**Project Algorithms Analysis Design**

**Image Qountization**

**Team Information**

**Team ID : T135**

**Team Member:**

|  |  |  |
| --- | --- | --- |
| **ID** | **Name** | **Email** |
| **20191700852** | **سارة محمد عبدالعال محمد** | **Sara20191700852@cis.asu.edu.eg** |
| **20191700832** | **منة الله محمد عرفان عبد الفتاح** | **menna2019170832@cis.asu.edu.eg** |
| **20191700319** | **شيماء مجدي عبد الفتاح احمد** | **Shaimaa20191700319@cis.asu.edu.eg** |

1. public class VertexParent
2. {
3. public bool done { get; set; } //O(1)
5. public double weightOfedges { get; set; } = double.MaxValue; //O(1)
6. public int cUrrentVertix { get; set; }//O(1)
7. public int Parentvertix { get; set; } = -1; //O(1)
8. }
9. public static List<RGBPixel> ListOfDC;
10. public static List<int> listOfIndex;
11. public static int FindDistinctColorsANDList(RGBPixel[,] ImageMatrix)
12. {
13. listOfIndex = new List<int>();
14. ListOfDC = new List<RGBPixel>();
15. //function O(N^2) , N->hight\*width
16. int R, G, B;
17. HashSet<int> Set = new HashSet<int>();
18. HashSet<RGBPixel> Setij = new HashSet<RGBPixel>();
19. for (int j = 0; j < GetHeight(ImageMatrix); j++) //O(N)
20. {
21. for (int i = 0; i < GetWidth(ImageMatrix); i++) //O(N)
22. {
23. R = ImageMatrix[j, i].red; //O(1)
24. G = ImageMatrix[j, i].green; //O(1)
25. B = ImageMatrix[j, i].blue; //O(1)
26. Set.Add(R + (G << 8) + (B << 16)); //O(1)
27. Setij.Add(ImageMatrix[j, i]); //O(1)
28. }
29. }
30. listOfIndex = Set.ToList(); //O(1) num of distinct colors
31. ListOfDC = Setij.ToList(); //O(1)
32. return listOfIndex.Count(); //O(1)
33. }
35. public static float CalculateElcideanDistance(RGBPixel V1, RGBPixel V2)
36. {
37. //function O(1) distance between two vertex
38. byte r1, r2, g1, g2, b1, b2;
40. r1 = V1.red;
41. g1 = V1.green;
42. b1 = V1.blue;
43. r2 = V2.red;
44. g2 = V2.green;
45. b2 = V2.blue;
46. return (float)Math.Sqrt((r2 - r1) \* (r2 - r1) + (g2 - g1) \* (g2 - g1) + (b2 - b1) \* (b2 - b1)); //O(1)
47. }
48. public static VertexParent[] vertixOf;//O(1)
49. public static double MinimumSpanning()//O(E Log(V)) O(n^2)
50. {
52. List<int> adjTosaveTree = new List<int>();//O(1)
53. Priorityqueue<VertexParent> queueNodeOf = new Priorityqueue<VertexParent>();//O(1)
55. vertixOf = new VertexParent[ListOfDC.Count];//O(1)
56. vertixOf[0] = new VertexParent() { weightOfedges = 0, Parentvertix = -1, cUrrentVertix = 0, done = false };
57. queueNodeOf.Enqueue(vertixOf[0], vertixOf[0].weightOfedges);
58. for (int i = 1; i < ListOfDC.Count; i++)//O(D)
59. {
60. vertixOf[i] = new VertexParent() { weightOfedges = CalculateElcideanDistance(ListOfDC[0], ListOfDC[i]), Parentvertix = -1, cUrrentVertix = i, done = false }; //O(1)
61. queueNodeOf.Enqueue(vertixOf[i], vertixOf[i].weightOfedges);
62. }
63. double cost = 0.0;
64. while (queueNodeOf.Count > 0)//O(N^2) whic N is size of queue
65. {
66. VertexParent getminver = queueNodeOf.Dequeue(); //O(1) the order of dequeue// to get the minimum priority of vertices
67. int u = getminver.cUrrentVertix;//O(1)
68. vertixOf[u].done = true;//O(1)
69. cost += getminver.weightOfedges; //O(1)
70. adjTosaveTree.Add(listOfIndex[getminver.cUrrentVertix]);
71. for (int iterator = 0; iterator < ListOfDC.Count; iterator++)//O(D)
72. {
73. if (!vertixOf[iterator].done) //O(1)
74. {

if (CalculateElcideanDistance(ListOfDC[u], ListOfDC[iterator]) < vertixOf[iterator].weightOfedges && CalculateElcideanDistance(ListOfDC[u], ListOfDC[iterator]) > 0)

{

vertixOf[iterator].Parentvertix = u; //O(1)

vertixOf[iterator].weightOfedges = CalculateElcideanDistance(ListOfDC[u], ListOfDC[iterator]); //O(1)

queueNodeOf.UpdatePriority(vertixOf[iterator], vertixOf[iterator].weightOfedges); //O(

}

}

}

}

return cost;

}

1. public struct Nodes\_of\_colores
2. {
3. public int nodeindex1 { get; set; }
4. public int nodeindex2 { get; set; }
5. public double Distance { get; set; }
6. public int connectednodeindex1 { get; set; }
7. public int connectednodeindex2 { get; set; }
9. }
10. public struct colorindecesmatrix
11. {
12. public int nodeindex1 { get; set; }
13. public int nodeindex2 { get; set; }

16. }
17. public struct Nodes
18. {
19. public int node { get; set; }
20. public int connectnode { get; set; }
21. public float Distance { get; set; }
23. }
24. public struct Nodes\_clusters
25. {
26. public int node { get; set; }
28. public double Distance { get; set; }
30. }
31. public static float[] Distance;
32. public static int[] pointvertices;
33. public static List<KeyValuePair<int, int>> DistictColorsPixels\_indeces;
35. public static List<int>[] DistictColorsPixels;
36. public static Dictionary<int, List<int>> Adjasent\_list = new Dictionary<int, List<int>>();
38. //function 3
39. public static List<HashSet<int>> FindTheClustersForDistictColor(List<int> list\_Msp, int numberofclusters)//O(K\*D)
40. {
42. // int counter\_forcalcmindistance = 0;
43. List<HashSet<int>> ClustersofColors = new List<HashSet<int>>();//O(1)
44. List<double> ResultOfEquation\_Distance = new List<double>();//O(1)
46. // Fill The adgacent List
47. Editmsp(list\_Msp, numberofclusters);//O(K\*D)
48. foreach (var item in list\_Msp)//O(D) D are distict color
49. {

52. if (!Adjasent\_list.ContainsKey(item))//O(1)
53. Adjasent\_list.Add(item, new List<int>());//O(1)
55. if (pointvertices[item] != -1 && !Adjasent\_list.ContainsKey(pointvertices[item]))//O(1)
56. Adjasent\_list.Add(pointvertices[item], new List<int>());//O(1)
58. if (pointvertices[item] != -1)//O(1)
59. {
60. Adjasent\_list[pointvertices[item]].Add(item);//O(1)
61. Adjasent\_list[item].Add(pointvertices[item]);//O(1)
62. }
63. }
65. HashSet<int> visitedNodes = new HashSet<int>();//O(1)
66. foreach (var vertex in Adjasent\_list)//O(v)v are number vertices in each list
67. {
68. if (!visitedNodes.Contains(vertex.Key))// O(v)
69. {
70. HashSet<int> set = new HashSet<int>();//O(1)
71. Removing\_repeats(ref visitedNodes, vertex.Key, ref set);
72. ClustersofColors.Add(set);//O(1)
74. }
75. }
76. return ClustersofColors;//O(1)
78. }

81. public static void Editmsp(List<int> list\_Msp, int number\_of\_clusters) //O(K\*D)
82. {
83. int loopcount = 0; //O(1)
84. int numditictcolor = list\_Msp.Count; //O(1)
85. pointvertices = new int[numditictcolor]; //O(1)
86. Distance = new float[numditictcolor]; //O(1)
87. while (loopcount < number\_of\_clusters - 1)
88. { //O(K)
89. int c = 0; //O(1)
90. int maxind = 0; //O(1)
91. float max\_distance = 0; //O(1)
93. foreach (var item in list\_Msp)
94. {//O(D)


98. if (Distance[item] > max\_distance)//O(1)
99. {
100. max\_distance = Distance[item];//O(1)
101. maxind = item;//O(1)
102. }
103. c++;//O(1)
104. }
105. pointvertices[maxind] = -1;//O(1)
106. Distance[maxind] = 0;//O(1)
107. loopcount++;//O(1)
108. }
110. }
112. public static void Removing\_repeats(ref HashSet<int> visited, int currentvertex, ref HashSet<int> cluster\_try)
113. {

116. cluster\_try.Add(currentvertex);//O(1)
117. visited.Add(currentvertex);//O(1)
118. List<int> list\_tring = Adjasent\_list[currentvertex];//O(1)
119. foreach (var neighbour in list\_tring)//O(v) v are vertices in each list in list\_tring
120. {
121. if (!visited.Contains(neighbour))//O(v)
122. Removing\_repeats(ref visited, neighbour, ref cluster\_try);
123. }



128. }

131. // function 4
132. public static Dictionary<int, int> FindTherepresentiveColorForeachcluster(List<HashSet<int>> ClustersofColors, int number\_distenctcolor)//O(D)=O(K\*N) D is ditinct color For every Cluster
133. {
134. int avrred = 0, avrgreen = 0, avrblue = 0;//O(1)
135. int counterrr;//O(1)
136. int finalcolor;//O(1)
137. int[] collection\_colorredsum = new int[number\_distenctcolor];//O(1)
138. int[] collection\_colorgreensum = new int[number\_distenctcolor];//O(1)
139. int[] collection\_colorbluesum = new int[number\_distenctcolor];//O(1)
140. Dictionary<int, int> colorandreprsentivecolor = new Dictionary<int, int>();//O(1)
141. foreach (var item1 in ClustersofColors)
142. {//O(K)
143. HashSet<int> hass = item1;//O(1)
144. counterrr = 0;//O(1)
145. avrred = 0;//O(1)
146. avrgreen = 0;//O(1)
147. avrblue = 0;//O(1)
148. finalcolor = 0;//O(1)
149. foreach (var item2 in hass)
150. {//O(n) n is ditinct color For every Cluster
151. collection\_colorredsum[counterrr] = (byte)(item2 >> 16);//O(1)
152. collection\_colorgreensum[counterrr] = (byte)(item2 >> 8);//O(1)
153. collection\_colorbluesum[counterrr] = (byte)(item2);//O(1)
154. counterrr = counterrr + 1;//O(1)
155. }
156. for (int c = 0; c < counterrr; c++) //O(D) D is ditinct color For every Cluster
157. {
158. avrred += collection\_colorredsum[counterrr];//O(1)
159. avrgreen += collection\_colorgreensum[counterrr];//O(1)
160. avrblue += collection\_colorbluesum[counterrr];//O(1)

163. }
164. avrred = avrred / counterrr;//O(1)
165. avrgreen = avrgreen / counterrr;//O(1)
167. avrblue = avrblue / counterrr;//O(1)
168. finalcolor = (avrred << 16) + (avrgreen << 8) + (avrblue);//O(1)
169. foreach (var item2 in hass)//O(D) D is ditinct color For every Cluster
170. {
171. colorandreprsentivecolor.Add(item2, finalcolor);//O(1)
172. }
174. Array.Clear(collection\_colorredsum, 0, counterrr);//O(D) D is ditinct color For every Cluster
175. Array.Clear(collection\_colorgreensum, 0, counterrr);//O(D) D is ditinct color For every Cluster
176. Array.Clear(collection\_colorbluesum, 0, counterrr);//O(D) D is ditinct color For every Cluster
177. }
178. return colorandreprsentivecolor;//O(1)

181. }
183. // function 5
184. public static void QuantizationTheImage(RGBPixel[,] Matrixforimagepath, Dictionary<int, int> colorandreprsentivecolor)
185. {
186. int color = 0;//O(1)
187. int counter\_rows = GetHeight(Matrixforimagepath);//O(1)
188. int counter\_columns = GetWidth(Matrixforimagepath);//O(1)
189. Dictionary<int, colorindecesmatrix> listcolors = new Dictionary<int, colorindecesmatrix>(counter\_rows \* counter\_columns);//O(1)
190. int counter\_loop1 = 0;//O(1)
191. int counter\_loop2 = 0;//O(1)
193. colorindecesmatrix struc = new colorindecesmatrix();//O(1)
194. while (counter\_loop1 < counter\_rows)
195. {//O(N)
197. while (counter\_loop2 < counter\_columns)//O(N)
198. {
199. int red = Matrixforimagepath[counter\_loop1, counter\_loop2].red;//O(1)
200. int blue = Matrixforimagepath[counter\_loop1, counter\_loop2].blue;//O(1)
201. int green = Matrixforimagepath[counter\_loop1, counter\_loop2].green;//O(1)
202. color = (red << 16) + (green << 8) + blue;//O(1)
203. struc.nodeindex1 = counter\_loop1;//O(1)
204. struc.nodeindex2 = counter\_loop2;//O(1)
205. listcolors.Add(color, struc);//O(1)
207. counter\_loop2 = counter\_loop2 + 1;//O(1)
208. }
209. counter\_loop1 = counter\_loop1 + 1;//O(1)
211. }
212. foreach (var item in listcolors)//O(N^2)
213. {
214. int value = colorandreprsentivecolor[item.Key];//O(1)
215. Matrixforimagepath[item.Value.nodeindex1, item.Value.nodeindex2].red = (byte)(value >> 16);//O(1)
216. Matrixforimagepath[item.Value.nodeindex1, item.Value.nodeindex2].green = (byte)(value >> 8);//O(1)
217. Matrixforimagepath[item.Value.nodeindex1, item.Value.nodeindex2].blue = (byte)(value);//O(1)

220. }
221. }
223. }
224. }
226. public class Priorityqueue<T>
227. {
228. public struct NodeOvertices
229. {
230. public double Priority; //O(1)
231. public T Object { get; set; } //O(1)
232. }
233. List< NodeOvertices > queue = new List< NodeOvertices >(); //O(1)
234. static int SizeOFHeap = -1; //O(1)
236. public int Count { get { return queue.Count; } } //O(1)
237. //public int Count1 { get; } // O(1)
238. public int CLeft(int i) //O(1)
239. {
240. return i \* 2 + 1; //O(1)
241. }
242. public int CRight(int i) //O(1)
243. {
244. return i \* 2 + 2; //O(1)
245. }
247. public void Exchange(int iiii, int jjj)
248. {
249. var temp = queue[iiii]; //O(1)
250. queue[iiii] = queue[jjj]; //O(1)
251. queue[jjj] = temp; //O(1)
252. }
253. public void Enqueue(T obj, double priority)
254. {
255. NodeOvertices node = new NodeOvertices () { Priority = priority, Object = obj };
256. queue.Add(node);
257. SizeOFHeap++; //O(1)
259. }
260. public T Dequeue() //O(1)
261. {
262. if (SizeOFHeap > -1) //O(1)
263. {
265. var returnVal = queue[0].Object; //O(1)
266. queue[0] = queue[SizeOFHeap];//O(1)
267. queue.RemoveAt(SizeOFHeap); //O(1) w
268. SizeOFHeap--;//O(1)


272. return returnVal;
273. }
274. else
275. throw new Exception("Queue is empty");
277. }
279. public void UpdatePriority(T obj, double priority) //O(s)
280. {

283. for (int i = 0; i <= SizeOFHeap; i++) //O(s) sizeOfheap
284. {
285. NodeOvertices node = queue[i];
286. if (object.ReferenceEquals(node.Object, obj))
287. {
288. node.Priority = priority;//O(1)
289. queue[i] = node;
290. BuildHeapMin(i);
291. MinimumHeapOfP(i);
292. }
293. }
294. }
295. public void MinimumHeapOfP(int i)
296. {
297. int l = CLeft(i); //O(1)
298. int R = CRight(i); //O(1)
299. int lowest = i; //O(1)
301. if (l <= SizeOFHeap && queue[l].Priority > queue[lowest].Priority)
302. {
303. lowest = l; //O(1)
304. }
305. else
306. {
307. lowest = i; //O(1)
308. }
309. if (R <= SizeOFHeap && queue[R].Priority < queue[lowest].Priority)
310. {
311. i = R; //O(1)
312. }
313. if (lowest != i)
314. {
315. Exchange(lowest, i);
316. MinimumHeapOfP(lowest);
317. }
318. }
320. private void BuildHeapMin(int i)
321. {
322. int pObject = (i - 1) / 2;
323. while (i >= 0 && queue[pObject].Priority < queue[i].Priority)
324. {
325. Exchange(i, pObject);
326. i = pObject;
327. }
328. }
330. }