Image Quantization Project

T89(Documentation)

- حسن وليد حسن السيد (20191700215)
- عبدالله رمضان السيد (2019700353)
- عصام الدين شريف جاب الله (20191700387)

Description:

The idea of color quantization is to reduce the number of colors in a full resolution digital color image (24 bits per pixel) to a smaller set of representative colors called color palette. Reduction should be performed so that the quantized image differs as little as possible from the original image. Algorithmic optimization task is to find such a color palette that the overall distortion is minimized.

Used Structure:

Hashset structure : to store a series of unique values,

Used functions:

```
static double distance(List<RGBPixel> dc , int parent_index , int child_index);
```

Details:

Grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters).

From the definition we can infer that:

1. **Set of objects** → are the set of distinct colors in the image (each color is a point in 3d (RGB) space)
2. **Similarity measure** is required for grouping similar set of point colors together → the Euclidean Distance between RGB intensity values of the color points can be used here.

   The Euclidean Distance between TWO colors is defined as:

$$\textbf{Distance} = \sqrt{(r1 - r2)^2 + (g1 - g2)^2 + (b1 - b2)^2}$$

Code:

```
1 reference
static double distance(List<RGBPixel> dc, int parent_index, int child_index)          //O(1)
{

    double r = (dc[parent_index].red - dc[child_index].red);                           //O(1)
    r = r * r; ;                                                                       //O(1)

    double g = (dc[parent_index].green - dc[child_index].green);                       //O(1)
    g = g * g;                                                                         //O(1)

    double b = (dc[parent_index].blue - dc[child_index].blue);                         //O(1)
    b = b * b;                                                                         //O(1)

    return Math.Sqrt(r + g + b);                                                       //O(1)
}
```

Analysis of the code:

Assignment operation => O(1)

Total = O(1)

DistinctColor Function:

Details:

Find all unique color in the image and giving each color a unique id and store them in List of distinctColor.

It is looping on the matrix (height,width) to find all possible unique color to allow accessing the Boolean array.

Code:

```
/////// distinct color/////
List<RGBPixel> distinct_color = new List<RGBPixel>();
long id1;                                       // O(1)
long id_R;                                      // O(1)
long id_G;
long id_B;
bool[] vaildation = new bool[16777216];         // O(1)
int Image_width = GetWidth(ImageMatrix);        // O(1)
int Image_Height = GetHeight(ImageMatrix);      // O(1)

foreach (RGBPixel pixel in ImageMatrix)         // O(D^2)
{
    id_R = pixel.red * 65536;           // O(1)
    id_G = pixel.green * 256;
    id_B = pixel.blue;                  // O(1)
    id1 = id_R + id_G + id_B;
    if (vaildation[id1] == false)       // O(1)
    {
        vaildation[id1] = true;         // O(1)
        distinct_color.Add(pixel);      // O(1)
    }
}
MessageBox.Show("Distinct Color = " + distinct_color.Count.ToString());
```

Analysis of the code:

$O(D^2)$ => where D number of distinctColor

MST Function:

Details:

Finding the minimum spanning tree in the quantization process by using Prim's algorithm the list of the distinct colors in which we need to find the minimum costs between the colors to be able to group them.

Analysis of the code:

Initializing priorities of each vertex: - O(V) → V the number of vertices.

Prim's Algorithm: - O(DElog(V)) → for each priority vertex

O(V)*modifying the priority O(log(V)) * for each edge O(E) .

Total: - O(V) + O(Elog(V)) = **O(Elog(V)).**

Code:

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////MST/////////////////////////////////
double[] weigth = new double[distinct_color.Count];             //O(1)
bool[] viste = new bool[distinct_color.Count];                  //O(1)
int[] parnt = new int[distinct_color.Count];                    //O(1)
int[] ch = new int[distinct_color.Count];                       //O(1)

for (int i = 0; i < distinct_color.Count; i++)                  //O(D)
{
    weigth[i] = double.MaxValue;                                //O(1)
    viste[i] = false;
}

weigth[0] = 0;
double sum = 0;
int num_Node = 0;
while (num_Node < distinct_color.Count)                         //O(D)
{
    int index_of_min = -1;
    double min = double.MaxValue;
    for (int j = 0; j < distinct_color.Count; j++)              //O(D)
    {
        if (viste[j] == false && weigth[j] < min)               //O(1)
        {
            min = weigth[j];
            index_of_min = j;
        }
    }
    viste[index_of_min] = true;

    for (int x = 0; x < distinct_color.Count; x++)              //O(D)
    {
        double dis = distance(distinct_color, index_of_min, x);
        if (dis < weigth[x] && viste[x] == false && dis > 0)    //O(1)
        {
            weigth[x] = dis;

            parnt[x] = index_of_min;
            ch[x] = x;
        }
    }
    num_Node++;
}
list_of_link = new List<link>();
for (int i = 0; i < distinct_color.Count; i++)                  //O(D)
{
    sum += weigth[i];
    list_of_link.Add(new link()                                 //O(1)
    {
        parent = parnt[i],
        child = ch[i],
        value = weigth[i]
    });

}
MessageBox.Show("MST = " + sum.ToString());
```

Cluster Function:

Details:

The clustering part in which we cut our MST into K-1 maximum parts to find the nearest colors.

Get number of clusters to group them later built on that number (means number of grouping).

Analysis of the code:

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.

Total: O(K*D)

Code:

```
//////////////////clusters////////////////////////////////
for (int k = 0; k < num_K - 1; k++)                  //O(K)
{
    int index = 0;
    double max_weight = 0;
    for (int h = 0; h < list_of_link.Count; h++)     //O(D)
    {

        if (list_of_link[h].value > max_weight) //------> O(1)
        {
            index = h; //------> O(1)
            max_weight = list_of_link[h].value; //------> O(1)
        }
    }
    int o_child = list_of_link[index].child;
    list_of_link[index] = new link()
    {
        parent = index,
        child = o_child,
        value = -1
    };

}                                                //------>    O(K*D)

HashSet<int>[] adj = new HashSet<int>[list_of_link.Count];  //O(1)
HashSet<int>[] list_of_clusters = new HashSet<int>[num_K];  //O(1)
HashSet<RGBPixel> list_palette = new HashSet<RGBPixel>();   //O(1)

for (int i = 0; i < num_K; i++)                             //O(K)
{
    list_of_clusters[i] = new HashSet<int>();
}
for (int s = 0; s < list_of_link.Count; s++)                //O(D)
{
    adj[s] = new HashSet<int>();
}
for (int w = 0; w < list_of_link.Count; w++)                //O(D)
{


    if (list_of_link[w].parent != w)                        //O(1)
    {
        int x = list_of_link[w].parent;                     //O(1)
        // Console.WriteLine("x =" + x);
        //HashSet<int> temp = new HashSet<int>();
        //temp.Add(x);
        adj[w].Add(x);
        //temp.Remove(x);
        //temp.Add(w);

        adj[list_of_link[w].parent].Add(w);
        //Console.WriteLine("W= "+w);
    }
}
```

```
bool[] v = new bool[adj.Length];
for (int i = 0; i < adj.Length; i++)                              // o(D)
{
    v[i] = false;
}
int num_of_empty_lists = 0;
int temp_new_adj = 0;
for (int i = 0; i < adj.Length; i++)                          //O(D)
{
    if (adj[i].Count == 0 && v[i] == false)                   //O(1)
    {
        v[i] = true;
        num_of_empty_lists++;
        list_of_clusters[temp_new_adj].Add(i);
        temp_new_adj++;
        //  result = (Distinctcolors[i.Value[j]].red << 16) + (Distinctcolors[i.Value[j]].green <<

    }
}
int remain_num_cluster = num_K - num_of_empty_lists;

int cont = 0;
for (int y = 0; y < remain_num_cluster; y++)
{

    if (cont < adj.Length)
    {
        while (adj[cont].Count == 0 || v[cont] == true)
        {
            cont++;
        }

        if (v[cont] == false && adj[cont].Count != 0)
        {
            list_of_clusters[y + temp_new_adj].Add(cont);
            v[cont] = true;
            foreach (var x in adj[cont])
            {
                if (v[x] == false)
                {
                    int index_x = x;
                    dfs(y + temp_new_adj, index_x);
                }
            }
        }
    }
    cont++;
}
```

```
void dfs(int i, int u)          //O(D)
{

    v[u] = true;                // O(1)
    list_of_clusters[i].Add(u);  //O(1)

    foreach (var x in adj[u])   //O(D)
    {
        if (v[x] == false && adj[x].Count != 0) // O(1)
        {
            dfs(i, x);
        }
    }
}
```

Average Function:

Details:

After storing the colors in the clusters we calculate the average of RGB of 2 colors and replace the colors, so we chose to implement the graph via adjacency list implemented as an array of list.

Code:

```
////////////////////////////////Average/////////////////////////////////////////
double sum_of_red;     //O(1)
double sum_of_green;     //O(1)
double sum_of_blue;     //O(1)
RGBPixel[] Guide = new RGBPixel[16777217];      //O(1)
int result;   //O(1)
int r;   //O(1)
int g;   //O(1)
int b;   //O(1)

for (int i = 0; i < list_of_clusters.Length; i++)   //worst case where is one clusters has all distinctcolors
{                                                   //or each color in one cluster then    O(D)
    sum_of_red = 0;
    sum_of_green = 0;
    sum_of_blue = 0;

    foreach (var x in list_of_clusters[i])
    {
        sum_of_red += distinct_color[x].red;
        sum_of_blue += distinct_color[x].blue;
        sum_of_green += distinct_color[x].green;

    }

    sum_of_red = sum_of_red / list_of_clusters[i].Count;            //O(1)
    sum_of_blue = sum_of_blue / list_of_clusters[i].Count;
    sum_of_green = sum_of_green / list_of_clusters[i].Count;

    RGBPixel obj = new RGBPixel(sum_of_red, sum_of_green, sum_of_blue);
    list_palette.Add(obj);
    foreach (var x in list_of_clusters[i])
    {
        r = distinct_color[x].red * 65536;                          //O(1)
        g = distinct_color[x].green * 256;
        b = distinct_color[x].blue;
        result = r + g + b;
        Guide[result] = obj;
    }

}
```

Analysis of the code:

Total: O(D)

Quantize the image:

Details:

Quantize the image by replacing the colors of each cluster by its representative color.

Code:

```
for (int h = 0; h < GetHeight(ImageMatrix); h++)                    //O(N)
{
    for (int w = 0; w < GetWidth(ImageMatrix); w++)                 //O(N)
    {
        result = ImageMatrix[h, w].red * 65536 + ImageMatrix[h, w].green * 256 + ImageMatrix[h, w].blue;
        ImageMatrix[h, w] = Guide[result];
    }
}                                                                   //O(N^2)

DisplayImage(ImageMatrix, PicBox);                                  //O(1)
```

Analysis of the code:

Total: O(N^2)

Bonus Function:

```
private static int Detect_K_number(List<link> List_OF_Link);
```

Details:

Grouping the distinct colors into clusters without knowing number of clusters by computing the standard deviation of the edges in MST and store it then remove which has the maximum one then repeat this steps until convergence.

Code:

```
1 reference
private static int Detect_K_number(List<link> List_OF_Link)                    //Total: O(E^2)
{
    double mean = 0, sum = 0, new_SV = double.MaxValue, old_SV = 0, max_sv = 0;    //O(1)
    int Num_K = 0, index_sv = -1, count;

    List<link> data = new List<link>(List_OF_Link);                            //O(1)
    Console.WriteLine("***************************");
    while (Math.Abs(new_SV - old_SV) > 0.0001)
    {
        Console.WriteLine("EQ=  "+ Math.Abs(new_SV - old_SV));
        Console.WriteLine("counter=  " +data.Count);
        Console.WriteLine("++++++++");
        if (data.Count > 0)                                                    //O(1)
        {
            old_SV = new_SV;
            foreach (link lk in data)                                          //O(E)
            {
                mean += lk.value;
            }
            mean = mean / data.Count;
            count = 0;
            index_sv = -1;
            max_sv = 0;
            foreach (link lk in data)                                          //O(E)
            {
                double term = (lk.value - mean);
                term = term * term;
                if (term > max_sv)                                             //O(1)
                {
                    max_sv = term;
                    index_sv = count;
                }
                count++;
                sum += term;
            }

            new_SV = sum / (data.Count-1);                                     //O(1)
            new_SV = Math.Sqrt(new_SV);
            bool res = Double.IsNaN(new_SV);                                   //O(1)
            if (res == true || new_SV == 0)                                    //O(1)
            {
                index_sv = 0;
            }
            data.RemoveAt(index_sv);                                           //O(E)

            mean = 0;
            sum = 0;
            Num_K++;
        }

    }
    Num_K--;
    return Num_K;
}
```

Analysis of the code:

Looping on the edges (O(E))

Total: O(E^2) where E is number of edges