|  |  |
| --- | --- |
| Faculty of Computer & Information Sciences  Ain Shams University  Subject: Analysis and Design of Algorithms  Year: (3rd year) undergraduate (CS)  Academic year: 2nd term 2021-2022 |  |

**Image Quantization Project**

**Team ID-T038**

**Department-CS**

TA-Mohamed Magdi

**Team Members:**

|  |  |
| --- | --- |
| Member ID | Name |
| 20191700729 | هادي إيهاب رجاء أحمد |
| 20191700728 | هادي أحمد عبد السلام عبد الحميد |
| 20191700730 | هادي عاطف سيد محمد |

**Notes:**

* We have used a linked library containing the implementation of Fast Priority Queue in order to use it in our MST.

For more information, refer to [here](https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp/wiki/Getting-Started)

Contents

[**1.** **Architecture** 3](#_Toc104015757)

[i. Edge Class 3](#_Toc104015758)

[**2.** **Graph construction** 4](#_Toc104015759)

[i. getDistinctColors() function: 4](#_Toc104015760)

[**Description:** 4](#_Toc104015761)

[**Code:** 5](#_Toc104015762)

[**Analysis**: 5](#_Toc104015763)

[ii. getEculideanDistance() function: 5](#_Toc104015764)

[**Description** 5](#_Toc104015765)

[**Code**: 6](#_Toc104015766)

[**Analysis** 6](#_Toc104015767)

[iii. codeColors() function: 6](#_Toc104015768)

[**Description** 6](#_Toc104015769)

[**Code:** 7](#_Toc104015770)

[**Analysis** 7](#_Toc104015771)

[iv. decodeColors() function: 7](#_Toc104015772)

[**Description** 7](#_Toc104015773)

[**Code:** 7](#_Toc104015774)

[**Analysis** 7](#_Toc104015775)

[**3.** **Minimum Spanning Tree** 8](#_Toc104015776)

[i. buildingMST() function: 8](#_Toc104015777)

[**Description** 8](#_Toc104015778)

[**Code:** 9](#_Toc104015779)

[**Analysis** 9](#_Toc104015780)

[**4.** **Palette Generation** 10](#_Toc104015781)

[i. getInxMaxEdge() function: 10](#_Toc104015782)

[**Description** 10](#_Toc104015783)

[**Code:** 10](#_Toc104015784)

[**Analysis** 10](#_Toc104015785)

[ii. removeEdge() function: 11](#_Toc104015786)

[**Description** 11](#_Toc104015787)

[**Code:** 11](#_Toc104015788)

[**Analysis** 11](#_Toc104015789)

[iii. getClusters() function: 11](#_Toc104015790)

[**Description** 11](#_Toc104015791)

[**Code** 12](#_Toc104015792)

[**Analysis** 13](#_Toc104015793)

[iv. getCentroid() function: 13](#_Toc104015794)

[**Description** 13](#_Toc104015795)

[Code: 14](#_Toc104015796)

[**Analysis** 14](#_Toc104015797)

[v. generatePalette() function: 15](#_Toc104015798)

[**Description** 15](#_Toc104015799)

[**Code** 15](#_Toc104015800)

[**Analysis** 15](#_Toc104015801)

# **Architecture**

## Edge Class

Extending from “FastPriorityQueueNode” Class

Attributes: - “vert” integer

- “parent” integer

# **Graph construction**

## getDistinctColors() function:

### **Description:**

No parameters

body: a hash set of integers is defined named “distinctSet” which will only add the unique colors of the image. Then, a nested for loop is created to pass on every single pixel in our input image “aimImage”. The outer loop passes on x-coordinates, while the inner loop passes on y-coordinates. Inside the inner loop, “codeColors” function is called taking the current pixel as an argument. The function returns an integer of the final encoded Color stored in “encodedColor” integer variable, which is added to “distinctSet” set in the following line. At the end of the function, a list named “listOfDistinct” is defined which will store the “distinctSet” after being converted to list then the count of the list is stored in “noColors” variable.

### **Code:**

public void getDistinctColors()

{

HashSet<int> distinctSet = new HashSet<int>();

for (int x = 0; x < aimImage.GetLength(0); x++)

{

for (int y = 0; y < aimImage.GetLength(1); y++)

{

int encodedColor = colorCoding.codeColors(aimImage[x, y]);

distinctSet.Add(encodedColor);

}

}

listOfDistinct = distinctSet.ToList();

noColors = listOfDistinct.Count;

}

### **Analysis**:

Function’s Order: O (outer loop) \*O (inner loop) \*O (inner body) + O (rest of the function)

Let width of image = N and height of image =H.

Note: in all test cases, the width is greater than or equal to the image’s height.

O (outer loop) = / O (inner loop) =

O (inner body) = O (code Colors) + O (Set insertion) =

O (rest of the function) =

Final Order =

## getEculideanDistance() function:

### **Description**

parameters: “src” & “dst” of type “Edge” Class which carries the vertex node and its parent node

body: the nodes of “src” and “dst” of type “Edge” are decoded using decodeColors() function to use red, blue and green separately and calculate the Eculidean distance between “src” and “dst” and return “res” variable storing this Ecuildean distance.

### **Code**:

public static double getEculideanDistance(Edge src, Edge dst)

{

colorCodingClass =new colorCodingClass();

double res;

RGBPixel srcRGB = colorCodingClass.decodeColors(src.vert);

RGBPixel dstRGB = colorCodingClass.decodeColors(dst.vert);

float X = dstRGB.red - srcRGB.red;

float Y = dstRGB.green - srcRGB.green;

float Z = dstRGB.blue - srcRGB.blue;

res = Math.Sqrt((X \* X) + (Y \* Y) + (Z \* Z));

return res;

}

### **Analysis**

Function’s Order:

## codeColors() function:

### **Description**

parameters: “pixel” of type “RGBPixel”.

body: the whole color code of “pixel” is stored inside an integer variable named “enCodedColor”. This is done by adding the red color (represented in 1 byte) after being shifted to the left by 16 bits to the green color (represented in 1 byte) after being shifted to the left by 8 bits. Then, the result is added to the blue color. Finally, “enCodedColor” variable, which holds the code of the three colors together, is returned at the end.

### **Code:**

public int codeColors(RGBPixel pixel)

{

int enCodedColor = (pixel.red << 16) + (pixel.green << 8) + pixel.blue;

return enCodedColor;

}

### **Analysis**

Function’s Order:

## decodeColors() function:

### **Description**

parameters: “codedColor” of type integer which carries the whole RGB code.

body: “res” variable of type “RGBPixel” carries three attributes of type “byte”; “red”, “green” and “blue”. We set red to “codedColor” after being shifted rightwards by 16 bits casted to byte. The same applies to green but 8 bits not 16. Finally, blue is set to “codedColor” casted to byte and at the end “res” variable is returned.

### **Code:**

public RGBPixel decodeColors(int codedColor)

{

RGBPixel res;

res.red = (byte)(codedColor >> 16);

res.green = (byte)(codedColor >> 8);

res.blue = (byte)(codedColor);

return res;

}

### **Analysis**

Function’s Order:

# **Minimum Spanning Tree**

## buildingMST() function:

### **Description**

No parameters

body: A fast priority queue of type edge “fQueue” is defined with the same size as the list of distinct colors size. In the first for loop, all the edges are enqueued inside “fQueue”. Each edge takes the current vertex of “listOfDistinct” with the vertex’s parent set to -1 and the weight is initialized to infinity. In the next line, we defined a variable of type float named “tmp”. Then, we relax all the edges using a while loop. The loop checks that the queue is not empty to start iterating. Inside the loop, “front” of type edge takes the dequeued queue’s front. The while loop mentioned above contains an if condition and a foreach loop. The if condition checks that the front is not the root. If so the “totalWeight” is increased by the value of the front priority and the front edge is added to “minSpanningTreeEdges” list of edges. Considering the foreach loop, it passes on each edge inside the “fQueue” except the dequeued front. Then, “tmp” variable carries the value that is returned from “getEculideanDistance” function calculating the distance between the current edge “e” and the front edge. If this “tmp” is smaller than the priority of current edge “e”, the parent of “e” is updated to the “vert” of front and the priority of “fQueue” is updated again to rearrange the queue.

### **Code:**

private void buildingMST()

{

FastPriorityQueue<Edge> fQueue = new FastPriorityQueue<Edge>(listOfDistinct.Count);

//All the edges are enqueued inside fQueue

//Each edge takes the current vertex of "listOfDistinct" with parent set to -1

//Edge's weight is set to infinity

for (int i = 0; i < listOfDistinct.Count; i++)

fQueue.Enqueue(new Edge(listOfDistinct[i], -1), int.MaxValue);

float tmp;

while (fQueue.Count != 0) //if the queue isn't empty

{

Edge front = fQueue.Dequeue(); //the queue's front is dequeued

if (front.parent != -1) //if the front is not the root

{

totalWeight+= front.Priority;

minSpanningTreeEdges.Add(front);

}

foreach (var e in fQueue) //each vertex inside the queue except front

{

tmp = (float)getDistanceClass.getEculideanDistance(e, front);

if (tmp < e.Priority)

{

e.parent=(front.vert); //setting the parent of e to front’s vert

fQueue.UpdatePriority(e, tmp); //Updating queue

}

}

}

}

### **Analysis**

Function’s Order: O (for loop) + O (while loop)

O (for loop) = #iterations \* O (body) =D times \* O (enqueue)=

D\*=

O (while loop) = #iterations\*(O (dequeue) + O (if cond.) +O (foreach)) =

=

# **Palette Generation**

## getInxMaxEdge() function:

### **Description**

Function: returns the index of maximum weighted edge.

parameters: “mst” List of edges that exist in the Minimum Spanning Tree

body: Firstly, “ind” & “max” variables are set to 0. Then, a for loop is entered to pass on each edge in “mst” list. If the weight (Priority) of the current element is greater than the max, the max is set to this weight and “ind” is set to the index of the current element. After finishing the loop, “ind” is returned.

### **Code:**

public int getInxMaxEdge(List<Edge> mst)

{

int ind = 0;

float max = 0;

for (int i = 0; i < mst.Count; i++)

{

if (mst[i].Priority > max)

{

max = mst[i].Priority;

ind = i;

}

}

return ind;

}

### **Analysis**

Function’s Order: O (for loop) + O (the rest of function)

Let E=mst.Count (number of edges)

Order of for loop: # times \* loop body= E \* 1 =

Order of the rest=

Function’s Order=

## removeEdge() function:

### **Description**

parameters: Edge “e”

body: Firstly, a new object of “Edge” Class is returned by sending the following attributes in its constructor; vertex, parent and weight. The sent vertex is “vert” of “e”. The sent parent is “parent” variable and the sent weight is -1 because setting it to a negative value means removing it as MST Algorithm isn’t applied on negative weighted edges.

### **Code:**

public Edge removeEdge(Edge e)

{

return new Edge(e.vert,e.parent,-1);

}

### **Analysis**

Function’s Order:

## getClusters() function:

### **Description**

parameters: “vertices” list of integers and “mst” list of edges

Function: returns a list of clusters

body: two dictionaries are defined; “adj”, “color” in addition to “clusters” list of “list of integers” type and “ind” variable set to -1. First for loop is created to pass on every element of “vertices” list. In each iteration, the current vertex with color “w” (refers to white) is added to “color” dictionary. Plus, the current vertex with an empty list of integers is added to “adj” dictionary. Second for loop is created to pass on every element of “mst” list. In each iteration, the priority of current edge of “mst” is checked. If it is not equal to -1 (removed), the “parent” will be added to the adjacency list of edge’s “vert” and vice versa. In the last for loop, it iterates on the size of vertices list checking if the color of current vertex is white. If so, “ind” is incremented by 1, a new list of integers is added to “clusters” list and DFS () function is called to explore the current vertex

### **Code**

public List<List<int>> getClusters(List<int> vertices,List<Edge> mst)

{

adj = new Dictionary<int, List<int>>();

color = new Dictionary<int, char>();

clusters = new List<List<int>>();

ind = -1;

for (int i = 0; i < vertices.Count; i++)

{

color.Add(vertices[i], 'w');

adj.Add(vertices[i],new List<int>());

}

for (int i = 0; i < mst.Count; i++)

{

if (mst[i].Priority != -1) //isn't a removed edge

{

adj[mst[i].vert].Add(mst[i].parent);

adj[mst[i].parent].Add(mst[i].vert);

}

}

for (int i = 0; i < vertices.Count ; i++)

{

if(color[vertices[i]]=='w')

{

ind++;

clusters.Add(new List<int>());

Dfs(vertices[i]);

}

}

return clusters;

}

### **Analysis**

Function’s Order: O (1st for loop) + O (2nd for loop) + O (3rd for loop) + O (the rest)

Let V= vertices.Count, E=mst.Count, K = number of clusters and

D=number of distinct colors of each cluster

O (1st for loop) = #iterations \* O (Body) =

V times \* =

O (2nd for loop) = #iterations \* O (Body) =

(E-K-1) times \* =

O (3rd for loop) = #iterations \* O (Body) =

K times \* =

O (the rest) =

Order =

## getCentroid() function:

### **Description**

Function: calculates the centroid of each cluster which means getting the representative color of each cluster.

No parameters

body: an object of “colorCodingClass” is instantiated named “codingClass”. Then, a palette dictionary is instantiated. Then, there is a nested for loop. The outer loop passes on each cluster from clusters list. Inside it, there are three integers; “red”, “blue” and “green” initialized to 0. The inner loop passes on every color of current cluster. In each iteration of the inner loop, an object “rGB” of type “RGBPixel” is defined carrying the result of the current color after being decoded. Then, “red” is increased by the value of “red” inside “rGB” and same applies to “blue” and “green”. After finishing the inner loop, all colors are divided by the number of colors of the current cluster. After that, “red”, “green” and “blue” will be passed to attributes, carrying the same names, of a new object of type “RGBPixel” named “p”. Then, “mean” variable of integer type will carry “p” after being coded. Finally, for each cluster both the color and its mean will be added to “palette” dictionary.

### Code:

public void getCentroid()

{

colorCodingClass codingClass = new colorCodingClass();

palette = new Dictionary<int, int>();

for (int i = 0; i < clusters.Count; i++) //for each cluster

{

int red = 0, green = 0, blue = 0;

for (int j = 0; j < clusters[i].Count; j++) //for each color

{

RGBPixel rGB = codingClass.decodeColors(clusters[i][j]);

red += rGB.red;

green += rGB.green;

blue += rGB.blue;

}

red /= clusters[i].Count;

green /= clusters[i].Count;

blue /= clusters[i].Count;

RGBPixel p;

p.red = (byte)red;

p.green = (byte)green;

p.blue = (byte)blue;

int mean = codingClass.codeColors(p);

foreach (var l in clusters[i])

{

palette.Add(l, mean);

}

}

}

### **Analysis**

Function’s Order: O (for loop)

Let Count of Clusters = K and D=number of distinct colors

Assume that each cluster “i” has the same number of nodes “L” as the others, where =D/K

O (for loop) = #iterations \*(O (inner loop) +O(foreach)+O(rest))

O (inner loop) =#iterations \* O (decode Colors) + O (rest)=

O (foreach) = #iterations \* O (body) = L \* =

= K \* (

(Distinct Colors)

## generatePalette() function:

### **Description**

Function: returning a color palette dictionary

parameters: “dis” List of integers, “mst” List of edges that exist in the Minimum Spanning Tree and integer “k”

body: Firstly, “mst” list values are passed to “TreeEdges” list. Integer “x” is set to “k” and integer “ind” is initialized. While x is greater than 1, “ind” variable is set to the returned value from “getInxMaxEdge” function of “TreeEdges” list then the edge at index “ind” is removed using “removeEdge” function and x is decremented. After the loop, a list named “c” carries the returned list of clusters from “getClusters” function. Then, “getCentroid” function is called to add elements to “palette” dictionary then “palette” dictionary is returned.

### **Code**

public Dictionary<int,int> generatePalette(List<int> dis,List<Edge> mst, int k)

{

TreeEdges = mst;

int x = k;

int ind;

while (x > 1)

{

ind = getInxMaxEdge(TreeEdges);

mst[ind] = removeEdge(TreeEdges[ind]);

x--;

}

List<List<int>> c = getClusters(dis, TreeEdges);

getCentroid();

return palette;

}

### **Analysis**

Function’s Order: O (while) + O (the rest)

O (while) = #iterations \* O (body) =

(K-1) times \* (O(getInxMaxEdge) + O(removeEdge)) =

O (the rest) = O (getClusters) + O (getCentroid) + O (1) =