Projet Programmation S6
Sujet n°1 : Comptage de cellules

Compte-rendu final

L'objectif de ce projet était de compter les cellules présentes dans une image en niveaux de gris en les isolants par des méthodes de morphologies mathématiques.

État du logiciel

La version minimale du projet consistait à compter les cellules sur une image préalablement seuillée en noir et blanc en utilisant des opérations morphologiques basiques telles que l'érosion, la dilatation et la reconstruction. L'étape suivante était de compter les composantes connexes de l'image par 4-connectivité. Toutes ces fonctionnalités sont présentes dans notre projet et opérationnelles. Il est alors possible de seuiller une image en niveaux de gris soit manuellement en renseignant le seuil, soit automatiquement et de compter le nombre de cellules présentes à l'aide d'une érosion manuelle.

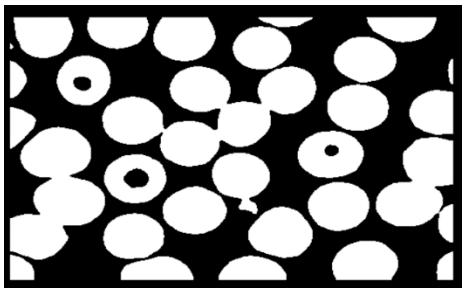

Image seuillée

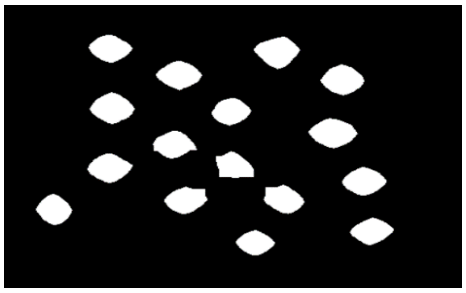

Image érodée

On peut développer les différentes étapes qui nous amène au comptage des composantes connexes :

La première étape consiste à créer différentes fonctions qui seront utiles pour la suite : des opérations booléennes dans *booleenne.c* (AND, OR, XOR) ainsi que des opérations morphologiques dans *morphologique.c* (érosion, dilatation, reconstruction). On peut noter que pour les fonctions érosion et dilatation qui, pour chaque pixel, doivent consulter les pixels voisins, nous avons utilisé une méthode dite "miroir" pour les pixels au bord. Elle consiste à ajouter un cadre d'un pixel autour de l'image qui est une copie des pixels du bord. De plus, nous avons ajouté un fichier *operation.c* qui regroupe des fonctions secondaires comme la création d'une image vide, la copie d'une image, la comparaison de deux images identiques ou non... Dans cette première étape, tout est fonctionnel.

Les autres étapes nécessaires jusqu'au comptage des cellules se trouvent dans le fichier *comptage.c*.

La deuxième étape consiste à supprimer les cellules au bord. L'implémentation choisie implique que les cellules "touchant" les cellules de bord après le seuillage sont aussi supprimées.

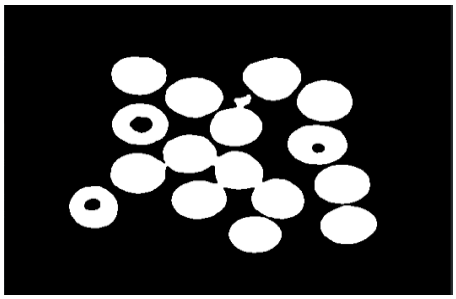

Image sans bord

L'étape suivante consiste à boucher les "trous" qui sont apparus dans les cellules avec le seuillage :

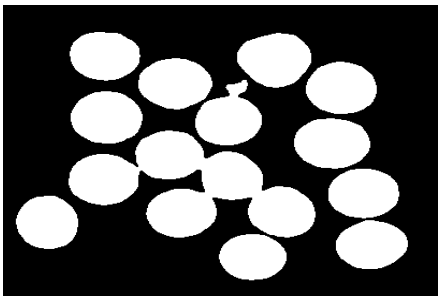

Image sans trous

Ensuite vient l'étape d'érosion des cellules. En effet, on remarque que certaines cellules se touchent, ce qui est un problème lors du comptage de composantes connexes, qui vient juste après. Dans la version minimale du projet, on propose à l'utilisateur de choisir le nombre d'érosions, qui doit être suffisamment grand pour éviter que les cellules se touchent, mais pas trop grand pour éviter que des cellules disparaissent.

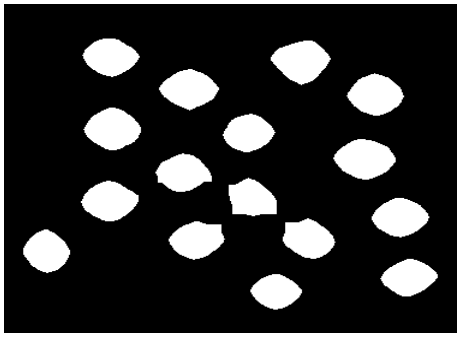

Image érodée (8 érosions)

Enfin la dernière étape est le comptage des composantes connexes, divisée en 2 fonctions : la fonction principale de comptage, ainsi qu'un algorithme de parcours en profondeur. Toutes les étapes décrites ci-dessus sont fonctionnelles.

De plus, une partie de la version avancée a été traitée : il est possible d'éroder de manière automatique avec la fonction *érodés ultimes* et les cellules comptées sont également numérotées.

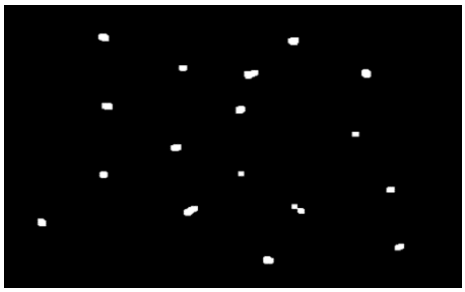

Image érodée

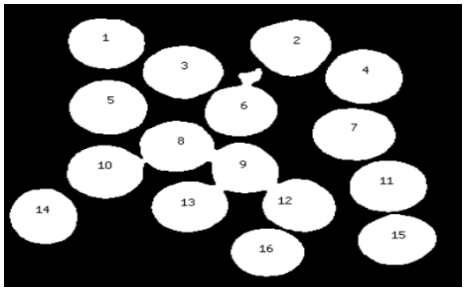

Image numérotée

Cependant, nous n'avons pas utilisé l'algorithme de Hoshen-Kopelman et la structure de données *union-find* proposés dans la version avancée, ni du marquage des cellules comptées par coloration.

Structure

Une seule structure de données est utilisée dans notre projet. Il s'agit de la structure que nous avons nommé *Image* qui comporte toutes les informations nécessaires pour le traitement d'une image du type PGM : sa taille (nombre de lignes et nombre de colonnes), et la valeur de chacun des pixels renseignés dans un tableau à deux dimensions. Pour chaque pixel, on utilise le type *unsigned char* qui permet de représenter une valeur entière de 8 bits, autrement dit un entier entre 0 et 255. Ce choix est justifié par le fait que nous traitons une image en niveaux de gris. Pour gérer les tableaux de pixels et la manipulation dynamique de la mémoire, on utilise des pointeurs de pointeurs, ici des *unsigned char**,* ce qui permet l'allocation contiguë.

Méthodologie

Notre projet comporte trois tests. Deux permettent de seuiller une image du type PGM, une manuellement en renseignant le seuil, et l'autre automatiquement. Ce sont respectivement *seuillage_manuel* et *seuillage_automatique*. Le troisième test est l'algorithme principal qui compte le nombre de cellules présentes dans une image du type PGM et se nomme *programme*. Pour exécuter chacun de ces tests, il faut taper la commande depuis le terminal *./bin/test* suivie du ou des paramètres suivants les tests. Les tests *seuillage_automatique* et *programme* ne prennent en paramètre que l'image du type PGM située dans le répertoire *fichiers* puis *images*. On y accède avec la commande *fichiers/images/image.pgm*. En plus de l'image, le test *seuillage_manuel* prend un deuxième paramètre et il s'agit du seuil. Par exemple pour exécuter le test *seuillage_maunel*, on écrira dans le terminal la commande : *./bin/seuillage_manuel fichiers/images/image.pgm seuil*.

Pendant tout le projet, nous avons également dû vérifier que nos différentes fonctions étaient correctes. C'est pourquoi nous avons fait le choix d'enregistrer les images correspondantes à chaque étape dans le dossier *fichier/traitement*. Comme les étapes sont faciles à vérifier pour un humain pour une seule image, nous pouvions vérifier le fonctionnement du programme en regardant les images, en supposant que le programme fonctionnerait avec d'autres images. De temps en temps, pour les fonctions intermédiaires, nous avons aussi utilisé la commande *printf* ou d'autres outils pour vérifier l'état de nos fonctions.

Performances et mémoire

Le temps d'exécution du code est relativement correct d'autant plus qu'à chaque étape du programme on crée une nouvelle image pour suivre la progression (environ 5 secondes pour le fichier *cellules.pgm* en utilisant la méthode d'érosion automatique). Lors de l'écriture du code, nous avons pensé à réduire au maximum les calculs et à optimiser certaines de nos fonctions dans le but de rendre le tout plus performant. Par exemple, la méthode d'Otsu a été optimisée comme indiqué dans le sujet. En ce qui concerne la mémoire, nous avons alloué le minium de fois possible et libéré la mémoire dès qu'on ne l'utilisait plus (en tout 141 807 108 bits alloués). Ainsi avec l'outil de développement Valgrind, on vérifie qu'on a bien libéré autant de fois que nous avons alloué.

Outils de développement

Lors de ce projet, nous avons utilisé plusieurs outils de développement tels que la compilation, Git ou encore Valgrind. La compilation ne donne aucune erreur. L'outil Git a été très souvent exploiter et tous les fichiers sont à jour. Valgrind nous a permis de mettre en évidence certains défauts de mémoire et ainsi les corriger. Cependant, nous n'avons pas utilisé l'outil Gdb car nous n'en avons pas eu besoin.

Organisation

Le projet en équipe s'est absolument bien déroulé et dans une bonne entente. Etant deux dans l'équipe il était très facile de se partager le travail et de se retrouver pour mettre en commun. Une personne s'est occupée de toute la partie de lecture d'image PGM, de la partie seuillage ainsi que de la partie numérotation des cellules. L'autre s'est concentrée sur toutes les opérations morphologiques nécessaire au comptage des cellules. Ensemble nous avons réfléchi à des possibles améliorations et à la finalisation du projet.

Notes

La participation et l'engouement des deux membres de l'équipe étaient similaires. Nous proposons alors la note de 10 à l'un comme à l'autre.

Conclusion

En conclusion, ce projet nous a permis d'acquérir la compétence « Coopérer dans une équipe ou en mode projet » en respectant les délais et les consignes imposés. De plus, grâce à ce projet en particulier nous avons beaucoup appris sur le traitement d'image et toutes les subtilités qui l'entourent.