T094
Image Quantization Documentation

1. Graph construction:

We start by initiating an array to hold all the colors, next we initiate an RGBPixel list to hold the distinct colors, then we loop over every element from the height then we call another function named Check() on each element.

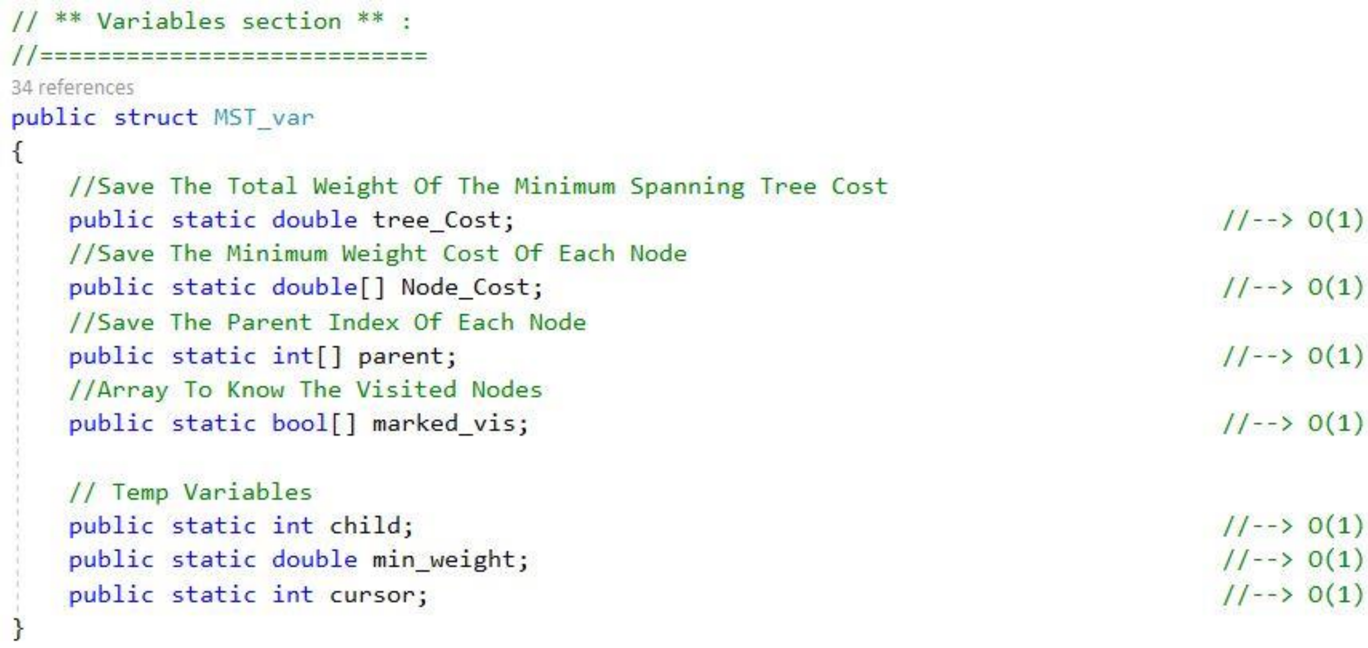
Check() function takes every element from the height, then loops over each element from the width of the image matrix giving a unique ID for every pixel of the image matrix, then check if this id is visited in the array named check, if not we map it as visited and add it to the distinct colors list.

```
public List<RGBPixel> Find_Distinct()
{
  check = new int[200000000];//------> O(1)
  List<RGBPixel> Distinct = new List<RGBPixel>();//------> O(1)
  for (int i = 0; i < ImageOperations.GetHeight(ImageMatrix); i++)//→O(n)
  {
     Check(check, ImageMatrix, i, Distinct);//------> O(n)
  }
  //------> O(n2) loop *body
  return Distinct ;//------> O(1)
}

public void Check(int[] check, RGBPixel[,] ImageMatrix, int i, List<RGBPixelDistinct)
 {
   int x = 65536;//------> O(1)
   int j = 0;//------> O(1)
   while (j < ImageOperations.GetWidth(ImageMatrix))  //------> O(n)*O(1) the whole loop * body
   {
     int g = ImageMatrix[i, j].green;//------> O(1)
     int r = ImageMatrix[i, j].red;//------> O(1)
     int b = ImageMatrix[i, j].blue;//------> O(1)
     r = g + b * x / 256 + r * x;//------> O(1)
     if (check[r] == 0) //------> O(1)
     {
      check[r] = Distinct.Count;//------> O(1)
      Distinct.Add(ImageMatrix[i, j]); //------> O(1)
     }
       j++;//------> O(1)+O(1)
    }
  }
```

2. Minimum spanning tree (MST)

In MST algorism we use prim algorism to find the minimum spanning tree but we don’t use priority queue we use arrays here a struct of variables we use:



```
// ** Variables section ** :
//===========================
34 references
public struct MST_var
{
    //Save The Total Weight Of The Minimum Spanning Tree Cost
    public static double tree_Cost;                                         //--> O(1)
    //Save The Minimum Weight Cost Of Each Node
    public static double[] Node_Cost;                                       //--> O(1)
    //Save The Parent Index Of Each Node
    public static int[] parent;                                             //--> O(1)
    //Array To Know The Visited Nodes
    public static bool[] marked_vis;                                        //--> O(1)

    // Temp Variables
    public static int child;                                                //--> O(1)
    public static double min_weight;                                        //--> O(1)
    public static int cursor;                                               //--> O(1)
}
```

```
// Function To Calculate The Minimum Spanning Tree :
        //=====================================================
        public double Build_Mst(ref List<KeyValuePair<KeyValuePair<int, int>, double>> edges, int Dcol_count,
List<RGBPixel> Col)
        {
            // ** Intialization Section ** :
            //==============================
            // MST_var Intialization
            MST_var.tree_Cost = 0;                                                       //--> O(1)
            MST_var.Node_Cost = new double[Dcol_count + 1];                                  //--> O(1)
            MST_var.parent = new int[Dcol_count + 1];                                       //--> O(1)
            MST_var.cursor = 1;                                                            //--> O(1)
            MST_var.marked_vis = new bool[Dcol_count + 1];                                //--> O(1)


            //Initialize The Minimum Weight Cost Of Each Node With 10^9
            for (int i = 1; i <= Dcol_count; i++)
            {
                MST_var.Node_Cost[i] = 1000000000;
            } //--> O(N) : where N is number of distinct colors

            MST_var.Node_Cost[1] = 0; //--> O(1)

            // MST Logic
            while (MST_var.cursor < Dcol_count) //--> O(Log(V)) : where V is the number of vertices
            {
                MST_var.marked_vis[MST_var.cursor] = true; //--> O(1)
                MST_var.min_weight = 1000000000; //--> O(1)
                MST_var.tree_Cost += MST_var.Node_Cost[MST_var.cursor]; //--> O(1)
                MST_var.child = 0; //--> O(1)

                for (int ch = 2; ch < Dcol_count; ch++)//--> O(E) : where E is the number of Edges
                {
                    // If this node not Visited yet
                    if (MST_var.marked_vis[ch] == false) //--> O(1)
                    {
                        //Calculate The Weight On The Edge Between The Current Vertix And its Children

                        double weight = ECl_Distance(Col[MST_var.cursor], Col[ch]); //--> O(1)

                        //Check If I Pushed The Same Vertix With A Smaller Cost
                        if (MST_var.Node_Cost[ch] > weight) //--> O(1)
                        {
                            //Updating The Weight Of The Child Node
                            MST_var.Node_Cost[ch] = weight; //--> O(1)
                            MST_var.parent[ch] = MST_var.cursor; //--> O(1)

                        }
                        if (MST_var.Node_Cost[ch] < MST_var.min_weight) //--> O(1)
                        {
                            MST_var.min_weight = MST_var.Node_Cost[ch]; //--> O(1)
                            MST_var.child = ch; //--> O(1)
                        }
                    }
                } //--> O(N) : where N is number of distinct colors

                if (MST_var.child == 0) break;

                edges.Add(new KeyValuePair<KeyValuePair<int, int>, double>(new KeyValuePair<int,
                int>(MST_var.cursor, MST_var.child), MST_var.min_weight));
                MST_var.cursor = MST_var.child;

            } //--> O(E Log(v))

            //Return The Total Weight Cost Of MST

            return MST_var.tree_Cost;
        }
```

3. Palette generation

```
public void calc(int j)
  {
      if (info.visited[j] == false)//------> O(1)
      {
          info.count = 0;//------> O(1)
          info.reds = 0;//------> O(1)
          info.greens = 0; //------> O(1)
          info.blues = 0;//------> O(1)
          Breadth_First_Search(j);
          info.reds /= info.count;//------> O(1)
          info.greens /= info.count;//------> O(1)
          info.blues /= info.count;//------> O(1)
          RGBPixel tmp = new RGBPixel();//------> O(1)
          tmp.red = (byte)info.reds;//------> O(1)
          tmp.green = (byte)info.greens;//------> O(1)
          tmp.blue = (byte)info.blues;//------> O(1)
          info.current++;//------> O(1)
          Clusters.Add(tmp);//------> O(1)
      }
  }
public List<int>[] clustr(List<int>[] adjList, List<KeyValuePair<KeyValuePair<int, int>, double>>
edges, int num_of_clusters, int K, int i)
  {

      while (num_of_clusters > K)//------> O(K)
      {
       KeyValuePair<int, int> Edge = new KeyValuePair<int, int>(edges[i].Key.Key,
       edges[i].Key.Value);//------> O(1)
          adjList[Edge.Key].Add(Edge.Value);//------> O(1)
          adjList[Edge.Value].Add(Edge.Key);//------> O(1)
          i++;//------> O(1)
          num_of_clusters--;//------> O(1)
      }
      return adjList;
  }
  public void Identify_Clusters(List<KeyValuePair<KeyValuePair<int, int>, double>> edges, int D, int K)
  {
      MergeSort(edges, 0, edges.Count - 1);
      adjList = new List<int>[D + 1];//------> O(1)
      info.visited = new bool[D + 1];//------> O(1)
      info.hold = new int[D + 1];//------> O(1)
      int num_of_clusters = D - 1, i = 0;//------> O(1)
      for (int j = 0; j < D; j++)
      {
          adjList[j] = new List<int>();
      }
      clustr(adjList, edges, num_of_clusters, K, i);
      info.current = 1;//------> O(1)
      Clusters.Add(new RGBPixel());
      for (int j = 1; j < D; j++)//------> O(D)
      {
          calc(j);
      }
      //==============================================================================================
      for (int h = 0; h < ImageOperations.GetHeight(ImageMatrix); h++)
      {
          // Replaces the pixels of the image matrix
          for (int j = 0; j < ImageOperations.GetWidth(ImageMatrix); j++)
          {
              int x = 65536;
              int r = ImageMatrix[h, j].red;
              int g = ImageMatrix[h, j].green;
              int b = ImageMatrix[h, j].blue;
              r = g + b * x / 256 + r * x;
              ImageMatrix[h, j] = Clusters[info.hold[Quantize.check[r]]];
          }
      }
      //================================================================================================
  }
```

Run Times:

SAMPLE TESTS

Image Name	# Distinct Colors	MST Sum	K	Time(S)	Time (MS)
Sample.Case1	5	730.5	3	0	0
Sample.Case2	3	322.6	2	0.015	15
Sample.Case3	2265	6106.2	500	0.078	78
Sample.Case4	69	1120.66	10	0.016	16
Sample.Case5	256	441.7	32	0.015	15

COMPLETE TESTS

Image Name	# Distinct Colors	MST Sum	K	Time(S)	Time (MS)
Small.Case1	8,708	11,785.1	192	1.031	1031
Small.Case2	10,265	19,888.8	2160	1.485	1485
Medium.Case1	27,410	40,616.4	1737	10.25	10250
Medium.Case2	20,041	44,831.7	**2284**	5.672	5672
Large.Case1	56,328	118,145.1	**3829**	44.485	44485
Large.Case2	54,223	80,957.2	25,666	39.75	39750