Université de Strasbourg

Année Universitaire 2019-20

Traitement d'Images et Géométrie Discrète

Méline Bour-Lang
Morgane Ritter
Nathan Roth

Table des matières

Introduction

Objectif

Pour ce projet, nous avons choisi l'article portant sur le calcul de l'arbre des formes en temps linéaire [1]. L'algorithme cité est capable de traiter des images *nD*. Puisque les exemples du papier se concentrent sur le cas où *n=2*, nous avons préféré traiter cette situation uniquement, où nous pouvions vérifier que nos résultats correspondaient bel et bien à ceux attendus. L'algorithme présenté par notre article ne traite que des images en niveau de gris.

L'objectif est de réaliser un arbre des formes, tel qu'illustré en **figure 1**, en un temps linéaire.

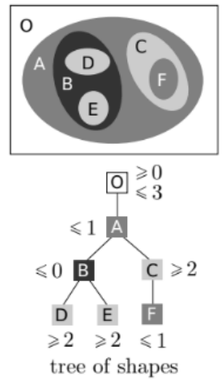

Figure 1 : exemple d'arbre des formes [1]

L'algorithme est découpé en trois étapes principales, que nous avons réalisées une après l'autre : l'interpolation de l'image, le tri des pixels obtenus et enfin la construction de l'arbre lui-même.

Présentation du programme

Nous avons opté pour le langage C++, que nous connaissions tous les trois, et qui nous a semblé approprié pour obtenir un algorithme performant. Il nous permettait également de profiter des nombreuses structures de donnée de la STL.

Nous nous reposons sur la librairie fournie [2], pour importer les images au format PGM.

Nous proposons également une représentation interactive de l'arbre des formes. Pour une image 2D en noir et blanc fournie en entrée, il est possible de voir la chaîne de parenté d'un pixel en passant

la souris sur ce dernier. L'affichage est géré par la librairie SFML, qui permet un prototypage relativement rapide, et qui offre des outils de dessin 2D.

Pour tester notre programme, il faut d'abord installer la librairie SFML :

sudo apt-get install libsfml-dev

Pour lancer le programme :

make clean && make && ./tos <filename.pgm> --display

Les différentes options disponibles sont listées via

./tos -h

Réalisation

Interpolation de l'image

Nous ajoutons tout d'abord à l'image un fond d'une largeur d'un pixel, avec une valeur équivalente à l'intensité médiane de l'image. Elle est ensuite interpolée, comme l'illustre la **figure 2**. Cette opération a pour but de créer un espace continu : le tri peut ainsi passer « entre » les pixels de l'image.

L'illustration de la figure se rapproche de la figure 5 proposée dans le papier [1]. Nous avons néanmoins réduit sa taille pour travailler.

L'interpolation génère de nouveaux pixels, auxquels nous avons choisi de donner différents types en fonction de leur construction :

- si un pixel appartient à l'image d'origine, il s'agit d'un pixel **Original**.
- si un pixel est généré par l'interpolation, il s'agit d'un pixel **New**.
- si un pixel représente un intervalle de valeurs (qui correspond à l'intensité de son voisinage), il s'agit d'un *interpixel*. S'il prend en compte les intensités de 2 voisins, nous l'appelons **Inter2** ; s'il en considère 4, **Inter4**.

L'interpolation est faite en plusieurs étapes. Tout d'abord, un pixel **New** est intercalé entre chaque pixel existant. Son intensité est fixée au maximum des pixels voisins. En effet, selon [1],

$$\forall\, h \in X, U(h) = \max\big(U(h') : h' \in st\big(cl(h)\big) \cap D\big)\; if\; h \in \frac{1}{2} H_1^n \backslash D$$

où U est l'image interpolée et h le pixel dont on recherche la valeur U(h). Cette valeur est le maximum des valeurs des pixels constituant l'ouverture de la fermeture morphologique contenant h.

Le calcul de l'intensité de certains de ces pixels **New** (ceux situés en diagonale des pixels d'origine) nécessite d'avoir complété les voisins des pixels **Original** en 4-connexité.

L'étape suivante consiste à intercaler des *interpixels* de façon similaire, comme précédemment en deux passages. Chaque *interpixel* stocke les valeurs intermédiaires séparant ces voisins. Les **Inter2** sont construits en premier : ils utilisent, selon leur position, les pixels du haut et du bas, ou bien de la gauche et de la droite. Les **Inter4** font de même en considérant l'ensemble de leurs voisins.

Ainsi, si un **Inter2** est intercalé entre deux pixels de valeurs 4 et 6, il stockera les valeurs 4, 5 et 6. Selon [1],

$$\forall\, h \in X, U(h) = span(U(h') : h' \in st(h) \cap D)\; if\; h \in X \backslash \frac{1}{2} H_1^n$$

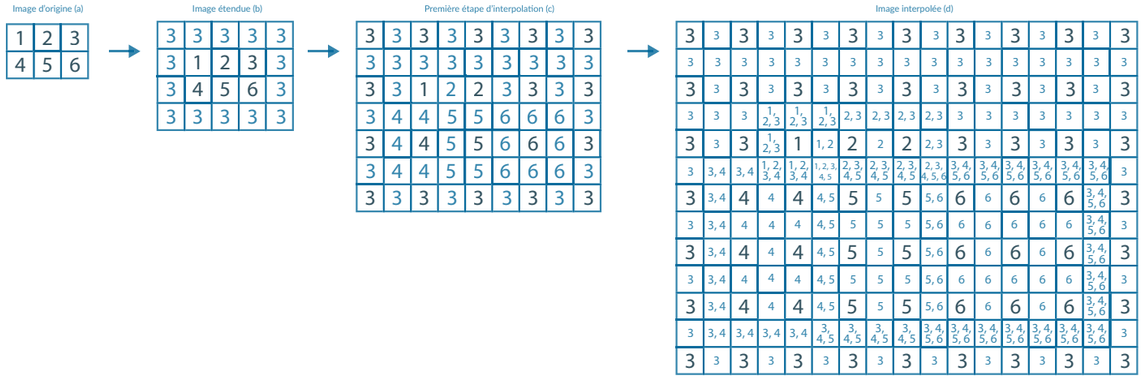

Figure 2 : Construction de l'image interpolée

Dans notre programme, l'image finale est stockée dans un tableau 1D de cellules. Une cellule est un pixel original, nouveau ou un *interpixel*. Pour optimiser la mémoire, nous ne stockons en fait que les valeurs minimale et maximale de l'intervalle d'un *interpixel*.

Tri des pixels de l'image

Suite à l'interpolation, nous pouvons désormais trier les pixels de l'image, grâce à la méthode de la file d'attente hiérarchisée, qui constitue en fait une « file d'attente de files d'attentes », telle qu'illustrée en **figure 3**.

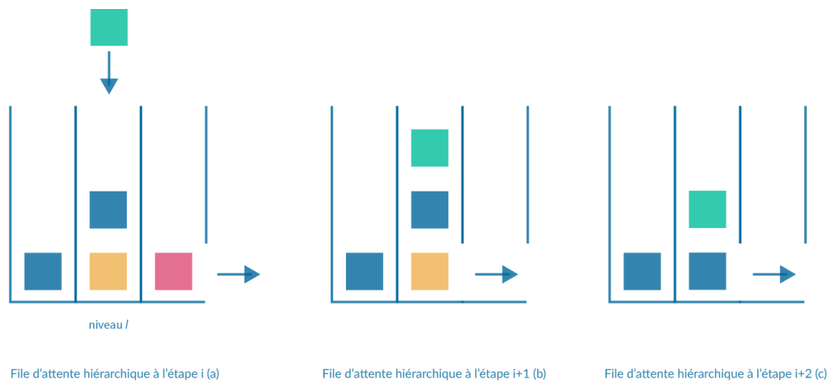

Figure 3 : file d'attente hiérarchique

Le premier pixel ajouté à la file d'attente hiérarchique appartient au bord de l'image. Ensuite, tant que celle-ci n'est pas vide, on choisit la prochaine file d'attente non vide l à traiter, ce qui nous donne un pixel. Celui-ci est ajouté au tri. On étudie son voisinage pour en ajouter les pixels à la file d'attente, et on recommence.

Chaque pixel est ajouté à un niveau l de la file d'attente en fonction de sa valeur (ou de sa valeur la plus proche de l, dans le cas des interpixels).

On parvient ainsi à trier les pixels des formes « externes » aux formes « internes ».

La difficulté de ce tri se situait au niveau de son évaluation : nous n'avions pas d'exemple d'exécution et il était plutôt compliqué de savoir si le résultat était exact ou non.

Nous avons choisi de représenter cette structure comme une table associative de files. La table associative a pour clé le niveau de la file correspondante. En effet, l'algorithme présenté requiert un accès indicé aux éléments (files) de la file de base, et cette même file ne semble finalement pas adopter un véritable comportement de file (voir opération **PRIORITY POP** et **PRIORITY PUSH** dans l'Algorithme 2 de [1]).

Construction de l'arbre

L'algorithme construisant l'arbre (*union-find*) prend en entrée le tri des pixels produit dans la partie précédente. Les pixels sont parcourus suivant le sens inverse du tri (on commence par les formes « internes » en allant vers les formes « externes ») ; on part ensuite des feuilles de l'arbre. La **figure 4** représente le déroulement de l'algorithme sur un exemple simple.

L'algorithme étant détaillé dans le papier, nous n'avons pas eu trop de difficultés à l'implémenter. Cependant, même après l'avoir déroulé à la main, nous n'avons pas compris l'intérêt d'une des variables (*zpar*). La compréhension de l'algorithme était donc ici l'aspect le plus compliqué.

Canonisation

La canonisation a pour objectif de produire un arbre des formes plus succinct. En effet, il n'est pas nécessaire que chaque forme de l'arbre soit représentée par l'ensemble de ses pixels : un seul d'entre eux suffit.

La canonisation de notre arbre a été compliquée par le fait que notre article [1] ne donnait pas d'explications détaillées, mais faisait référence à un algorithme mentionné dans l'article [3]. Ce dernier n'utilisant pas la même structure de données et n'ayant pas besoin de traiter les *interpixels*, son application directe ne donnait pas de résultats satisfaisants et s'est révélé être une étape difficile du développement.

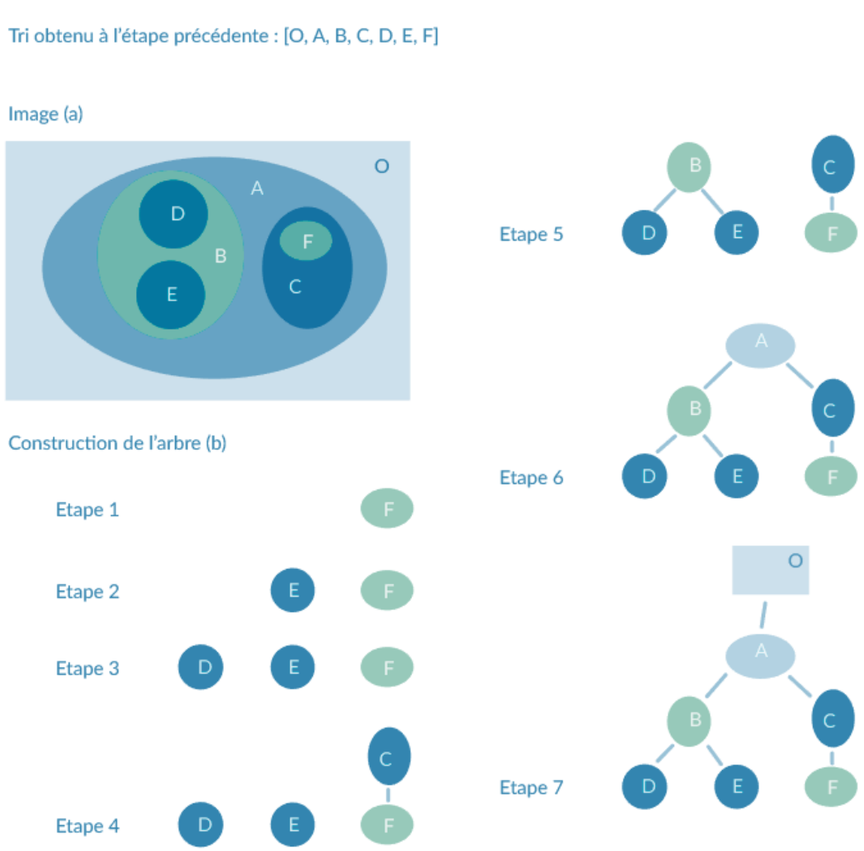

Figure 4 : Construction de l'arbre des formes, schéma reproduit à partir de [1]

Après avoir travaillé sur la question, nous avons compris que le niveau l, mémorisé lors de l'étape de tri pour chacun des pixels traités, permet de traiter tous les pixels et interpixels de façon équivalente. Il représente en quelque sorte l'intensité correspondant à chacun.

Nous avons donc choisi une autre solution, qui consiste à, pour définir le parent canonisé d'un pixel, remonter tous ses parents jusqu'à tomber sur un parent de niveau l différent. Cela permet d'obtenir un pixel référent par forme.

Dans le cas de la forme la plus externe de l'image, cette solution ne peut pas fonctionner, puisqu'on ne trouvera pas de parent de niveau différent. Il a donc fallu gérer ce cas en vérifiant si le pixel courant et son parent n'était pas un seul et même pixel.

Avec cette solution, le chemin de parenté était, la plupart du temps, uniquement constitué

d'*interpixels*. Il a donc été nécessaire, avant de déterminer l'élément canonique, de vérifier que celui trouvé soit bien de type **Original**. Si ce n'est pas le cas, nous remontons ses ancêtres jusqu'à trouver un pixel de type **Original**, qui devient donc l'élément canonique de la forme courante.

Désinterpolation

Assigner des types à chacun de nos pixels a facilité cette étape : il suffit de supprimer tous les pixels de l'image qui ne sont pas de type **Original**. Le chemin de parenté ayant été « nettoyé » des autres types de pixel, il n'y a aucun traitement algorithmique dans cette étape.

Tests et Analyse

Description des jeux d'essai

Différents types d'images ont pu être produits notamment :

- Une reproduction de l'image de la **figure 2**
- Une reproduction de l'image de la figure 5 dans **[1]**
- Une reproduction d'une image équivalente à la figure 2 dans **[1]**
- D'autres images correspondant mieux aux types d'images attendu par l'algorithme, c'est-à-dire des images comportant peu d'intensités différentes et ayant des formes clairement définies

Exemple de visualisation de résultat

On peut observer sur la **figure 5a** un chemin de parenté partant de l'unique pixel noir et remontant vers les formes les plus externes, c'est-à-dire la cellule blanche et la fond noir entourant cette cellule. L'élément canonique de la forme, c'est-à-dire un pixel, est encadré.

Nous avons choisi de représenter le chemin de parenté avec des couleurs pour identifier le type des pixels :

- Pixels **Original** : rouge
- Pixels **New** : vert
- *Interpixels* : bleu

On remarque grâce à cette coloration que l'arbre des formes ainsi obtenu ne contient effectivement plus aucun *interpixel* ou pixel **New**. Le chemin de parenté est représenté en rouge, puisqu'il est composé uniquement de pixels d'origine.

Les étapes de canonisation et de désinterpolation donnent donc le résultat attendu.

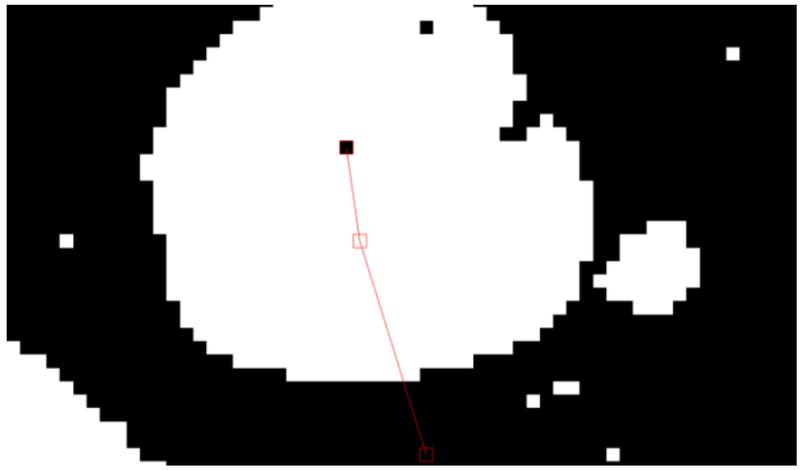

Figure 5a : Exemple de lien de parenté déduit de l'arbre des formes obtenu sur l'image *34000-cells.pgm*

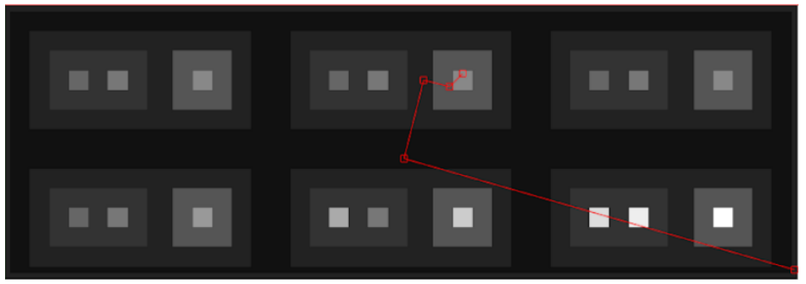

Figure 5b : Un chemin de parenté sur une de nos image test (*4860-test5.pgm*). Le pixel sélectionné est celui au centre du carré gris.

Résultats des tests

Nous avons réalisé des tests sur un jeu d'images plutôt important. A partir de ceux-ci, nous avons construit le graphique tel qu'illustré en **figure 7**. Il représente le temps d'exécution en fonction de la taille de l'image. Ces tests ont été effectués sur un ordinateur avec un processeur Intel® Core™ i7-3537U comportant 4 cœurs et avec 6Gb de RAM, en désactivant l'affichage.

Voici la commande utilisée pour réaliser ces tests sous un terminal bash :

for f in <Dossier-Images>/*; do find $f -printf «%f «>> <Fichier-Temps>; (/usr/bin/time -f%E ./tos $f) >text.txt 2>> <Fichier-Temps>; done

Cela nous a permis d'obtenir les résultats sous un format pratique, avec des données fiables et sans nécessité de les lancer un par un à la main.

Les résultats sont détaillés dans les **figures 6a**, **6b** et **7**.

Performances

Le passage aux grandes images a été difficile, étant donné que la complexité de l'algorithme obtenue dans un premier temps n'était pas linéaire. En effet l'algorithme n'était alors pas praticable pour des moyennes et grandes images. Il mettait par exemple plus de 4h pour traiter une image de taille 500*500 px.

Après un certain nombre de tests, il s'est avéré que la majorité de l'algorithme s'effectuait dans un temps raisonnable, sauf pour l'étape de canonisation. Ce résultat était logique : tous les autres algorithmes que nous avons implémentés viennent du papier [1], qui annonce un résultat quasi-linéaire. Nous avons donc, par la suite, modifié cette étape, ce qui nous a permis d'obtenir des temps d'exécution raisonnables (de quelques secondes à moins de 10 minutes) pour des moyennes et grandes images.

La librairie OpenMP nous a permis d'améliorer fortement les performances de notre algorithme en parallélisant automatiquement de nombreuses boucles. Elle a cependant posé quelques problèmes suite à une mauvaise gestion de certaines sections critiques. Faute de temps, ces boucles n'ont pas été totalement parallélisées.

Nombre de pixels	Nom de fichier	Temps d'exécution (en minutes:secondes:millisecondes)
859	test4	0:00.09
1339	test6	0:00.17
4000	synth_noise	0:00.32
4860	test5	0:00.40
19000	numbers	0:01.28
16000	tree	0:01.57
34000	cells	0:02.54
40000	bat200	0:03.04
22500	test7	0:03.28
65500	texte	0:03.93
65500	texte_bruit	0:04.07
64500	bulles	0:04.66
65500	laiton	0:04.72
65500	circuit	0:04.77
65500	bridge	0:04.81
65500	coffee	0:04.87
65500	cat	0:04.88
65500	coffee	0:04.92
65500	cat	0:04.93
65500	blobs	0:05.08
65500	baboon_grey	0:05.17
65500	zebras	0:05.43
65500	bloodCells	0:05.44
65500	pcb_gray	0:05.45
50000	sphere	0:05.89
65500	rice	0:06.12

Figure 6a : Temps d'exécution correspondant à chaque fichier

Nombre de pixels	Nom de fichier	Temps d'exécution (en minutes:secondes:millisecondes)
65500	retina2	0:06.25
79000	soil	0:06.54
88500	fissures	0:06.83
93000	keyb	0:07.10
91000	image_sombre	0:07.68
65500	particules	0:08.23
83000	peppers	0:09.26
83300	blob1	0:09.38
193000	image_sombre2	0:12.06
262000	objets	0:19.74
262000	house	0:20.02
262000	livingRoom	0:20.46
262000	baboon	0:21.46
262000	barbara	0:21.65
262000	lena_grey	0:22.28
262000	lena	0:22.28
153000	barrat4	0:23.82
306000	barrat2	0:24.22
262000	pepperAndSalt	0:27.01
300000	houses_rot	0:27.34
405000	test9	0:31.82
250000	test8	0:32.93
699300	circuit	0:47.35
845000	test10	0:54.40
624000	houses	0:55.75
922000	test11	1:45.85
2000000	AOI_GreenBand	3:42.23
2080000	test12	3:42.32
3146000	Noyau_Slice68	7:16.16

Figure 6b : Temps d'exécution correspondant à chaque fichier

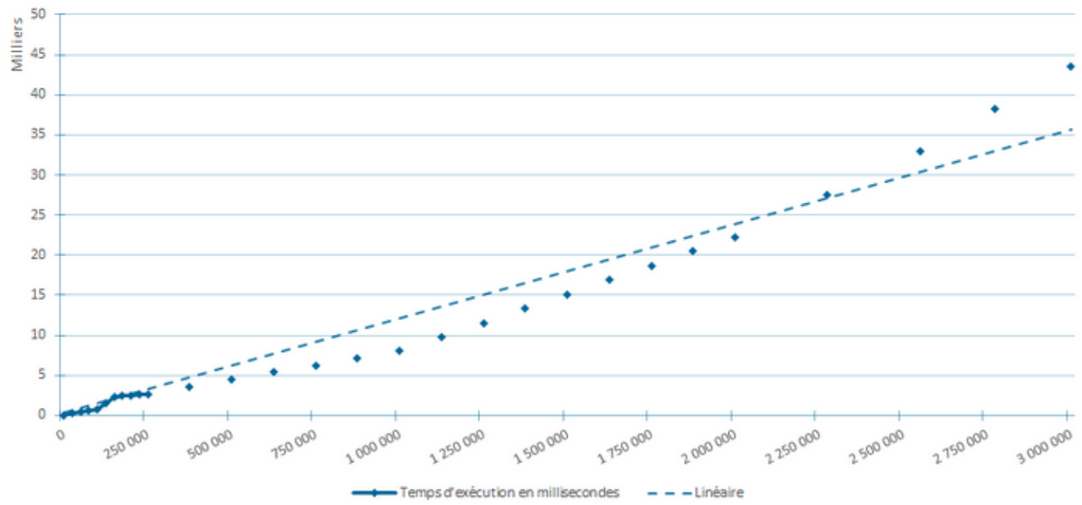

Figure 7 : Temps d'exécution en millisecondes en fonction du nombre de pixels

Répartition des tâches

Chaque semaine, nous nous sommes rencontrés afin de travailler ensemble sur le programme. Notre temps de travail est donc équilibré : nous avons réfléchi ensemble sur les différents algorithmes, nous sommes répartis différents sujets sur les mêmes plages horaires. C'était surtout une occasion pour *débugger* ensemble des situations compliquées. Nathan a néanmoins travaillé seul sur la structure du code, et sur l'affichage graphique. Nous estimons donc que son temps de travail représente 38% du projet, contre 31% pour Méline et Morgane.

Le détail de la répartition des tâches se trouve en **figure 8**.

Etape	Morgane	Nathan	Méline
Pré-traitement de l'image, structure du code, et implémentation des structures de données		X	
Pseudo-code de l'algorithme d'interpolation	X		X
Code de l'algorithme d'interpolation	X		
Pseudo-code de l'algorithme de tri		X	X
Code de l'algorithme de tri			X
Code de la file d'attente hiérarchique (pour le tri)		X	
Pseudo-code de l'algorithme de construction de l'arbre	X		X
Code de l'algorithme de construction de l'arbre	X	X	X
Debug des trois algorithmes	X	X	X
Canonisation et *debug*	X	X	X
Optimisation	X	X	
Affichage graphique de l'arbre		X	
Désinterpolation		X	
Tests de performances et statistiques	X		
Template du rapport et schémas			X
Rédaction du rapport	X	X	X

Figure 8 : répartition des tâches

Conclusion

Ce projet était complexe mais très intéressant. Il présentait de nombreux *challenges*. Nous ne connaissions pas le concept d'arbre des formes avant de nous lancer. De plus la compréhension de l'article fut difficile. En effet, certains choix de notation rendent confus la lecture de certains passages et ont provoqué plus d'une incompréhension au sein de l'équipe. Certains algorithmes sont également peu détaillés, notamment la canonisation. Malgré ces difficultés, notre algorithme présente les performances attendues par l'article [1].

Ce projet a permis de mettre une dernière fois à l'épreuve nos capacités de gestion de projet en équipe. Nous avons acquis de nombreux automatismes depuis le début de nos études et nous avons pu les appliquer ici sans difficulté et avec efficacité. Nous avons ainsi fini dans les temps, ce qui nous a laissé la possibilité d'améliorer les performances et de réaliser les tests. Nous sommes satisfaits du travail que nous avons effectué ainsi que des résultats que nous avons obtenus.

Références

[1] T. Géraud, E. Carlinet, S. Crozet et L. Najman. *A quasi-linear algorithm to compute the tree of shapes of nD images* (2013).

[2] Bibliothèque LibTIM [*http://github.com/bnaegel/libtim*]

[3] C. Berger, T. Géraud, R. Levillain, N. Widynski, A. Baillard et E. Bertin. *Effective component tree computation with application to pattern recognition in astronomical imaging* (2007).

Méline Bour-Lang
Morgane Ritter
Nathan Roth