Image Colour Quantization

- Ahmed essam mohamed ismail (section 2)

- **Finding the distinct colors from the Image matrix**

This constructor of the class Graph builds a list of vertices which contains the distinct colors of the image .

This method allows for the construction of the list of distinct vertices in Θ (height*width).

Code

```
/// <summary>
/// Iterates on the 2D array (ImageMatrix) and stores the distinct colors in a list of vertices
/// </summary>
/// <param name="ImageMatrix"></param>
1 reference
public Graph(RGBPixel[,] ImageMatrix)
{
    Vertices = new List<RGBPixel>();
    VerticesExist = new Dictionary<string, bool>();
    Edges = new List<Edge>();
    int Height = ImageOperations.GetHeight(ImageMatrix);
    int Width = ImageOperations.GetWidth(ImageMatrix);
    for (int i = 0; i < Height; i++)
    {
        for (int j = 0; j < Width; j++)
        {
            ImageMatrix[i, j].ConvertRGBToString();
            if (VerticesExist.ContainsKey(ImageMatrix[i, j].RGBString))
                continue;
            AddVertex(ImageMatrix[i,j]);
        }
    }
}
```

Analysis

The complexity of this loop is

Θ (height*width) +convert RGB to string function complexity which is O(1) + Dictionary<key,value>.Containskey method complexity which is O(1)+AddVertex method complexity which is O(1) …that makes the function executes in O(N^2)

- **Convert_RGB_To_String method analysis**

This function converts the vertex to a string that represents it uniquely.

Code

```
/// <summary>
/// Converts red ,blue and green bytes to strings , Then Concatenates them in one string
/// </summary>
1 reference
public void ConvertRGBToString()
{
    string r, g, b;
    r = Convert.ToString(red);
    g = Convert.ToString(green);
    b = Convert.ToString(blue);
    while (r.Length != 3)
    {
        r = "0" + r;
    }
    while (g.Length != 3)
    {
        g = "0" + g;
    }
    while (b.Length != 3)
    {
        b = "0" + b;
    }
    RGBString = r + g + b;
}
```

Analysis

The function contains 3 loops each iterates a maximum of 3 times which means the total complexity is O(1).

- **Construct The Mst**

The function constructs the minimum spanning tree with prim's algorithm

Code

```
public void ConstructMST()
{
    Edges = new List<Edge>();
    int DisconnectedVertices = Size - 1;
    RGBPixel Vertex = Vertices[0];
    double MinimumWeight, Weight;
    int MinimumValue = 0;
    MSTWeight = 0;
    MSTVertices[0] = new Tuple<int, double>(Vertex.Number, -1);
    Edges_Arr = new Edge[100000];
    int Edges_Size = 0;
    while (DisconnectedVertices != 0)
    {
        MinimumWeight = int.MaxValue;
        for (int i = 0; i < Size; i++)
        {
            if (MSTVertices[i].Item2 == -1)
            {
                continue;
            }
            Weight = Math.Sqrt(((Vertex.red - Vertices[i].red) * (Vertex.red - Vertices[i].red))
                    + ((Vertex.green - Vertices[i].green) * (Vertex.green - Vertices[i].green)) +
                     ((Vertex.blue - Vertices[i].blue) * (Vertex.blue - Vertices[i].blue)));
            if (Weight < MSTVertices[i].Item2)
            {
                MSTVertices[i] = new Tuple<int, double>(Vertex.Number, Weight);
            }
            else
            {
                Weight = MSTVertices[i].Item2;
            }
```

```
            if (Weight < MinimumWeight)
            {
                MinimumWeight = Weight;
                MinimumValue = Vertices[i].Number;
            }
        }
        Edges.Add(new Edge(Vertices[MinimumValue], Vertices[MSTVertices[MinimumValue].Item1], MSTVertices[MinimumValue].Item2));
        Edges_Arr[Edges_Size] = new Edge(Vertices[MinimumValue], Vertices[MSTVertices[MinimumValue].Item1], MSTVertices[MinimumValue].Item2);
        Edges_Size++;
        MSTWeight += MinimumWeight;
        Vertex = Vertices[MinimumValue];
        MSTVertices[MinimumValue] = new Tuple<int, double>(Vertex.Number, -1);
        DisconnectedVertices--;
    }
    //QuickSort(0, Edges.Count - 1, Edges, Edges.Count);
    Edges = Merge_Sort(Edges_Arr, Edges_Size).ToList();
    //Bubble(Edges, Edges.Count);
    //Edges.Sort((Edge E1, Edge E2) =>  E1.Weight.CompareTo(E2.Weight));
    return;
}
```

Analysis

-The outer loop iterates V times “V represents the number of distinct colours” and the inner loop iterates V times which makes the nested loop iterates V^2 times and all other operations in the nested loop executes in Θ(1) .
-Math.sqrt function complexity is O(LogN)
-Dictionary access complexity is O(1)
 Which means the total complexity is Θ(V^2)

- **Merge Sort**

  The function sorts the minimum spanning tree using-merge sort.

Code

```
public Edge[] Merge_Sort (Edge [] E,int Size)
{
    if (Size == 1) return E;
    int MidPoint = Size / 2;
    Edge[] Left = new Edge[MidPoint];
    Edge[] Right = new Edge[Size - MidPoint];
    for (int i = 0; i < MidPoint; i++)
    {
        Left[i] = E[i];
    }
    for (int i = MidPoint; i < Size; i++)
    {
        Right[i - MidPoint] = E[i];
    }
    Left = Merge_Sort(Left, MidPoint);
    Right = Merge_Sort(Right, Size - MidPoint);
    return Combine(Left, Right,MidPoint,Size-MidPoint) ;
}
```

```
public Edge[] Combine (Edge[]Right , Edge[]Left , int LeftSize,int RightSize)
{
    Edge[] Sorted = new Edge[RightSize + LeftSize];
    for (int k = 0,i=0,j=0; k < LeftSize+RightSize; k++)
    {
        if (i>=LeftSize){
            Sorted[k] = Right[j];
            j++;
        }
        else if (j>=RightSize){
            Sorted[k] = Left[i];
            i++;
        }
        else if (Left[i].Weight <= Right[j].Weight){
            Sorted[k] = Left[i];
            i++;
        }
        else{
            Sorted[k] = Right[j];
            j++;
        }
    }
    return Sorted;
}
```

Analysis

Splitting and merging of the array takes Θ **(N) where is N is the length of the array .**

The merge sort function calls itself 2 times this result in a recurrence relation equal to

$T(N) = 2T(N/2) + \Theta(N)$

Using master theorem to calculates its complexity

a=2 , b=2 , F(N)=N

N^(a(Log(b)) = N^1

= N = F(N) (case 2 master theorem)

This means that the total complexity of this relation is Θ(N logN) .

- **Clustering using minimum spanning tree**

The function constructs a number of clusters using the MST.

Code

```
public void Clustering( int NumberOfClusters)
{
    List<int> Rank, Parent;
    List<PalletteNode> Palette;
    Rank = new List<int>();
    Parent = new List<int>();
    Palette = new List<PalletteNode>();
    int forests=Size;
    initialize(Vertices,Rank,Parent,Palette);
    for (int i = 0; i < Size-1 && NumberOfClusters != forests; i++)
    {
            Merge(Edges[i].Either().Number, Edges[i].Other(Edges[i].Either()).Number, Rank, Parent, Palette);
            forests--;
    }
    for (int i = 0; i < Size; i++)
    {
        int a = FindParent(i,Parent);
        if (a==i)
        {
            Palette[i].CalculatePallette();
        }
    }
    GeneratePallette(Parent,Palette);
    return;
}
```

Analysis

Before the first loop the function initializes the components with O(V) complexity.

The first loop at most iterates E "E represents the number of edges in the MST "times exactly K times and calls the function Merge which executes in O(N)complexity"Find Parent function Complexity and all other iteration in Merge function takes O(1) to execute".

The second loop iterates V times and calls Find parent function which executes in O(N) complexity and Calculate pallette which executes in O(1) complexity.

This means the function complexity is O(K*N).

- **Generate Pallette**

  That function calculates the pixels values of the new Image after clustering

  Code

```
void GeneratePallette (List<int> Parent,List<PalletteNode> Pallette)
{
    for (int i = 0; i < Size; i++)
    {
        int parent = FindParent(i, Parent);
        VerticesExist[Vertices[i].RGBString] = Pallette[parent].Vertex;
    }
}
```

  Analysis

  The loop iterates V times and it call Find Parent function Which executes in O(1) because it calculated the parent of the vertices before.

  This mean the function complexity is O(V).

- **Construct the New Image**

  Changes the image matrix to the new image matrix values

  Code

```
public void ConstructTheNewImage(RGBPixel[,] ImageMatrix)
{
    int Height = ImageOperations.GetHeight(ImageMatrix);
    int Width = ImageOperations.GetWidth(ImageMatrix);
    for (int i = 0; i < Height; i++)
    {
        for (int j = 0; j < Width; j++)
        {
            ImageMatrix[i, j] = VerticesExist[ImageMatrix[i, j].RGBString];
        }
    }
}
```

  Analysis

  The complexity of this loop is
  Θ (height*width) that makes the function executes in O(N^2)
  Height and width represents the image dimensions.

- **QuickSort**

  This function sorts the edges by its weight with the quicksort algorithm.

Code

```
/// <summary>
/// Sorts the edges in an ascending order by its weight and pass it by refrence
/// </summary>
/// <param name="startIndex">the index of the first element of the subset of edges</param>
/// <param name="finalIndex">the index of the last element of the subset of edges</param>
/// <param name="E">the List of edges</param>
/// <param name="N">Size of the Edges List</param>
3 references
public void QuickSort(int startIndex, int finalIndex, List<Edge> E , long N)
{
    if (startIndex >= finalIndex) return;
    int i = startIndex, j = finalIndex;
    while (i <= j)
    {
        while (i < N && E[startIndex].Weight >= E[i].Weight) i++;
        while (j > -1 && E[startIndex].Weight < E[j].Weight) j--;
        if (i <= j)
        {
            Swap(i, j, E);
        }
    }
    Swap(startIndex, j, E);
    QuickSort(startIndex, j - 1, E, N);
    QuickSort(i, finalIndex, E, N);
}
```

Analysis

Splitting the array as possible result in sub-problems of size n/2 . Then the total number of levels becomes logn and the complexity of each level is Θ (N).

Using the master theorem method

1. Best Case

   T(N)=2T(N/2)+ Θ (N).

   a=2 , b=2 , F(N)=N

   Then N^(a log(b)) = N^(2 log(2)) = N^1

   Which is equals to F(N) … accepting case 2 of the method

   The complexity is O(N log(N))

2. Worst Case

   Choosing the largest or the smallest element in the array

   The algorithm will call the function N-1 times . each level

Complexity cost Θ (N).

$T(N) = T(N-1) + \Theta(N)$.

- **Find Parent**

  The function takes an element index and searches for its parent recursively.

Code

```
/// <summary>
/// finds the element parent in the graph
/// </summary>
/// <param name="Child">child index</param>
/// <returns>Child's parent index</returns>
5 references
public int FindParent(int Child)
{
    if (Child == Parent[Child])
        return Child;
    return Parent[Child] = FindParent(Parent[Child]);
}
```

Analysis

The function Find_Parent in the best case executes in $\Theta(1)$ which is mean the element is a direct child to the Set's Parent and in the worst case in O(N) which checks for the child's parent in the whole graph.

remark

The function accepts pass compression method by storing the result and returns the cached result when the same inputs occur again.

- **Kmean clustering algorithm**

  The function clusters the image to K clusters using K mean algorithm.

  ## Code

```
public void K_MeanClustring (int NumberOfClusters)
{
    List<PalletteNode> Clusters = new List<PalletteNode>();
    List<PalletteNode> NewClusters = new List<PalletteNode>();
    List<int> Parent = new List<int>();
    for (int i = 0; i < Size; i++)
    {
        Parent.Add(new int());
        if (i >= NumberOfClusters)
            continue;
        Clusters.Add(new PalletteNode());
        NewClusters.Add(new PalletteNode(Vertices[i]));
    }
    while (!EqualClusters(Clusters, NewClusters, NumberOfClusters))
    {
        InitializeNewClusters(NewClusters,Clusters,NumberOfClusters);
        for (int i = 0; i < Size; i++)
        {
            double MinimumDistance = double.MaxValue;
            for (int j = 0; j < NumberOfClusters; j++)
            {
                double Temp = ((Clusters[j].red - Vertices[i].red) * (Clusters[j].red - Vertices[i].red))
                            + ((Clusters[j].green - Vertices[i].green) * (Clusters[j].green - Vertices[i].green))
                            + ((Clusters[j].blue - Vertices[i].blue) * (Clusters[j].blue - Vertices[i].blue));
                if (Temp < MinimumDistance)
                {
                    MinimumDistance = Temp;
                    Parent[i] = j;
                }
            }
```

```
            NewClusters[Parent[i]].red += Vertices[i].red;
            NewClusters[Parent[i]].green += Vertices[i].green;
            NewClusters[Parent[i]].blue += Vertices[i].blue;
            NewClusters[Parent[i]].count++;
        }
        for (int i = 0; i < NumberOfClusters; i++)
        {
            if (NewClusters[i].count == 0)
                continue;
            NewClusters[i].red /= NewClusters[i].count;
            NewClusters[i].green /= NewClusters[i].count;
            NewClusters[i].blue /= NewClusters[i].count;
        }
    }
    GeneratePalletteKMean(Clusters, Parent,NumberOfClusters);
}
```

Analysis

The first loop iterates V times and all its content executes in 1 operation .

The second loop is nested,the outer one iterates R times “R represents how many time the clusters repositioned to get the correct values of the new colours “ and calls Equal Clusters function which its complexity is O(K) and there is 2 inner loops each of them iterates K times and all their inner instructions takes 1 operation to execute.
This means the function complexity is O(V*R).

- **Generate Palette Kmean**

Code

```
public void GeneratePalletteKMean(List<PalletteNode> Clusters ,List<int> Parent, int K)
{
    for (int i = 0; i < K; i++)
    {
        Clusters[i].Vertex = new RGBPixel();
        Clusters[i].Vertex.red = (byte)(Clusters[i].red );
        Clusters[i].Vertex.green = (byte)(Clusters[i].green );
        Clusters[i].Vertex.blue = (byte)(Clusters[i].blue );
    }
    for (int i = 0; i < Size; i++)
    {
        VerticesExist[Vertices[i].RGBString] = Clusters[Parent[i]].Vertex;
    }
}
```

Analysis

The first loop iterates K times and the second one iterates V times . This means the function complexity is O(V).