**Team 149**

**Image Quantization Project**

**Milestone One Documentation**

**Student 1**

**علاء احمد عبد التواب عبد الواحد**

**20191700891ID :**

**Student 2**

**سليمان سنوسي سليمان مبروك**

**20191700290ID :**

**Student 3**

**اميرة ياسر محمد عطيه**

**20191700141ID:**

**image compression and Segmentation using algorithmic quantization techniques: K-means, MST-Prim, DFS.**

**Content:-**

1. **Graph construction description and code.**
2. **Minimum spanning tree code.**
3. **Palette generation code.**
4. **Detailed analysis of the above codes.**

**Used Programming Language:-** C#

**Used Structures:-**

Struct Edge : - contains parent node, destination node and the weight of edge between them.

struct Edge

{

public int V1; // Source node

public int V2; // Destination node

public float Weight;// Weight of edge

}

**Used Classes:-**

Vertex\_Parent : - contains current node(V) and parent node(P).

class Vertex\_Parent : FastPriorityQueueNode

{

// Constructor

public Vertex\_Parent(int vertex, int? parent)

{ V = vertex; P = parent; }

public int V { get; set; } // current vertex

public int? P { get; set; } // parent vertex

}

**Used Data Structures:-**

Fast Priority Queue , HashSet , List , Dictionary.

**Used functions:-**

1. 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.

Construct the Graph is implemented on two steps:

**1- Finding Distinct Colors :-**

First getting the width and the height of the image and degree of colors of it, getting R and G and B of each pixel of the image and put each byte of it in the right ordered place.

**Description :-** Extracting distinct colors from ImageMatrix array by looping on it and mapping an encrypted integer value of each PixelRGB to a Hashset to prevent of color values repetition.

Encrypting the value of color is made by the following:-

Encrypted value = (Red value shifted left by 16 + Green value shifted left by 8 + Blue value) .

Then turning the Hashset into a list for the ease of use later on.

Then returning the list of distinct colors.

**Code :-**

public static List<int> GetDistinctcolors(RGBPixel[,] ImageMatrix)

{

int w = ImageMatrix.GetLength(1), h = ImageMatrix.GetLength(0), R, G, B, H = 0;

HashSet<int> D\_colors = new HashSet<int>();

while ( H < h)

{

int W = 0;

while ( W < w)

{

R = ImageMatrix[H, W].red;

G = ImageMatrix[H, W].green;

B = ImageMatrix[H, W].blue;

//Not to System.outOfMemory Exception

int res = (R << 16) + (G << 8) + B;

D\_colors.Add(res);

W++;

}

H++;

}

List<int> Distinct\_colors = D\_colors.ToList();

return Distinct\_colors;

}

Space: O(D2), D is the number of distinct colors in the image

Time : O(N2), N is the image width (or height)

**Analysis of the code :-**

Looping :- O(N^2) 🡪 N is the width or height of ImageMatrix.

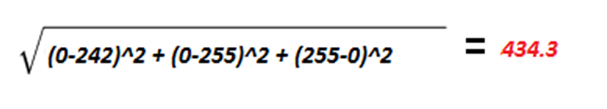
Hashing :- O(1) 🡪 hashset Add() .

Turning to list :- O(D) 🡪 D is the number of distinct colors.

Total :- O(N^2) + O(1) + O(D) = O(N^2) .

**2- calculating distances between each distinct color and the rest :-**

By Applying Euclidean Distance Equation to Calculate Distance Between each distinct color and the rest.



The parameters of this function is from datatype class Vertex\_Parent which contain the vertex and its parent node .

**Description:-**

Calculates the weight of the edge between two vertices by decrypting the integer values of their color value back to the RGB value and implementing the equation :

Weight = (R2-R1)2 + (G2-G1)2 + (B2+B1)2 .

**Code:**

public static float ecludiean(byte r1,byte r2,byte g1,byte g2,byte b1,byte b2) {

return (float)Math.Sqrt((r2 - r1) \* (r2 - r1) + (g2 - g1) \* (g2 - g1) + (b2 - b1) \* (b2 - b1));}

private static float Calculate\_Weight(Vertex\_Parent V1, Vertex\_Parent V2)

{

byte r1,r2,g1,g2,b1,b2;

// Shifting Not to System.outOfMemory Exceptio

//RGB 24 bits R 8,G 8,B 8 shift right

r1 = (byte)(V1.V >>16 );

g1 = (byte)(V1.V >>8 );

b1 = (byte)(V1.V);

r2 = (byte)(V2.V >>16);

g2 = (byte)(V2.V >>8);

b2 = (byte)(V2.V);

float rs = ecludiean(r1, r2, g1, g2, b1, b2);

//return result of ecludiean distance between two vertexes

return rs;

}

**Analysis of the code:-** O(1).

Ecludiean Function is O(1).

And time when using it to calculate distance between each distinct color and the rest in other functions needs **Time:** **O(D2)**, D is the number of distinct colors in the image.

1. **Construct  a**[**minimum-spanning-tree algorithm**](https://en.wikipedia.org/wiki/Minimum_spanning_tree#Algorithms)**(a** [**greedy algorithm**](https://en.wikipedia.org/wiki/Greedy_algorithm)**in**[**graph theory**](https://en.wikipedia.org/wiki/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

Using MST-Prim algorithm

Minimum Spanning tree of the Distinct Color Graph :

**Description :-**

Building the MST chain using the list of the distinct colors(D) .

Using Prim's algorithm and FastPriorityQueue (linked library).

By adding the minimum priority to the MST each time.

**Code :-**

public static List<Edge> Minimum\_Spanning\_Tree(List<int> connected\_undirected\_weighted\_graph)

{

// array contains nodes and their's parent nodes.

Vertex\_Parent[] Vert\_Parent = new Vertex\_Parent[connected\_undirected\_weighted\_graph.Count];

// initializing the first node.

Vert\_Parent[0] = new Vertex\_Parent(connected\_undirected\_weighted\_graph[0], null);

// Fast Priority queue sorts the priority of edges' weights.

FastPriorityQueue<Vertex\_Parent> Fast\_Queue = new FastPriorityQueue<Vertex\_Parent>(connected\_undirected\_weighted\_graph.Count);

// inserting the first node into the queue.

Fast\_Queue.Enqueue(Vert\_Parent[0], 0);

int i = 1;

while (i < connected\_undirected\_weighted\_graph.Count)

{

// intializing all the weights with OO value.

Vert\_Parent[i] = new Vertex\_Parent(connected\_undirected\_weighted\_graph[i], null);

Fast\_Queue.Enqueue(Vert\_Parent[i], int.MaxValue);

i++;

}

List<Edge> Minimum\_Spanning\_Tree\_List = new List<Edge>();

float weight;

while (Fast\_Queue.Count != 0)

{

//minimum priority .

Vertex\_Parent Top = Fast\_Queue.Dequeue();

//it is not the starting node?.

if (Top.P != null)

{

Edge E;

E.V1 = Top.V;

E.V2 = (int)(Top.P);

E.Weight = (float)(Top.Priority);

// add the minimum weight to the MST.

Minimum\_Spanning\_Tree\_List.Add(E);

}

// Update the priority.

foreach (var unit in Fast\_Queue)

{

// calculates the weight between the current node and the top node.

weight = Calculate\_Weight(unit, Top);

if (weight < unit.Priority)

{

unit.P = Top.V;

Fast\_Queue.UpdatePriority(unit, weight);

}

}

}

return Minimum\_Spanning\_Tree\_List;

}

**Analysis of the code : -**

Initializing priorities of each vertex: - O(D) 🡪 D the number of distinct colors.

Prim's Algorithm: - O(DElog(D)) 🡪 for each priority queue vertex O(D)\*modifying the priority O(log(D)) \* for each edge O(E) .

Total: - O(D) + O(DElog(D)) = O(DElog(D)).

**Time:** **proportional to** **O(E Log(V))**, E is number of edges, V is number of vertices

1. **Palette generation code:**

**Description:**

After Extracting the K clusters from the minimum spanning tree with maximal spacing between them using K-Means algorithm.

Looping on each Vertex of all clusters and got RGB of each vertex and Finding the representative color of each cluster by applying R = (R / Number of vertices of each cluster) ,G = (G / Number of vertices of each cluster) , B = (B / Number of vertices of each cluster) and return the representative color of each cluster value = (R << 16) + (G << 8) + (B).

**Code :-**

public static Dictionary<int, int> Generate\_Palette(ref List<HashSet<int>> Clusters)

{

Dictionary<int, int> Result\_colors\_Palette = new Dictionary<int, int>();

int R,G,B, representative\_color = 0;

foreach (var cluster in Clusters)

{

R = 0; G = 0; B = 0;

foreach (var vertex in cluster)

{

R = (R + (byte)(vertex >> 16));

G = (G + (byte)(vertex >> 8));

B = (B + (byte)(vertex));

}

R = (R / cluster.Count);

G = (G / cluster.Count);

B = (B / cluster.Count);

representative\_color = (R << 16) + (G << 8) + (B);

foreach (var vertex in cluster)

{

Result\_colors\_Palette.Add(vertex, representative\_color);

}

}

return Result\_colors\_Palette;

}

**Analysis of the code : -**

External Foreach :- O(C) ; C = Number of Clusters .

First Internal Foreach :- O(V) ; V = Number of vertices of each cluster.

Second Internal Foreach :- O(V) ; V = Number of vertices of each cluster

Total: - O(C) \* (O(V) + O(V)) = O(CV) proportional TO O(D), D is the number of distinct colors.

Time: O(D), D is the number of distinct colors.