**Image Quantization**

**Team 32**

**TA) Dr. Mohamed Magdy**

|  |  |  |
| --- | --- | --- |
| SC – 4 | 20191700606 | محمود محمد أحمد عمران |
| SC - 4 | 20191700620 | مروان محمد عبد الحليم |
| SC – 2 | 20191700274 | سارة عادل الجبالي |

**Introduction:**

**Image Quantization** is an image processing technique that reduces the size of an image by reducing the number of colors.

**To do it we made 4 tasks:**

1. Stored all distinct colors in the image.
2. Calculated the distance (cost) between each node and the correct adjacent node.
3. Correct adjacent node is calculated by choosing the least cost at a certain vertex.
4. Choose K value that represents the number of clusters.
5. Clusters are groups that contains least-distance between colors in this cluster.
6. Calculate the representative color for each cluster by adding all red, green, blue values of all colors and divide by their number.
7. Assign each pixel in the image with its representative color.

**Graph Description:**

Function distinct color calculates the distinct colors of the whole picture.

The function iterates (width x height) times to get all pixels color.

**5 dictionaries are used to calculate the colors:**

**Red:** an array of dictionary using the red value to index a specific dictionary.

**destinctRed:** dictionary to know the initialized dictionaries in (Red).

**Green:** 2d array dictionary that holds all distinct colors using the value of red as a first index and the value of green as a second index then the value of blue is for the key and the Value of this dictionary is the color itself.

**distinctGreen:** - as the distinctRed dictionary- a dictionary that holds the initialized dictionaries of Green to use it to loop and store all these separate colors in dictionary called (Color).

**Color:** The dictionary that holds all distinct colors to use them after.

**Steps:**

* Check if the red value of the given color exists.
  + If so, check if the green value of the color exists in (Red) dictionary.
    - If so, check if the blue value exists in the (Green) dictionary.
      * If it exists, then it is stored color.
      * If no, then create a dictionary of Green - with the red and green value of the given color as the I and j indices - ,put the blue value as a key and store the color in the VALUE of this key.
    - If no, add the color to the this (Red) dictionary and add it also to the (Green) one.
  + If no, create new dictionary of (Red) , also create new (Green) one and add the color to them.
* Loop in (Green) 2d dictionary with the values of (distinctGreen) dictionary to extract the initialized dictionaries.
* Loop in each dictionary – in last step – to extract all distinct colors (different in blue values).

**Analysis:**

|  |  |
| --- | --- |
| Dictionary<int, RGBPixel>[] Red = new Dictionary<int, RGBPixel>[256]; | * O(1) |
| * Dictionary<int, RGBPixel>[,] Green = new Dictionary<int,RGBPixel>[256, 256]; | * O(1) |
| * Dictionary<int, int> distinctRed = new Dictionary<int, int>(); | * O(1) |
| * Dictionary<RGBPixel, int> distinctGreen = new Dictionary<RGBPixel, int>(); | * O(1) |
| * Dictionary<int, RGBPixel> Colors = new Dictionary<int, RGBPixel>(); | * O(1) |
| Dictionary<int, Edges> edges = new Dictionary<int, Edges>(); | * O(1) |
| Dictionary<int, int> max\_Indicies = new Dictionary<int, int>(); | * O(1) |
| bool lastColor = false, check = false; | * O(1) |
| * int ImageWidth, ImageHeight, indices\_Count = 0, k; | O(1) |
| * float Total\_Cost = 0; | O(1) |
| struct Edges  {  public float distance;  public RGBPixel firstColor;  public RGBPixel secondColor;  };   * Edges e; | O(1) |
| * ImageWidth = ImageOperations.GetWidth(ImageMatrix); | * O(1 x (1 x (GetLenght)) |
| * ImageHeight = ImageOperations.GetHeight(ImageMatrix | * 🡪 O(1 x (1 x (GetLenght)) |
| Distinct\_Colors();  for (int i = 0; i < ImageHeight; i++)  {  for (int j = 0; j < ImageWidth;j++)  {   * if(distinctRed.ContainsKey(ImageMatrix[i,j].red))   {   * if (!Red[ImageMatrix[i, j].red].ContainsKey * (ImageMatrix[i, j].green))   {   * Red[ImageMatrix[i, j].red].Add * (ImageMatrix[i, j].green, * ImageMatrix[i, j]);      * Green[ImageMatrix[i,j].red, * ImageMatrix[I,j].green]   = new Dictionary<int, RGBPixel>();   * Green[ImageMatrix[i, j].red, * ImageMatrix[i, j].green].Add * (ImageMatrix[i, j].blue, ImageMatrix[i, j]);      * distinctGreen.Add(ImageMatrix[i, j], 1);   } | O(below code)  O(Height x Body1)  **Body1:**  O(Width x Body2)  **Body2:**  O(1 x Body3)  **Body3:**  O(1 x Body4)  **Body4:**  O(1)  O(1)  O(1)  O(1) |
| else  {   * Red[ImageMatrix[i, j].red] =   new Dictionary<int, RGBPixel>();    Red[ImageMatrix[i, j].red].Add  (ImageMatrix[i, j].green, ImageMatrix[i, j]);    distinctRed.Add(ImageMatrix[i, j].red, 1);  Green[ImageMatrix[i, j].red,  ImageMatrix[i, j].green]  = new Dictionary<int, RGBPixel>();  Green[ImageMatrix[i, j].red,  ImageMatrix[i, j].green].Add  (ImageMatrix[i, j].blue, ImageMatrix[i, j]);  distinctGreen.Add(ImageMatrix[i, j], 1);  }  }  } | O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  **Total order of distinct fun:**  **O(Width x Height)**  or  **O(N2)** ,N is width or height. |
| int counter = 0; | * O(1) |
| foreach (KeyValuePair<RGBPixel, int> i in distinctGreen)  {  foreach (KeyValuePair<int, RGBPixel> j in  Green[i.Key.red, i.Key.green])  {  Colors.Add(counter++, j.Value);  }  } | * O(N x Body1) x   ,N num of colors.  **Body1:**  **O(N x Body2)**  **Body2:**  **O(1)**  Total Complexity =  O(N2) |
| Console.WriteLine(Colors.Count); | * O(1)   Total Complexity of graph:  O(N2) |

**Calculate\_Distance();**

|  |  |
| --- | --- |
| edges = new Dictionary<int, Edges>();  Dictionary<int, float> Key = new Dictionary<int, float>();  bool[] inMST = new bool[Colors.Count];  int leastIndex = 0, counter = 0;  Key.Add(0, 0);  for (int vertix = 0; vertix < Colors.Count - 1; vertix++)  {  float least = 10000000;  int newVertix = leastIndex;  inMST[leastIndex] = true;  for (int adjecent = 0; adjecent < Colors.Count; adjecent++)  {    if (inMST[adjecent] == false)  {  float red = Colors[newVertix].red -  Colors[adjecent].red,  green = Colors[newVertix].green –  Colors[adjecent].green,  blue = Colors[newVertix].blue –  Colors[adjecent].blue;  float sum = (float)Math.Sqrt(red \* red + green \*  green + blue \* blue);  if (vertix == 0)  {  Key.Add(adjecent, sum);  }  else if (sum < Key[adjecent])  {  Key[adjecent] = sum;  }  if (Key[adjecent] < least)  {  least = Key[adjecent];  leastIndex = adjecent;  }  }  if (adjecent == Colors.Count - 1)  {  e = new Edges();  e.distance = least;  e.firstColor = Colors[newVertix];  e.secondColor = Colors[leastIndex];  edges.Add(counter++, e);  Total\_Cost += least;  }  }    }  Console.WriteLine("Total Cost = " + Total\_Cost); | O(1)  O(1)  O(1)  O(1)  O(1)  **O(N x Body1)**  Where N =Distinct Color  Body1 :  O(1)  O(1)  O(1)  **O(N x Body2)**  Where N =Distinct Color  Body2 :  **O(1)**  O(1)  O(1)  O(1)  O(1)  **O(1)**  O(1)  **O(1)**  O(1)  **O(1)**  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  O(1)  Body2= O(1)=> Prim’s Algorithm  O(1)  Total Complexity of Distance:  O(N2) |

void Clusters();

|  |  |
| --- | --- |
| float max\_Distance = -1;  int max\_Index = 0; | O(1)  O(1) |
| for (int i = 0; i < k - 1; i++)  { | O(K) |
| for (int j = 0; j < edges.Count; j++)  { | O(D) |
| if (edges[j].distance > max\_Distance)  { | O(1) |
| max\_Distance = edges[j].distance;  max\_Index = j;  } | O(1)  O(1) |
| }  max\_Indicies.Add(i , max\_Index);  max\_Distance = -1; | O(1)  O(1) |
| e = new Edges();  e.distance = -1;  e.firstColor = edges[max\_Index].firstColor;  e.secondColor = edges[max\_Index].secondColor;  edges[max\_Index] = e;  } | O(1)  O(1)  O(1)  O(1)  O(1) |
| O(K x D) | |
| int last\_min = 0, min = 100000 , delete\_index = 0 ; | O(1) |
| for (int i = 0; i < k; i++)  { | O(K) |
| foreach (KeyValuePair<int,int> j in max\_Indicies)  { | O(D) |
| if (j.Value < min)  {  min = j.Value;  delete\_index = j.Key;  }  } | O(1)  O(1)  O(1) |
| Calculate\_representative\_color(last\_min, min); | O(D) |

|  |  |
| --- | --- |
| last\_min = min + 1;  max\_Indicies.Remove(delete\_index); | O(1)  O(1) |
| if (max\_Indicies.Count == 1 && !lastColor)  {  foreach(KeyValuePair<int, int> x in max\_Indicies)  {  count = max\_Indicies[x.Key];  }  } | O(1)  O(1)  O(1) |
| if (check)  lastColor = false; | O(1)  O(1) |
| if (count == edges.Count - 1 && !check)  {  lastColor = true;  check = true;  } | O(1)  O(1)  O(1) |
| min = 100000;  }  } | O(1) |
| **O(K x D)** | |

void Calculate\_representative\_color(int last\_min, int min)

|  |  |
| --- | --- |
| if (min == 100000)  min = edges.Count - 1; | O(1)  O(1) |
| RGBPixel tmp = new RGBPixel();  float r = 0, g = 0, b = 0;  int counter = 0; | O(1)  O(1)  O(1) |
| for (int i = last\_min; i <= min; i++)  { | O(D) |
| r += edges[i].firstColor.red;  g += edges[i].firstColor.green;  b += edges[i].firstColor.blue;  counter++;  } | O(1)  O(1)  O(1)  O(1) |
| if (min == edges.Count - 1 && !lastColor)  { | O(1) |
| r += edges[min].secondColor.red;  g += edges[min].secondColor.green;  b += edges[min].secondColor.blue;  counter++; | O(1)  O(1)  O(1)  O(1) |
| tmp.red = (byte)Math.Round(r / counter);  tmp.green = (byte)Math.Round(g / counter);  tmp.blue = (byte)Math.Round(b / counter); | O(1)  O(1)  O(1) |
| Green[edges[min].secondColor.red, edges[min].secondColor.green][edges[min]  .secondColor.blue] = tmp;  } | O(1) |
| else  {  tmp.red = (byte)Math.Round(r / counter);  tmp.green = (byte)Math.Round(g / counter);  tmp.blue = (byte)Math.Round(b / counter);  } | O(1)  O(1)  O(1) |
| for (int i = last\_min; i <= min; i++)  {  Green[edges[i].firstColor.red, edges[i].firstColor.green][edges[i].firstColor.blue] = tmp;  } | O(D) |
| Console.WriteLine("Red:" + "(" + tmp.red +")"+ " Green:" + "(" + tmp.green + ")" + " Blue:" + "(" + tmp.blue + ")"); | O(1) |
| **O(D)** | |