
5.  stdSUM_loop /= STDcount;
6.  stdSUM_loop = Math.Sqrt(stdSUM_loop);

```

we added a condition to handle mathematical exception to stop the loop

```

1.  if (STDcount == 0) break; // handle error

```

after ending the last loop then we have broken our MST to K subtrees

the next step is to BFS on it to create the clusters

```

1.  for (int i = 0; i < graph.Count; i++) // O(D)
2.      if (partnersOfTheTree[i] == false)
3.      {
4.          clusters.Add(new List<int>()); // O(1)
5.          q.Enqueue(i); // O(1)

```

we add every point to get its direct neighbors to create the cluster

```

1.  while (q.Count != 0) // O(D)
2.      {
3.
4.          // temp to hold peek to reduce time then Dequeue
5.          var tmpQ = q.Dequeue();
6.          // adding all direct connactions
7.          for (int j = 0; j < graph[tmpQ].Count; j++) // O(D)
8.              if (partnersOfTheTree[graph[tmpQ][j]] == false) // O(1) // to avoid retake
9.                  q.Enqueue(graph[tmpQ][j]); // O(1)
10.         partnersOfTheTree[tmpQ] = true; // O(1) // mark as taken
11.         clusters[cluster_counter].Add(tmpQ); // O(1) // add to cluster
12.
13.     }

```

The overall complexity is  $O(K * D * E)$

## 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 clusters

```

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 overall “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 its 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, 265, 265];

```

```

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++)//O(N)
87.         {
88.             for (x = 0; x < Width; x++)//O(N)
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.                // check if the color is present in the list
104.                if (colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] == false)//O(1)
105.                {
106.                    // sets the color is present in the list
107.                    colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] = true;//O(1)
108.                    // adds the color to list
109.                    colorSet.Add(Buffer[y, x]);//O(1)
110.                }
111.            }
112.            p += nOffset;
113.        }
114.        original_bm.UnlockBits(bmd);
115.    }
116.
117.
118.    //Console.WriteLine(colorSet.Count + " Distinct colors ");
119.    DistinctTime.Stop();
120.    overAllTime.Stop();
121.    MessageBox.Show(colorSet.Count.ToString() + " Number of distinct colors in time " + DistinctTime.ElapsedMilliseconds + " ms");

```

```

122.
123.     return Buffer;
124. }
125. /// <summary>
126. /// Get the height of the image
127. /// </summary>
128. /// <param name="ImageMatrix">2D array that contains the image</param>
129. /// <returns>Image Height</returns>
130. public static int GetHeight(GBPixel[,] ImageMatrix)
131. {
132.     return ImageMatrix.GetLength(0);
133. }
134. /// <summary>
135. /// Get the width of the image
136. /// </summary>
137. /// <param name="ImageMatrix">2D array that contains the image</param>
138. /// <returns>Image Width</returns>
139. public static int GetWidth(GBPixel[,] ImageMatrix)
140. {
141.     return ImageMatrix.GetLength(1);
142. }
143. /// <summary>
144. /// Display the given image on the given PictureBox object
145. /// </summary>
146. /// <param name="ImageMatrix">2D array that contains the image</param>
147. /// <param name="PictureBox">PictureBox object to display the image on it</param>
148. public static void DisplayImage(GBPixel[,] ImageMatrix, PictureBox PictureBox)
149. {
150.     // Create Image:
151.     //=====
152.     int Height = ImageMatrix.GetLength(0);
153.     int Width = ImageMatrix.GetLength(1);
154.     Bitmap ImageBMP = new Bitmap(Width, Height, PixelFormat.Format24bppRgb);
155.     unsafe
156.     {
157.         BitmapData bmd = ImageBMP.LockBits(new Rectangle(0, 0, Width, Height), ImageLockMode.ReadWrite,
ImageBMP.PixelFormat);
158.         int nWidth = 0;
159.         nWidth = Width * 3;
160.         int nOffset = bmd.Stride - nWidth;
161.         byte* p = (byte*)bmd.Scan0;
162.         for (int i = 0; i < Height; i++)
163.         {
164.             for (int j = 0; j < Width; j++)
165.             {
166.                 p[2] = ImageMatrix[i, j].red;
167.                 p[1] = ImageMatrix[i, j].green;
168.                 p[0] = ImageMatrix[i, j].blue;
169.                 p += 3;
170.             }
171.
172.             p += nOffset;
173.         }
174.         ImageBMP.UnlockBits(bmd);
175.     }
176.     SaveFileDialog dialog = new SaveFileDialog();
177.     ImageBMP.Save("myfile.png", ImageFormat.Png);
178.     PictureBox.Image = ImageBMP;
179. }
180. /// <summary>
181. /// Apply Gaussian smoothing filter to enhance the edge detection
182. /// </summary>
183. /// <param name="ImageMatrix">Colored image matrix</param>
184. /// <param name="filterSize">Gaussian mask size</param>
185. /// <param name="sigma">Gaussian sigma</param>
186. /// <returns>smoothed color image</returns>
187. public static List<edge> CalculateMST(int numberOfDistinctColors, List<GBPixel> colorSet)
188. {
189.     Stopwatch MSTTime = new Stopwatch();
190.     MSTTime.Start();
191.     overAllTime.Start();

```

```

192. // bool array to check optimized vertcies
193. bool[] partnersOfTheTree = new bool[numberOfDistinctColors];
194. int least = 0, localLeast = 0;
195. double min = 99999, dist;
196. // holds the edge to reduce access time
197. edge tmp;
198. List<edge> MST = new List<edge>();
199. // adds a fake edge to start the MST indexing from 1
200. MST.Add(new edge(0, 0, 0)); //O(1)
201. int k = 1;
202. // loops from 0 to all other vertcies
203. while (k < numberOfDistinctColors) //O(N)
204. {
205.     // calculating the distance
206.     byte red1 = colorSet[localLeast].red; //O(1)
207.     byte green1 = colorSet[localLeast].green; //O(1)
208.     byte blue1 = colorSet[localLeast].blue; //O(1)
209.     byte red2 = colorSet[k].red; //O(1)
210.     byte green2 = colorSet[k].green; //O(1)
211.     byte blue2 = colorSet[k].blue; //O(1)
212.     double red = (red1 - red2) * (red1 - red2); //O(1)
213.     double green = (green1 - green2) * (green1 - green2); //O(1)
214.     double blue = (blue1 - blue2) * (blue1 - blue2); //O(1)
215.     dist = Math.Sqrt(red + green + blue); //O(1)
216.     // adding the edge to the initial MST
217.     MST.Add(new edge(dist, 0, k)); //O(1)
218.     // checks if it is the minimum edge to start from its end
219.     if (dist < min) //O(1)
220.     {
221.         // change the minimum
222.         min = dist; //O(1)
223.         // save the index of the next start over
224.         least = k; //O(1)
225.     }
226.     k++;
227. }
228. k = 1;
229. // looping from every vertex exept of 0 to the others
230. while (k < numberOfDistinctColors) //O(N)
231. {
232.     //reset the min for every vertex
233.     min = 99999;
234.     //set the startPoint as optiml way found
235.     partnersOfTheTree[least] = true;
236.     // reserve the start point index to prevent fomr modifications
237.     localLeast = least;
238.     // check the distance to all others from the current start point
239.     for (int j = 1; j < numberOfDistinctColors; j++) //O(D)
240.     {
241.         // exclude the optimal points if found
242.         if (partnersOfTheTree[j] == false) //O(D)
243.         {
244.             // calculating the distance
245.             byte red1 = colorSet[localLeast].red; //O(1)
246.             byte green1 = colorSet[localLeast].green; //O(1)
247.             byte blue1 = colorSet[localLeast].blue; //O(1)
248.             byte red2 = colorSet[j].red; //O(1)
249.             byte green2 = colorSet[j].green; //O(1)
250.             byte blue2 = colorSet[j].blue; //O(1)
251.             double red = (red1 - red2) * (red1 - red2); //O(1)
252.             double green = (green1 - green2) * (green1 - green2); //O(1)
253.             double blue = (blue1 - blue2) * (blue1 - blue2); //O(1)
254.             dist = Math.Sqrt(red + green + blue); //O(1)
255.             // hold the edge to reduce access time
256.             tmp = MST[j]; //O(1)
257.             //check if it is a better way to this vertex
258.             if (dist < tmp.distance) //O(1)
259.             {
260.                 //replace the distance
261.                 MST[j].distance = dist; //O(1)
262.                 //replace the starting index

```

```

263.         MST[j].points[0] = localLeast;//O(1)
264.     }
265.     //check if it is the best edge in the tree to set the next start point
266.     if (tmp.distance < min)//O(1)
267.     {
268.         //change the min
269.         min = tmp.distance;//O(1)
270.         // reserve the index
271.         least = tmp.points[1];//O(1)
272.     }
273. }
274. }
275. k++;
276. }
277. double MST_SUM = 0;
278. //calculating the MST SUM
279. for (int i = 0; i < MST.Count; i++) MST_SUM = MST_SUM + MST[i].distance;//O(N)
280. MSTTime.Stop();
281. overAllTime.Stop();
282. phase1Time = overAllTime.ElapsedMilliseconds;
283. //Console.WriteLine(MST_SUM + " MST SUM");
284. //int MSTCOUNT = MST.Count()-1;
285. MessageBox.Show(MST_SUM.ToString() + " MST SUM IN TIME " + MSTTime.ElapsedMilliseconds + " ms");
286. //Console.WriteLine(MSTCOUNT - 1 + " MST COUNT");
287. //double MEAN = MST_SUM / (MST.Count() - 1);
288. //Console.WriteLine(MEAN + " MST MEAN");
289. ///List<double> std = new List<double>();
290. //double stdSUM = 0;
291. //int current = 0;
292. //double currentMean = MEAN;
293. //int current_MSTCOUNT = MSTCOUNT;
294. //for (int i = 1; i < MSTCOUNT; i++) stdSUM += Math.Pow((MST[i].distance - MEAN), 2);
295. //stdSUM /= (MSTCOUNT - 1);
296. //stdSUM = Math.Sqrt(stdSUM);
297. //Console.WriteLine(stdSUM + " MST STD");
298. //Console.WriteLine("-----");
299. return MST;
300. }
301. public static List<List<int>> clustering(int numberOfDistinctColors, List<edge> MST, int k)
302. {
303.
304.     Stopwatch clusteringTime = new Stopwatch();
305.     clusteringTime.Start();
306.     overAllTime.Restart();
307.     List<List<int>> clusters = new List<List<int>>();
308.     int[] indexer = new int[numberOfDistinctColors];
309.     int cluster = 1, unClusterd = numberOfDistinctColors, numberOfClusters = 0, p1, p2, I1, I2;//O(1)
310.     edge tmp;
311.     MST.Sort(); // O(Nlog(N))
312.     // loop on every edge to get it's 2 points
313.     for (int i = 1; i < numberOfDistinctColors; i++)//O(E)
314.     {
315.         // holds the edge to reduce access time
316.         tmp = MST[i];
317.         // holds the edge points and their indexer value to reduce access time
318.         p1 = tmp.points[0]; p2 = tmp.points[1]; I1 = indexer[p1]; I2 = indexer[p2];
319.         // if the clusters are found
320.         if (numberOfClusters + unClusterd == k)//O(1)
321.         {
322.             //if all points are clusterd then break ;
323.             if (unClusterd == 0) break;//O(1)
324.             // if the point references 0 as not clusterd
325.             if (I1 == 0)
326.             {
327.                 // create a new cluster to hold this point
328.                 List<int> tempCluster = new List<int>();//O(1)
329.                 tempCluster.Add(p1);//O(1)
330.                 clusters.Add(tempCluster);//O(1)
331.                 indexer[p1] = cluster;//O(1)
332.                 cluster++;//O(1)
333.                 numberOfClusters++;//O(1)

```

```

334.         unClusterd--; //O(1)
335.     }
336.     // if the point references 0 as not clustered
337.     if (I2 == 0) //O(1)
338.     {
339.         // create a new cluster to hold this point
340.         List<int> tempCluster = new List<int>(); //O(1)
341.         tempCluster.Add(p2); //O(1)
342.         clusters.Add(tempCluster); //O(1)
343.         indexer[p2] = cluster; //O(1)
344.         cluster++; //O(1)
345.         numberOfClusters++; //O(1)
346.         unClusterd--; //O(1)
347.     }
348.     continue;
349. }
350. // if both are clusterd and not the same cluster
351. if (I1 != 0 && I2 != 0 && I1 != I2) //O(1)
352. {
353.     int keep = I2; //O(1)
354.     //change every value of the indexer of the 2nd point to reference the cluster of the 1st point
355.     clusters[keep - 1].ForEach(l => indexer[l] = I1); //O(N)
356.     //union the 2 clusters
357.     clusters[I1 - 1].AddRange(clusters[keep - 1]); //O(Log(N))
358.     // clears the deprecated cluster
359.     clusters[keep - 1].Clear(); //O(1)
360.     // reduce the number of clusters
361.     numberOfClusters--; //O(1)
362. }
363. //if only one point is clusterd
364. else if (I1 != 0)
365. {
366.     //set the reference to the other point to reference the same cluster as the first
367.     indexer[p2] = I1; //O(1)
368.     //add the other to its cluster
369.     clusters[I1 - 1].Add(p2); //O(1)
370.     // reduce the number of unClusterd
371.     unClusterd--; //O(1)
372. }
373. //if only one point is clusterd
374. else if (I2 != 0) //O(1)
375. {
376.     //set the reference to the other point to reference the same cluster as the first
377.     indexer[p1] = I2; //O(1)
378.     //add the other to its cluster
379.     clusters[I2 - 1].Add(p1); //O(1)
380.     // reduce the number of unClusterd
381.     unClusterd--; //O(1)
382. }
383. // if both are not clusterd
384. else
385. {
386.     // create new cluster
387.     List<int> tempCluster = new List<int>();
388.     // add both points
389.     tempCluster.Add(tmp.points[0]); //O(1)
390.     tempCluster.Add(tmp.points[1]); //O(1)
391.     clusters.Add(tempCluster); //O(1)
392.     // set the reference to both points as the cluster number
393.     indexer[p1] = cluster; //O(1)
394.     indexer[p2] = cluster; //O(1)
395.     cluster++; //O(1)
396.     numberOfClusters++; //O(1)
397.     unClusterd -= 2; //O(1)
398. }
399. }
400. }
401. clusteringTime.Stop();
402. overAllTime.Stop();
403. MessageBox.Show("the number of clusters " + k + " in time " + clusteringTime.ElapsedMilliseconds + " ms");
404. return clusters;

```

```

405.     }
406. public static List<RGBPixel> generatePallete(List<List<int>>> clusters, List<RGBPixel> colorSet, ref int[, ] mapper)
407. {
408.     overAllTime.Start();
409.     Stopwatch generatePalleteTime = new Stopwatch();
410.     generatePalleteTime.Start();
411.     //re intialize the mapper to overwrite any old data
412.     mapper = new int[256, 256, 256]; //O(1)
413.     List<RGBPixel> palletaa = new List<RGBPixel>();
414.     int clusterCounter = 0, count = clusters.Count;
415.     for (int i = 0; i < count; i++) //O(N)
416.     {
417.         // holds the color
418.         RGBPixel current; //O(1)
419.         double red = 0, green = 0, blue = 0; //O(1)
420.         int numberOfClusterElements = clusters[i].Count(); //O(1)
421.         // clculates the avarage of red , green and blue
422.         foreach (var j in clusters[i])
423.         {
424.             red = red + colorSet[j].red;
425.             green = green + colorSet[j].green;
426.             blue = blue + colorSet[j].blue;
427.             //sets the reference to the color to its cluster number
428.             mapper[colorSet[j].red, colorSet[j].green, colorSet[j].blue] = clusterCounter; //O(1)
429.         }
430.         // overcoming the error from clustering step by skipping every empty cluster
431.         if (clusters[i].Count == 0) continue;
432.         // round to handle double numbers error
433.         current.red = (byte) Math.Round(red / numberOfClusterElements); //O(1)
434.         current.green = (byte) Math.Round(green / numberOfClusterElements); //O(1)
435.         current.blue = (byte) Math.Round(blue / numberOfClusterElements); //O(1)
436.         // add the color to the pallet list in the same reference to the mapper
437.         palletaa.Add(current); //O(1)
438.         // change ther reference
439.         clusterCounter++; //O(1)
440.     }
441.     generatePalleteTime.Stop();
442.     overAllTime.Stop();
443.     MessageBox.Show("generating Pallet done in time " + generatePalleteTime.ElapsedMilliseconds + " ms");
444.     return palletaa;
445. }
446. public static RGBPixel[,] Quantize(RGBPixel[,] ImageMatrix, List<RGBPixel> p, int[, ] mapper)
447. {
448.     Stopwatch QuantizeTime = new Stopwatch();
449.     QuantizeTime.Start();
450.     overAllTime.Start();
451.     int Height = GetHeight(ImageMatrix);
452.     int Width = GetWidth(ImageMatrix);
453.     RGBPixel[,] Filtered = new RGBPixel[Height, Width];
454.     //loop to get every pixel in the picture to replace
455.     for (int i = 0; i < Height; i++) //O(N)
456.         for (int j = 0; j < Width; j++) //O(N)
457.         {
458.             RGBPixel quantized = new RGBPixel();
459.             //gets the pallet reference from the mapper then saves the color
460.             quantized = p[mapper[ImageMatrix[i, j].red, ImageMatrix[i, j].green, ImageMatrix[i, j].blue]]; //O(1)
461.             //replace the old color by its representative
462.             Filtered[i, j].red = quantized.red; //O(1)
463.             Filtered[i, j].green = quantized.green; //O(1)
464.             Filtered[i, j].blue = quantized.blue; //O(1)
465.         }
466.     QuantizeTime.Stop();
467.     overAllTime.Stop();
468.     MessageBox.Show("image mapping colors done in time " + QuantizeTime.ElapsedMilliseconds + " ms");
469.     MessageBox.Show("over all time is " + (overAllTime.ElapsedMilliseconds + phase1Time) + " ms");
470.     MessageBox.Show("over all time is " + ((overAllTime.ElapsedMilliseconds + phase1Time) / 1000) + " S");
471.     MessageBox.Show("over all time is " + ((overAllTime.ElapsedMilliseconds + phase1Time) / 1000 / 60) + " M");
472.     return Filtered;
473. }
474. }
475. }

```

## The code of images quantization with bonus

```
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.         public static List<List<int>> graph = new List<List<int>>();
44.         /// <summary>
45.         /// Open an image and load it into 2D array of colors (size: Height x Width)
46.         /// </summary>
47.         /// <param name="ImagePath">Image file path</param>
48.         /// <returns>2D array of colors</returns>
49.         public static RGBPixel[,] OpenImage(string ImagePath, ref int[, ] mapper, ref List<RGBPixel> colorSet)
50.         {
51.             //List<edge> edges = new List<edge>();
52.             colorSet = new List<RGBPixel>();
53.             bool[, ] colorsBoolArray = new bool[256, 265, 265];
54.             Bitmap original_bm = new Bitmap(ImagePath);
55.             int Height = original_bm.Height;
56.             int Width = original_bm.Width;
57.             RGBPixel[,] Buffer = new RGBPixel[Height, Width];
58.             graph = new List<List<int>>();
59.             Stopwatch DistinctTime = new Stopwatch();
60.             DistinctTime.Start();
61.             overAllTime.Start();
62.             unsafe
63.             {
64.
65.                 BitmapData bmd = original_bm.LockBits(new Rectangle(0, 0, Width, Height), ImageLockMode.ReadWrite,
original_bm.PixelFormat);
66.                 int x, y;
67.                 int nWidth = 0;
```



```

68.         bool Format32 = false;
69.         bool Format24 = false;
70.         bool Format8 = false;
71.         if (original_bm.PixelFormat == PixelFormat.Format24bppRgb)
72.         {
73.             Format24 = true;
74.             nWidth = Width * 3;
75.         }
76.         else if (original_bm.PixelFormat == PixelFormat.Format32bppArgb || original_bm.PixelFormat == PixelFormat.Format32bppRgb
|| original_bm.PixelFormat == PixelFormat.Format32bppPArgb)
77.         {
78.             Format32 = true;
79.             nWidth = Width * 4;
80.         }
81.         else if (original_bm.PixelFormat == PixelFormat.Format8bppIndexed)
82.         {
83.             Format8 = true;
84.             nWidth = Width;
85.         }
86.         int nOffset = bmd.Stride - nWidth;
87.         byte* p = (byte*)bmd.Scan0;
88.         for (y = 0; y < Height; y++)//O(N)
89.         {
90.             for (x = 0; x < Width; x++)//O(N)
91.             {
92.                 if (Format8)
93.                 {
94.                     Buffer[y, x].red = Buffer[y, x].green = Buffer[y, x].blue = p[0];
95.                     p++;
96.                 }
97.                 else
98.                 {
99.                     Buffer[y, x].red = p[2];
100.                    Buffer[y, x].green = p[1];
101.                    Buffer[y, x].blue = p[0];
102.                    if (Format24) p += 3;
103.                    else if (Format32) p += 4;
104.                }
105.                // check if the color is present in the list
106.                if (colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] == false)//O(1)
107.                {
108.                    // sets the color is present in the list
109.                    colorsBoolArray[Buffer[y, x].red, Buffer[y, x].green, Buffer[y, x].blue] = true;//O(1)
110.                    // adds the color to list
111.                    colorSet.Add(Buffer[y, x]);//O(1)
112.                    graph.Add(new List<int>());
113.                }
114.            }
115.            p += nOffset;
116.        }
117.        original_bm.UnlockBits(bmd);
118.    }
119.
120.
121.    //Console.WriteLine(colorSet.Count + " Distinct colors ");
122.    DistinctTime.Stop();
123.    overAllTime.Stop();
124.    MessageBox.Show(colorSet.Count.ToString() + " Number of distinct colors in time " + DistinctTime.ElapsedMilliseconds + " ms");
125.
126.    return Buffer;
127. }
128. /// <summary>
129. /// Get the height of the image
130. /// </summary>
131. /// <param name="ImageMatrix">2D array that contains the image</param>
132. /// <returns>Image Height</returns>
133. public static int GetHeight(RGBPixel[,] ImageMatrix)
134. {
135.     return ImageMatrix.GetLength(0);
136. }
137. /// <summary>

```

```

138.    /// Get the width of the image
139.    /// </summary>
140.    /// <param name="ImageMatrix">2D array that contains the image</param>
141.    /// <returns>Image Width</returns>
142.    public static int GetWidth(GBPPixel[,] ImageMatrix)
143.    {
144.        return ImageMatrix.GetLength(1);
145.    }
146.    /// <summary>
147.    /// Display the given image on the given PictureBox object
148.    /// </summary>
149.    /// <param name="ImageMatrix">2D array that contains the image</param>
150.    /// <param name="PictureBox">PictureBox object to display the image on it</param>
151.    public static void DisplayImage(GBPPixel[,] ImageMatrix, PictureBox PictureBox)
152.    {
153.        // Create Image:
154.        //=====
155.        int Height = ImageMatrix.GetLength(0);
156.        int Width = ImageMatrix.GetLength(1);
157.        Bitmap ImageBMP = new Bitmap(Width, Height, PixelFormat.Format24bppRgb);
158.        unsafe
159.        {
160.            BitmapData bmd = ImageBMP.LockBits(new Rectangle(0, 0, Width, Height), ImageLockMode.ReadWrite,
ImageBMP.PixelFormat);
161.            int nWidth = 0;
162.            nWidth = Width * 3;
163.            int nOffset = bmd.Stride - nWidth;
164.            byte* p = (byte*)bmd.Scan0;
165.            for (int i = 0; i < Height; i++)
166.            {
167.                for (int j = 0; j < Width; j++)
168.                {
169.                    p[2] = ImageMatrix[i, j].red;
170.                    p[1] = ImageMatrix[i, j].green;
171.                    p[0] = ImageMatrix[i, j].blue;
172.                    p += 3;
173.                }
174.                p += nOffset;
175.            }
176.            ImageBMP.UnlockBits(bmd);
177.        }
178.        SaveFileDialog dialog = new SaveFileDialog();
179.        ImageBMP.Save("myfile.png", ImageFormat.Png);
180.        PictureBox.Image = ImageBMP;
181.    }
182.    /// <summary>
183.    /// Apply Gaussian smoothing filter to enhance the edge detection
184.    /// </summary>
185.    /// <param name="ImageMatrix">Colored image matrix</param>
186.    /// <param name="filterSize">Gaussian mask size</param>
187.    /// <param name="sigma">Gaussian sigma</param>
188.    /// <returns>smoothed color image</returns>
189.    public static List<edge> CalculateMST(int numberOfDistinctColors, List<GBPPixel> colorSet)
190.    {
191.        Stopwatch MSTTime = new Stopwatch();
192.        MSTTime.Start();
193.        overAllTime.Start();
194.        // bool array to check optimized vertcies
195.        bool[] partnersOfTheTree = new bool[numberOfDistinctColors];
196.        int least = 0, localLeast = 0;
197.        double min = 99999, dist;
198.        // holds the edge to reduce access time
199.        edge tmp;
200.        List<edge> MST = new List<edge>();
201.        // adds a fake edge to start the MST indexing from 1
202.        MST.Add(new edge(0, 0, 0)); //O(1)
203.        int k = 1;
204.        // loops from 0 to all other vertcies
205.        while (k < numberOfDistinctColors) //O(N)
206.        {
207.

```

```

208. // calculating the distance
209. byte red1 = colorSet[localLeast].red;//O(1)
210. byte green1 = colorSet[localLeast].green;//O(1)
211. byte blue1 = colorSet[localLeast].blue;//O(1)
212. byte red2 = colorSet[k].red;//O(1)
213. byte green2 = colorSet[k].green;//O(1)
214. byte blue2 = colorSet[k].blue;//O(1)
215. double red = (red1 - red2) * (red1 - red2);//O(1)
216. double green = (green1 - green2) * (green1 - green2);//O(1)
217. double blue = (blue1 - blue2) * (blue1 - blue2);//O(1)
218. dist = Math.Sqrt(red + green + blue);//O(1)
219. // adding the edge to the initial MST
220. MST.Add(new edge(dist, 0, k));//O(1)
221. graph[localLeast].Add(k);//O(1)
222. graph[k].Add(localLeast);//O(1)
223. // checks if it is the minimum edge to start from its end
224. if (dist < min)//O(1)
225. {
226.     // change the minimum
227.     min = dist;//O(1)
228.     // save the index of the next start over
229.     least = k;//O(1)
230. }
231. k++;
232. }
233. k = 1;
234. // looping from every vertex except of 0 to the others
235. while (k < numberOfDistinctColors)//O(N)
236. {
237.     //reset the min for every vertex
238.     min = 99999;
239.     //set the startPoint as optimal way found
240.     partnersOfTheTree[least] = true;
241.     // reserve the start point index to prevent from modifications
242.     localLeast = least;
243.     // check the distance to all others from the current start point
244.     for (int j = 1; j < numberOfDistinctColors; j++)//O(D)
245.     {
246.         // exclude the optimal points if found
247.         if (partnersOfTheTree[j] == false)//O(D)
248.         {
249.             // calculating the distance
250.             byte red1 = colorSet[localLeast].red;//O(1)
251.             byte green1 = colorSet[localLeast].green;//O(1)
252.             byte blue1 = colorSet[localLeast].blue;//O(1)
253.             byte red2 = colorSet[j].red;//O(1)
254.             byte green2 = colorSet[j].green;//O(1)
255.             byte blue2 = colorSet[j].blue;//O(1)
256.             double red = (red1 - red2) * (red1 - red2);//O(1)
257.             double green = (green1 - green2) * (green1 - green2);//O(1)
258.             double blue = (blue1 - blue2) * (blue1 - blue2);//O(1)
259.             dist = Math.Sqrt(red + green + blue);//O(1)
260.             // hold the edge to reduce access time
261.             tmp = MST[j];//O(1)
262.             //check if it is a better way to this vertex
263.             if (dist < tmp.distance)//O(1)
264.             {
265.                 graph[j].Remove(MST[j].points[0]);//O(N)
266.                 graph[MST[j].points[0]].Remove(j);//O(N)
267.                 //replace the distance
268.                 MST[j].distance = dist;//O(1)
269.                 //replace the starting index
270.                 MST[j].points[0] = localLeast;//O(1)
271.                 graph[j].Add(MST[j].points[0]);//O(1)
272.                 graph[MST[j].points[0]].Add(j);//O(1)
273.             }
274.             //check if it is the best edge in the tree to set the next start point
275.             if (tmp.distance < min)//O(1)
276.             {
277.                 //change the min
278.                 min = tmp.distance;//O(1)

```

```

279.         // reserve the index
280.         least = tmp.points[1]; //O(1)
281.     }
282. }
283. }
284.     k++;
285. }
286. MST_SUM = 0;
287. //calculating the MST SUM
288. for (int i = 0; i < MST.Count; i++) MST_SUM = MST_SUM + MST[i].distance; //O(N)
289. MSTTime.Stop();
290. overAllTime.Stop();
291. phase1Time = overAllTime.ElapsedMilliseconds;
292. //Console.WriteLine(MST_SUM + " MST SUM");
293. MessageBox.Show(MST_SUM.ToString() + " MST SUM IN TIME " + MSTTime.ElapsedMilliseconds + " ms");
294. //Console.WriteLine(MSTCOUNT - 1 + " MST COUNT");
295. //Console.WriteLine(MEAN + " MST MEAN");
296. ///List<double> std = new List<double>();
297. //double stdSUM = 0;
298. //int current = 0;
299.
300.     return MST;
301. }
302. public static double MST_SUM = 0;
303. public static List<List<int>> clustering(int numberOfDistinctColors, List<edge> MST, int k)
304. {
305.     Stopwatch clusterTime = new Stopwatch();
306.     clusterTime.Start();
307.     overAllTime.Restart();
308.     int cluster_counter = 0; //O(1)
309.     List<List<int>> clusters = new List<List<int>>(); //O(1)
310.     Queue<int> q = new Queue<int>(); //O(1)
311.     bool[] partnersOfTheTree = new bool[numberOfDistinctColors]; //O(1)
312.     MST.Sort(); //O(Nlog(N))
313.     int MSTCOUNT = MST.Count() - 1; //O(1)
314.     double MEAN = MST_SUM / MSTCOUNT; //O(1)
315.     double stdSUM = 0; //O(1)
316.     int STDcount = MSTCOUNT - 1; //O(1)
317.     for (int i = 0; i < MSTCOUNT; i++) //O(E) bounds loop
318.         stdSUM += (MST[i].distance - MEAN) * (MST[i].distance - MEAN);
319.     stdSUM /= STDcount;
320.     stdSUM = Math.Sqrt(stdSUM);
321.     double stdSUM_loop = stdSUM;
322.     int h = 0; //O(1)
323.     int r = MSTCOUNT - 1; //O(1)
324.     while (Math.Abs(stdSUM - stdSUM_loop) >= 0.0001 || h == 0) //O(E) bounds loop
325.     {
326.         stdSUM = stdSUM_loop;
327.         stdSUM_loop = 0;
328.         if (Math.Abs(MST[r].distance - MEAN) >= MST[h].distance - MEAN)
329.         {
330.             graph[MST[r].points[0]].Remove(MST[r].points[1]); //O(N)
331.             graph[MST[r].points[1]].Remove(MST[r].points[0]); //O(N)
332.             MEAN = MEAN * (STDcount + 1);
333.             MEAN = MEAN - MST[r].distance;
334.             MEAN = MEAN / STDcount;
335.             r--;
336.             for (int i = h; i < r; i++) //O(E) bounds loop
337.                 stdSUM_loop += (MST[i].distance - MEAN) * (MST[i].distance - MEAN);
338.             stdSUM_loop /= STDcount;
339.             stdSUM_loop = Math.Sqrt(stdSUM_loop);
340.         }
341.         else
342.         {
343.             graph[MST[h].points[0]].Remove(MST[h].points[1]); //O(N)
344.             graph[MST[h].points[1]].Remove(MST[h].points[0]); //O(N)
345.             MEAN = MEAN * (STDcount + 1);
346.             MEAN = MEAN - MST[h].distance;
347.             MEAN = MEAN / STDcount;
348.             h++;
349.             for (int i = h; i < r; i++) //O(E) bounds loop

```

```

350.         stdSUM_loop += (MST[i].distance - MEAN) * (MST[i].distance - MEAN);
351.         stdSUM_loop /= STDcount;
352.         stdSUM_loop = Math.Sqrt(stdSUM_loop);
353.     }
354.     STDcount--;
355.     if (STDcount == 0) break;
356. }
357. for (int i = 0; i < graph.Count; i++)//O(D)
358.     if (partnersOfTheTree[i] == false)
359.     {
360.         //Console.WriteLine(stdSUM + " MST STD");
361.         //Console.WriteLine("-----");
362.         clusters.Add(new List<int>());//O(1)
363.         q.Enqueue(i);//O(1)
364.         while (q.Count != 0)
365.         {
366.             //for (int i = 1; i < MSTCOUNT; i++) stdSUM += Math.Pow((MST[i].distance - MEAN), 2);
367.             var tmpQ = q.Dequeue();
368.             for (int j = 0; j < graph[tmpQ].Count; j++)//O(N)
369.                 if (partnersOfTheTree[graph[tmpQ][j]] == false)//O(1)
370.                     q.Enqueue(graph[tmpQ][j]);//O(1)
371.             partnersOfTheTree[tmpQ] = true;//O(1)
372.             clusters[cluster_counter].Add(tmpQ);//O(1)
373.             //double currentMean = MEAN;
374.             //int current_MSTCOUNT = MSTCOUNT;
375.         }
376.         cluster_counter++;
377.     }
378. clusterTime.Stop();
379. overAllTime.Stop();
380. MessageBox.Show("number of clusters " + clusters.Count + " --> clustering done in " + clusterTime.ElapsedMilliseconds + " ms");
381. return clusters;
382. }
383. public static List<RGBPixel> generatePallette(List<List<int>> clusters, List<RGBPixel> colorSet, ref int[, ] mapper)
384. {
385.     overAllTime.Start();
386.     Stopwatch generatePalletteTime = new Stopwatch();
387.     generatePalletteTime.Start();
388.     //re intialize the mapper to overwrite any old data
389.     mapper = new int[256, 256, 256];//O(1)
390.     List<RGBPixel> palletaa = new List<RGBPixel>();
391.     int clusterCounter = 0, count = clusters.Count;
392.     for (int i = 0; i < count; i++)//O(N)
393.     {
394.         // holds the color
395.         RGBPixel current;//O(1)
396.         double red = 0, green = 0, blue = 0;//O(1)
397.         int numberOfClusterElements = clusters[i].Count();//O(1)
398.         // clculates the avarage of red , green and blue
399.         foreach (var j in clusters[i])
400.         {
401.             red = red + colorSet[j].red;
402.             green = green + colorSet[j].green;
403.             blue = blue + colorSet[j].blue;
404.             //sets the reference to the color to its cluster number
405.             mapper[colorSet[j].red, colorSet[j].green, colorSet[j].blue] = clusterCounter;//O(1)
406.         }
407.         // overcoming the error from clustering step by skipping every empty cluster
408.         if (clusters[i].Count == 0) continue;
409.         // round to handle double numbers error
410.         current.red = (byte)Math.Round(red / numberOfClusterElements);//O(1)
411.         current.green = (byte)Math.Round(green / numberOfClusterElements);//O(1)
412.         current.blue = (byte)Math.Round(blue / numberOfClusterElements);//O(1)
413.         // add the color to the pallet list in the same reference to the mapper
414.         palletaa.Add(current);//O(1)
415.         // change ther reference
416.         clusterCounter++;//O(1)
417.     }
418.     generatePalletteTime.Stop();
419.     overAllTime.Stop();
420.     MessageBox.Show("generating Pallet done in time " + generatePalletteTime.ElapsedMilliseconds + " ms");

```

```

421.     return palletaa;
422. }
423. public static RGBPixel[,] Quantize(RGBPixel[,] ImageMatrix, List<RGBPixel> p, int[,.] mapper)
424. {
425.     Stopwatch QuantizeTime = new Stopwatch();
426.     QuantizeTime.Start();
427.     overAllTime.Start();
428.     int Height = GetHeight(ImageMatrix);
429.     int Width = GetWidth(ImageMatrix);
430.     RGBPixel[,] Filtered = new RGBPixel[Height, Width];
431.     //loop to get every pixel in the picture to replace
432.     for (int i = 0; i < Height; i++)//O(N)
433.         for (int j = 0; j < Width; j++)//O(N)
434.             {
435.                 RGBPixel quantized = new RGBPixel();
436.                 //gets the pallet reference from the mapper then saves the color
437.                 quantized = p[mapper[ImageMatrix[i, j].red, ImageMatrix[i, j].green, ImageMatrix[i, j].blue]];//O(1)
438.                 //replace the old color by its representative
439.                 Filtered[i, j].red = quantized.red;//O(1)
440.                 Filtered[i, j].green = quantized.green;//O(1)
441.                 Filtered[i, j].blue = quantized.blue;//O(1)
442.             }
443.     QuantizeTime.Stop();
444.     overAllTime.Stop();
445.     MessageBox.Show("image mapping colors done in time " + QuantizeTime.ElapsedMilliseconds + " ms");
446.     MessageBox.Show("over all time is " + (overAllTime.ElapsedMilliseconds + phase1Time) + " ms");
447.     MessageBox.Show("over all time is " + ((overAllTime.ElapsedMilliseconds + phase1Time) / 1000) + " S");
448.     MessageBox.Show("over all time is " + ((overAllTime.ElapsedMilliseconds + phase1Time) / 1000 / 60) + " M");
449.     return Filtered;
450. }
451. }
452. }

```

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, 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.generatePallette(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. }

```

Sample tests:

Time	Task 1 (ms)	Task 2 (ms)	Task 3 (ms)	Task 4 (ms)	Task 5 (ms)	Total (ms)	Total (S)
Case 1	0	0	0	0	0	0	0
Case 2	0	0	0	0	0	1	0.001
Case 3	1	144	1	1	0	148	0.148
Case 4	6	1	0	1	6	17	0.017
Case 5	2	1	0	1	1	7	0.007

Complete tests:

Time	Task 1 (ms)	Task 2 (ms)	Task 3 (ms)	Task 4 (ms)	Task 5 (ms)	Total (ms)	Total (S)
Small 1	2	2174	2	2	1	2184	2.184
Small 2	8	3107	2	5	7	3138	3.138
Mid 1	94	24036	9	6	109	24255	24.255
Mid 2	252	12716	5	5	338	13442	13.442
Large 1	1300	76232	33	9	1601	79178	79.178
Large 2	2083	92243	16	7	2673	97024	97.024

Time re calculations with bonus :

Sample tests:

Time	Task 1 (ms)	Task 2 (ms)	Task 3 (ms)	Task 4 (ms)	Task 5 (ms)	Total (ms)	Total (S)	#Clusters
Case 1	0	0	0	0	0	1	0.001	4
Case 2	0	0	0	0	0	0	0	2
Case 3	1	150	16	1	0	170	0.170	1455
Case 4	5	1	1	1	19	33	0.033	66
Case 5	1	4	0	0	21	30	0.030	255

Complete tests:

Time	Task 1 (ms)	Task 2 (ms)	Task 3 (ms)	Task 4 (ms)	Task 5 (ms)	Total (ms)	Total (S)	#Clusters
Small 1	2	2411	91	2	0	2529	2.529	1348
Small 2	9	3406	156	4	7	3677	3.677	1743
Mid 1	84	24474	148	4	94	24974	24.974	591
Mid 2	212	14593	360	4	281	15699	15.699	2286
Large 1	1143	91717	247	8	1394	95156	95.156	480
Large 2	1897	82618	59	4	2277	88506	88.506	93