Un algorithme de complexité linéaire pour le calcul de l'arbre des formes

Edwin Carlinet Sébastien Crozet Thierry Géraud

EPITA Research and Development Laboratory (LRDE),
14-16 rue Voltaire, F-94270 Le Kremlin-Bicêtre, France.

prénom.nom@lrde.epita.fr

Résumé

L'arbre des formes (AdF) est une représentation morphologique hiérarchique de l'image qui traduit l'inclusion des ses lignes de niveaux. Il se caractérise par son invariance à certains changement de l'image, ce qui fait de lui un outil idéal pour le développement d'applications de reconnaissance des formes. Dans cet article, nous proposons une méthode pour transformer sa construction en un calcul de Max-tree. Ce dernier a été largement étudié au cours des dernières années et des algorithmes efficaces (dont certains parallèles) existent déjà. Nous proposons également une optimisation qui permet d'accélérer son calcul dans le cas classique des images 2D. Il en découle un algorithme simple, efficace, s'exécutant linéairement en fonction du nombre de pixels, avec une faible empreinte mémoire, et qui surpasse les algorithmes à l'état de l'art.

Mots Clef

Arbres des formes, algorithmes, morphologie mathématique.

Abstract

The Tree of Shapes is a morphological tree-based representation of an image describing the inclusion of its level lines. It features many invariances to image changes, which makes it well-suited for a lot of applications in image processing and pattern recognition. In this paper, we propose a way of turning this algorithm into a Max-Tree computation. The latter has been widely studied, and many efficient algorithms (including parallel ones) have been developed. Furthermore, we develop a specific optimization to speed-up the common 2D case. It follows a simple *and* efficient *algorithm, running in* linear *time with a low memory footprint, that outperforms other current algorithms. For Reproducible Research purpose, we distribute our code as free software.*

Keywords

Tree of shapes, algorithms, mathematical morphology.

1 Introduction

L'arbre des formes (AdF) est une structure hiérarchique traduisant l'inclusion des lignes de niveaux de l'image. Ces dernières sont les contours de formes (voir Figure 1). C'est une structure intéressante car les lignes de niveaux sont invariantes aux inversions et aux changements de contraste. C'est particulièrement intéressant en traitement d'images, en reconnaissance des formes et en vision par ordinateur [12], où l'AdF a montré son efficacité à traiter des images peu contrastées, avec des changements d'illuminations et des changements de prises de vues [27, 6] (voir Figure 3). La force de cette représentation réside dans sa versatilité. Comme le Min- et Max-tree, il permet d'appliquer des filtres connexes avancés [25, 32, 29, 36] (Figures 2a et 2d) qui peuvent s'adapter facilement à des applications spécifiques. Les profils d'attributs morphologiques sur l'AdF ont été utilisés en télédétection pour la classification d'images hyperspectrales [14, 15, 16, 24]. Dans [33], l'AdF est utilisé pour détecter et supprimer par élagage le bruit dans les images ultrasons. En ce qui concerne les images naturelles, l'AdF sert de descripteur des formes pour sélectionner les régions stables de l'image [2], et pour l'analyse de texture [34], en évitant de devoir gérer deux arbres de composantes comme c'est le cas dans l'algorithme originel des MSER. Dans [18], il est utilisé pour la sélection d'objets (Figure 2c) et l'*alpha matting*, en permettant de calculer des plus courts chemins exempts de tout problème topologique. Avec des stratégies avancées de sélection ou d'élimination de nœuds, certains auteurs [7, 37, 38] ont développé des filtrages par minimisation énergétique pour la simplification et la segmentation de

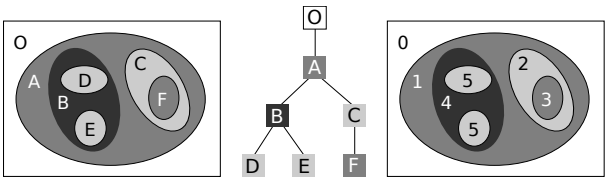
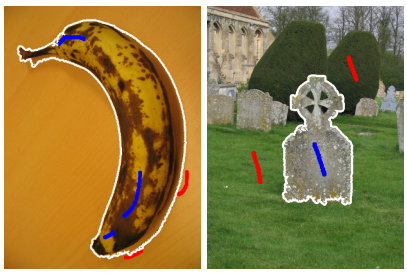


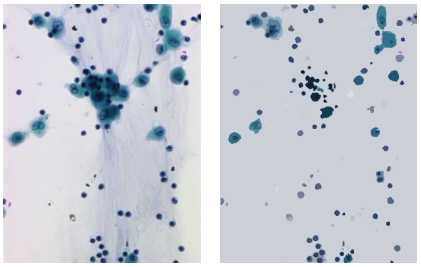
FIGURE 1 – Image originale (à gauche), son AdF (au milieu), et la carte d'ordre d'inclusion (à droite) calculée par notre algorithme.

(a) Filtre de grain

(b) Simplification



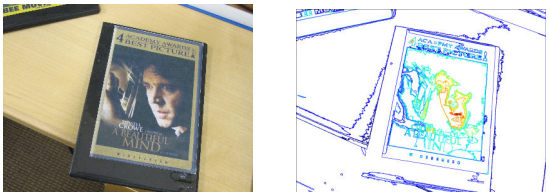
(c) Détourage



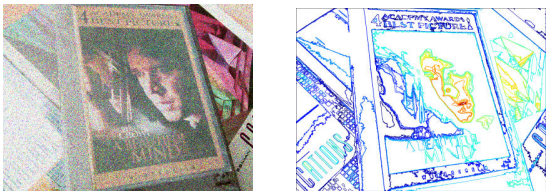
(d) Filtrage de formes

FIGURE 2 – Quelques applications utilisant l'arbre des formes (AdF). (a) Les filtres de grain sont utilisés pour l'extraction de canevas en supprimant le texte clair ou foncé [26]. (b) Sélection de nœuds par minimisation énergétique pour la simplification ou segmentation d'images [35, 37, 38]. (c) Segmentation interactive avec peu de marqueurs sur l'AdF [9]. (d) Filtrage par attribut dans l'espace des formes pour la cytologie avec critère de circularité [36].

scènes (voir Figure 2b). L'AdF n'est pas seulement limité aux images en niveaux de gris puisqu'il a été étendu aux couleurs et aux images multivariées [10]. De plus, des considérations de topologie discrète liées à l'AdF sont discutées dans [3, 21], et son lien avec une mesure de saillance



(a) Une image (à gauche) et ses lignes de niveaux significatives (à droite).



(b) Une vue différente de la pochette de film de (a). Les lignes de niveaux de cette nouvelle image correspondent globalement à celles de (a), ce qui signifie que l'AdF encode correctement le contenu des deux images.

FIGURE 3 – Robustesse des lignes de niveaux par rapport à des transformations de l'espace et des valeurs. Les couleurs des lignes correspondent à leur niveau d'inclusion dans l'AdF.

très populaire est décrite dans [22].

Néanmoins, l'AdF reste sous-exploité en dehors de la communauté de Morphologie Mathématique. Comme son potentiel n'est plus à démontrer, c'est sûrement dû au manque d'un algorithme *rapide* et *facile à écrire* (ainsi qu'à une implémentation publique). Aujourd'hui, il existe quatre algorithmes d'AdF. La première approche, la Fast Level Line Transform [27] (FLLT), consiste à calculer et fusionner les min- et max-trees. La seconde, la Fast Level Set Transform [11] (FLST), extrait chaque branche de l'arbre en partant d'une feuille (un extremum regional sans trous) et la fait croître vers la racine jusqu'à rencontrer une bifurcation. Ce processus est réitéré jusqu'à ce que toutes les branches soient extraites. La troisième méthode de Song [31] est une approche *top-down* qui suit les récursivement lignes de niveaux depuis la bordure. L'algorithme est cependant restreint aux images à grilles hexagonales, ou peut être appliqué sur une grille rectangulaire avec une 6-connexité.

Dans cet article, nous proposons un nouvel algorithme basé sur la dernière approche de [20] qui est simple et efficace. Il est simple parce qu'il transforme le calcul d'AdF en un calcul de max-tree pour lequel il existe de nombreux algorithmes. Cette transformation est expliquée Section 2. Aussi, cet algorithme est efficace parce que les algorithmes de max-tree le sont, et que toute la complexité algorithmique leur est déléguée. De plus, dans la Section 3, nous proposons une optimisation 2D qui améliore le temps de calcul de notre méthode et sera comparée avec les autres algorithmes dans la Section 4.

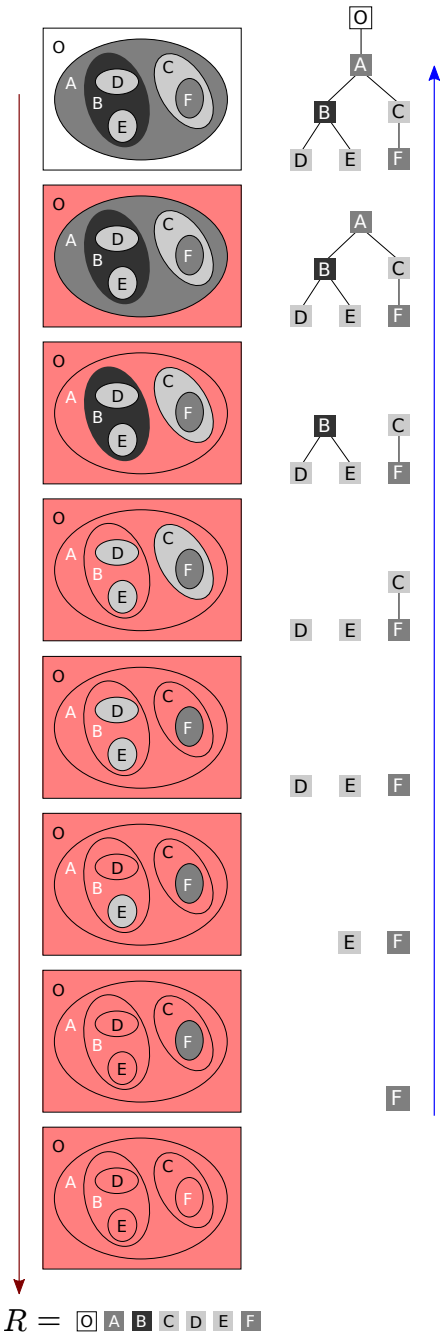


FIGURE 4 – Déroulement étape par étape de l'algorithme quasi-linéaire [20].

```
1: function PriorityPush(q, p, F, λ)
2:     [a, b] ← F(p)
3:     if a > λ then λ' ← a
4:     else if b < λ then λ' ← b
5:     else λ' ← λ
6:     Insert(q[λ'], p)

7: function PriorityPop(q, λ)
8:     if q[λ] is empty then
9:         λ' ← level next to λ such as q[λ'] is not empty
10:        λ ← λ'
11:    return (λ, Pop(q[λ]))

12: function Sort(F)
13:    for all p do deja_vu(p) ← false
14:    Push(q[f(p∞)], p∞)
15:    deja_vu(p∞) ← true
16:    λ ← f(p∞)
17:    while q is not empty do
18:        (λ, p) ← PriorityPop(q, λ)
19:        f♭(p) ← λ
20:        Append(R, p)
21:        for all n ∈ N(p) such as not deja_vu(n) do
22:            PriorityPush(q, n, F, λ)
23:            deja_vu(n) ← true
24:    return (R, f♭)

25: function FindRoot(zpar, x)
26:    if zpar(x) = x then return x
27:    else zpar(x) ← FindRoot(zpar, zpar(x))
28:    return zpar(x)

29: function UnionFind(R)
30:    for all p do zpar(p) ← undef
31:    for i ← N − 1 to 0 do
32:        p ← R[i]
33:        parent(p) ← p
34:        zpar(p) ← p
35:        for all n ∈ N(p) such as zpar(n) ≠ undef do
36:            r ← FindRoot(zpar, n)
37:            if r ≠ p then
38:                parent(r) ← p
39:                zpar(r) ← p
40:    return parent

41: function ComputeToS(f)
42:    F ← InterpolateAndImmerse(f)
43:    (R, f♭) ← Sort(F)
44:    parent ← UnionFind(R)
45:    Canonicalize(f♭, R, parent)
46:    return Emerse(R, parent)
```

FIGURE 5 – Algorithme quasi-linéaire [20].

2 Un algorithme linéaire de calcul d'arbre des formes

2.1 Définition de l'arbre des formes

Considérons une image numérique nD, notée f, comme une fonction définie sur une grille régulière cubique (précisément, $f : \mathbb{Z}^n \to \mathbb{Z}$). Pour gérer correctement les sous-ensembles de $\mathbb{Z}^n$ et leur complémentaire, nous considérerons les couples de connexités c_{2n} et c_{3^n-1} (en 2D c_4 et c_8, pour respectivement la 4-connexité et la 8-connexité). Pour $\lambda \in \mathbb{Z}$, les ensembles de niveaux inférieurs et supérieurs de f sont définis respectivement

par :

$$[\,f < \lambda\,] \;=\; \{\,p \mid f(p) < \lambda\,\}$$
$$\text{et } [\,f \geq \lambda\,] \;=\; \{\,p \mid f(p) \geq \lambda\,\}.$$

Nous pouvons en déduire deux nouveaux ensembles, $\mathcal{T}_{\min}(f)$ et $\mathcal{T}_{\max}(f)$, composés des composantes connexes des ensembles de niveaux respectivement inférieurs et supérieurs de f :

$$\mathcal{T}_{\min}(f) \;=\; \{\,\Gamma \in \mathcal{CC}_{c_{2n}}([\,f < \lambda\,])\,\}_\lambda$$
$$\text{et } \mathcal{T}_{\max}(f) \;=\; \{\,\Gamma \in \mathcal{CC}_{c_{3^n-1}}([\,f \geq \lambda\,])\,\}_\lambda,$$

où $\mathcal{CC}$ est l'opérateur qui donne un ensemble de composantes connexes. Les éléments de $\mathcal{T}_{\min}(f)$ et $\mathcal{T}_{\max}(f)$, dotés de la relation d'inclusion, donne respectivement deux arbres duaux : le min-tree et le max-tree de f. À l'aide de l'opérateur Sat de bouchage de cavités (en 2D, bouchage de trous), nous pouvons alors définir un nouvel ensemble de composantes :

$$\begin{aligned}\mathfrak{S}(f) \;=& \{\,\mathrm{Sat}_{c_{3^n-1}}(\Gamma);\ \Gamma \in \mathcal{T}_{\min}(f)\,\} \\ &\cup\ \{\,\mathrm{Sat}_{c_{2n}}(\Gamma);\ \Gamma \in \mathcal{T}_{\max}(f)\,\}.\end{aligned}$$

Grâce à la relation d'inclusion, les composantes de ce nouvel ensemble s'arrangent en un arbre, appelé *arbre des formes* de f [27]. En effet, pour toute paire de composantes, Γ et Γ' de $\mathfrak{S}(f)$, on a : $\Gamma \subset \Gamma'$ ou $\Gamma' \subset \Gamma$ ou $\Gamma \cap \Gamma' = \emptyset$. Une composante de $\mathfrak{S}(f)$ s'appelle une *forme* de f et, grâce à l'application de l'opérateur de bouchage, les formes de f n'ont pas de cavités. De plus, l'arbre d'inclusion des formes est également l'arbre d'inclusion des lignes de niveaux (les contours des formes). Une illustration a été donnée en Figure 1.

2.2 L'algorithme quasi-linéaire

Géraud et al. [20] ont décrit un algorithme de calcul quasi-linéaire d'AdF basé sur l'Union-Find. Cet algorithme est rappelé en Figure 5 (les numéros de lignes dans les paragraphes qui suivent y font référence). L'algorithme comprend quatre étapes principales :

Interpolation. L'image est agrandie d'un facteur 2 en utilisant une interpolation min, max ou median pour obtenir une image bien-composée [5]. Cette étape assure l'unicité de l'AdF et permet de choisir une certaine paire de connexités objet/fond. Par exemple, l'interpolation *max* traduit la semi-continuité supérieure de l'image, i.e., les coupes de niveaux inférieurs sont 4-connexes et les coupes supérieures sont 8-connexes. Le médian est utilisé pour obtenir une *vraie* auto-dualité dans le cas 2D [21].

Immersion. L'image est transformée en une carte d'intervalles F sur la grille de Khalimsky. Les pixels intermédiaires (0- et 1-faces) sont utilisés pour représenter toutes les lignes de niveaux qui passent entre deux pixels originaux (2-faces). L'interpolation et l'immersion, appelées en ligne 42, sont illustrées en Figure 6.

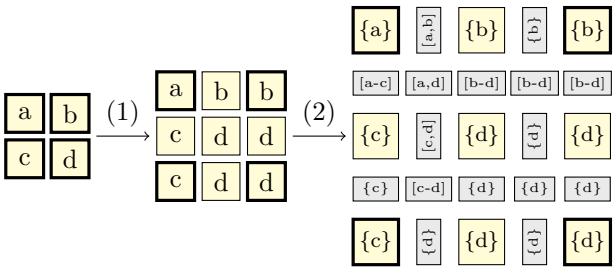


FIGURE 6 – Interpolation (1) et immersion (2) d'une image. On suppose que $a \leq b \leq c \leq d$ pour l'exemple. Les pixels originaux ont une bordure épaisse, les 2-faces sont en jaune, et les pixels intermédiaires (0/1-faces) sont en gris.

Tri des pixels. Une propagation continue est effectuée sur F commençant de la bordure (mais celle-ci peut commencer de n'importe quel point pour choisir la racine de l'arbre) comme montré par la flèche rouge en Figure 4. L'ordre de traitement des pixels est sauvegardé dans un vecteur R, ainsi que le niveau auquel ils ont été visités (sauvegardés dans l'image $f_\flat$). Le tri (lignes 12 à 24) s'appuie sur une file de priorité (tableau q) et sur les opérations d'insertion et d'extraction par priorité (respectivement PRIORITYPUSH ligne 1 et PRIORITYPOP ligne 7).

Construction de l'arbre Les pixels sont traités dans le sens inverse de R de façon à ce que l'algorithme d'Union-Find puisse maintenir les composantes connexes disjointes et construire l'arbre de bas en haut (flèche bleue en Figure 4). Il s'agit l'algorithme classique de construction d'arbre de composantes à partir de l'Union-Find (lignes 29 à 40) [19, 1].

Deux étapes supplémentaires sont la canonisation de l'arbre (appelée ligne 45) et l'émersion (appelée ligne 46), qui est l'opération inverse de "interpolation et immersion".

2.3 Transformation en carte d'ordre

Crozet et Géraud [17] et Carlinet et Géraud [10] ont noté une propriété flagrante mais intéressante de l'arbre des formes : il est possible de retrouver l'arbre de composantes d'une image dont les pixels ont été valués par leur profondeur dans l'arbre. En effet, il suffit alors de calculer le max-tree de l'image de profondeur. En fait, non seulement la *profondeur* mais n'importe quelle valuation *d'ordre topologique* de l'arbre est possible.

Cette propriété est le fondement de notre approche puisque de nombreux algorithmes efficaces de calcul de Max-tree existent (cf. [8] et le plus récent étant [23]). On doit donc juste s'assurer que les étapes préliminaires sont suffisamment efficaces. En particulier, l'étape de *tri des pixels* est maintenant en charge de produire une carte Ord (à la place de R) correspondant à un tri topologique de l'AdF. Cet algorithme est donné en Figure 7 et la carte d'ordre est illustré en Figure 1 (droite).

L'algorithme traite les pixels niveau par niveau avec un

function COMPUTEORDERMAP(f, F, p_∞)
  $\mathrm{Q} \leftarrow \emptyset$; $\mathrm{R} \leftarrow \emptyset$
  $\mathrm{Ord}(x) \leftarrow -1$ forall x
  $\lambda^{\mathrm{old}} \leftarrow f(p_\infty)$
  INSERT(Q, $(\lambda^{\mathrm{old}}, p_\infty)$)
  $d \leftarrow 0$
  while $\mathrm{Q} \neq \emptyset$ **do**
    $(\lambda, p) \leftarrow$ POP(Q, λ^{old})
    if $\lambda^{\mathrm{old}} \neq \lambda$ **then** $d \leftarrow d + 1$
    $\mathrm{Ord}(p) \leftarrow d$
    APPEND(R, p)
    for all $n \in \mathcal{N}_4(p)$ such that $\mathrm{Ord}(n) = -1$ **do**
      $[a, b] \leftarrow F(n)$
      if $\lambda < a$ **then** INSERT(Q, (a, n))
      else if $\lambda > b$ **then** INSERT(Q, (b, n))
      else INSERT(Q, (λ, n))
    $\lambda^{\mathrm{old}} \leftarrow \lambda$
  return (R, Ord)

FIGURE 7 – Procédure de tri.

front de propagation. Ce dernier conserve les pixels triés dans le tableau associatif Q afin de permettre leur accès depuis le niveau courant du front. À chaque fois que le niveau courant change, le nombre de niveaux traités d est incrémenté et lorsqu'un pixel p est visité, on stocke son ordre de traitement dans $\mathrm{Ord}(p)$ et on l'insère dans le tableau des pixels visités R.

En Figure 7, on s'appuie sur le type de données abstrait suivant :

— INSERT(Q, (λ, p)) :
insère le pixel p au niveau (clef) λ.
— POP(Q, k) :
récupère et supprime la plus petite paire (λ, p) telle que $k \leq \lambda$ ou la plus grande paire (λ, p) telle que $\lambda < k$.

Ces fonctions sont typiquement implémentées à l'aide de files hiérarchiques pour les images faiblement quantifiées et d'arbres rouge-noir pour les images fortement quantifiées.

2.4 L'AdF transformé en Max-tree

Une fois la carte d'ordre construite, on a juste à calculer son Max-tree pour obtenir son AdF. Dans la plupart des cas, la profondeur maximale est telle que l'on peut appliquer un algorithme linéaire de Max-tree (comme celui de [30]). Dans le cas contraire, on utilise un algorithme de Max-tree quasi-linéaire comme celui de [28] (en évitant l'étape de tri puisque R a déjà été calculé). Le lecteur peut se référer à [8] pour une comparaison des différents algorithmes disponibles.

3 Optimisation 2D

L'*interpolation* suivie de l'*immersion* impliquent un temps de calcul et une empreinte mémoire prohibitifs puisque le nombre d'éléments est alors multiplié par 16. Heureusement, cette simulation de la paire de 4/8-connexités peut être optimisée en évitant l'étape d'*interpolation*. Une carte de connexion remplace alors les inter-pixels. Pour obtenir le même comportement que si l'algorithme était appliqué à la représentation originale, on doit s'assurer que la représentation optimisée possède les mêmes ensembles de niveaux. C'est pourquoi, nous utilisons une *immersion* spécifique qui dépend de la configuration locale des pixels.

Pour les 1- ou 2-faces, les règles restent les mêmes. Elles diffèrent pour les 0-faces p si les quatre pixels voisins a, b, c, d forment un point selle. Sans perte de généralité, on suppose $a \leq b < c \leq d$; alors, $F(p) = [c, d]$. On simule ainsi la semi-continuité supérieure en empêchant les lignes de niveaux inférieurs à c de passer à travers le point selle avec la 4-connexité.

Utiliser la 4-connexité avec cette nouvelle interpolation est cependant trop restrictif puisque les 2-faces valuées par c et d d'un point selle peuvent ne pas connecter correctement si elles sont atteintes pendant la propagation de niveaux de gris croissants. Ainsi, en plus de la 4-connexité, une carte de connexion est utilisée pour ajouter les connexions diagonales manquantes en fonction de la configuration locale des 2-faces comme montré en Figure 9. Le rôle de cette carte est d'agir comme si on utilisait les inter-pixels interpolés de la représentation initiale. Elle a donc été établie pour que les ensembles de niveaux supérieurs et inférieurs soient les mêmes dans les deux cas. L'équivalence des deux représentations est montrée en Figure 8. Cette carte définit ensuite le graphe de connexion utilisé pendant le calcul de la *carte d'ordre* et du *max-tree*.

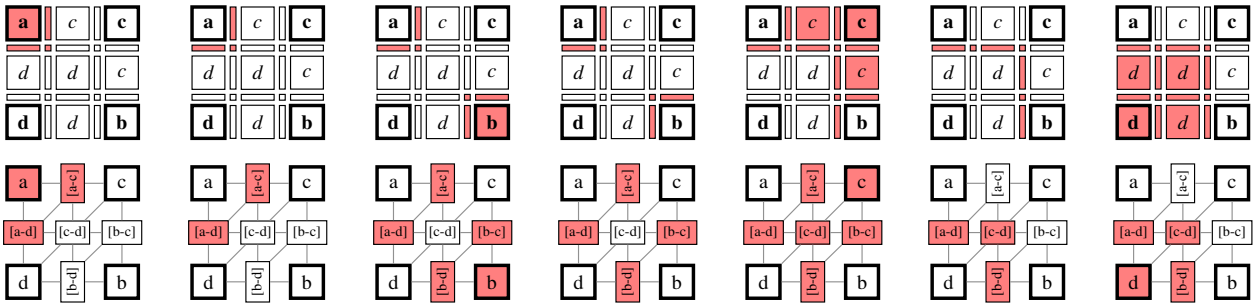
4 Analyse des performances

4.1 Complexité et empreinte mémoire

Soit n le nombre de pixels *après* immersion, i.e., si k est le nombre de pixels de l'image d'entrée, $n = 4k$.

Pour les images faiblement quantifiées, le calcul de la *carte d'ordre* est linéaire en utilisant les files hiérarchiques. La complexité du calcul du max-tree est dépendante de la valeur maximale de la carte de profondeur. Le pire cas est atteint pour une image de carrés imbriqués, auquel cas la profondeur est de l'ordre de $\sqrt{(n)}$. Pour les images naturelles, la profondeur maximale est telle que l'on peut appliquer un algorithme linéaire de Max-tree à base de file hiérarchique. Dans le cas contraire, on peut se rabattre sur un algorithme à base d'*Union-Find* qui nous assure ainsi une compléxité quasi-linéaire au pire. En conséquence, le processus complet est linéaire en pratique (et quasi-linéaire au pire).

Pour les image fortement quantifiées, le calcul de la *carte d'ordre* est en $n \log n$, puisque trier les pixels dans le front



(a) $\lambda \leq a$ (b) $a < \lambda < b$ (c) $\lambda = b$ (d) $b < \lambda < c$ (e) $\lambda = c$ (f) $c < \lambda < d$ (g) $d \leq \lambda$

FIGURE 8 – Équivalence entre l'*immersion* originale (ligne du haut) et l'*immersion* optimisée (ligne du bas) dans le cas d'une configuration en point selle. Pour chaque niveau λ, l'ensemble de niveaux sélectionné et son complément contiennent les mêmes composantes connexes (en terme de face d'origine).

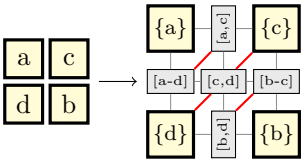


FIGURE 9 – Immersion pour la version optimisée dans le cas d'une configuration de point-selle.

utilise généralement un arbre rouge-noir. Le calcul de Max-tree reste inchangé et le processus complet est $n \log n$.

En comparaison à [20], notre calcul de la *carte d'ordre* et leur *étape de tri* sont similaires et partagent la même complexité. Notre approche se différencie ensuite par un calcul de max-tree plus efficace que leur étape d'*union-find*.

Concernant l'utilisation mémoire, soit I la taille d'un `int` en octet. Le calcul de la *carte d'ordre* nécessite $4nI$ ou $5nI$ octets pour $\mathrm{Q}, \mathrm{R}, \mathrm{Ord}$ (en fonction de l'implémentation) et la carte de connexion. La mémoire utilisée pour le Max-tree dépend de l'algorithme choisi (voir [8]) et requiert entre $2nI$ et $3nI$ octets.

4.2 Comparaison avec les algorithmes existants

On compare notre algorithme à l'algorithme de la FLST de Megawave, à Song [31] (implémentation fournie par les auteurs), et à Géraud et al. [20]. On teste chaque algorithme sur une base de 8 images naturelles (de 20-MPix) dont la taille varie de 1 à 16 Mpix. Le minimum de 5 répétitions a été retenu. Les expériences ont été menées sur un Intel Core i7 7500U, 2.7Ghz et 8Gb de RAM. La Figure 10 montre la vitesse moyenne et l'écart type sur le jeu de données.

Une première observation est que la méthode de Song [31] n'a pas pu exécutée sur les images supérieures à 4MPix à cause d'une trop large consommation mémoire. Ensuite, tous les algorithmes ont en pratique un comportement *quasi* linéaire par rapport à la taille même si certains ont une complexité quadratique au pire. Néanmoins,

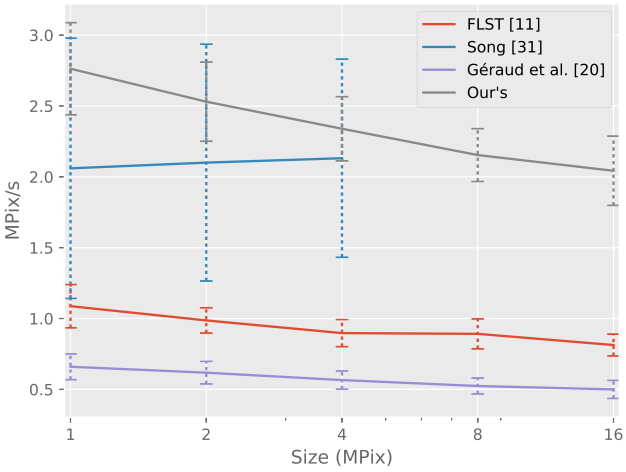


FIGURE 10 – Comparaison de la vitesse de calcul des algorithmes par rapport à la taille de l'image (la courbe la plus haute étant la meilleure).

la vitesse de la FLST, de Géraud et al. [20] et de notre algorithme dépendent de la taille de l'image et décroit de 5 à 10% lorsque l'image double de taille. Ce résultat pourrait sembler contradictoire avec le complexité théorique qui est linéaire. Néanmoins, ce comportement est attendu sur ce type d'algorithme qui effectue un traitement non-séquentiel des pixels de l'image. L'accès aux données de l'image de façon aléatoire entraine un coût qui s'explique par les contraintes matérielles, notament en matière de cache mémoire.

Finalement, notre algorithme est 4 fois plus rapide que Géraud et al. [20] et 2.5 fois plus rapide que la FLST. Il est aussi plus stable que Song [31] qui montre des temps de traitement très variables en fonction du contenu de l'image. En moyenne, notre algorithme est aussi plus rapide que Song [31] mais cette différence tend à s'amoindrir lorsque la taille de l'image augmente. Dans ces résultats, l'opti-

misation 2D joue un rôle important puisqu'elle permet de traiter 4 fois moins de pixels et améliore ainsi d'un facteur 2 le temps de calcul.

5 Implémentation

À des fins de *recherche reproductible*, le code source de l'algorithme décrit dans cet article est disponible à l'adresse :

`http://publications.lrde.epita.fr/carlinet.18.rfiap`

6 Conclusion et perspective

Nous avons introduit un nouvel algorithme de calcul de l'AdF avec une complexité moyenne linéaire. L'idée principale a été de transformer le calcul de l'AdF en un calcul de Max-tree pour tirer bénéfice de l'efficacité des algorithmes d'arbres de composantes existants. Aussi, une optimisation spécifique au cas 2D a été proposée pour réduire l'occupation mémoire et augmenter la vitesse de calcul en évitant le recours à une interpolation inter-pixels.

Notre algorithme surpasse en temps de calcul les autres méthodes de l'état de l'art. Aussi, la parallélisation de notre approche n'a pas été évaluée ici par équité avec les autres algorithmes qui sont séquentiels. Néanmoins, la stratégie de parallélisation décrite dans [17] peut être appliquée et des algorithmes de Max-tree parallèles existent pour les images faiblement et fortement quantifiées [8].

Notre approche est généralisable en nD (voir [20] et [4]) mais l'optimisation décrite pour le cas 2D ne tient plus. Ainsi, on a besoin de multiplier la taille de l'image par 4^n, ce qui devient impraticable pour les grosses images. Dans ce cas, la seule alternative reste la FLLT [13].

Par conséquent, dans nos travaux futurs, nous essayerons d'étendre notre optimisation mémoire 2D aux grilles de dimensions supérieures.

Références

[1] C. Berger, T. Géraud, R. Levillain, N. Widynski, A. Baillard, and E. Bertin. Effective component tree computation with application to pattern recognition in astronomical imaging. In *Proc. of IEEE ICIP*, volume 4, pages 41–44, San Antonio, TX, USA, 2007.

[2] P. Bosilj, E. Kijak, and S. Lefèvre. Beyond MSER : Maximally stable regions using tree of shapes. In *Proc. of the British Machine Vision Conf. (BMVC)*, pages 169.1–169.13, 2015.

[3] N. Boutry, T. Géraud, and L. Najman. On making nD images well-composed by a self-dual local interpolation. In *Proc. of the Intl. Conf. on Discrete Geometry for Computer Imagery (DGCI)*, volume 8668 of *LNCS*, pages 320–331. Springer, 2014.

[4] N. Boutry, T. Géraud, and L. Najman. How to make nD functions well-composed in a self-dual way. In *Proc. of ISMM*, volume 9082 of *LNCS*, pages 561–572. Springer, 2015.

[5] N. Boutry, T. Géraud, and L. Najman. A tutorial on well-composedness. *Journal of Mathematical Imaging and Vision*, 60(3) :443–478, 2018.

[6] F. Cao, J.-L. Lisani, J.-M. Morel, P. Musé, and F. Sur. *A Theory of Shape Identification*, volume 1948 of *LNM*. Springer, 2008.

[7] J. Cardelino, G. Randall, M. Bertalmio, and V. Caselles. Region based segmentation using the tree of shapes. In *Proc. of IEEE ICIP*, pages 2421–2424, 2006.

[8] E. Carlinet and T. Géraud. A comparative review of component tree computation algorithms. *IEEE Trans. on Image Processing*, 23(9) :3885–3895, 2014.

[9] E. Carlinet and T. Géraud. Morphological object picking based on the color tree of shapes. In *Proc. of IPTA*, pages 125–130, 2015.

[10] E. Carlinet and T. Géraud. MToS : A tree of shapes for multivariate images. *IEEE Trans. on Image Processing*, 24(12) :5330–5342, 2015.

[11] V. Caselles and P. Monasse. *Geometric Description of Images as Topographic Maps*, volume 1984 of *LNM*. Springer, 2009.

[12] V. Caselles, B. Coll, and J.-M. Morel. Topographic maps and local contrast changes in natural images. *International Journal of Computer Vision*, 33(1) :5–27, 1999.

[13] V. Caselles, E. Meinhardt-Llopis, and P. Monasse. Constructing the tree of shapes of an image by fusion of the trees of connected components of upper and lower level sets. *Positivity*, 12(1) :55–73, 2008.

[14] G. Cavallaro, M. Dalla Mura, J. A. Benediktsson, and A. Plaza. Remote sensing image classification using attribute filters defined over the tree of shapes. *IEEE Trans. on Geoscience and Remote Sensing*, 54(7) : 3899–3911, 2016.

[15] G. Cavallaro, M. Dalla Mura, E. Carlinet, T. Géraud, N. Falco, and J. Benediktsson. Region-based classification of remote sensing images with the morphological tree of shapes. In *Proc. of IEEE IGARSS*, pages 5087–5090, 2016.

[16] G. Cavallaro, N. Falco, M. Dalla Mura, and J. A. Benediktsson. Automatic attribute profiles. *IEEE Trans. on Image Processing*, 26(4) :1859–1872, 2017.

[17] S. Crozet and T. Géraud. A first parallel algorithm to compute the morphological tree of shapes of nD images. In *Proc. of IEEE ICIP*, pages 2933–2937, 2014.

[18] A. Dubrovina, R. Hershkovitz, and R. Kimmel. Image editing using level set trees. In *Proc. of IEEE ICIP*, pages 4442–4446, 2014.

[19] T. Géraud. Ruminations on Tarjan's Union-Find algorithm and connected operators. In *Proc. of ISMM*, volume 30 of *Computational Imaging and Vision*, pages 105–116, Paris, France, 2005. Springer.

[20] T. Géraud, E. Carlinet, S. Crozet, and L. Najman. A quasi-linear algorithm to compute the tree of shapes of nD images. In *Proc. of ISMM*, volume 7883 of *LNCS*, pages 98–110. Springer, 2013.

[21] T. Géraud, E. Carlinet, and S. Crozet. Self-duality and discrete topology : Links between the morphological tree of shapes and well-composed gray-level images. In *Proc. of ISMM*, volume 9082 of *LNCS*, pages 573–584. Springer, 2015.

[22] T. Géraud, Y. Xu, E. Carlinet, and N. Boutry. Introducing the Dahu pseudo-distance. In *Proc. of ISMM*, volume 10225 of *LNCS*, pages 55–67. Springer, 2017.

[23] M. Götz, G. Cavallaro, T. Géraud, M. Book, and M. Riedel. Parallel computation of component trees on distributed memory machines. *IEEE Trans. on Parallel and Distributed Systems*, 2018. To appear.

[24] L. Gueguen and R. Hamid. Toward a generalizable image representation for large-scale change detection : Application to generic damage analysis. *IEEE Trans. on Geoscience and Remote Sensing*, 54(6) : 3378–3387, 2016.

[25] R. Jones. Connected filtering and segmentation using component trees. *Computer Vision and Image Understanding*, 75(3) :215–228, 1999.

[26] G. Lazzara, T. Géraud, and R. Levillain. Planting, growing and pruning trees : Connected filters applied to document image analysis. In *Proc. of IAPR DAS*, pages 36–40, 2014.

[27] P. Monasse and F. Guichard. Fast computation of a contrast-invariant image representation. *IEEE Trans. on Image Processing*, 9(5) :860–872, 2000.

[28] L. Najman and M. Couprie. Building the component tree in quasi-linear time. *IEEE Trans. on Image Processing*, 15(11) :3531–3539, 2006.

[29] G. Ouzounis and M. Wilkinson. Mask-based second-generation connectivity and attribute filters. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 29(6) :990–1004, 2007.

[30] P. Salembier, A. Oliveras, and L. Garrido. Antiextensive connected operators for image and sequence processing. *IEEE Trans. on Image Processing*, 7(4) : 555–570, 1998.

[31] Y. Song. A topdown algorithm for computation of level line trees. *IEEE Trans. on Image Processing*, 16 (8) :2107–2116, 2007.

[32] E. Urbach and M. Wilkinson. Shape-only granulometries and grey-scale shape filters. In *Proc. of the Intl. Symp. on Mathematical Morphology (ISMM)*, pages 305–314. CSIRO, 2002.

[33] N. Widynski, T. Géraud, and D. Garcia. Speckle spot detection in ultrasound images : Application to speckle reduction and speckle tracking. In *Proc. of IEEE IUS*, pages 1734–1737, 2014.

[34] G. Xia, J. Delon, and Y. Gousseau. Shape-based invariant texture indexing. *International Journal of Computer Vision*, 88(3) :382–403, 2010.

[35] Y. Xu, T. Géraud, and L. Najman. Salient level lines selection using the Mumford-Shah functional. In *Proc. of IEEE Intl. Conf. on Image Processing (ICIP)*, pages 1227–1231, 2013.

[36] Y. Xu, T. Géraud, and L. Najman. Connected filtering on tree-based shape-spaces. *IEEE Trans. on Pattern Analysis and Machine Intel.*, 38(6) :1126–1140, 2016.

[37] Y. Xu, T. Géraud, and L. Najman. Hierarchical image simplification and segmentation based on Mumford-Shah-salient level line selection. *Pattern Recognition Letters*, 83(3) :278–286, 2016.

[38] Y. Xu, E. Carlinet, T. Géraud, and L. Najman. Hierarchical segmentation using tree-based shape spaces. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 39(3) :457–469, 2017.