CS-107 : Mini-Projet 1
Reconnaissance d'empreintes digitales

Yanis De Busschere, Barbara Jobstmann, Jamila Sam

-

version 1.1

Table des matières

1 Présentation **3**

1.1 Structure d'une empreinte digitale . . . 3

1.2 Comment extraire les minuties ? . . . 4

1.3 Comment comparer des empreintes digitales ? . . . 5

2 Structure et code fourni **6**

2.1 Structure . . . 6

2.2 Code fourni . . . 6

2.3 Tests . . . 7

3 Tâche 1 : Squelettisation **10**

3.1 Méthodes préliminaires . . . 10

3.1.1 Méthode `getNeighbours` . . . 10

3.1.2 Méthode `blackNeighbours` . . . 11

3.1.3 Méthode `transitions` . . . 12

3.1.4 Méthode `identical` . . . 12

3.2 Méthodes principales . . . 12

3.3 Tests . . . 14

4 Tâche 2 : Localisation et calcul de l'orientation **14**

4.1 Fonction `connectedPixels` . . . 15

4.2 Fonction `computeSlope` . . . 16

4.3 Fonction `computeAngle` . . . 20

4.4 Fonction `computeOrientation` . . . 22

4.5 Fonction `extract` . . . 22
4.6 Tests . . . 23

5 Tâche 3 : Comparaison **23**
5.1 Présentation de l'algorithme . . . 24
5.2 Méthode `applyTransformation` . . . 25
5.3 Méthode `matchingMinutiaeCount` . . . 26
5.4 Méthode `match` . . . 26
5.5 Tests . . . 27

6 Complément théorique **27**
6.1 Représentation ARGB . . . 27
6.2 Format binaire et nuance de gris . . . 28

Ce document utilise des couleurs et contient des liens cliquables. Il est préférable de le visualiser en format numérique.

1 Présentation

Que se soit pour une enquête policière ou pour déverrouiller nos téléphones, les logiciels traitant les empreintes doivent pouvoir les reconnaitre et les différencier. Le but de ce projet est d'implémenter un programme capable de comparer des images d'empreintes digitales entre elles. Concrètement, étant données des images d'empreintes digitales, il s'agit de pouvoir, dire si elles proviennent du même doigt.

1.1 Structure d'une empreinte digitale

Les empreintes digitales sont uniques à chaque individu. Leur structure générale varie cependant très peu. En effet, on définit seulement trois structures principales pour 95% des individus : structure en boucle (figure 1), structure en verticille (figure 2) et structure en arc (figure 3).

FIG. 1 : Empreinte avec structure en boucle

FIG. 2 : Empreinte avec structure en verticille

FIG. 3 : Empreinte avec structure en arc

Il est donc très difficile de comparer deux empreintes digitales en regardant leur structure générale.

Cependant, il existe sur toutes les empreintes digitales des points distinctifs :

- les points singuliers globaux :
  - noyau ou centre (figure 4), c'est-à-dire, le lieu de convergence des stries,
  - delta (figure 6), lieu de divergence des stries ;
- les points singuliers locaux, appelés **minuties**. Il en existe une dizaine de types différents, mais dans ce projet on s'intéressera seulement aux deux principaux : les terminaisons

(figure 5) et les bifurcations (figure 7). Dans ce projet, nous nous limiterons à l'utilisation de ces deux types de minuties qui s'avèrent suffisantes pour obtenir des résultats de comparaison corrects dans un grand nombre de cas. En utiliser davantage n'augmenterait pas sensiblement la précision de notre programme tout en nécessitant de recourir à un algorithme plus complexe que celui que nous allons mettre en oeuvre.

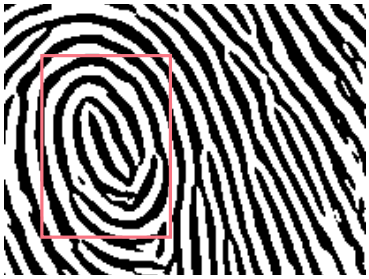

FIG. 4 : Noyau

FIG. 6 : Delta

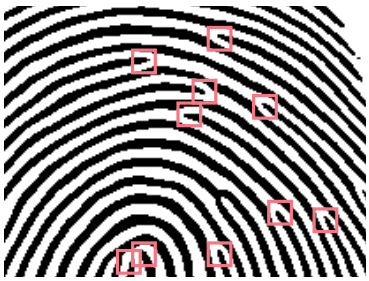

FIG. 5 : Terminaison

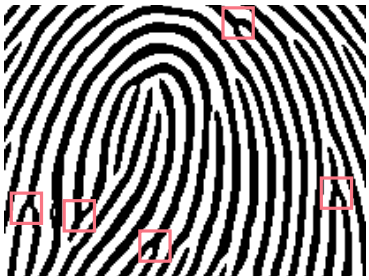

FIG. 7 : Bifurcation

1.2 Comment extraire les minuties ?

Extraire les minuties est l'une des étapes primordiales pour la comparaison d'empreintes digitales. Ce traitement se réalise en trois étapes :

1. le prétraitement (ou pré-processing) des images, qui consiste à améliorer la qualité de l'image et à la transformer au format binaire (voir Format binaire et nuance de gris). **Cette partie ne rentre pas dans le cadre du projet**.

2. la squelettisation (ou thinning), qui permet une extraction plus facile. Cette étape consiste à transformer tous les traits de l'image pour qu'ils aient une épaisseur d'un pixel. Un pixel est l'unité de base d'une image, c'est le plus petit point de couleur. Cette transformation maintient la structure de l'image tout en la simplifiant pour que le traitement soit plus facile.

3. et, la localisation des minuties et le calcul des orientations.

Vous pouvez visualiser ces étapes sur la figure 8.

Les images vous seront données déjà pré-traitées. Vous devrez donc uniquement réaliser les étapes 2 et 3.

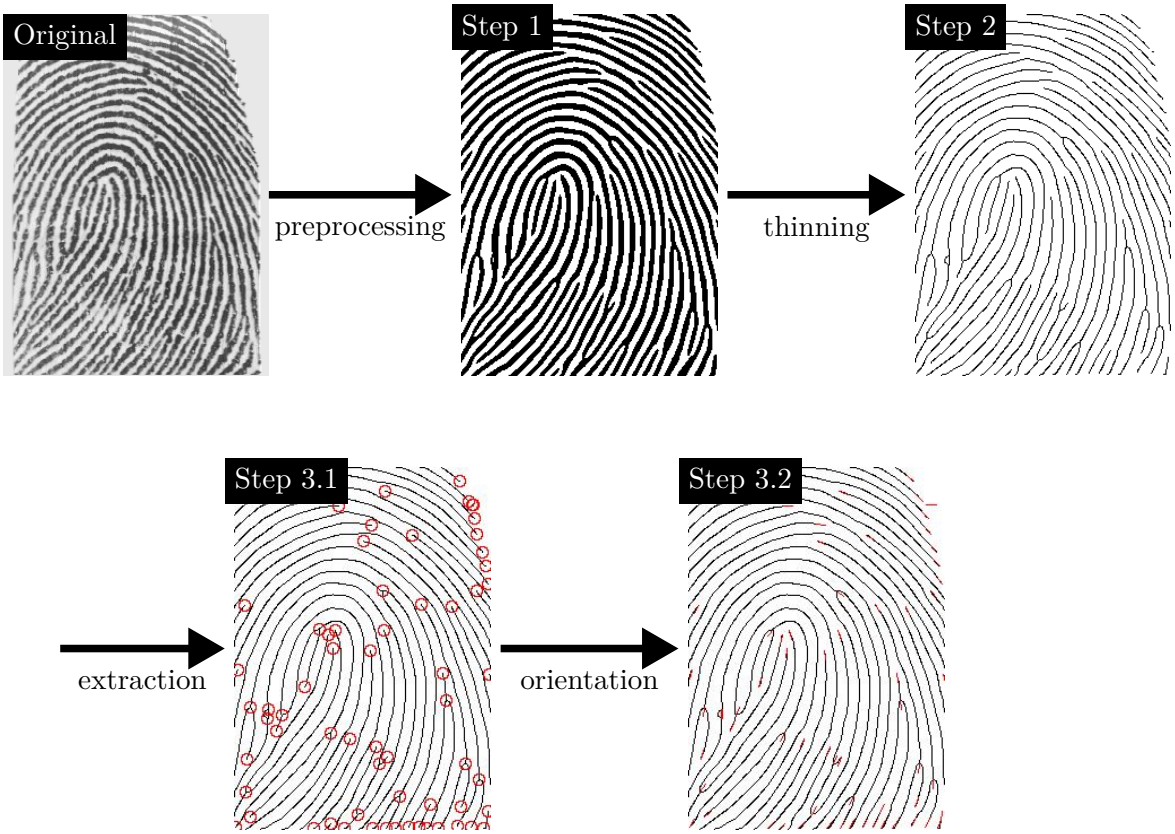

FIG. 8 : Les étapes de l'extraction des minuties

1.3 Comment comparer des empreintes digitales ?

Il existe de nombreuses méthodes pour comparer les empreintes digitales. Nous allons utiliser l'une des plus simples, mais aussi l'une des plus reconnues : comparer les minuties.

Pour comparer deux empreintes avec cette méthode, il convient donc dans un premier temps de trouver ces points distinctifs pour les deux images que nous voulons comparer. Une fois les points distinctifs trouvés, on extrait leurs coordonnées ainsi que leur orientation. Ce sont ces coordonnées et orientations que nous allons comparer. Si les deux images contiennent un certain nombre de points qui ont les mêmes coordonnées et les mêmes orientations nous considèrerons que les empreintes digitales sont identiques, c'est-à-dire, qu'elles appartiennent au même doigt, sinon, nous dirons quelles sont différentes.

Cette méthode prend en compte les différences de rotation et de translation possibles dans une image. Ainsi, si dans une image, l'empreinte digitale est de travers ou si elle n'est pas centrée, l'algorithme sera tout de même en mesure de détecter de quel doigt elle provient.

2 Structure et code fourni

2.1 Structure

Le projet, dans sa partie obligatoire, est articulé en trois parties :

1. l'implémentation des fonctions nécessaires à la squelettisation d'une image binaire,
2. l'implémentation des fonctions permettant d'extraire les coordonnées et l'orientation des minuties,
3. l'implémentation des fonctions permettant de dire si deux ensembles de minuties proviennent du même doigt.

- Tout le code obligatoire devra être réalisé dans le fichier `Fingerprint.java`.
- 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**[a]. Vérifiez que ce programme **compile** bien, avant de soumettre.
- Vous trouverez dans le dossier fourni `resources/`, **quelques fichiers que vous pourrez utiliser** comme jeu de données **pour tester vos méthodes**.

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

2.2 Code fourni

La manipulation de fichiers/images et de fenêtres étant fastidieuse et trop avancée pour ce cours, une partie du code vous est donnée. Le fichier `Helper.java` vous simplifie l'interaction avec les images et les conversions :

- `boolean[][] readBinary(String)` et `boolean writeBinary(String, boolean[][])` permettent de lire et d'écrire des images dont le format est un tableau de pixels au format binaire (voir Format binaire et nuance de gris).
- les méthodes `int[][] readARGB` et `boolean writeARGB(String, int[][])` permettent de lire et d'écrire des images dont le format est un tableau de pixels au format ARGB (voir Représentation ARGB). Utiliser des images ARGB n'est pas strictement nécessaires pour le projet, mais cela peut s'avérer utile pour en vérifier le bon fonctionnement. Vous pouvez par exemple utiliser les méthodes décrites plus tard pour dessiner des cercles, des traits ou les minuties avec des couleurs,
- les méthodes `void show(int[][] array, String title)` et `void show(boolean[][] array, String title)` permettent d'afficher des images aux formats ARGB et binaire,

- les méthodes `int[][] fromBinary(boolean[][])` et `boolean[][] toBinary(int[][])` permettent de passer de ARGB vers le binaire et inversement,
- des méthodes permettant de debugger votre code vous sont données. Ces méthodes permettent de dessiner des lignes, des cercles et les minuties : `void addLine(/*...*/)` `void addCircle(/*...*/)`, `void drawMinutia(/*...*/)`.

2.3 Tests

Important : La vérification du comportement correct de votre programme vous incombe. Le fichier fourni `Main.java`, partiellement rédigé, vous servira à tester vos développements de façon simple. Il vous revient la tâche de compléter `Main.java` en y invoquant vos méthodes de façon adéquate pour vérifier l'absence d'erreur dans votre programme.

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 donc aussi des tests vérifiant comment sont gérés les *cas particuliers*. Ainsi, il est important que votre programme traite correctement toute donnée d'entrée *valide*.

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 débuggage, et vous aident à raisonner quant au comportement d'une fonction. **Nous supposerons que les arguments des fonctions sont valides** (sauf exceptions explicitement mentionnées).

Pour garantir cette hypothèse, nous vous invitons à utiliser les assertions Java[1]. 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 qu'un paramètre de méthode `key` n'est pas `null`, vous pouvez écrire `assert tab != null;` au début de la méthode. Un exemple d'utilisation d'assertion est donné dans la coquille de la méthode `getNeighbours` dans le fichier `FingerPrint.java`.

> Les assertions doivent être activées pour fonctionner. Ceci se fait en lançant le programme avec l'option "-ea" [Lien cliquable ici].

Finalement, pour tester le fonctionnement général de votre programme, vous disposez de plusieurs ressources :

- dans `resources/fingerprints`, vous disposez d'un ensemble d'empreintes digitales déjà prétraitées. La base de données contient 16 doigts et 8 empreintes pour chaque doigt. Le nom des images est composé de deux numéros. Le premier correspond au numéro du doigt et le deuxième au numéro de l'empreinte. Par exemple `1_1.png` et `1_2.png` sont

[1] Nous aurons l'occasion d'y revenir en détail, mais leur utilisation est assez intuitive pour que nous puissions déjà y recourir

les empreintes 1 et 2 du premier doigt tandis `2_1.png` est la première image d'un doigt différent. Cette base de données est complexe. Le programme que nous allons faire ne sera pas parfait et pourra se tromper, c'est-à-dire, qu'il pourra dire que deux empreintes digitales proviennent du même doigt alors que c'est faux, ou, dans l'autre sens, qu'il pourra dire qu'elles proviennent de doigts différents alors qu'elles appartiennent au même doigt. Vous pouvez vous fier aux tests proposés dans `Main.java` pour certains résultats attendus et, en cas de doutes, vous pouvez nous demander.

- via ce lien, vous pouvez disposer du même ensemble d'empreintes digitales, mais qui n'ont pas subi de prétraitement. Vous pourrez les stocker au besoin dans `resources/original_fingerprints`. À noter que vous ne pouvez pas les utiliser dans votre projet telles qu'elles. Libre à vous d'implémenter le prétraitement comme bonus si vous le souhaitez, mais ce n'est pas nécessaire pour avoir la note maximale et cela demande beaucoup de travail et de recherches.

- dans `resources/test_inputs` et `resources/test_outputs`, vous trouverez des images permettant de vérifier le fonctionnement de votre code avec nos résultats. Vous trouverez notamment une image `skeletonTest.png`. Il s'agit d'une empreinte digitale ayant déjà subi la squelettisation. Cela vous permettra de passer à l'étape 2 et 3 sans avoir complété la première entièrement.

Voici un résumé des consignes/indications principales à respecter pour le codage du projet :

- Les paramètres des méthodes seront considérés comme exempts d'erreur, sauf mention explicite du contraire.
- Les entêtes des méthodes fournies doivent rester inchangées : le fichier `SignatureCheck.java` ne doit donc pas comporter de fautes de **compilation** au moment du rendu.
- 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 !**
- La vérification du comportement correct de votre programme vous incombe. Néanmoins, nous fournissons le fichier `Main.java`, illustrant comment invoquer vos méthodes pour les tester. Les exemples de tests ainsi fournis sont non exhaustifs et vous êtes autorisé.e.s/encouragé.e.s à modifier `Main.java` pour faire d'avantage de vérifications. Vos efforts en terme de test seront aussi rétribués.
- Votre code devra respecter les conventions usuelles de nommage.
- Le projet sera codé sans le recours à des librairies externes. Si vous avez des doutes sur l'utilisation de telle ou telle librairie, posez-nous la question et surtout faites attention aux alternatives que Eclipse vous propose d'importer sur votre machine.
- Votre projet **ne doit pas être stocké sur un dépôt public** (de type github). Pour ceux d'entre vous qui sont familiers avec git, ceci peut aussi être utile : https ://gitlab.epfl.ch/.

3 Tâche 1 : Squelettisation

La première tâche de ce projet consiste à programmer un algorithme capable de tracer le squelette d'une image. C'est-à-dire, une fonction qui, étant donnée une image binaire, est capable d'affiner tous ses traits pour qu'ils aient une épaisseur d'un pixel. Le but de cette étape est de simplifier l'image le plus possible toute en gardant sa structure. Cela simplifiera grandement l'extraction des minuties.

Vous allez travailler avec des images déjà pré-traitées (c'est à dire que vous partirez de l'étape 1 de la figure 8). Chacune de ces images est stockée dans un tableau à deux dimensions de booléens. La valeur booléenne `true` représente un pixel noir et la valeur `false`, un pixel blanc. Vous noterez que le système de coordonnées usuellement utilisé pour représenter une image est particulier. **L'origine est située dans le coin en haut à gauche de l'image**. Chaque ligne de l'image correspond à un tableau de booléens. On procédera donc comme suit pour accéder aux pixels d'une image qui fait 100 pixels de largeur et 100 pixels de hauteur :

```
image[0][0]   = true; // accès au pixel en haut à gauche
image[99][0]  = true; // accès au pixel en bas à gauche
image[0][99]  = true; // accès au pixel en haut à droite
image[99][99] = true; // accès au pixel en bas à droite
```

La classe fournie, `Helper.java`, est codée selon cette convention.

3.1 Méthodes préliminaires

L'algorithme opère sur tous les pixels noirs ayant huit voisins, c'est-à-dire, tous les pixels ne se situant par sur le bord de l'image. Pour chacun de ces pixels, l'algorithme va déterminer s'il est essentiel à l'intégrité de l'empreinte digitale ou s'il peut être supprimé.

Pour déterminer si un pixel P est essentiel (pertinent), il suffit de regarder ses huit voisins directs $(P_0, \ldots, P_7)$. Par conventions, on les nomme comme suit :

P_7	P_0	P_1
P_6	P	P_2
P_5	P_4	P_3

3.1.1 Méthode `getNeighbours`

Commencez par écrire la méthode `getNeighbours` capable d'extraire les voisins d'un pixel dans l'ordre voulu : P_0, P_1, ..., P_7.

```
boolean[] getNeighbours(boolean[][] image, int row, int col)
```

Cette fonction prend en paramètre l'image ainsi que les coordonnées du pixel dont on veut extraire les voisins. Le type de retour est un tableau de booléens. L'élément à l'index 0 doit contenir la valeur de P_0, celui à l'index 1 celle de P_1 et ainsi de suite. Le tableau doit contenir 8 éléments exactement. Il vous incombe de tester si tous les voisins existent dans l'image. Si un

des pixels n'est pas dans l'image, il sera retourné comme s'il existait et était blanc. Si la colonne ou la ligne passée en paramètre n'est pas dans l'image, la fonction devra retourner `null`. Vous trouverez deux exemples en figure 9.

FIG. 9 : Exemple de résultat de `getNeighbours`

3.1.2 Méthode `blackNeighbours`

Écrivez maintenant la méthode `blackNeighbours`.

```
int blackNeighbours(boolean[] neighbours)
```

Cette méthode sera utilisée pour compter le nombre de pixels noir dans le tableau résultant d'un appel à la méthode `getNeighbours`. Elle retourne donc un entier entre 0 et 8. Voir figure 10 pour un exemple. Le pixel rouge a 3 voisins noirs.

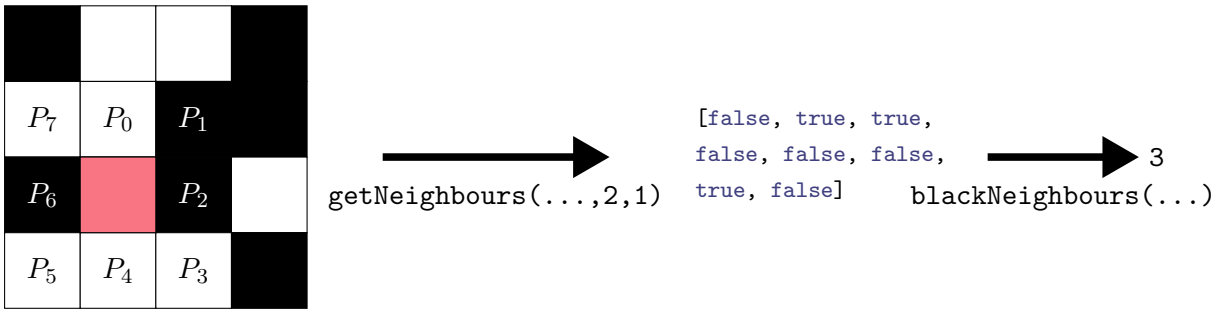

FIG. 10 : Exemple de résultat de `blackNeighbours`

3.1.3 Méthode `transitions`

Pour évaluer si un pixel est essentiel, on utilise un dernier critère : le nombre de transitions de blanc vers noir dans la séquence P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_0.

Par exemple dans la séquence : `[blanc, noir, noir, blanc, blanc, blanc, noir, blanc, blanc]`. Le nombre de transitions est 2. On passe une fois de blanc à noir lorsque l'on passe de P_0 à P_1 et une fois encore lorsque l'on passe de P_5 à P_6.

Visuellement, cet exemple se traduit ainsi (voir figure 11).

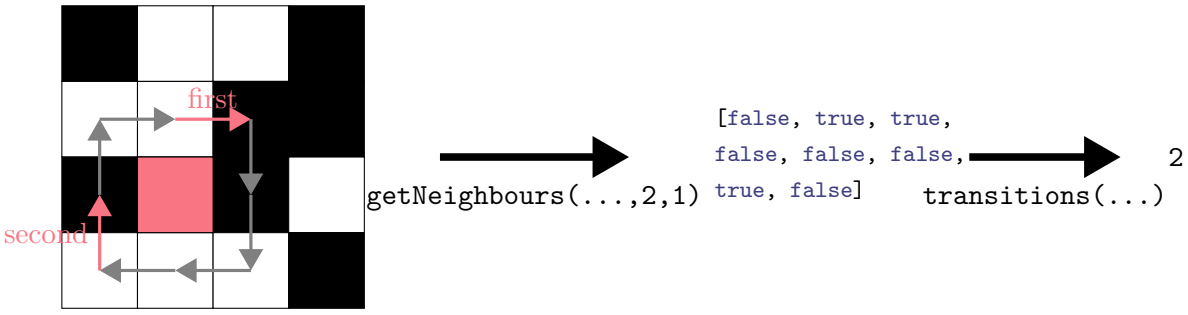

FIG. 11 : Exemple nombre de transitions

Écrivez la méthode `transition` qui, comme `blackNeighbours` prend comme unique paramètre le résultat de `getNeighbours`. Cette méthode retournera le nombre de transitions pour le pixel à qui appartiennent ces voisins.

```
int transitions(boolean[] neighbours)
```

3.1.4 Méthode `identical`

L'algorithme de squelettisation va itérativement supprimer les pixels non pertinents jusqu'à ce que l'image se stabilise (atteigne la squelettisation). Il est donc utile pour le mettre en oeuvre de disposer d'une méthode capable de dire si une image est identique à une autre. Cela permettra de savoir si une image n'a pas changé d'une itération à la suivante. On saura alors que tous les pixels sont pertinents et que l'algorithme peut s'arrêter. Écrivez pour cela la méthode suivante qui retourne `true` si les images passées en paramètre sont identiques et `false` sinon.

```
boolean identical(boolean[][] image1, boolean[][] image2)
```

3.2 Méthodes principales

Avec l'aide des fonctions implémentées précédemment, nous sommes maintenant capables de dire si un pixel est important pour l'intégrité de l'empreinte digitale en testant certaines conditions. Ces conditions sont groupées en deux étapes et sont testées indépendamment, c'est-à-dire, que l'on teste d'abord tous les pixels avec les conditions de la première étape ce qui permettra de supprimer certains pixels non pertinents. Puis, l'on teste tous les pixels avec les conditions de

la deuxième l'étape et l'on élimine ainsi d'autres pixels non pertinents. L'algorithme effectue les deux étapes l'une après l'autre. Les deux étapes sont exécutées exactement le même nombre de fois. Après avoir exécutés les deux étapes, on vérifie si un pixel a été modifié. Tant qu'un pixel a été modifié, on exécute à nouveaux les deux étapes.

L'algorithme prend en paramètre une image d'origine, qui doit resté inchangée, et produire une image épurée des pixels estimés non pertinents. Cette dernière est au départ identique à l'image d'origine puis est ensuite itérativement modifiée comme suit :

> **Étape 1**
> Tous les pixels de l'image d'origine sont testés. Les pixels satisfaisants toutes les conditions suivantes dans cette image doivent être considérés comme non pertinents et être effacés (mis à blanc) dans l'image résultat :
>
> 1. Le pixel est noir,
> 2. Le tableau des 8 voisins du pixel est non nul,
> 3. $2 \leq$ `blackNeighbours()` ≤ 6,
> 4. `transitions()` $= 1$,
> 5. P_0 ou P_2 ou P_4 est blanc,
> 6. P_2 ou P_4 ou P_6 est blanc.
>
> **Étape 2**
> L'image résultant de l'étape précédente est considérée comme image de départ. Il s'agira de produire une nouvelle image résultat, à partir de cette image de départ qui doit rester inchangée.
>
> Les pixels satisfaisants toutes les conditions suivantes dans l'image de départ doivent être effacés dans l'image résultat :
>
> 1. Le pixel est noir,
> 2. Le tableau des 8 voisins du pixel est non nul,
> 3. $2 \leq$ `blackNeighbours()` ≤ 6,
> 4. `transitions()` $= 1$,
> 5. P_0 ou P_2 ou P_6 est blanc,
> 6. P_0 ou P_4 ou P_6 est blanc.
>
> **Itération**
> Tant qu'il y a des changements à l'étape 1 ou 2, on répète les deux étapes. Pour vérifier s'il y a eu un changement, vous pouvez utiliser la méthode `identical()`.

L'algorithme sera réparti en deux méthodes :

- une méthode principale qui mettra en oeuvre l'itération et vérifiera si l'image a changé après un appel aux deux étapes :

```
boolean[][] thin(boolean[][] image)
```

- et, une méthode qui appliquera l'étape 1 ou 2 et retournera une image épurée des pixels non pertinents (conformément au critère de chaque étape) :

```
boolean[][] thinningStep(boolean[][] image, int step)
```

Notez qu'une seule méthode est utilisée pour les deux étapes, car les traitements sont très similaires. Le paramètre `step` indique quelle étape est mise en oeuvre (valant 0 pour l'étape 1 et 1 pour l'étape 2).

Attention : pour copier un tableau bi-dimmensionnel, n'utilisez la méthode prédéfinie `Arrays.copyOf` que si vous savez exactement ce que vous faites. Privilégiez autrement pour le moment, une copie explicite, faite élément par élément, au moyen de deux boucles imbriquées[2].

3.3 Tests

Utilisez le programme `Main.java` pour tester vos développements. Les exemples qui y sont fournis ne sont pas exhaustifs : **complétez-les** selon ce qu'il vous semble judicieux de faire, pour tester l'ensemble des méthodes développées dans cette partie du projet.

4 Tâche 2 : Localisation et calcul de l'orientation

Dans cette partie, il s'agit maintenant de trouver la position et l'orientation des minuties en utilisant le squelette de l'empreinte digitale (voir figure 12 pour un exemple d'extraction).

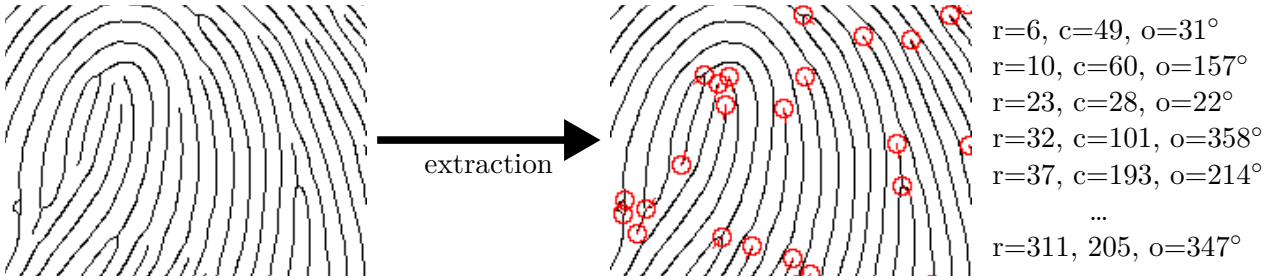

FIG. 12 : Exemple d'extraction des minuties

Pour trouver les coordonnées des minuties, on itère sur chaque pixel et on détermine s'il s'agit d'un point d'intérêt ou non. Une fois qu'une minutie a été trouvée, on détermine son orientation.

Pour calculer l'orientation, vous devrez écrire trois méthodes utilitaires :

- en section 4.1, vous écrirez la méthode `boolean[][] connectedPixels(boolean[][] image, int row, int col, int distance)`
- en section 4.2, vous écrirez la méthode `double computeSlope(boolean[][] connectedPixels, int row, int col)`

[2]Nous reviendrons un peu plus tard dans le semestre sur les notions de copie de surface et copie profonde (« deep vs shallow copy »)

- en section 4.3, vous écrirez la méthode `double computeAngle(boolean[][] connectedPixels, int row, int col)`

Grâce à ces trois méthodes, en section 4.4, vous devrez écrire la fonction `int computeOrientation(boolean[][] image, int row, int col, int distance)`.

Finalement, en section 4.5, vous écrirez la fonction `List<int[]> extract(boolean[][] image)` qui itère sur chaque pixels, détermine s'il s'agit d'une minutie et appelle `computeOrientation(...)`.

4.1 Fonction `connectedPixels`

Pour calculer l'orientation, on utilise les pixels proches de la minutie et qui y sont connectés. Par exemple, dans la figure 13 à gauche, la minutie est en rouge et les pixels qui nous intéressent sont en vert foncé. Les pixels restants, en noir, ne nous intéressent pas. Ils ne sont pas connectés ou ne sont pas assez proches. Un pixel est considéré comme suffisament proche s'il se situe dans le carré de taille `2*distance + 1` centré sur la minutie. `distance` est un paramètre de la fonction. Attention, il est possible que cette région dépasse l'image (déborde). Pensez à prendre les précautions nécessaires.

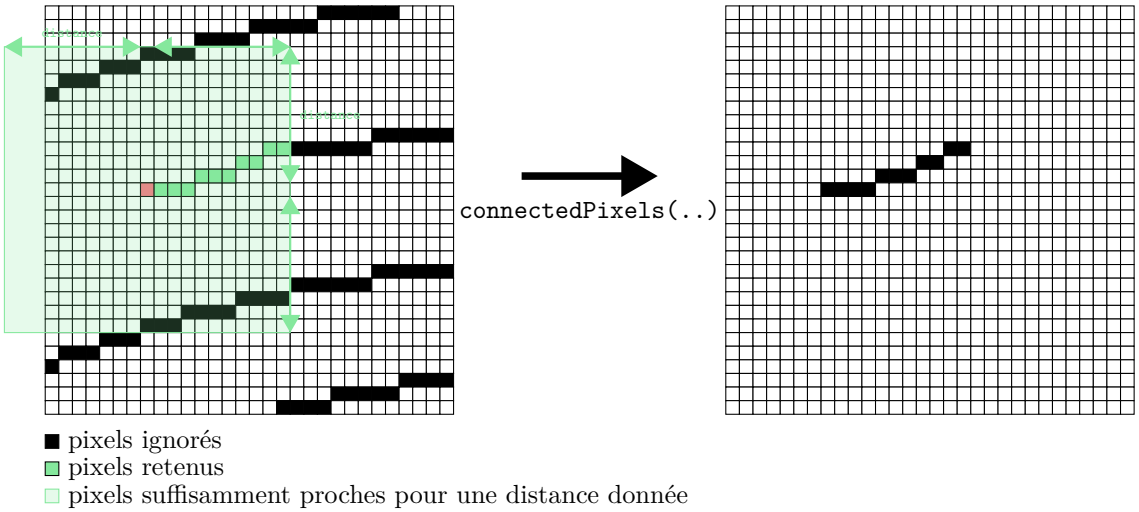

Fig. 13 : Pixels utilisés pour calculer l'orientation

Nous allons écrire une fonction capable de trouver tous ces pixels (qui sont en vert foncé ou en rouge, voir la figure 13 à gauche et le résultat à droite).

```
boolean[][] connectedPixels(boolean[][] image, int row, int col,
    int distance)
```

Elle prend l'image ainsi que les coordonnées de la minutie et la distance discutée plus haut. Elle retournera un tableau à deux dimensions de booléens. Le tableau devra avoir la même taille que l'image. Les éléments de ce tableau devront être `true` si

1. le pixel à ces coordonnées est noir, et
2. le pixel est connecté à notre minutie, et
3. le pixel se trouve dans le carré de taille `2*distance + 1` centré sur la minutie.

Sinon, l'élément devra être `false`. Voir figure 14 pour un exemple.

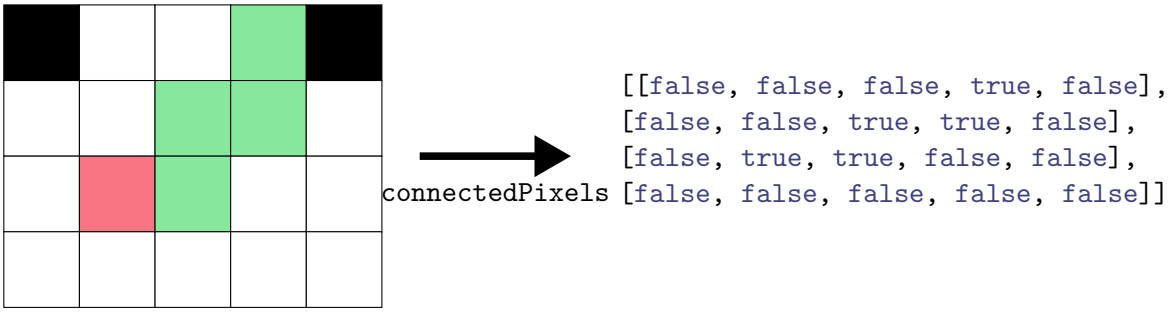

FIG. 14 : Exemple de résultat de `connectedPixels(image, 2, 1, 2)` : ici l'image donnée en paramètre est celle représentée à gauche (le résultat de `connectedPixels` a la même taille que `image`)

Idée 1 : Une idée simple d'algorithme pour `connectedPixels` consiste à itérer sur l'ensemble de l'image un nombre indéfini de fois à l'aide d'une boucle `while`. Lors de chaque itération, on regarde pour chaque pixel, s'il est noir, s'il se trouve à coté d'un pixel déjà dans le tableau des pixels connectés et s'il se trouve dans la distance requise. Alors, s'il respecte ces trois conditions, on l'ajoute au tableau des pixels connectés. Tant que l'on trouve de nouveaux pixels, on refait une itération sur toute l'image. Lorsque l'on ne trouve de plus de pixels, cela veut dire qu'on a trouvé tous les pixels connectés.

Idée 2 : Si vous avez réalisé le devoir du MOOC de la semaine 4 (exercice 3 sur les clotures), une idée d'algorithme très similaire à celui suggéré en page 10 de l'énoncé peut être utilisée pour `connectedPixels` .

4.2 Fonction `computeSlope`

Pour calculer l'orientation d'une minutie m, on cherche à trouver la droite qui passe par m et qui est la plus proche possible des points environnants connectés. Pour trouver cette droite, on procède mathématiquement en utilisant une méthode nommée régression linéaire simple (des explications sont fournies dans l'encadré de la page suivante).

La droite qui donne la direction de la minutie m a pour équation :

$$y = a \cdot x \text{ avec } \begin{cases} a = \frac{\sum x \cdot y}{\sum x^2}, & \text{si } \sum x^2 \geq \sum y^2 \\ a = \frac{\sum y^2}{\sum x \cdot y}, & \text{si } \sum x^2 < \sum y^2 \end{cases}$$

où (x, y) sont les coordonnées (dans le repère prenant son origine en m) des pixels noirs dans le tableau des pixels connectés. Ces coordonnées ainsi que la droite sont définis dans le repère mathématique usuel (origine en bas à gauche et axe y montant vers le haut).

Pour calculer la droite il suffit donc de déterminer la pente a. Il faut pour cela calculer les sommes des formules ci-dessus mais en tenant compte du fait que dans la représentation des images utilisée, le repère prend son origine en haut à gauche (avec l'axe des y dirigé vers le bas). Soit $[row_m][col_m]$ les coordonnées de la minutie dans l'image, en posant row la ligne à laquelle se trouve un pixel vert et col sa colonne dans l'image, on peut calculer les coordonnées x et y comme suit :

$$\begin{aligned} x &= col - col_m \\ y &= -(row - row_m) = row_m - row \end{aligned}$$

Note : le moins dans le calcul de y vient du fait que l'axe des y ne varie pas dans la même direction pour les deux repères.

Exemple concret : dans la figure 14, la minutie m se trouve sur la ligne 2 et la colonne 1 dans l'image. Pour trouver son orientation, on doit calculer les trois sommes utilisées dans la formule précédente pour tous les pixels verts. C'est-à-dire, la somme des x multipliés par y et les sommes des x^2 et des y^2. Plus précisément, les pixels verts dans cette exemple ont pour coordonnées (x, y) dans le repère usuel dont l'origine est en m : (1, 0), (1, 1), (2, 1), (2, 2) et donc les sommes sont :

$$\begin{aligned} \sum x \cdot y &= 1 \cdot 0 + 1 \cdot 1 + 2 \cdot 1 + 2 \cdot 2 = 7 \\ \sum x^2 &= 1^2 + 1^2 + 2^2 + 2^2 = 10 \\ \sum y^2 &= 0^2 + 1^2 + 1^2 + 2^2 = 6 \end{aligned}$$

Si $\sum x^2 \geq \sum y^2$, la droite qui donne l'orientation dans la figure 14 à finalement pour pente $a = 7/10$.

Écrivez la fonction qui calcule cette pente :

```
public static double computeSlope(boolean[][] connectedPixels, int
    row, int col)
```

Cette fonction prend des paramètres analogues à ceux de la fonction `connectedPixels` mais au lieu de l'image complète de l'empreinte digitale, elle prend celle résultant des appels à `connectedPixels`, c'est-à-dire, celle ne contenant que les pixels qui sont proches et connectés à la minutie. Le paramètre distance n'y est pas nécessaire

Il existe un cas particulier pour lequel les pixels connectés sont mieux approximés par une ligne verticale ($\sum x^2 = 0$). Dans ce cas, la fonction doit retourner la valeur `Double.POSITIVE_INFINITY`.

Explication - Régression linéaire simple

La compréhension des détails mathématiques n'est pas nécessaire pour la réussite du projet. Vous pouvez le contenter d'appliquer l'équation précédente. L'explication est destinée aux plus curieux d'entre vous.

Dans cette méthode, on cherche à trouver une droite d'équation $y = a \cdot x + b$ qui se rapproche le plus de nos points. Vous pouvez observer en figure 15 un exemple de droite qui passe proche des points et en figure 16 une droite qui reste éloignée des points.

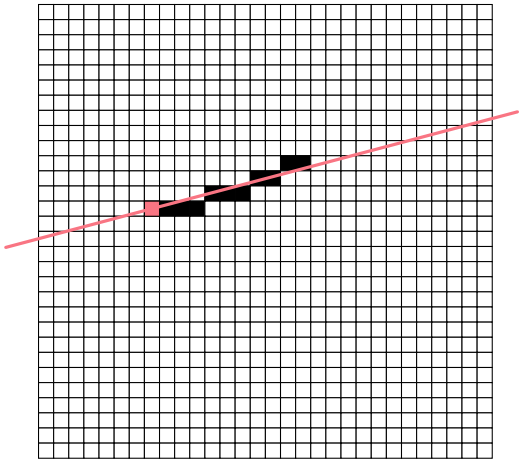

FIG. 15 : Droite proche de nos points

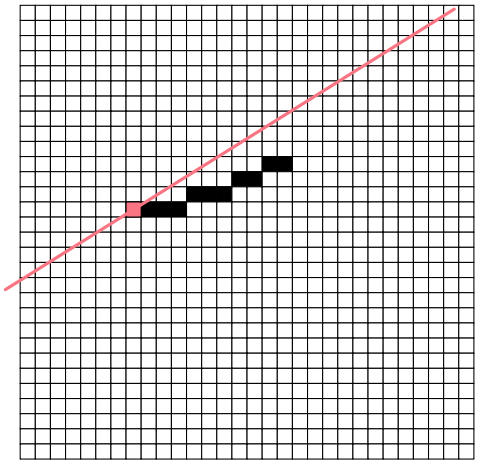

FIG. 16 : Droite éloignée de nos points

Pour décider si une droite passe proche de nos points, on peut calculer la somme des distances entre notre droite et chaque point et essayer de minimiser cette somme. Sur la figure 17, vous pouvez visualiser les distances en rouge.

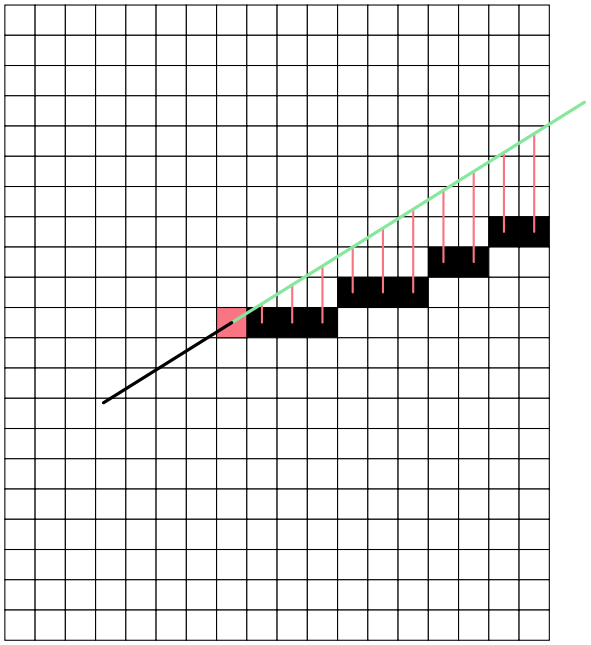

FIG. 17 : Distance à la droite

Cependant, lorsque que la droite passe en dessous de certains points et au-dessus d'autres points, des valeurs négatives apparaissent en calculant la différence. Pour ne pas avoir de valeur négative dans nos calculs, on utilise la distance au carré.

Finalement, si l'on choisit comme origine de notre repère Cartésien le point de minutie, la droite prend la forme $y = a \cdot x$, car l'ordonné à l'origine, b, est égale à 0. Il est donc possible de calculer la distance au carré totale comme

$$\begin{aligned} error &= \sum(\text{line value} - \text{pixel value}) \\ &= \sum(ax - y)^2 \\ &= \sum(a^2x^2 - 2axy + y^2) \text{ , pour toute paire } (x, y) \end{aligned}$$

Pour trouver le a qui donne la droite la plus proche des points, on veut minimiser la distance au carré, $error$. Pour cela, on dérive $error$ et on cherche quand elle s'annule :

$$\begin{aligned} &\frac{d(error)}{dx} = 0 \\ \iff &\sum(2ax^2 - 2xy) = 0 \\ \iff &a = \frac{\sum xy}{\sum x^2} \end{aligned}$$

Cette solution pose des problèmes en pratique. Lorsque la droite voulue s'approche de la verticale, il devient de plus en dur de trouver un résultat optimal.

Pour cela, on imagine que l'on tourne l'image de 90° pour retomber sur une droite horizontale. Cela revient à inverser les x et y. Alors, $\sum xy$ reste $\sum xy$ mais $\sum x^2$ devient $\sum y^2$. Il faut ensuite inverser la pente pour la remettre dans la sens original de l'image. Et on obtient :

$$a = \frac{\sum xy}{\sum y^2}$$

Cependant, il faut prendre en compte le fait que l'image avait été tournée et retourner la pente de 90°. Pour ce faire on inverse a :

$$a = \frac{1}{\frac{\sum xy}{\sum y^2}} = \frac{\sum y^2}{\sum xy}$$

On va utiliser cette nouvelle formule si les pixels connectés s'étendent plus selon les y que selon les x. C'est-à-dire, si $\sum y > \sum x$ ce qui est équivalent à dire que la somme des $\sum y^2 > \sum x^2$:

$$\begin{cases} a = \frac{\sum x \cdot y}{\sum x^2}, & \text{si } \sum x^2 \geq \sum y^2 \\ a = \frac{\sum y^2}{\sum x \cdot y}, & \text{si } \sum x^2 < \sum y^2 \end{cases}$$

4.3 Fonction `computeAngle`

Nous savons à ce stade calculer la direction de la minutie. Il nous reste cependant encore deux possibilités quant à son orientation (voir figure 18), il nous faut donc maintenant nous intéresser au calcul du sens.

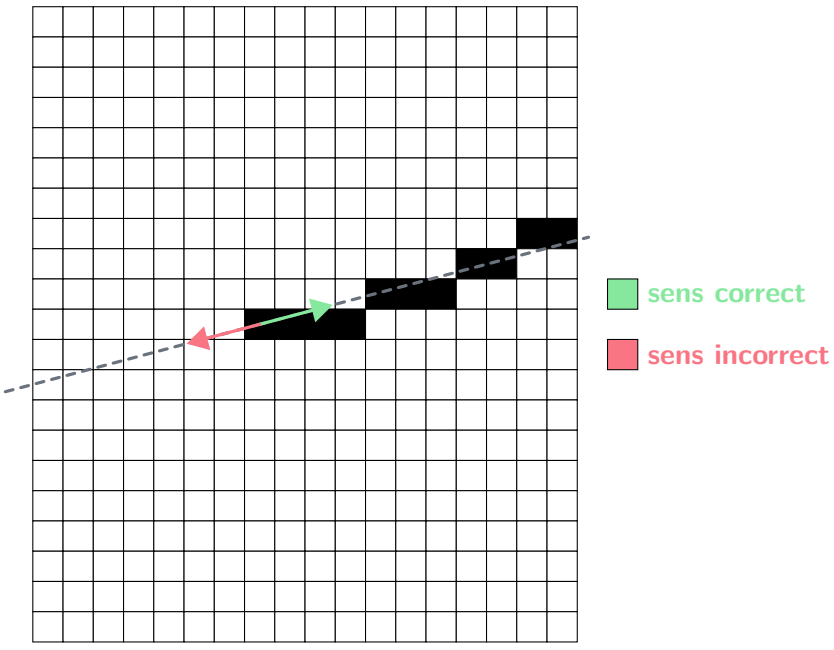

Fig. 18 : Exemple sens d'une minutie

Le sens est déterminé par l'angle que forme le vecteur direction choisi avec l'axe des x. La

fonction arctan() (`atan` en Java) appliquée à la pente de la droite calculée précédemment, permet en principe de calculer l'angle recherché en radians. Cependant, comme l'angle ainsi calculé se trouvera entre $-\pi/2$ et $\pi/2$. Il est impossible *a priori* de distinguer si l'on se trouve dans le 1er ou le 3ème cadrans (0-90° ou 180-270°), ou dans le 2ème ou le 4ème cadrans (90-180° ou 270-360°).

Nous allons donc pour distinguer ces cas choisir d'orienter la minutie dans le sens où il y a le plus de pixels. Pour trouver dans quel sens il y en a le plus, on imagine la perpendiculaire à notre droite et on compte le nombre de pixels qui sont *au-dessus* de cette nouvelle droite et le nombre de pixels qui sont en dessous. Cette nouvelle droite a pour équation :

$$y = -\frac{1}{a}x \text{ , où } a \text{ est le coefficient directeur de la droite donnant la direction}$$

Dans la figure 19, on peut observer la droite perpendiculaire en rouge et la droite qui donne la direction en gris. La zone verte contient tous les pixels qui sont *au-dessus* de la droite. La zone bleue contient ceux en dessous.

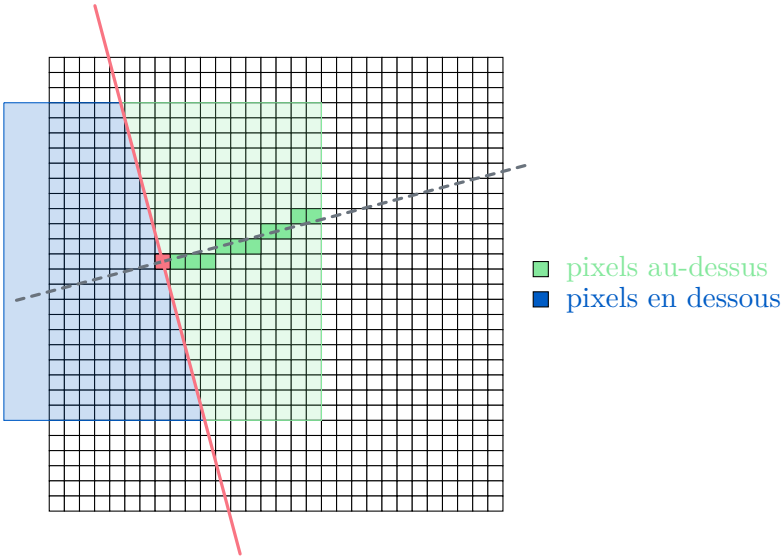

FIG. 19 : Séparation des pixels par la droite perpendiculaire

Donc, un point qui se trouve aux coordonnées (x, y) est au-dessus si $y \geq -\frac{1}{a}x$. Sinon, il est en dessous. Les coordonnées (x, y) correspondent ici encore coordonnées du repère mathématique usuel, centré sur la minutie.

Pour calculer le bon angle, il faudra donc ajouter π à l'angle calculé au moyen de $arctan()$ si :

- cet angle est positif, mais le nombre de pixels en dessous de la perpendiculaire est supérieur au nombre de pixels au-dessus,
- ou, si cet angle est négatif, mais le nombre de pixels en dessous de la perpendiculaire est inférieur au nombre de pixels au-dessus.

Il vous est demandé d'implémenter la fonction qui retourne l'angle, en radians, indiquant l'orientation de la minutie selon l'algorithme précédemment suggéré :

```
public static double computeAngle(boolean[][] connectedPixels,
    int row, int col, double slope)
```

Cette fonction prend les mêmes paramètres que `computeSlope` et le résultat de cette dernière, `slope`.

Note : N'oubliez pas de faire attention au cas particulier où la pente vaut `Double.POSITIVE_INFINITY`. Dans ce cas, l'angle est $\pi/2$ ou $-\pi/2$ selon la direction de la ligne : droite vers le haut ou droite vers le bas. Dans le premier cas les pixels connectés sont au dessous de la minutie et dans le second en dessus.

4.4 Fonction `computeOrientation`

Avec les méthodes précédentes, écrivez la méthode `computeOrientation` qui retourne l'angle de l'orientation en degrés, entre 0° à 359°, arrondi à l'entier le plus proche.

```
public static int computeOrientation(boolean[][] image, int row,
    int col, int distance) {
```

Lorsque que vous appellerez cette méthode, le paramètre distance sera la constante fournie `ORIENTATION_DISTANCE`.

Cette méthode devra, dans l'ordre :

1. appeler `connectedPixels(...)`,
2. appeler `computeSlope(...)`
3. appeler `computeAngle(...)`
4. retourner l'angle en degrés, positif et arrondi à l'entier le plus proche. Vous pouvez utiliser les fonctions standard de Java `Math.round(...)` est `Math.toDegrees(...)`. Si un angle est strictement négatif il suffit de lui ajouter 360 pour en faire un angle positif.

4.5 Fonction `extract`

Grâce aux fonctions précédentes, il nous est possible de calculer l'orientation de n'importe quelle minutie. Il ne nous reste plus qu'à les trouver. Pour cela, il vous est demandé d'implémenter la fonction principale `extract` :

```
public static List<int[]> extract(boolean[][] image)
```

Cette fonction prend en paramètre uniquement l'image et retourne une liste de minuties. La liste est du type `List<int[]>`. Chaque `int[]` correspond à une minutie. Le premier élément est la ligne occupée par la minutie, le deuxième la colonne et le troisième l'orientation en degrés. Pour simplifier, nous n'extrairons (et par la suite ne comparerons) que les minuties qui ont 8 voisins contenus dans l'image.

Pour trouver ces minuties, on itère sur toute l'image en partant du pixel 1 et non 0 et en s'arrêtant à l'avant dernière colonne pour garantir que tous les pixels considérés ont effectivement 8 voisins contenus dans l'image. On détermine pour chacun de ces pixels s'il s'agit d'une minutie. Comme nous l'avons vu en introduction, nous allons utiliser deux points spécifiques, les terminaisons et les bifurcations. Les terminaisons sont caractérisées par un nombre de transitions (tel que retourné par la méthode `transitions()`) dans les pixels voisins valant 1. C'est-à-dire, qu'il y a un seul trait qui part/vient de ce pixel. Pour les bifurcations, le nombre de transitions vaut 3, car il y a trois traits qui viennent/partent du pixel. Ainsi, on cherche tous les pixels ayant une ou trois transitions.

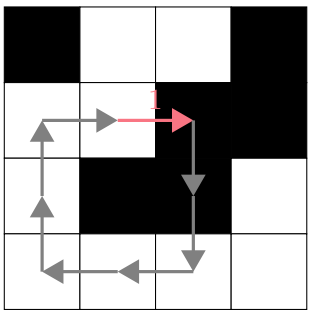

FIG. 20 : Terminaison (transitions = 1)

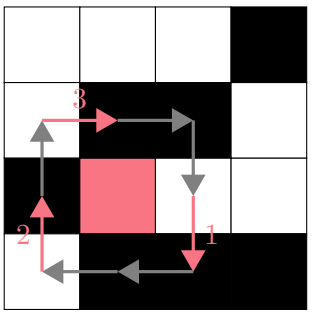

FIG. 21 : Bifurcation (transitions = 3)

4.6 Tests

Utilisez le programme `Main.java` pour tester vos développements. Les exemples qui y sont fournis ne sont pas exhaustifs : **complétez-les** selon ce qu'il vous semble judicieux de faire, pour tester l'ensemble des méthodes développées dans cette partie du projet.

5 Tâche 3 : Comparaison

Les deux étapes précédentes permettent d'extraire de n'importe quelle image la liste de minuties correspondantes. Nous allons maintenant écrire une fonction permettant de comparer deux listes de minuties pour dire si elles proviennent de la même empreinte digitale.

Notez qu'il ne suffit pas de comparer les deux listes et vérifier que tous les minuties sont aux mêmes positions et avec les mêmes orientations. Les images d'une même empreinte digitale peuvent être déplacées horizontalement (translation selon les colonnes), verticalement (translation selon les lignes) et orientées dans n'importe quelle direction. En outre, en fonction de la pression ou de l'angle avec lequel on capture l'empreinte digitale, les minuties peuvent apparaitre légèrement plus proche ou plus loin les unes des autres. Dans la figure 22, vous pouvez voir deux empreintes digitales du même doigt. La deuxième a subi une translation vers le bas et, elle est plus orientée vers la droite que la première. Ainsi, les minuties n'auront pas les mêmes coordonnées ni les mêmes orientations alors qu'elles proviennent du même doigt.

Fig. 22 : Exemple de deux empreintes digitales provenant du même doigt ayant une translation et rotation différente

5.1 Présentation de l'algorithme

Pour déterminer si deux ensembles de minuties proviennent de la même empreinte digitale, nous allons tenter de les superposer de toutes les manières possibles et nous allons regarder si une de ces superpositions donne une similarité suffisante. C'est ce qu'on appelle une résolution par « force brute ».

Afin de tester toutes les solutions possibles, nous allons essayer de superposer chaque minutie m_2 de la deuxième liste sur chaque minutie m_1 du premier ensemble. Intuitivement, cela revient à déplacer la deuxième empreinte digitale (c'est-à-dire appliquer une translation) pour que les coordonnées de m_2 soient les mêmes que celles de m_1. Il faut aussi tourner la deuxième image pour aligner les orientations de m_1 et m_2. Dans la figure 23, vous pouvez observer la superposition de deux empreintes digitales. Les minuties m_1 et m_2 ont été choisies au hasard dans cet exemple. Il s'agit donc de tester toutes les paires de m_1 et m_2 et regarder si une de ces paire donne une bonne superposition. Dans ce cas on estimera que les deux empreintes digitales proviennent du même doigt. Finalement, notez que dans cet exemple, l'image entière de la deuxième empreinte digitale a été déplacée mais, en pratique, vous n'aurez pas à calculer les nouvelles coordonnées de tous les pixels, uniquement ceux des minuties.

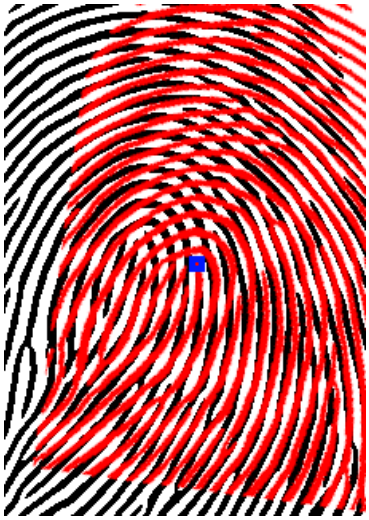

FIG. 23 : Exemple de superposition : l'empreinte digitale ayant subie une translation et rotation en rouge ; le point bleu indique la position des deux minuties alignés

Vous devrez commencer par écrire une méthode qui applique la transformation à toutes les minuties de la liste. Cette méthode s'appelle `applyTransformation`. Puis, vous devrez écrire une méthode qui compte le nombre de minuties qui se superposent : `matchingMinutiaeCount`. Finalement, vous serez en mesure d'écrire la méthode `match` qui retourne `true` si deux empreintes digitales proviennent du même doigt et `false` sinon.

5.2 Méthode `applyTransformation`

L'implémentation de `applyTransformation` utilise deux méthodes utilitaires, `applyRotation` et `applyTranslation`.

Implémentez la première méthode utilitaire prenant comme paramètre une minutie (`minutia`), le centre de la transformation (`centerRow` et `centerCol`) et la rotation (`rotation`). Cette méthode retourne les nouvelles coordonnées et la nouvelle orientation de la minutie après avoir appliqué la rotation.

```
public static int[] applyRotation(int[] minutia, int centerRow,
    int centerCol, int rotation)
```

Les nouvelles coordonnées et l'orientation sont calculées en utilisant les formules suivantes, où `row`, `col` et `orientation` sont les paramètres de la minutie originale et `x`, `y`, `newX`, `newY` sont des variables temporaires :

$$
\begin{aligned}
\texttt{x} &= \texttt{col} - \texttt{centerCol} \\
\texttt{y} &= \texttt{centerRow} - \texttt{row} \\
\texttt{newX} &= \texttt{x} \times \cos(\texttt{rotation}) - \texttt{y} \times \sin(\texttt{rotation}) \\
\texttt{newY} &= \texttt{x} \times \sin(\texttt{rotation}) + \texttt{y} \times \cos(\texttt{rotation}) \\
\texttt{newRow} &= \texttt{centerRow} - \texttt{newY} \\
\texttt{newCol} &= \texttt{newX} + \texttt{centerCol} \\
\texttt{newOrientation} &= (\texttt{orientation} + \texttt{rotation}) \mod 360
\end{aligned}
$$

`newRow` et `newCol` seront arrondis (`Math.round`) et transtypés en `int`. `newOrientation` sera aussi transtypé en `int`. L'ensemble retourné par `applyRotation` contiendra donc dans l'ordre : `newRow`, `newCol` et `newOrientation`. **Note :** n'oubliez pas que cos et sin prennent comme paramètre un angle en radians alors que la rotation est passée à la fonction en degrés.

Implémentez ensuite la seconde méthode utilitaire prenant comme paramètre une minutie (`minutia`), la translation verticale (`rowTranslation`) et la translation horizontale (`colTranslation`) à y appliquer. Cette méthode retourne les nouvelles coordonnées de la minutie après avoir appliqué la translation. L'orientation ne change pas.

```
public static int[] applyTranslation(int[] minutia, int
    rowTranslation, int colTranslation)
```

Les nouvelles coordonnées sont calculées en utilisant les formules suivantes :

$$\begin{aligned} \texttt{newRow} &= \texttt{row} - \texttt{rowTranslation} \\ \texttt{newCol} &= \texttt{col} - \texttt{colTranslation} \\ \texttt{newOrientation} &= \texttt{orientation} \end{aligned}$$

Écrivez enfin une méthode `applyTransformation` qui applique dans l'ordre une rotation et une translation à une minutie donnée. Vous écrirez aussi une surcharge de cette méthode qui l'appelle sur chacune des minuties d'une liste :

```
public static int[] applyTransformation(int[] minutia, int
    centerRow, int centerCol, int rowTranslation, int
    colTranslation, int rotation)
public static List<int[]> applyTransformation(List<int[]>
    minutiae, int centerRow, int centerCol, int rowTranslation,
    int colTranslation, int rotation)
```

5.3 Méthode `matchingMinutiaeCount`

Il s'agit maintenant de compter le nombre de minuties qui se superposent dans deux listes de minuties données en paramètre. Implémentez la méthode suivante pour réaliser ce traitement :

```
public static int matchingMinutiaeCount(List<int[]> minutiae1,
    List<int[]> minutiae2, int maxDistance, int maxOrientation)
```

Elle prend en paramètre deux listes de minuties et regarde pour chaque entrée m_1 de la première liste s'il y a une entrée m_2 de la deuxième liste qui se trouve au même endroit et avec la même orientation. On considère que deux minuties se superposent si la distance Euclidienne qui les sépare est inférieure ou égale à `maxDistance` et on considère qu'elles ont la même orientation si leur orientation diffère d'au plus `maxOrientation` (compris). Pour rappel la distance Euclidienne entre deux pixels est donnée par la formule $\sqrt{(\texttt{row}_1 - \texttt{row}_2)^2 + (\texttt{col}_1 - \texttt{col}_2)^2}$.

5.4 Méthode `match`

Finalement, écrivez la méthode :

```
public static boolean match(List<int[]> minutiae1, List<int[]>
    minutiae2)
```

Cette méthode doit retourner `true` si les deux listes de minuties proviennent de la même empreinte digitale et `false` sinon. On considère que deux listes de minuties proviennent de la même empreinte si l'on peut trouver au moins `FOUND_THRESHOLD` minuties identiques moyennant l'application de transformations. Vous utiliserez pour `maxDistance` la constante fournie `DISTANCE_THRESHOLD` et pour `maxOrientation` la constante `ORIENTATION_THRESHOLD`. La méthode `applyTransformation` sera donc invoquée sur les minuties m_2 de la deuxième liste pour essayer de repérer si elles peuvent se superposer à toute minutie m_1 de la première. Lors de l'appel à cette méthode :

- `centerRow` et `centerCol` (centre de la rotation) correspondent à la position de m_1 ;
- `rowTranslation` et `colTranslation` (translations verticale et horizontale) sont calculées comme la différence entre les coordonnées ligne et colonne de m_2 et m_1 : c.-à-d. ligne (resp. colonne) de m_2 moins ligne (resp.colonne) de m_1 ;
- `rotation` est la différence d'orientation entre m_2 et m_1. Cependant, comme le calcul des orientations n'est pas parfaitement précis, vous devrez tester avec toutes les entiers compris dans [`rotation - MATCH_ANGLE_OFFSET`, `rotation + MATCH_ANGLE_OFFSET`].

5.5 Tests

Utilisez le programme `Main.java` pour tester vos développements. Les exemples qui y sont fournis ne sont pas exhaustifs : **complétez-les** selon ce qu'il vous semble judicieux de faire, pour tester l'ensemble des méthodes développées dans cette partie du projet.

6 Complément théorique

6.1 Représentation ARGB

La représentation ARGB d'un pixel à l'aide d'un entier se fait de la manière suivante :
La couleur est décomposée en 4 composants ayant une valeur comprise entre 0 et 255 :

- **Alpha** est l'opacité du pixel. Si elle est à 0 alors le pixel est « invisible », peu importe ses autres composants. Si elle est entre 1 et 254 alors il est transparent et laisse passer les couleurs de l'image derrière celui-ci. Si sa valeur est 255, alors le pixel est parfaitement opaque.
- **Red** est la valeur d'intensité de la lumière rouge du pixel.
- **Green** est la valeur d'intensité de la lumière verte du pixel.
- **Blue** est la valeur d'intensité de la lumière bleue du pixel.

En considérant que le bit de poids le plus faible d'un entier ait pour indice 0 et que le bit de poids fort ait pour indice 31, alors ces 4 composants sont placés sur un même entier de telle sorte : les

bits 31 à 24 définissent la valeur alpha, les bits 23 à 16 la valeur rouge, les bits 16 à 8 la valeur verte et les bits 7 à 0 la valeur bleue. Par exemple, si l'on veut représenter un pixel rouge en ARGB on a besoin que le composant alpha et rouge soit au maximum 255. Ce qui nous donne :

Alpha	Rouge	Vert	Bleu
255	255	0	0
0xFF	0xFF	0x00	0x00
11111111	11111111	00000000	00000000
31 24	23 16	15 8	7 0

La manière la plus pratique pour représenter une couleur en Java est d'utiliser l'écriture hexadécimale (plus concise). Par exemple pour définir le pixel rouge nous pouvons écrire :
`int red = 0xFF_FF_00_00;`

6.2 Format binaire et nuance de gris

La représentation d'un pixel en nuance de gris ne nécessite que 1 byte. Cependant, en Java les bytes sont signés et ne peuvent représenter que des valeurs entre -128 et 127. Dans ce projet, on préfèrera donc utiliser des entiers.

Il s'agit d'une valeur comprise entre 0 et 255. Une valeur de 0 représente un pixel noir tandis qu'une valeur de 255 représente un pixel blanc. Et toutes les valeurs entres représentent une nuance de gris.

La représentation binaire ne nécessite qu'un booléen. La valeur `true` représente un pixel noir tandis que la valeur `false`, un pixel blanc.