

16

**Image Color Quantization**

**Team number: 48**

**Introduction** :

The goal of this project 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.

**How to do that ?**

Most standard techniques treat color quantization as a problem of clustering points in three-dimensional space, where the points represent colors found in the original image and the three axes represent the three-color channels.

**Our Approach :**

1. Find the **distinct colors** *D = {d1, d2, d3 ….dm}* from theinput image. Can be known from the image histogram.

-We have RGBPixel[,] ImageMatrix; which represents the pixels of the image in a 2D array of RGBPixel

And then by simply looping over each pixel and adding it to a

HashSet<int> hashDistinctColors; which does not allow duplicates. It takes O(H\*W) where H is the height of the image and W is its width.

-Also, we have converted the RGBPixel for a single integer for

Faster computation and that is done by this function :

public static RGBPixel GetRGBPixel(int rgb)

{

return new RGBPixel((byte)(((1<<8)-1)&(rgb)), (byte)(((1<<8)-1)&(rgb>>8)),

(byte)(((1<<8)-1)&(rgb>>16)));

}

Which Takes O (1) .

We can retrieve the pixels value using this function :

public static int GetRGBInteger(int red, int green, int blue)

{

return red + (green << 8) + (blue << 16);

}

Which also take O (1).

Final step is to map the colors into the list such that The first element of the list contains the first color, the second element contains the second and so on .

private void MapColorsIntoList()

{

distinctColors = new List<int>();

int j = 0;

foreach (var color in hashDistinctColors)

{

distinctColors.Add(color);

}

}

It's done in O(D) where D is the number of the unique color.

}

2-Construct a [minimum-spanning-tree-algorithm](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Algorithms) From the colors

- A spanning tree is an acyclic subgraph of a graph G,

which contains all the vertices from G. The minimum

spanning tree (MST) of a weighted graph is the minimum-

weight spanning tree of that graph

* In our case we don't need to construct a graph due to high number of edges (up to D^2) which consumes a lot of memory
* What we did was calculating the edges cost per request
* We used Prim algorithm which is famous algorithm for finding MST .
* The Lazy implantation of Prims algorithm takes O(ELogV)time and O(E) space which in our case is not suitable
* We have decided to use Eager implantation since it takes the same time but with O(V) space
* Note: E is the number of edges while V is the number of vertices, in our case E equals roughly D^2, and V=D.
* In order to achieve Eager Implantation we needed to use a special data structure called Minimum Indexed priority queue (MIPQ) which can insert and delete the minimum, and can change the key by specifying and index .
* Inserting, deleting, decreasing key of the MIPQ takes O(LogN) time, while checking if it's empty or contains a key takes O(1)
* We have used this reference to help us with the implantation <http://algs4.cs.princeton.edu/43mst/IndexMinPQ.java.html>

**Analysis Of code :**

* **The order of time to find the minimum spanning tree is**

**O(V^2 log v)**

* **The space complexity is O(v)**

Code of MST Graph

|  |
| --- |
| public void GetMst() //O(logv+v^2logv+vlogv)=O(v^2logv)  {  distTo[0] = 0; //O(1)  ipq.Insert(0, distTo[0]); //O(logv)  while (!ipq.IsEmpty()) //O(V\*Order Of the body) O(v^2logv+vlogv)  {  int j = ipq.DeleteMin(); //O(log V)  Proccess(j); //O(vlogV)  }  }  private void Proccess(int j) //O(VlogV)  {  visited[j] = true; //O(1)  for (int i = 0; i < V; ++i) //O(VlogV)  {  if (visited[i]) continue; //O(1)  int edgeCost = Util.CalculateEdgeValue(j, i); //O(1)  if (edgeCost < distTo[i]) //O()  {  distTo[i] = edgeCost; //O(1)  edgeTo[i] = new Edge(edgeCost, j, i); //O(1)  if (ipq.Contains(i)) ipq.DecreaseKey(i, distTo[i]); //O(log v)  else ipq.Insert(i, distTo[i]); //O(log v))  }  }  } |

**3- Clustering**

* our approach is to cut the maximum k-1 edges.
* So we have sorted the edges according to max

this.costs = mst.OrderBy(node => node.cost).ToArray();  O(ElogE)

* Then begin to mark the K-1 edges with cost -1

for (int i = costs.Length - 1; i >= (costs.Length - (K - 1)); i--) O(K)

{

int destination = costs[i].to;  O(1)

edgeTo[destination].cost = -1; O(1)

}

1. **MST(minimum spanning tree) Clustering**

* then we have to construct adjacency list for all edges

for (int i = 0; i < edgeTo.Length; ++i) O(E)

{

adj[edgeTo[i].from].Add(edgeTo[i].to ); O(1)

adj[edgeTo[i].to].Add(edgeTo[i].from); O(1)

}

-We start with calculating every connected graph to calculate The average of Colors Rgb , we declared visited array to prevent revisiting of cycles.

for (int i = 0; i < edgeTo.Length; i++) O(V)

{

if (visited[i] == 1) continue; O(1)

int red = 0, green = 0, blue = 0 ; O(1)

List<int> ClusterNodes = new List<int>(); O(1)

int NumberOfNodes = ColorPalette(i, ref red, ref green, ref blue ,ref ClusterNodes) + 1; O(V)

for (int j = 0; j < ClusterNodes.Count; j++) O(V)

Palette.Add(ImageQuantizer.distinctColors[ClusterNodes[j]],

new RGBPixel((byte)(redNumberOfNodes), (byte)(green / NumberOfNodes), (byte)(blueNumberOfNodes))); O(1)

}

ColorPalette function is a DFS function and Order of DFS is O(E+V) , but in our case E = (V-1) so that the Complexity of function ColorPalette is O(E+V) = O(V+V-1) = O(V).

public int ColorPalette(int node, ref int red, ref int green, ref int blue, ref List<int> ClusterNodes)

{

RGBPixel RGBValue = Util.GetRGBPixel(ImageQuantizer.distinctColors[node]); O(1)

red += Convert.ToInt32(RGBValue.red); O(1)

green += Convert.ToInt32(RGBValue.green); O(1)

blue += Convert.ToInt32(RGBValue.blue); O(1)

ClusterNodes.Add(node); O(1)

visited[node] = 1; O(1)

int ans = 0; O(1)

for (int i = 0; i < adj[node].Count; i++) O(V)

{

int v = adj[node][i]; O(1)

if (edgeTo[node].from == v && edgeTo[node].cost == -1 || edgeTo[v].from == node && edgeTo[v].cost == -1) continue; O(1)

if (visited[v] == 0) O(1)

{

ans += 1 + ColorPalette(adj[node][i], ref red, ref green, ref blue, ref ClusterNodes);

}

}

return ans; O(1)

}

**B-Sorting Algorithm Clustering**

* Sorting algorithm give us a greedy choice because the colors have been sorted according to their distance from the white color and at the ST(spanningtree) I am sure that each color is connected to the nearest color to it and to one color only so that we get the k-1 clusters sequentially.

for (int i = 0; i < edgeTo.Length; i++) O(V)

{

int start = i, end = i; O(1)

int red = 0; O(1)

int green = 0; O(1)

int blue = 0; O(1)

int NumberOfNodes = 1; O(1)

do O(V)

{

RGBPixel RGBValue = Util.GetRGBPixel(ImageQuantizer.distinctColors[i]); O(1)

red += Convert.ToInt32(RGBValue.red); O(1)

green += Convert.ToInt32(RGBValue.green); O(1)

blue += Convert.ToInt32(RGBValue.blue); O(1)

i++; O(1)

NumberOfNodes++; O(1)

end++; O(1)

} while (i < edgeTo.Length && edgeTo[i].cost != -1);

i--; O(1)

NumberOfNodes--; O(1)

end--; O(1)

for (int j = start; j <= end; j++) O(V)

Palette.Add(ImageQuantizer.distinctColors[j],

new RGBPixel((byte)(red/NumberOfNodes), (byte)(green / NumberOfNodes),

(byte)(blue /NumberOfNodes))); O(1)

}