2 Sujet n° 1 : Comptage de cellules

D'après des discussions avec M. Michel Desvignes.

Objectif

L'objectif de ce projet est de **compter** les cellules présentes dans une image en niveaux de gris en les **isolant** par des méthodes de morphologie mathématique.

2.1 Introduction

Lorsqu'on réalise une culture cellulaire, il est essentiel de pouvoir **compter** les cellules présentes dans la culture à un instant donné. Cela permet de savoir si la culture fonctionne correctement et si de nouvelles cellules s'y développent.

Pour réaliser cette tâche, on peut compter les cellules manuellement. Pour simplifier le processus, on place généralement une grille au dessus de la culture, puis, à l'aide d'un microscope, on compte les cellules dans chaque case de la grille. En sommant finalement tous les comptes intermédiaires, on obtient le nombre total de cellules. Ce processus est long et fastidieux.

Des méthodes de traitement d'images permettent heureusement de réaliser cette tâche **automatiquement**, à partir d'une image de la culture cellulaire comme celle de la Figure 2. C'est l'objectif de ce projet.

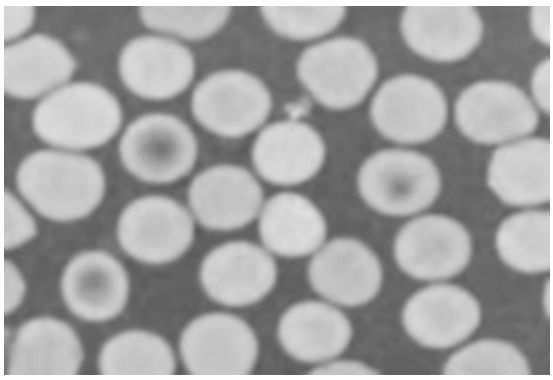


Figure 2 – Image d'une culture cellulaire.

2.2 Étapes préparatoires

2.2.1 Lecture d'une image au format `PGM`

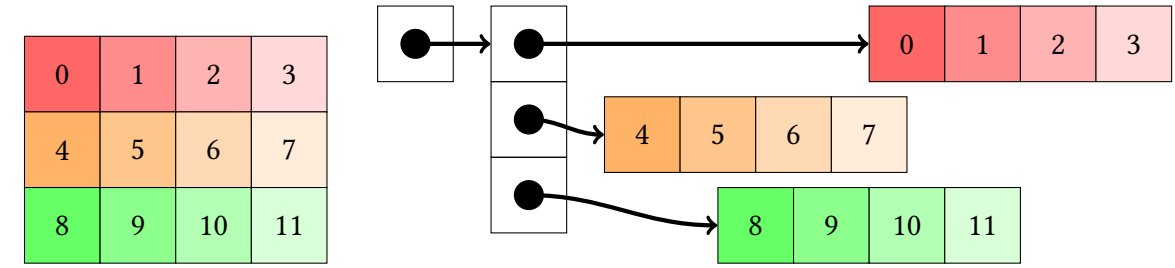
Il vous faudra donc dans un premier temps écrire une fonction qui lit une image au format `PGM` et la stocke en mémoire dans un tableau à deux dimensions. Le format `PGM` binaire est décrit dans l'annexe B.

Lors de l'allocation de la matrice des pixels, vous prendrez garde à réaliser une allocation mémoire **contiguë**. Ceci est illustré dans la Figure 3. On souhaite stocker un matrice de trois lignes et quatre colonnes (voir Figure 3a). On souhaite également pouvoir accéder à un élément de la matrice avec la syntaxe suivante : `matrice[ligne][colonne]`.

La première solution, présentée en Figure 3b, consiste à d'abord allouer un tableau de pointeurs pour les lignes, puis à allouer chaque ligne individuellement. Dans ce cas, lors des allocations successives des lignes, rien ne garantit que l'allocateur mémoire les placera côte à côte en mémoire. Cela peut avoir un coût en performances important, à cause des défauts de cache.

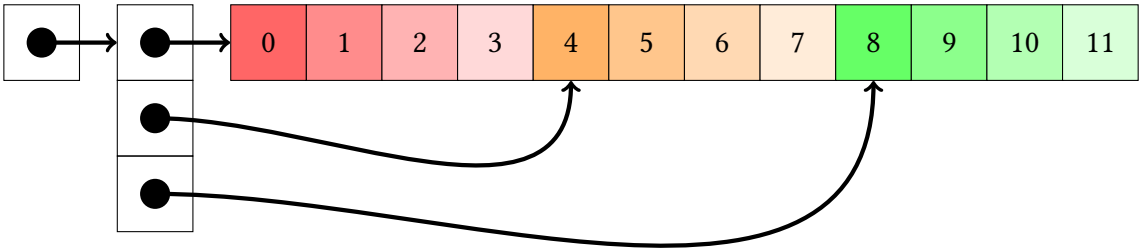
Une autre solution, présentée en Figure 3c, consiste à d'abord allouer un tableau de pointeurs pour les lignes, comme précédemment, puis à faire **une seule allocation** pour l'intégralité de la matrice, sur le pointeur d'indice 0. Il faut ensuite faire pointer les pointeurs d'indice 1 à `nb_lignes` - 1 vers la bonne partie de la zone mémoire allouée.

En plus des raisons de performances évoquées au dessus, le fait de réaliser une allocation contiguë permet, lors de la lecture ou l'écriture d'un fichier, de travailler avec l'intégralité des données à la fois, en les considérant comme un seul bloc binaire.



(a) Matrice 3×4 à stocker

(b) Allocation non-contiguë en mémoire



(c) Allocation contiguë en mémoire

FIGURE 3 – Allocations mémoire (adapté de http://c-faq.com/aryptr/dynmuldimary.html).

Lors de la lecture de l'image dans un fichier `PGM`, il peut être utile de conserver d'autres informations utiles, comme le nombre de lignes et de colonnes. Vous définirez une structure, voire un nouveau type, adaptés pour représenter les images.

Vous écrirez ensuite une fonction dont le rôle sera d'écrire une image dans un fichier. Vous commencerez par écrire directement l'image lue. Cela vous permettra de vérifier que la lecture et l'écriture fonctionnent correctement.

Vous pourrez vérifier que les deux fichiers sont identiques avec la commande suivante :

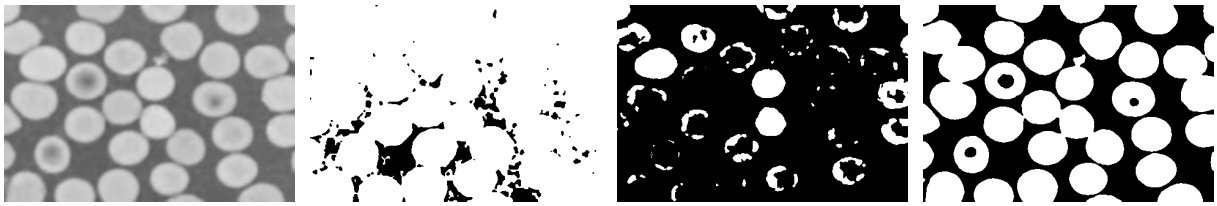
```
$ diff image.pgm copie.pgm
```

Vous n'oublierez pas d'écrire des fonctions pour libérer la mémoire à la fin du programme !

2.2.2 Seuillage manuel

Le premier traitement à appliquer à l'image consiste à la seuiller, c'est-à-dire à passer d'une image en niveaux de gris, dont la valeur des pixels va de 0 **à** 255, à une image binaire (ou "noir et blanc"), dont la valeur des pixels est de 0 **ou** 255.

Le seuil définit, pour chaque pixel de l'image, s'il sera noir ou blanc après seuillage. Fixer correctement la valeur de ce seuil est important, comme illustré en Figure 4, où l'image de la Figure 4a est seuillée avec différentes valeurs de seuil. Si le seuil est trop bas (Figure 4b) ou trop haut (Figure 4c), on perd des détails de l'image. Un seuil bien choisi (Figure 4d) permet de bien séparer les objets, qui deviennent blancs, de l'arrière-plan, qui devient noir.



(a) Image de départ en niveaux de gris (b) Image seuillée avec un seuil trop bas (c) Image seuillée avec un seuil trop haut (d) Image seuillée avec un seuil bien choisi

FIGURE 4 – Image seuillée avec différentes valeurs de seuil.

Écrivez une fonction qui réalise le seuillage d'une image et le programme de test associé. Vous écrirez le programme de test de façon à ce qu'il puisse prendre en paramètre le nom de l'image à traiter. Vous pourrez faire en sorte que le seuil puisse être passé en paramètre lui aussi. Par exemple, pour un seuil fixé à 150, voici la commande que vous lancerez :

```
$ ./bin/test_seuillage image.pgm 150
```

2.2.3 Seuillage automatique : méthode d'Otsu

La méthode d'Otsu permet de déterminer automatiquement la valeur du seuil à partir de l'histogramme de l'image.

Histogramme L'histogramme d'une image représente la distribution des valeurs des pixels de l'image. La Figure 5 montre une image et son histogramme. Sur cet histogramme, on observe deux modes, l'un autour de la valeur 100 et l'autre autour de la valeur 200. Le premier correspond à l'arrière-plan sombre, le second aux cellules plus claires. La méthode d'Otsu permet de déterminer la valeur du seuil qui sépare au mieux ces deux modes.

Écrivez une fonction qui calcule l'histogramme de l'image et le stocke dans un tableau.

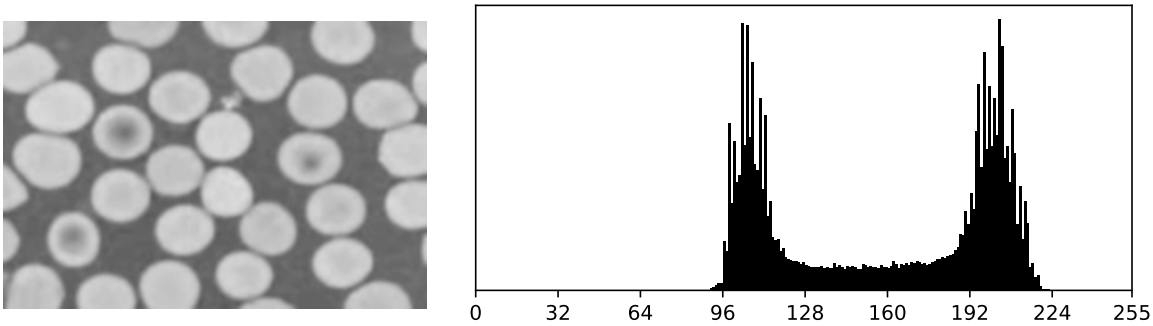


FIGURE 5 – L'image et son histogramme.

Méthode d'Otsu La méthode d'Otsu énumère toutes les valeurs possibles du seuil, de 0 à 255, et trouve celle qui maximise la variance inter-classe, c'est-à-dire celle qui sépare le mieux les deux modes. La variance inter-classe σ_b^2 est définie dans l'équation (1).

$$\sigma_b^2(s) = \omega_0(s)\omega_1(s)[\mu_0(s) - \mu_1(s)]^2 \tag{1}$$

où $\omega_0(s)$ et $\omega_1(s)$ sont les probabilités de classe, définies dans l'équation (2)

$$\omega_0(s) = \sum_{i=0}^{s-1} h(i) \quad \text{et} \quad \omega_1(s) = \sum_{i=s}^{255} h(i) \tag{2}$$

et $\mu_0(s)$ et $\mu_1(s)$ sont les moyennes empiriques des classes, définies dans l'équation (3).

$$\mu_0(s) = \frac{\sum_{i=0}^{s-1} ih(i)}{\omega_0(s)} \quad \text{et} \quad \mu_1(s) = \frac{\sum_{i=s}^{255} ih(i)}{\omega_1(s)} \tag{3}$$

Le seuil calculé est la valeur de s pour laquelle la variance inter-classe $\sigma_b^2(s)$ est maximale.

Vous écrirez une fonction qui réalise le seuillage automatique, puis vous écrirez un programme de test qui permet de seuiller automatiquement une image dont le nom est passé en paramètre.

2.2.4 Utilitaire de génération d'images de test

Pour tester le fonctionnement de vos programmes, nous vous fournissons un utilitaire écrit en Python qui permet de générer un image `PGM`.

Par exemple, pour générer une image de 3×3 pixels noire avec un pixel blanc au centre, vous lancerez la commande suivante :

```
$ python2.7 generate_test_image.py NNN NBN NNN
```

2.3 Version minimale

2.3.1 Opérations booléennes

Nous aurons besoin pour la suite de réaliser des opérations booléennes, ou logiques, entre deux images. Nous définissons donc les trois opérations suivantes : intersection (ET logique), union (OU logique) et OU exclusif.

Les tables de vérité de ces opérations sont rappelées dans la Table 1.

im_A[i][j]	im_B[i][j]	$im_{INTERSECTION}$[i][j]	im_{UNION}[i][j]	im_{XOR}[i][j]
N	N	N	N	N
N	B	N	B	B
B	N	N	B	B
B	B	B	B	N

TABLE 1 – Tables de vérité des opérations intersection, union et OU exclusif.

Toutes ces opérations pré-supposent que les images d'entrée sont de même dimension. Il serait intéressant de le vérifier grâce à `assert`.

⚠ Mot-clé du langage C

`union` est un mot-clé du langage C, vous ne pouvez donc pas l'utiliser comme nom de fonction.

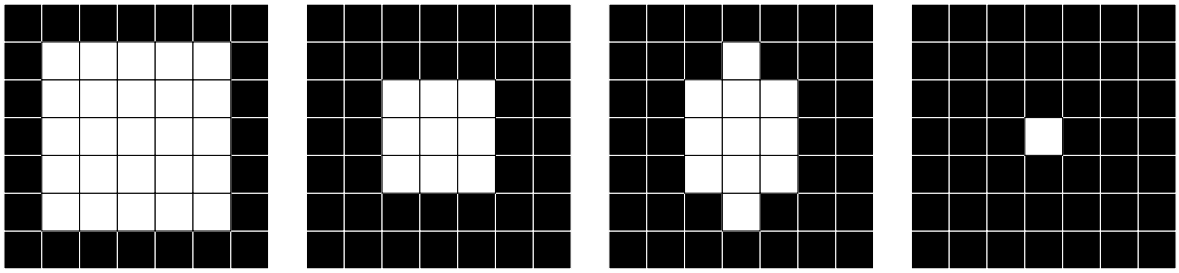
2.3.2 Opérations morphologiques basiques

La morphologie mathématique est une branche des mathématiques très utile pour le traitement des images. Elle définit des opérations applicables à des images, en particulier celles en noir et blanc, ce qui nous intéressera dans ce projet. Nous nous limiterons aux opérations simples dans le cadre de ce projet.

Érosion L'opération d'érosion consiste à calculer le **minimum local** de chaque groupe de 3×3 pixels dans l'image de départ. Si dans un groupe de 3×3 pixels **tous** les pixels sont blancs, alors le pixel en sortie de l'opération est blanc. Si dans un groupe de 3×3 pixels il y a **au moins un** pixel noir, alors le pixel en sortie de l'opération est noir.

L'opération d'érosion est illustrée en Figure 6 où deux exemples de son application sont donnés. L'érosion réduit la surface des zones blanches dans l'image.

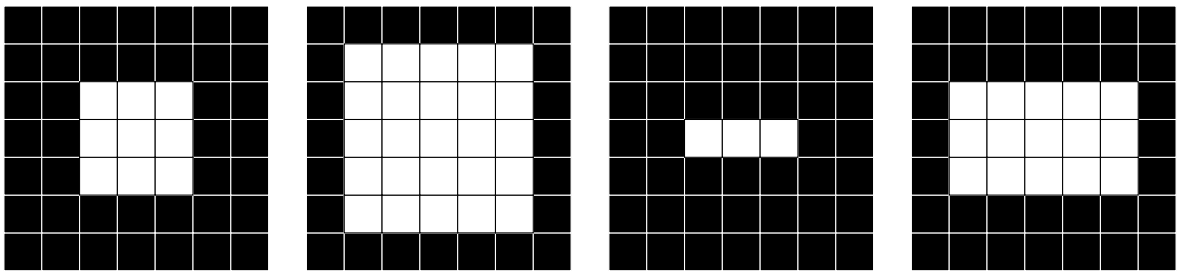
Dilatation L'opération de dilatation consiste à calculer le **maximum local** de chaque groupe de 3×3 pixels dans l'image de départ. Si dans un groupe de 3×3 pixels **tous** les pixels sont noirs, alors le pixel en sortie de l'opération est noir. Si dans un groupe de 3×3 pixels il y a **au moins un** pixel blanc, alors le pixel en sortie de l'opération est blanc.



(a) Avant érosion (b) Après érosion (c) Avant érosion (d) Après érosion

FIGURE 6 – Deux exemples d'application de l'opération d'érosion.

L'opération de dilatation est illustrée en Figure 7 où deux exemples de son application sont donnés. La dilatation augmente la surface des zones blanches dans l'image.



(a) Avant dilatation (b) Après dilatation (c) Avant dilatation (d) Après dilatation

FIGURE 7 – Deux exemples d'application de l'opération de dilatation.

2.3.3 Reconstruction

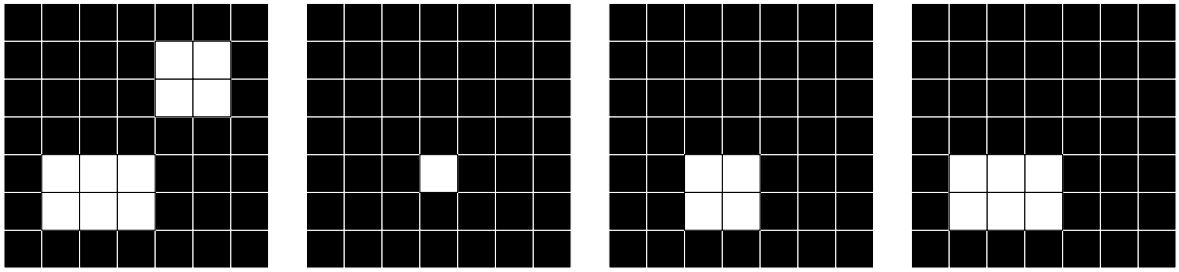
Une autre opération importante de morphologie est la reconstruction. Cette dernière consiste à faire croître une image **graine** dans l'image. On dilate la graine dans les zones blanches de l'image, jusqu'à ce que ces dernières soient complètement remplies. La reconstruction est décrite dans l'Algorithme 1.

Algorithme 1 Reconstruction morphologique

```
graine_dilatee = dilatation(graine)
image_reconstruite = intersection(image, graine_dilatee)
faire
    graine_dilatee = dilatation(image_reconstruite)
    image_reconstruite = intersection(image, graine_dilatee)
tant que image_reconstruite change
```

Pour écrire cette fonction, vous aurez besoin de comparer deux images, afin de savoir si `image_reconstruite` a changé. Vous utiliserez `memcmp` pour ça.

La Figure 8 illustre l'opération de reconstruction sur un exemple. À chaque itération de la boucle `while`, la graine croît dans les zones blanches de l'image, jusqu'à remplir complètement la zone du bas. La boucle s'arrête ensuite car l'image reconstruite ne change plus. La reconstruction est terminée : on a bien "reconstruit" la zone blanche du bas de l'image.



(a) Image (b) Graine (c) Première répétition (d) Deuxième répétition

FIGURE 8 – Illustration des étapes de l'opération de reconstruction morphologique.

L'opération de reconstruction définie ci-dessus est très utile pour "sélectionner" certains éléments de l'image. L'effet que la reconstruction aura sur l'image dépend donc de la graine.

2.3.4 Suppression des cellules au bord

Pour supprimer les cellules au bord de l'image, on applique l'opération de reconstruction en utilisant comme graine un cadre blanc d'un pixel au bord de l'image, sur fond noir. La Figure 9 montre quelques étapes de cette reconstruction. Les cellules au bord sont "sélectionnées" par la reconstruction. On termine en faisant le OU-exclusif entre l'image reconstruite et l'image initiale (Figure 10).

FIGURE 9 – Étapes de la reconstruction d'un cadre blanc dans l'image seuillée.

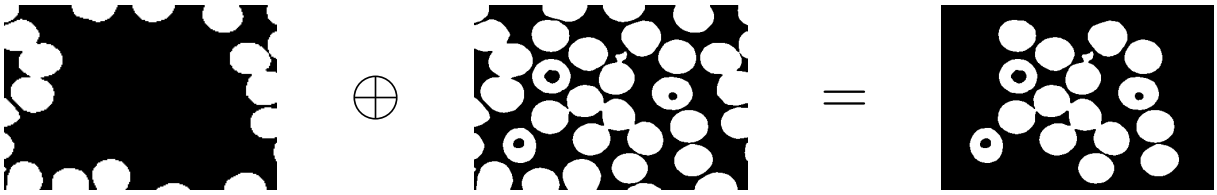


FIGURE 10 – Suppression des cellules au bord par OU-exclusif avec un cadre reconstruit.

2.3.5 Bouchage des trous

Certaines cellules de l'image seuillée présentent un **trou**. Ce trou correspond à une zone sombre au milieu de ces cellules dans l'image initiale en niveaux de gris. La Figure 11 montre trois cellules pour lesquelles ce phénomène se produit.

(a) Image seuillée (b) Image en niveaux de gris

FIGURE 11 – Mise en évidence dans l'image seuillée et l'image en niveaux de gris des cellules présentant un trou dans l'image seuillée.

Pour remplir des cellules avec un trou, on applique l'opération de reconstruction sur l'image **inversée** avec la même graine que précédemment : un cadre blanc d'un pixel sur fond noir, de la taille de l'image. Cela remplit progressivement l'espace entre les cellules, ne laissant finalement que les cellules elles-mêmes.

La Figure 12 montre quelques étapes de la reconstruction du cadre dans l'image inversée. Le fond est progressivement rempli, ne laissant finalement que des cellules sans trou. Il faut finalement inverser à nouveau l'image pour obtenir l'image seuillée initiale avec les trous bouchés.

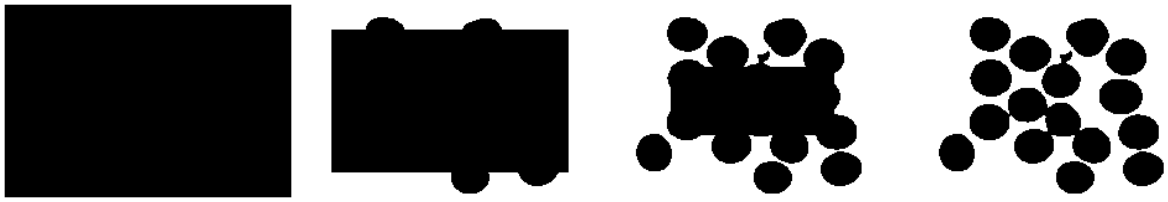


FIGURE 12 – Étapes de la reconstruction d'un cadre blanc dans l'image seuillée.

2.3.6 Érosion manuelle

Dans l'image finalement obtenue, certaines cellules se touchent. C'est un problème pour les compter. Pour séparer ces cellules, on érode l'image un certain nombre de fois, fixé manuellement. La Figure 13 montre une image avec les cellules bien séparées.

Nous avons fait le choix ici de réaliser cette érosion manuelle à la fin, mais nous aurions aussi pu la réaliser avant de supprimer les cellules au bord.

2.3.7 Comptage des composantes connexes par parcours en profondeur

La dernière étape consiste à compter les composantes connexes de l'image. Deux pixels appartiennent à la même composante connexe s'ils sont en contact par un de leurs quatre

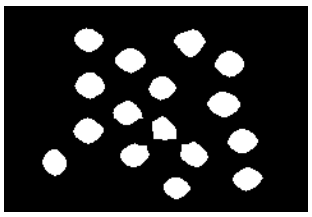


FIGURE 13 – Cellules bien séparées après érosion.

côtés : on considère donc la 4-connectivité. On considèrera que des pixels qui sont en contact par un angle n'appartiennent pas à la même composante connexe, ce qui serait le cas si on considérait la 8-connectivité.

Pour compter les composantes connexes, on se basera sur un algorithme de parcours en profondeur (DFS pour *depth-first search*) récursif sur les pixels blancs de l'image. Celui-ci est décrit dans l'Algorithme 2.

Algorithme 2 Algorithme de recherche en profondeur dans les pixels d'une image

```
1: fonction DFS(image, ligne, colonne)
2:     si image[ligne][colonne] n'a pas encore été visité alors
3:         si image[ligne][colonne] est blanc alors
4:             Marquer image[ligne][colonne] comme visité
5:             DFS(image, ligne-1, colonne)        ▷ Visite du voisin du haut
6:             DFS(image, ligne+1, colonne)        ▷ Visite du voisin du bas
7:             DFS(image, ligne, colonne-1)        ▷ Visite du voisin de gauche
8:             DFS(image, ligne, colonne+1)        ▷ Visite du voisin de droite
```

Une fois cet algorithme implémenté, on comptera les composantes connexes en effectuant une recherche à partir de tous les pixels de l'image. On prendra garde de ne pas visiter des pixels que l'on aura déjà visités. Chaque nouvelle recherche réussie correspondra à une nouvelle composante connexe.

2.4 Pour aller plus loin

2.4.1 Érodés ultimes

Dans la section 2.3.6, nous avons réalisé une érosion **manuelle** : nous avons fixé arbitrairement le nombre de répétitions afin de séparer suffisamment les cellules qui se touchaient, sans les faire disparaître. Il serait plus intéressant de réaliser ces érosions successives en déterminant le nombre d'itérations de manière automatique.

Un problème qui apparaît alors est que certaines cellules sont plus grosses que d'autres. En conséquence, les érosions successives supprimeront les plus petites cellules, tandis qu'elles conserveront les plus grosses. Comment savoir alors quand arrêter l'érosion ?

Une solution consiste à identifier les **érodés ultimes**, c'est-à-dire, pour chaque cellule, sa dernière version érodée avant qu'elle ne disparaisse. L'Algorithme 3 décrit cette méthode. Son principe est le suivant :

- on érode l'image (ligne 4) puis on reconstruit la version érodée dans l'image de départ (ligne 6),
- on détecte avec un OU-exclusif les éléments que l'érosion avait fait disparaître (ligne 7) : ce sont les résidus,
- on accumule les résidus avec une union (ligne 8),
- l'image courante devient l'image érodée (ligne 9),
- on répète ces opérations tant que l'image courante n'a pas été complètement érodée (ligne 10), c'est-à-dire tant qu'elle n'est pas vide.

Pour une cellule de taille suffisante, l'érosion puis la reconstruction réalisées aux lignes 4 et 6 s'annulent. En revanche, pour un érodé ultime, la reconstruction est impossible puisque l'érosion l'a fait disparaître. Ces deux cas sont discriminés par l'opération OU-exclusif puis les cas où la cellule a disparu, correspondant aux érodés ultimes, sont accumulés.

Algorithme 3 Érodés ultimes

```
1: erodes_ultimes, erodee, residus, reconstruite : des images vides
2: courante = image
3: faire
4:    erodee = erosion(courante)
5:    Effacer reconstruite
6:    reconstruite = reconstruction(courante, erodee)
7:    residus = OU_exclusif(reconstruite, courante)
8:    erodes_ultimes = union(residus, erodes_ultimes)
9:    échanger erodee et courante
10: tant que courante n'est pas vide
```

La Figure 14 montre un exemple d'érodés ultimes.

FIGURE 14 – Érodés ultimes.