Image Quantization Analysis

Graph Construction

- Since we have a dense graph it is not feasible to construct the full weighted graph for images.
- We instead calculate the required weight in the making of MST.

Minimum Spanning Tree Code

- We chose Prim's algorithm in getting the MST since we don't have a constructed graph and also we have a dense graph.

To get the MST we must have a list of the **distinct colors**, we get it by the following function:

`GetDistinctColorsList(ImageMatrix)`

- Takes **O(L*W)**, *where L is the length and W is the width of the image.*

```
private static List<int> GetDistinctColorsList(RGBPixel[,] ImageMatrix) //O(L*W)
{
    HashSet<int> distinctColors = new HashSet<int>();

    foreach (RGBPixel pixel in ImageMatrix) // O(L*W) if C# handles resizing
well
        distinctColors.Add(RGBPixel.Hash(pixel));

    List<int> colorsList = distinctColors.ToList();
    return colorsList;
}
```

We get the MST using Prim's algorithm by the following function:

`Prim(distinctColorsList)`

- Takes **O(D^2)** *where D is number of distinct colors in the image*.

```
public static int[] Prim(List<int> distinctColors) //O(D^2)
{
    int V = distinctColors.Count;
    int[] parent = new int[V];
    int[] key = new int[V];
    bool[] mstSet = new bool[V];
    //initialize all keys as infinite
    for (int i = 0; i < V; i++)             //O(D)
    {
        key[i] = int.MaxValue;
        mstSet[i] = false;
    }
    key[0] = 0;
    parent[0] = -1;
    for (int i = 0; i < V - 1; i++)             //O(D^2)
    {
        int u = GetMinimumKey(key, mstSet); //(D)
```

```
            mstSet[u] = true;
            //relax all edges connected to u
            for (int v = 0; v < V; v++)         //O(D)
            {
                int distance = ColorQuantization.GetWeight( //O(1)
                    distinctColors[u],
                    distinctColors[v]
                    );
                if (distance != 0
                    && mstSet[v] == false
                    && distance < key[v]
                    )
                {
                    parent[v] = u;

                    key[v] = distance;
                }
            }
        }
        return parent;
    }
```

The function `GetMinimumKey()` is used in `Prim()` :

- Takes **O(D)**, ***where D is the number of distinct colors in the image.***

```
    private static int GetMinimumKey(int[] key, bool[] mstSet)  //O(D)
    {
        int min = int.MaxValue;
        int min_index = -1;
        for (int v = 0; v < key.Length; v++)        //O(D)
        {
            if (mstSet[v] == false && key[v] < min)
            {
                min = key[v];
                min_index = v;
            }
        }
        return min_index;
    }
```

Palette Generation Code

We use the function `GetColorPalette` to get and generate the palette, this function take a list of clusters `List<List<RGBPixel>> clusters` and return a list that each index `i` is an index of a cluster and the value of that index is the representative color for the whole cluster `i`.

`GetColorPallette(clusters)`

- Takes **O(D)**, *where D is the number of distinct colors in the image.*

```
    public static List<RGBPixel> GetColorPallette(List<List<RGBPixel>> clusters)
    //O(D)
    {
        // for every member of cluster sum all values and get the mean for the sum
        List<RGBPixel> colorPallete = new List<RGBPixel>();
```

```
    for (int clusterIndex = 0; clusterIndex < clusters.Count; clusterIndex++)
    //this will loop over all nodes in all clusters,
    //since number of nodes is D then this whole loop takes
    //O(D)
    {
        int sumRed = 0, sumGreen = 0, sumBlue = 0;
        int numberOfColorsInCluster = clusters[clusterIndex].Count;

        foreach (RGBPixel pixel in clusters[clusterIndex])
        {
            sumRed += pixel.red;
            sumBlue += pixel.blue;
            sumGreen += pixel.green;

        }
        sumRed = (int)Math.Ceiling((double)sumRed / numberOfColorsInCluster);
        sumGreen = (int)Math.Ceiling((double)sumGreen /
numberOfColorsInCluster);
        sumBlue = (int)Math.Ceiling((double)sumBlue / numberOfColorsInCluster);

        byte red = Convert.ToByte(sumRed);
        byte green = Convert.ToByte(sumGreen);
        byte blue = Convert.ToByte(sumBlue);

        RGBPixel representitaveColor = new RGBPixel(red, green, blue);
        colorPallete.Add(representitaveColor);
    }
    return colorPallete;
}
```