Jules HIRTZ
Cedran BASTIEN

Dans le cadre de ce projet, l'objectif est de comparer la solution proposée par le professeur avec celle imaginée par un algorithme visant à réduire le nombre de couleurs d'une image. Pour faciliter la compréhension, nous allons diviser cette comparaison en trois parties distinctes.

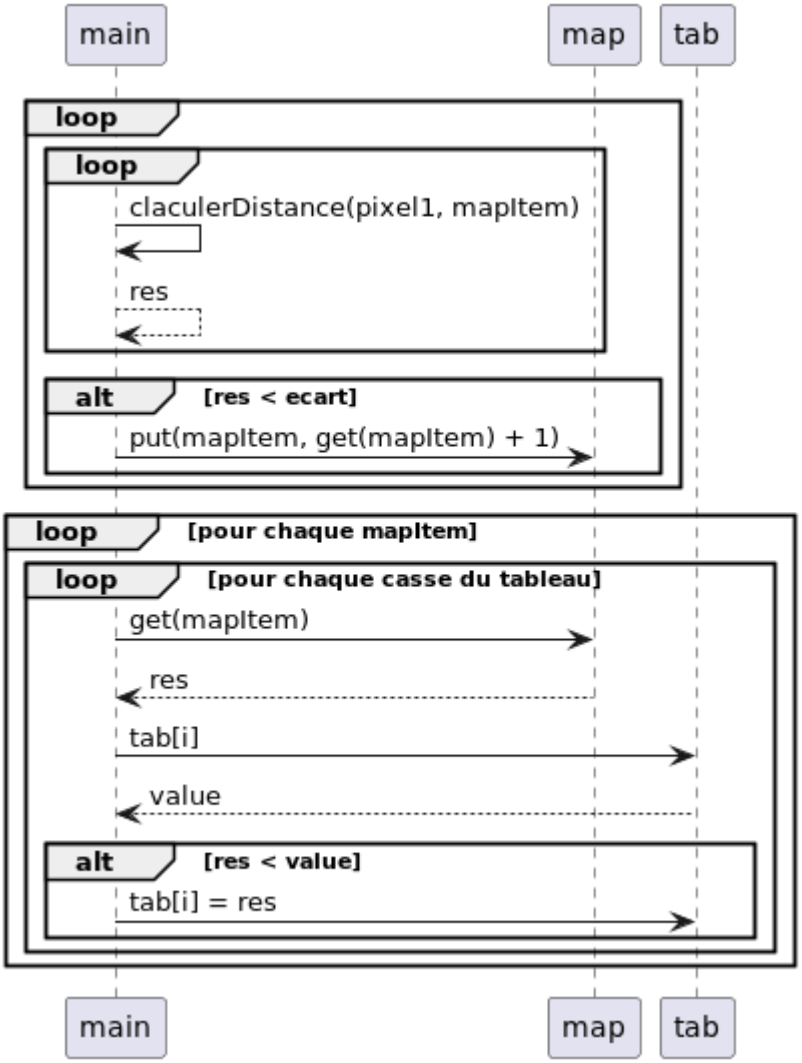
1. Notre idée initiale et ses problèmes :
Notre idée de départ consistait à récupérer le nombre d'occurrences de chaque couleur dans l'image, puis à sélectionner les X premières couleurs les plus fréquentes pour générer une nouvelle image à nombre limité de couleurs (X correspondant au nombre de couleurs souhaitées). Ces X couleurs étaient stockées dans un tableau. Pour effectuer ce classement, nous avons utilisé une HashMap pour stocker les couleurs et leurs fréquences, puis nous les avons triées afin de remplir le tableau des couleurs utilisé pour créer la nouvelle image. Afin d'éviter d'avoir une couleur par pixel, nous avons introduit une tolérance entre les couleurs de la HashMap. Si la distance entre la couleur courante et une couleur de la HashMap était inférieure à une tolérance prédéfinie, la couleur du pixel était considérée comme la couleur testée de la HashMap. Une fois le tableau de X couleurs récupérées, chaque pixel de l'image d'origine était remplacé par la couleur du tableau ayant la plus petite distance par rapport à la couleur du pixel courant. Cette nouvelle image était ensuite enregistrée.
Algorithme :

```
public static void main(String[] args) throws IOException {
        doRefacter(Integer.parseInt(args[0]));
    }

    public static void doRefacter(int nbColor) throws IOException {


        // Getting image
        BufferedImage image = ImageIO.read(new
File("images_etudiants/originale.jpg"));

        // Getting length
        int width = image.getWidth();
        int heigth = image.getHeight();

        // Setting colors tab
        // Set data
        long ecart = 15000;
        Map<String, Integer> map = new HashMap<>();

        // Browse image
        for (int i = 0; i < heigth; i++) {
            for (int j = 0; j < width; j++) {
                // Get RGB
```

```
                int[] RGB = getRgb(image.getRGB(j, i));

                // Check if color exist
                boolean isExist = false;
                for (String item : map.keySet()) {
                    System.out.print("x: " + j + " y: " + i + " ");
                    System.out.println(claculerDistance(getRgb(map.get(item)),
RGB));
                    if (claculerDistance(getRgb(map.get(item)), RGB) < ecart) {
                        map.put(item, map.get(item) + 1);
                        isExist = true;
                        break;
                    }
                }

                // Add color if not exist in map
                if (!isExist) {

                    map.put(Integer.toString(image.getRGB(j, i)) , 1);
                }




            }


        }

        System.out.println(map);

        // Set up tab
        tab = new int[nbColor];
        for (int k = 0; k < nbColor; k++){
            tab[k] = map.get(map.keySet().toArray()[k]);
        }

        // Sort map by value in a list
        for (String item : map.keySet()) {
            for (int k = 0; k < nbColor; k++) {
                System.out.printf(item+  "");
                if (map.get(item) > map.get(tab[k])) {
                    tab[k] = Integer.parseInt(item);
                    break;
                }
            }
        }

        // Creating new image of the right length
        BufferedImage newImage = new BufferedImage(width, heigth,
BufferedImage.TYPE_3BYTE_BGR);
```

```
        // pour chaque ligne
        for(int i = 0; i<heigth; i++){
            // Pour chaque colone
            for (int j = 0; j<width; j++){
                // Set RGB ou each pixel
                newImage.setRGB(j,i, lessDistance(image.getRGB(j,i)) );
            }
        }

        // Saving new image
        ImageIO.write(newImage, "png", new File("Q2.png"));
    }

    public static int[] getRgb(int RGB) {
        int[] res = {(RGB & 0xff), (RGB & 0xff00), (RGB & 0xff0000)};
        return res;
    }

    public static int lessDistance(int rgb) {
        int min = (int) Double.MAX_VALUE;
        for (int item : tab) {
            if (claculerDistance(getRgb(item), getRgb(rgb)) < min) {
                min = item;
            }
        }
        return min;
    }

    public static long claculerDistance(int[] RGB1, int[] RGB2) {
        return (long) (Math.pow((RGB1[0] - RGB2[0]), 2) + Math.pow((RGB1[0] -
RGB2[0]), 2) + Math.pow((RGB1[0] - RGB2[0]), 2));
    }
```

Conception :



Cependant, après avoir implémenté, testé et débogué cet algorithme, nous avons constaté qu'il ne fonctionnait pas comme prévu. Tout d'abord, le remplissage de la HashMap prenait beaucoup trop de temps, dépassant les 10 minutes pour une image d'environ 800 * 800 pixels.


```
{-14931200=235, -13353463=211, -1412864=2812, -12955114=1532, -14601974=151, -14601971=158, -14601972=206, -15063033=204, -14209783=2299, -14666486=156, -15063035=204, -14209781=1791, -16315903=149,
-14666482=153, -14666484=148, -15523323=204, -4571392=510, -13547227=1021, -13613295=589, -13547223=1021, -14076919=589, -14797821=151, -12165598=764, -1606656=508, -14405881=204, -15327227=252, -13352148=764,
-16051718=166, -14081024=764, -15590144=252, -11836358=764, -16051712=763, -13744632=203, -14797818=122, -14535919=198, -14797819=150, -744182=507, -12692696=763, -11836355=763, -13284332=199, -11656784=195,
-14076906=254, -14401018=150, -14797811=253, -14671866=198, -14076916=150, -14671868=253, -15390976=147, -13813734=252, -14671864=251, -16382720=251, -14929396=251, -15194869=251, -14929397=198, -13226487=251,
-12568309=251, -14601964=251, -4050944=251, -13480429=251, -12361688=251, -12959991=251, -12041720=251, -13417171=251, -14139390=150, -14605052=251, -12107253=251, -13356535=197, -15658235=251, -3008832=251,
-13744618=251, -12887773=251, -2538240=251, -13744622=251, -15726080=196, -13943803=196, -16118526=196, -16185344=250, -14535927=250, -13552617=250, -14073848=150, -14535921=150, -14271741=150, -13158653=194,
-12099530=250, -14139380=250, -2080256=250, -13816832=250, -14871037=192, -16185338=250, -13417199=250, -13942509=133, -13547251=250, -13878257=250, -1674749=250, -15987959=250, -14073838=150, -10981312=250,
-15853054=250, -14736891=250, -14206208=187, -14271724=149, -12633079=250, -13547243=250, -14605046=250, -13157346=249, -15064312=149, -13744606=249, -13092343=181, -15064314=249, -15064316=249, -13681398=249,
-12372728=249, -14534649=149, -14146048=249, -12306418=249, -15459870=172, -13811173=249, -15126775=249, -14338550=146, -13221625=249, -13811177=145, -13679888=249, -12893171=249, -15987968=166, -14534639=249,
-14277886=249, -14998772=156, -14204928=150, -14338560=149, -13349343=249, -14668277=148, -11835081=249, -5560064=249, -4047360=249, -13942512=125, -14802423=249, -13091831=249, -14802425=249, -12757224=249,
-13151206=249, -13941219=249, -1018880=249, -16513792=148, -14732025=147, -12897524=147, -14732026=147, -16448251=249, -14273019=249, -16448253=249, -13418488=249, -13219317=147, -15856128=249, -14474487=249,
-15856126=249, -14474489=249, -15856124=249, -14732028=147, -15856122=249, -12366065=147, -15328507=249, -14543103=147, -3594240=249, -15196668=249, -16184832=147, -15524604=249, -14606336=249, -15524606=249,
-14204904=249, -14011125=249, -16184830=249, -15330816=249, -10210541=249, -14863861=144, -14204906=248, -14347008=248, -15392766=248, -12695794=248, -14863863=147, -14863865=144, -15460093=146, -14338540=146,
-16184828=248, -14142961=248, -15787006=248, -14801142=248, -686592=248, -14801140=146, -13548521=145, -5632256=248, -15986688=145, -15986686=248, -13022451=145, -15129848=248, -15986687=145, -14335480=145,
-15921656=248, -15129846=145, -13220331=248, -14673664=248, -13090552=145, -13941233=248, -15066106=248, -13550839=248, -14939136=145, -13684470=145, -13094901=248, -14339827=145, -15325433=145, -12894451=248,
-13882358=248, -13021148=248, -12241397=248, -12894452=248, -15721213=248, -12698874=248, -14407675=248, -15000315=248, -13218027=248, -12758505=248, -3658752=248, -15193596=145, -15589116=145, -14603262=248,
-14335478=248, -13156891=248, -12362975=248, -14733308=144, -14533368=144, -15591424=122, -14799102=144, -14533366=248, -14733308=144, -14733309=144, -14733306=144, -12698862=120, -3377664=248, -15265792=144,
-13085670=248, -14402809=134, -14402803=144, -15195391=248, -14476288=143, -14340352=120, -13349874=248, -13621245=120, -15130877=248, -13349870=248, -15391487=119, -11572934=248, -15130875=248, -14338037=143,
-13223403=248, -14672379=247, -15526399=247, -15391488=143, -15526397=247, -13612766=247, -13416694=143, -12960498=143, -13487871=137, -14868218=247, -14402813=143, -14868216=247, -13565962=247, -15987449=247,
-13416696=143, -15199744=143, -15128569=143, -15195387=143, -13349860=247, -13550871=247, -15655673=247, -15062515=143, -15195389=143, -16250874=247, -14600182=141, -14273533=247, -14009310=247, -15063805=247,
-14600188=141, -14732542=141, -3792128=247, -14604544=141, -13415378=247, -12893684=247, -14604543=140, -14600188=140, -13088725=247, -14273535=140, -13152495=139, -16250880=128, -14406382=247, -13877756=139,
-14600192=139, -14009326=139, -14011642=139, -13683967=247, -13750523=247, -14732530=139, -13349846=246, -13750525=139, -16119037=246, -13349844=246, -16121344=139, -13156854=246, -16119034=246, -16119035=246,
```


De plus, les valeurs d'occurrence des couleurs dans la HashMap étaient trop similaires pour permettre un classement objectif et utilisable. Il était donc difficile de remplir correctement le

tableau des couleurs. La capture d'écran de l'image générée avec cette méthode montre déjà une redondance de valeurs au début de la HashMap. En conséquence, nous avons abandonné cette approche et nous nous sommes tournés vers une autre solution.

2. L'autre solution : K-means
Nous avons ensuite implémenté une nouvelle solution en utilisant l'algorithme de K-means fourni par le professeur. Cette méthode consiste à regrouper les couleurs similaires dans des "groupes" en fonction de leur proximité dans l'espace des couleurs.

Algorithme :

```
Fonction getRgb(RGB)
    res[0] = (RGB & 0xff)
    res[1] = (RGB & 0xff00) >> 8
    res[2] = (RGB & 0xff0000) >> 16
    Retourner res

Fonction calculerDistance(RGB1, RGB2)
    Retourner (RGB1[0] - RGB2[0])^2 + (RGB1[1] - RGB2[1])^2 + (RGB1[2] -
RGB2[2])^2

Fonction KMean()
    Ng = 2
    image = ChargerImage("images_diverses_small/animaux/ours.png")
    debut = ObtenirTempsActuel()
    tab[Ng]

    Pour i de 0 à Ng
        x = EntierAleatoire(0, largeur(image))
        y = EntierAleatoire(0, hauteur(image))
        rgb = ObtenirRGB(image, x, y)
        tab[i] = rgb

    carryOn = Vrai
    groupes = HashMap()

    Tant que carryOn
        Pour i de 0 à Ng
            groupes[i] = ListeVide()

        Pour i de 0 à hauteur(image)
            Pour j de 0 à largeur(image)
                rgb = ObtenirRGB(image, j, i)
                min = EntierMaximal()
                indice = 0

                Pour k de 0 à Ng
```

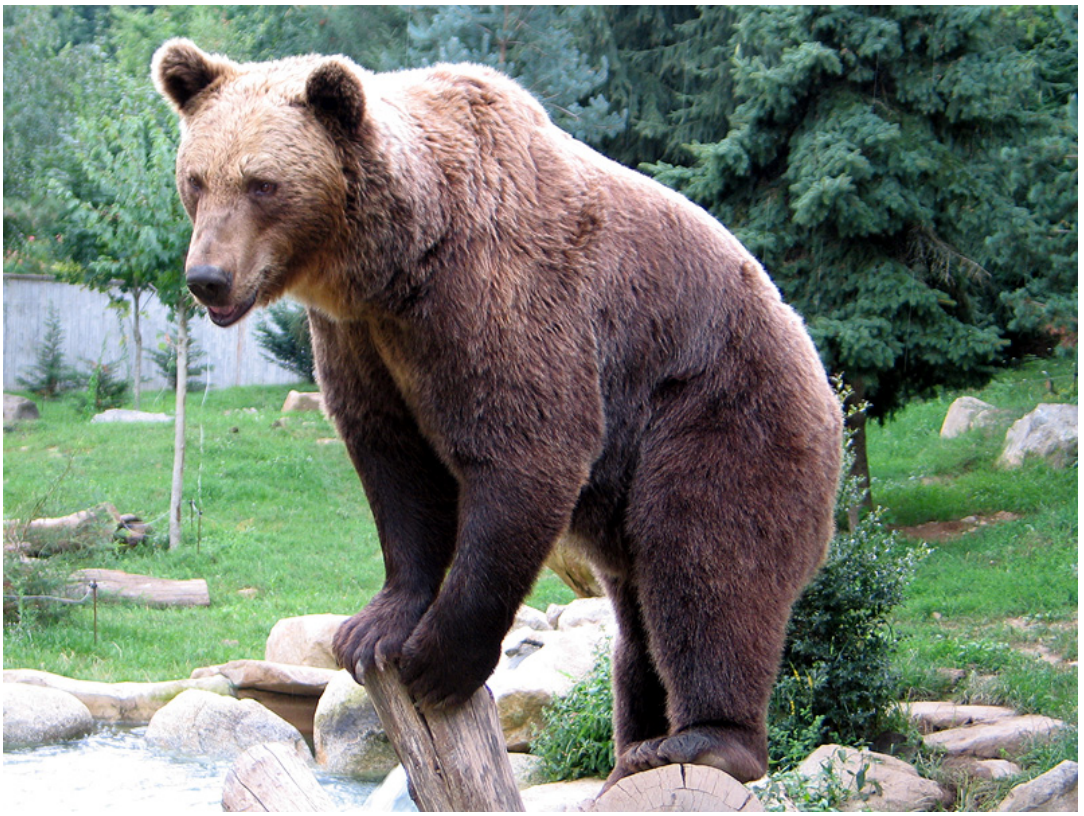
```
                    distance = calculerDistance(getRgb(tab[k]),
getRgb(rgb))
                    Si distance < min
                        min = distance
                        indice = k

                AjouterPixel(groupes[indice], Pixel(j, i, rgb))

        convergence = Vrai

        Pour i de 0 à Ng
            sumRed = 0
            sumGreen = 0
            sumBlue = 0
            pixels = groupes[i]

            Pour chaque pixel dans pixels
                rgb = getRgb(pixel.rgb)
                sumRed = sumRed + rgb[0]
                sumGreen = sumGreen + rgb[1]
                sumBlue = sumBlue + rgb[2]

            newRed = sumRed / taille(pixels)
            newGreen = sumGreen / taille(pixels)
            newBlue = sumBlue / taille(pixels)
            newCentroid = CréerRGB(newRed, newGreen, newBlue)

            Si newCentroid != tab[i]
                tab[i] = newCentroid
                convergence = Faux

        Si convergence
            carryOn = Faux

    nouvelleImage = CréerImage(largeur(image), hauteur(image))

    Pour i de 0 à hauteur(image)
        Pour j de 0 à largeur(image)
            rgb = ObtenirRGB(image, j, i)
            min = EntierMaximal()
            indice = 0

            Pour k de 0 à Ng
                distance = calculerDistance(getRgb(tab[k]), getRgb(rgb))
                Si distance < min
                    min = distance
```

```
                    indice = k

                DéfinirRGB(nouvelleImage, j, i, tab[indice])

    EnregistrerImage(nouvelleImage, "resultat.png")
    Fin = ObtenirTempsActuel()

    Afficher("Traitement terminé. Le résultat a été enregistré dans le
fichier resultat.png.")
    Afficher("Temps d'exécution : " + (Fin - debut) + " ms pour " + Ng +
" groupes.")
```

Nous avons exécuté cet algorithme avec différents nombres de groupes de couleurs et mesuré les temps d'exécution associés. Voici les résultats obtenus :

- Temps d'exécution : 474 ms pour 2 groupes.
- Temps d'exécution : 669 ms pour 3 groupes.
- Temps d'exécution : 1272 ms pour 4 groupes.
- Temps d'exécution : 793 ms pour 5 groupes.
- Temps d'exécution : 1746 ms pour 6 groupes.
- Temps d'exécution : 2271 ms pour 7 groupes.

Exemple de résultats :
Image de départ



pour 7 groupes

pour 4 groupes

pour 2 groupes

3. Ce que nous avons appris grâce à K-means et nos conclusions :
En utilisant l'algorithme de K-means, nous avons pu résoudre plus

efficacement le problème de réduction du nombre de couleurs dans une image. Nous avons constaté que l'algorithme produisait des résultats satisfaisants en termes de qualité visuelle, et les temps d'exécution étaient raisonnables même avec un nombre plus élevé de groupes de couleurs. En analysant les statistiques fournies, nous avons remarqué une augmentation générale du temps d'exécution à mesure que le nombre de groupes augmentait. Cela peut être attribué à la complexité croissante de la tâche de regroupement des couleurs en plusieurs clusters distincts.

En conclusion, ce projet nous a permis de comprendre l'importance de définir clairement le problème à résoudre et les données associées pour trouver une solution appropriée. Nous avons également réalisé qu'une fois que nous sommes en mesure de modéliser précisément un problème, nous pouvons appliquer différentes approches algorithmiques pour le résoudre. L'utilisation de l'algorithme de K-means nous a donné des résultats plus satisfaisants que notre idée initiale, démontrant ainsi l'importance de choisir le bon algorithme pour chaque situation.