Image Quantization Project

Team no. 61

Islam Hany Heggy

Norhan Khalf Saad

Menna Gamal Abdelrazek

Menna Mohamad Mostafa

Maha Mohamad Fathy

Project Documentation

Brief Description:

This project is based on reducing the number of colors in a digital color image to a smaller set of representative colors. Reduction should be formed so the resulted image is not much different from the original one.

Uses:

1. It can be used in image compression by reducing the colors number without distorting the image.
2. It can be used in displaying images on devices with limited number of colors

Technical Details

Quantization used in reducing colors by replacing the nearest colors with an average color which reduces the different color representation in the image and this may distort the image in some cases.

This project is divided into classes, the main class is ImageOperations class which contains all the operations done on the image starting from opening the image, compressing it and finally showing it on the screen.

The image colors (RGB) are imported in a 2d array called ImageMatrix.

This project is implemented in C#.

ImageOperations Class Functions:

OpenImage.

Getheight.

Getwidth.

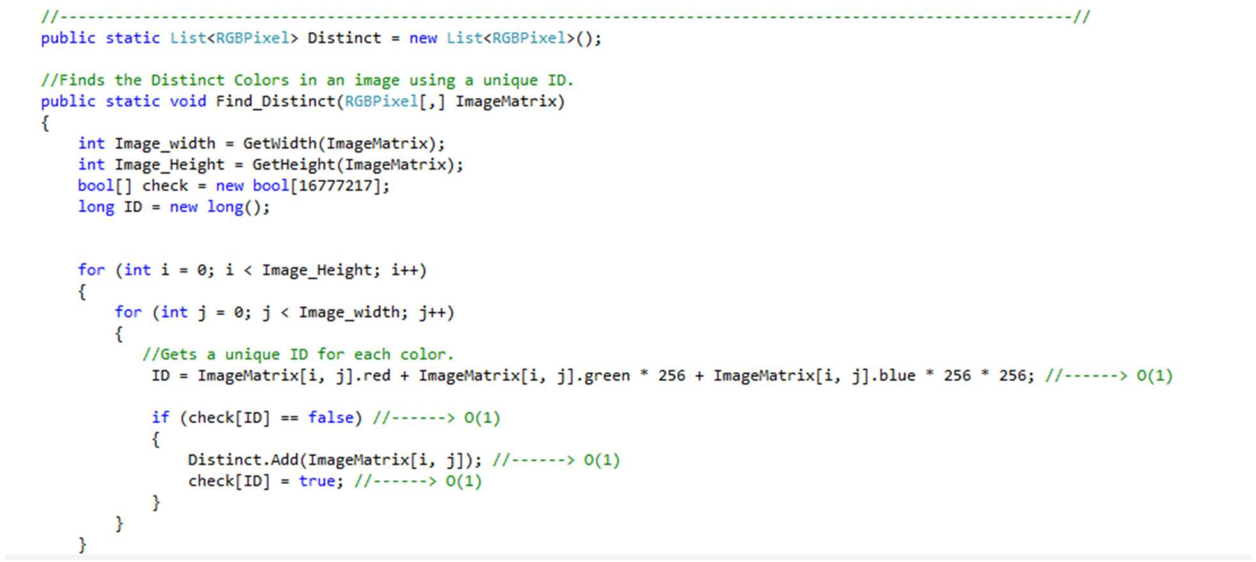
Find_Distinct.

Quantize_Image.

DisplayImage.

Going through Project:

1. To start working on the project you should find the distinct colors to find smaller representative colors.

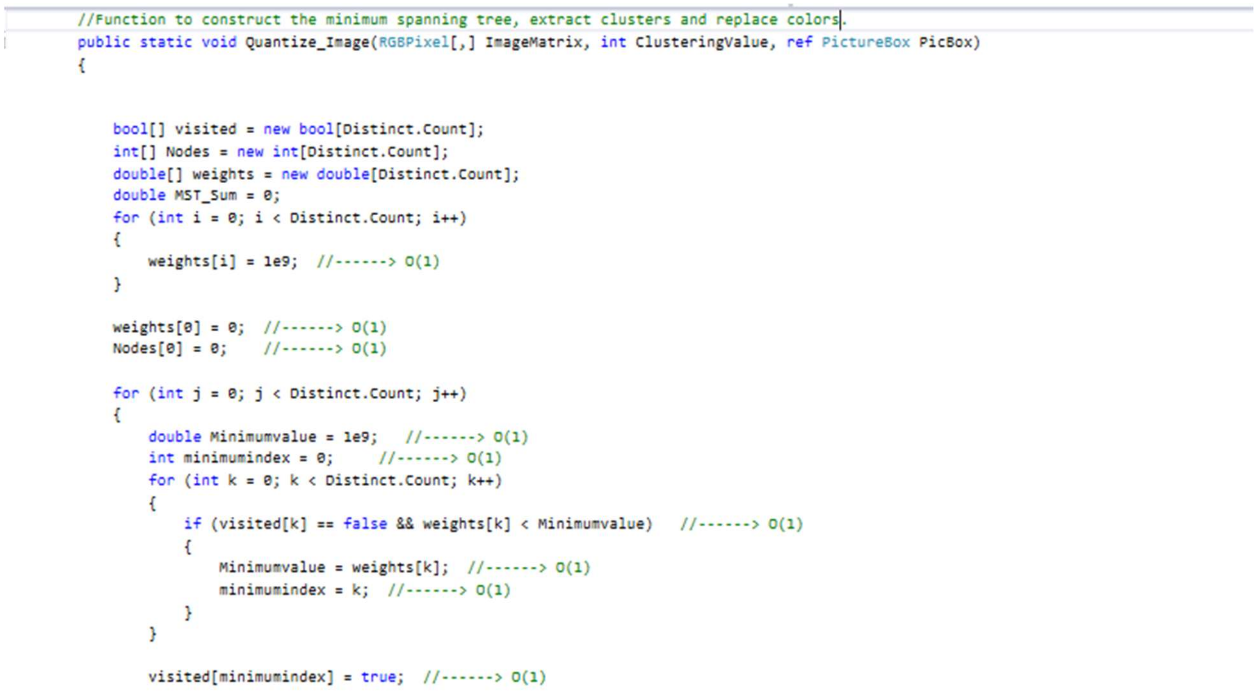


```
//----------------------------------------------------------------------------------------------------------//
public static List<RGBPixel> Distinct = new List<RGBPixel>();

//Finds the Distinct Colors in an image using a unique ID.
public static void Find_Distinct(RGBPixel[,] ImageMatrix)
{
    int Image_width = GetWidth(ImageMatrix);
    int Image_Height = GetHeight(ImageMatrix);
    bool[] check = new bool[16777217];
    long ID = new long();


    for (int i = 0; i < Image_Height; i++)
    {
        for (int j = 0; j < Image_width; j++)
        {
           //Gets a unique ID for each color.
            ID = ImageMatrix[i, j].red + ImageMatrix[i, j].green * 256 + ImageMatrix[i, j].blue * 256 * 256; //------> O(1)

            if (check[ID] == false) //------> O(1)
            {
                Distinct.Add(ImageMatrix[i, j]); //------> O(1)
                check[ID] = true; //------> O(1)
            }
        }
    }
```

Find_Distinct is responsible for extracting all unique colors and pushing them into a list.
This function loops on the 2d array image matrix to find all possible distinct colors while accessing with a unique id to allow accessing the Boolean array with O(1) operation at a time.

We tried using a hashset but with large inputs it makes collisions and needs resizing which costs a time O(D), where D is the distinct colors in the list.
The Boolean array can consume memory than O(D^2) in small inputs but starting from 4096 colors it uses less than O(D^2) of memory which is great.
This function works on time equals O(Image_Height*Image_width).

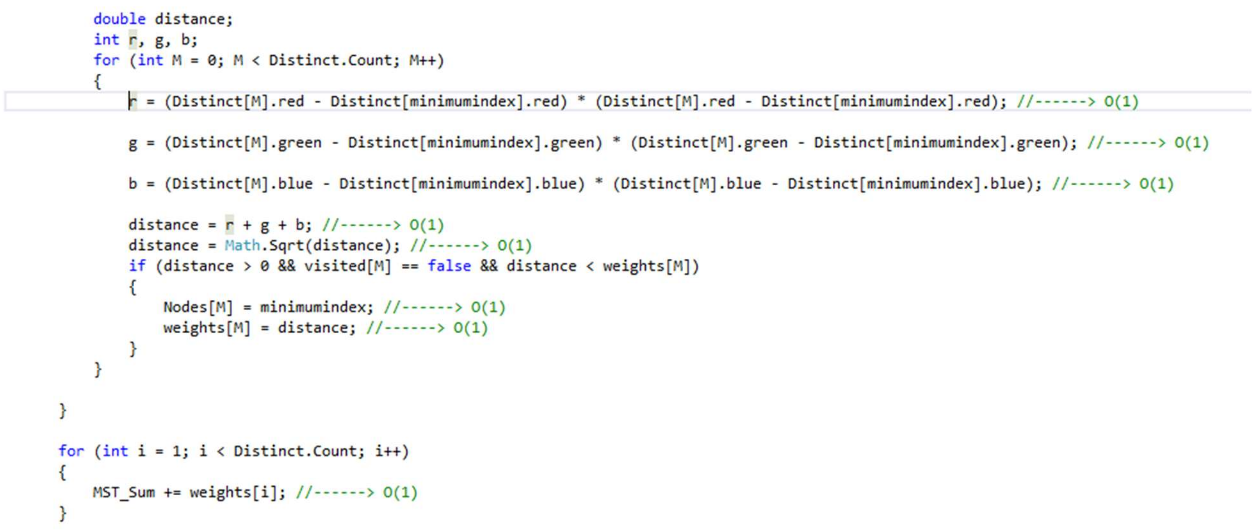
2. After that we started constructing the minimum spanning tree with least cost in one time without constructing a fully connected graph as it consumes huge memory resulting in loss of resources.



```
//Function to construct the minimum spanning tree, extract clusters and replace colors.
public static void Quantize_Image(RGBPixel[,] ImageMatrix, int ClusteringValue, ref PictureBox PicBox)
{


    bool[] visited = new bool[Distinct.Count];
    int[] Nodes = new int[Distinct.Count];
    double[] weights = new double[Distinct.Count];
    double MST_Sum = 0;
    for (int i = 0; i < Distinct.Count; i++)
    {
        weights[i] = 1e9;  //------> O(1)
    }

    weights[0] = 0;  //------> O(1)
    Nodes[0] = 0;    //------> O(1)

    for (int j = 0; j < Distinct.Count; j++)
    {
        double Minimumvalue = 1e9;   //------> O(1)
        int minimumindex = 0;     //------> O(1)
        for (int k = 0; k < Distinct.Count; k++)
        {
            if (visited[k] == false && weights[k] < Minimumvalue)   //------> O(1)
            {
                Minimumvalue = weights[k];  //------> O(1)
                minimumindex = k;  //------> O(1)
            }
        }

        visited[minimumindex] = true;  //------> O(1)
```

Finding the minimum spanning tree is our next step in the quantization process in which we need to find the minimum costs between the colors to be able to group them after that.

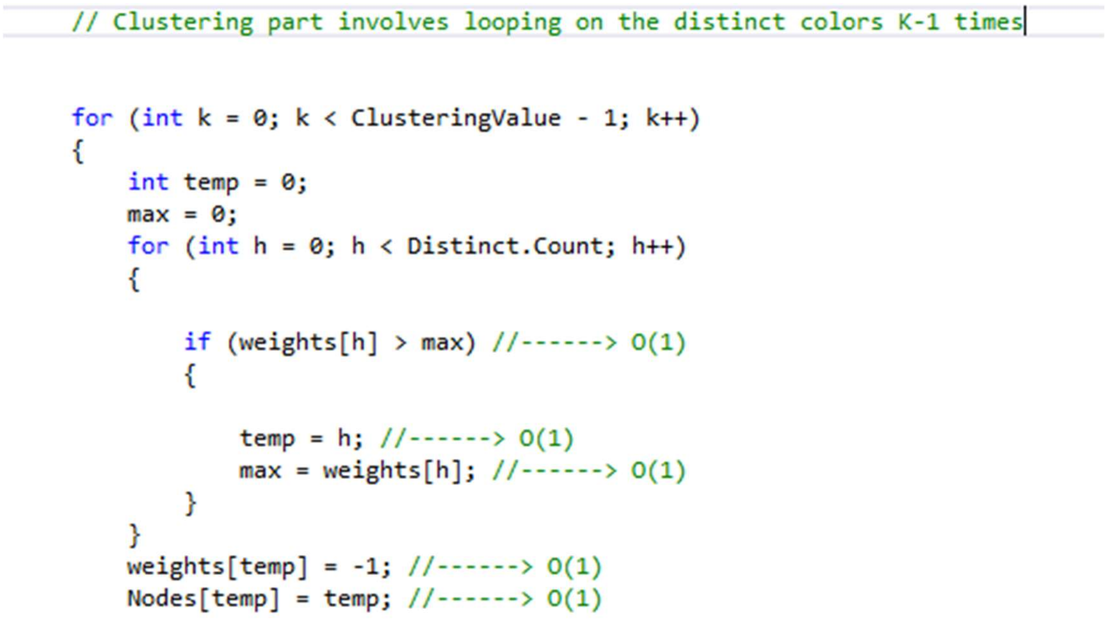
We took the list of distinct colors, applied prim’s algorithm and constructed the tree.



```
        double distance;
        int r, g, b;
        for (int M = 0; M < Distinct.Count; M++)
        {
            r = (Distinct[M].red - Distinct[minimumindex].red) * (Distinct[M].red - Distinct[minimumindex].red); //------> O(1)

            g = (Distinct[M].green - Distinct[minimumindex].green) * (Distinct[M].green - Distinct[minimumindex].green); //------> O(1)

            b = (Distinct[M].blue - Distinct[minimumindex].blue) * (Distinct[M].blue - Distinct[minimumindex].blue); //------> O(1)

            distance = r + g + b; //------> O(1)
            distance = Math.Sqrt(distance); //------> O(1)
            if (distance > 0 && visited[M] == false && distance < weights[M])
            {
                Nodes[M] = minimumindex; //------> O(1)
                weights[M] = distance; //------> O(1)
            }
        }

    }

    for (int i = 1; i < Distinct.Count; i++)
    {
        MST_Sum += weights[i]; //------> O(1)
    }
```

This function has a maximum time equals O(D^2), where D is the number of distinct colors.

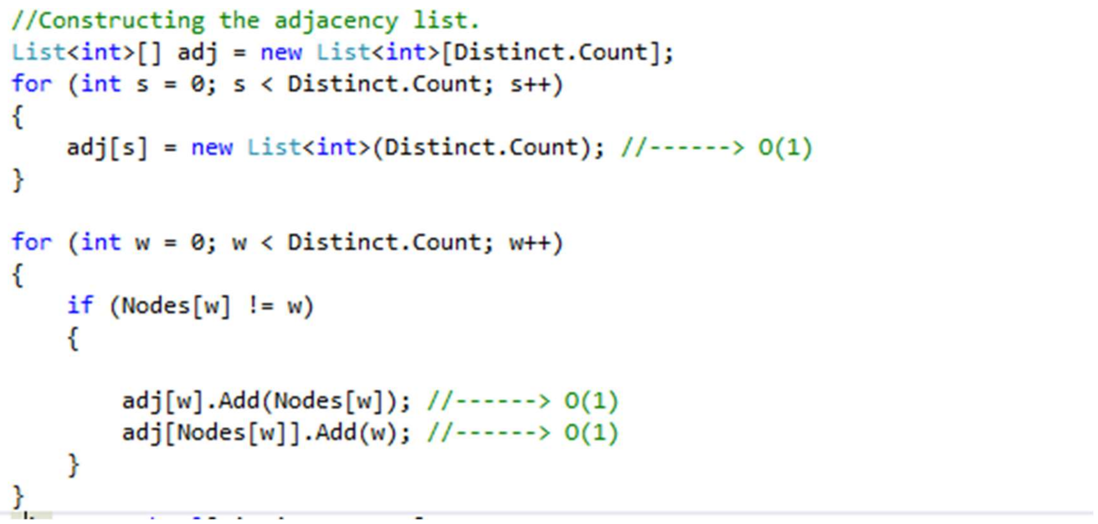
3. After that we go through the clustering part in which we cut our MST into K-1 maximum parts to find the nearest colors.



```
    // Clustering part involves looping on the distinct colors K-1 times


    for (int k = 0; k < ClusteringValue - 1; k++)
    {
        int temp = 0;
        max = 0;
        for (int h = 0; h < Distinct.Count; h++)
        {

            if (weights[h] > max) //------> O(1)
            {

                temp = h; //------> O(1)
                max = weights[h]; //------> O(1)
            }
        }
        weights[temp] = -1; //------> O(1)
        Nodes[temp] = temp; //------> O(1)
```

This function has a maximum time equals O(K*D) where K is the number of clusters and D is the number of Distinct colors.

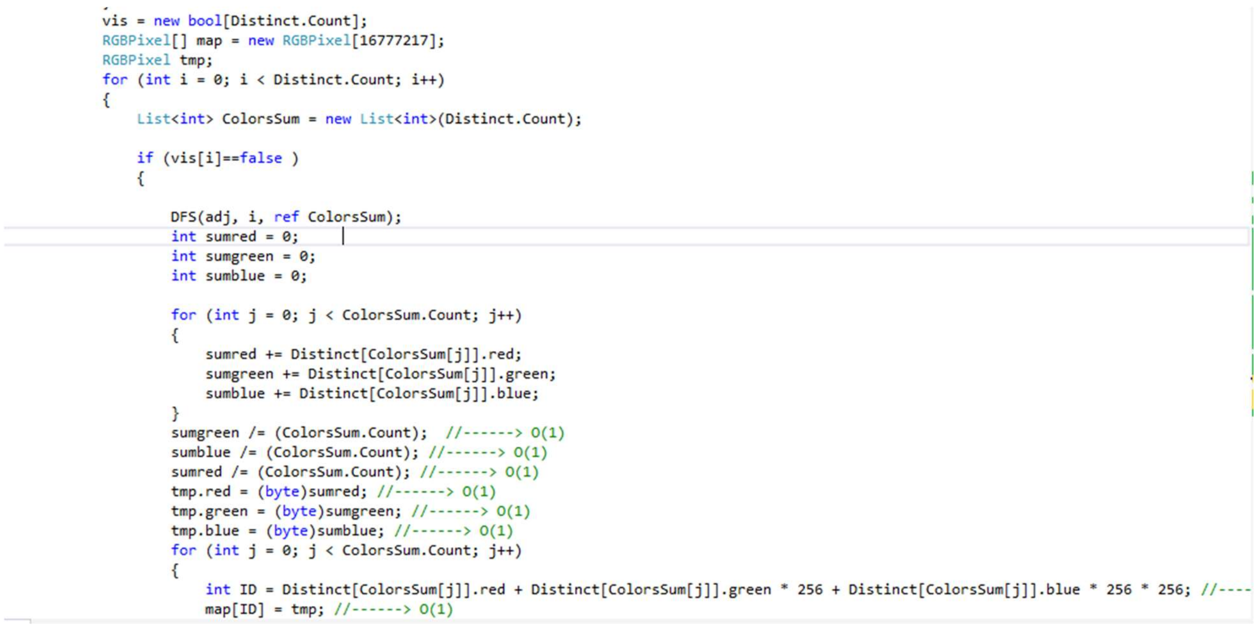
4. In this step, we should construct a graph from our final MST and then pass by all clusters get the sum then the average and replace the colors, so we chose to implement the graph via adjacency list implemented as an array of list.



```
//Constructing the adjacency list.
List<int>[] adj = new List<int>[Distinct.Count];
for (int s = 0; s < Distinct.Count; s++)
{
    adj[s] = new List<int>(Distinct.Count); //------> O(1)
}

for (int w = 0; w < Distinct.Count; w++)
{
    if (Nodes[w] != w)
    {

        adj[w].Add(Nodes[w]); //------> O(1)
        adj[Nodes[w]].Add(w); //------> O(1)
    }
}
```

This function takes the MST and put it in our adjacency list in time complexity equals to O(1) per insertion.
This Function has a maximum time complexity equals O(D) where D is the number of Distinct colors.

5. Finally we will move on the graph with DFS and sum all the cluster colors and get the average and then get the original color replaced by the quantized one through the unique ID



```
vis = new bool[Distinct.Count];
RGBPixel[] map = new RGBPixel[16777217];
RGBPixel tmp;
for (int i = 0; i < Distinct.Count; i++)
{
    List<int> ColorsSum = new List<int>(Distinct.Count);

    if (vis[i]==false )
    {

        DFS(adj, i, ref ColorsSum);
        int sumred = 0;
        int sumgreen = 0;
        int sumblue = 0;

        for (int j = 0; j < ColorsSum.Count; j++)
        {
            sumred += Distinct[ColorsSum[j]].red;
            sumgreen += Distinct[ColorsSum[j]].green;
            sumblue += Distinct[ColorsSum[j]].blue;
        }
        sumgreen /= (ColorsSum.Count);  //------> O(1)
        sumblue /= (ColorsSum.Count); //------> O(1)
        sumred /= (ColorsSum.Count); //------> O(1)
        tmp.red = (byte)sumred; //------> O(1)
        tmp.green = (byte)sumgreen; //------> O(1)
        tmp.blue = (byte)sumblue; //------> O(1)
        for (int j = 0; j < ColorsSum.Count; j++)
        {
            int ID = Distinct[ColorsSum[j]].red + Distinct[ColorsSum[j]].green * 256 + Distinct[ColorsSum[j]].blue * 256 * 256; //----
            map[ID] = tmp; //------> O(1)
```

and we will clear the distinct colors list to prevent the accumulation and call the display image function.

This project has a total time complexity equals O(D^2), where D is the distinct colors numbers.