Image Quantization

Finding distinct colors:

- Description:

  We iterate over the image and check if the color of each pixel was seen before if not, we store it in an array called dis_color

  Note: we check if the color was seen before through a boolean 3d array of size 256*256*256 to enable us to try all combinations of colors, for example: exist[0][0][255] = 1 means that the color 0,0,225 was seen before and is already stored in the dis_color array

- Code:

```
public static int dis_counter = 0;//O(1)
        public static RGBPixel[]  Distinct_colors( RGBPixel[,]
ImageMatrix, int hight, int width)
        {
            bool[,,] exist = new bool[256, 256, 256];//O(1)

            for (int i = 0; i < hight; i++)//O(n^2) = //O(height*width)
            {
                for (int j = 0; j < width; j++)//O(n) = O(width)
                {
                    if (!exist[ImageMatrix[i, j].red, ImageMatrix[i,
j].green, ImageMatrix[i, j].blue])//O(1)
                    {
                        exist[ImageMatrix[i, j].red, ImageMatrix[i,
j].green, ImageMatrix[i, j].blue] = true;//O(1)
                        dis_counter++;//O(1)
                    }
                }
            }
```

```
            RGBPixel[] dis_color = new RGBPixel[dis_counter];//O(1)
            Array.Clear(exist, 0, exist.Length);//O(1)
            int cnt = 0;//O(1)
            for (int i = 0; i < hight; i++)//O(n^2) = //O(height*width)
            {
                for (int j = 0; j < width; j++)//O(n) = O(width)
                {
                    if (!exist[ImageMatrix[i, j].red, ImageMatrix[i,
j].green, ImageMatrix[i, j].blue])//O(1)
                    {
                        exist[ImageMatrix[i, j].red, ImageMatrix[i,
j].green, ImageMatrix[i, j].blue] = true;//O(1)
                        dis_color[cnt++] = ImageMatrix[i, j];//O(1)
                    }
                }
            }
            return dis_color;//O(1)
        }
```

Overall finding distinct colors time complexity: O(n^2) = O(height*width)

Graph construction:

- Description:
  - We calculate the distances between each distinct color and all other distinct colors
- Code:

```
public mst_graph(RGBPixel[] arr)
        {
            //initialization
            a = ImageOperations.dis_counter;//O(1)
            bool[] visited = new bool[a + 1];//O(1)
            int idx = 1, source = -1;//O(1)
            res = new arc[a];//O(1)
            for (int i = 0; i < a; ++i) //O(d)
                res[i] = new arc();//O(1)
            double min = double.MaxValue;//O(1)
            /*calculated the distance between the first distinct color
and all other colors
             * and we store these edges and mark the first distinct
color as visited.*/
           visited[0] = true;//O(1)
            int v = 0;//O(1)
            for (int j = 1; j < ImageOperations.dis_counter; j++)//O(d)
            {
                double db = arr[0].blue - arr[j].blue;//O(1)
                db = Math.Pow(db, 2);//O(1)
                double dr = arr[0].red - arr[j].red;//O(1)
                dr = Math.Pow(dr, 2);//O(1)
                double dg = arr[0].green - arr[j].green;//O(1)
                dg = Math.Pow(dg, 2);//O(1)
                double ww = Math.Sqrt(db + dr + dg);//O(1)
                res[idx].s = 0;//O(1)
                res[idx].d = j;//O(1)
                res[idx].w = ww;//O(1)
                idx++;//O(1)
                if (min > ww)//O(1)
                {
```

```
                    min = ww;//O(1)
                    v = j;//O(1)
                }
            }
            visited[v] = true;//O(1)
            //Calculate distances from the new source and update the
edges till the mst is constructed
            for (int e = 0; e < a - 2; e++)//O(d^2)
            {
                min = double.MaxValue;//O(1)
                source = v;//O(1)
                for (int j = 0; j < ImageOperations.dis_counter;
j++)//O(d)
                {
                    if (visited[j])//O(1)
                    { continue; }//O(1)
                    double db = arr[source].blue - arr[j].blue;//O(1)
                    db = Math.Pow(db, 2);//O(1)
                    double dr = arr[source].red - arr[j].red;//O(1)
                    dr = Math.Pow(dr, 2);//O(1)
                    double dg = arr[source].green - arr[j].green;//O(1)
                    dg = Math.Pow(dg, 2);//O(1)
                    double ww = Math.Sqrt(db + dr + dg);//O(1)
                    if (ww < res[j].w)//O(1)
                    {
                        res[j].s = source;//O(1)
                        res[j].w = ww;//O(1)
                        res[j].d = j;//O(1)
                    }
                    if (min > res[j].w)//O(1)
                    {
```

```
                min = res[j].w;//O(1)
                v = j;//O(1)
            }
        }
        visited[v] = true;//O(1)

    }
```

Overall graph construction time complexity: O(d^2)

Minimum spanning tree:

MST construction steps

1. First, we calculate the distance between the first distinct color (the first source) and all other colors and we store these edges and mark the first distinct color as visited.
2. Then, we choose the smallest distance to the destination color
3. Now, the destination color becomes our new source and we mark it as visited.
4. We calculate the distance between the source and all other distinct colors except the visited ones and update the edge if we could reach the same destination with less distance than the one already stored in that edge
5. We repeat the steps from 2 to 4 until the number of edges is equal to the number of distinct colors - 1 (The number of edges in the mst)

Conclusion: we applied prim's algorithm which builds the MST while calculating the distances as described in the steps above
So we don't have to use memory to store all the d^2 edges, we only store the edges constructing the MST (d-1 edges)

The prim's algorithm enabled us to store the MST edges while calculating the distances, so we don't have to store all the the graph edges then use them to find the MST, the conditions below enables us to store the MST edges only

```
if (ww < res[j].w)//O(1)
                    {
                        res[j].s = source;//O(1)
                        res[j].w = ww;//O(1)
                        res[j].d = j;//O(1)
                    }
                    if (min > res[j].w)//O(1)
                    {
                        min = res[j].w;//O(1)
                        v = j;//O(1)
                    }
```

Then we sort the MST edges, and loop on them to calculate the MST cost as shown below

```
        Array.Sort(res);//O(d log(d))
          //Calculate the minimum spanning tree cost
          double minimumCost = 0;//O(1)
          for (int i = 1; i < ImageOperations.dis_counter; ++i)//O(d)
          {
              //Console.WriteLine(res[i].s + " -- "
              //                  + res[i].d
              //                  + " == " + res[i].w);
              minimumCost += res[i].w;//O(1)
          }
          minimumCost= (double)Math.Round((decimal)minimumCost, 2,
```

```
MidpointRounding.ToEven);//O(1)
            Console.WriteLine("Cost of MST = "+ minimumCost);//O(1)
            Console.WriteLine("Number of distinct colors = " +
ImageOperations.dis_counter);//O(1)
        }
```

Palette generation

- **Description:**

  We iterate over the distinct colors and add each color to the sum color of the cluster to which the color belongs.
  Then we divide this sum by the number of colors in the cluster
  Note: we have a struct called node. It has an attribute called parent which represents the cluster id to which a node belongs

- **Code:**

```
public Dictionary<int, RGBPixel> average(node[] nodes, int k,
RGBPixel[] arr)
        {

            RGBPixelD[] sum = new
RGBPixelD[ImageOperations.dis_counter];//O(1)
            bool[] visited = new
bool[ImageOperations.dis_counter];//O(1)
            int[] cnt = new int[ImageOperations.dis_counter];//O(1)
            Dictionary<int, RGBPixel> avg = new Dictionary<int,
RGBPixel>(k);//O(k)
            for (int i = 0; i < ImageOperations.dis_counter; i++)//(d)
            {
```

```
                sum[nodes[i].parent].red += arr[i].red;//O(1)
                sum[nodes[i].parent].blue += arr[i].blue;//O(1)
                sum[nodes[i].parent].green += arr[i].green;//O(1)
                cnt[nodes[i].parent]++;//O(1)
            }
            for (int i = 0; i < ImageOperations.dis_counter; i++)//O(d)
            {
                if (!visited[nodes[i].parent])//O(1)
                {
                    RGBPixel y = new RGBPixel();//O(1)
                    double red = sum[nodes[i].parent].red /
(double)cnt[nodes[i].parent];//O(1)
                    double green = sum[nodes[i].parent].green /
(double)cnt[nodes[i].parent];//O(1)
                    double blue = sum[nodes[i].parent].blue /
(double)cnt[nodes[i].parent];//O(1)
                    sum[nodes[i].parent].red =
(byte)Math.Round((decimal)red, 0);//O(1)
                    sum[nodes[i].parent].blue =
(byte)Math.Round((decimal)blue, 0);//O(1)
                    sum[nodes[i].parent].green =
(byte)Math.Round((decimal)green, 0);//O(1)
                    visited[nodes[i].parent] = true;//O(1)
                    y.red = (byte)sum[nodes[i].parent].red;//O(1)
                    y.green = (byte)sum[nodes[i].parent].green;//O(1)
                    y.blue = (byte)sum[nodes[i].parent].blue;//O(1)
                    avg[nodes[i].parent] = y;//O(1)
                }

            }
```

```
//Console.WriteLine("The color palette:");//O(1)
            //foreach (var x in avg)//O(k)
            //{

            //    Console.WriteLine(x.Value.red + ", " + x.Value.green +
", " + x.Value.blue);//O(1)
            //}
            return avg;//O(1)
        }
```

Palette generation overall time complexity: O(d)

Mapping

- Description:

  We iterate over the image and replace each pixel color with the average of the colors in the cluster to which it belongs

  Note: To do the mentioned steps, we need to map each color in the image to an id so that we can know the cluster to which a color belongs.

  So we iterate over the image and give each color an id through a 3d array called map

  For example: map[0][0][255] = 5 means that the color 0,0,255 has the id 5

- Code:

```
public RGBPixel[,] mapping(Dictionary<int, RGBPixel> avg, node[] nodes,
int height, int width, RGBPixel[,] image, RGBPixel[] my)
        {
```

```
            int[,,] map = new int[256, 256, 256];//O(1)
            for (int i = 0; i < ImageOperations.dis_counter; i++)//O(d)
            {
                map[my[i].red, my[i].green, my[i].blue] = i;//O(1)
            }
            for (int i = 0; i < height; i++)//O(n^2) = O(height*width)
            {
                for (int j = 0; j < width; j++)//O(n) = O(width)
                {
                    image[i, j] = avg[nodes[map[image[i, j].red,
image[i, j].green, image[i, j].blue]].parent];//O(1)

                    // Console.WriteLine(nodes[image[i, j].id].parent);
                    //Console.WriteLine(image[i, j].id);
                }
            }

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

Mapping overall time complexity: O(n^2) = O(width*height)

Clustering

- Description:
  - **The algorithm followed is kruskal disjoint sets with union by rank and path compression**

- First, we define each node as a separate cluster(each node has itself as a parent and it's rank is 0)
- We iterate over the sorted MST edges, and union the nearest two nodes in one cluster by finding the parent of each node till we reach the desired number of clusters
- **Find parent:** When we call find parent for a node, we traverse up the tree until we reach the root which is the parent of itself. We assign the root as the parent of the node so that we don't have to traverse the tree each time find parent is called for this node(Path compression)
- **Union:** We join the smaller rank tree to the higher rank tree(if both have equal rank, we join any one of them to the other)
- We stop the algorithm when we reach the desired number of clusters
- Finally, we iterate over all the nodes, and find parent of each one of them, so that each node has the root of its cluster as its parent

- Code:

```
public node[] clustering(int clusters)
        {
            int K = ImageOperations.dis_counter;//O(1)
            node[] nodes = new node[ImageOperations.dis_counter];//O(1)
            for (int i = 0; i < ImageOperations.dis_counter; i++)//O(d)
                nodes[i] = new node();//O(1)

            for (int i = 0; i < ImageOperations.dis_counter; i++)//O(d)
            {
```

```
                nodes[i].parent = i;//O(1)
                nodes[i].rank = 0;//O(1)
            }
            int e = 0;//O(1)
            int idx = 1;//O(1)
            while (e < ImageOperations.dis_counter - 1)//O(d log(d))
            {
                arc current_arc = new arc();//O(1)
                current_arc = res[idx++];//O(1)
                int x = find_parent(nodes, current_arc.s);//O(log(d))
                int y = find_parent(nodes, current_arc.d);//O(log(d))
                if (x != y)//O(1)
                {
                    e++;//O(1)
                    if (K == clusters)//O(1)
                    {
                        for (int j = 0; j < ImageOperations.dis_counter;
j++)//O(d log(d))
                        {
                            nodes[j].parent = find_parent(nodes,
nodes[j].parent);//O(log(d))
                            //Console.WriteLine(j + " " +
nodes[j].parent);
                        }
                        return nodes;//O(1)
                    }
                    union(nodes, x, y);//O(log(d))
                    K--;//O(1)
                }
            }
            if (K == clusters)//O(1)
```

```
            {
                for (int j = 0; j < ImageOperations.dis_counter;
j++)//O(d log(d))
                {
                    nodes[j].parent = find_parent(nodes,
nodes[j].parent);//O(log(d))
                    //Console.WriteLine(j + " " + nodes[j].parent);
                }

                return nodes;//O(1)
            }

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

Clustering overall time complexity: O(d log(d))

Find_parent code:

```
int find_parent(node[] nodes, int i)
        {
            if (nodes[i].parent != i)//O(1)
                return nodes[i].parent = find_parent(nodes,
nodes[i].parent);//O(log(d))

            return nodes[i].parent;//O(1)
        }
```

Find_parent overall time complexity: O(log(d))

Union code:

```
void union(node[] nodes, int x, int y)
        {
            int x_parent = find_parent(nodes, x);//O(log(d))
            int y_parent = find_parent(nodes, y);//O(log(d))

            if (nodes[x_parent].rank < nodes[y_parent].rank)//O(1)
            {
                nodes[x_parent].parent = y_parent;//O(1)
            }
            else if (nodes[x_parent].rank > nodes[y_parent].rank)//O(1)
            {
                nodes[y_parent].parent = x_parent;//O(1)
            }
            else
            {
                nodes[x_parent].parent = y_parent;//O(1)
                nodes[y_parent].rank++;//O(1)
            }
        }
```

Union overall time complexity: O(log(d))

Main Form code:

```
private void quantizebtn_Click(object sender, EventArgs e)
        {
            var execution_watch =
System.Diagnostics.Stopwatch.StartNew();//O(1)
            int h = ImageOperations.GetHeight( ImageMatrix);
            int w = ImageOperations.GetWidth( ImageMatrix);
            var my = ImageOperations.Distinct_colors(ImageMatrix, h,
w);//O(n^2) = O(height*width)
            mst_graph grap = new mst_graph(my);//O(d^2)
            var nodes = grap.clustering(clusters);//O(d log(d))
            Dictionary<int,RGBPixel> avg = grap.average(nodes, clusters,
my);//O(d)
            ImageMatrix = grap.mapping(avg, nodes, h, w, ImageMatrix,
my);//O(n^2) = O(height*width)
            ImageOperations.DisplayImage(  ImageMatrix, pictureBox2);
            execution_watch.Stop();//O(1)
            Console.WriteLine("Execution time = " +
execution_watch.ElapsedMilliseconds/1000 + "s " +
execution_watch.ElapsedMilliseconds % 1000 + "ms");//O(1)

        }
```

Output sample