2021
2022

TEAM ID: T077

Image Quantization

السكشن	القسم	رقم الجلوس	الاسم
5	SC	20191700793	يوسف نادر ميشيل صبحي
6	CS	20191700793	يوسف ناصر صابر بيلاطس
4	SC	20191700460	كيرلس نبيل منير فهمي

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.

Graph

Why graph?

The graph is an essential input for constructing the MST. Constructing a graph will be in two steps first one is to get the unique pixel from the image in "Graph" class. Second step is to calculate the weight (distance) between each unique pixel. This step is exhaustive process as it will take time and storage also it will take time to index it in the process of MST, so we decide to not store it and only calculate the weight (distance) that we need in MST.

Graph construction

Graph will be constructed in "Graph" class. It consists of

- public int distinctColors
- public RGBPixel [,] ImageMatrix
- public RGBPixel [] UniqueColors
- public Graph (RGBPixel [,] ImageMatrix)
- private void GetUniqueColors ()

Public int distinctColors

An integer variable which is initialized with zero. It refers to the number of distinct colors in the image.

public RGBPixel [,] ImageMatrix

Array 2D that stores the image pixels as RGBPixel type.

public RGBPixel [] UniqueColors

Array 2D that will be used to store only the unique pixels from ImageMatrix array.

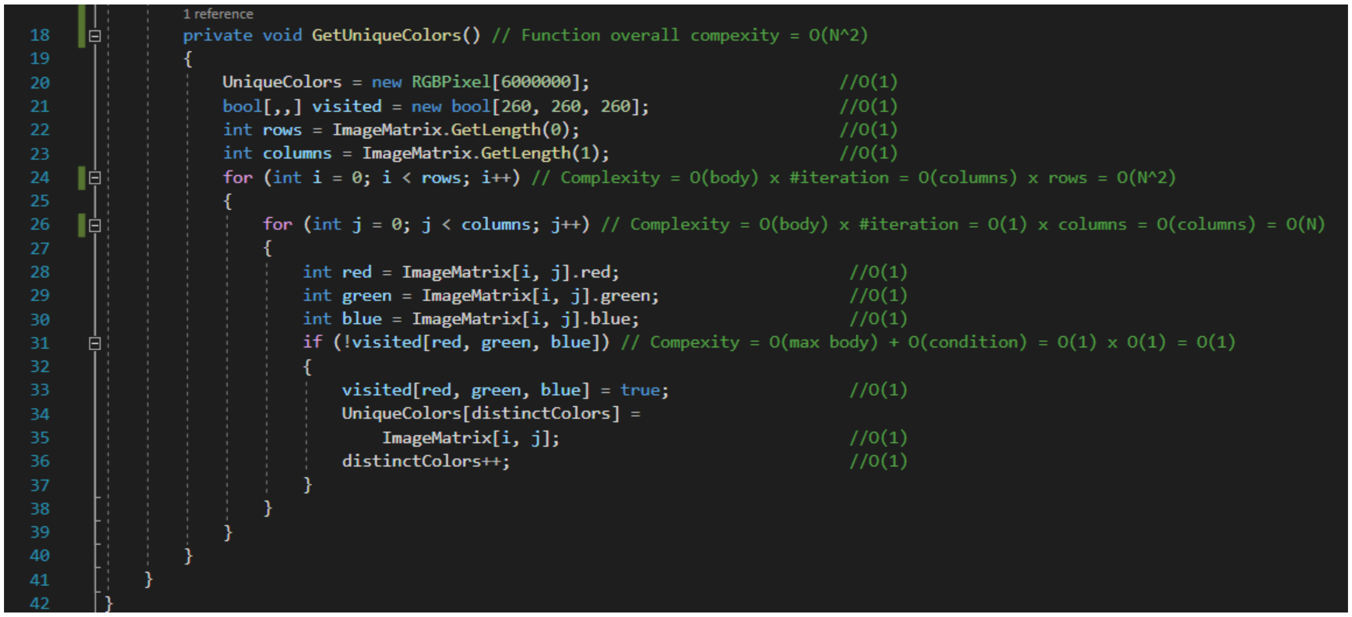
public Graph (RGBPixel [,] ImageMatrix)

A constructor that is used to initialize the ImageMatrix with the one in the parameter list of the constructor.

Calling "GetUniqueColors ()" function is the second usage for the constructor which get fill UniqueColors matrix once the user instantiate an object from the class to avoid using empty matrix.

Private void GetUniqueColors ()

The function aims to find the unique pixel from the image by initializing an 3D Boolean array which is initially false and loop on the image pixels and mark the pixel not visited with true.



```
1 reference
private void GetUniqueColors() // Function overall compexity = O(N^2)
{
    UniqueColors = new RGBPixel[6000000];                    //O(1)
    bool[,,] visited = new bool[260, 260, 260];              //O(1)
    int rows = ImageMatrix.GetLength(0);                     //O(1)
    int columns = ImageMatrix.GetLength(1);                  //O(1)
    for (int i = 0; i < rows; i++) // Complexity = O(body) x #iteration = O(columns) x rows = O(N^2)
    {
        for (int j = 0; j < columns; j++) // Complexity = O(body) x #iteration = O(1) x columns = O(columns) = O(N)
        {
            int red = ImageMatrix[i, j].red;                 //O(1)
            int green = ImageMatrix[i, j].green;             //O(1)
            int blue = ImageMatrix[i, j].blue;               //O(1)
            if (!visited[red, green, blue]) // Compexity = O(max body) + O(condition) = O(1) x O(1) = O(1)
            {
                visited[red, green, blue] = true;            //O(1)
                UniqueColors[distinctColors] =
                    ImageMatrix[i, j];                       //O(1)
                distinctColors++;                            //O(1)
            }
        }
    }
}
    }
}
```

Figure 1: GetUniqueColors()

Graph complexity

The complexity of Graph class related to GetUniqueColors () function. So, the overall complexity $O(N^2)$

Minimum spanning tree

Why MST?

Minimum spanning tree is the way that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Achieving this by using Prim algorithm.

Prim construction

Prim algorithm is used to find the MST and it produce parent array, value array and minimumSpanningTreeCost

MST construction will be in "Prim" class, and it consists of

- private readonly Graph
- public int [] parent
- public double [] values
- public Prim (Graph graph)
- private int selectMinimumVertex (double [] values, bool [] setMST, int numberOfVertex)
- public double MST ()

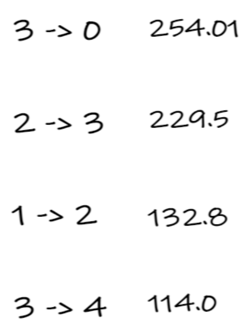
private readonly Graph graph

MST will use this variable to access the distinct color array for calculating the size of distinct colors and calculating the distances as this is the second step in calculating the map this way optimize the space and time complexity.

public int [] parent

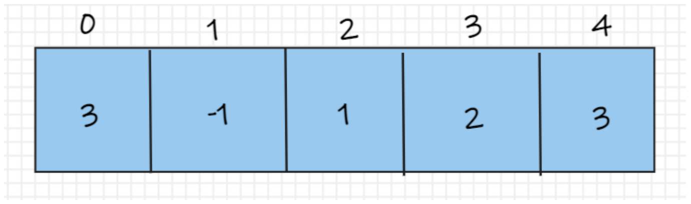
This array will store the parent of each vertex as the value will be the source and the index will be destination.

Example: In sample test case one



3 -> 0	254.01
2 -> 3	229.5
1 -> 2	132.8
3 -> 4	114.0

Figure 2: Graph

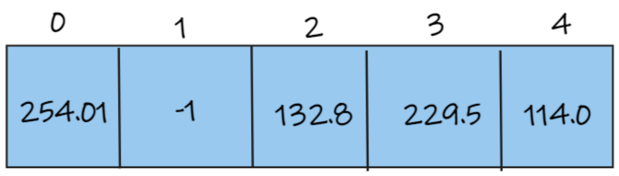


0	1	2	3	4
3	-1	1	2	3

Figure 3: parent array

public double [] values

This array will store the value (weight) of each edge mapped to parent array



0	1	2	3	4
254.01	-1	132.8	229.5	114.0

Figure 4: value array

public Prim (Graph graph)

A constructor that initializes the graph to prepare it for MST to use it.

private int selectMinimumVertex (double [] values, bool [] setMST, int numberOfVertex)

This function helps the MST to choose what vertex to visit next as it iterates on all vertices. This choice is based on if statement as it should be not visited, and its value is the smallest in the graph.

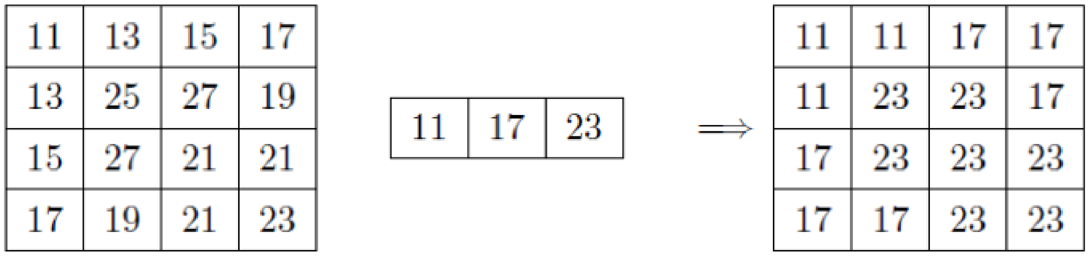
Finally, it returns integer number that indicates the index of the minimum weighted vertex

public double MST ()

The main objective of this function is to calculate the minimumSpanningTreeCost and update value and parent arrays by the end of the function to form an MST.

The function starts with initializing the arrays with the default values, then iterates on (V – 1) "Number of edges" then use selectMinimumVertex () function to choose the minimum weighted vertex and start to relax the graph and update each iteration.

In graph $G\ (V\ , E\ )$, where V is the number of vertices and E is the number of edges so when we extract the tree $G'\ (V'\ , E'\ )$, where $V' = V$, $E' \epsilon E$, $E' = V - 1$



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

11	17	23

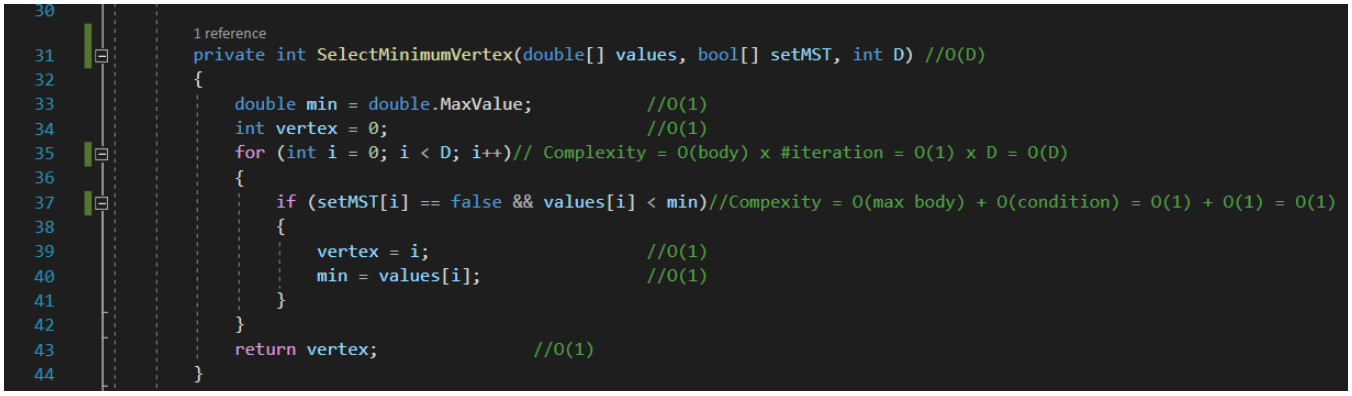
$\Longrightarrow$

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

Figure 5: Example 1 for color palette

MST Complexity

selectMinimumVertrc Complexity

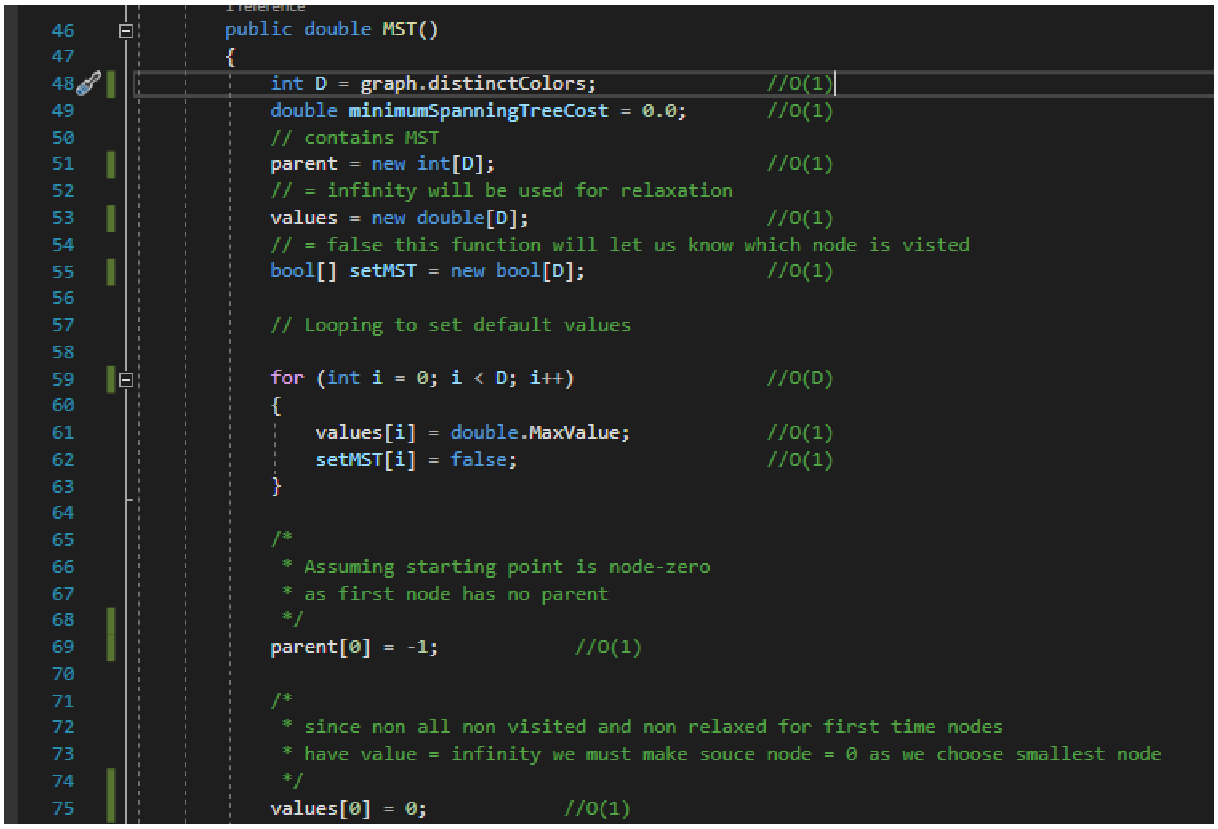


```
1 reference
private int SelectMinimumVertex(double[] values, bool[] setMST, int D) //O(D)
{
    double min = double.MaxValue;           //O(1)
    int vertex = 0;                         //O(1)
    for (int i = 0; i < D; i++)// Complexity = O(body) x #iteration = O(1) x D = O(D)
    {
        if (setMST[i] == false && values[i] < min)//Compexity = O(max body) + O(condition) = O(1) + O(1) = O(1)
        {
            vertex = i;                     //O(1)
            min = values[i];                //O(1)
        }
    }
    return vertex;                //O(1)
}
```

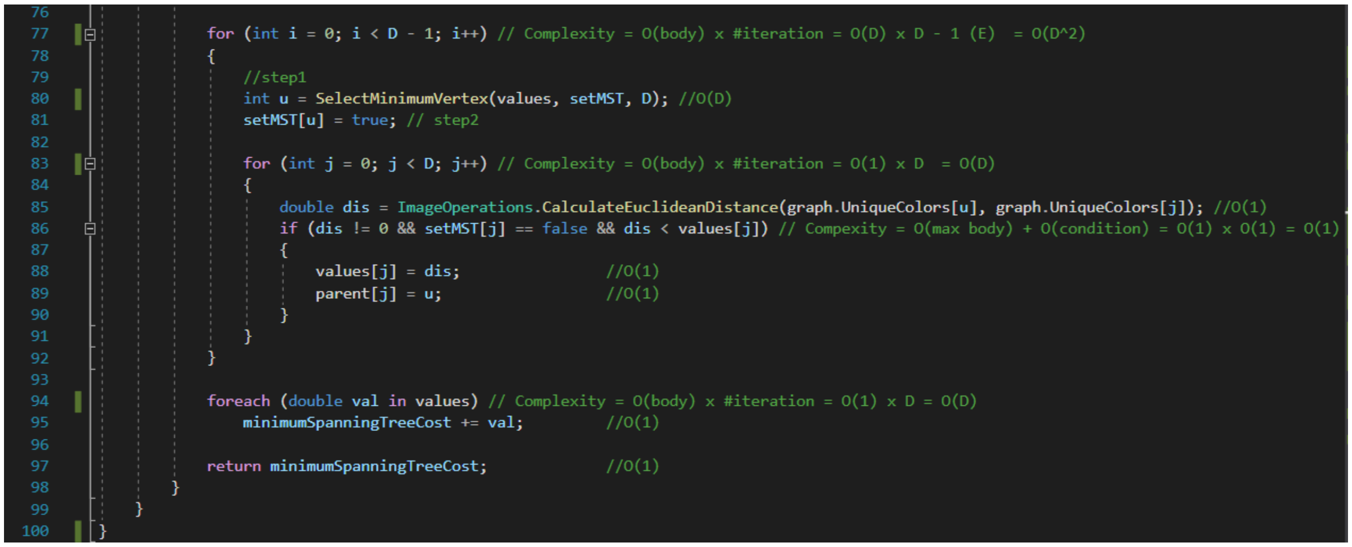
Figure 6: SelectMinimumVertex function

The function complexity depends on the number of distinct colors (V), Complexity $O(D)$

MST Complexity



```
        1 reference
        public double MST()
        {
            int D = graph.distinctColors;                 //O(1)
            double minimumSpanningTreeCost = 0.0;         //O(1)
            // contains MST
            parent = new int[D];                          //O(1)
            // = infinity will be used for relaxation
            values = new double[D];                       //O(1)
            // = false this function will let us know which node is visted
            bool[] setMST = new bool[D];                  //O(1)

            // Looping to set default values

            for (int i = 0; i < D; i++)                   //O(D)
            {
                values[i] = double.MaxValue;              //O(1)
                setMST[i] = false;                        //O(1)
            }

            /*
             * Assuming starting point is node-zero
             * as first node has no parent
             */
            parent[0] = -1;             //O(1)

            /*
             * since non all non visited and non relaxed for first time nodes
             * have value = infinity we must make souce node = 0 as we choose smallest node
             */
            values[0] = 0;             //O(1)
```



```

            for (int i = 0; i < D - 1; i++) // Complexity = O(body) x #iteration = O(D) x D - 1 (E)  = O(D^2)
            {
                //step1
                int u = SelectMinimumVertex(values, setMST, D); //O(D)
                setMST[u] = true; // step2

                for (int j = 0; j < D; j++) // Complexity = O(body) x #iteration = O(1) x D  = O(D)
                {
                    double dis = ImageOperations.CalculateEuclideanDistance(graph.UniqueColors[u], graph.UniqueColors[j]); //O(1)
                    if (dis != 0 && setMST[j] == false && dis < values[j]) // Compexity = O(max body) + O(condition) = O(1) x O(1) = O(1)
                    {
                        values[j] = dis;                //O(1)
                        parent[j] = u;                  //O(1)
                    }
                }
            }

            foreach (double val in values) // Complexity = O(body) x #iteration = O(1) x D = O(D)
                minimumSpanningTreeCost += val;         //O(1)

            return minimumSpanningTreeCost;             //O(1)
        }
    }
}
```

The function complexity depends on the nested loop that iterates on the edges then search for the minimum weighted node for all vertices Complexity = $O(D^2)$

Palette generation

Why palette?

As quantization is process of reduction number of colors in the image so we should know what colors to use and what colors to replace. This will happen by the helping of the color palette which mapped the similar color to their representative in the color palette.

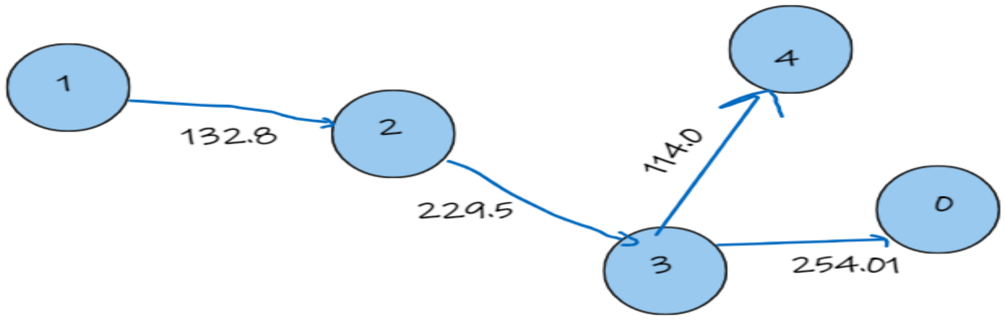


Figure 7: MST

Palette construction

"Cluster" class will be used to construct the palette. Process of constructing is based on clustering algorithm which aims to map each vertex (pixel or color) to a cluster which all members of that cluster have comparable properties then choose a representative for each cluster and put it in color palette finally replace each vertex in the image to its representative from the color palette.

The algorithm assumes that all vertices are initially in one cluster then removing (k – 1) most expensive weight from the connected MST and then apply BFS algorithm on the vertices to form k different trees each tree node is begin from the point of removal

- int numberOfClusters
- Graph graph
- Prim prim
- public Cluster (Prim prim, Graph graph, int K)
- private RGBPixel [,] GeneratePallete ()
- public RGBPixel [,] GetQuantizedImage ()

int numberOfClusters

Number of desire clusters. Taken as an input from the "MainForm".

Graph graph

Graph variable allows us to use UniqueColors array to map the colors.

Prim prim

Prim variable allows us to parent and value arrays that stores the MST.

public Cluster (Prim prim, Graph graph, int K)

A constructor that initializes graph, prim, and number of clusters once the object is instantiated to avoid accessing any empty items.

private RGBPixel [,] GeneratePallete ()

"GeneratePalette" function is responsible for producing "K" clusters as it takes the "MST" from the prim and remove the most (K – 1) expensive weighted edges and then make sub trees.

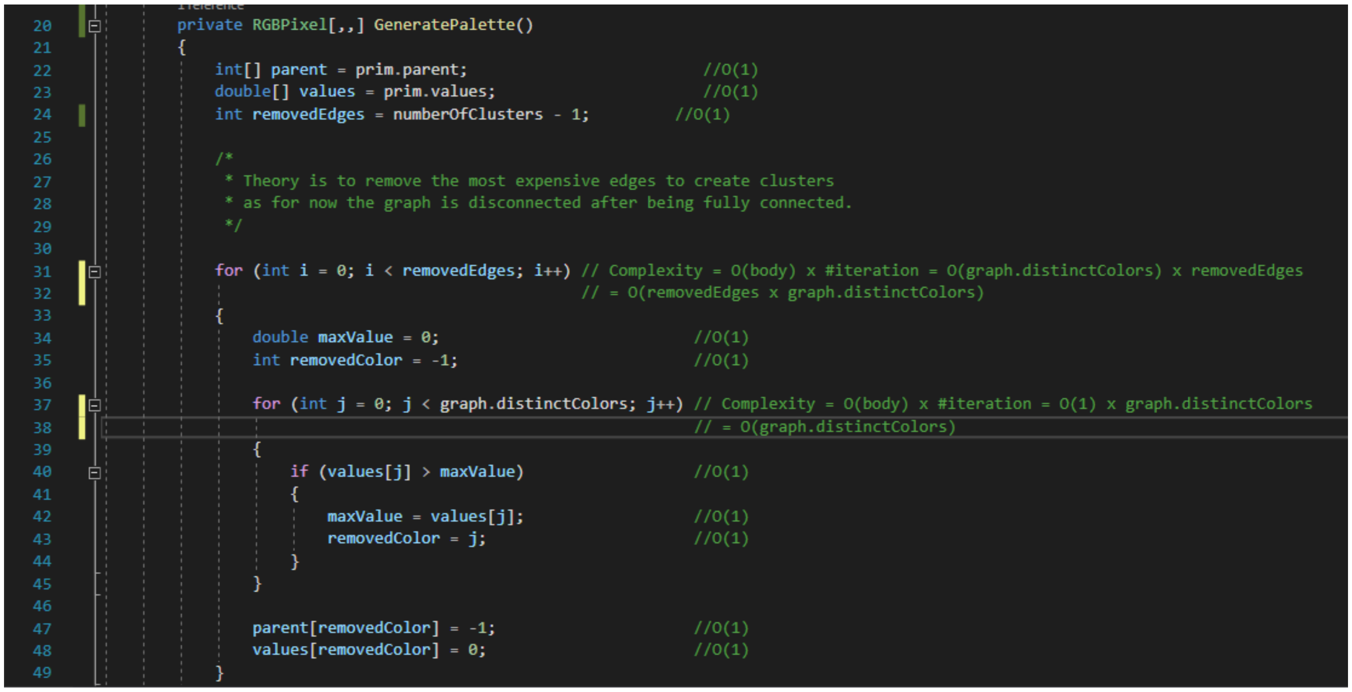
public RGBPixel [,] GetQuantizedImage ()

" GetQuantizedImage" function to map each pixel in "ImageMatrix" to its representative color in map (color palette).

Return the ImageMatrix so that we can show it in the form.

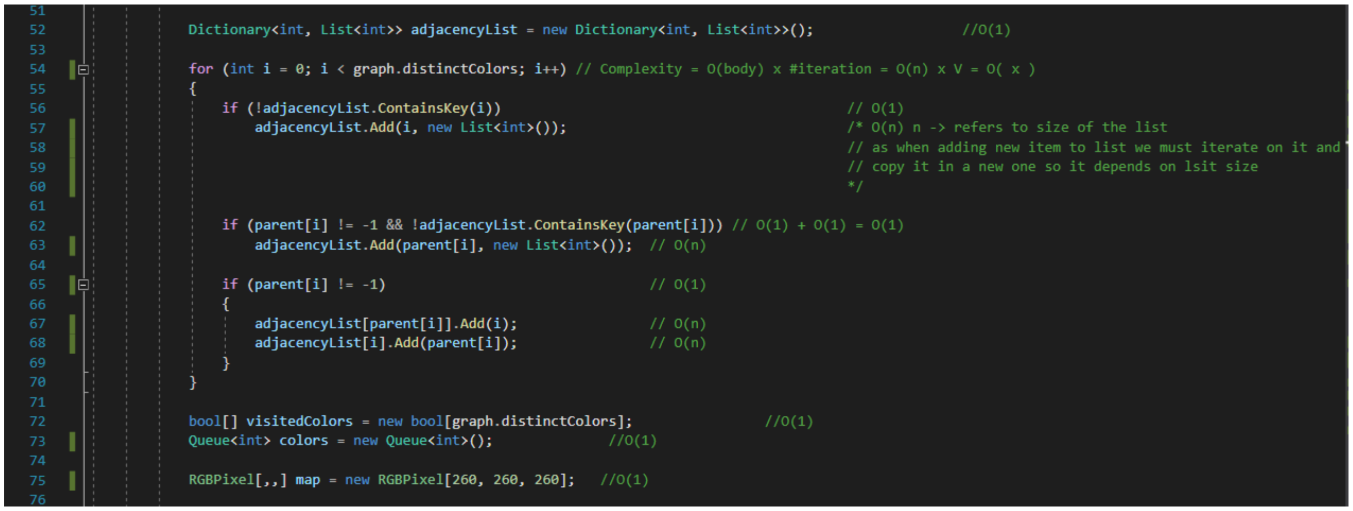
Palette Complexity

GeneratePalette

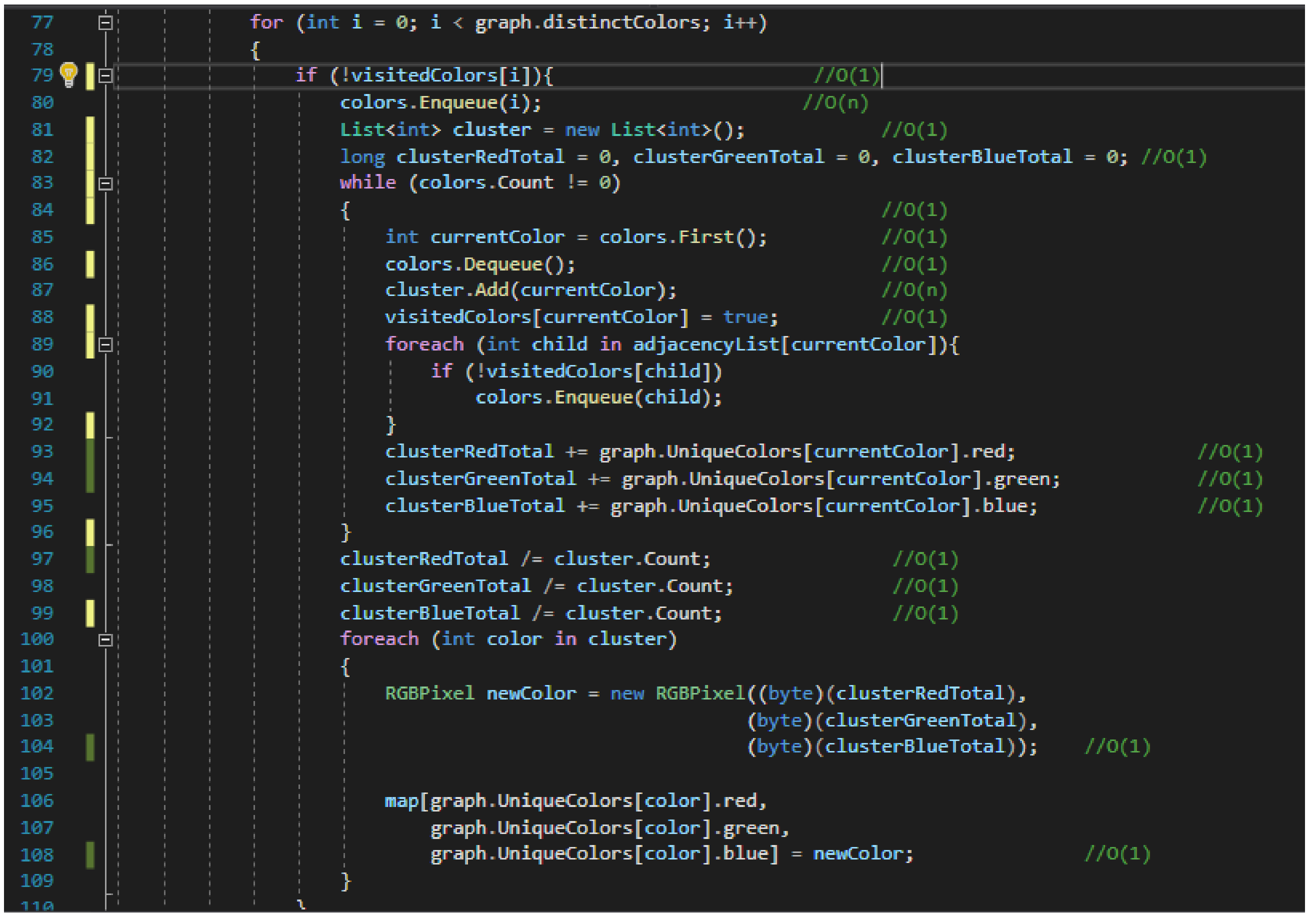


```
private RGBPixel[,,] GeneratePalette()
{
    int[] parent = prim.parent;                          //O(1)
    double[] values = prim.values;                       //O(1)
    int removedEdges = numberOfClusters - 1;          //O(1)

    /*
     * Theory is to remove the most expensive edges to create clusters
     * as for now the graph is disconnected after being fully connected.
     */

    for (int i = 0; i < removedEdges; i++) // Complexity = O(body) x #iteration = O(graph.distinctColors) x removedEdges
                                           // = O(removedEdges x graph.distinctColors)
    {
        double maxValue = 0;                            //O(1)
        int removedColor = -1;                          //O(1)

        for (int j = 0; j < graph.distinctColors; j++) // Complexity = O(body) x #iteration = O(1) x graph.distinctColors
                                                        // = O(graph.distinctColors)
        {
            if (values[j] > maxValue)                   //O(1)
            {
                maxValue = values[j];                   //O(1)
                removedColor = j;                       //O(1)
            }
        }

        parent[removedColor] = -1;                      //O(1)
        values[removedColor] = 0;                       //O(1)
    }
```

The function complexity depends on

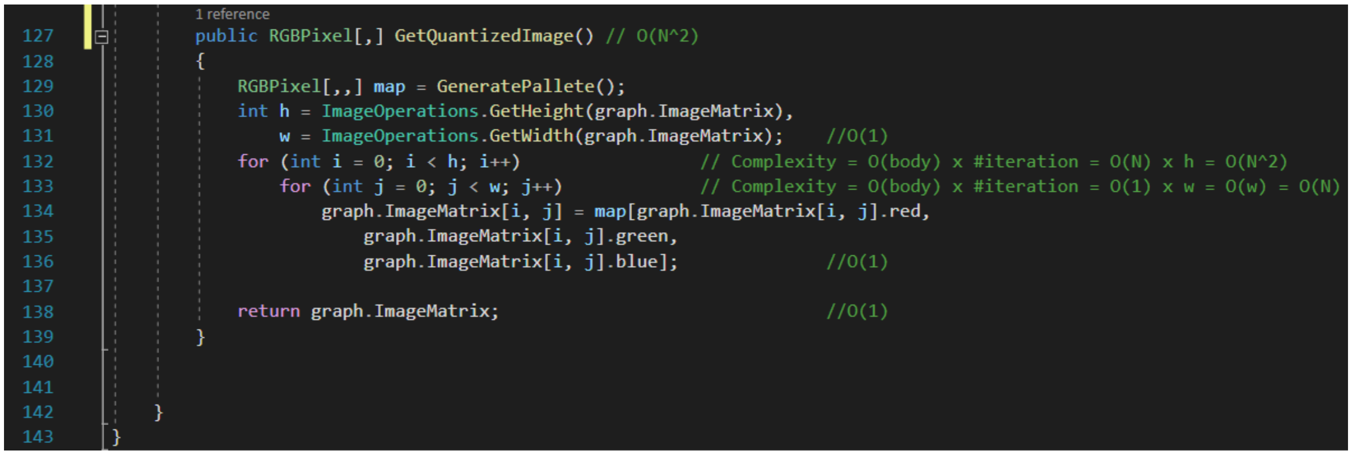


```
            Dictionary<int, List<int>> adjacencyList = new Dictionary<int, List<int>>();                  //O(1)

            for (int i = 0; i < graph.distinctColors; i++) // Complexity = O(body) x #iteration = O(n) x V = O( x )
            {
                if (!adjacencyList.ContainsKey(i))                                       // O(1)
                    adjacencyList.Add(i, new List<int>());                               /* O(n) n -> refers to size of the list
                                                                                         // as when adding new item to list we must iterate on it and
                                                                                         // copy it in a new one so it depends on lsit size
                                                                                         */

                if (parent[i] != -1 && !adjacencyList.ContainsKey(parent[i])) // O(1) + O(1) = O(1)
                    adjacencyList.Add(parent[i], new List<int>());  // O(n)

                if (parent[i] != -1)                                // O(1)
                {
                    adjacencyList[parent[i]].Add(i);                // O(n)
                    adjacencyList[i].Add(parent[i]);                // O(n)
                }
            }

            bool[] visitedColors = new bool[graph.distinctColors];                  //O(1)
            Queue<int> colors = new Queue<int>();                 //O(1)

            RGBPixel[,,] map = new RGBPixel[260, 260, 260];   //O(1)
```



```
            for (int i = 0; i < graph.distinctColors; i++)
            {
                if (!visitedColors[i]){                        //O(1)
                    colors.Enqueue(i);                        //O(n)
                    List<int> cluster = new List<int>();              //O(1)
                    long clusterRedTotal = 0, clusterGreenTotal = 0, clusterBlueTotal = 0; //O(1)
                    while (colors.Count != 0)
                    {                                                 //O(1)
                        int currentColor = colors.First();            //O(1)
                        colors.Dequeue();                             //O(1)
                        cluster.Add(currentColor);                    //O(n)
                        visitedColors[currentColor] = true;           //O(1)
                        foreach (int child in adjacencyList[currentColor]){
                            if (!visitedColors[child])
                                colors.Enqueue(child);
                        }
                        clusterRedTotal += graph.UniqueColors[currentColor].red;              //O(1)
                        clusterGreenTotal += graph.UniqueColors[currentColor].green;          //O(1)
                        clusterBlueTotal += graph.UniqueColors[currentColor].blue;            //O(1)
                    }
                    clusterRedTotal /= cluster.Count;                  //O(1)
                    clusterGreenTotal /= cluster.Count;                //O(1)
                    clusterBlueTotal /= cluster.Count;                 //O(1)
                    foreach (int color in cluster)
                    {
                        RGBPixel newColor = new RGBPixel((byte)(clusterRedTotal),
                                                         (byte)(clusterGreenTotal),
                                                         (byte)(clusterBlueTotal));    //O(1)

                        map[graph.UniqueColors[color].red,
                            graph.UniqueColors[color].green,
                            graph.UniqueColors[color].blue] = newColor;            //O(1)
                    }
```

GetQuantizedImage Complexity



```
1 reference
public RGBPixel[,] GetQuantizedImage() // O(N^2)
{
    RGBPixel[,,] map = GeneratePallete();
    int h = ImageOperations.GetHeight(graph.ImageMatrix),
        w = ImageOperations.GetWidth(graph.ImageMatrix);    //O(1)
    for (int i = 0; i < h; i++)                // Complexity = O(body) x #iteration = O(N) x h = O(N^2)
        for (int j = 0; j < w; j++)            // Complexity = O(body) x #iteration = O(1) x w = O(w) = O(N)
            graph.ImageMatrix[i, j] = map[graph.ImageMatrix[i, j].red,
                graph.ImageMatrix[i, j].green,
                graph.ImageMatrix[i, j].blue];             //O(1)

    return graph.ImageMatrix;                              //O(1)
}


    }
}
```

The function complexity depends on the size of the picture as we loop on the picture and replace every color by its representative from color palette (map) Complexity $O(N^2)$