CS-107 : Mini-projet 1
Où est Charlie ?

V. Bazzucchi, B. Chatelain, B. Jobstmann, J. Sam

Version 1.0

Table des matières

1 Présentation

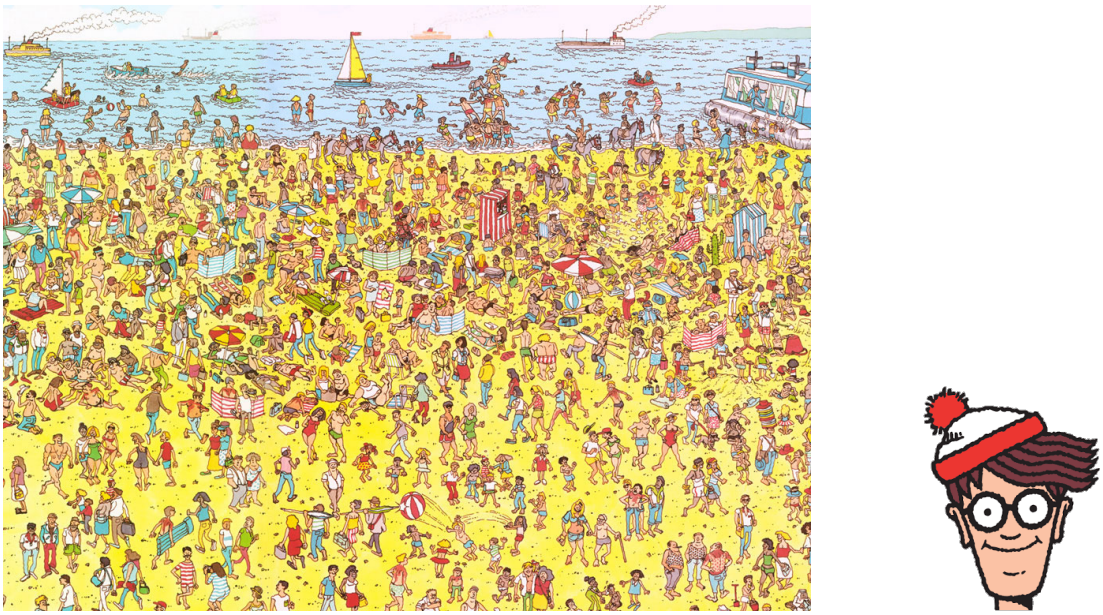

FIG. 1 : Trouvez Charlie (à droite) à la plage.

Le but du projet est de reconnaître un motif dans une image, par exemple Charlie à la plage (Figure 1).

Les algorithmes que nous allons utiliser sont à la base de la ”vision” par ordinateur : en reconnaissant des motifs dans une image ils permettent à l'ordinateur de ”voir”. La détection de visages, aujourd'hui présente dans de nombreuses applications sur ordinateurs et smart-phones, en est un exemple d'utilisation typique. De nombreuses autres applications existent évidemment, allant de la supervision de production par des ordinateurs industriels à la robotique et aux voitures autonomes.

Nous allons dans ce projet nous concentrer sur deux algorithmes qui ont une base commune : dans les deux cas, nous déplacerons une fenêtre de la taille du motif recherché sur l'image de fond. Pour chaque position, nous calculerons une valeur indiquant à quel point la portion d'image ressemble au motif. Ensuite nous utiliserons une couleur bien contrastée pour mettre en évidence les positions du motif dans l'image.

Le premier algorithme, traité dans la Section 4, calcule une *distance* entre le motif et la portion d'image étudiée. Cette approche a l'avantage de la simplicité mais peut échouer si le motif et l'image de fond ont des conditions de lumière différentes. Le second algorithme, basé sur la notion dite de ”corrélation croisée” (Section 6), permet de remédier à ce défaut. La corrélation croisée mesure la *similarité* entre deux images en calculant le produit de deux matrices.

Pour pouvoir appliquer ces algorithmes, nous avons besoin de transformer les images et d'en extraire les valeurs de pixels. Ce sujet est traité dans la Section 3.

Après avoir calculé la distance ou la similarité pour chaque position possible du motif dans l'image de fond, nous aurons besoin d'extraire la (les) position(s) qui nous intéresse(ent).

L'approche la plus simple consiste à retenir la (les) position(s) avec la distance la plus petite ou la similarité la plus grande. Ceci fait l'objet de la Section 5.

2 Structure et code fourni

2.1 Structure

Le projet est divisé en quatre étapes :

1. **Traitement d'images** – Représenter des couleurs (ce qui nous permettra de modéliser les pixels) et des images comme des tableaux de pixels.
2. **Distance entre images** – Calculer pour chaque position possible du motif dans l'image, la distance entre les valeurs des pixels de la portion d'image étudiée et du motif en produisant une matrice dont chaque élément est une distance.
3. **Localisation du motif** – Implémenter différentes stratégies pour localiser, à l'intérieur de la matrice produite lors de l'étape précédente, la (les) aire(s) la (les) plus proche(s) du motif.
4. **Corrélation croisée** – Calculer pour chaque position possible du motif dans l'image, la corrélation croisée entre les valeurs des pixels de la portion d'image étudiée et du motif en produisant une matrice dont chaque élément est une corrélation croisée. L'utilisation de la corrélation nous permettra de localiser le motif dans des images avec des conditions de lumières différentes.

Il y a quatre fichiers à compléter :

- `ImageProcessing.java` pour la partie 1 ;
- `DistanceBasedSearch.java` pour la partie 2 ;
- `Collector.java` pour la partie 3 ;
- `SimilarityBasedSearch.java` pour la partie 4.

Les entêtes des méthodes à implémenter sont fournies et **ne doivent pas être modifiées**.

Le fichier fourni `SignatureChecks.java` donne l'ensemble des signatures à ne pas changer. Il sert notamment d'outil de contrôle lors des soumissions. Il permettra de faire appel à toutes les méthodes requises **sans en tester le fonctionnement**[1]. Vérifiez que ce programme compile bien avant de soumettre.

La vérification du comportement correct de votre programme vous incombe. Néanmoins, pour vous aider dans vos tests, nous fournissons dans le fichier `Main.java`, des exemples simples vous montrant comment appeler et tester vos méthodes. **Ces tests ne sont bien sûr pas exhaustifs et vous êtes encouragés à les compléter et modulariser d'avantage si nécessaire.**[2]

Lors de la correction de votre mini-projet, nous utiliserons des tests automatisés, qui passeront des entrées générées aléatoirement aux différentes fonctions de votre programme. Il y aura aussi des tests vérifiant comment sont gérés les cas particuliers. Ainsi, il est important que votre programme traite correctement n'importe quel entrée valide.

[1] Cela permet de vérifier que vos signatures sont correctes et que votre projet ne sera pas rejeté à la soumission.

[2] Vous pouvez créer d'autres fichiers dans le même esprit que `Main.java`. Il ne vous est pas demandé d'utiliser des "tests unitaires" pour ceux d'entre vous qui savent ce que c'est (même si ce n'est pas interdit d'en faire).

Remarque Nous vous indiquerons, à chaque étape, les conditions de validité des entrées.

En dehors des méthodes imposées, libre à vous de définir toute méthode supplémentaire qui vous semble pertinente.

Modularisez et tentez de produire un code propre !

2.2 Code fourni

Les couleurs peuvent être modélisées au moyen de valeurs numériques (voir Section 8). Un pixel (point dans une image) peut, quant à lui, simplement être assimilé à une couleur.

Dans ce projet, nous représenterons donc les images comme des tableaux à deux dimensions de pixels (`int[][]` ou `double[][]`). La première dimension représente la ligne, la deuxième la colonne (en utilisant la même convention que vous voyez en algèbre linéaire pour les matrices). Les indexes des lignes et des colonnes commencent à 0 et indiquent le décalage à partir du pixel en haut à gauche de l'image, ainsi `image[1][2]` retourne la valeur qui se trouve à la deuxième ligne (à partir du haut) et à la troisième colonne (à partir de la gauche).

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	**(1,2)**	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)

FIG. 2 : indexes (ligne,colonne) d'une matrice ayant 3 lignes et 4 colonnes

La manipulation de fichiers et de fenêtres étant fastidieuse et trop avancée pour ce cours, une partie du code vous est donnée. En effet, le fichier `Helper.java` vous simplifie l'import, l'export et l'affichage d'images en Java :

- Les fonctions `read(String path)` et `write(String path, int[][] array)` permettent de lire et écrire des images dans divers formats (JPG, PNG, BMP...).
- La méthode `show(String title, int[][] array)` affiche une image à l'écran, et attend que l'utilisateur ferme la fenêtre pour continuer l'exécution.
- La méthode `drawBox(int row, int col, int width, int height, int[][] dst)` (qui admet des surcharges) dessine un rectangle rouge sur l'image `dst`. Le rectangle est caractérisé par son point haut gauche, dont les coordonnées sont `row, col`, par sa largeur `width` et sa hauteur `height`.

2.3 Tests et assertions

Important : il vous incombe de **vérifier à chaque étape que vous produisez bel et bien des données correctes** avant de passer à l'étape suivante qui va utiliser ces données.

Pour ce qui est de la gestion des cas d'erreurs, il est d'usage de tester les paramètres d'entrée des fonctions ; e.g., vérifier qu'un tableau n'est pas nul, et/ou est de la bonne dimension, etc. Ces tests facilitent généralement le debuggage, et vous aident à raisonner quant au comportement d'une fonction.

Nous partirons de l'hypothèse que le programme doit s'arrêter si les paramètres d'entrée sont invalides.

Pour chaque fonction que vous compléterez, **veuillez indiquer en commentaire vos hypothèses** ("Requirement : xxx") sous lesquelles cette fonction se déroule correctement. Puis, **ajoutez les tests requis** ("Assert : xxx") pour vérifier que les inputs soient bien conformes à vos hypothèses.

Example de validations des inputs sur une simple fonction de division

```
/**
 * Computes the result of a / b
 * @param a  : the nominator. Requirement : none, any integer
 * @param b  : the denominator. Requirement : b cannot be 0
 * @return an double value corresponding to a/b
 */
public static double divide(int a, int b) {
  assert b != 0;
  double result = a
  return result/b;
}
```

Assertions. Tout d'abord, il faut activer les assertions en lançant le programme avec l'option "-ea" [Lien cliquable ici]. Puis, une assertion s'écrit sous la forme

```
assert expr;
```

avec `expr` une expression booléenne. Si l'expression est fausse, alors le programme lance une erreur et s'arrête, sinon il continue normalement.

Par exemple, pour vérifier que l'image analysée `image` n'est pas null, vous pouvez écrire

```
assert image != null;
```

au début de la fonction manipulant l'image en question.

2.4 Divers

La description des étapes est volontairement courte et concise. Le but est que vous trouviez par vous même les solutions aux problèmes impliqués.

Un certain nombre d'images sont fournies pour tester les différentes parties de votre projet. **Vous pouvez tester avec vos propres images, mais le format supporté est uniquement le `.png`**.

Enfin, la mise en oeuvre du mini-projet implique de connaître certains concepts de base : ce qu'est un pixel, une couleur ou une représentation binaire. Une partie de ces éléments vous aura été expliquée en cours. Certaines notions sont également décrites dans les compléments en fin de document.

Commencez par lire ces compléments dans les grandes lignes pour appréhender les points importants impliqués.

3 Traitement d'images

Dans cette première partie, vous développerez les outils fondamentaux pour manipuler des images. Une image étant un tableau à deux dimensions de couleurs RGB (voir les compléments théoriques, Section 8), vous aurez besoin d'accéder à chaque composante (R, G ou B) des pixels et de convertir un pixel donné en niveau de gris.

Ceci permettra de faire des manipulations sur des images entières comme la conversion d'une image RGB en niveaux de gris et vice-versa.

Votre première tâche consiste donc à compléter le fichier `ImageProcessing.java` pour réaliser ces traitements. À noter qu'il y a perte d'information lors de la conversion en gris ou en noir et blanc, rendant ces opérations irréversibles.

Voici un exemple de code qui lit une image depuis le disque et qui extrait les différentes composantes d'un pixel et qui convertit des images

```
// On charge une image depuis le disque
int[][] rgbImage = Helper.read("mickey-mouse.jpg");

// Affiche la valeur du pixel en x=50, y=100
// l'entier affiché est un code de couleur
// dans la representation RGB
System.out.println(rgbImage[100][50]);

// Lit les niveaux de rouge, vert et bleu du pixel en x=50, y=100
int red = ImageProcessing.getRed(rgbImage[100][50]);
int green = ImageProcessing.getGreen(rgbImage[100][50]);
int blue = ImageProcessing.getBlue(rgbImage[100][50]);

// Calcule le niveau de gris en x=50, y=100
double grayPixel = ImageProcessing.getGray(rgbImage[100][50]);

// Convertit l'image en niveau de gris
double[][] grayLevelsImage = ImageProcessing.toGray(rgbImage);

// Convertit l'image en niveau de gris en une image en "couleurs"
   (au format RGB - l'image reste grise car l'information sur la
   couleur a été perdue)
int[][] rgbImage2 = ImageProcessing.toRGB(grayLevelsImage);
```

Cette partie sera réalisée au moyen des opérateurs sur les bits. **Vous ne devrez pas avoir recours à des classes non vues dans le cadre du cours**. Pour toutes les parties suivantes, vous êtes libre d'utiliser tout outillage de votre choix vous semblant approprié.

3.1 Manipulation d'un pixel

Commencez par implémenter les fonctions `getRed(int RGB)`, `getGreen(int RGB)` et `getBlue(int RGB)` qui retournent séparément les différentes composantes (rouge, verte et bleue respective-

ment) d'une couleur (et donc d'un pixel). Cette couleur est donnée sous la forme d'un entier, et le résultat est retourné en tant qu'entier compris entre 0 et 255.

Ces fonctions vous permettront d'implémenter ensuite `getGray(int RGB)` qui calcule et retourne la moyenne des composantes d'une couleur, toujours entre 0 et 255.

L'opération inverse doit être effectuée par différentes surcharges de la méthode `getRGB` :

- `getRGB(int red, int green, int blue)` encode dans un entier les trois composantes de couleur (voir l'exemple donné dans la dernière page des compléments du Section 8).
- `getRGB(double gray)` fait de même à partir d'un niveau de gris ; les trois composantes seront alors identiques et vaudront `gray` (à arrondir).

Les signatures de méthodes sont fournies.

Attention Votre code doit gérer les valeurs qui ne sont pas entre 0 et 255 : un `int` reçu en paramètre qui ne serait pas entre 0 et 255 doit être limité à ces valeurs (ramené à 0 s'il est plus petit que 0 et ramené à 255 s'il est plus grand que 255). Lisez les commentaires descriptifs des méthodes tels que fournis pour savoir quand procéder à ces vérifications/traitements.

```
int color = 0b11110000_00001111_01010101;
// -> 15732565 (0xF00F55)

// On extrait les differentes valeurs
getRed(color);                // -> 240 (0xF0)
getGreen(color);              // -> 15 (0x0F)
getBlue(color);               // -> 85 (0x55)
double gray = getGray(color); // -> 113.33333

// On encode des couleurs
getRGB(0, 0, 255);    // -> 255 (0x0000ff)
getRGB(127.0);            // -> 8355711 (0x7f7f7f)

// Notez que les composantes sont contraintes dans [0, 255]
getRGB(-175, 0, 255); // -> 255 (0x0000ff)
getRGB(-255);         // -> 0 (0x000000)
```

3.2 Manipulation d'une image complète

En utilisant les fonctions précédentes, implémentez

- `toGray(int[][] image)` qui convertit une image donnée en format RGB vers une image en niveaux de gris (`double[][]`) ;
- `toRGB(double[][] gray)` qui convertit dans l'autre sens une image en niveaux de gris en une image au format RGB

Pour cela, vous devez créer un tableau de la bonne taille, et utiliser `getGray` ou `getRGB` pour chaque pixel.

```java
// Une image couleur 2x2
int[][] image = {
  {0x20c0ff, 0x123456},
  {0xffffff, 0x000000}
};

// On la convertit en gris
double[][] gray = toGray(image);
// -> {
//      {159.66666, 52.0},
//      {255.0, 0.0}
//    };

// On revient vers du RGB
int[][] back = toRGB(gray);
// -> {
//      {10526880, 3421236},
//      {0x343434, 0}
//    };
```

3.3 Tests

Testez les différentes fonctionnalités que vous venez de coder en vous inspirant du fichier fourni `Main.java`

4 Distance entre images

Dans cette partie nous allons comparer une portion de l'image de fond avec le motif recherché. Pour cela nous allons calculer pour chaque position possible du motif une mesure de la distance entre les pixels du motif et les pixels du fond. La composante de transparence des pixels va délibérément être ignorée pour commencer.

Nous allons mesurer la distance en utilisant l'**erreur moyenne absolue** (mean absolute error, MAE).

4.1 Calcul de l'Erreur Absolue Moyenne

Si on note l'ensemble des composantes de chaque pixel comme $C = \{ROUGE, VERT, BLEU\}$, on peut définir l'erreur absolue entre la valeur M du pixel du motif et la valeur I du pixel du fond :

$$EA(M, I) = \frac{\sum_{c \in C} |M_c - I_c|}{|C|}$$

Implémentez ce calcul dans la méthode

```
double pixelAbsoluteError(int patternPixel, int imagePixel)
```

dans le fichier `DistanceBasedSearch.java`. Les deux arguments de cette méthodes sont des entiers représentant les valeurs RGB des pixels du motif et de l'image de fond.

La fonction EA nous permet de définir la fonction EAM (erreur absolue moyenne) calculant l'erreur absolue moyenne entre deux images M,I de mêmes dimensions :

$$EAM(M, I) = \frac{1}{d} \sum_{(i,j) \in dim(P)} EA(M(i, j), I(i, j))$$

où d est le nombre de pixels de M (ou de I vu qu'elles ont même taille) et $M(i, j)$ est la valeur du pixel à la position i, j de l'image M.

4.2 Calcul de la distance pour une position possible du motif

Supposons maintenant que le motif soit superposé à la portion de l'image de fond (de même taille que le motif) dont le coin supérieur gauche a pour indices (row, col) ; où row est l'indice de la ligne et col l'indice de la colonne dans l'image.

Il est possible de modifier la fonction EAM pour accepter les coordonnées (row, col) du point haut gauche de la portion d'image de fond à comparer avec le motif :

$$EAM(M, I, row, col) = \frac{1}{d} \sum_{(i,j) \in dim(M)} EA(M(i, j), I(row + i, col + j))$$

Implémentez la méthode se chargeant de ce calcul :

```
double meanAbsoluteError(int row, int col, int[][] pattern, int[][]
    image)
```

dans le fichier `DistanceBasedSearch.java`.

Vérification des arguments Un motif ou une image seront considérés comme valides s'ils ont au moins un pixel. Les indices *row* et *col* devront être tels que le motif, si son coin supérieur gauche est positionné à cet endroit, puisse être entièrement contenu dans l'image.

4.3 Création de la matrice de distances

Nous allons maintenant déplacer le motif sur l'image de fond en calculant pour chaque position possible du motif l'erreur absolue moyenne. Nous créerons comme résultat une matrice dont **chaque élément** (row, col) **est l'erreur absolue moyenne entre le motif et la portion d'image de même taille ayant pour coin supérieur gauche les indices** (row, col)**.**

Définissez pour cela la méthode

```
double[][] distanceMatrix(int[][] pattern, int[][] image)
```

dans le fichier `DistanceBasedSearch`, qui initialise et remplit la matrice de distances.

Vérification des arguments Un motif ou une image seront considérés comme valides s'ils ont au moins un pixel. Le motif est valide s'il peut au moins une fois être entièrement contenu dans l'image.

Attention ! Pour une première implémentation de cette méthode vous déplacerez le motif sur l'image de fond en choisissant comme point haut gauche les pixels $[0, W - w] \times [0, H - h]$ (W et w étant les largeurs de l'image de fond et du motif et H et h étant les hauteurs de l'image de fond et du motif.
Des stratégies alternatives sont présentées dans la section "Pour aller plus loin" : si le motif était coupé par le bord de l'image, cette approche ne le détecterait pas, vu que nous nous arrêtons avant d'arriver à la première occurrence de son pixel haut gauche. Pour chercher le motif en proximité des bords nous présenterons brièvement deux techniques : le *mirroring* et le *wrapping*.

4.4 Visualisation de la matrice de distances

A des fin de "debugging", nous pouvons visualiser la matrice de distances en interprétant chaque distance comme un niveau de gris : une grande valeur indique une grande distance.

Pour visualiser la matrice de distance, implémentez le corps de la méthode

```
int[][] matrixToRGBImage(double[][] matrix, double min, double max)
```

dans le fichier `ImageProcessing.java`. En plus de la matrice, cette méthode reçoit la valeur maximale et minimale que chaque élément de celle-ci peut avoir.

L'idée est de fournir à la méthode `getRGB` définie dans `ImageProcessing.java` des valeurs dans l'intervalle $[0, 255]$. En connaissant les valeurs minimale et maximale m, M des éléments de la matrice, nous pouvons obtenir le niveau des gris g correspondant à la valeur x en calculant :

$$g = \frac{x - m}{(M - m)} \cdot 255$$

`matrixToRGBImage` produit un tableau à deux dimensions de valeurs RGB (`int[][]`) que vous pouvez visualiser à l'aide de la méthode utilitaire `Helper.show` ou sauvegarder en utilisant `Helper.write`.

Question : Quelles sont les valeurs de M et m pour la matrice de distances dans le cas général ? Est-ce toujours le cas ?

Vérification des arguments une matrice à convertir en RGB sera considérée comme valide si elle contient au moins une ligne et une colonne. Les valeurs de `min` et `max` seront considérées comme valides. Leur choix peut en effet dépendre de situations spécifiques. Par exemple, si une matrice de distances contient une fois la valeur 1 et que toutes les (nombreuses) autres valeurs sont comprises entre 0 et 0.1, cela peut avoir du sens de prendre 0.1 comme maximum possible au lieu de 1. Si `min` est supérieur à `max`, vous pouvez simplement intervertir les deux valeurs.

A ce stade vous considérerez donc simplement `min` et `max` comme des paramètres d'ajustement des niveaux de gris : plus on couvre une plage large, plus on garantit que toutes les valeurs de la matrice de distances seront prises en compte mais au risque de perdre en degré de contraste.

4.5 Tests

Testez les différentes fonctionnalités que vous venez de coder en vous inspirant du fichier fourni `Main.java`.

Vous pouvez utiliser `matrixToRGBImage` à des fins de "debugging".

Par exemple, dans `testDistanceBasedSearch`, après la calcul de la matrice de distances, vous pouvez faire afficher la matrice de distance en niveaux de gris :

```
Helper.show(ImageProcessing.matrixToRGBImage(distance, 0, 255),
    "Distance");
```

Vous devriez voir apparaître les taches les plus foncées aux deux endroits qui correspondent au coin supérieur gauche du motif `onions.png`.

5 Localisation du motif

Pour trouver la position du motif, nous allons étudier la matrice de distances. En particulier nous chercherons les coordonnées, dans la matrice, de l'élément le plus petit : cela équivaut à chercher, dans l'image de fond, le point haut-gauche du rectangle de mêmes dimensions que le motif qui minimise la distance de celui-ci.

Dans un premier temps, nous considérerons que le motif n'apparaît qu'une seule fois dans l'image et ensuite nous supprimerons cette contrainte pour chercher n apparitions du motif dans l'image.

Vérification des arguments pour toute cette partie, une image sera considérée comme valide si elle a au moins un pixel.

5.1 Recherche d'un extremum

On peut considérer notre matrice de distances comme une collection **non ordonnée** de valeurs. La recherche d'un extremum (minimum or maximum) dans une collection est un problème classique d'algorithmique. Étant donné que les valeurs ne sont pas ordonnées, nous sommes obligés de parcourir toute la matrice pour trouver le maximum ou le minimum, ce qui impose une complexité minimale de $\mathcal{O}(d)$ (d étant le nombre d'éléments de la matrice).

La stratégie classique consiste à stocker la meilleure valeur (maximum ou minimum) dans une variable et de comparer le contenu de cette variable avec les autres valeurs, en remplaçant le contenu de la variable par la valeur courante si celle-ci est *meilleure* (plus grande dans le cas de recherche du maximum ou plus petite dans le cas de recherche du minimum).

Implémentez la recherche d'un extremum dans le corps de la méthode

```
findBest(double[][] matrix, boolean smallestFirst)
```

dans le fichier `Collector.java`. Le dernier argument de la méthode est vrai si on recherche le minimum et faux si on cherche le maximum. La méthode retourne un tableau de taille 2 dont le premier élément est l'index de la colonne et le deuxième élément est l'index de la ligne (coordonnées row, col) de l'extremum dans la matrice de distances.

Vous pourriez avoir besoin des $-\infty$ et $+\infty$ pour représenter la valeur initiale de la valeur la plus grande ou la plus petite. Comme nous l'avons indiqué précédemment, Java fournit les variables `Double.NEGATIVE_INFINITY` et `Double.POSITIVE_INFINITY`.

5.2 Recherche des n meilleures valeurs

Pour chercher les coordonnées des n meilleures valeurs nous présenterons deux possibles stratégies : une plus simple mais plus coûteuse ici et une deuxième, plus efficaces mais plus complexe dans la section 7 ("Pour aller plus loin").

Comme vous pouvez l'imaginer, une stratégie très simple consiste à utiliser n fois la technique expliquée ci-dessus **en éliminant chaque valeur choisie** de la matrice. La complexité de cette

technique devient alors $\mathcal{O}(n \times d)$ (avec d le nombre d'éléments de la matrice).

Pour "éliminer" les éléments déjà choisis nous pouvons les remplacer par des valeurs spéciales, par exemple $\pm\infty$.

Implémentez cette stratégie dans le corps de la méthode

```
findNBest(int n, double[][] matrix, boolean smallestFirst)
```

dans le fichier`Collector.java` qui retournera un tableau $n \times 2$ dont chaque ligne contient les coordonnées row, col d'un élément. Le deuxième argument de la méthode est vrai si on souhaite chercher les coordonnées des n plus petites valeurs de la matrice et faux si on est intéressé aux n plus grandes valeurs de celle-ci.

Attention ! Le contenu de la matrice reçue par la méthode **ne doit pas être modifié**, pensez donc à travailler sur une copie.

5.3 Première utilisation du programme

Nous avons désormais tous les outils nécessaires pour effectuer notre première recherche de motif dans une image. Pour faire cela, créez un programme principal `Program` dans le dossier `main`.

Ce programme devra :

1. Lire l'image de fond et l'image motif à l'aide des méthodes `Helper.read`
2. Obtenir la matrice de distances en utilisant `DistanceBasedSearch.distanceMatrix`
3. Localiser les portions d'images minimisant la distance avec `Collector.findBest` ou `Collector.findNBest`
4. Dessiner les rectangles autour de ces portions avec la méthode utilitaire `Helper.drawBox`
5. Afficher ou enregistrer l'image ainsi obtenue à l'aide de `Helper.show` ou `Helper.write`

Le fichier fourni `Main` peut naturellement vous servir de source d'inspiration.

6 Corrélation croisée

Nous allons maintenant introduire une nouvelle grandeur pour comparer le motif à une portion d'image. Cette technique nous permet de développer une détection du motif plus robuste qui sera, par exemple, résistante au changement de condition de lumière entre motif et image.

À partir d'une image et d'un motif en niveaux de gris, nous allons calculer pour chaque position possible du motif dans l'image ce que l'on appelle la " **corrélation croisée**" entre la portion d'image et le motif. Cette grandeur mesure la similarité entre la portion d'image et le motif, ainsi nous chercherons les portions d'image qui *maximisent la corrélation*.

Vérification des arguments pour toute cette partie, une image sera considérée comme valide si elle a au moins un pixel.

6.1 Définition du coefficient de corrélation

Il est possible d'estimer le coefficient de corrélation d'un échantillon $\{(x_i, y_i) \mid 1 \leq i \leq n\}$ en calculant :

$$r(x,y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}$$

où $\bar{x}$ est la moyenne des x_i et $\bar{y}$ est la moyenne des y_i, c'est-à-dire

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

On remarque aussi que $-1 \leq r \leq 1$.

Cet indicateur nous renseigne sur la *dépendance linéaire* entre deux variables. Pour plus d'informations, voir https://fr.wikipedia.org/wiki/Corr%C3%A9lation_(statistiques).

Dans le cadre de ce projet nous nous permettons de simplifier en utilisant les niveaux des gris des pixels comme échantillons et en affirmant que **plus le coefficient de corrélation est grand, plus les deux échantillons sont similaires**.

6.2 Calcul de la corrélation croisée entre le motif et une portion d'image

Nous avons besoin de modifier la formule précédente pour calculer le coefficient de corrélation en considérant seulement une portion de l'image.

Si on note r, c les coordonnées du point haut gauche de la portion d'image à comparer avec le motif, le calcul devient alors

$$NCC(M, I, r, c) = \frac{\sum_{(i,j)\in dim(M)} [I(r+i, c+j) - \bar{W}] \times [M(i,j) - \bar{M}]}{\sum_{(i,j)\in dim(M)} [I(r+i, c+j) - \bar{W}]^2 \times \sum_{(i,j)\in dim(M)} [M(i,j) - \bar{M}]^2}$$

où $\bar{W}$ est la moyenne des valeurs des niveaux de gris de la portion d'image.

Implémentez ce calcul dans la méthode

```
public static double normalizedCrossCorrelation(int row, int col,
                                double[][] pattern, double[][] image)
```

dans le fichier `SimilarityBasedSearch.java`

La validité des arguments sera traitée selon les mêmes critères que lors du calcul de la matrices de distances.

Modularisez votre code en définissant des méthodes auxiliaires. Dans ce but, le calcul de la moyenne d'une portion d'image doit être défini dans une méthode

```
static double windowMean(double[][] matrix, int row, int col,
                                    int width, int height)
```

Attention Un cas limite sur lequel nous portons votre attention est celui où la portion d'image ou le motif sont uniformes (toutes les valeurs sont égales). Cela se traduit par un dénominateur nul. Dans ce cas, le calcul **doit retourner -1**. Il faudrait en outre vérifier que les indexes `row, col` ne provoquent pas d'erreurs d'accès aux tableaux.

Attention Cette méthode sera invoquée pour chaque position possible du motif dans l'image. Essayez donc d'en minimiser l'impact sur les performances en réduisant le nombre de boucles et en calculant toutes les constantes en dehors des boucles. Veuillez à ne pas modifier la signature de la méthode et produire une code lisible. Dans le cadre de ce projet, **la lisibilité est plus importante que les performances**.

6.3 Calcul de la corrélation pour toutes les positions possibles du motif

Comme dans la Section 4, nous allons créer une matrice dont **chaque élément** (r, c) **est le coefficient de corrélation entre le motif et la portion d'image obtenue en considérant les pixels dans le rectangle ayant comme vertex haut gauche le pixel** (r, c) **du fond et ayant les mêmes dimensions du motif.**

Autrement dit, on définit l'élément r, c de la matrice de corrélation comme $(C)_{r,c} = NCC(M, I, r, c)$

Implémentez la création de la matrice de corrélation dans la méthode :

```
public static double[][] similarityMatrix(double[][] pattern,
    double[][] image)
```

dans `SimilarityBasedSearch.java`

Les tests sur les arguments à effectuer sont les mêmes que dans `distanceMatrix`.

6.4 Utilisation du programme

La matrice de similarité peut être utilisée de façon analogue à la matrice de distances. Il y a cependant deux différences très importantes :

1. Alors que la distance doit être minimisée, la corrélation doit être maximisée pour trouver la position de la portion d'image la plus similaire au motif.
2. La méthode `similarityMatrix` accepte deux images en niveaux des gris. Utilisez pour cela `ImageProcessing.toGray`.

Vous êtes désormais en mesure de chercher la position d'un motif à l'intérieur d'une image en utilisant la matrice de corrélation.

Compléter `Program` de sorte à ce qu'il utilise la matrice de similarité pour effectuer différentes recherche de motifs dans des images ; dont le fameux Charlie à la plage. Inspirez vous du `Main` fourni, mais pensez à modulariser proprement.

Remarques :

- Selon les consignes données, l'accent a été mis sur la lisibilité plutôt que les performances. Certains traitements peuvent donc être relativement longs. Il est considéré comme normal que trouver Charlie à la plage prenne l'ordre de quelques minutes (~2).
- Les images fournies `image.png`, `image-dark.png` et `image-light.png` permettent de voir la différence existant entre l'approche basée sur le calcul de distances et celle basée sur la corrélation croisée. La méthode basée sur le calcul de distance ne parvient à trouver `pattern.png` que dans `image.png` alors que la méthode par corrélation croisée arrive aussi à trouver le motif dans des variantes plus claire ou plus foncée de l'image.

7 Pour aller plus loin

Nous vous présentons ici des extensions possibles pour ce premier mini-projet.

Faites attention lors de l'implémentation de ces bonus à ne pas modifier les signatures des fonctions : **l'utilisation du programme ne doit pas changer**.

Il est impératif de coder les extensions dans des méthodes séparées de celles que vous avez déjà codées.

7.1 *Wrapping* et *Mirroring* des bords des images

Difficulté : Basse

Attention ! L'implémentation de ce bonus directement dans les méthodes existantes causera l'échec de certains tests automatiques.

Créez donc des *nouvelles* méthodes, éventuellement en exploitant la surcharge des noms. Par exemple `distanceMatrix(int[][] pattern, int[][] image, String strategy)` où `strategy` peut être *`"wrap"`* ou *`"mirror"`*.

L'idée est de ne pas déplacer le motif de $(0,0)$ à $(W-w, H-h)$ mais de $(0,0)$ à (W,H). Le problème se pose donc quand le point haut gauche du motif se trouve en (r,c) avec $r \geq H-h$ ou $c \geq W-w$ car on essaiera d'accéder à des pixels (r',c') avec $r' \geq H$ ou $c' \geq W$. Supposons que l'on travaille en une seule dimension : nous avons n valeurs dans un tableau et nous essayons d'accéder à l'élément i avec $i \geq n$.

indexes	0	1	2	...	$n-2$	$n-1$
valeurs	5	180	9	..	15	-3

Les deux solutions que vous nous proposons consistent à calculer i' à partir de $i \geq n$ tel que $i' \in [0, n-1]$:

- **wrapping** : i' est le reste de la division entière de i par n : $i' = i \bmod n$.

indexes	0	1	2	...	$n-2$	$n-1$	0	...
valeurs	5	180	9	...	15	-3	5	...

- **mirroring** : $i' = n-2-(i \bmod n)$ (seulement si $i \geq n$). Cela équivaut à refléter les éléments en utilisant le dernier comme point de symétrie.

indexes	0	1	2	...	$n-2$	$n-1$	$n-2$	...
valeurs	5	180	9	...	15	-3	15	...

Vous pouvez généraliser ces formules en d dimensions.

7.2 Tri des valeurs de la matrice

Difficulté : Moyenne

Comme nous l'avons mentionné, la recherche des n meilleurs valeurs implémentée dans `findNBest` n'est pas optimisée. En effet son coût est de $\mathcal{O}(n \cdot d)$. Nous pouvons le réduire à $\mathcal{O}(d \log d)$ en **triant les coordonnées des éléments de la matrice par ordre croissant/décroissant des éléments**.

Par exemple, à partir de la matrice :

8.3	5	7.1
2.0	3.3	-1

on veut produire le tableau dont chaque ligne contient les coordonnées row, col des éléments, triées par valeur des éléments.

1	2
1	0
1	1
0	1
0	2
0	0

Après avoir trié les coordonnées, trouver les coordonnées des n meilleurs éléments équivaut à retourner les n premières (ou dernières) lignes de ce tableau.

Nous vous proposons ici une version récursive et adaptée de l'algorithme de tri **quicksort**.

Implémentation Avant de commencer le tri, créez un `ArrayList` de points. On définit un point comme un `int[]` de taille 2. Vous devez donc obtenir un tableau dynamique de points, un point par élément de la matrice, chaque point ayant les coordonnées x, y d'un élément.

On est alors prêt pour notre algorithme de tri :

1. Si le tableau dynamique à trier est vide, retournez-le.
2. Trouvez le *pivot*. Pour simplifier nous allons considérer que le premier élément du tableau dynamique reçu est le pivot.
3. Créez trois nouveau tableaux dynamiques. Ils contiendront respectivement les coordonnées des valeurs plus petites, égales et plus grandes que le pivot.
4. Invoquez la fonction de tri sur les trois tableaux ainsi générés. (appels récursif)
5. Fusionnez les trois tableaux, désormais triés, dans le bon ordre à l'aide de la méthode `addAll` des tableaux dynamiques.
6. returnez le resultat

Il faudra ensuite transformer le `ArrayList<int[]>` en `int[][]` en prenant les n premières ou dernières valeurs (en utilisant par exemple la méthode `.subList` des `ArrayList`.

7.3 Utiliser la transparence d'images

Difficulté : Élevée

Comme expliqué dans le complément théorique, les images aux format `png` sont composées de pixels au format **ARGB**, c'est-à-dire qu'il y a une quatrième composante α qui représente l'opacité du pixel : une valeur nulle indique que le pixel est complètement visible alors que `0xFF` indique que le pixel est invisible.

Ce dernier bonus consiste à modifier les deux algorithmes de recherche du motif pour tenir en compte le niveau de transparence des pixel du motif. Par exemple dans le calcul de l'erreur absolue moyenne du premier algorithme, il ne faudra considérer que les contributions des pixels qui ne sont pas invisibles.

Nous avons marqué cet extension comme à difficulté élevée car elle comporte de très nombreuses modifications au code.

Veillez à travailler dans de nouvelles méthodes et non pas dans celles existantes.

8 Complément théorique – Couleurs, pixels et binaire

Une image est formée par une infinité de rayons de lumière qui viennent heurter notre rétine. Toutefois, nos yeux et notre cerveau ne sont pas capables de percevoir et encore moins de gérer une quantité infinie d'information. C'est un problème récurrent dans de nombreux domaines, et les ordinateurs n'y font pas exception. La solution consiste à approximer l'image par une quantité finie de valeurs. On parle donc d'approximation *discrète* d'un phénomène *continu*.

Il est fort probable que vous connaissiez la notion de *pixel* (littéralement *picture element*), qui représente un point de l'écran. Dans la même idée, une image dite *matricielle* est composée d'une grille à deux dimensions de pixels. En d'autres termes, il s'agit d'une matrice de couleurs. Bien qu'il existe d'autres formats, chacun ayant leurs propres avantages et faiblesses, nous nous contenterons de cette représentation pour cet exercice. Il s'agit en effet du modèle le plus simple et efficace pour le traitement d'image.

Quant aux couleurs elles-mêmes, il est aussi nécessaire de les discrétiser. Les couleurs s'obtiennent par synthèse d'un nombre limité de couleurs primaires.

La peinture utilise un modèle de synthèse dit soustractif[3]. C'est avec un tel modèle que l'on peut dire que mélanger du jaune et du bleu donne du vert. A contrario, les modèles dit additifs combinent les *lumières* de plusieurs sources colorées dans le but d'obtenir une lumière colorée.

La synthèse additive[Lien] utilise généralement trois lumières colorées : une rouge, une verte et une bleue (RGB en anglais pour red, green, blue). Le mélange de ces trois lumières colorées en proportions égales donne la lumière blanche. L'absence de lumière donne du noir.

Les LEDs des écrans utilisent le procédé de la synthèse additive. Nous allons donc utiliser RGB[Lien] pour représenter nos couleurs, bien qu'il soit tout à fait possible d'utiliser d'autres modèles et de les combiner.

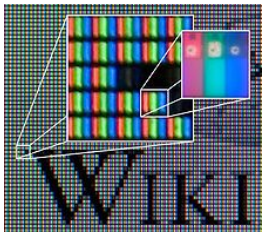

Fig. 3 : Pixel d'un écran

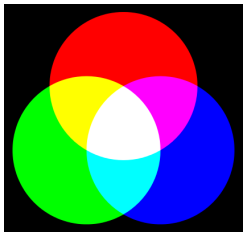

Fig. 4 : Modèle additif

Fig. 5 : Modèle soustractif

Pour être précis, une couleur est représentée sur 3 *bytes* (soit 24 *bits*). Cela permet $2^8 = 256$ niveaux de rouge, vert et bleu, allant de 0 à 255. Ainsi, on peut représenter $256^3 = 16\,777\,216$ couleurs différentes. L'oeil humain étant capable de percevoir dans l'ordre de 300 000 nuances seulement, il n'est donc pas nécessaire d'avoir plus de précision ! L'unité de base de la majorité des processeurs étant un entier 32 bits, il est possible d'y stocker les 3 composantes. Il nous reste même 8 bits inutilisés, représentant parfois la transparence que l'on nomme *alpha* (d'où le modèle *ARGB*).

[3] Soustractif, car plus on ajoute des couleurs, **moins** il y a de luminosité

Coleurs	**Inutilisé/Alpha**	**Rouge**	**Vert**	**Bleu**
Chiffres binaires	00000000	00100000	11000000	11111111
Chiffres décimaux	0	32	192	255

Le binaire a évidemment un rôle majeur à jouer en programmation. Même si vous ne les avez probablement pas encore rencontrées, il existe des opérations qui permettent de manipuler les bits des entiers. Toutes les opérations booléennes ont leurs équivalents binaires.

```
// Notre couleur en binaire
int x = 0b00000000_00100000_11000000_11111111;
// -> 2146559 (0x20c0ff)

// Décale de 8 bits vers la droite
int y = x >> 8; // -> 0b00000000_00100000_11000000

// ET binaire, ce qui a pour effet de ne garder que les 8
// premiers bits
int z = y & 0b11111111; // -> 0b11000000
// On a bien récupéré notre composante verte à 192
// on aurait aussi pu écrire : int z = y & 0xff;
```