



**Ain Shams University**  
**Faculty of Computer and Information Science**  
**Scientific Computing department**

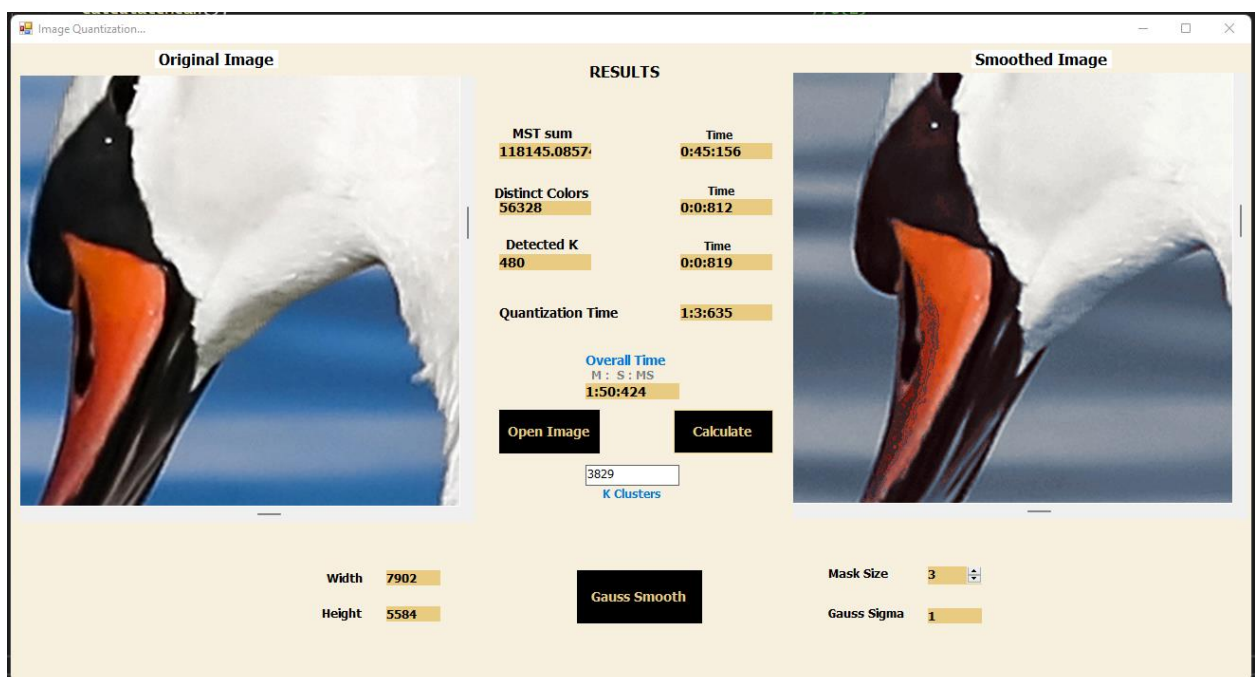
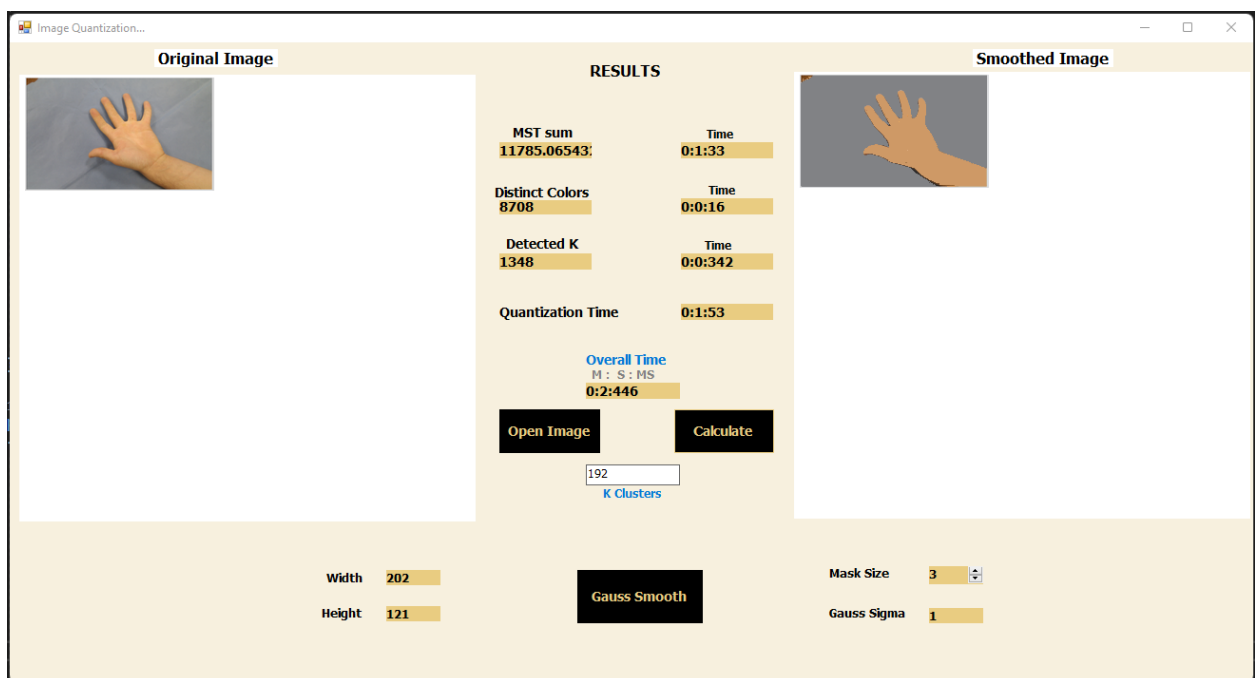
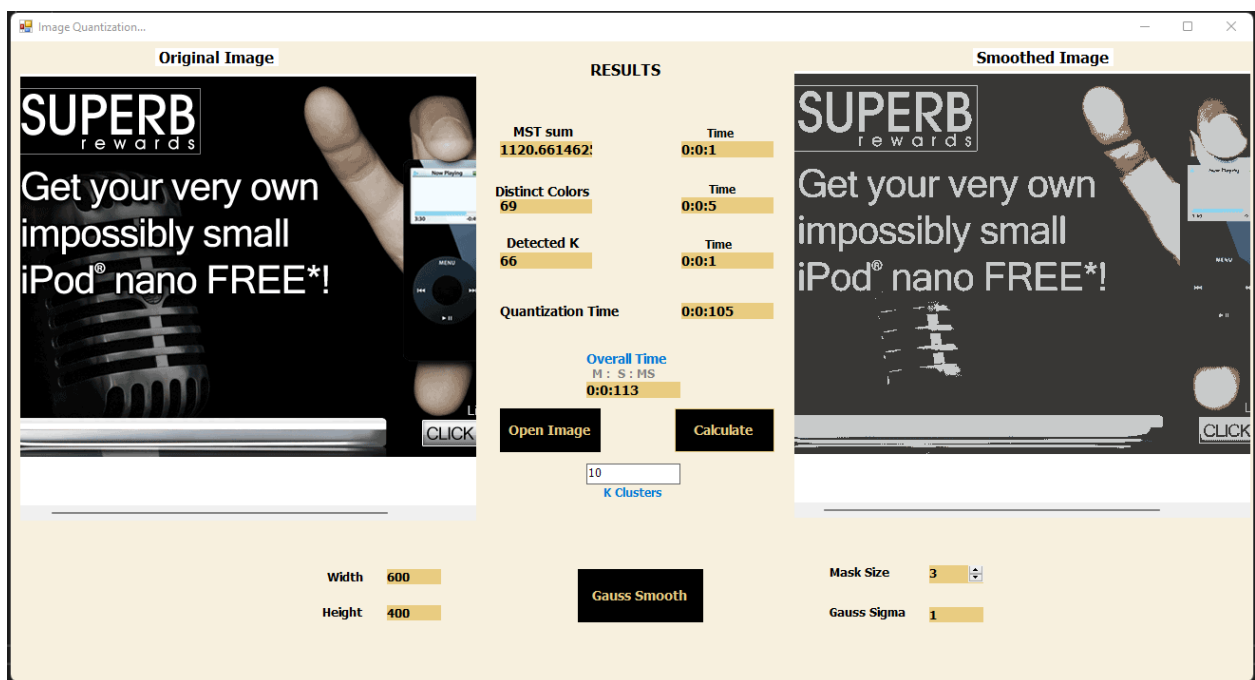
**Project Title**  
**Image Quantization**

**By Team 143**

<b>Name</b>	<b>ID</b>	<b>Section</b>
<b>Nour Mohamed Hussein Kamaly</b>	<b>20191700701</b>	<b>5</b>
<b>Nourhan Abdel-Karim Khalaf Abdel-Hafez</b>	<b>20191700716</b>	<b>5</b>
<b>Mohammed Nour-Elden Abbas Ismael</b>	<b>20191700583</b>	<b>4</b>
<b>Abdul-Rahman Sayed Ali Mohammed</b>	<b>20191700339</b>	<b>3</b>

**Under the supervision of**

**Dr. Ahmed Salah**  
**Computer Science Department,**  
**Faculty of computer and Information Science**  
**Ain Shams University**



## What is Image Quantization?

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.

An example of color quantization is depicted in the following Figure. First, a color palette is found by using clustering algorithm and then the original image values are replaced by their closest values in the palette.

11	13	15	17
13	25	27	19
15	27	21	21
17	19	21	23

11	17	23
----	----	----

 $\Rightarrow$ 

11	11	17	17
11	23	23	17
17	23	23	23
17	17	23	23

Figure 1.1: Example of quantization, where the original image is shown left, the color palette in the middle, and the quantized image in the right.



Figure 1.2: 24-bit parrots image quantized to the 16 colors.

## Main Steps

Color quantization consists of two main steps:

1. **Palette Generation:** A palette generation algorithm finds a smaller representative set of colors  $C = \{c_1, c_2, c_3, \dots, c_k\}$  from the  $D$  distinct colors.
2. **Quantization:** by mapping the original colors to the palette colors.

SO, we created five functions in this project to reduce the quantized image as little as possible from the original image and reduce the number of colors in a full-resolution digital color image (24 bits per pixel) to a smaller set of representative colors (**color palette**).

*Let's know each function, what it does and how.*

## Functions Description:

### ➤ **Get Distinct Colors.**

- **Name:** getDistinctColors.
- **input:** ImageMatrix.
- **output:** List of distinct RGB pixels.
- **Description:** Extract distinct color from image matrix.
- **Overall Complexity:**  $O(N^2)$

### ➤ **Minimum Spanning Tree.**

- **Name:** minimumSpanningTree.
- **input:** DistinctColors.
- **output:** Array of struct of MST vertices.
- **Description:** Construction Minimum Spanning Tree.
- **Overall complexity:**  $O(D^2)$ .

### ➤ **Clusters Construction.**

- **Name:** getKClusters.
- **Input:** array of struct of MST vertices, number of clusters, list of distinct colors.
- **Output:** dictionary composed of each distinct color and the number of clusters it belongs to.
- **Description:** adds colors with minimum edge weight to the same cluster.
- **Overall complexity:**  $O(K*D)$ .

### ➤ **Get Cluster's Representative Color.**

- **Name:** getClusterRepresentitive.
- **Input:** dictionary of clusters, list of distinct colors.
- **Output:** dictionary composed of the ID of the cluster and an array of size 3 representing its representative color in red, green, and blue.
- **Description:** loops over the distinct colors and calculates the mean of the colors belonging to the same cluster.
- **Overall complexity:**  $O(D)$ .

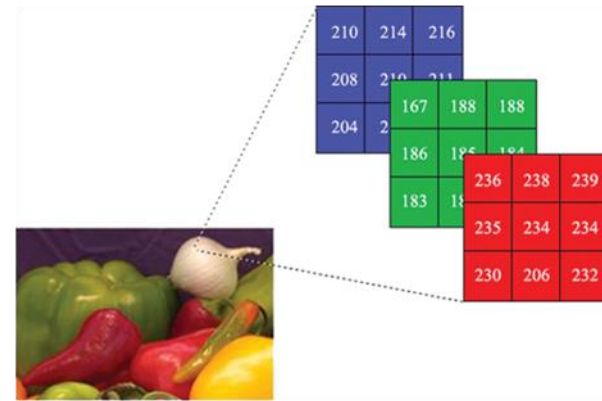
### ➤ **Quantization.**

- **Name:** Quantize.
- **input:** ImageMatrix, ClustersColors, Clusters, MapColor.
- **output:** returned Image matrix after reducing the number of colors in a full resolution.
- **Description:** map each color in the image matrix to his representative color in the palette.
- **Overall complexity:**  $O(N^2)$ .

## Intensity

The intensity of a pixel is expressed within a given range between a minimum and a maximum value [Inclusive], based on the color depth of the pixel.

True Color images have intensity from the darkest (0) and lightest (255) of three different color channels, **Red**, **Green**, and **Blue**. Each channel has a range from 0 to 255 as shown in Figure below. So we need  $8+8+8=24$  bits to represent 1 pixel color which means we have  $2^{24} = 16,777,216$  different colors.



## Image Quantization Algorithm

To Apply the Single-linkage Clustering algorithm on the Image Quantization Problem, we need to:

1. Find the distinct colors  $D = \{d_1, d_2, d_3 \dots d_m\}$  from the input image. Can be known from the image histogram.

## Get Distinct Colors Function:

```
1 public Dictionary<int, int> MapColor; //0(1)
2 public List<RGBPixel> getDistinctColors(RGBPixel[,] ImageMatrix)
3 {
4     int counter = 0; //0(1)
5     MapColor = new Dictionary<int, int>();
6     //3D Array to mark visited color from the ImageMatrix.
7     bool[, ,] visited_color = new bool[256, 256, 256]; //0(1)
8
9     RGBPixel color;
10
11     List<RGBPixel> dstincted_color = new List<RGBPixel>(); //0(1)
12
13     int Height = ImageMatrix.GetLength(0); //0(1)
14     int Width = ImageMatrix.GetLength(1); //0(1)
15
16     for (int i = 0; i < Height; i++) //0(N)
17     {
18         for (int j = 0; j < Width; j++) //0(N)
19         {
20             color = ImageMatrix[i, j]; //0(1)
21
22             if (visited_color[color.red, color.green, color.blue] == false) //0(1)
23             {
24                 visited_color[color.red, color.green, color.blue] = true; //0(1)
25                 dstincted_color.Add(color); //0(1)
26
27                 //Conver each RGB-Pixel to hexadecimal.
28                 string Rstring, Gstring, Bstring, hexColor; int intColor; //0(1)
29
30                 Rstring = color.red.ToString("X"); //0(1)
31                 if (Rstring.Length == 1) Rstring = "0" + Rstring; //0(1)
32
33                 Gstring = color.green.ToString("X"); //0(1)
34                 if (Gstring.Length == 1) Gstring = "0" + Gstring; //0(1)
35
36                 Bstring = color.blue.ToString("X"); //0(1)
37                 if (Bstring.Length == 1) Bstring = "0" + Bstring; //0(1)
38
39                 //Convert hexadecimal to integer.
40                 hexColor = Rstring + Gstring + Bstring; //0(1)
41                 intColor = Convert.ToInt32(hexColor, 16); //0(1)
42
43                 //Map each integer representation of distict color with its index.
44                 MapColor.Add(intColor, counter); //0(1)
45
46                 counter++; //0(1)
47             }
48         }
49     }
50     return dstincted_color; //0(1)
51     //Complexity of function: O(N*N) ==> O(N^2) Overall.
52 }
```

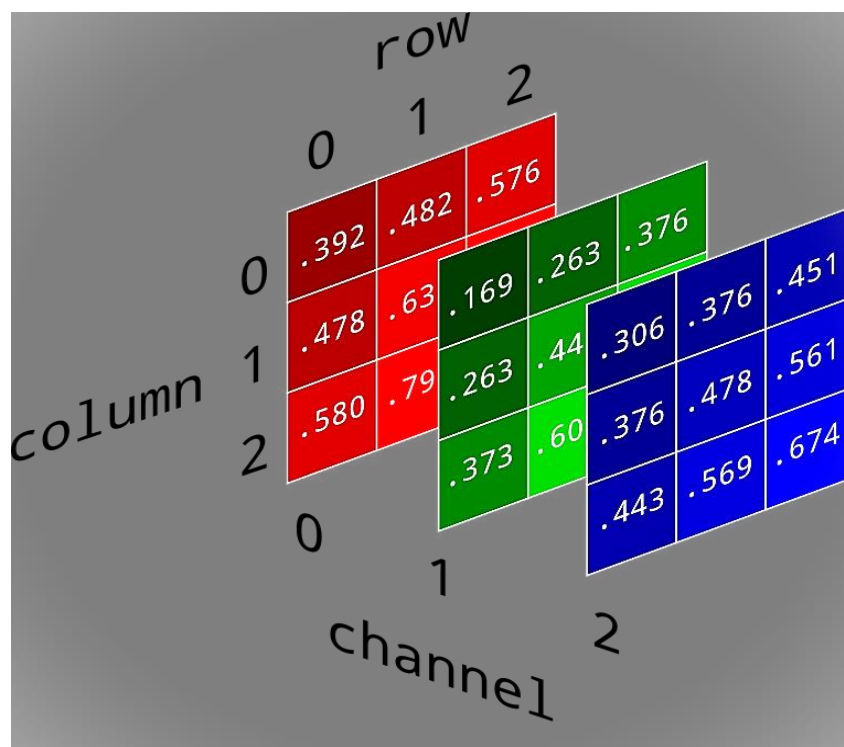
## Description:

We extract the distinct color from the image in this function 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 (*distinct colors*). This was returned from this function.

## Steps:

- We make a 3D array with size [256,256,256] as three-channel to map colors as RGB and marked each color as visited to prevent it from repeating colors. **Complexity:  $O(1)$ .**
- Loop (2D) over image matrix pixels and check whether each pixel is visited(marked) or not. **Complexity:  $O(\text{Height} * \text{Width}) \Rightarrow O(N^2)$ .**
- When we marked color as visited over iterations also convert color to (Hexadecimal) to using it after that as integer instead of Struct of (RGBPixels). **Complexity:  $O(1)$ .**
- We created a dictionary called (Mapcolor) to map each color in the distinct color list with its index in it because we are using it in quantization after that. **Complexity:  $O(1)$ .**

2. Construct



a fully-connected

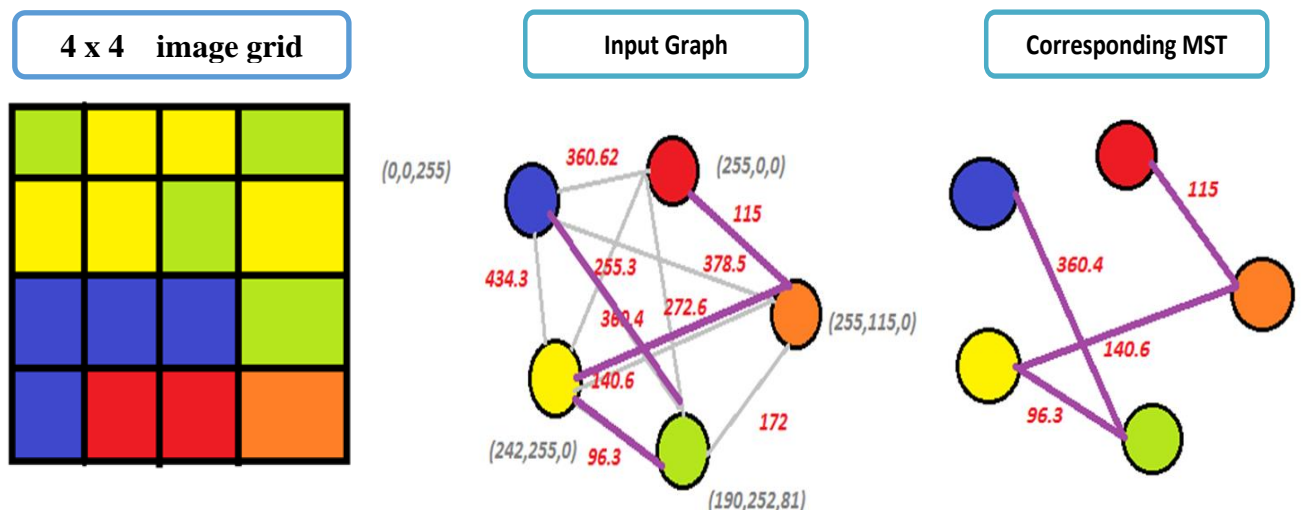


**undirected weighted graph  $G$  with**

- $D$  vertices (number of distinct colors).
- Each pair of vertices is connected by a single edge.
- Edge weight is set as the Euclidean Distance between the RGB values of the 2 vertices.

3. Construct a minimum-spanning-tree algorithm (a greedy algorithm in graph theory)


- **Input:** connected undirected weighted graph
- **Output:** a tree that keeps the graph connected with minimum total cost
- **Methodology:** treats the graph as a forest and each node is initially represented as a tree. A tree is connected to another only and only if it has the least cost among all available.
- **Conclusion:** the Minimum Spanning Tree is an implementation of single linkage clustering Strategy that repeats merging distinct points with minimal distances into a single cluster



- To get distance between two colors use *Euclidean Distance*:


The Euclidean Distance between TWO colors is defined as:

$$\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$



**(243,255,0)**

**96.3**



**(190,252,81)**

$$\sqrt{(243-190)^2 + (255-252)^2 + (0-81)^2} = 96.3$$

# Minimum Spanning Tree:

```
1 public double sum_mst; //O(1)
2 public Vertex[] minimumSpanningTree(List<RGBPixel> DistinctColors)
3 {
4     sum_mst = 0;
5     int vertexCount = DistinctColors.Count; //O(1)
6
7     Vertex[] vertices = new Vertex[vertexCount]; //O(1)
8
9     //Initialize struct of vertices with (Key -> Max value) - (parent -> -1) - (child -> index).
10    for (int i = 0; i < vertexCount; i++) //O(V)
11    {
12        vertices[i] = new Vertex() { Key = int.MaxValue, Parent = -1, child = i }; //O(1)
13    }
14
15    //Set random vertex key to zero.
16    vertices[0].Key = 0; //O(1)
17
18
19    double minimumEdge, weight; //O(1)
20    int current_vertex = 0; //O(1)
21
22
23    while (current_vertex < vertexCount) //O(V)
24    {
25        //mark vertex as visited to prevent make cycles.
26        vertices[current_vertex].Visited = true; //O(1)
27        minimumEdge = int.MaxValue; //O(1)
28
29        //start from the root which its parent = -1.
30        int child_vertex = 0; //O(1)
31
32        //Summation the MST edges.
33        sum_mst += vertices[current_vertex].Key; //O(1)
34
35        for (int i = 0; i < vertexCount; i++) //O(V)
36        {
37            if (vertices[i].Visited == false) //O(1)
38            {
39                //Calculate weight between current vertex and others vertices.
40                RGBPixel Currentvertex = DistinctColors[current_vertex]; //O(1)
41                RGBPixel Childvertex = DistinctColors[i]; //O(1)
42
43                weight = Math.Sqrt((Currentvertex.red - Childvertex.red) * (Currentvertex.red - Childvertex.red) +
44                    (Currentvertex.blue - Childvertex.blue) * (Currentvertex.blue - Childvertex.blue) +
45                    (Currentvertex.green - Childvertex.green) * (Currentvertex.green - Childvertex.green)); //O(1)
46
47                //Replace key if weight smaller than the key of child.
48                if (vertices[i].Key > weight) //O(1)
49                {
50                    vertices[i].Key = weight; //O(1)
51                    vertices[i].Parent = current_vertex; //O(1)
52                }
53
54                //Set minimumEdge if key smaller than the current value.
55                if (vertices[i].Key < minimumEdge) //O(1)
56                {
57                    minimumEdge = vertices[i].Key; //O(1)
58                    child_vertex = i; //O(1)
59                }
60            }
61        }
62
63        if (child_vertex == 0) //O(1)
64        {
65            break; //O(1)
66        }
67
68        //Set current vertex index with the child vertex index to find next minimum edge.
69        current_vertex = child_vertex; //O(1)
70    }
71    return vertices; //O(1)
72 }
73 //Complexity of function: O( V + (V*V) ) ==> O(V^2) Overall
74 }
75 }
```

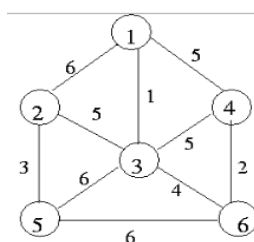
## Description:



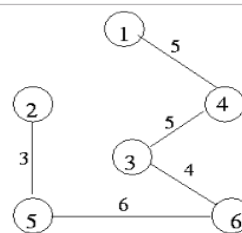
We should execute two functions here the first is construct fully-connected undirected weighted graph and the second is Minimum spanning tree but we compress them in one function with Complexity Upper bound:  $O(D^2)$  and Exact bound:  $\Theta(D \cdot E)$  to reduce Complexity of program overall.

## Steps:

- Create Struct of Vertex and loop over count of vertices to store all edges in it ( From(parent) : To(child)  $\Rightarrow$  Weight(key) ) with initialize some attributes of struct (Key  $\Rightarrow$  Max\_value), (parent  $\Rightarrow$  -1), (child  $\Rightarrow$  index). **Complexity:  $O(D^2)$**
- Initialize randomly any key of vertex with equal zero. **Complexity:  $O(1)$**
- Into the outer while loop we marked vertex as visited and set (minimum edge) with Max\_Value and enter the inner loop and search on minimum weight over edges and update (minimum edge) then start the next iteration in outer loop with child vertex was selected then repeat this cycle till finish all vertices. **Complexity:  $O(D^2) - \Theta(D \cdot E)$**

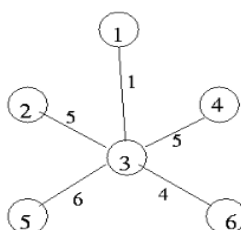


A connected graph.

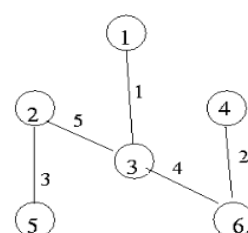


A spanning tree with cost = 23

### 4. Extract



Another spanning tree with cost 21



MST, cost = 15

the

desired number of clusters (K) with maximum distances between each other.

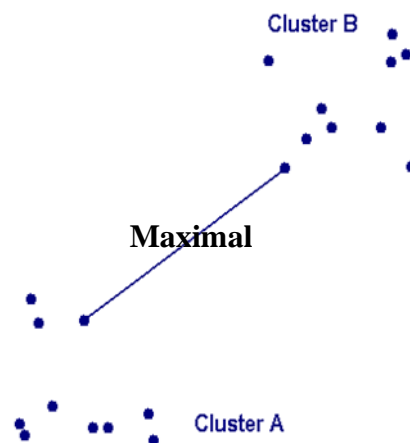
5. Find the representative color of each cluster.

## Clustering

Definition:

**Clustering** is the task of 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).

1. **Grouping of points** with minimal Distances between them in one cluster → means (is equivalent to) producing clusters with maximal spacing.



2. We can assume the number of groups/clusters.

Objective

Given a **distance measure** and a **desired number of clusters** → produce **K clusters** with maximal Spacing which means grouping distinct points with minimal distances into one cluster.

## Single-linkage Clustering

With single linkage method (also called nearest neighbor method), the distance between two clusters is the minimum distance between an observation in one cluster and an observation in the other cluster which is defined as the Euclidean Distance.

# Clusters Construction:

```
1 // key is the color number, value is the number of cluster it is belonging to
2 public Dictionary<int, int> Clusters;
3 //public static PriorityQueue<Vertex> SortedMST;
4 public FibonacciHeap<double, edges> SortedMST;
5 public List<edges> alledges;
6 public Dictionary<int, int> getKClusters(Vertex[] MST, int K, List<RGBPixel> DistinctColors)
7 {
8     SortedMST = new FibonacciHeap<double, edges>(); //O(1)
9     Clusters = new Dictionary<int, int>(); //O(1)
10    alledges = new List<edges>(MST.Length); //O(1)
11    int ctr; //O(1)
12    for (ctr = 0; ctr < MST.Length; ctr++) //O(E)
13    {
14        Clusters.Add(MST[ctr].child, ctr); //O(1)
15        alledges.Add(new edges() { source = MST[ctr].Parent, destination = MST[ctr].child, weight = MST[ctr].Key }); //O(1)
16    }
17    for (ctr = 0; ctr < alledges.Count; ctr++) //O(E)
18    {
19        SortedMST.Enqueue(alledges[ctr].weight, alledges[ctr]); //O(1)
20    }
21    edges SmallestDistance = SortedMST.Dequeue().Value; //O(log(E))
22    for (ctr = 0; ctr < (DistinctColors.Count - K); ctr++)
23    {
24        SmallestDistance = SortedMST.Dequeue().Value; //O(log(E))
25        Union(Clusters[SmallestDistance.source], Clusters[SmallestDistance.destination]); //O(D) (D is distinct colors)
26    }
27
28    return Clusters;
29 }
```

## Description:

We should be converting the minimum spanning tree to clusters and each cluster is with minimal Distances between its vertices and producing maximal spacing with other clusters.

## Steps:

- Create (alledges) to Sort minimum spanning tree with (Fibonacci heap) and create dictionary to store clusters in it as (key => vertex, value => index). **Complexity: O(1)**
- Loop over count of vertices and store (alledgs) to sort this list with Fibonacci heap to use it when construct clusters. **Complexity: O(E)**
- Loop over count of vertices to (enqueue) all edges in queue. **Complexity: O(log(E))**
- Remove from queue specific edge with minimum weight and put source and destination in one cluster with using (Union function). **Complexity: O(K\*D)**

## Union:

```

1 public void Union(int ReplaceBy, int Replaced)
2 {
3     int[] Keys = Clusters.Keys.ToArray();
4     for (int ctr = 0; ctr < Clusters.Count; ctr++) // O(D)
5     {
6         if (Clusters[Keys[ctr]] == Replaced) // O(1)
7         {
8             Clusters[Keys[ctr]] = ReplaceBy; // O(1)
9         }
10    }
11 }

```

## Description:

**Pick vertex and search about it then change its index and any vertex has same index to destination's index.**

## Get Cluster's Representative Color:

```

1 public Dictionary<int, int[]> getClusterRepresentative(Dictionary<int, int> Clusters, List<RGBPixel> DistinctColors)
2 {
3     Dictionary<int, int[]> ClustersColors = new Dictionary<int, int[]>(); // O(1)
4     Dictionary<int, int> NumOfElementsPerCluster = new Dictionary<int, int>(); // O(1)
5
6     //Cluster : key -> color number, value -> cluster number
7     //Cluster Colors: key -> cluster number, value -> representative color in red, green , blue
8     //Dinstinct colors : list of RGBPixels that can be accessed by index
9
10    foreach (var ClusterNumber in Clusters) // O(D)
11    {
12        if (!ClustersColors.ContainsKey(ClusterNumber.Value)) // O(1)
13        {
14            ClustersColors.Add(ClusterNumber.Value, new int[3]); // O(1)
15            NumOfElementsPerCluster.Add(ClusterNumber.Value, 0); // O(1)
16        }
17    }
18    foreach (var cluster in Clusters) // O(D)
19    {
20        int ColorNumber = cluster.Key; // O(1)
21        ClustersColors[cluster.Value][0] += DistinctColors[ColorNumber].red; // O(1)
22        ClustersColors[cluster.Value][1] += DistinctColors[ColorNumber].green; // O(1)
23        ClustersColors[cluster.Value][2] += DistinctColors[ColorNumber].blue; // O(1)
24        NumOfElementsPerCluster[cluster.Value]++;
25    }
26    foreach (var cluster in ClustersColors) // O(K)
27    {
28        ClustersColors[cluster.Key][0] /= NumOfElementsPerCluster[cluster.Key];
29        ClustersColors[cluster.Key][1] /= NumOfElementsPerCluster[cluster.Key];
30        ClustersColors[cluster.Key][2] /= NumOfElementsPerCluster[cluster.Key];
31    }
32    //Overall complixity of function: O(D+D+K) ==> O(D)
33    return ClustersColors;
34 }

```

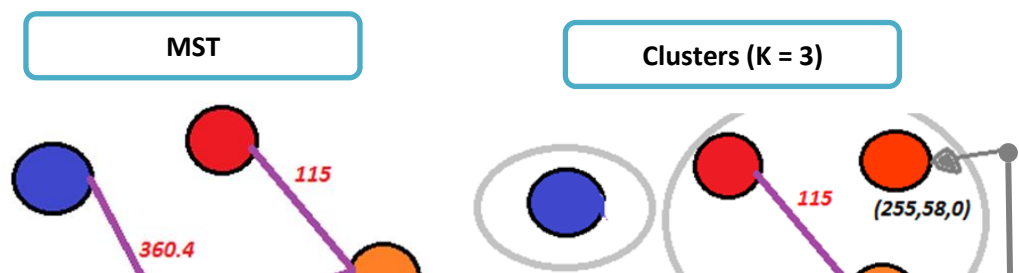
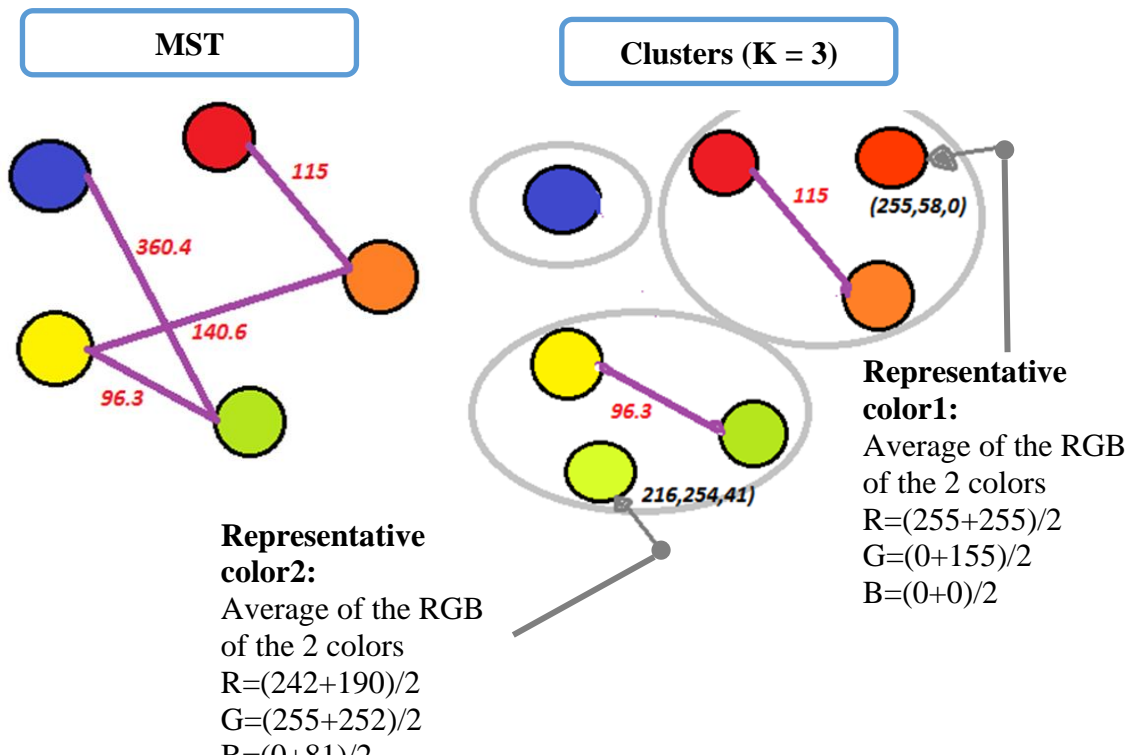
## Description:

**We compute the average of component RGB (Red, Blue, Green) for each cluster and each cluster has one representative color.**

- Cluster: key => color number, value => cluster number.
- Cluster Colors: key => cluster number, value => representative color in (red, green, blue)
- Distinct colors: list of RGBPixels that can be accessed by index.

## Steps:

- Loop over count of vertices and each vertex then compute component of RGB (red, blue, green) cluster and add each color to its channel. **Complexity:  $O(D)$**
- Loop over count of clusters to compute the average representative color. **Complexity:  $O(K)$**



# Quantization:

```
1 public RGBPixel[,] Quantize(RGBPixel[,] ImageMatrix, Dictionary<int, int[]> ClustersColors, Dictionary<int, int> Clusters, Dictionary<int, int> MapColor)
2 {
3     RGBPixel color;
4     int Height = ImageMatrix.GetLength(0);           //o(1)
5     int Width = ImageMatrix.GetLength(1);           //o(1)
6
7     for (int i = 0; i < Height; i++)                //o(N)
8     {
9         for (int j = 0; j < Width; j++)              //o(N)
10        {
11            color = ImageMatrix[i, j];                //o(1)
12
13            string Rstring, Gstring, Bstring, hexColor; int intColor; //o(1)
14            Rstring = color.red.ToString("X");         //o(1)
15            if (Rstring.Length == 1) Rstring = "0" + Rstring; //o(1)
16            Gstring = color.green.ToString("X");       //o(1)
17            if (Gstring.Length == 1) Gstring = "0" + Gstring; //o(1)
18            Bstring = color.blue.ToString("X");        //o(1)
19            if (Bstring.Length == 1) Bstring = "0" + Bstring; //o(1)
20
21            hexColor = Rstring + Gstring + Bstring;    //o(1)
22            intColor = Convert.ToInt32(hexColor, 16); //o(1)
23
24            int colorIndex = MapColor[intColor];       //o(1)
25            int ClusterNumber = Clusters[colorIndex]; //o(1)
26
27            ImageMatrix[i, j].red = (byte)ClustersColors[ClusterNumber][0]; //o(1)
28            ImageMatrix[i, j].green = (byte)ClustersColors[ClusterNumber][1]; //o(1)
29            ImageMatrix[i, j].blue = (byte)ClustersColors[ClusterNumber][2]; //o(1)
30
31        }
32    }
33
34    return ImageMatrix;
35
36    //Total Complexity o(N^2)
37 }
```

## Description:

We replace each pixel in image matrix with representative color which exist in the palette of color.

## Steps:

- Loop (2D) over image matrix pixels and replace each pixel with representative color. **Complexity:  $O(\text{Height} * \text{Width}) \Rightarrow O(N^2)$ .**
- When we loop over image also convert color to (Hexadecimal) to map with it in distinct color list to know original color replace it with representative color. **Complexity:  $O(1)$ .**

## Bonus:

## Automatically Detect Clusters:



### ➤ Calculate Mean:

- Name: calculateMean.
- input: alledges.
- output: Calculate mean.
- Description: Calculate mean of all edges.
- Overall Complexity:  $O(E)$

### ➤ Calculate Standard deviation:

- Name: calculateStandardDeviation.
- input: alledges.
- output: Array of struct of MST vertices.
- Description: Calculate Standard Deviation of all edges.
- Overall Complexity:  $O(E)$

### ➤ K-Clusters Detection:

- Name: KClustersDetection.
- input: Mean, Standard deviation of all edges.
- output: Number of detected clusters.
- Description: Detect clusters.
- Overall Complexity:  $O(E^2)$

## Description:

Clustering the distinct color points into disjoint groups without knowing the predefined number of clusters using the following criteria:

1. Computes the standard deviation of the edges in minimum spanning tree and store it
2. Choose an edge that leads to max standard deviation reduction once it is removed to obtain a set of two disjoint sub-trees.
3. Compute standard deviation reduction in current iteration.
4. Repeat steps (2) and (3) until convergence → the difference between standard deviation reduction in current iteration and previous iteration is very small  $< 0.0001$

**Complexity:  $O(N^2)$**

## Automatically Detect Clusters:

```

1 public List<edges> edges;
2 public double mean = 0;
3 public double standardDeviation = 0;
4 public double max = double.MinValue;
5 public int MaxIndex;
6 public int k;
7 public double previous = 0;
8
9 /// <summary>
10 /// Detection Number of clusters Automatically.
11 /// </summary>
12 /// <function name="calculateMean">calculate mean of all edges</function>
13 /// <function name="calculateStandardDeviation">calculate standard division of all edges</function>
14 /// <function name="KClustersDetection">Calculate number of K</function>
15 /// <param name="alldges"> All edges of MST</param>
16 /// <returns>Number of detected clusters</returns>
17
18 public void initializer(List<edges> alldges)
19 {
20     edges = alldges; //O(1)
21     k = 0; //O(1)
22 }
23 public void calculateMean()
24 {
25
26     double sum = 0; //O(1)
27
28     for (int i = 0; i < edges.Count; i++) //O(E)
29     {
30         sum += edges[i].weight; //O(1)
31     }
32
33     mean = sum / edges.Count; //O(1)
34     //Complexity of function: O(E+1) ==> O(E) Overall.
35 }
36
37 public void calculateStandardDeviation()
38 {
39     calculateMean(); //O(E)
40
41     double sum = 0; //O(1)
42     for (int i = 0; i < edges.Count; i++) //O(E)
43     {
44         if ((edges[i].weight - mean) * (edges[i].weight - mean) > max) //O(1)
45         {
46             max = (edges[i].weight - mean) * (edges[i].weight - mean); //O(1)
47             MaxIndex = i; //O(1)
48         }
49
50         sum += ((edges[i].weight - mean) * (edges[i].weight - mean)); //O(1)
51     }
52
53     max = double.MinValue; //O(1)
54     standardDeviation = sum / (edges.Count - 1); //O(1)
55     standardDeviation = Math.Sqrt(standardDeviation); //O(1)
56     //Complexity of function: O(E + 1) ==> O(E) Overall.
57 }
58
59 public int KClustersDetection()
60 {
61
62     calculateStandardDeviation(); //O(E)
63
64     while (Math.Abs(standardDeviation - previous) >= 0.0001) //O(E)
65     {
66         edges.RemoveAt(MaxIndex); //O(1)
67         previous = standardDeviation; //O(1)
68         calculateStandardDeviation(); //O(E)
69         k++; //O(1)
70     }
71     return k; //O(1)
72     //Complexity of function: O( E + E + E * ( E + E ) ) ==> O(E^2) Overall.
73 }
74 }
75
76 return ImageMatrix;
77 //Total Complexity o(N^2)
78 }

```