Thèse de Doctorat
Sorbonne Université
École Doctorale EDITE (ED130)

A Generic, Efficient, and Interactive Approach to Image Processing with Applications in Mathematical Morphology

Baptiste Esteban

Une approche générique, performante et interactive du traitement d'images avec des applications en morphologie mathématique

Soutenue le Jeudi 21 Décembre 2023

Jury:

Président	**Pr. Olivier LEZORAY**	GREYC/Université de Caen
Rapporteur	**Pr. Pascal MONASSE**	LIGM/ENPC/Université Gustave Eiffel
Rapporteur	**Pr. Benjamin PERRET**	LIGM/ESIEE/Université Gustave Eiffel
Examinateur	**Dr. Frédéric PESCHANSKI**	LIP6/Sorbonne Université
Examinateur	**Dr. Deise SANTANA MAIA**	CRIStAL/Université de Lille
Directeur	**Pr. Didier VERNA**	LRE/EPITA
Encadrant	**Dr. Edwin CARLINET**	LRE/EPITA
Encadrant	**Dr. Guilaume TOCHON**	LRE/EPITA

Acknowledgments

The last five years at LRDE (now LRE) have been a rich experience, both humanly and professionally. All the people I have met brought me wonderful advices to succeed in my PhD.

How could I not start by thanking you, Guillaume. You made me dive into the wonderful field of image processing, and more particularly hierarchical representations of images, when I was looking for an internship at the end of my third year at EPITA, and you accompanied me from the beginning of this internship until the end of my PhD. You encouraged me to delve into this adventure which is the preparation of a thesis and you suggested adding the software aspect of my thesis in collaboration with Edwin as a second supervisor. Thus, my second thank goes to you, Edwin. You greatly contributed improving my knowledge about mathematical morphology and you mentored me on image processing software development. Thanks to you, my C++ skills significantly improved, which greatly helped me throughout this journey. Finally, for my dear director, Didier, I want to thank you for the complementarity you brought to this wonderful trio of supervisors. You made me have nightmares about red pen traces on manuscripts for a long time but that brought me rigor when writing and helped me improve my English skills. To the three of you, thank you very much for always being there when I needed you.

I would like to thank Benjamin Perret and Pascal Monasse, who accepted to review this thesis, Olivier Lézoray who agreed to be the president of the jury, and Frédéric Peschanski and Deise Santana Maia to be part of this jury. Special thanks go to Benjamin and Frédéric who gave me valuable advices during the committees to succeed in this thesis.

The achievement of this thesis would not have been possible without the people I met and worked with at EPITA. I first want to thank the other PhD students who supported me during these three years. To Charles, my dear beer companion who has been one of my greatest support, Michaël, my (partially)

New Zealand companion, who has been impossible to convince during all our political debates (and there was a lot), Caroline, my Brazilian companion who learned French incredibly fast (and I hope not too much of mine), Yizi, for our Sunday afternoon discussions about Paris historical map segmentation at 2Bis, Florian, for your riddles coupled with jokes (I never found an answer), Adam, for the parties at your home which were great moments of relaxation, ZZ and Minh, who tried to initiate me to communism, Anissa, Giulio, Glen, Antoine M., and Théo. I also must thank Valentine and Antoine B., who shared the PhD student office and greatly contributed to its conviviality.

Then the permanent researchers at LRE. I am grateful toward the permanents from the Image team, notably Élodie, Jonathan, Nicolas, Joseph, and Olivier. With Guillaume and Edwin, you were my teachers and a large part of what I know about Image Processing comes from your classes (for Jonathan, I still have an *insomnie* folder in my computer just in case). The discussions I had with you solved a lot of my issues during these last five years. Furthermore, you introduced me to teaching, which turned out to be a wonderful experience. I hope that, in the future, I will have opportunities to work with you again.

I want to thank all the members of the Automata team for their conviviality, and particularly Amazigh for the large amount of jokes you taught me (and I reuse them EVERY SINGLE DAY). I also want to thank Nida for our work breaks, David for your conviviality, Daniela for your help with the whole amount of administrative stuff I had to do during these last three years, and Theo (the boss), who accepted me to pursue a PhD at LRDE and gave me useful advice these last years. I also want to thank people who left the laboratory during my PhD, and particularly Clément for your technical support and Étienne for our morning talks.

Finally, this adventure would not have been possible without the constant support of my friends and family. I particularly want to thank Charles V. (still you), Charles G. (not the same), Henri, Thomas, and Sarah.

Image processing libraries play an important role in the researcher toolset and should respect three criteria: genericity, performance, and interactivity. In short, genericity boosts code reuse and algorithm flexibility for various data inputs, while performance speeds up experiments and supports real-time applications. Additionally, interactivity allows software evolution and maintenance without full recompilation, often through integration with dynamic languages like Python or Julia. The first two criteria are not straightforward to reach with static languages such as C++ or Rust which require knowing some information at compile-time to optimize generated machine code related to the different input and output data types of an algorithm. The latest criterion usually requires waiting until runtime to obtain type information and is thus performed at the cost of runtime efficiency. The work presented in this thesis aims to go beyond this limitation in the context of image processing algorithms. To do so, a methodology to develop generic algorithms whose type information about its input and output data may be known either at compile-time or at runtime is presented. This methodology is evaluated on different image processing algorithmic schemes, and it is concluded that the performance gap between the runtime and compile-time versions of the construction algorithm for hierarchical representations of images is negligible. As an application, hierarchical representations are employed to expand the applicability of grayscale noise level estimation to color images to enhance its genericity. That raises the importance of studying the impact of such corruption in the hierarchies built on noisy images to improve their efficiency in the presence of noise. It is demonstrated that the noise has an impact on the tree structure, and this impact is related to some kinds of functional in the context of energy optimization on hierarchies.

Les bibliothèques de traitement d'images jouent un rôle important dans la boîte à outils du chercheur et devraient respecter trois critères : généricité, performance et interactivité. La généricité favorise la réutilisation du code et la flexibilité des algorithmes pour diverses structures de données en entrée, tandis que la performance accélère les expériences et permet l'utilisation d'algorithmes dans le cas d'applications en temps réel. De plus, l'interactivité dans la chaîne de traitement d'une image permet d'effectuer des expérimentations en échangeant des données avec cette dernière. Ce dernier critère est généralement obtenu en ajoutant du dynamisme à la bibliothèque, et plus particulièrement en interfaçant ses fonctionnalités à un langage dynamique. Les deux premiers critères peuvent être atteints avec des langages statiques tels que C++ ou Rust, qui exigent la connaissance de certaines informations au moment de la compilation pour optimiser le code machine généré en fonction des différents types de données d'entrée et de sortie d'un algorithme. Le dernier critère nécessite généralement d'attendre jusqu'à l'exécution pour obtenir des informations sur le type, et est donc réalisé au détriment de la vitesse d'exécution. Le travail présenté dans cette thèse vise à dépasser cette limitation dans le contexte d'algorithmes de traitement d'images. Pour ce faire, une méthodologie visant à développer des algorithmes génériques dont les informations sur les types d'entrée et de sortie peuvent être connues soit au moment de la compilation, soit à l'exécution, est présentée. Cette méthode est évaluée sur différents schémas algorithmiques de traitement d'images, et il est conclu que l'écart de performance entre les versions où l'information de type est connu à la compilation et à l'exécution de l'algorithme de construction pour les représentations hiérarchiques d'images est négligeable. En tant qu'application, les représentations hiérarchiques sont utilisées pour étendre l'applicabilité de l'estimation du niveau de bruit en niveaux de gris aux images en couleur afin d'améliorer leur caractère générique. Cela soulève l'importance d'étudier l'impact d'une telle altération dans les images à partir desquelles les représentations hiérarchiques sont construites pour améliorer l'efficacité de leurs applications en présence de bruit. Il est démontré que le bruit a un impact sur la structure arborescente, et cet impact est lié à certains types de fonctionnelles dans le cas où les hiérarchies sont contraintes par une énergie.

Résumé long

1 Introduction

Le domaine du traitement d'images représente un champ de recherche exigeant une vaste gamme d'outils pour concevoir et expérimenter de nouvelles méthodologies. Ces outils englobent des utilitaires, des programmes intégrant des fonctionnalités de traitement d'images, accessibles soit en ligne de commande, comme c'est le cas d'ImageMagick [201], soit au sein d'interfaces graphiques, à l'instar de Gimp [209]. De plus, ces outils incluent des bibliothèques offrant diverses implémentations d'algorithmes utilisables dans un ou plusieurs langages de programmation. Enfin, ils englobent également des environnements de programmation tels que Matlab [95], Jupyter [168], ou Pluto.jl [169], intégrant diverses fonctionnalités ergonomiques basées sur un langage de programmation et se situant à l'intersection des utilitaires graphiques et des bibliothèques.

Dans le cadre de cette thèse, l'objet d'étude concerne les bibliothèques de traitement d'images. Elles occupent une position centrale parmi les autres outils présentés du fait de leur utilisation pour la séparation des différentes fonctionnalités fournies par les utilitaires, ou pour une utilisation au sein d'un d'environnement de programmation. Ces bibliothèques de traitement d'images devraient satisfaire trois critères fondamentaux pour permettre une utilisation optimale : *généricité*, *performance* et *interactivité*. La *généricité* [146] représente une méthodologie en programmation visant à améliorer la réutilisabilité d'un algorithme sur différentes structures de données, à condition que celles-ci respectent une interface prédéfinie, évitant ainsi de prendre en compte l'implémentation spécifique de la structure. Ainsi, ce critère permet d'appliquer un même algorithme à diverses images, en tenant compte des différentes caractéristiques telles que le domaine de définition, l'espace des valeurs d'un pixel ou l'implémentation, pouvant varier en fonction du contexte.

Listing 0.1: Exemple d'algorithme de traitement d'images générique C++

```cpp
template <typename C> requires Container<C>
typename C::value_type sum(const C& cont) {
    typename C::value_type res{}
    for (auto it = cont.begin(); it != cont.end(); ++it)
        res += it;
    return res
}
```

La *performance* des fonctionnalités d'une bibliothèque de traitement d'images permet son utilisation sur des images de grandes dimensions, ainsi que le développement d'applications en temps réel lorsque les performances atteintes le permettent. Enfin, le critère d'*interactivité* a une importance particulière dans le processus de recherche, notamment dans le cadre des expérimentations. Il permet au chercheur d'incorporer et d'acquérir des données tout au long des expériences. Cette interaction est généralement facilitée par l'utilisation de langages dynamiques tels que Python ou Julia, dotés d'écosystèmes scientifiques étendus fournissant les ressources essentielles pour une conduite efficace du processus expérimental.

La bibliothèque de traitement d'images Pylene, implémentée en C++ moderne (C++20), vise à atteindre au mieux ces trois critères. Cette objectif constitue un défi complexe en raison de divers facteurs. Dans des langages tels que C++ ou Rust, le critère de généricité est implémenté à travers l'utilisation de paramètres et repose sur le mécanisme de monomorphisation. Ce processus génère, pour chaque combinaison de paramètres, du code fortement spécialisé et optimisé par le compilateur. Cela permet d'atteindre dans le même temps le critère de performance. Cependant, le dynamisme sur lequel repose généralement le critère d'interactivité est difficile à obtenir car la vérification de type s'effectue à la compilation, nécessitant ainsi de connaître tous les paramètres avant l'exécution.

Ainsi, l'objectif de cette thèse est le suivant : intégrer le critère d'interactivité dans des algorithmes génériques en C++, tout en mesurant et en limitant la perte de performance occasionnée par cette intégration. Nous nous concentrons sur des algorithmes de morphologie mathématique, et plus particulièrement les représentations hiérarchiques d'images, en adaptant l'implémentation de leur construction à la généricité dans un contexte dynamique. Afin de montrer l'utilité de ces adaptations dans un contexte de recherche en traitement d'images, nous utilisons ces représentations hiérarchiques dans le cadre d'images bruitées, notamment dans le contexte de l'estimation du niveau de bruit. Étant donné que cette application met en évidence le manque de compréhension de l'impact du bruit sur une représentation hiérarchique d'image, nous analysons l'évolution de la structure de l'arbre qui la représente en fonction du niveau de bruit affectant l'image sur laquelle l'arbre est construit. Ainsi, cette thèse se positionne à l'intersection des domaines du génie logiciel et du traitement d'images.

2 Généricité et traitement d'images

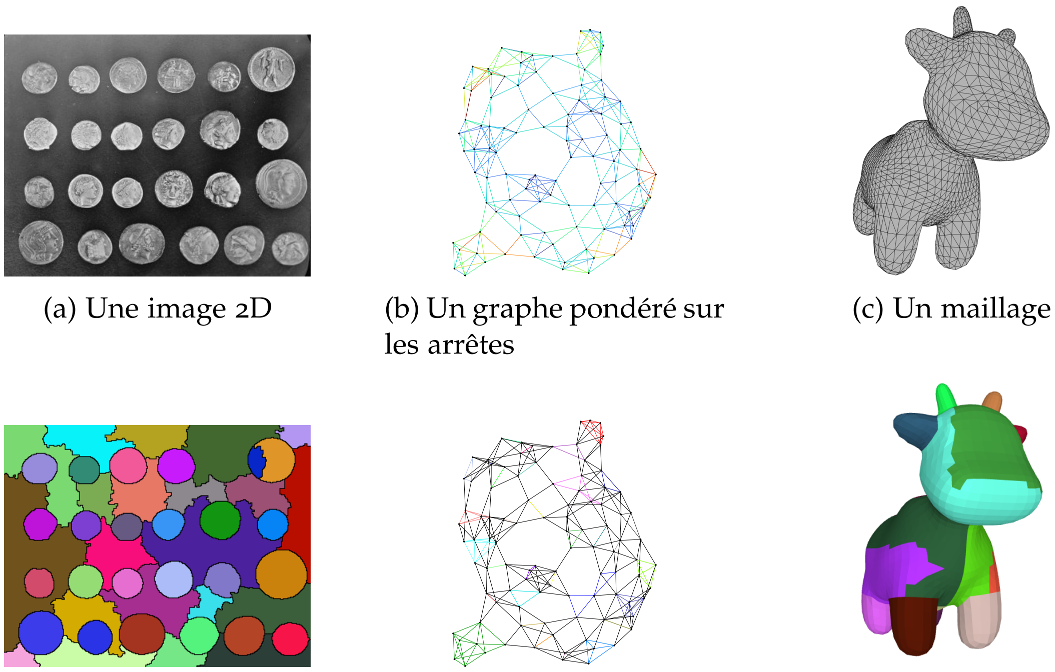
La généricité [146] est une approche de programmation permettant la réutilisation d'algorithmes sur différentes structures de données. Ces dernières doivent satisfaire un *concept* [54], c'est-à-dire un ensemble d'opérations et d'axiomes sur cette structure de données, de sorte que l'algorithme puisse lui être appliqué. Par exemple, le Listing 0.1 présente un exemple d'algorithme générique prenant en entrée n'importe quel conteneur dont le type est précisé par le paramètre de *template* `C` respectant le concept `Container`, qui énumère les opérations ainsi que les redéfinitions de type utilisées dans le cadre de l'algorithme, et produisant en sortie une valeur représentant la somme de toutes les valeurs du conteneur.

En traitement d'images, une image peut être définie par une fonction f tel que

$$f : \Omega \to \mathcal{V}$$

avec Ω le domaine de définition de la fonction et $\mathcal{V}$ son ensemble de valeurs. À partir de cette définition, il est possible de définir le concept d'image comme étant une structure de données possédant des opérations similaires à celles d'une fonction : accès au domaine de définition, à l'ensemble des valeurs et à la valeur associée à un point du domaine. Le choix de l'implémentation d'une image peut donc être multiple du fait que le concept ne dépend pas de l'implémentation mais de l'interface. Par exemple, une image constante peut être implémentée par une simple valeur, et l'accès à n'importe quel point du domaine retournera la même valeur.

L'ensemble des valeurs de l'image peut changer en fonction du contexte : une image pourra être définie comme ayant un ensemble d'entiers dont les valeurs représentent l'intensité du gris dans l'image, ou comme un ensemble de triplets, ces derniers représentant une couleur dans l'espace Rouge-Vert-Bleu (RVB). De plus, le choix du domaine de définition ne se limite pas à un rectangle pour des images en dimension 2, mais il est possible de définir une image sur un hyperrectangle pour qu'elle soit de dimension n, ainsi que sur des graphes, qu'ils soient pondérés sur les arêtes ou sur les sommets, ou bien encore sur



(a) Une image 2D

(b) Un graphe pondéré sur les arrêtes

(c) Un maillage

(d) Ligne de partage des eaux de (a)

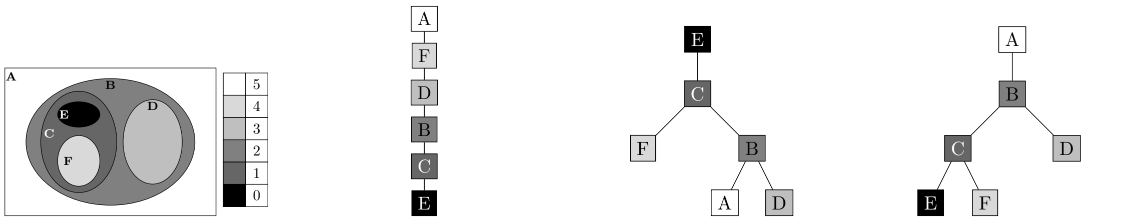
(e) Ligne de partage des eaux de (b)

(f) Ligne de partage des eaux de (c)

Figure 1: Illustration de la généricité en traitement d'images [119]

des maillages triangulaires, très utilisés dans le domaine de la visualisation 3D. En ajoutant des caractéristiques à ces domaines de définition, telles que le voisinage d'un point du domaine, il est possible d'appliquer divers algorithmes de traitement d'images. La Figure 1 illustre cette notion de généricité en traitement d'images, où l'algorithme de ligne de partage des eaux, largement utilisé pour la segmentation d'image et relevant du domaine de la morphologie mathématique, est appliqué sur différents types d'images [119].

Les représentations hiérarchiques d'images [24] se présentent sous la forme de piles de partitions (partielles), où les régions s'agrandissent à mesure que l'on progresse dans la hiérarchie. Trois des ces représentations sont illustrées dans la Figure 2. Étant donné que la morphologie mathématique [188, 196, 153], domaine auxquelles appartiennent ces représentations, implique des opérations de nature générique, comme observé précédemment pour l'algorithme de ligne de partage des eaux, il est envisageable de construire ces représentations pour différents types d'images. Par exemple, une catégorie de hiérarchies de lignes de partage des eaux [47] est élaborée à partir de graphes pondérés sur les arêtes, la construction reposant sur la valeur de la pondération. Ce graphe peut être créé à partir d'un ensemble de points en utilisant l'algorithme des k plus proches



(a) Une image (b) Arbre min de (a) (c) Arbre max de (a) (d) Arbre des formes

Figure 2: Représentations hiérarchiques d'image

```
                template <typename I,
                          typename O,
                          typename F>
                void map(I, O, F)
                     ↙        ↓        ↘
map<image2d<uint8_t>,   ...   map<mesh<uint8_t>,
    image2d<double>,              mesh<rgb8>,
    inv>                          lut>
```

Figure 3: Illustration de l'explosion combinatoire

voisins, ou encore à partir de la relation de voisinage entre les triangles d'un maillage ou les points d'une image. Ainsi, les représentations hiérarchiques, du fait de leurs diverses applications telles que la segmentation d'image [163] ou l'estimation du niveau de bruit [63] dans une image en niveaux de gris, suscitent un intérêt particulier en tant que domaine d'application de la généricité statico-dynamique proposée dans la section suivante.

3 La généricité statico-dynamique pour le traitement d'images

Tel qu'expliqué précédemment, les langages statiques offrent peu de flexibilité en raison de leur nature pour concevoir une bibliothèque de traitement d'images respectant le critère d'interactivité. Le langage C++ dispose d'un mécanisme de généricité statique basé sur la monomorphisation : pour chaque combinaison de paramètres d'une fonction, du code machine est généré à la compilation, permettant d'obtenir un algorithme particulièrement optimisé pour cette combinaison de paramètres. Néanmoins, en raison du grand nombre de structures d'image, il est difficile, voire impossible, de spécifier chaque combinaison de paramètres à la compilation, ce qui entraîne une explosion

```
template <class T>
struct buffer2d
{
  T& operator()(point2d p);
  rect2d domain() const;          Interface
- - - - - - - - - - - - - - - - - - - - - - -
  T* data;                        Détails
};                                d'implémentation
```

(a) Valeurs typées statiquement

```
struct buffer2d_any
{
  void* operator()(point2d p);
  rect2d domain() const;          Interface
- - - - - - - - - - - - - - - - - - - - - - -
  void* data;                     Détails
  size_t element_size;            d'implémentation
};
```

(b) Valeurs typées dynamiquement

Figure 4: Structures d'image avec différentes stratégies de typage des valeurs

combinatoire. Cela est illustré par la Figure 3, dans laquelle la fonction `map` prend en entrée une image et applique une opération dont le résultat est stocké dans une seconde image. Pour cette fonction, il existe une multitude de types d'images d'entrée et de sortie possibles, ainsi que d'opérations réalisables sur les valeurs d'une image. Ainsi, l'objectif est de supprimer le besoin de spécialiser l'algorithme pour chacun de ces différents types tout en limitant la perte de performance à l'exécution. De plus, il est avantageux de conserver la possibilité d'avoir des algorithmes typés statiquement dans le cas où l'application ne nécessite pas d'interactivité. Ainsi, dans cette section, le principe de généricité statico-dynamique est présenté avec pour objectif de remplir ces deux critères.

3.1 Modèles d'image et d'algorithme pour la généricité statico-dynamique

Comme précisé précédemment, il existe un très grand nombre d'images pouvant être appliqué à un algorithme générique en raison de la combinatoire liée au domaine de définition, à l'ensemble des valeurs et à l'implémentation de l'image. Dans cette thèse, le domaine considéré pour une image est un rectangle en 2 dimensions. Ainsi, les différents paramètres étudiés concernent l'implémentation de l'image et son ensemble de valeurs.

En C++, diverses techniques sont disponibles pour introduire du dynamisme. Parmi celles-ci, l'effacement de type est une technique largement utilisée dans la Standard Template Library (STL) [198] pour des conteneurs tels que `std::any` [53], qui stocke un objet dont le type n'est connu qu'à l'exécution et effectue une conversion à l'utilisation, ou `std::function`, similaire à `std::any` mais limitée aux objets appelables comme une fonction, se basant ainsi sur un tableau virtuel pour l'appel des fonctions. L'effacement de type stocke ainsi un objet dans un espace mémoire alloué à la taille de l'objet, mais ne nécessite pas l'information de type à la compilation.

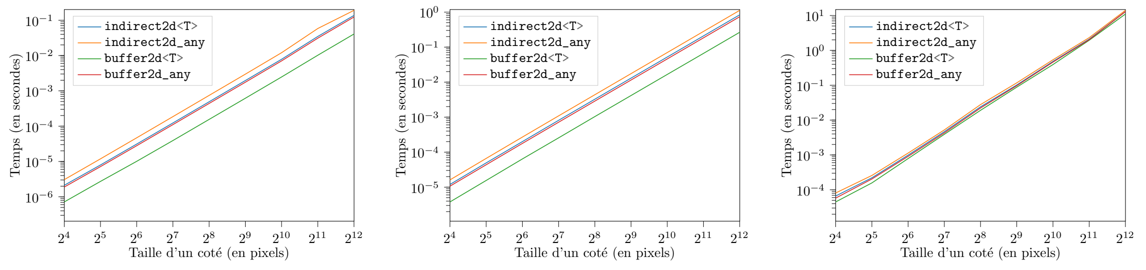
Ainsi, pour parvenir à un typage dynamique de l'implémentation d'une image ou de ses valeurs, l'effacement de type est employé. En ce qui concerne

Listing 0.2: Algorithme statico-dynamique C++

```cpp
template <class I, class Op>
void generic_elementwise_op(I a, I b, I out, Op& op) {
    for (auto p : a.domain())
        op(a(p), b(p), out(p))
}
```

les valeurs de l'image, un tableau est alloué, dont la taille est le produit du nombre de pixels d'une image et de la taille en octets de l'espace nécessaire pour stocker une valeur. De plus, pour parcourir l'image, cette information est stockée dans les détails de l'implémentation. Cette structure d'image est comparée à une implémentation statique caractérisée par une liste de paramètre de template dans la Figure 4. Dans cette figure, les structures de données représentant une image implémentée comme un tableau contigu sont divisées en deux parties : une interface commune aux deux images, avec une opération d'appel de fonction prenant un point de l'image et retournant une valeur pour ce point, ainsi qu'une opération permettant d'obtenir le domaine d'une image. La valeur pour un point de l'image est soit celle du pixel, soit l'adresse le contenant. Les détails de l'implémentation stockent les informations permettant de manipuler cette image à partir de l'interface. L'extension au dynamisme de l'implémentation d'une image est similaire à la méthode proposée pour les valeurs d'une image : l'effacement de type est utilisé par le biais du conteneur `std::function`, qui stocke l'implémentation à l'exécution et retourne une valeur pour un pixel donné lors de son appel de fonction prenant un point de l'image en argument. Ainsi, seuls les détails d'implémentation sont modifiés, l'interface restant inchangée. Ce modèle est donc étendu aux valeurs statiquement et dynamiquement typées, permettant l'utilisation de n'importe quelle implémentation d'une image et augmentant ainsi la flexibilité de la structure au prix d'une indirection.

La programmation générique repose sur une interface commune entre différentes structures de données, permettant à un algorithme d'être réutilisable indépendamment de l'implémentation de la structure. Les quatre modèles d'image présentés précédemment possèdent cette interface commune. Cependant, la gestion du type des valeurs nécessite d'adapter les algorithmes en fonction du caractère statique ou dynamique du type. Dans le premier cas, une valeur est retournée, tandis que dans le second cas, un pointeur vers la valeur est retourné au lieu de la valeur elle-même. Ainsi, la méthode proposée s'inspire de la fonction `qsort` de la bibliothèque standard du langage C, qui prend un tableau de valeurs non typées, des informations sur le tableau



(a) Maximum (b) Dilatation (c) Arbre max

Figure 5: Évaluation des structures d'image à trois schémas algorithmiques

et le type, ainsi qu'une fonction de comparaison, et trie le tableau. Ainsi, il est possible d'implémenter des algorithmes génériques en spécifiant les opérations nécéssaires en argument de la même manière que pour la fonction de comparaison de `qsort`, tel qu'illustré par le Listing 0.2. Comme la fonction est paramétrée, le type doit être spécifié à la compilation. Dans le cas d'un typage de valeur dynamique, l'utilisation d'un `std::function` est valide pour effectuer les conversions nécessaires pour manipuler les valeurs, mais cela nécessite néanmoins une nouvelle indirection.

Trois algorithmes avec des schémas d'accès à la donnée différents sont évalués et le résultat est affiché dans la Figure 5, Les `buffer2d` représentent des implémentations intégrant directement dans leur structure un tableau contigu de valeurs, tandis que les `indirect2d` stockent leur implémentation dans la mémoire avec un type effacé. L'observation des courbes de performance permet d'évaluer le coût des indirections en fonction du schéma algorithmique utilisé. L'application de l'algorithme de construction de l'arbre max sur des images effetuant des indirections, que ce soit sur ses valeurs ou son implémentation, résulte en un écart plus faible avec son application sur une image sans indirection par rapport aux autres schémas algorithmiques en raison de son parcours dans la mémoire qui rend difficile des optimisations telles que la vectorisation des instructions par le compilateur ou l'accès à la mémoire tampon du processeur. Par conséquent, l'utilisation du dynamisme pour les valeurs et les implémentations d'une image dans cet algorithme, et plus généralement dans la construction d'une représentation hiérarchique d'image, revêt un intérêt particulier en terme de performance.

3.2 Application des concepts dans la construction de l'arbre max

Les résultats de l'évaluation des performances de l'algorithme de construction de l'arbre max montrent un écart de performance négligeable entre les structures requérant une connaissance des informations de type à la compilation et à l'exécution. En conséquence, une étude approfondie de l'algorithme de Berger [16] est entreprise. Dans un premier temps, nous identifions que l'accès aux valeurs d'une image, qu'il soit effectué directement ou indirectement, avec ou sans la connaissance des informations de leur type, intervient à deux moments précis de l'algorithme : lors du tri des valeurs et lors de la canonicalisation de l'arbre. Cette dernière étape consiste à attribuer à chaque nœud de l'arbre, correspondant à un pixel d'une composante, un nœud représentatif à cette composante comme parent. Ainsi, la structure de l'*union-find* au cœur de l'algorithme de construction n'exige en aucun cas de connaître ces informations. Elle repose sur des objets créés à partir des informations de domaine de l'image qui, dans ce manuscrit, sont limitées à des images 2D. Enfin, une partie de ces objets est renvoyée par l'algorithme et est typée statiquement car, dans le cas d'une image 2D, peu importe l'implémentation de l'image ou le type des valeurs, celles-ci peuvent être connues à la compilation.

Les expérimentations menées pour étendre l'évaluation des performances de l'algorithme de construction de l'arbre max utilisent les structures d'image précédentes, tout en explorant de nouvelles approches pour les structures d'image données en entrée de l'algorithme. La première approche repose sur l'utilisation de *projecteurs*, qui sont des fonctions transformant une valeur de l'image en un type différent. Ainsi, à chaque lecture de la valeur d'un pixel de l'image, cette valeur est convertie dans un type donné. Cette projection peut être réalisée de manière statique, où les informations de type en entrée et en sortie du projecteur sont connues dès la compilation, ou de manière dynamique, où seule l'information de type de sortie est connue à la compilation. Cette approche permet de compiler l'algorithme une seule fois en connaissant statiquement les types de valeurs, évaluant ainsi uniquement le coût de la conversion. La seconde approche implique la conversion et la copie des valeurs de l'image dans une nouvelle image, où les valeurs sont typées statiquement, offrant un accès direct à l'implémentation. Ensuite, l'algorithme de construction est appliqué à cette nouvelle image. Bien que cette approche n'accepte qu'un seul type d'image en entrée de l'algorithme de construction, elle nécessite l'allocation d'une nouvelle image et une copie complète de l'image originale.

Ces différentes approches sont évaluées de manière statique et dynamique concernant le type des valeurs de l'image, ainsi que de manière directe et indirecte par rapport à l'implémentation de cette image. À cet effet, toutes les images d'entrée sont encodées sur 8 bits, et la projection ainsi que la conversion transforment une valeur sur 8 bits en une valeur sur 64 bits. Les variations de performance par rapport à la structure d'image statique sur les valeurs et l'accès direct à l'implémentation indiquent que la projection présente un coût plus élevé que les structures d'images précédemment étudiées. Cependant, ce coût est considérablement plus faible dans le cas de la conversion, surtout lorsque les images ont des dimensions comprises entre 64×64 et 512×512, où celui-ci devient négatif.

4 Applications en morphologie mathématique dans le contexte d'image bruitée

L'efficacité de l'utilisation des représentations hiérarchiques a été démontrée dans diverses applications telles que la segmentation [83] ou la détection d'objet [222]. Ainsi, pour illustrer la nécessité d'avoir leurs algorithmes de construction dans un contexte interactif en utilisant un langage dynamique tel que Python au sein d'un environnement dynamique tel que Jupyter pour le prototypage de méthodes, deux applications ont été étudiées. La première vise à étendre une méthode d'estimation du niveau de bruit pour les images en niveaux de gris [63] aux images couleur dans le but de rendre la méthode plus générique. La seconde application est l'étude de l'influence du bruit dans une image sur la structure d'une représentation hiérarchique construite sur celle-ci.

4.1 L'estimation du niveau de bruit d'une image

Il existe différentes méthodes pour estimer le niveau de bruit dans une image. Certaines dépendent d'hypothèses a priori sur la nature statistique du bruit [127, 147, 236], tandis que d'autres estiment d'abord la nature du bruit avant d'en estimer les paramètres [11, 204]. Dans [63], nous avons proposé d'adapter la méthode de [204] au contenu d'une image, cette méthode étant basée sur des patches rectangulaires. L'objectif est d'estimer une fonction de niveau de bruit (FNB) [127] définie sous la forme d'un polynôme de degré 2, où chaque coefficient correspond au paramètre d'une distribution statistique du bruit. Ainsi, estimer la FNB équivaut à estimer la nature du bruit corrompant l'image et ses paramètres simultanément. De plus, une telle fonction permet de représenter un bruit mixte tel que le bruit Poisson-Gaussien.

Le problème avec la méthode de [63] réside dans son application limitée aux images en niveaux de gris. En effet, deux des éléments clés de cette méthode, à savoir le coefficient de rang τ de Kendall et l'arbre des formes (une

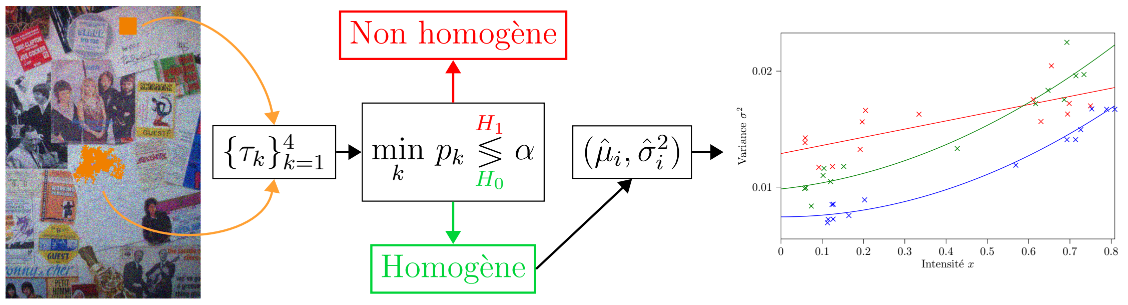

Non homogène
$\{\tau_k\}_{k=1}^4$
$\min_k p_k \overset{H_1}{\underset{H_0}{\lessgtr}} \alpha$
$(\hat{\mu}_i, \hat{\sigma}_i^2)$
Homogène
Variance σ^2
Intensité x

Figure 6: Méthode d'estimation de niveau de bruit multivarié

représentation hiérarchique de l'image basée sur la morphologie mathématique, représentant l'inclusion des régions d'une image), se fondent sur le rang des valeurs de l'image. Or, un tel rang n'est pas naturel pour des valeurs multivariées représentant des couleurs. Ce problème est bien connu dans le domaine de la morphologie mathématique [176] et a fait l'objet d'études approfondies [5].

Pour pallier le problème du rang des valeurs dans une image, deux outils sont introduits dans la méthode. Tout d'abord, l'utilisation de l'arbre des formes multivarié [38] (AdFM) en remplacement de l'arbre des formes classique. L'AdFM, applicable aux images couleur, notamment pour des tâches de segmentation d'image en utilisant la fonctionnelle de Mumford-Shah [240], qui aboutit à une partition optimale de l'image en fonction de cette fonctionnelle, nécessite peu de modifications dans la partie segmentation de la méthode. Ensuite, le calcul du coefficient de rang τ de Kendall n'est plus effectué directement sur l'image, mais sur une carte de rang des valeurs multivariées obtenue par l'apprentissage d'un treillis complet [124]. Ainsi, l'adaptation de ces outils aux images couleur permet d'obtenir une estimation de la fonction de niveau de bruit multivariée (FNBM), constituée d'une FNB pour chaque canal, comme illustré par la Figure 6. Cette estimation vise à identifier les régions homogènes de l'image, c'est-à-dire celles ne contenant que du bruit, afin de calculer les moyennes et variances empiriques de chacune de ces régions. La FNBM est ensuite estimée à partir de ces informations.

En plus de l'adaptation à la couleur, diverses optimisations ont été ajoutées afin d'améliorer la précision de l'estimation. Pour évaluer cette précision, la méthode par blocs de [204] a été étendue à la couleur, et la précision de cette estimation a été comparée à celle obtenue en utilisant une erreur relative moyenne (ERM) entre une FNBM résultant de l'estimation et une FNBM dont les coefficients, connus à l'avance, sont utilisés pour ajouter du bruit à une image. Les deux estimations ont été réalisées sur 150 images naturelles [42],

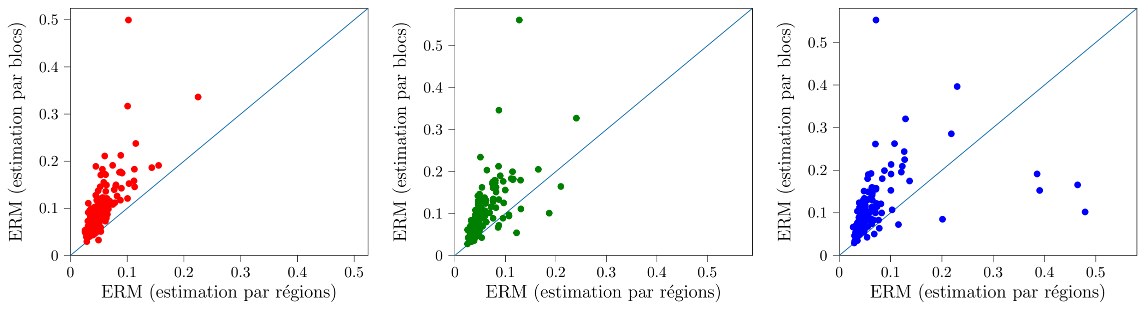


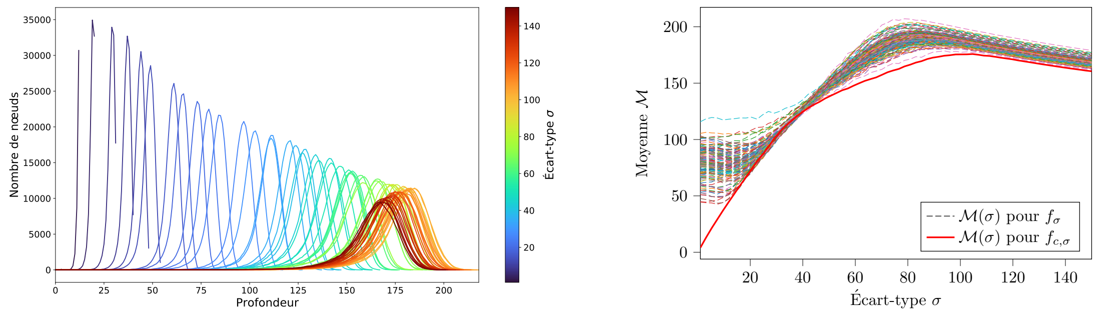
Figure 7: Comparaison entre l'estimation par blocs et l'estimation par régions

et les ERM sont présentées dans la Figure 7. Ces résultats mettent en évidence que l'adaptation au contenu de l'image résulte en une estimation plus précise par rapport à l'utilisation de blocs, comme en témoigne un ERM plus faible pour la méthode proposée sur chaque canal.

De plus, l'estimation en couleur a été étendue à toute la hiérarchie, plutôt que de se limiter à un sous-ensemble de partitions. Du fait que les résultats obtenus avec l'AdFM soient moins précis que ceux de la méthode précédemment proposée, l'arbre ω [195], une hiérarchie de partitions calculée à partir d'une dissimilarité locale (distance entre deux pixels voisins) couplée à un critère global (taille de la diagonale de la boîte englobante de l'espace des valeurs d'une région), est utilisé. Cette hiérarchie permet d'obtenir une similarité en termes de précision avec la méthode précédemment proposée. L'utilisation d'une hiérarchie complète n'améliorant pas les résultats de l'estimation, la question suivante se pose : quel est l'impact du bruit présent dans une image sur la structure d'une représentation hiérarchique ?

4.2 L'impact du bruit d'une image sur la structure de sa hiérarchie

Dans la section précédente, l'estimation du niveau de bruit est réalisée en utilisant toutes les régions d'une représentation hiérarchique d'image, visant à obtenir des régions homogènes ne contenant que du bruit. Cependant, cette approche ne conduit pas à une amélioration par rapport à la méthode précédemment développée qui utilise seulement un sous-ensemble des régions de la hiérarchie. Cela soulève des questions sur l'influence du bruit dans une image sur la construction d'une représentation hiérarchique. Dans le cadre de l'estimation du niveau de bruit d'une image, l'arbre ω est employé. Dans cette partie, nous considérons l'arbre α pour étudier l'impact du bruit sur sa structure. Cette hiérarchie est fondamentale pour la construction de l'arbre ω, ainsi que



(a) Évolution des histogrammes de profondeur des nœuds de l'arbre construit à partir d'une image constante bruitée

(b) Évolution des moyennes de ces histogrammes sur la base de 150 images [42]

Figure 8: Distribution des profondeurs des nœuds de l'arbre α en fonction du niveau de bruit.

pour d'autres hiérarchies en raison de leurs liens [49].

Parmi les différentes possibilités pour étudier la structure de l'arbre, la profondeur de chaque nœud de l'arbre est utilisée. La profondeur d'un nœud correspond au nombre de nœuds sur le chemin entre le nœud courant et la racine de l'arbre. Pour cette étude, la distribution de ces profondeurs, calculée à partir d'un histogramme, est étudiée. Afin de mesurer l'impact du bruit d'une image sur la structure de sa hiérarchie, les distributions de profondeurs des nœuds d'un arbre α construit sur une image constante bruitée avec différents niveaux de bruit (voir Fig. 8a) sont observées, puis leurs moyennes sont comparées avec celles obtenues à partir d'arbres construits sur 150 images naturelles (voir Figure 8b). Ces observations montrent que pour de faibles niveaux de bruit, le contenu de l'image prédomine par rapport au bruit. À mesure que ce paramètre augmente, les moyennes des distributions de profondeurs pour l'image constante bruitée et les images naturelles bruitées évoluent de manière similaire jusqu'à atteindre un certain niveau. Au-delà de ce seuil, cette évolution devient décroissante. Cette dernière observation est liée au bornage des valeurs de l'image dans l'intervalle $[\![0..255]\!]$, créant ainsi de nouvelles composantes dans l'image.

De plus, l'impact du bruit sur l'évolution du nombre de nœuds non persistants dans une hiérarchie [83] est évalué en fonction du niveau de bruit de l'image. Ces nœuds représentent des régions qui n'appartiennent à aucune partition optimale $\mathbf{P}^*$, selon une énergie de la forme $E_\lambda(\mathbf{P}^*) = D(\mathbf{P}^*) + \lambda C(\mathbf{P}^*)$, où D est un terme de fidélité aux données de l'image, C est un terme de régularisation et λ est un paramètre de régularisation de cette énergie. Pour cette évaluation, nous avons utilisé deux énergies : la fonctionnelle de

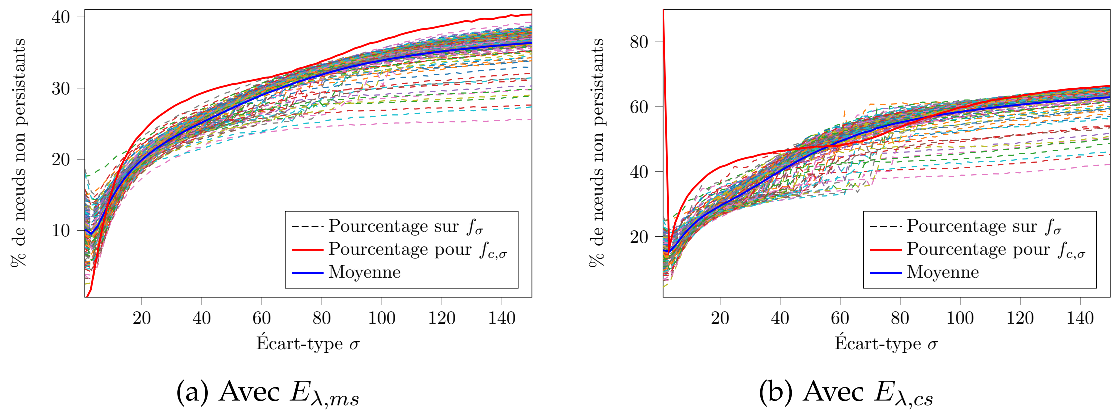


Figure 9: Évolution du nombre de nœuds de l'arbre non-persistants en fonction du niveau de bruit

Mumford-Shah [142] ($E_{\lambda,ms}$) ainsi qu'une autre ($E_{\lambda,cs}$) dérivée de celle-ci en remplaçant le terme de régularisation par la somme des valeurs aux contours d'une région. Les évolutions sont affichées dans la Figure 9, où la courbe bleue représente le pourcentage moyen de nœuds non persistants dans les 150 images, et la courbe rouge représente ce même pourcentage pour l'image constante bruitée. Nous pouvons déduire de ces courbes que l'utilisation de la fonctionnelle $E_{\lambda,cs}$ est plus pertinente que $E_{\lambda,ms}$ dans un contexte d'image bruitée, car elle élimine un plus grand nombre de nœuds non robustes au bruit dans la hiérarchie.

5 Conclusion et perspectives

Cette thèse présente une méthodologie permettant de fusionner la généricité statique et dynamique au sein d'un même algorithme, visant ainsi à respecter de manière optimale le critère d'interactivité pour une bibliothèque de traitement d'images. Ensuite, la généricité statico-dynamique est appliquée à un algorithme de représentation hiérarchique d'images appartenant à la morphologie mathématique. Cela est motivé par le constat que la perte de performances dans un contexte dynamique demeure négligeable par rapport aux performances dans un contexte statique. Ainsi, afin de démontrer l'utilité d'une bibliothèque performante intégrant ces trois critères dans le contexte des représentations hiérarchiques d'images, une application est proposée pour l'estimation du niveau de bruit. Cette application englobe notamment l'extension d'une méthode, initialement conçue pour les images en niveaux de gris, aux images couleur, dans le but d'améliorer les capacités génériques de cette méthode, puis à une hiérarchie complète afin d'en améliorer la, ce qui ne fut pas le cas, menant à l'étude de l'impact du bruit d'une image à la construction d'une hiérarchie.

Ainsi, de nombreuses perspectives s'ouvrent à la suite de cette thèse. Dans le cadre de la généricité statico-dynamique appliquée au traitement d'images, seuls l'espace des valeurs et l'implémentation sont actuellement pris en compte, le domaine étant limité à une grille rectangulaire en 2D. Cependant, comme illustré par la Figure 0.1, le champ des possibilités en matière de domaine est vaste. Celui-ci englobe les graphes, les maillages, ainsi que d'autres types de domaines en 2D, tels qu'une grille hexagonale par exemple. Ainsi, dans un premier temps, la généricité statico-dynamique pour le traitement d'images sera étendue à différents types de domaines. Par ailleurs, dans le contexte de la généricité statico-dynamique en nD, l'exploitation de la compilation à la volée sera examinée pour réduire l'écart de performance entre les versions statiques et dynamiques, en particulier pour générer du code machine vectorisé à l'exécution. Enfin, dans le domaine des représentations hiérarchiques d'images, la parallélisation de l'algorithme de construction à travers la division de l'image en diverses sous-images, appelées *tuiles*, ainsi que le calcul de l'arbre sur chacune de ces tuiles en vue de les fusionner en un seul arbre, seront étudiés afin de minimiser les pertes de performance.

Dans la partie traitement d'images, l'estimation du niveau de bruit s'effectue sans considérer la corrélation du bruit entre ces canaux due à la fusion des pixels lors du démosaïquage pour former l'image en couleur. Ainsi, l'estimation de la FNBM doit être adaptée pour prendre en compte cette problématique, de même que son estimation. Par ailleurs, l'évaluation de cette estimation repose sur une mesure d'erreur de précision, mais ses résultats ne sont pas analysés dans le contexte d'une application pratique, comme le débruitage. Ainsi, la méthode sera intégrée dans un algorithme de débruitage, et ses résultats seront comparés à ceux obtenus avec une estimation par bloc. En ce qui concerne l'évaluation de l'impact du bruit dans une image sur la construction d'une hiérarchie, les résultats obtenus seront utilisés pour élaborer une mesure de similarité entre deux hiérarchies, prenant en considération le contenu de l'image. Cette mesure sera ensuite mise en relation avec le niveau de bruit de l'image. Par conséquent, cela ouvre la voie à plusieurs applications utilisant les hiérarchies, telles que la segmentation ou le débruitage, pouvant être réalisées en prenant en compte le bruit affectant la structure de l'image, couplé à l'estimation du niveau de bruit développée précédemment.

Table of Contents

Chapter 1

Introduction

Mathematical morphology [153, 188] is a widely studied field and is used for several image processing tasks such as segmentation [19], filtering [189], classification [232], and more. Hierarchical representations of images [24] belong to mathematical morphology and allow the representation of an image as a tree. Such a multiscale representation is interesting as objects of interest in an image may be described by several regions, and the best fitting one depending on an application may be selected for use.

On the other hand, machine learning has greatly improved the efficiency of different image processing tasks in the past few years, notably through the use of neural networks from the deep learning field [78]. Such advances, on the will to associate artificial intelligence and mathematical morphology [22], have led to the incorporation of morphological operations into deep neural networks by building new network layers [4, 90, 132]. Recently, such associations have been extended to hierarchical representations of images. For example, the max-tree is used to create a new loss function based on some attributes computed on it for filtering purposes [162]. This new loss function has been used to automatically select markers for a segmentation procedure [67]. Furthermore, hierarchical segmentations may also be learned by a neural network and predicted as is the case in [116].

There exist numerous libraries to conceptualize and experiment with deep neural networks. Among them, Tensorflow [1] and PyTorch [160] are two well-known libraries to efficiently operate on tensors and thus build neural networks. Tensorflow is a library implemented in C++ and provides an interface to some other languages such as Python or Java. On the other hand, PyTorch is implemented in Python but for performance purposes, some of its functionalities are implemented in C++. Despite their differences, which are out of the scope of this thesis, these libraries fit in very well in the Python ecosystem: for example, they provide interoperability functions with NumPy [85], a library dedicated to the manipulation of arrays which is widely used in Python for scientific programming and whose arrays serve as a usual representation for n-dimensional images. Thus, performing image processing through deep learning in Python is executed with ease.

The Python programming language is a popular choice for research purposes. Its dynamic nature provides interactivity that facilitates the conception and manipulation of an experimental pipeline. With static programming languages, a modification such as a change of object type given as input to a pipeline requires recompiling the program as type checking is performed at compile-time. In Python, this is carried out at runtime such that a program can be modified during execution. Jupyter notebooks [168] enhance the ergonomy of such interactive capabilities by providing a graphical and extendible environment facilitating the reproducibility of the experiments. Furthermore, in addition to NumPy and the different deep learning frameworks cited above, its ecosystem contains plenty of scientific libraries for different application fields such as Scikit-Image [230] for image processing or NetworkX [84] for graph manipulation, but also some others which may be used whatever the field such as Scipy [227] which contains a large set of scientific functionalities to be used in conjunction with NumPy, or Matplotlib [92] for visualization purposes. However, the usual way to execute Python code is the usage of CPython [68], which interprets a given program and thus results in an overhead in terms of performance.

To avoid such a runtime overhead, several image processing libraries such as OpenCV [27], Pink [45], and Vigra [110] are implemented in C++. This language allows writing efficient implementations of algorithms because it generates specialized code according to its input argument types. However, such a process being performed at compile-time, dynamism is hard to reach without impacting the performance of a given functionality. These libraries provide Python interfaces but they have few generic capabilities: Vigra and OpenCV are specialized to n-dimensional images and their functionalities are brought to Python with a limited set of input types, particularly for the ones of the image values. On the other hand, Higra [164] is specialized in either edge-weighted or vertex-weighted graphs: images are required to be converted into those objects before an algorithm is applied.

Pylene [208] is an image processing library, specialized in mathematical morphology but not limited to, which aims to be generic, efficient, and interactive. It is implemented in modern C++ (C++20) and makes use of the language features to provide performant and generic implementations of image processing algorithms. Furthermore, some of its functionalities are exposed to a Python interface, but it still lacks of interactivity due to the nature of the language. Thus, the objective of this thesis is the development of a methodology that combines genericity and dynamism for static languages with the lowest loss of performance in the context of image processing algorithms.

Contributions

We propose the concept of *static-dynamic genericity* for image processing, a methodology that allows implementing algorithms such that they can be applied on image structures that do not necessarily know some information at compile-time. We evaluate the cost of the dynamism for such implementations, and more specifically in the context of a hierarchical representation construction algorithm. We thus propose to use our methodology for two applications. The first one is an extension of a noise level estimation pipeline for grayscale images to multivariate images using hierarchical representations. The second one is the study of the evolution of the structure of such hierarchies when built on noisy images according to their noise level.

Manuscript organization

This manuscript is divided into three parts which are themselves divided into several chapters.
Part I: Genericity and Image Processing. The first part recalls the basic notions used in our contributions but also highlights the links between them.

- **Chapter 2: Generic Programming.** This chapter introduces the basic concepts of *generic programming* [146], a methodology that allows writing an algorithm once and applying it to a wide range of data structures while respecting a given set of constraints on their interface. Furthermore, a comparison of the features provided by different programming languages to perform generic programming is described, with particular attention on the ones from C++ which is used throughout this thesis.

- **Chapter 3: Generic Image Processing and Its Applications.** In this chapter, the mathematical definition of an image is recalled and it is explained how generic programming can be applied in image processing from this definition. To this aim, different image structures are described, taking into account the fact that their value set, domain, and implementation may vary without impacting the implementation of generic algorithms. Finally, a comparison of the different image processing libraries providing mathematical morphology functionalities is provided with particular attention on their generic, performant, and interactive capabilities.

- **Chapter 4: Hierarchical Representations of Images.** This chapter reviews hierarchical representations of images, which are multiscale tools from mathematical morphology widely used for different image processing tasks such as segmentation or object detection.

Part II: A Static-Dynamic Approach to Image Processing. The second part is dedicated to combine at most genericity, performance, and interactivity for image processing algorithms. It evaluates for several cases the cost of dynamism required for interactivity, with particular attention on the max-tree construction, a hierarchical representation of images.

- **Chapter 5: Static-Dynamic Genericity for Image Processing.** The notion of *static-dynamic genericity* is described in this chapter for 2D images. This methodology allows to write generic algorithms such that some information about the input objects may be known either at compile-time or at runtime. Then, different algorithmic schemes used in image processing are implemented to compare the performance of different image structures whose information are either static or dynamic and evaluate the cost of dynamism for such algorithms.

- **Chapter 6: Static-Dynamic Hierarchy Construction.** From the results obtained in the previous chapter, special attention is given to a max-tree construction algorithm. More specifically, its adaptation to static-dynamic genericity is studied. Furthermore, new image structures are proposed and evaluated for such algorithms to be used dynamically.

Part III: Applications in Image Processing. The last part applies the concepts developed in the previous one to applications using hierarchical representations of images in the presence of noise.

- **Chapter 7: Noise Level Estimation using Hierarchical Representations.** This chapter proposes the extension of a noise level estimation pipeline for grayscale images to color images. This extension raises the issue of the ordering relationship of multivariate values, which is not natural and tackled by the use of two tools from mathematical morphology. Furthermore, the initial estimation is improved to reach the best results in terms of precision.

- **Chapter 8: Noise Impact on Hierarchical Representations.** This last contribution consists of studying the impact of the noise in an image on the structure of a hierarchical representation built from this noisy image, in this chapter the α-tree. This is performed on a structural attribute, but also in the context of an energy minimization process in order to obtain an optimal hierarchy according to a given functional.

Chapter 9: Conclusion and Perspectives This last chapter, divided into two parts concludes the work described all along this document and describes several perspectives for each axis.

Part I

Genericity and Image Processing

Chapter 2

Generic Programming

This chapter gives an overview of generic programming, a methodology to divide a software library into several reusable components. To this aim, this chapter is partitioned into several sections: the first one defines generic programming by studying a concrete function, which suffers from its lack of reusability, and refines its different elements to obtain a generic version. The C++ features for generic programming are then tackled, since it is the main language used throughout this thesis. These features are then compared with the ones from other programming languages supporting genericity. Finally, we discuss concepts, which are families of types with the same interface and respecting some axioms which establish a predefined behavior.

2.1 From concreteness to genericity

2.1.1 Algorithms and programming

Nowadays, computer programs form the basis of modern society. Whatever the field, be it finance, medicine, science, or leisure, not being comfortable with them may become a major issue. These programs are designed and created by programmers, and their fundamental elements are algorithms.

Definition 1 *[44] An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.*

In the first volume of his famous series *The Art of Computer Programming* [108], Knuth makes the analogy between algorithms and cooking recipes. The input

Listing 2.1: Concrete sum function C++

```
int sum(const std::vector<int>& vec) {
    int res = 0;
    for (int i = 0; i < vec.size(); i++)
        res += e;
    return res;
}
```

data are the ingredients of the recipe and the output data is the cooking dish. Thus, the computational procedure is the ordered sequence of steps required to obtain the dish.

The process of translating an algorithm into a given programming language is called *algorithm implementation*. The resulting source code, which is part of a program, is then transformed during the compilation process to result in machine code directly understandable by the hardware during execution, or interpreted by another program. An example of algorithm implementation in the C++ language is given in Listing 2.1. This algorithm takes as input a collection of elements and outputs the sum of all the elements. It is implemented as a *function*, a programming language feature inspired by mathematical functions, taking as an argument a vector of integer values and returning an integer value.

An *object* is an entity manipulated by a program storing some values in memory and a *type* represents an abstract set of objects [167]. Furthermore, a type is endowed with a specific set of operations usable by the object. In Listing 2.1, two types are used: the `int` type is the set of all the integer values supported by the computer, and the `std::vector<int>` type is the set of all the vectors containing objects of type `int`.

The main issue with the implementation of the sum function is its lack of reusability: it can only be used with `std::vector<int>` objects. However, this algorithm can be applied to any numeric value (integers encoded with different amount of bits, floating-point numbers, user-defined implementations of numeric values, ...), but also to any implementation of a collection (linked lists, sets, ...), which is not the case with the current implementation. Thus to use this algorithm on an object representing a set of floating-point values, it is necessary to write a new function. Finally, when developing a library, such functionality should be available for all types, whose amount is potentially infinite. This is not feasible, and every new function may potentially introduce new programming errors, making the library unmaintainable.

Listing 2.2: Generic sum function C++

```cpp
template <typename Container>
typename Container::value_type sum(const Container& cont) {
    typename Container::value_type res{};
    for (auto it = cont.begin(); it != cont.end(); ++it)
        res += *it;
    return res;
}
```

2.1.2 Generic programming

The previously raised issues require a methodology such that an algorithm is implemented only once and accepts objects belonging to a wide variety of types as input data. Generic programming [146] fulfills these requirements.

Definition 2 *Generic programming is a methodology abstracting algorithms and data structures to design reusable components of a software or a library.*

Writing generic algorithms requires defining a common interface for all kinds of objects manipulated by them. This interface is composed of the operations and the other elements abstract enough for the generic algorithm to not rely on the implementation of the object. These are named *concepts* [54].

Definition 3 *[54] A concept is the combination of a set of axioms satisfied by a data type and a set of operations on it.*

Musser *et al.* state that generic programming is *requirements-oriented programming* [143]. After the development of numerous generic libraries in different programming languages [145, 200], the Standard Template Library (STL) [198] has been one of the most advanced generic libraries developed and is still widely used nowadays. It is based on the idea that it is not only a set of data structures or algorithms, but especially a set of requirements [144]. Thus, requirements are at the basis of concepts.

As an example of a generic algorithm, the `sum` function from Listing 2.1 is abstracted to result in its generic version in Listing 2.2. This function takes as input any collection of elements respecting a given set of requirements, as for the type of the objects stored in it. The traversal of the container itself is performed in a generic way using iterators, inspired by pointers from the C programming language [105], allowing to not rely on a specific access means. A more formal description of these concepts is given later in this chapter.

Listing 2.3: Template parameter list C++

```
template <typename T, bool B, template <class> typename C> // (1)
template <typename... Ts>                                  // (2)
```

2.2 Generic programming features from C++

Generic programming relies on a wide variety of features provided by programming languages. In this section, features from the C++ language are studied. This language is chosen as a basis since it is the main language used in this thesis, but also because its generic abilities have been intensively studied and used for various applications.

2.2.1 The C++ programming language

The C++ programming language [203] is widely used in both industry and academia. It supports various programming paradigms including procedural programming, functional programming, and object-oriented programming, useful to perform generic programming. It is a compiled language: the resulting applications are executed directly on the machine, making it ideal for performance-oriented application. It is also a statically typed language: the type of every expression must be known at compile time. All along this thesis, we use the C++20 programming language standard [96].

2.2.2 Template parameter list

Generic programming in C++ relies on a language feature named *template parameter list*.

Definition 4 *A template parameter list is a list of types and constant values to be processed at compile time by a compatible feature from the language. A feature endowed with a template parameter list is called a template entity.*

Template entities are families of entities: they behave as models. Their mechanism is based on monomorphization: for each combination of parameters used and known at compile time, a new version of the entity is generated by the compiler. This version is highly optimized at compile time, enabling it to reach the best performance for the given combination of parameters. However, the downside of such a mechanism is the combinatorial expense when the parameters depend on some runtime information: it requires anticipating all

Listing 2.4: Templated function C++

```
template <typename T, bool B>
T process_generic(T value) {
    T res;
    // Processing using value
    return res;
}
```

the combinations of parameters and instantiating each template entities with them, generating a large amount of machine code. This phenomenon is known as *code bloat*.

Listing 2.3 shows two template parameter lists. They are preceded by the keyword `template` and declared between angle brackets. In (1), the template has three parameters. The first one is a type parameter, declared with the keyword `typename`. It has to be noted that a type parameter may also be declared with the keyword `class`, and that there is no difference between them in the context of a template parameter list. The second parameter is a constant value, known at compile time. Finally, the last parameter is a template template parameter, taking as a parameter a templated type. The list (2) is a variadic list of template parameters. There may have zero or more template parameters of the same kind as the ones described previously.

2.2.3 Templated functions

Functions are the basis of algorithm implementation in C++.

Definition 5 *A C++ function is composed of two main parts: a signature and a body. The function signature is the set composed of the name of the function, its result type and its argument list. A function endowed with a template parameter list is called a templated function.*

It has to be noted that this definition of a C++ function is simplified to only rely on the necessary elements required to define a generic function and use it. Thus, some C++ features are not discussed in this chapter such as the `const` or `volatile` qualifiers for a function.

A mathematical analogy with a templated function is a parameterized function defined by

$$\forall T \in R, f_T(x)$$

where f_T is a family of mathematical functions. When T is set to a given value belonging to a set R, f_T behaves like a factory creating a new function being part of this family. In this new one, T has been replaced by the value it has been set to. Thus, f_T is an abstract model of a function. Applied to C++, a templated function behaves similarly: it is a family of functions, and its instantiation results in a concrete function.

Listing 2.4 illustrates a templated function endowed with two template parameters, a type parameter `T` and value parameter `B`. As it has been noted previously, templated functions represent families of function, and their instantiation results in a concrete function.

Important: All along this thesis, when working with functions, the two terms *argument* and *parameter* are used to designate different elements. A function argument is one of its inputs, which may be known at runtime, whereas a function parameter is an element given as input to the template parameter list and must be known at compile time.

Relationship with polymorphism - The templated function displayed in Listing 2.4 relies on a language mechanism similar to *polymorphism*, and more specifically *parametric polymorphism*. Polymorphism, as opposed to monomorphism, is a mechanism allowing function arguments to have more than one type. This term first appeared in a course given by Strachey in 1967 [202] and is then refined by Cardelli *et al.* [30]. Polymorphism is divided into two major categories, universal and ad-hoc polymorphism, each of them being divided into two subcategories. Universal polymorphic functions accept as an argument a possibly infinite amount of argument types whereas this amount is very limited in the case of ad-hoc polymorphism. Parametric polymorphism belongs to the category of universal polymorphism. However, templated functions are not parametric polymorphic functions in the sense of the definition given by Cardelli *et al.*, stating that the executed code for universal polymorphic functions is strictly the same whatever the input argument type, which is not the case for templated functions, whose code is generated and highly optimized at compile time, and thus differs depending on the input type argument. Furthermore, in the context of generic programming, the code of a generic function may vary according to its input type argument: for example, the sum function from Listing 2.2 can take as an input different kinds of containers, each of them having its own implementation, as noted by Musser *et al.* [143].

2.2.4 Templated data structures

Data structures are composed of objects used to represent complex data. In C++, they are represented as structures or classes and may hold operations to

Listing 2.5: Templated data structure C++

```cpp
template <typename T>
class vector {
// Interface
public:
    // * Type aliases
    using value_type = T;
    using index_type = int;
    // ...
    // * Operations
    vector();
    value_type& operator[](index_type i);
    // ...
// Implementation details
private:
    T* data;
    int size;
    // ...
};
```

manipulate them.

Definition 6 *A C++ class (or structure) is a set of objects and operations, respectively named member variables and member operations, each of them having a given level of visibility. It may also have type aliases and constant values known at compile time. A class (or structure) endowed with a template parameter list is a templated class (or structure).*

Listing 2.5 illustrates a templated class. It is split into two components: the interface and the implementation details. The interface is the set of information visible to the user and required by the generic algorithm to run correctly. Its visibility, in this code sample, is indicated by the `public` keyword. This interface is composed of several elements such as function members or type aliases. A type alias is the definition of a new name for a given type. It is either fixed or depends on some information given by the parameters. In the listing, two types are renamed through type aliases: the type of the index values is fixed to integer and the type of the container element values depends on the type parameter.

In contrast with the interface, the implementation details are the hidden parts of the data structure. They are declared non-visible using the `private` keyword, such that they cannot be used outside the object. However, the implementation

Listing 2.6: Concept definition and its usage C++

```cpp
template <typename C>
concept Container = requires(C c) {
    typename C::iterator;
    typename C::value_type;
    { c.begin() } -> std::same_as<typename C::iterator>;
    { c.end() } -> std::same_as<typename C::iterator>;
}
&& AdditiveMonoid<typename C::value_type>
&& ForwardIterator<typename C::iterator>;

template <typename C> requires Container<C>
typename C::value_type sum(const C& cont) { /* ... */}
```

of the function members from the interface highly relies on them. In the case of the vector structure in the listing, these details are the number of elements in the vector and its implementation is a buffer of data.

2.2.5 Concepts and constraints

As mentioned above, generic algorithms rely on a set of requirements from the input data to be processed. The C++ language does not force these requirements to exist, as is the case for other languages. However, it provides numerous ways to check them on templated functions or data structures, the preferred and more modern practice being the usage of *concepts* presented in 2006 [80] and integrated into the C++ standard in 2020. Note that this term is used for both the set of requirements and the C++ feature and as it denotes related ideas, one theoretical and another applicative, we use the same word for both.

Before the addition of C++ concepts to the standard, some hacks were developed to simulate them [190, 191]. Another strategy to have a similar mechanism is the Static C++ Object-Oriented Paradigm (SCOOP) [29, 74], which combines generic programming with object-oriented programming to create a static interface using the Curiously Recurring Template Pattern idiom (CRTP) [43], enabling the advantages of object-oriented inheritance to have an interface without the usual runtime overhead induced by the usage of C++ virtuals.

Definition 7 *A C++ constraint is a list of requirements on a data structure. It is composed of several language features such as type aliases or function members, but may*

Listing 2.7: Template specialization example C++

```cpp
struct unknown {};
struct fast {};
struct slow {};

template <typename T>
struct trait { using type = unknown; };

template <>
struct trait<uint8_t> { using type = fast; };

template <>
struct trait<double> { using type = slow; };
```

also be composed of some other constraints. A concept is a named constraint.

Listing 2.6 shows the definition and the usage of a standardized concept. It is a template entity defined by way of the `concept` keyword, and it enumerates a set of constraints specified using the `requires` keyword. Furthermore, this concept also checks some requirements on its type aliases by using other concepts, as is the case by respectively checking the `value_type` and `iterator` with the `AdditiveMonoid` and `ForwardIterator` concepts. Finally, there are different syntaxes in C++ to check if a template parameter respects a concept, such as the use of a `requires` after the template parameter list declaration as illustrated in the listing to check the container type.

2.2.6 Specialization and static dispatch

Static languages have the advantage of optimizing generated machine code at compile time when enough information is known [77, 99]. This has the effect of making algorithms run faster. Furthermore, using generic algorithms on some data structures may have the effect of slowing down their execution. In this case, it is better to have a specialized algorithm for a specific data structure to take advantage of its properties.

C++ provides mechanisms to specialize template entities for a given set of parameters: this is called *template specialization*. Template specialization may be full when all parameters are set to a specific value, or partial when only a subset of parameters is fixed. Thus, templated functions may be specialized for specific types to provide an optimized implementation of an algorithm for a given data

Listing 2.8: Different possibilities to perform static dispatch C++

```
/* (1) Tag dispatching */
template <typename T>
void dispatch(T v, tag1) { /* ... */ }
template <typename T>
void dispatch(T v, tag2) { /* ... */ }
template <typename T>
void perform_dispatch(T v) { dispatch(v, choice_tag_t<T>{}); }

/* (2) Concepts */
template <typename T> requires Cond1<T>
void dispatch(T v) { /* ... */ }
template <typename T> requires Cond2<T>
void dispatch(T v) { /* ... */ }
```

structure, which may be different from the generic version. Another use case of template specialization is illustrated in Listing 2.7, which is the specialization of a data structure `trait` defining a type alias `type` according to a given template parameter. Such a structure is a well-known tool for generic programming in C++ called a *type trait*. Type traits define information about a type in a given context. For example, in the C++ standard library, several type traits are defined such as the `numeric_limits` providing constant values including the minimum or the maximum values of a numeric type.

Finally, template entities may be specialized for a group of types and not only one type. Furthermore, the choice of the correct algorithm may be decided by a given set of attributes related to the input type. We name this choice *static dispatch*. There are several ways to perform static dispatch in the context of algorithms. Two of them are presented in Listing 2.8. The first one is called *tag dispatching*. It consists of choosing the correct algorithm to execute using a tag related to the input type as is the case in the listing with `choice_tag_t`, which is a type alias to the tag contained in a type trait. It is based on overloading, which is a kind of polymorphism where a function has a single name but different implementations, and whose dispatch is performed according to the type of its arguments. The second means of dispatch relies on C++ concepts and constraints. In this case, the choice is performed by using the function whose concept is the best fitting one related to the template parameters. The usage of concepts is the simplest mean to perform static dispatch, but also the most readable one. There exist other methods for static dispatch such as the usage of the `enable_if` type trait, based on SFINAE (Substitution Failure Is Not An

Listing 2.9: Ocaml parametric function and its usage OCaml (REPL)

```
# let first l = match l with              (* Function declaration *)
    | e::q -> e
    | []   -> raise (invalid_arg "Empty list");;
val first : 'a list -> 'a = <fun>          (* Function signature *)
# let l1 = [1; 2; 3];;                    (* List declaration *)
val l1 : int list = [1; 2; 3]
# first l1;;                              (* Application *)
- : int = 1
```

Error) [225], but its usage is more complicated than the two previous ones, and it results in code difficult to read compared to the usage of concepts.

2.3 Comparison with other languages

In the previous section, generic features from the C++ programming language were studied. However, there exists plenty of different means to achieve genericity, depending on the programming language. In this section, some other methods to write generic algorithms are presented and compared with the features from the C++ language.

2.3.1 Parameterization

In C++, generic programming relies on template parameters. This language is not the only one to use parameters. These parameters are either explicit, meaning that they are explicitly declared in a list, as is the case for C++, Java [7], Ada [10], C# [89], Swift [94], Rust [107], and many others, or implicit, signifying that the parameters are not written explicitly but are deduced by the compiler during the definition of an object and inferred at usage to have the correct type. Implicit parameters are notably used in languages from the ML family such as Standard ML [139] or OCaml [140]. Finally, some languages such as Haskell [131] provide function parameters which are either implicit or explicit.

Explicit parameters are discussed widely in the previous section on the C++ languages with the template parameter lists. As for implicit ones, an illustration is given in Listing 2.9. In this listing, a function `first` is defined in an OCaml interpreter. It takes as input a list and returns the first element of this list. In the signature, the input type is `'a list` and the return type is `'a`. The `'a` type is the implicit parameter and its type is deduced when the function is executed using a mechanism named *type inference* [138], which determines the type of an expression based on the type of its subexpressions. This is illustrated when applying the function on the list of integer values `l1`. Type inference is also used in C++ when using the `auto` keyword when declaring a function or a variable.

Dynamically typed languages perform type checking at runtime. However, they can mix dynamic and static typing through for example gradual typing [192, 193]. Gradual typing is based on type annotations, which are indications about the type of expressions in the code that are checked at compile time. Python provides type annotations, but they are only used for static analysis and have no effect at compile time with classical interpreters such as CPython [68]. In order to provide generic parameters, it relies on type variables [177]. The Mypy [117] static type checker for Python programs is inspired by gradual typing and is used along with Python type annotation. Furthermore, type annotations are not limited to gradual typing but may be used for different purposes. In Julia [20] or Common Lisp [217], they are used to strictly check dynamic types. They may also be used in Common Lisp to remove such a checking, resulting in efficient machine code with weak typing.

2.3.2 Generic modules

Modules originate from the Modula programming language [235]. A module is a programming feature designed to organize functionalities developed in software or libraries. It is a set of declarations of several features from a language such as data structures, type aliases, or functions. It is usually divided into different parts: an interface, which defines all the required declarations, and their implementation. A module may be parameterized, thus improving its flexibility and making them generic. It is similar to a C++ class by its definition, however, unlike C++ classes, they cannot be instantiated.

A generic module in Ada, named a generic package, and its usage in a function are displayed in Listing 2.10. In Ada, the interface of a module is named the package *signature* and its implementation is the package *body*. For readability, only the package signature is displayed. This package is parameterized by a generic parameter `T` in a generic part declared with the `generic` keyword. Then several elements are declared such as the provided type and operations. They aim to create and manipulate an optional value that is an object which may not have any value. The data structure `Option` storing this value is then declared in the last part of the package. Its fields are not visible to the user due to the `private` keyword preceding the declaration of the `Option` record, but the type itself is still accessible as it is declared in the previous part of the package. This package is used in the listing in a function by defining a new package and setting

Listing 2.10: Generic package signature Ada

```ada
-- Package signature
generic
    type T is private;
package Optional is
    type Option is limited private;

    function Create_Empty return Option;
    function Create_Value(E : T) return Option;

    function Is_Empty(O : Option) return Boolean;
    function Get(O : Option) return T;
    procedure Set(O : in out Option; E: T);
private
    type Option is record
        Value : T;
        Empty : Boolean;
    end record;
end Optional;

-- Usage in a function
procedure SomeFunction is
    package Optional_Int is new Optional(T => Integer);
    value : Optional_Int.Option;
begin
    value := Optional_Int.Create_Empty;
    -- ...
end SomeFunction;
```

the parameter `T` to the `Integer` type. Then, the type and operations of this new package are used to define a new variable and initialize it.

Ada's packages are not the only kind of generic module. Languages from the ML family provide an advanced module system [130, 218] composed of module signatures, module structures, and functors. The last elements of these components, the functors, are mappings from ML module structures to another module structure used to define generic modules. The CLU language uses a mechanism similar to the classes in C++ for its modules. These are named *clusters* [126] and can be parameterized with a type. They are composed of a list of operations that are visible or not, and an internal data representation that is hidden from the user.

2.3.3 Requirements checking

Requirements checking is a language mechanism used to check if one or more parameters respect a list of requirements. Such a mechanism is studied for C++ in section 2.2.5. C++ concepts check the behavior of a type according to a list of requirements. Thus, type validity entirely depends on its structure, and any type fulfilling these requirements is accepted as a valid type. Such a type system is named a *structural type system* [167]. In addition to C++, several other languages implement structural typing. The CLU language allows putting some constraint in a `where` clause to restrain the type parameter to a specific set of operations. Other examples of structural type checking are the usage of *interfaces* in Go [59] or Ada constraints in the `generic` section of the package signature.

As opposed to structural type systems, *nominal type systems* [167] are based on names and control the validity of a parameter by paying attention to the names related to a given type. In Julia, types are organized as a hierarchy. This hierarchy, represented as a tree, has the `Any` abstract type as a root and concrete types as leaves. An abstract type is a category of type that cannot be instantiated and has no field in its declaration. Such types represent internal nodes in the hierarchy and are used to constrain the type parameter, which must be a concrete type, to have as an ancestor a particular abstract type. Swift uses a more sophisticated system called *protocols*, which is a list of required language features related to a name that must be implemented in the object. The class implementing these features must be related to this protocol and when writing a generic function, the parameter must be bound to the protocol for its described features to be used in the generic function. Similar mechanisms exist in Java and C# under the name of *interface*, or in Rust as *traits*, illustrated in Listing 2.11.

This listing is divided into three parts, illustrating the usage of Rust traits in the context of generic programming. The first part defines a trait with the `trait` keyword, containing a function and a type alias. Indeed, this trait originates from the Rust standard library to implement the + operator [219]. The second part of the listing is the implementation of the trait for a given type, starting with the keyword `impl`. Compared to some languages such as Java or C#, the implementation of traits is distinct from the implementation of the types and of the other traits. This allows to extend existing data structures from existing libraries with traits created by the user, and thus improves the generic capabilities in terms of flexibility. Finally, the generic `sum` function from Listing 2.2 is implemented in Rust. The parameter `C` and its type alias are bounded to some traits in a `where` clause, ensuring the validity of the input parameter by making it implement the required functionalities.

Listing 2.11: Trait bounds and their usage Rust

```rust
trait Add<Rhs = Self> {
    type Output;
    fn add(self, other: Rhs) -> Self::Output;
}

impl Add for SomeType {
    type Output = SomeType;
    fn add(self, other: Rhs) -> Self::Output { /* ... */ }
}

fn sum<C>(cont: C) -> C::Item
where
    C: IntoIterator,
    C::Item: Add<Output = C::Item> + Default,
{
    let mut res = C::Item::default();
    let mut it = cont.into_iter();
    while let Some(e) = it.next() {
        res = res + e;
    }
    return res;
}
```

2.3.4 Discussion

This section discusses generic features provided by other programming languages. These have advantages, but also drawbacks when compared to C++ generic features. Data abstraction is of prime importance in generic programming. Modules and classes have different properties and represent the main features to represent abstract data in this chapter. Depending on the programming languages, modules and classes can be endowed with parameters, this improving code reusability. Even if they are similar, they have major differences: a class is intended to represent an entity with a given behavior while a module is a set of different language features such as functions, data structures, constants, etc... Thus, classes can be instantiated to result in objects whereas modules cannot. Another difference between them is the fact that classes come from object-oriented programming: they benefit from features such as inheritance, which can be useful for many situations. For example, class implementations in programming languages relying on monomorphization may be divided into two classes: a base class that is not parameterized and then inherits to another class that is parameterized. Thus, the code which does not depends on any parameter is just compiled once.

Finally, requirements checking is divided into two categories: nominal and structural. A structural type system checks if a type respects some requirements whereas a nominal type system requires a type to belong to a named category. In some languages, the nominal type system checks if all the operations related to a name are implemented, and does not accept other operations to be used. This is not the case for a structural type system, which just checks if some operations exist, but does not restrict the usage of other ones. Thus, nominal type systems are safer due to their restrictions but are not as flexible as structural type systems.

2.4 Concepts

In section 2.1.2, concepts are defined theoretically as a set of operations and axioms on them. Such concepts are then applied to programming languages in sections 2.2.5 and 2.3.3. However, they are not studied in depth in the previous sections. So, in this section, we define some basic concepts, used in a wide variety of applications. To this aim, we take as a baseline the named requirements from the STL [198] and the concepts described in *Elements of Programming* [199].

2.4.1 Basic concepts

In this part, some basic concepts are considered. They rely on a few operations. Some of them are illustrated in Figure 2.1. These diagrams are similar to UML diagrams. They describe the required interface and the list of axioms for a type respecting a given concept. The name of the concept is endowed with a type parameter, a similar notation for the C++ concepts. So the concept Defaultable<T> means *all type T respecting the requirements of the concept Defaultable*.

This Defaultable concept allows defining a default value for a type. It has been used in the listings illustrating the `sum` function in C++ by the use of the default constructor or in Rust with the `Default` trait. The second concept is the Copyable one. It requires defining an operation such as the C++ copy constructor that copies the value of an object stored in a given memory location into another memory location. The Convertible one is an interesting concept in the sense that it requires a conversion operation from one type into another one. This conversion may be implicit, as it is the case for example in a function using coercion polymorphism, or explicit, using a specific function.

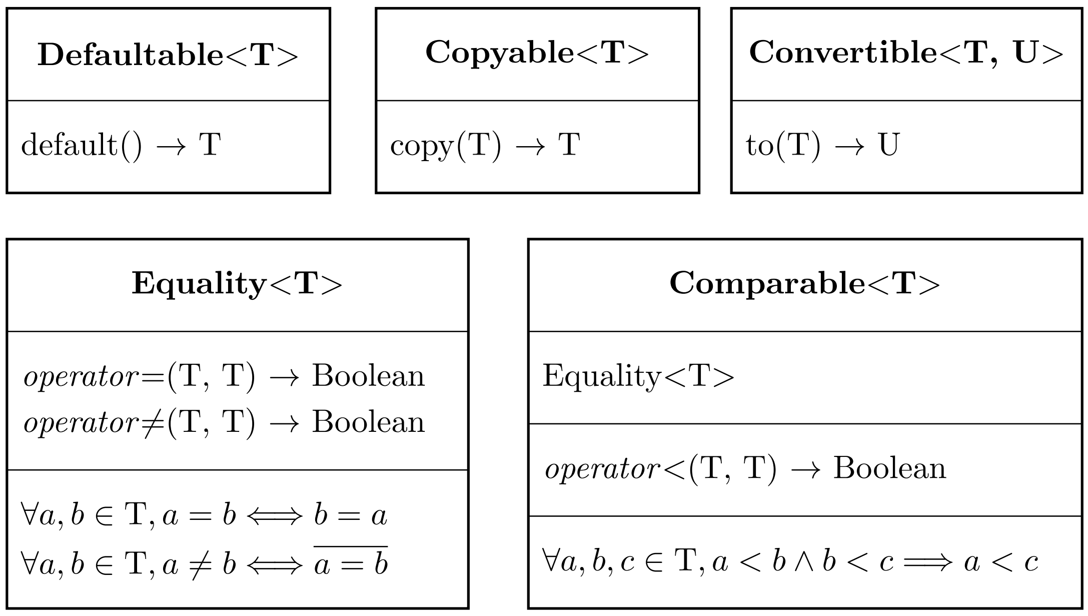

Defaultable<T>
default() → T
Copyable<T>
copy(T) → T
Convertible<T, U>
to(T) → U
Equality<T>
$operator{=}(T, T) \rightarrow$ Boolean
$operator{\neq}(T, T) \rightarrow$ Boolean
$\forall a, b \in T, a = b \Longleftrightarrow b = a$
$\forall a, b \in T, a \neq b \Longleftrightarrow \overline{a = b}$
Comparable<T>
Equality<T>
$operator{<}(T, T) \rightarrow$ Boolean
$\forall a, b, c \in T, a < b \wedge b < c \Longrightarrow a < c$

Figure 2.1: Basic concepts definitions

The two last concepts of Figure 2.1 are the only ones in this figure to define some mathematical axioms. For ease of readability, only a subset of axioms is illustrated as an example for these concepts. The first concept is the Equality concept, defining two operations in the first part to denote if two elements are the same or not, and two axioms, the first one being the symmetric property of equality and the second one defining the inequality by a logical relationship. The second concept is the Comparable one. It defines a total ordering by first extending the Equality concept, then defining an operation stating if an object is strictly lower than another of the same type, and finally defining a transitive axiom.

There exists many useful concepts which are not described in depth. Among them, a Predicate concept can be cited, defining an operation returning a boolean value related to its input arguments. In a more general context, the Callable concept is defined to check if a given type can be used as a function. Such types respecting this concept in C++ are lambda functions, functors, or function pointers. Argument types and result type may be defined inside this concept or in another one.

2.4.2 Concept refinement

In the previous section, some basic concepts have been described. These concepts usually define a few operations. By combining these concepts, it

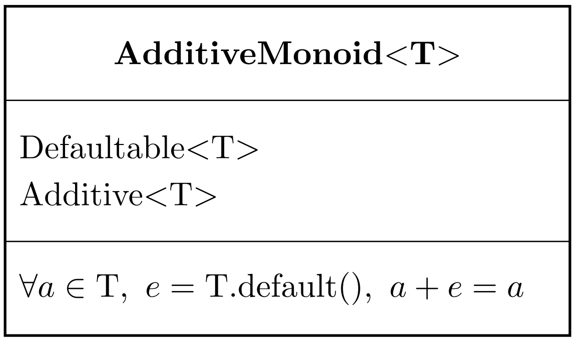

AdditiveMonoid<T>
Defaultable<T>
Additive<T>
$\forall a \in \mathrm{T},\ e = \mathrm{T.default}(),\ a + e = a$

Figure 2.2: Additive Monoid concept

becomes easier to write complex concepts. Thus, designing basic concepts is a mean to increase their reusability. Concept refinement relies on the extension of a concept. In some ways, it is similar to inheritance in object-oriented programming: its aim is to extend a concept, by adding operations or axioms.

An example of concept refinement is illustrated in Figure 2.2. This concept is used previously in this chapter as the concept modeling the family of accepted types for the values of the elements stored in a container given as input to the `sum` function, but it has not been yet clearly defined. It represents an additive monoid $(\mathrm{T}, e, +)$ whose element set is all the possible values from a type T, e is the neutral element, belonging to T, and it is endowed by the addition operator. To this aim, this concept is composed of two basic concepts, the Defaultable, defined in the previous section, and the Additive concepts, requiring the addition operation and stating as an axiom its associativity. However, these concepts are not enough to define completely an additive monoid: even though they bring all the required operations, there is no guarantee the default value is a neutral element for any type. To overcome this issue, the two previous concepts are refined with a new axiom defining the identity element axioms.

2.4.3 Container-related concepts

Concepts refinement is a widely used methodology in practice. In the STL, it is used in several entities and notably the containers. A container is a collection of objects stored in a memory location managed by the container itself. In modern C++ (C++11 and beyond), containers in the STL are divided into three categories: *sequence* containers, *associative* containers, and *unordered associative* containers. Sequence containers store their elements linearly unlike associative containers storing their elements in ordered data structures, usually balanced trees, or unordered associative containers which store their elements in hash tables. In addition to these categories of containers, represented by a concept, an

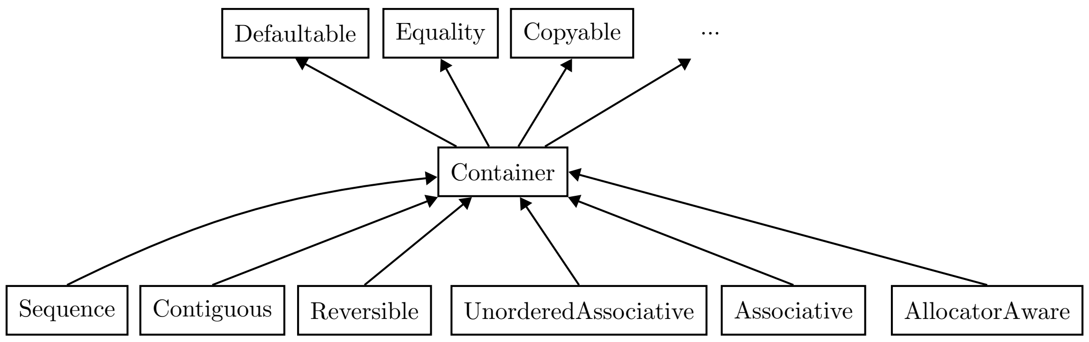


Figure 2.3: Container concept refinement hierarchy

STL container may fit different concepts, illustrated in Figure 2.3 as a hierarchy of refinement. For example, a Sequence container can also be a Contiguous container if its elements are stored contiguously in memory, as is the case for `std::vector` from the STL.

Containers provide numerous means to traverse them according to their use cases and implementations. For example, a vector uses integer-based indexing, a linked list follows the link at each node or a map uses key-based indexing. All these access mechanisms are specialized to one container and restrict the possibilities to implement generic algorithms. Thus, there is a need of abstracting the means to traverse a container. To this aim, *iterators* are generic structures allowing to iterate on all the elements of a container whatever its nature. Depending on the language, there may be different kinds of iterators and refinements, but in most cases, iterators provide two main operations: a read access to the value at a given position of the container and an iteration to its next element. Iterators are used in the `sum` function in Listing 2.2. In this example, an iterator on the first element is created using the `begin` function, which is a requirement of the Container concept. It is traversed from one element to the next one until it reaches a sentinel iterator obtained with the `end` function using the increment operation and its value is accessed by dereferencing it using `*it`.

2.5 Conclusion

In this chapter, we studied different notions about generic programming. We started with the definition of a concrete algorithm, which is not flexible in terms of input object type, and we refined this algorithm by solving the issues raised by the concrete implementation to obtain a modular algorithm by the mean of generic programming. We then used the C++ language as a basis to study generic features in order to write such algorithms. Generic programming being a methodology adopted by a large amount of programming languages nowadays, we compared the generic features from the most used ones with the features provided by the C++ language. Finally, some examples of concepts, which are the basis of abstract modeling in generic programming, are studied.

The notions of this chapter have been widely used since this methodology gained prominence. The examples of the STL and the generic Ada library are the first well-defined generic standard libraries. Furthermore, languages supporting genericity have, for most of them, such kind of generic standard libraries. Moreover, generic libraries are not limited to standard libraries: one can cite scientific computing, such as Eigen [82] a library for linear algebra, graph manipulation with the Boost Graph Library [194], finite state machines manipulation, using the VCSN library [55] or image processing, which is the subject of the next chapter.

Chapter 3

Generic Image Processing and Its Application

In this chapter, the link between generic programming and image processing is established to obtain software tooling that fulfills all the requirements for scientific or industrial research and applications. To this aim, the notion of image is first defined, and several representations are described. The notion of generic programming is then extended to image processing such that algorithms can be applied on the various image representations. Finally, a review of the various suitable tooling is provided, with their advantages and disadvantages, with a special focus on libraries.

3.1 Image representations

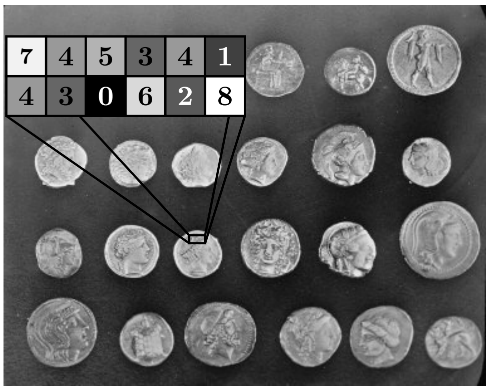
Image processing relies on a wide variety of image representations. To manipulate these representations, it is required to define them properly.

Definition 8 *An image f is defined as the function*

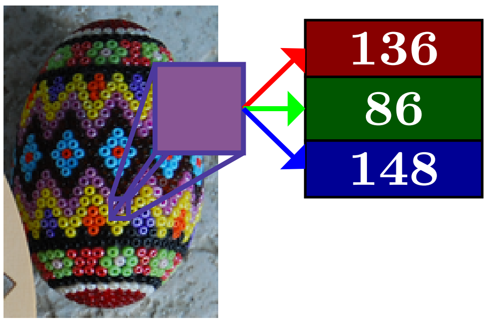
$$f : \Omega \to \mathcal{V} \tag{3.1}$$

where Ω is the definition domain of f and $\mathcal{V}$ is its set of values. Elements of Ω are called points of the image and a pair $(p, v) \in \Omega \times \mathcal{V}$ is called a pixel of the image.

Based on this definition, a brief overview of the different image representations is presented in this section.



(a) Grayscale image



(b) Color image

Figure 3.1: Image defined as a rectangular 2D buffer

3.1.1 N-dimensional images

Classical representations of images discussed in this section are images defined as n-dimensional matrices. They are the most usual representation of images, particularly in the case with $n = 2$. These images, named 2D images, are defined on a rectangular domain $\Omega \subset \mathbb{Z}^2$ and their points $p = (p_1, p_2)$ are elements of this set. Depending on their implementation, these coordinates may be either the pair (x, y) of position x in the abscissa and y in the ordinate of a rectangular domain or the pair (l, c) of position in a matrix representation of an image such that l is the row and c is the column of the matrix element. Even the origin of the domain may vary: it can be located at the top-left of the image, but some implementations also locate it at the bottom left.

Image values vary according to the context they are acquired and manipulated in. Figure 3.1 illustrates different values for 2D images. In the image in Figure 3.1a, the values stored in the matrix are univariate: they are represented by a single value. Such images are often named *grayscale* images and their value at a given position represents the gray intensity of a pixel. It is common that grayscale values are encoded as unsigned integers on 8 bits such that $\mathcal{V} \subseteq [\![0 - 255]\!]$. However, image values may also be defined on a larger set of values. For example, image labeling may require the ability to have more than 256 possible label values. Furthermore, some applications such as astronomical imaging necessitate floating point encoding to keep all the details from the acquisition and avoid possible degradation due to the quantization that would be performed for data transport.

The human perception is not gray, but based on colors. Colors provide more detailed visual information and are thus widely used in image processing. Color values are encoded as multivariate values, often with three components, but not limited to. Figure 3.1b is an illustration of a color image whose values are encoded as multivariate values. In this case, the RGB color space (standing for Red Green Blue) is used. As indicated by its name, an RGB value encodes three intensities, one for each color channel, and the resulting color is obtained by additive mixing. In addition to RGB, there exists many color spaces, each of them having interesting properties according to the application. For example, the CIELAB color space [93] is designed to represent the colors as they are perceived by humans. It is decomposed into three channels: the $L*$ channel encodes the lightness and the $a*$ and $b*$ channels encode the color. It is particularly useful when computing the distance between two colors, which is close to the human perceptual difference, and is performed by means of the Euclidean distance. Other color spaces widely used, notably in image and video compression, are the color spaces encoding the luminance and the chrominance of a color such as YUV or YCbCr. They are notably used in a subsampling process of the chrominance, the luminance handling most of the values information.

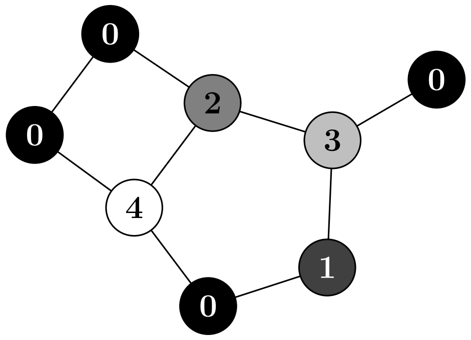
By definition, n-dimensional images are not limited to 2D images, but can have as many dimensions as required. In such cases, the domain is not restrained to a rectangle but is generalized to a hyperrectangle. These kinds of images are used in different contexts: for example, 1D images are used to represent 1D signals such as sound waves, 3D images may be used for medical imaging, biology or to represent 2D+t images, where t is a temporal axis, 4D images can be used to represent a 3D+t image, etc...

3.1.2 Graph-based images

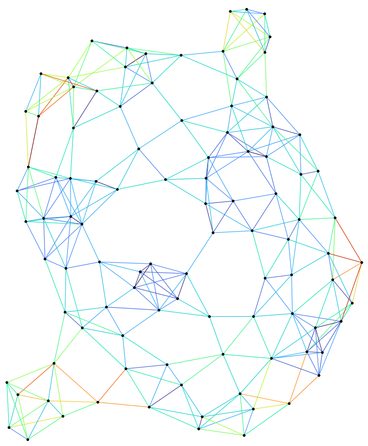
Graphs are useful representations with several applications in image processing [123]. Let $G = (V, E)$ be a graph whose vertices are defined in the set V and edges in the set $E \subseteq V^2$. The graph is either directed, such that for two vertices $u, v \in V$, $(u, v) \neq (v, u)$, or undirected, with $(u, v) = (v, u)$. Let w_v and w_e be two mappings defined by $w_v : V \to \mathcal{V}_v$ and $w_e : E \to \mathcal{V}_e$. These mappings w_v and w_e represent respectively the *vertices weights* and the *edges weights* of a graph. Thus, (G, w_v) is a vertex-weighted graph, (G, w_e) is an edge-weighted graph, and (G, w_v, w_e) is a weighted graph on vertices and edges.

Remark: In this thesis, if not specified, the graph is assumed to be undirected.

Illustrations of these graphs are depicted in Figure 3.2. In this figure, two graphs are shown. The first one, in Figure 3.2a, is weighted on the vertices with grayscale values. The second one, in Figure 3.2b, is weighted on the edges with univariate values. In this figure, the color of the edges represents the edge

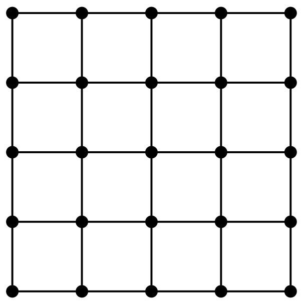


(a) Vertex weighted graph

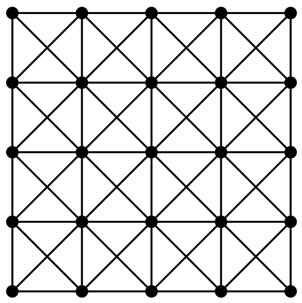


(b) Edge weighted graph (the color of an edge represents its weight)

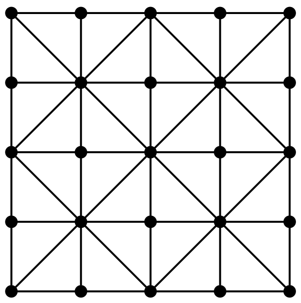
Figure 3.2: Weighted graph representations



(a) 4-adjacency graph



(b) 8-adjacency graph



(c) Khalimsky graph

Figure 3.3: Adjacency graphs for 2D images

weights and are mapped from blue to red.

Graphs can be used to represent some image properties such as the adjacency relationship between two pixels. When working on vertices, two adjacent vertices have a common edge linking them. Similarly, when working on edges, an edge is adjacent to another one if they share a common vertex. Applied to n-dimensional images, it is possible to define the neighborhood of a pixel from an image by means of an *adjacency* graph. Three examples of adjacency graphs are illustrated in Figure 3.3. The ones displayed in Figures 3.3a and 3.3b are derived from the notion of 4-neighborhood and 8-neighborhood from digital topology [109]. In these graphs, a vertex represents a pixel of the image and the edge characterizes the adjacency relationship between two vertices. Similarly, Figure 3.3c illustrates the adjacency relationship in a Khalimsky grid [106].

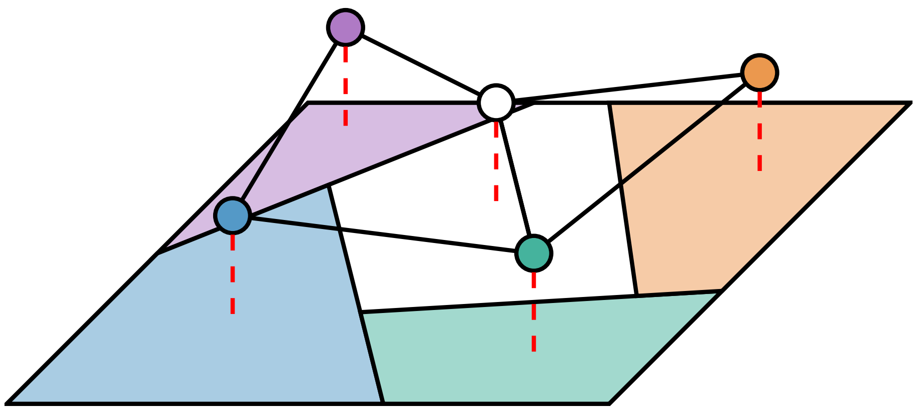


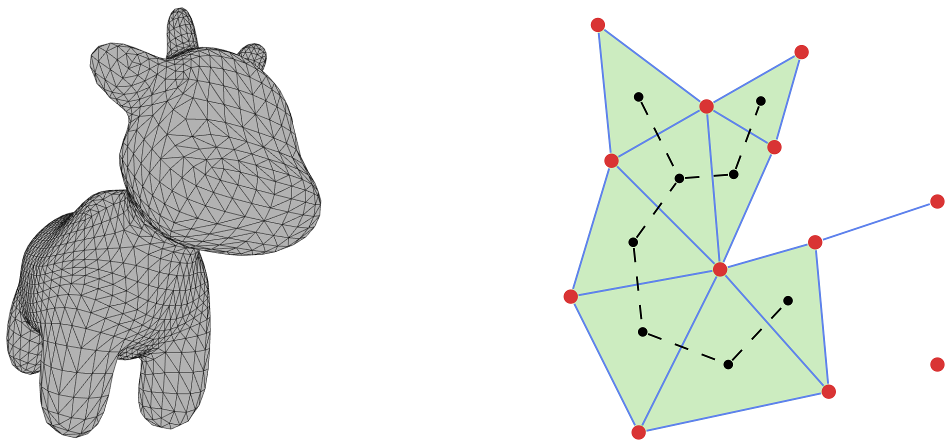
Figure 3.4: Region Adjacency Graph of a partition

An image can be divided into n disjoint regions R_i composed of connected points of the domain Ω according to an adjacency relationship. A partition P is defined as the set $P = \{R_i \mid i \in [\![0..n-1]\!]\}$ of the disjoint regions of an image such that $\cup_{i<n} R_i = \Omega$. Thus, an image can be defined directly on the regions of the partition instead of the elementwise pixels, and graphs may be used on these regions to create an adjacency relationship between them. Such graphs are called Region Adjacency Graphs (RAG) and an illustration is displayed in Figure 3.4, where a RAG is built on a partition composed of 5 regions, each of them being differentiated by a given color. Such graphs are used for different applications such as segmentation [220] or image retrieval [41].

3.1.3 Simplicial complexes

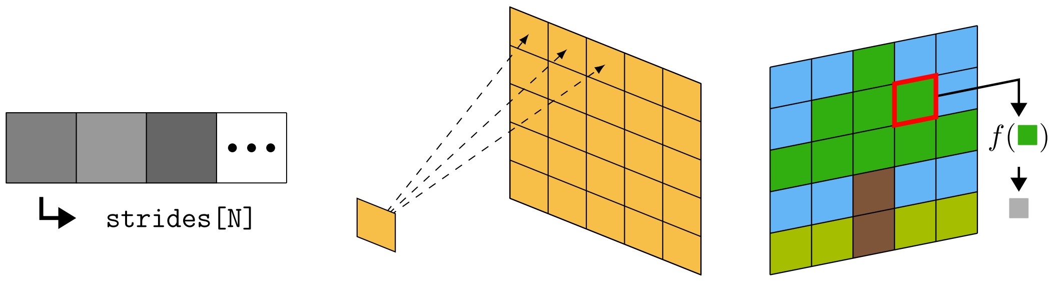
Simplicial complexes are usually used to represent polygon meshes whose faces are triangles, as illustrated in Figure 3.5a. A simplicial n-complex is a set of simplices, which are themselves sets of points whose cardinality is between 1 and $n+1$. A simplex (singular of simplices) with a cardinality i is called a $(i-1)$-simplex. The example in Figure 3.5b illustrates these definitions. It represents a simplicial 2-complex composed of 0-simplices, which are points (in red), 1-simplices, which are lines (in blue), and 2-simplices, which are triangles (in green). It has to be noted that from these definitions, simplicial complexes may be seen as a generalization of graphs. Indeed, a graph is a simplicial 1-complex: its vertices are the 0-simplices and its edges are 1-simplices.

The adjacency relationship between elements of a simplicial complex can be retrieved by defining an adjacency graph using n-simplices as vertices and $(n-1)$-simplices as edges. Thus, if the simplicial complex is of dimension 1, that is if it is a graph, the definition is the same as the one defined in section 3.1.2.



(a) A polygon mesh (b) A simplicial complex

Figure 3.5: A polygon mesh represented as a simplicial complex



(a) Buffer image (b) Constant image (c) View image

Figure 3.6: Different implementations of a 2D image

If it is a simplicial 2-complex, vertices are triangles and edges are the common line between two triangles, as illustrated in Figure 3.5b with the graph whose vertices are black points and edges are dashed black lines.

3.1.4 Implementations

A last important part of image representations is their implementation. In the previous sections, the two main parts of an image given in Definition 8 were discussed: its domain and its value set. To avoid memory space waste or to increase the performance of memory access, an image may have different implementations.

Three implementations are given as examples in Figure 3.6. All these structures are represented in order to be used for 2D images but their usage with weighted graphs or simplicial complexes is valid. The first way to store values in an image is the usage of a buffer, which is a sequential structure that can be accessed in constant time, as illustrated in Figure 3.6a. In this case, all the values are stored in a buffer that may be contiguous, meaning that there is no unused space between two elements of the buffer. This property makes the sequential traversal of an image faster due to the caching of the processor, which reduces the access time to the data. To access the values of images represented by a buffer, two pieces of information are important: the coordinates encoding a point $p \in \Omega$ and the strides s. A stride is a value encoding the number of elements in the buffer to step over in order to access the next element for a given dimension. In that way, a pixel value located at position p is accessed by means of an offset from the beginning of the buffer computed by $\langle p, s \rangle$ with $\langle \cdot, \cdot \rangle$ being the inner product. Therefore, this access mean is not limited to 2D images and can be used to access a pixel value of any n-dimensional image. The order in which the values are stored constiguously in the buffer may differ. A buffer is considered C-contiguous when transitioning to the next element results in a change in the last coordinate first. An F-contiguous buffer is the opposite of a C-contiguous buffer, that means the first coordinate is first changed. This ordering of values may be handled by the strides of a buffer.

However, this representation has the disadvantage to take $|\Omega| \times \text{sizeof}(v)$ bytes in memory for $v \in \mathcal{V}$ a pixel value and $\text{sizeof}(v)$ the operation returning the number of bytes encoding v. This is necessary in some situation such as when the image is writable at any position or when all the elements differ too much for optimization to be performed. When it is not necessary, different implementations can be used. The first case is the situation where only one element is stored. If the writable property of an image is not required, the image can be stored as only one value, and this value is accessed at any point of the image, as illustrated in Figure 3.6b. Another implementation when the image contains a few elements different from a particular one is the combination of a constant image with a sparse matrix, a data structure having different implementations but whose objective is to only store values different from 0.

A last example of image implementation is the *view* [179], named from C++ views. Views are non-owning images: they point to an image but may modify some of its properties. For example, the domain may be a subset $D \subseteq \Omega$ of the image domain Ω. In that case, the view plays the role of the image with a different domain, but the data are owned by the original image. Such view is called a *slicing* view. Another example of view is a *transform* view: it performs a lazy, on-demand transformation of the image. This is illustrated in Figure 3.6c, in which the image is an RGB image, but at read access, the returned value is the one from the green channel. In that case, the view may be seen as the image composed of the green channel values. Another interesting property of

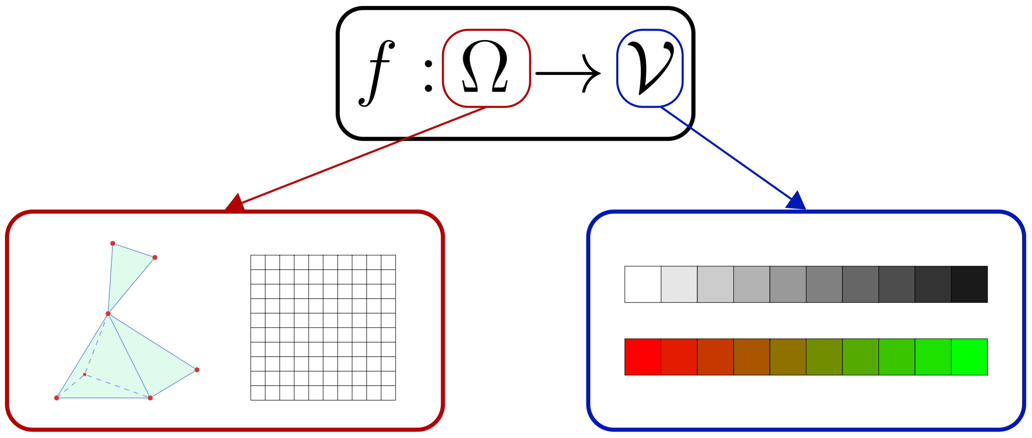


Figure 3.7: Generic image representation

view images is their composability: a view may be composed of several views: for example, if one wants the value of the green channel of an RGB image at a particular region of interest, it can use the two previously explained views.

3.2 Generic image processing

In the previous section, an overview of different image representations was given. These images share common properties allowing to write generic algorithms for them. This section summarizes the idea behind a large amount of work that has been performed about generic image processing [110, 118, 71, 178].

3.2.1 Image abstraction

Definition 8 states that an image can be represented as a function taking as input a value from a domain Ω and returning a value from a set $\mathcal{V}$. Based on such a definition, it is desirable to use any structure as a domain and to value each of its elements. This is illustrated in Figure 3.7. The domain may be composed of any element of a simplicial complex such as a point, a line, or a triangle, but also any kind of grid (rectangular, hexagonal, etc...). Values may also vary depending on the use case of the application or the input format.

Applied to generic programming, it is necessary to define a concept to be able to manipulate the images. A simplified, yet practicable, `Image` concept is illustrated in Figure 3.8 which is similar to the concept presented in [122]. This concept first defines three type aliases: a type for the points, which should be the element type of the domain on which the image is defined, a type for the

Image<I>

type *point_type*
type *value_type*
type *pixel_type*

domain() $\rightarrow$ range<point_type>
values() $\rightarrow$ range<value_type>
pixels() $\rightarrow$ range<pixel_type>
operator()(point_type) $\rightarrow$ value_type

Figure 3.8: Image concept

values of the image, and a last type for the pairs (point, value). This last element is very important when combining genericity and performance as demonstrated in [121]. Then, four operations for these types: the `domain`, `values`, and `pixels` result in a range containing respectively all the points, the values, and the pixels of an image. Depending on the image implementation, they may take care of the memory layout to perform an efficient traversal of the image. The fourth operation represents a random access operation, which may not be available on all kinds of image implementation, but fitting Definition 8 in terms of function representation of an image. More detailed concepts for the image structure are proposed in [180].

Image adjacency relationship is a requirement for a wide variety of algorithms. In the previous section, it is represented as an adjacency graph for the different image representations. However, this representation is too restrictive for all its usage, and a more global definition is required to exploit all its capabilities. To this aim, a *window* is defined as a function taking a point as an argument and resulting in a set of points. It is used to represent the *neighborhood* of a given point of a domain. Thus, it can represent a 4-adjacency relationship by taking a given point as an argument and returning the range containing up to the four neighboring points. Furthermore, such windows can be used to represent *structuring elements* in mathematical morphology, but also convolutional kernels when the points are weighted by a value. This representation is thus flexible enough to be used by generic algorithms.

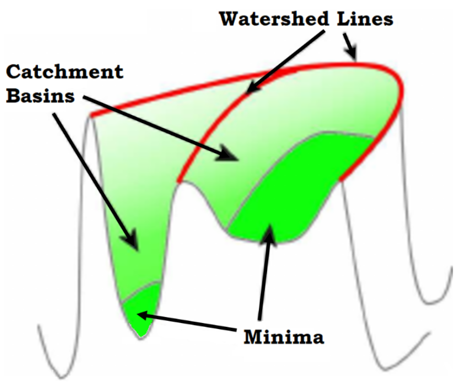


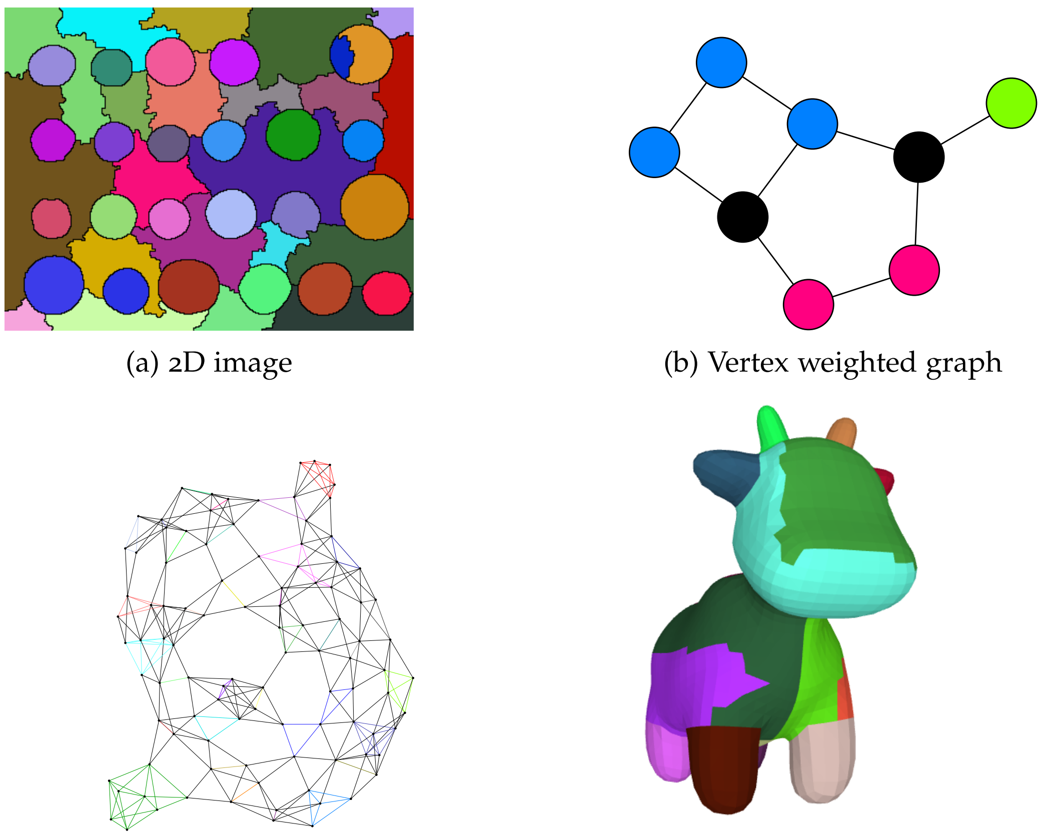
Figure 3.9: Illustration of the watershed segmentation principle (from [8])

3.2.2 Application to image processing algorithms

The usage of generic programming in an image processing framework has been studied in several applications. In [61], a general framework to work with generic mathematical morphology algorithms is proposed. In [119], Levillain *et al.* apply these concepts to a generic image processing library and give an example with a generic watershed algorithm [19, 137] applied to different image domains. In this section, this example illustrates generic image processing algorithms.

The watershed [19] is a marker-based segmentation algorithm considering the image as a topographic relief. Starting from particular regions of the image, which often are its regional minima, the segmentation process goes from pixels with the lowest intensity adjacent to these regions to pixels with the highest intensity. Thus, each marker defines a catchment basin and the watershed algorithm simulates the filling of these basins until two of them meet. In this case, a watershed line can be created at the meeting point. The principle of this algorithm is illustrated in Figure 3.9.

From this definition, the watershed algorithm requires some properties on the image structure and on its values. First, its domain should be endowed with an adjacency relationship. In this case, a window can be used to express the neighborhood. The advantage of using a window independently from the image comes from the fact that a given image structure can define different adjacency relationships, such as illustrated in Figure 3.3 in the case of 2D images. Furthermore, image values need to be totally ordered, implying that $\mathcal{V}$ must be a complete lattice, which is a set endowed with an order relationship. This is a well-known requirement of mathematical morphology [176], the field the watershed algorithm belongs to.


(a) 2D image (b) Vertex weighted graph

(c) Edge weighted graph (d) Simplicial complex

Figure 3.10: Watershed segmentation applied to different kinds of image

Figure 3.10 illustrates the watershed applied to different images respecting the previously listed requirements. In Figure 3.10a, the watershed is computed on the morphological gradient of the image displayed in Figure 3.1a, and the adjacency used is the 8-adjacency relationship displayed in Figure 3.3b. Furthermore, for readability, an opening by area is performed on the gradient image. Figures 3.10b and 3.10c are the results of the watershed computation performed directly on the vertex-weighted graph from Figure 3.2a and the edge-weighted graph from Figure 3.2b using the adjacency relationship described in section 3.1.2. Finally, Figure 3.10d is the result of the watershed on the simplicial complex computed from the mesh in Figure 3.5a. The values used on each triangle are their maximum curvature values [166, 181].

Listing 3.1: Generic morphological operation C++

```cpp
template <typename A, typename I, typename SE>
I morphological_operation(I in, SE se) {
    I out(in.domain());
    for (auto p : in.domain()) {
        A acc;
        acc.take(in(p));
        for (auto q : se(p)) {
            if (in.domain().has(q)) {
                acc.take(in(q));
            }
        }
        out(p) = acc.to_result();
    }
    return out;
}
```

3.2.3 Algorithmic canvas

Generic image processing algorithms are used on different image structures which can have generic value types. Thus, having one implementation for one operation is useful to avoid code duplication. In image processing, some algorithms differ only on a particular operation. In such cases, it is convenient to implement the generic algorithm with generic images, but also generic operations. Such algorithms are called algorithmic canvases [60, 75].

In Listing 3.1, an example of a generic algorithmic canvas is given as a C++ function for the basic morphological operators such as erosion or dilation [188]. The function takes two arguments: an image `in` and a structuring element `se`. Furthermore, the first template parameter, denoted by `A`, is the type of an operation to be performed in the canvas. In the context of this algorithm, the operation takes the form of an accumulator, which is a generic object which can be used to compute statistics in the same way as Boost in its Accumulators library [155]. This morphological operator function creates an output image, then traverses all the pixels of the image and for each of them, it performs an operation on this pixel. This operation consists in consuming all the values at the locations contained in the structuring element window if these points belong to the domain using the `take()` method of the accumulator. Finally, the result of the operation, obtained by the `to_result()` method of the accumulator, is set as the value of the output image at the current location of the input image. Thus, erosion and dilation can be implemented using this canvas just by setting an infimum accumulator (for erosion) or a supremum accumulator (for dilation).

In addition to this one, there exists a wide variety of canvases for mathematical morphology: the union-find [207] canvas, usually used for connected component labeling, can be exploited to implement morphological connected operators [233], component trees [16] or watersheds [135]. Another example of a canvas is the breadth-first thinning [120] from digital topology which results in different outputs related to the constraints given as arguments.

3.3 Image processing libraries

In the previous section, we have discussed the notion of genericity in the context of image processing, and we have given examples of generic algorithms for mathematical morphology. In this section, we list the required criteria an image processing library should follow, and we give an overview of some available libraries. More specifically, we focus on the mathematical morphology capabilities of these libraries, since it is the main field of image processing this thesis is focused on, and we carefully observe how much they respect the listed criteria.

3.3.1 Image processing tooling

Tooling in scientific research is of prime importance. In image processing, there exist several kinds of tools, each of them having its use case. The first category is the set of *utilities and graphical tools*. These tools do not require any programming ability as they are compiled programs to be used as it. Among them, ImageMagick [201] is a good example of a command line tool that enables to perform basic manipulations of an image such as filtering, denoising, or color space conversion. Furthermore, it is able to work with image storage by extracting metadata such as the EXIF of an image, its compression algorithm, or its file format. Graphical tools such as GIMP [209] and ImageJ [25] extend the functionalities of command line tools by providing a graphical user interface (GUI). They provide interactive and ergonomic usage by means of the GUI. Furthermore, they usually allow extending their functionalities by means of an API in order to implement some plugins.

The second category of tools is the *programming environments*. These environments are usually composed of a text editor to program scripts to make some image processing pipeline. Furthermore, they are endowed with some convenient tools such as image viewers, format, and display functionalities for data tables which are 2D tables containing data and whose columns can have different data types, such as the ones provided by Pandas in Python [134]. Application programming interfaces (API) may also be used to extend programming environments and provide user interfaces at the top of the developed scripts. Programming environments are available in different forms: some are complete GUI applications, such as Matlab [95] or Spyder [197], and others take the form of interactive notebooks, such as Pluto.jl [169] or IPython/Jupyter [161, 168].

A third category is *domain-specific languages* (DSL) [69, 76]. Different DSL have been designed for different purposes. For example, Halide [173] has for objective to simplify the implementation of high-performance algorithms on images and arrays. The language is embedded in C++ so it uses some C++ tools, but it builds an abstract syntax tree (AST) from the usage of constructor and operator overloading and either uses JIT compilation at runtime or emits machine code to be linked in a program. The Milena [122] library, which is part of the Olena platform, also provides some kind of DSL built on top of macros to simplify writing generic algorithms and make them more readable. Functionalities provided by this DSL are, for example, generic image traversal, or some aliases on generic traits, whose names, and so understanding, are simplified by their usage.

The last category of tooling is the *libraries*. Libraries provide algorithms and data structures to be manipulated by them. These libraries can either be specific to one subfield of image processing, as is the case of the Morph-M [112] or Higra [164] libraries, or a general purpose as is the case for Scikit-Image [230] or OpenCV [27]. In this thesis, we focus on libraries since it is the most common one and may be used along with the tools belonging to other categories.

3.3.2 Requirements of an image processing library

As stated above, libraries are the most popular tools for image processing. They can take several forms: some are dynamic language packages while others are compiled libraries. Some libraries mix these two forms to create some hybrid packages: a compiled library in some static language such as C++ for performance and a top-level layer implemented in a dynamic language for ergonomic purposes using the functionalities of the compiled library. By doing so, they attempt to reach three important criteria for image processing libraries: genericity, performance, and interactivity.

Genericity - Generic programming has been widely explained since the beginning of this document. Writing generic image processing algorithms is of prime importance as library maintainers do not want to duplicate algorithms implementations for different kinds of types in order to decrease the amount of code to be maintained. Furthermore, image processing is a field that requires flexibility, meaning that it should be able to handle any new type at the lowest cost. This flexibility is justified by the increasing amount of image value types, but also to handle image implementation from other libraries. To this aim, the generic image interface must be well-defined, as explained in the previous section so that the cost to adapt any image implementation to this concept is minimal to be applied to generic algorithms.

Performance - Performance is an important criterion when working with large images. Some processing may take a long time when the image dimensions and the quantization of the values are large. An example of such images is found in the context of astronomy for the Naroo program [174], where the acquired images are quantized on 16 bits with 2560×2160 pixels. Furthermore, image processing algorithms may be applied in the context of real-time applications, and performance is crucial in this situation. Finally, image datasets have become very large to process with the increasing usage of deep neural networks. For example, the ImageNet dataset [56] contains more than 14 million images, and the Celeba dataset [102] which contains 30000 images with a large resolution (1024×1024).

Interactivity - Interactivity is important when experimenting with algorithms. It allows the development and manipulation of new algorithms without having to recompile the whole pipeline. Furthermore, interactivity is of prime importance in the context of image transport, whose value type can be unknown at compile time. It is usually done by means of dynamic languages such as Python, Julia, or Matlab. Nowadays, these languages provide programming environments, which make them widely used for scientific research. Different standards are used to ease the interoperability of different tools: for example, Python users apply image processing algorithms to n-dimensional images by means of Numpy `ndarrays`.

In practice, the three criteria are difficult to combine. For example, in C++, genericity and performance can be used along with templates. Indeed, as a static process, template mechanisms are performed at compile time so that specialized algorithms are optimized for each input parameter. However, such a combination is achieved at the cost of interactivity, which requires waiting runtime to obtain some type information. On the other hand, the usage of dynamic languages allows interactivity and genericity, but suffers in terms of performance from actions made at runtime such as the type checking. This overhead is even greater when the language is interpreted. In the context of interpreted languages, this lack of performance is reduced by means of Just-In-Time (JIT) compilation, but it has several drawbacks. For example, one can cite in Python the PyPy [210] implementation of Python or Numba [115], but they come with some limitations in terms of available features compared to the CPython [68] reference implementation. Another example is the Julia [20] programming language, which succeeds to associate these criteria, but at the cost of a slow compilation when a complex parametric function is first used.

3.3.3 Review of image processing libraries

As mentioned above, the criteria for an image processing library are difficult to reach without some compromise. Thus, in this part, an overview of image processing libraries is given. Precisely, we focus on libraries providing mathematical morphology procedures as it is the main subject of this thesis, and we focus on the evaluation of the adherence to the three criteria listed above.

At first, the libraries for mathematical morphology are studied. These include Mamba-Image [17], Smil [64] and Morph-M [112]. Mamba-Image is a library for prototyping and educational purposes. It is designed in two parts: a core library implemented in C and a top layer in Python. The core library does not provide features, but the operations to implement them. These operations are highly optimized thanks to SIMD (Single Instruction Multiple Data) programming. They are brought to Python using the Swig generator [13], which is widely used to generate interfaces in different scripting languages for scientific libraries [12], and they are used to implement the features. On the other hand, Morph-M is implemented in C++ and provides three layers for mathematical morphology: a *templated layer* for generic processing, an *interface layer*, where the functionalities of the templated layer are instantiated to be used in a non-templated context, and an *interpreted language layer*, which use the interface layer to provide functionalities in Python. Furthermore, this library provides connections with other tools such as Numpy. Finally, the Smil library combines the best of the two previous ones: it aims to provide a generic library optimized by means of parallelized morphological algorithms to be used for real-time application. It also provides an interface to several languages such as Python or Java by means of Swig.

Pink [45] is a general-purpose library, but it has a special focus on mathematical morphology and digital topology. It is implemented in C and provides an interface in Python by means of Boost.Python [2] and C++ code encapsulating the Pink library features and data structures. It relies on a type-erased image structure and dispatches a compatible algorithm implementing a feature of the library to the correct type at runtime. This library can also be seen as a platform: it provides numerous command line utilities, either in C or in Python thanks to its interface.

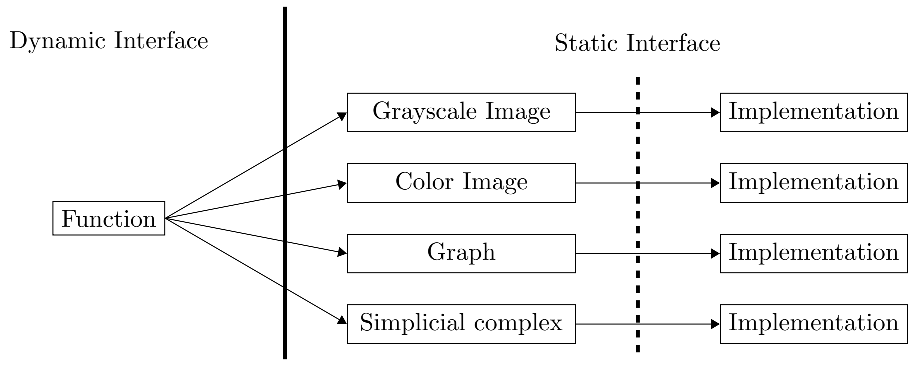
The Scikit-Image library [230] is widely used by scientists for image processing prototyping. It is designed as an extension of Scipy [227], and uses some of its functionalities. For example, the morphological operations such as erosion and dilation in Scikit-Image are the ones used by the `scipy.ndimage` module, with some improvements in terms of interface. For that reason, the Scikit-Image library is discussed taking into account the Scipy module. Its elementary features are implemented in C as extension modules, making this library a hybrid package, but also using the Cython transpiler [14]. A transpiler takes input source code and outputs another source code, either in the same language or not. The Cython transpiler thus takes as input Cython source code, which is a language at the intersection between Python and C, and outputs source code in C containing a Python extension module implemented using the CPython API. The image model of this library is based on Numpy, and genericity is ensured for the value type of the image. Numpy has a C API and brings some compatibility with Cython. Thus, it is well-suited for the development of algorithms in these performance-oriented languages.

The OpenCV library [27] uses type-erased image. Images information such that its dimensions or its value type are known at runtime, and its interface uses dynamic dispatch to use a procedure whose value type is static and thus optimized. More specifically, to reduce the amount of code generated by the compiler when templated entities are specialized, the library holds templated core features such as filters in the context of mathematical morphology which are specialized at dispatch and then run. Its interactivity in Python is performed by generating automatically C++ code for its functionalities and bridging them using the CPython API. Furthermore, as OpenCV relies on a dynamic interface when used in C++, its interface slightly differs from the one in C++, and it accepts Numpy arrays as an image implementation.

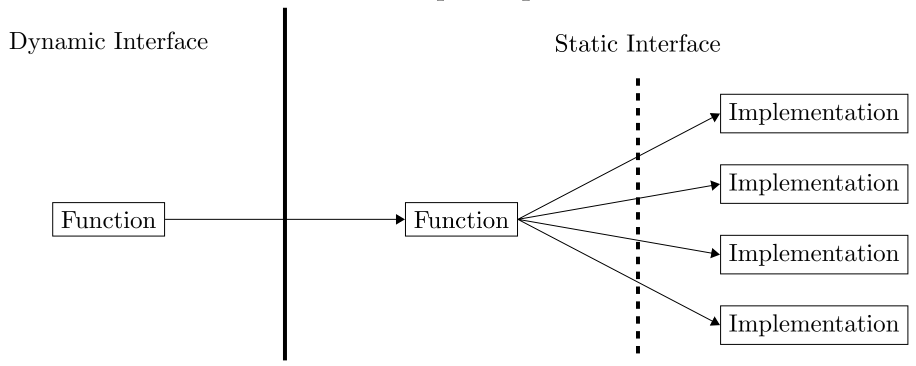
Finally, a category of C++ libraries are fully templated libraries. Among them are Vigra [110], Milena [122], Higra [164], and Pylene [208]. Vigra's objective is to provide a generic library for image processing following the design of the STL [111]. It has many data structures such as images or graphs but also many algorithms to apply to these data structures. It provides a dynamic interface in Python by means of Boost.Python on specialized algorithms, but it restrains the number of available image value types that can be used to limit the amount of generated machine code, and it even extends Numpy array to handle semantics on the values of the images. Milena, on the other hand, is designed as a concept-oriented library [74] to apply algorithms on any kind of image as long as they respect the required interface. Some attempts at designing a dynamic interface for Milena are performed: the first one relies on the specialization of algorithms and image data structure and exposes their functionalities in Python by means of the Swig generator. The second attempt consists in using some JIT

	Core language	Generic values	Generic domain	Interface	Interface bridge
OpenCV [27]	• C++	✓	✗	Dynamic with type erasure	• Python • Java • Javascript • ...
Scikit-Image [230]	• Python • Cython	✓	✗	N/A	N/A
Pink [45]	• C • C++	✓	✗	Static	• Python
Smil [64]	• C++	✓	✓	Static	• Python • Ruby • Octave • Java
Mamba-Image [17]	• C • Python	✓	✗	Dynamic	• Python
Vigra [110]	• C++	✓	✓	Static	• Python • Matlab
Higra [164]	• C++	✓	✓	Static	• Python
Milena [122]	• C++	✓	✓	Static	• Python
Pylene [208]	• C++	✓	✓	Static	• Python

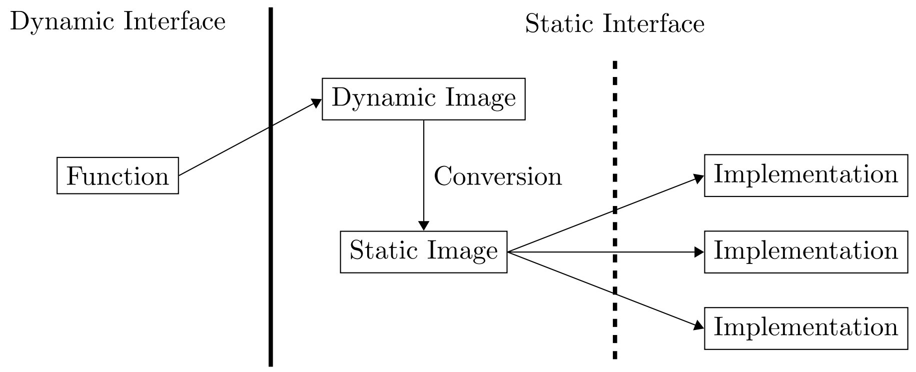
Table 3.1: Comparison of the state-of-the-art libraries



(a) One function per implementation



(b) One function per functionality



(c) Hybrid design for Pylena

Figure 3.11: Illustrations of the different interface bridge possibilities

compilation techniques in order to compile and bridge functionalities in C++ to the dynamic interface on demand, as explained in the perspectives of the Levillain thesis [118].

Higra is a modern C++ library whose aim is to provide functionalities for hierarchical representations construction and manipulation on graphs. It is generic on values by means of templates in C++, but also on the domain as the main manipulated objects are graphs, which can represent the adjacency relationship of the domain of an image or a simplicial complex as explained in sections 3.1.2 and 3.1.3. It is designed to differentiate the value and the domain. As a consequence, the algorithms usually take as arguments at least two objects: the graph and its weights, either on vertices or on edges. Using such a design has one advantage when providing an interactive interface in Python, which is done using the Pybind11 library [97]: the amount of generated code is reduced since only the values are specialized. However, the semantic interpretation provided by image processing libraries is lost since these tables do not make the difference between edges and vertices weights.

Pylene is the successor of Milena. It is specialized in efficient and generic mathematical morphology but is not limited to it. It uses modern C++ features to solve the issue raised by the older standard of C++ (pre-C++11) in the context of the development of the Milena library such as its complex internals or the lack of ergonomy for the users induced by the language [72]. Its objective is to meet the three criteria listed above as closely as possible. In terms of genericity, it makes use of the C++20 concepts to model the image requirements but also to make a hierarchy of refinement in order to handle a wide variety of images [179, 180]. It provides a dynamic interface in Python by means of the Pybind11 library called Pylena, but it has the same limitation as the previously studied libraries.

These libraries are summarized in Table 3.1. For all of them, the language chosen to implement the library is a performance-oriented language. Even Scikit-Image uses the Cython transpiler to take advantage of the compiled nature of the C language or implements some features by means of other libraries such as Scipy or Numpy, which heavily use such languages. This illustrates the significant role of the performance criterion. Furthermore, most of these libraries use languages providing facilities for generic programming, such as C++ or Python. In this context, all the libraries provide generic value capabilities in their algorithms, either in a static way, with a dispatch performed at compile time, or in a dynamic way, in which the correct functionality is chosen at runtime. However, not all of them provide generic domain capabilities, and the libraries succeeding the most to have generic domains by handling various structures are Higra, Milena and Pylene.

The interactivity criterion is achieved by three means for the libraries. These are illustrated in Figure 3.11. The first way to bring a static interface to a dynamic interface, illustrated in Figure 3.11a, is to specialize each implementation of functionality to different image types and let a utility such as Boost.Python or Pybind11 perform the bridge and handle the dispatch to the correct implementation when the function is called from the dynamic interface, hiding this part from the developer of the library. This is the most widespread way of bridging static interface in C++ to a dynamic language, but it requires generating a large amount of code due to monomorphization. Libraries doing so include Higra and Vigra. The second method to make a bridge, illustrated in Figure 3.11b, relies on the existing static interface: only one function is exposed to a dynamic interface. This has several advantages such as the static and the dynamic interface being quite similar or the fact that the dynamic interface can be generated automatically, such as it is the case in OpenCV. However, this lets the library developers handle the dispatch to the correct implementation if there are several ones. Finally, Pylena, the dynamic interface of Pylene, uses an hybrid solution displayed in Figure 3.11c. It has a single entrypoint from the dynamic interface with a function taking a dynamic image obtained from Numpy arrays, and then converts it to the correct static type to perform the dispatching on instantiated templated functions. This solution has the advantage of having one entrypoint per functions, but requires the developpers of the library to handle the dispatch and it generates a large amount of code.

3.4 Conclusion

This chapter has introduced the notion of an image as a function and gives an overview of different structures which comply with this definition, either in terms of structure or value. Then, this relation was used to define a generic image concept, and its usage was demonstrated by applying the notion of a generic image to a generic image processing algorithm from the field of mathematical morphology. Next, various libraries providing mathematical morphology features were examined, with a specific focus on their adherence to the criteria. In the next chapter, hierarchical representations of images will be presented. These ones can be implemented as generic algorithms and used in different applications.

Chapter 4

Hierarchical Representations of Images

In this chapter, hierarchical representations of images are introduced. To this aim, this chapter is decomposed into three parts: first, the notion of hierarchical representation is explained by the usage of partitions and partial partitions, which divide these representations into two categories: inclusion hierarchies and partitioning hierarchies. Then, the representation of hierarchies as trees is discussed, along with their possible implementations. Furthermore, an overview of the different hierarchies used in this document is given. Finally, the possibility to build these hierarchies on different image domains is analysed and an example is given with a hierarchical watershed.

4.1 Representing images as hierarchies

Hierarchical representations of images are widely used tools in image processing. They are reviewed in [24], and divided into two categories: *inclusion hierarchies* and *partitioning* hierarchies. Inclusion hierarchies are useful when working with the regional extrema of an image while partitioning hierarchies allow to describe complex images whose regions of interest are not extremal. Thus, the choice of the hierarchy to be used depends on the application. In this section, the basis of hierarchical representations is presented to explain the division into these two categories.

As explained in the previous chapter, the image domain can be divided into several regions R, which are sets of points from Ω such that for any two regions R_i and R_j, $R_i \cap R_j = \emptyset$. Let $P^\setminus = \{R_i \mid i \in [\![0..n-1]\!]\}$ be a partial partition. This

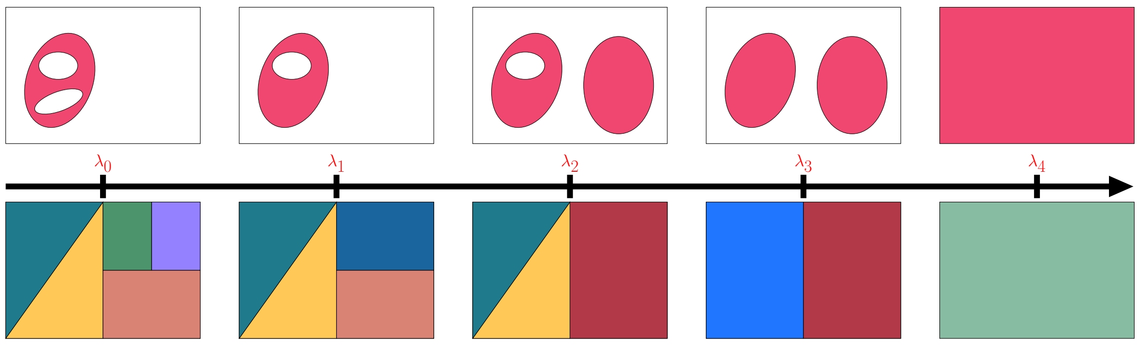


Figure 4.1: Lattice of partial partitions (top) and partitions (bottom)

set of regions forms a partition P if $\cup_i R_i = \Omega$. Let $\mathbf{P} = \{P_0, ..., P_{n-1}\}$ be a set of partitions (or partial partitions) of a domain Ω. Such sets forms a complete lattice [175, 187] $(\mathbf{P}, \leq)$ if for any two partitions (or partial partitions) $P_i, P_j \in \mathbf{P}$, $P_i \leq P_j \iff \forall R_i \in P_i, \exists R_j \in P_j, R_i \subseteq R_j$.

Figure 4.1 illustrates the notion of a lattice of partial partitions, in the top row, and a lattice of partitions, in the bottom row. In this figure, partitions and partial partitions are indexed by a value λ_i such that for all indices λ_i and λ_j, $\lambda_i \leq \lambda_j \iff P_i \leq P_j$. In the figure, the values λ_i are sorted from left to right. In its first row, regions belonging to each partial partition are colored in red. From the partial partition indexed at λ_0 to λ_4, new regions appear, until the whole domain is covered. In the case of partitions, the whole domain is covered, but regions are merged to form coarser regions until the partition indexed at λ_4 contains only one region R_0 such that $R_0 = \Omega$.

Thus, with the notion of partitions, partial partitions, and their lattices, the two categories of hierarchical representations are described:

- **Inclusion hierarchies -** Inclusion hierarchies are ordered stacks of partial partitions such that the smallest part of the domain is covered by the partial partition at the bottom of the hierarchy while the whole domain is covered by the one at the top of the hierarchy.

- **Partitioning hierarchies -** Partitioning hierarchies are ordered stacks of partitions such that fine regions are represented by the smallest partitions while the partition at the top of the hierarchy contains only one region covering the whole domain of the image.

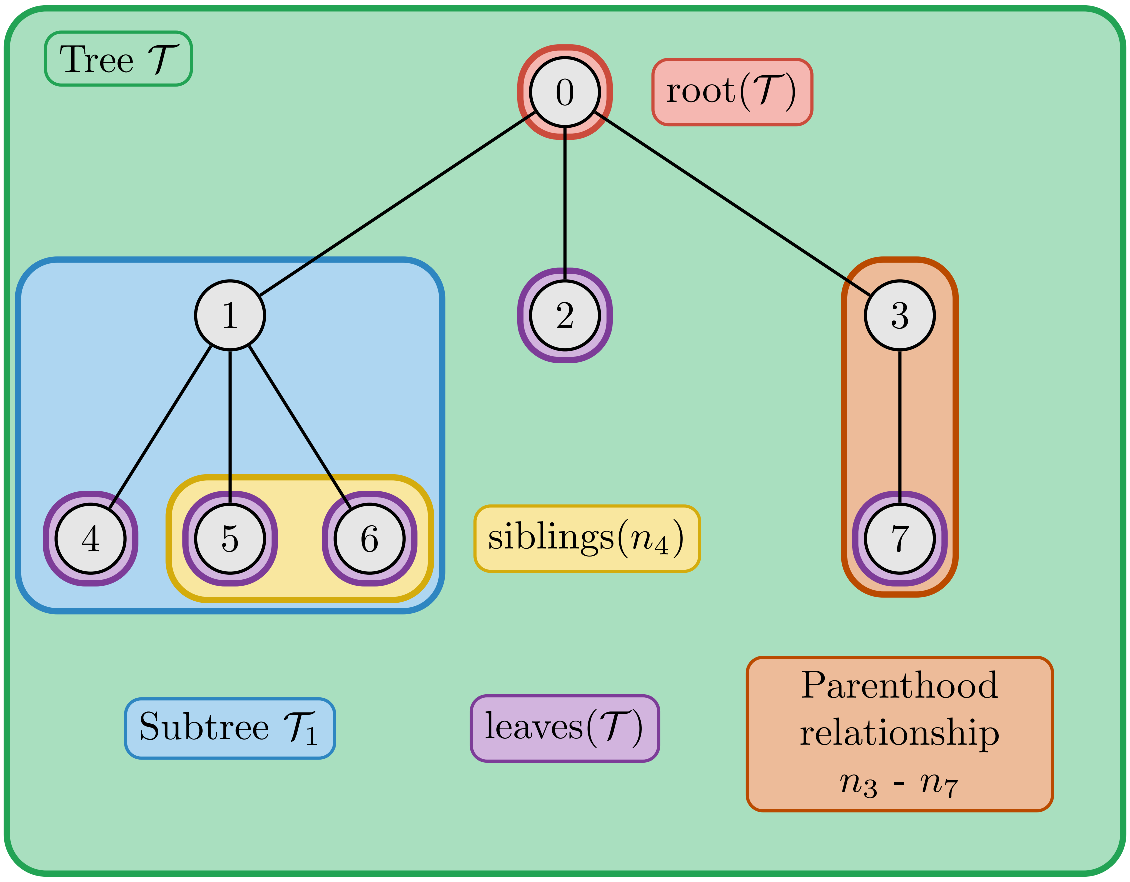
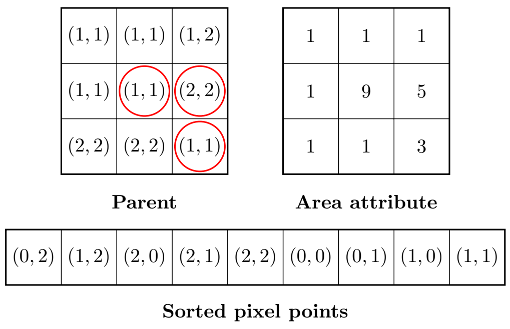


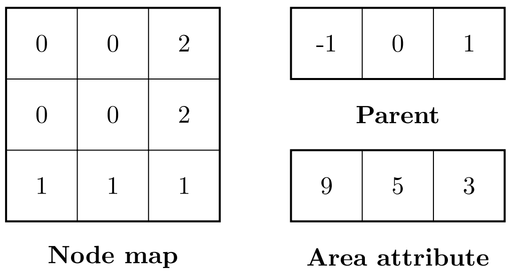
Figure 4.2: Tree terminologies

4.2 Tree-based representations of hierarchies

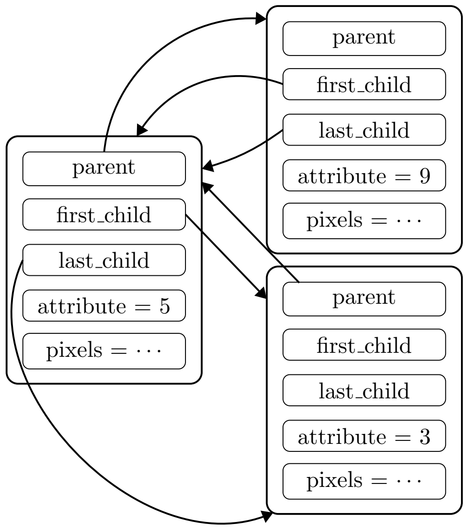
Hierarchical representations are usually represented by trees encoding the relation between the regions of a hierarchy. Let $G = (V, E)$ be a graph. Let the sequence $\pi(v_1, v_n) = (v_1, v_2, ..., v_n)$ with $v_i \in V$ be a path between two vertices v_1 and v_n such that there exists an edge $e_{v_i,v_{i+1}} \in E$ between any two consecutive vertices v_i and v_{i+1} of the sequence. A graph is said to be connected if there exists a path between any pair of vertices of the graph, and the graph is acyclic if this path is unique. A tree is defined as a connected acyclic graph. In the context of hierarchical representations, rooted trees, which are trees with one vertex designed as the root, are used. The terminology *node* for a tree is equivalent to a vertex of a graph G. The *parent* of a node is its consecutive node in its path to the root. Thus, the root of the tree is the only node that does not have any parent. Conversely, a *child* of a node n is a node that has n as its parent. The *siblings* of a node n which has n_p as its parent are the other nodes having



(a) Tree implementation in [16]



(b) Tree implementation in Pylene [208]



(c) Implementation of a tree as a linked list

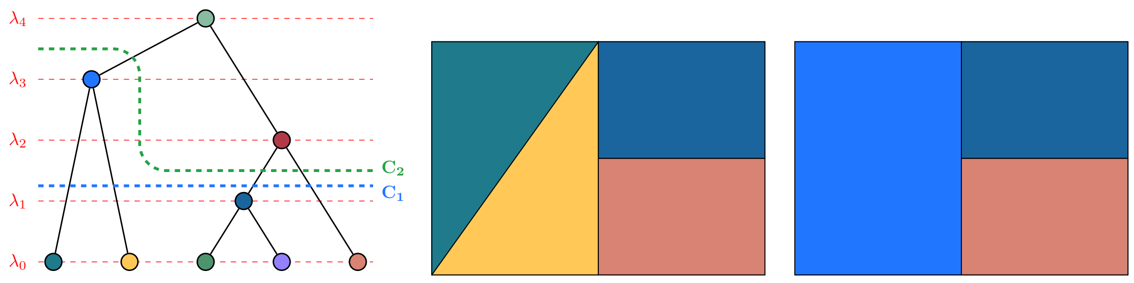
Figure 4.3: Different implementations of a tree

n_p as their parent. The nodes without any child are the *leaves* of the tree. A *descendant* of a node n is a node having n in its path to the root. Conversely, an *ascendant* of a node n is a node having n as a descendant. The subtree rooted in n is a tree whose nodes are composed of all the descendants of n and n itself, and whose parenthood relationship is preserved except for n which is the root. In the following, a tree is denoted by $\mathcal{T}$, and this notation is subscripted when using a particular hierarchical representation.

Tree terminologies are illustrated in Figure 4.2. The nodes of the tree are indexed by the value displayed and denoted by n_i for a node whose displayed value is i. In this figure, the tree is rooted at n_0, and the leaves are the nodes in $\{n_2, n_4, n_5, n_6, n_7\}$. The parenthood relationship is illustrated in the case of the pair of nodes (n_3, n_7): n_3 is the parent of n_7 and n_7 is its child. The siblings of n_4 are n_5 and n_6, the three nodes having the same parent n_1. The node n_1 is a descendant of n_0, as well as all the nodes of the tree except n_0 itself since it is the root of $\mathcal{T}$, and the descendants of n_1 are in the set $\{n_4, n_5, n_6\}$.

To implement a hierarchical representation, two data structures can be used to represent a tree, as discussed in [87]. The first data structure is the usage of a mapping from a node to its parent node. Such mappings are usually implemented as tables and the nodes are represented as indices of a table. In the context of the max-tree [16] or the tree of shapes [73] construction algorithms, this mapping takes the form of an image: each position of the image is linked to a *representative* point, corresponding to a node of the tree and mapping to the representative point of the parent node. Using an image to represent the parenthood relationship of a tree takes advantage of the nature of inclusion hierarchies, which represent the merging of the components of an image until the whole domain is covered, to know in advance the size of the structure and to allocate it only once. This representation is illustrated in Figure 4.3a. In the *parent* image, the representative points are circled in red and they map to their parent representative point, except for the root of the tree which is mapped to itself. The sorted pixel points define the topological order of the tree such that it can be traversed from root to leaves or the other way around. Finally, the attributes of the tree, which are information about the tree or the underlying image such as the number of pixels in a component represented by a node of the tree, are stored in similar images.

However, implementing partitioning hierarchies, which represent the merging of two or more regions in the lattice of partitions, using the representation of [16] is not possible as its size is not fixed. Thus, in this case, the mapping used is a table [87, 152, 242], and is either allocated once with a too large portion of memory or reallocated many times. The first strategy results in a waste of allocated memory, while the second creates an overhead during the hierarchy construction algorithms. A third strategy consists in estimating the size of the tree before its construction, as it is performed for the α-tree [242] by observing the histogram of the dissimilarity of the pixels. The implementation of a tree in the Higra library [164] uses a table to represent the parenthood relationship. The leaves of the tree, indexed from 0 to $(n-1)$, represent the n pixels of an image (represented as the vertices of a graph) and the m regions of a hierarchy are indexed from n to $(m-1)$. The Pylene library [208] mixes the representation of a tree from Higra and Berger *et al.* [16], as illustrated in Figure 4.3b. As in Higra, the parenthood relationship of the different nodes is stored in a table. Each pixel of the image is linked to a node of the tree by the use of a node map which has the same dimensions as the input image. The usage of a table has several advantages compared with the previous representation of the parenthood relationship based on images: in addition to being able to represent partitioning hierarchies, the parent table is 1D and does not rely on the input image whatever its number of dimensions. This holds true for the attribute array as well. Thus, nodes are represented by indices and not by n-dimensional points. Only the node map is dependent on the input image. For a generic implementation of a hierarchy construction algorithm, only the



(a) A tree $\mathcal{T}$ (b) Partition induced by $\mathbf{C_1}$ (c) Partition induced by $\mathbf{C_2}$

Figure 4.4: Tree representation of a hierarchy with cuts

number of dimensions of the node map has to be taken into account for the combinatorial calculation.

The other data structure used to represent trees is the usage of a linked list [86, 23]. It is composed of several inner data structures representing the needed information for a node, and each of them is linked to another one to represent the relationship between the nodes. Its usage has some advantages: it allows more flexibility when adding and removing a node of the tree, each link being represented by a pointer. When using tables, the deletion operation may require shifting the nodes and updating their indices. However, it requires more memory than tables due to the storage of pointers for the parenthood relationship, but also for the storage of the attributes, which are difficult to store in tables since a linked list node has no index. Furthermore, this implementation needs to allocate dynamically each structure, which can lead to serious runtime overhead. An example of a tree implemented as a linked list is illustrated in Figure 4.3c.

These trees, when linked with hierarchical representations of images, encode the relation between the regions of the different (partial) partitions. Figure 4.4 illustrates the tree representation of a hierarchy, and more particularly, Figure 4.4a is the tree representation of the hierarchy induced by the set of partitions from Figure 4.1. Each node is related to a region, colored by the same color as in Figure 4.1. Furthermore, each node is related to a value, which is the scale of appearance of a region and corresponds to the index of the partition in which it appears. A *cut* $\mathbf{C}$ is a (partial) partition composed of regions represented by the nodes of the tree. This cut is said *horizontal* if all the regions of the cut belong to the same (partial) partition of the hierarchy represented by the tree. Two examples of cuts are illustrated in Figures 4.4b and 4.4c. The cut $\mathbf{C_1}$ is horizontal: all its regions belong to the partition of the hierarchy indexed at λ_1. Conversely, the cut $\mathbf{C_2}$ is not horizontal: the bright blue region appears in the partition indexed at λ_3, while the two others are merged to form a new region in the partition indexed at λ_2. Thus, all the regions do not belong to the same partition of the hierarchy.

4.3 An overview of hierarchical representations of images

As mentioned above, hierarchical representations can be divided into two categories [24]: inclusion hierarchies and partitioning hierarchies. Each category has different representations with their own properties. In this section, an overview of these representations is provided.

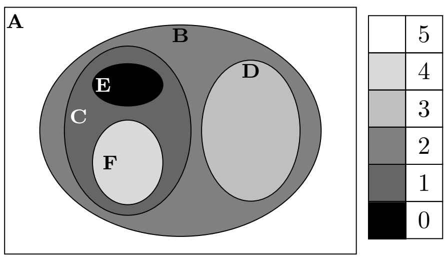
4.3.1 Inclusion hierarchies

In this part, inclusion hierarchies are studied. More specifically, three representations are discussed: the max-tree [183] and its dual, the min-tree, and the tree of shapes [141].

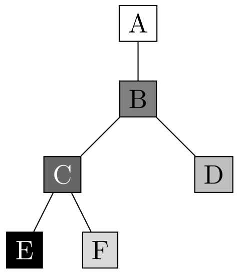
The max-tree and its dual: the min-tree

Let $f : \Omega \to \mathcal{V}$ be an image whose value set $\mathcal{V}$ is endowed with an order relationship $\leq$. Let $[f \geq \lambda] = \{p \in \Omega | f(p) \geq \lambda\}$ be the upper thresholds set and $[f \leq \lambda] = \{p \in \Omega | f(p) \leq \lambda\}$ be the lower thresholds set, with $\lambda \in \mathcal{V}$. Let $\mathcal{CC}(X)$ be the operator returning the set of connected component of a set X. The set $\mathfrak{C}^{\geq} = \cup_{\lambda \in \mathcal{V}} \{X | X \in \mathcal{CC}([f \geq \lambda])\}$ denotes the set of upper connected components and the set $\mathfrak{C}^{\leq} = \cup_{\lambda \in \mathcal{V}} \{X | X \in \mathcal{CC}([f \leq \lambda])\}$ denotes the set of lower connected components. The max-tree $\mathcal{T}_{\geq}$ and the min-tree $\mathcal{T}_{\leq}$ respectively represents the inclusion relationship of the connected component of $\mathfrak{C}^{\geq}$ and $\mathfrak{C}^{\leq}$ and are illustrated in Figures 4.5c and 4.5d when built on the image displayed on Figure 4.5a. In addition to the parenthood relationship, the nodes of the max-tree and the min-tree usually handle a link to the components they represent in addition to their value.

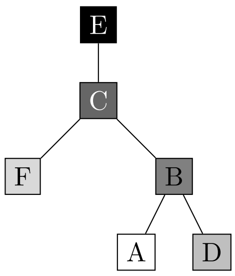
The max-tree has been proposed under the name of *component tree* by Jones [100, 101] in order to filter and segment an image using some attribute, and in [183] by Salembier *et al.* under the name max-tree. Filtering an image using the max-tree or the min-tree is a three steps method: tree construction, tree filtering, and image restitution. There exist numerous ways to build a tree from an image, and the choice of the algorithm depends on the property of the image, but also on the machine, as explained in [34]. In this article, the authors give a review of the different max-tree construction algorithms and provide a decision



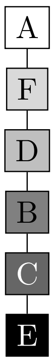
(a) An image



(b) Tree of shapes $\mathcal{T}_\circ$ of (a)



(c) Max-tree $\mathcal{T}_\geq$ of (a)



(d) Min-tree $\mathcal{T}_\leq$ of (a)

Figure 4.5: The three inclusion hierarchies presented in this chapter. Each node represents a component of the image in Figure 4.5a

tree for that choice. Concerning the image restitution, there exist different ways to output an image from the tree, but one of the most simple consists in giving the value related to a node at all the pixels represented by this node.

The filtering of the max-tree and the min-tree relies on different kinds of criteria. Among them, attributes computed on nodes are widely used, and their usage for filtering operations on these trees results in connected operators by attributes. These attributes can be increasing or not. Let $\mathcal{A}$ be an attribute computed on every node of a tree $\mathcal{T}$. An attribute $\mathcal{A}$ is increasing if for any node n of $\mathcal{T}$, except the root, $\mathcal{A}(n) < \mathcal{A}(n_p)$, with n_p the parent node of n. Such increasing attributes include the area of a region (its number of pixels) [226], the dynamics [81], or the volume [224]. The filtering of a tree $\mathcal{T}$ based on an increasing attribute is simple: given a threshold λ, all subtrees of $\mathcal{T}$ rooted in r are removed if $\mathcal{A}(r) < \lambda$.

Several situations require to use non-increasing attributes. For example, the filtering of rectangular objects in an image is based on the rectangularity attribute, which is the ratio between the area of a region and its bounding box. In this case, the attribute value will be near 1 when the object is rectangular, meaning that the region almost fills the bounding box. However, when traversing the tree from the leaves to the root, the rectangular object will be merged with the other element of the image, which creates a non linear variation of the attribute along a branch of the tree. In this context, the filtering of the tree is more complex.

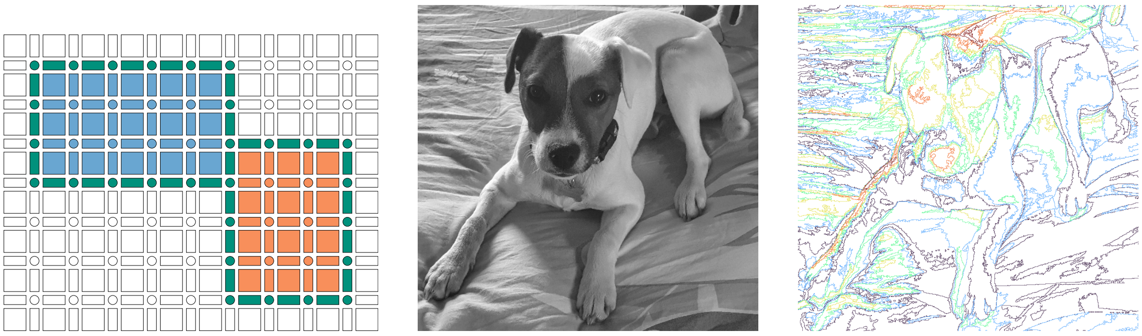
In [183], four rules are proposed to perform tree filtering on a max-tree when non-increasing attributes are used as a criterion. Furthermore, the subtractive rule is introduced in [223]. These rules are described below using an attribute $\mathcal{A}$ and a threshold λ:

- *Direct*: A node n is removed if $\mathcal{A}(n) < \lambda$.
- *Min*: A node n is removed if $\mathcal{A}(n) < \lambda$ or if one of its ancestors is removed.
- *Max*: A node n is removed if $\mathcal{A}(n) < \lambda$ and all its descendant nodes are removed as well.
- *Viterbi*: The removal of a node is based on an optimization problem based on the paths from the leaves to the root. This rule is based on the Viterbi algorithm [228] and is explained in details in [183].
- *Subtractive*: This rule is similar to the *direct* rule, except that the value related to the nodes of the subtree rooted in n are lowered by the amount of the value of the node n.

Filtering using the max-tree and the min-tree usage is presented above, but they can be used for different purposes. In [16], the max-tree is used for object detection in astronomical data, and in [183], video filtering is performed. Furthermore, it is used in the context of image compression in [221]. Finally, a last example of application is feature extraction [156] in order to create correspondences between pairs of images using MSER [133].

The tree of shapes

The tree of shapes [141], also known as the topographic map, is a hierarchical representation combining the max-tree and the min-tree representations. A shape $\mathcal{C}$ is a connected component of an image with its holes filled. Formally, it belongs to the set of shapes $\mathfrak{C}^{\circ} = \{\text{Sat}(X) | X \in \mathfrak{C}^{\leq}\} \cup \{\text{Sat}(X) | X \in \mathfrak{C}^{\leq}\}$ with $\text{Sat}(X)$ the operator filling the holes of a connected component X. The inclusion relationship between the element of $\mathfrak{C}^{\circ}$ forms the tree of shapes $\mathcal{T}^{\circ}$. This tree is illustrated in Figure 4.5b when built on the image Figure 4.5a. As for the max-tree and the min-tree, each node of the tree of shapes is related to the value of the connected component it is merged, but also with its pixels in the image.



(a) Shapes topology in $\mathcal{K}_\Omega$ (b) A natural image (c) Level lines of (b)

Figure 4.6: Shapes related to level lines

The tree of shapes also encodes the inclusion relationship of the level lines of the image. These level lines are indeed the borders of the shapes $\mathcal{C} \in \mathfrak{C}^\circ$, and are denoted by $\partial\mathcal{C}$. In recent construction algorithms [32, 73], the authors use a Khalimsky grid, noted $\mathcal{K}_\Omega$, which is a cubical complex representation of a hyperrectangular domain Ω in order to build the tree. Similarly to simplicial complexes, this grid is composed of n-faces. Such a grid is illustrated in 2D in Figure 4.6a. In this figure, the 0-faces are depicted by circles the 1-faces by rectangles, and the 2-faces by squares. The 2-faces represent the pixel locations in the original image while the 0-faces and 1-faces represent the interpixel ones. Thus, they are used to represent both the shapes and the level lines of the image. This figure thus illustrates two shapes, in blue and orange, each of them enclosed by level lines in green. It has to be noted that two shapes do not share the same pixel but can share the same border, as it is illustrated in that figure. Figure 4.6c illustrates the level lines of the natural image in Figure 4.6b. The color of these lines shows the depth of the node representing the level lines in the tree of shapes, varying from purple (low depth) to red (deep depth).

Several applications use the tree of shapes. First, it is used for filtering [141] in a similar manner as the max-tree. For example, it is possible to define a grain filter by computing the area of each node, and then remove all the nodes whose area does not meet a thresholding criterion. The case of non-increasing criteria uses a more sophisticated methodology in the shape spaces [237, 239] in which a min-tree $\mathcal{T}_\leq$ is built on the tree of shapes seen as a vertex-weighted graph and then filtered. In this case, the values related to each node of $\mathcal{T}_\leq$ are increasing, and the filtering is then simple to perform. A non-exhaustive list of applications that can be performed with the tree of shapes includes object detection [238], text detection [129], energy-based image simplification [240], local feature detection [241], interactive segmentation [39], medical imaging [37] and noise level estimation [63].

4.3.2 Partitioning hierarchies

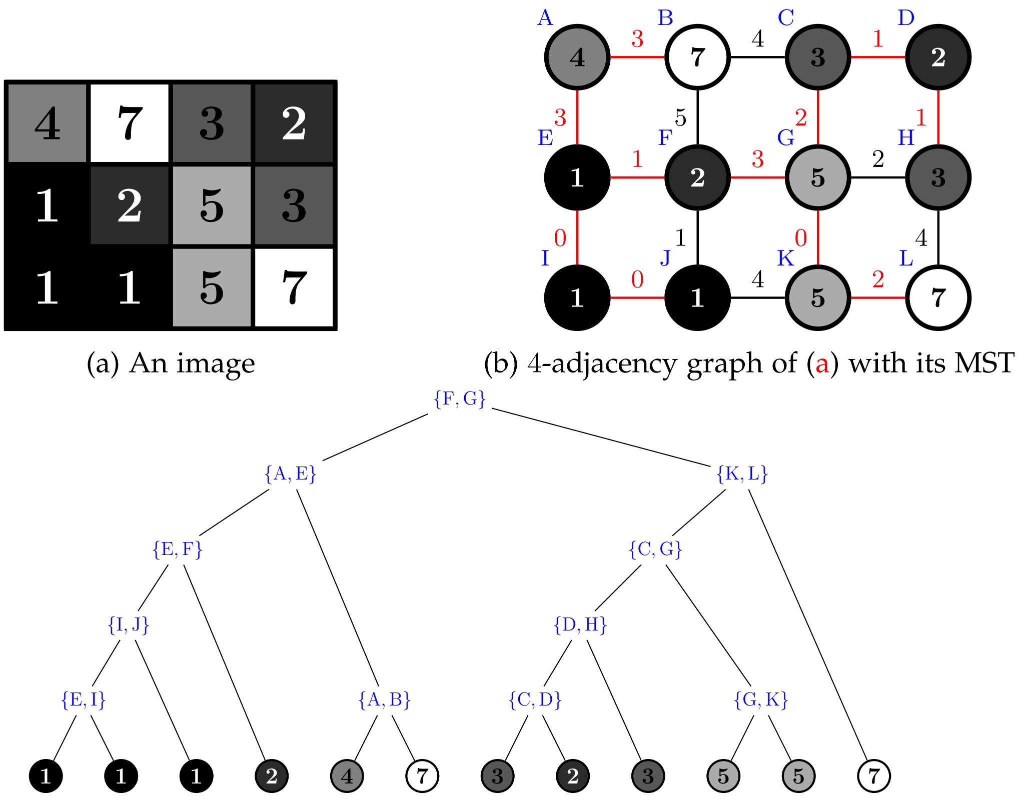
In the previous section, inclusion hierarchies were investigated. As stated above, these hierarchies represent stacks of partial partitions. In this section, partitioning hierarchies, which represent stacks of partitions, are overviewed.

Binary partition trees

Binary partition trees [182] (BPT) represent the iterative merging of regions into new regions of a partition. Each node has either two children, meaning that it represents the merging of two neighboring regions, or none: the node is thus a leaf and represents a region of the initial partition. Its construction is based on a bottom-up process relying on two parameters: an *initial partition* of an image, and a *merging procedure*. The first parameter, the initial partition, can be obtained by different means such as the watershed algorithm [19] creating regions based on markers, or the SLIC algorithm [3] which results in superpixels based on the combination of spatial and colorimetric similarity.

The merging procedure itself relies on two parameters: a region model based on some of its properties, and a merging criterion. A usual region model consists in taking the average value of a region. Furthermore, in the case the image defines values in the RGB space, a color space conversion can be performed. For example, the CIELAB is defined to be near the human perception in terms of distance between colors, which may be used as a merging criterion.

A particular BPT, coming from the framework of graph-based mathematical morphology [151], and more specifically edge-weighted graphs, is the Binary Partition Tree by Altitude Ordering [49] (BPTAO). Let $G = (V, E)$ be a graph with V its vertex set and E its edge set. Let $w_e : E \to \mathcal{V}_e$ be a mapping from the edge set E to a value set $\mathcal{V}_e$ endowed with an ordering relationship $\leq$. This mapping weights the graph G such that (G, w_e) is an edge-weighted graph. Let $\prec$ be a binary relation such that for any two edges $u_i, u_j \in E$, only one of the relations $u_i \prec u_j$, $u_j \prec u_i$ or $u_i = u_j$ holds. The relation $\prec$ is an *altitude ordering* if for any two $u_i, u_j \in E$, $u_i \prec u_j \Leftrightarrow w_e(u_i) \leq w_e(u_j)$. Let $k \in [\![1..|E|]\!]$ and $\mathcal{B}_0 = \{\{v\} \mid v \in V\}$. The partial binary partition hierarchy $\mathcal{B}_k$ at rank k for an altitude ordering $\prec$ is defined by $\mathcal{B}_k = \mathcal{B}_{k-1}^{u_k^{\prec}} \cup \{\mathcal{CC}_{v_i}(\mathcal{B}_{k-1}) \cup \mathcal{CC}_{v_j}(\mathcal{B}_{k-1})\}$ with $u_k^{\prec} = \{v_i, v_j\}$ the k^{th} edge in the ordering relationship $\prec$, $\mathcal{B}_{k-1}^{u_k^{\prec}}$ the set $\mathcal{B}_{k-1} \setminus \{\mathcal{CC}_{v_i}(\mathcal{B}_{k-1}), \mathcal{CC}_{v_j}(\mathcal{B}_{k-1})\}$ and $\mathcal{CC}_v(X)$ the operator returning the component of X containing v. The partial binary partition hierarchy at rank $|E|$ is called the *binary partition hierarchy by altitude ordering*. The tree induced by this hierarchy is the binary partition tree by altitude ordering for $\prec$, denoted by $\mathcal{T}_{\prec}$. Each node of $\mathcal{T}_{\prec}$represents the merging of the regions $\mathcal{CC}_{v_i}(\mathcal{B}_{k-1})$ and $\mathcal{CC}_{v_j}(\mathcal{B}_{k-1})$ and is related to the edge $u_k^{\prec} = \{v_i, v_j\}$. Thus, the value related to each node, called *altitude*, is $w_e(u_k^{\prec})$.



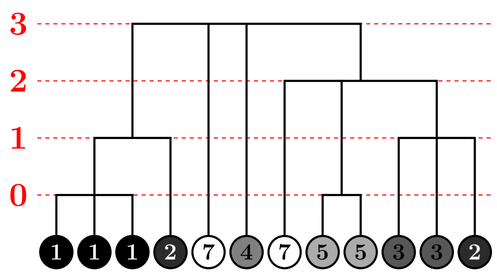
(a) An image

(b) 4-adjacency graph of (a) with its MST

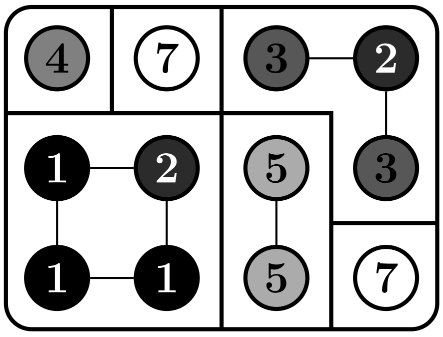
(c) BPTAO of (b)

Figure 4.7: BPTAO on a graph and its relation with the MST

Two partial binary partitions $\mathcal{B}_{k-1}$ and $\mathcal{B}_k$ can be identical if the edge $u_k^{\prec}$ has its vertex already connected in $\mathcal{B}_{k-1}$. Furthermore, from the definition of an altitude ordering, there might exist several altitude orderings for one graph. In [152], the altitude ordering is built from a minimum spanning tree (MST) using the Kruskal algorithm. These are illustrated in Figure 4.7, which shows the BPTAO built on the 4-adjacency graph of the image in Figure 4.7a. This graph is weighted on its edges by a L_1 distance, an index is given to each vertex by the capital letter in blue, and the edges belonging to its MST are highlighted in red. The order of the edges of the MST obtained during the construction of the BPTAO results in the altitude ordering $\{E, I\} \prec \{G, K\} \prec \{I, J\} \prec \{C, D\} \prec \{D, H\} \prec \{E, F\} \prec \{C, G\} \prec \{K, L\} \prec \{A, B\} \prec \{A, E\} \prec \{F, G\}$. Due to the links between the hierarchies proposed in [49], the BPTAO is related to numerous hierarchies such as the α-tree or the hierarchy of minimum spanning forest (MSF), which are studied in the following.



(a) The α-tree $\mathcal{T}_\alpha$ of Figure 4.7a



(b) The 1-partition of $\mathcal{T}_\alpha$

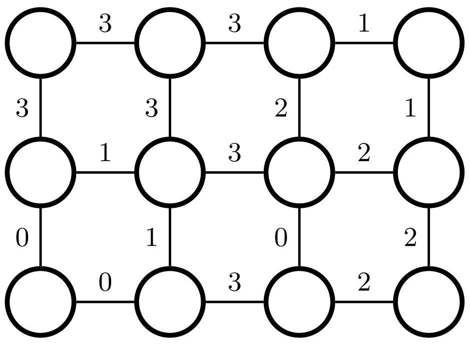
Figure 4.8: The α-tree of Figure 4.7a and its 1-partition

The α-tree and constrained connectivity

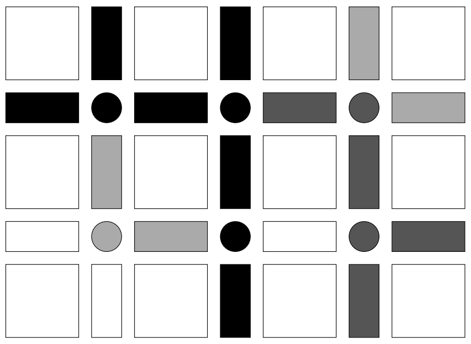
The α-tree [157, 158], also known as the quasi-flat zones hierarchy [49, 148], is the hierarchy representing the stack of partitions composed of α-connected components [195]. Let $\pi(p_1, p_n)$ be a path between two points $p_1, p_n \in \Omega$. The points p_1 and p_n are said to be α-connected if for every consecutive point p_i and p_{i+1}, for $i \in [\![1..n-1]\!]$, $w_e(\{p_i, p_{i+1}\}) \leq \alpha$. In this context, the image is seen as an edge-weighted graph (G, w_e), and the domain Ω is defined on the vertices of the graph. The edge weights w_e are used as a dissimilarity. An α-connected component, denoted by α-CC, is a component composed of α-connected points. An α-partition is a partition composed of α-connected components. The value α is a scale parameter for the α-partitions: when it is increasing, the α-connected components are merged to result in coarser α-connected components. This relation of merging is represented by the parenthood relationship of the α-tree, and the value related to each node is the scale α at which the α-connected component appears in the hierarchy.

These concepts are illustrated in Figure 4.8. In Figure 4.8a, the α-tree, represented as a dendrogram, is built on the image in Figure 4.7a. To this aim, the 4-adjacency graph in Figure 4.7b is used, as well as its weights for the dissimilarity. This tree illustrates the relation between several hierarchical representations [49]: first, the α-tree is a simplified version of the BPTAO, in which the consecutive nodes with the same altitude are merged to result in only one node per altitude. Then, the α-tree is indeed the min-tree of the MST, displayed in red in Figure 4.7b. Figure 4.8b illustrates a horizontal cut with $\alpha = 1$, which is the 1-partition of the hierarchy. In this partition, only the points with a dissimilarity $w_e(\{p_i, p_{i+1}\}) \leq 1$ are connected.

In practice, the α-tree is not used directly in applications since it suffers from a major issue often referred to as the *leakage effect* [159]. This leakage



(a) Saliency map $\Psi(\mathcal{T}_\alpha)$



(b) Saliency map $\Psi(\mathcal{T}_\alpha)$ immersed in a Khalimsky grid $\mathcal{K}_\Omega$

Figure 4.9: A saliency map and its mean of visualization for 4-adjacency

merges regions in the hierarchy which are different in term of content but whose similarity between two connected points is sufficient enough to be merged. To tackle this issue, Soille [195] proposes to add a global constraint on the regions of the α-tree. Let ϖ be a global constraint on a region. In practice, ϖ is often the range of the value set of a region R in the case the image is univariate, such that $\varpi(R) = \max\{\|f(p) - f(q)\|_1,\ \forall p, q \in R,\ p \neq q\}$. An (α, ω)-connected component is thus the largest α-connected component satisfying the global constraint G such that (α, ω)-CC $= \max\{\alpha_i\text{-CC} \mid \alpha_i \leq \alpha \wedge \varpi(\alpha_i\text{-CC}) \leq \omega\}$. It is interesting to note that $\alpha \geq \omega$ is equivalent to $\alpha = \omega$. This leads to the definition of an ω-connected components such as ω-CC $= \max\{\alpha_i\text{-CC} \mid \varpi(\alpha_i\text{-CC}) \leq \omega\}$. As for the α-tree, the relation of merging of the ω-connected components results in the ω-tree, denoted by $\mathcal{T}_\omega$, and each node is related to the ω level at which the regions are merged to form a new region.

From the beginning of this section, hierarchies are defined in terms of regions, but can also be represented as a set of contours. Such representations have already been studied in various contexts [6, 48, 83, 149] and are named *saliency maps* (or *ultrametric contour maps*). Let $G = (V, E)$ be a graph with V its vertex set and E its edge set. Let $\Psi(\mathcal{T}) : E \rightarrow \mathcal{V}_e$ be a mapping from an edge to a set of values on the edges. For an edge $u = \{v_i, v_j\}$, the value $\Psi(\mathcal{T})(u)$ is the value of the lowest node of the tree $\mathcal{T}$ such that v_i and v_j belong to the same component. Figure 4.9 illustrates by two means the saliency map of the α-tree $\mathcal{T}_\alpha$ from Figure 4.8a. In the first illustration in Figure 4.9a, the mapping is applied to all the edges of the graph and displayed. It has to be noted that by thresholding and removing the edges lower than a given threshold α, an α-partition is obtained. Figure 4.9b is the result of the immersion of $\Psi(\mathcal{T}_\alpha)$ in a Khalimsky grid $\mathcal{K}_\Omega$. This representation is possible in 2D only if the graph is a 4-adjacency graph, but it allows a visual representation of the saliency map, and thus the hierarchy in terms of contours. To build this representation, the 2-faces are set to 0, the 1-faces are set to the edge value of the saliency map they represent and the 0-faces are set to the maximum value of the adjacent 1-faces. Note that for visualization purposes and in the following of this thesis, saliency map values are inversed: the lowest values are bright and the greatest values are dark. Finally, the tree representation of any hierarchy can be built from the saliency map using the α-tree [48].

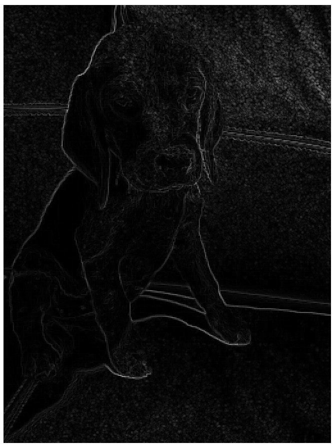
Hierarchical watersheds

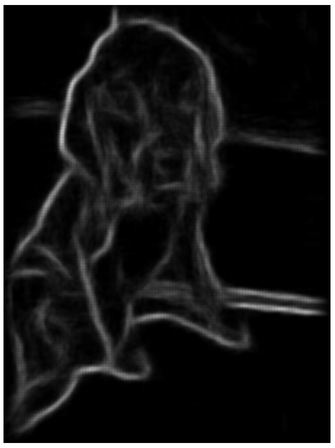
Watershed segmentation [19] is the main tool for segmentation in mathematical morphology. Several examples of watershed segmentations on different structures are illustrated in the previous chapter to demonstrate its generic capabilities. Its operation is straightforward: it floods regions called catchment basins from predefined markers until two regions meet, creating a watershed line at this meeting point. In practice, the watershed is not run directly on an image, but on its gradient, allowing to use the local minima of this gradient as markers. However, such a gradient usually has a great number of local minima, and the resulting segmentation is oversegmented. Several improvements to reduce this effect have been proposed such as the usage of user input markers, gradient filtering, or the usage of hierarchies based on the watershed [18, 47, 136].

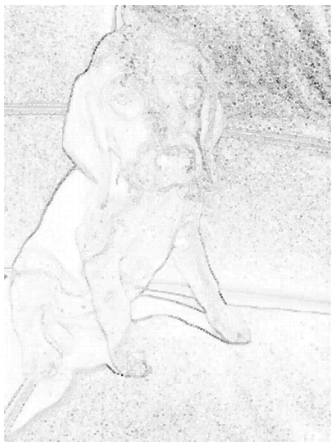
A particular class of hierarchical watershed belonging to the framework of edge-weighted graphs is based on the minimum spanning forests [47], which are stacks of watershed cuts [46]. These hierarchies are built by a bottom-up process by ordering markers such as minima using a given criterion. The marker order can be specified using the BPTAO [49, 152]. Furthermore, this criterion is usually the extinction value [224] of an attribute such as the area or the dynamics [81], the result being a hierarchical watershed (HWS) by attribute.

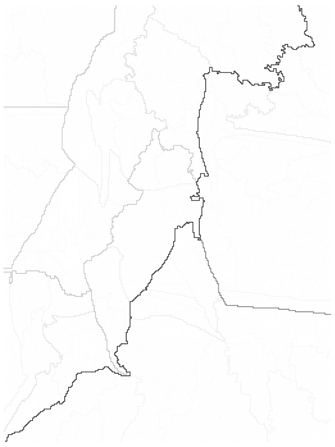
Saliency maps of hierarchical watersheds compared with the α-tree are given in Figure 4.10. The first row of this figure illustrates the inputs to obtain the hierarchies: Figure 4.10a is the input image from which the gradients in Figures 4.10b and 4.10c are computed. The first gradient weights the edges of the 4-adjacency graph of the image using an L_2 distance between two adjacent pixels values, while the second uses a gradient computed using an edge detector based on the random forest framework and named Structured Edge Detection (SED) [58]. The usage of a learned gradient usually produces better results in terms of contour detection as noted in [163]. This is illustrated in the figure: the contour of the top of the head of the dog is not retrieved when using a L_2 gradient, while it is the case using SED. The influence of the gradient is

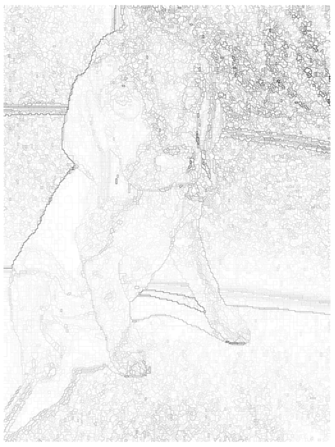
(a) An image

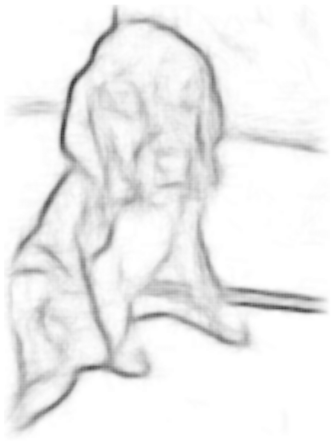

(b) A gradient (L_2 norm)

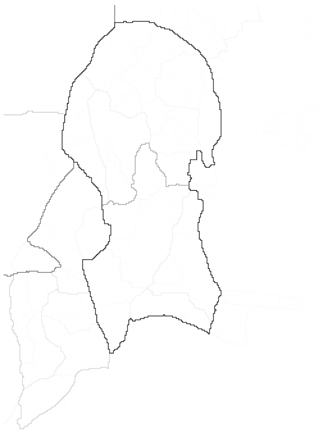

(c) A gradient (SED)


(d) α-tree


(e) HWS Area


(f) HWS Dynamics


(g) α-tree


(h) HWS area

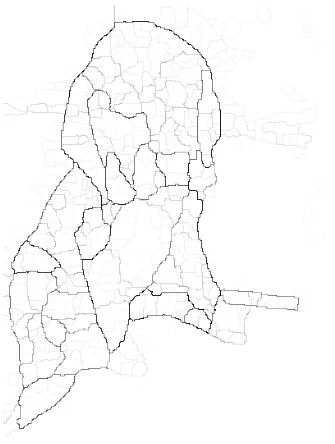
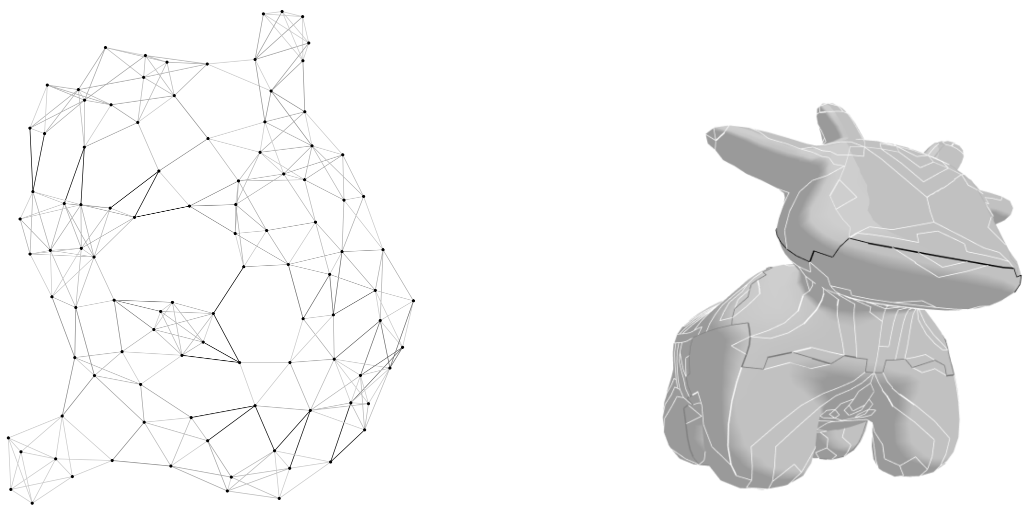

(i) HWS Dynamics

Figure 4.10: Saliency maps related to the gradient and the hierarchy. First row: Image and two processed gradients. Second row: saliency maps obtained from hierarchies computed on an L_2 gradient. Third row: saliency maps obtained from hierarchies computed on a SED gradient.



(a) On an edge-weighted graph (b) On a simplifical complex

Figure 4.11: Hierarchical watershed by area built on an edge-weighted graph and on a simplicial complex

directly observable in the saliency maps, by comparing the second row (L_2 gradient) and the third row (SED gradient). In the two cases, the α-tree produces contours that are not clear and whose segmentation results would be difficult to use in practice. Conversely, hierarchical watersheds produce thin contours. Furthermore, when computed on the SED gradient, the hierarchies exhibit more accurate contours in terms of object detection.

These hierarchies are evaluated for segmentation purposes [163], and it is concluded that they have results close to the state of the art. Furthermore, the usage of the BPTAO in concordance with saliency maps in the context of hierarchical watersheds results in even more efficient hierarchies, by either combining hierarchical watersheds computed with different attributes [50, 48, 186, 184] or by using the watersheding operator on any kind of hierarchy [185] in order to transform it into a hierarchical watershed.

Except for the tree of shapes, the construction of the hierarchical representations presented in this chapter may be implemented in a generic way on different image domains. One of the parameters of the construction of a hierarchy is the knowledge of the adjacency relationship of the elements of the domain Ω. As explained in Chapter 3, this relationship may be represented as a graph such that these hierarchies can be built on a graph either weighted on vertices or edges. The Higra library [164] relies on such graphs to build hierarchies. Figure 4.11 illustrates this concept by building a hierarchical watershed on two adjacency graphs obtained from the domain of the given image and computing the saliency map of the hierarchy. In Figure 4.11a, this graph is built on a set of points by the mean of the k-nearest neighbors algorithm based on a distance L_2 between two points. The edges are weighted with this distance between two points. In Figure 4.11b, the graph is obtained from the adjacency relationship of the triangles, and the edges are weighted using the triangle curvatures [181].

4.4 Conclusion

This chapter concludes the first part of this document by presenting hierarchical representations of images. First, the notion of hierarchical representation was recalled, and its representation as a tree was introduced. The usage of trees enables simple and efficient processing on these hierarchies. These representations are at the heart of the contributions of the thesis, from their implementation to their usage in the different applications.

Part II

A Static-Dynamic Approach to Image Processing

Chapter 5

Static-Dynamic Genericity for Image Processing

In this chapter, we focus on generic programming in C++ for image processing, and more precisely we develop a methodology that allows writing algorithms that are either static, meaning that input data types are known at compile time, or dynamic, meaning that this knowledge depends on runtime information. Furthermore, we demonstrate that our methodology is extendable to other programming languages relying on the monomorphization mechanism. Finally, we analyze the performance of different image processing algorithm patterns, and we show that some information is more important to be known at compile-time depending on this pattern.

5.1 Issues with static genericity for image processing

Software libraries are sets of algorithms and data structures that aim to allow the reusability of program components. In Chapter 3, the three criteria for such libraries providing image processing functionalities were discussed: genericity, performance, and interactivity. Generic programming allows to reuse algorithms with different kinds of images, whatever their domain type, value type, or implementation. On the other hand, performance makes algorithms usable in real-time application, or reduce the execution speed for large image datasets or big images in terms of amount of pixels. Finally, interactivity is of prime importance when some information is only known at runtime, or to bring algorithms and data structures in an interactive environment such as Jupyter [168] or Pluto.jl [169].

Listing 5.1: Dynamic dispatch of templated functions with `variant` C++

```cpp
template <Image I>
void iota(I& img) { /* ... */ }

using image_t = std::variant<image2d<uint8_t>, image2d<rgb8>,
↪   image3d<uint8_t>, graph<double>, simplicial_complex<uint8_t>>;

void dispatch_iota(image_t& img) {
    std::visit([](auto& img_s) { iota(img_s); }, img);
}

int main(void) {
    image_t img = build_image();
    dispatch_iota(img);
}
```

The C++ programming language is a performance-oriented language whose programs are optimized at compile-time. It is one of the main languages used to implement image processing libraries as observed in the survey in section 3.3.3. As explained in Chapter 2, it provides generic capabilities using templates, a list of types and compile-time values. As C++ is a statically typed language, the type of all the expressions composing algorithm implementations must be known at compile-time, as well as the parameters of the template entities. Furthermore, templates rely on a monomorphization process: for every instantiation of a template entity, machine code is generated for it, and the compiler optimizes the corresponding machine code. Thus, the genericity and performance criteria are naturally fulfilled. However, these characteristics make the interactivity criterion not straightforward to reach.

There exist different means to perform dynamism in C++ for interactivity purposes, and these are described in Section 5.2. To illustrate the issues of interactivity for image processing in C++, the sample code in Listing 5.1 uses a variant [154], which is a type-safe union introduced in the C++ 17 standard. A union is a structure defining several objects with different types but storing only one of these objects at a time. All the different types are specified in the template parameter list. Then, a generic function named `iota` accepts any kind of object while they respect the concept `Image`. As this function is templated, its parameter has to be known at compile-time, but the objective of this sample code is to add interactivity at runtime. Thus, the function `dispatch_iota` takes as input an object whose type is the image variant type, which handles numerous

Listing 5.2: Dynamic dispatch of templated functions with any C++

```cpp
template <Image I>
void iota(I& img) { /* ... */ }

void dispatch_iota(std::any& img) {
    if (auto* img_s = std::any_cast<image2d<uint8_t>>(img))
        iota(*img_s);
    else if (auto* img_s = std::any_cast<graph<double>>(img))
        iota(*img_s);
    /* Other image structures ...*/
    else
        throw std::invalid_argument("Invalid image type");
}

int main(void) {
    std::any img = build_image(); /* Return a valid image */
    dispatch_iota(img);

    std::any index = std::make_any<int>(10);
    dispatch_iota(index); /* Raises an exception */
}
```

possible image types, and then dispatches to the correct type by means of the `std::visit` function. The issue of the usage of this function is that for all the image types defined in the variant, machine code for the `iota` function is generated. Furthermore, the same is true for all the value type parameters of the different image structures. Thus, the combinatorial of all the image types must be known in advance in the variant list, which is unmaintainable due to the uncountable amount of image types, as explained in Chapter 3.

Unlike variants, which are able to store objects whose types belong to a limited set, the `std::any` container [53] stores any object whatever its type. About type safety, `std::variant` is more restrictive since it checks that the object belongs to the set of accepted type at compile-time. On the other hand, such checks are performed at runtime with `std::any`, as illustrated by Listing 5.2. In this listing, the same operation is performed as in Listing 5.1, but using the `std::any` container. The function `build_image` creates a new image and stores it in a `std::any` object. This function is used in the `main` function to create an image which is given to the `dispatch_iota` function, which will get the object stored in the container by returning a pointer to the object with the correct type by the means of the `std::any_cast` function. If the type parameter of this function does not correspond to the type of the object stored in the container, a `nullptr` is returned. The dispatch is thus performed at runtime by trying to perform a conversion until the correct type is reached, or an exception is raised if the underlying type is not handled. This is illustrated by the two cases in the main function. The dispatch to the correct type requires to specify all the valid types by writing all the branches of the dispatch, resulting in a lot of boilerplate code to maintain. Such a dispatch can be factorized by the use of C++ metaprogramming in conjonction with type lists, but still necessitates to take into account the large majority of types such that it does not remove the combinatorial explosion mentioned previously.

In this chapter, we present a new methodology that improves the interactivity of generic image processing algorithms in C++ with the least modifications to make in existing generic algorithms. We focus on the value type of the image, but also their implementation. Then, we extend the concepts presented in the C++ language into the Rust language, with a generic internal mechanism similar to C++, but with other differences. Finally, we evaluate this new methodology on different algorithmic schemes, and we show that the cost of the indirections induced by the added dynamism is negligible for some classes of algorithms.

5.2 Dynamism in C++

Dynamism in C++ relies on the notions of inclusion polymorphism in the context of object-oriented programming. In C++, a class may inherit from one or more classes: it may access their operations and data based on certain access specifiers. Furthermore, these classes may define operations that are not implemented: in this case, the classes are *abstract classes* and the operations that are not implemented are *pure virtual functions*. An abstract class defines the interface its subclasses have to respect. In contrast with generic programming in C++, abstract classes are comparable to concepts. However, the concrete type does not require to be known at compile-time. Listing 5.3, defines an abstract class `Base` that defines a pure virtual function `process`, and two classes deriving from `Base` and implementing the function `process`. In the `main` function, the knowledge of the type of the variable `v` depends on some runtime information and is thus allocated dynamically as the size of the object cannot be known at compile-time. The dispatch of the `process` function, when it is used, depends on an internal mechanism called *virtual tables*, which may be seen as a table of pointers to function initialized at the object creation.

The usage of inheritance has several drawbacks for dynamism in C++. First, the pointer semantic is different from the value semantic: using pointers in

Listing 5.3: Dynamism with abstract classes C++

```
struct Base {
    virtual void process() = 0;
};

struct Derived1 : public Base {
    void process() { /* ... */ }
};

struct Derived2 : public Base {
    Derived2(int v) : attr(v) {}
    void process() { /* ... */ }
    int attr;
};

int main(int argc, char* argv[]) {
    bool cond = parse_arguments(argc, argv);
    std::unique_ptr<Base> v;
    if (cond)
        v = std::make_unique<Derived1>();
    else
        v = std::make_unique<Derived2>(/*...*/);
    v->process();
}
```

conjonction with values is not straightforward in a generic functions when their type is given as a template parameter. Furthermore, its usage requires dynamic allocations, which have an impact on the runtime performance of a program. For example, the usage of such a mechanism to define an image requires at least three interfaces: one for the value set, one for the domain, and the last one for the image implementation. Each of these elements would be dynamically allocated as their nature is not known at compile-time, as well as pixel information such as their positions or their values. In addition to this, the usage of a pointer induces *indirections* since no inlining can be performed due to the dynamism. Finally, adding a new type to the set of accepted types requires creating a new class inheriting from the base class, which is not the case with concepts that check the behavior of a given object type instead of its name.

Some of these issues may be solved by means of *type erasure*, which is a pattern hiding the underlying type of an object. This pattern is heavily used

```
template <class T>
struct buffer2d
{
  T& operator()(point2d p);
  rect2d domain() const;          Interface
- - - - - - - - - - - - - - - - - - - - - - - -
  T* data;                    Implementation
};                               details
```

(a) Statically typed image

```
struct buffer2d_any
{
  void* operator()(point2d p);
  rect2d domain() const;          Interface
- - - - - - - - - - - - - - - - - - - - - - - -
  void* data;                 Implementation
  size_t element_size;           details
};
```

(b) Dynamically typed image

Figure 5.1: Two implementations of an image encoded as a contiguous buffer

to perform runtime polymorphism, but without having to define a new class inheriting from a given base class. Runtime polymorphism with type-erasure is implemented in several libraries such as Boost.Typeerasure [231], Folly [66], or Dyno [57], which is a library specialized for runtime polymorphism using type-erasure and providing several mechanisms to optimize its usage depending on the use case. In the same manner, this mechanism is used in the STL in different cases. The `std::function` type can hold any callable object without having to know its implementation at compile-time, and without having, for such a callable, to inherit from a base class thanks to type-erasure. Finally, the `std::any` container [53] uses type-erasure to store objects of any type and performs type-safe conversion at runtime to use such an object.

The proposed approach to static-dynamic genericity is in fact inspired by Java generics internal mechanism. Indeed, Java generics are only syntactic tools that allow performing type checking at compile-time, but then transform the code using type erasure to remove the usage of generics in order to be usable by old versions of the Java Virtual Machine (JVM). Thus, in the next section, we present our method that relies on type erasure to bring dynamism to C++ generic algorithms and convert at runtime the object values to the correct type in order to be manipulated.

5.3 Static-dynamic genericity in C++

In this section, the solution to bring interactivity to image processing algorithms based on type-erasure is presented. The objective is to handle different value sets for a 2D image at runtime, but also different implementations of such images. Thus, in this section, we present an adaptation of image structure and algorithms to handle either static or dynamic genericity for image processing purposes.

```
template <class T>
struct indirect2d {
  T& operator()(point2d p);
  rect2d domain() const;                 Interface
- - - - - - - - - - - - - - - - - - - - - - - - - -
  std::function<T&(point2d)>            Implementation
                m_access;                  details
};
```

(a) Statically typed image

```
struct indirect2d_any {
  void* operator()(point2d p);
  rect2d domain() const;                  Interface
- - - - - - - - - - - - - - - - - - - - - - - - - -
  std::function<void*(point2d)>         Implementation
                m_access;                  details
};
```

(b) Dynamically typed image

Figure 5.2: Image structures with indirection on implementation

5.3.1 Image models

In Chapter 3, a classical implementation for a 2D image is defined as a buffer of values. Such buffer may be a contiguous table: this increases the performance of the image traversal by storing groups of successive pixel values in the memory cache of the processor, which has a faster access speed compared to the memory of the computer. Figure 5.1 illustrates two image data structures implemented as a buffer of data. They have the same interface, and thus respect the same concept in terms of generic programming: an access operator to the value of a pixel at a given position is defined in addition to a function returning the domain of the image. However, they differ in terms of implementation: the image structure from Figure 5.1a statically handles the information about the pixel value type using C++ template. Thus, such an image model may only be used when the input type is known before the usage of the compiled program and is limited in the context of interactivity, as previously explained.

The second structure in Figure 5.1b uses a type-erase buffer to store the pixel values. The information about the type of pixel values is stored at runtime in the implementation details. In the listing, such information is represented by the size in bytes of one pixel value in memory. Thus, the image can be traversed taking into account this information for the computation of the strides. Furthermore, thanks to type-erasure, the image is not templated: the type of the image is thus defined at runtime, and this allows to generate machine code for only one type of image. To this aim, the type of the buffer of value is `void*`, meaning that the values stored are untyped.

The two structures presented previously implement a 2D image as a buffer of data to store the information. However, in Chapter 3, different implementations of an image are discussed: an image may be implemented as a constant image, with a single value, a sparse matrix, or a view [179]. The buffers from Figure 5.1 can be replaced with any of those different implementations if it is known at compile-time. However, there exist some cases in which the implementation needs to be chosen at runtime depending on the data stored in an image file in the same way the pixels value type may require to be selected dynamically.

Image type	**Access policy**	**Static value type**	$f(p)$ **return type**
`buffer2d<T>`	Direct to the buffer	✓	`T&`
`indirect2d<T>`	Indirect	✓	`T&`
`buffer2d_any`	Direct to the buffer	✗	`void*`
`indirect2d_any`	Indirect	✗	`void*`

Table 5.1: Summary of the different image structures and their properties

Figure 5.2 illustrates two implementations of image structure, similar to the two image structures in Figure 5.1 in which the interface and the implementation details represent the two parts of the image structure, but this implementation is type-erased by the means of a `std::function` container. This container, briefly discussed in section 5.2, stores any object respecting the `Callable` concept, meaning that it can be called as a function with zero or more arguments and may return a value as specified by its template parameters. This is performed by the means of type-erasure such that the type of the callable can be known at runtime. As it is able to handle any kind of callable, the case of a function object is particularly interesting: such an object stores the implementation of the image, and its function call operator accesses this implementation and returns the value at a given position. This operation is performed when the accessor to a pixel value at a given position, located in the interface of the image, is called. Thus, accessing a pixel value of the image is performed at the cost of an indirection. Finally, as for the image structures in Figure 5.1, the information about the pixel values type can be static or dynamic, and it is handled by the implementation.

These four image structures are summarized in Table 5.1. The different access policies are important in terms of performance: when such access is indirect, a jump to a new function is generated in the machine code, and this downgrades the performance. This is due to the fact that the function is only known at runtime, and thus cannot be optimized conversely to a direct access which may be inlined. Furthermore, the knowledge of the type of the value at compile-time is important since the compiler may generate vectorized SIMD instructions, which enable several values to be computed in one instruction. Such a compiler optimization is not possible for dynamic values since their type is only known at runtime. Generating SIMD instructions would be achievable when using a Just-In-Time compiler, which is not the case for C++ compilers. Finally, for the same reasons as the value type knowledge, the return type of the

Listing 5.4: qsort function prototype and its usage C

```c
// Prototype
void qsort(void* tab, size_t nmemb, size_t size,
           int (*compare)(const void*, const void*))

// Comparison
int compare_uint16(const void* _a, const void* _b) {
    const uint16_t a = *((uint16_t*)_a);
    const uint16_t b = *((uint16_t*)_b);
    if (a < b) return -1;
    else if (a > b) return 1;
    else return 0;
}

// Usage
int main(void) {
    uint16_t tab[10] = /* ... */
    qsort(tab, 10, sizeof(uint16_t), compare_uint16);
}
```

image pixel value access operator is either statically typed, so a reference to a value is returned for read-write access, or dynamically typed such that a pointer to the first pixel value byte is returned. For the second case, the pointer value is required to be converted to the correct static type before being used. This is studied in the following section about the design of static-dynamic algorithms for image processing.

5.3.2 Application to algorithms

The application of the previously studied image structures on generic algorithms requires some adaptations. The case of a statically typed image such as `buffer2d<T>` and `indirect2d<T>` is relatively simple as the values are statically typed: the information about the pixels value type is known at compile-time and computations on these value are handled by the compiler in the machine code generation. However, dynamically typed pixel values are required to be converted in order to be manipulated: the value returned by the image pixel value accessor is a pointer to the first byte of the value at a given position, and pointers are not pixel values.

In the C programming language standard library, the `qsort` function, whose

Listing 5.5: Generic algorithm applied to the different structures C++

```cpp
// Generic algorithm
template <class I, class Op>
void generic_elementwise_op(I a, I b, I& out, Op& op) {
    for (auto p : a.domain())
        op(a(p), b(p), out(p))
}

// Statically typed values
template <class T, class Op>
requires (std::invocable<Op, T, T, T&>)
void elementwise_op(buffer2d<T> a, buffer2d<T> b, buffer2d<T>& out,
↪   Op& op) {
    generic_elementwise_op(a, b, out, op);
}

// Dynamically typed values
void elementwise_op(buffer2d_any a, buffer2d_any b, buffer2d_any&
↪   out, std::function<void(const void*, const void*, void*)> op)
{
    generic_elementwise_op(a, b, out, op);
}
```

prototype is displayed in Listing 5.4, is a routine that sorts the values of an array according to an input function handling their ordering. The input array to be sorted is type-erased: it is a pointer represented by a `void*`. Thus, any array, whatever its value type, can be sorted with this function. Two attributes of the array are required by the `qsort` function: the argument `nmemb` is the number of elements in the table, and the argument `size` is the size in bytes of one element. Thus, the product of these two arguments is the whole size in bytes of the array. Furthermore, these give indications about how to traverse the array, and thus how to retrieve the address of the first byte of a value. Finally, the comparison function takes as arguments the pointers to two values and returns an integer value related to the ordering relationship between these values. Such a function is displayed in the listing under the name `compare_uint16`. This function first requires converting the pointers to their static type and then accessing their value to be able to compare two elements of the table. With all these elements, the `qsort` function can be used to sort any table, as it is illustrated in the `main` function.

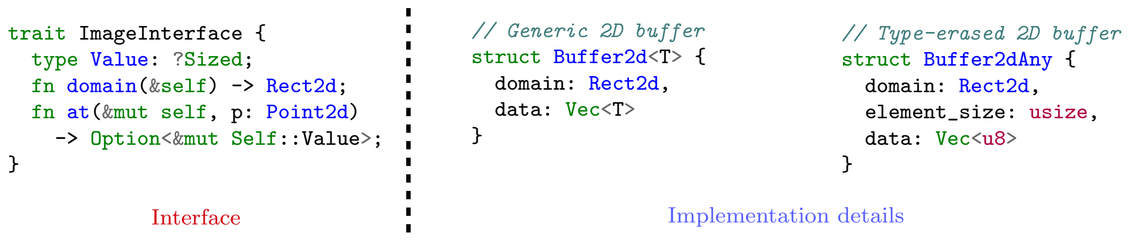
To adapt the generic algorithms to either take statically or dynamically typed images, the model proposed by the `qsort` function is extended to templated functions. Since the information about the dynamically typed values is stored directly in the image object, it is not required in the function prototype. Furthermore, these are useless in the case statically typed values are used. However, the operations on the image, similar to the `qsort` comparison function, are required. Such a generic algorithm is displayed in Listing 5.5. It aims to perform an elementwise operation `op` between two images `a` and `b` and to store the result in an image `out`. This algorithm uses the interface of the previously presented image structures and thus can be used with any of them whatever the pixel value type is static or dynamic. Furthermore, the operation is performed in-place: no value is returned. Thus, in the case of dynamically typed pixel values, the different conversions to read and write a value are performed by the function `op` in a similar manner as for the `compare` function.

In the second part of the listing, the generic function is specialized into two functions, one for static value type and the other for dynamic value type. These two functions only call the `generic_elementwise_op` function, but their arguments differ in terms of types. In the first case, the function `elementwise_op` is templated: the pixel value type along with the operation type must be known at compile-time. In this case, the operation does not perform any conversion and is likely to be inlined by the compiler if its code size is small enough. Thus, the `op` function only specifies the operation to be performed in the algorithm. In the second specialization, the function is not templated: all the information may be known at runtime only. Due to this, the operation is stored in a `std::function` container, and it has to perform the type conversion when performing the operation and as this type is only known at runtime, this justifies the usage of such a utility. Thus, using this algorithm on an image whose values are dynamically typed adds an indirection for the operation since it cannot be inlined by the compiler. Finally, it has to be noted that these specializations are extendable to the image structures whose access to the implementation is indirect in the same manner according to the dynamism of their pixel values.

Table 5.2 summarizes the different operations related to each image structure and the impact on the performance they induce in terms of indirections. For the templated structures, whose value type is handled statically, the operation is templated and thus it is known at compile-time, so that it may be inlined. In such a case, when the access is direct to the buffer, no indirections are made since all the information is known at compile-time. For image structures whose values are type-erased, the function is known at runtime and is represented as a pointer to function. Thus, in the generic algorithm in Listing 5.5, an indirection is performed for each operation on every pixel value. Thus, in the case the

Image type	Operation type	# Indirections
`buffer2d<T>`	`void max<T>(T a, T b, T& out)`	0
`indirect2d<T>`	`void max<T>(T a, T b, T& out)`	1
`buffer2d_any`	`void (*max)(const void* a, const void* b, void* out)`	1
`indirect2d_any`	`void (*max)(const void* a, const void* b, void* out)`	2

Table 5.2: Summary of the different operations according to the image structures



```
trait ImageInterface {
  type Value: ?Sized;
  fn domain(&self) -> Rect2d;
  fn at(&mut self, p: Point2d)
    -> Option<&mut Self::Value>;
}
```

```
// Generic 2D buffer
struct Buffer2d<T> {
  domain: Rect2d,
  data: Vec<T>
}
```

```
// Type-erased 2D buffer
struct Buffer2dAny {
  domain: Rect2d,
  element_size: usize,
  data: Vec<u8>
}
```

Interface
Implementation details

Figure 5.3: Rust implementation of a 2D buffer

implementation and the pixel value type are known at runtime, the cost of a pixel value access is 2.

5.4 Extension to the Rust programming language

In the previous section, the principle of static-dynamic genericity for image processing algorithms implemented in C++ was presented. To demonstrate that this approach is not limited to the C++ programming language, it is applied in this section to the Rust programming language. The choice of this language comes from the fact that it is a performance-oriented language with some generic capabilities based on parameters as explained in Chapter 2. Furthermore, it has some differences with C++ such as its nominal type system or its memory management system, which aims to be safe unless specified in terms of the memory ownership of an object.

Rust data structures are declared using the `struct` keyword and are similar to the declaration of a C structure. To add some operations related to the data structure, the usage of an `impl` block is required. Furthermore, as explained in Chapter 2, Rust is a nominal language, meaning that its concept checking is based on a name and not on a structure such that the common interface of different data structures is defined by the means of a trait, and this trait is implemented for the different structures.

Listing 5.6: Indirection on the implementation in Rust Rust

```
trait ImplementationAccess {
    type Output: ?Sized;
    fn at(&mut self, p: Point2d) -> Option<&mut Self::Output>;
}

struct Indirect2d<T> {
    /* ... */
    access: Box<dyn ImplementationAccess<Output = T>>
}

impl<T> ImageInterface for Indirect2d<T> {
    /* ... */
    fn at(&mut self, p: Point2d) -> Option<&mut Self::Output> {
        self.access.at(p)
    }
}
```

Thus, the implementation of a 2D buffer, with either dynamically or statically typed values, relies on the separation of the interface and the implementation details in the same way as their C++ version, and are displayed in Figure 5.3. In this figure, the two data structures contain the fundamental information described in the previous section, but the values are stored in a buffer of `u8` values for the dynamically typed image. This has no impact on the dynamism of the type, this value type being chosen as its element size in byte is 1 to allow type-erasure. About the `ImageInterface`, the type of the values returned by the pixel value accessor is not necessarily a type whose size is known at compile-time: this allows for the dynamic image structure to return a value whose size is dynamic. In practice, instead of returning a pointer, a reference to a slice of `u8` values is returned for dynamic images, this one containing the pixel value at a given position. This is due to the fact that Rust pointers have a different semantic than object, as in C++, and this makes them difficult to use in the implementation of the trait.

The indirect access to the implementation of an image is based on the usage of a trait object. Trait objects follow a similar idea as type erasure for runtime polymorphism: the underlying type is known at runtime and it is valid if it has a given interface, which is defined by a trait in Rust. Thus, the idea is similar to the usage of the `std::function` in the C++ version of the indirect access of the implementation. The Rust version of the indirect access to the implementation

Listing 5.7: Elementwise operations algorithm in Rust

Rust

```rust
// Statically typed version
fn static_elw_op<T, Op>(img1: &Buffer2d<T>, img2: &Buffer2d<T>,
                        out: &mut Buffer2d<T>, mut op: Op)
where:
    T: Copy
    Op: FnMut(&T, &T, &mut T)
{
    generic_elementwise_op(img1, img2, out, op);
}

// Dynamically typed version
fn dyn_elw_op(img1: &Buffer2dAny, img2: &Buffer2dAny,
              out: &mut Buffer2dAny,
              op: Box<dyn FnMut(&[u8], &[u8], &mut [u8])>)
{
    generic_elementwise_op(img1, img2, out, op)
}
```

of an image is illustrated in Listing 5.6 in the case of a statically typed image. In this listing, the `ImplementationAccess` trait must be implemented by the structure handling the implementation of the object or by the implementation itself since in Rust, a trait can be implemented by any structure and thus does not require any wrapper. Then, the image structure, named `Indirect2d`, has a trait object named `access`. As a trait object is dynamically allocated, it is stored in a `Box` object, which handles the memory at runtime. Finally, the implementation of the `ImageInterface` trait for the image structure illustrates the access to the underlying implementation of the call to the function `at` from the `ImplementationAccess` trait implementation.

The usage of these image structures is similar to the C++ adaptation of generic algorithms to the static-dynamic genericity methodology. Thus, Listing 5.7 shows the usage of the Rust version of `generic_elementwise_op` from Listing 5.5 in the context of statically or dynamically typed image pixel values. The C++ version and the Rust version are very similar: in the static version, the function is parameterized. In the dynamic version, no parameters are used, and the operation is stored in a trait object such that it can be known at runtime. A major difference with the C++ version for the dynamic elementwise operation is the fact that the operation takes unsized slices of `u8` values. To get the correct type in order to perform the operation, pointers to the slice values

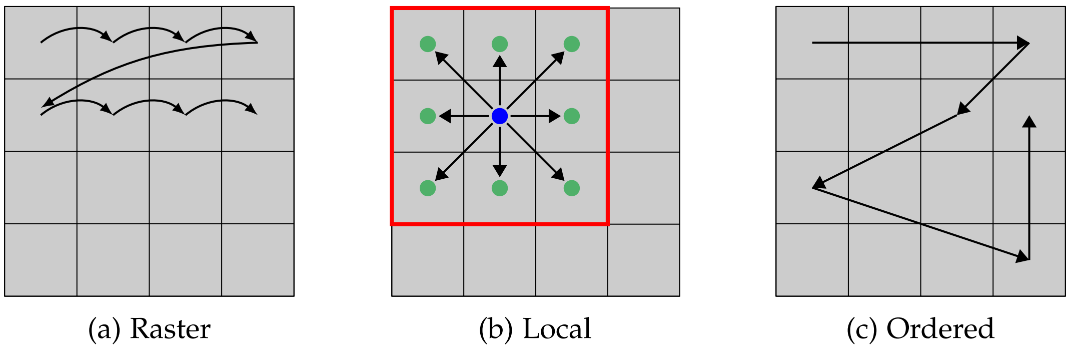


Figure 5.4: Algorithmic patterns for the experiments

are obtained and then converted into the correct static type by the means of the `std::mem::transmute` function for the operation to be performed . As the validity of such a conversion cannot be checked at compile-time, the operation is marked as `unsafe`, as well as any operation performed on pointers.

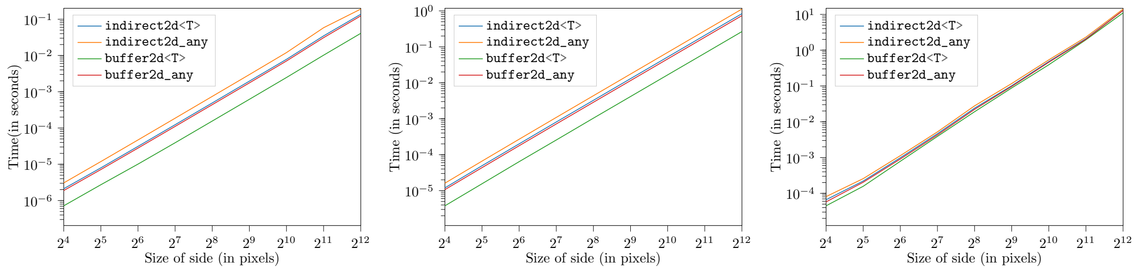
5.5 Experiments

In this section, the previous image models are compared when applied to different algorithmic schemes. To be consistent with the previous section of this chapter, these comparisons use algorithms and image models implemented in C++, but also in Rust to validate static-dynamic genericity.

5.5.1 Experimental set-up

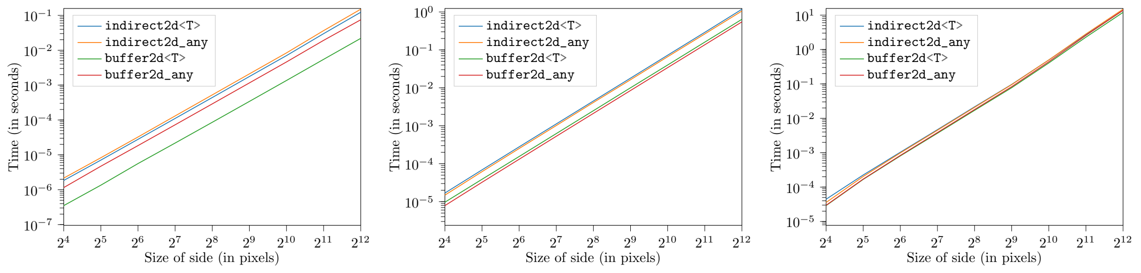
In order to evaluate the cost of the different image structures, three algorithmic schemes are used. These are illustrated in Figure 5.4. The *raster* scheme represents the traversal of the image in the same order the values are stored in memory. In this evaluation, the pixels are stored contiguously row by row. The *local* pattern traverses the image pixel by pixel in a similar manner as the raster pattern, but for each pixel, neighboring pixels are read to perform an operation. Finally, the *ordered* pattern uses a predefined order to pass through all the pixels of the image, and in a large majority of cases does not traverse them contiguously.

In practice, these patterns are implemented by means of three image processing algorithms. The raster pattern is implemented through the elementwise algorithm from Listing 5.5, and its operation is the maximum value between the pixel values at the same position in two images. The local pattern is implemented through a morphological dilation [188], and the neighborhood for



(a) Raster pattern (b) Local pattern (c) Ordered pattern

Figure 5.5: C++ benchmarks on the different algorithmic schemes



(a) Raster pattern (b) Local pattern (c) Ordered pattern

Figure 5.6: Rust benchmarks on the different algorithmic schemes

each pixel is represented by a square structuring element of size 1. Finally, the ordered pattern is implemented through Berger's max-tree algorithm [16], and the predefined order of traversal is defined by the descending order of all the pixel values.

The evaluation is performed on square images of dimension $s \times s$ with $s \in \{2^n | n \in [\![4..12]\!]\}$ randomly generated. The values are encoded as unsigned integers on 8 bits. The experiments have been performed on a Linux Debian 11 machine equipped with a processor Intel i7-3770, 3.40GHz. The C++ benchmarks have been run using the Google Benchmark [79] library from binary compiled with GCC 10.2.1 using the optimization flags *-o3*, *-ftree-vectorize*, *-mavx*, and *-unroll-loops*. The Rust benchmarks have been compiled with the Rustc compiler using the third optimization level (*-C opt-level=3*) and the measurements have been performed using the Criterion.rs library [51].

5.5.2 Evaluation

The results of the different benchmarks on the algorithm implementations in C++ and Rust are respectively displayed in Figures 5.5 and 5.6. For all the

	Statically Typed	Yes		No	
	Direct Access	Yes	No	Yes	No
C++	Raster	+0%	+176%	+183%	+367%
	Local	+0%	+208%	+174%	+283%
	Ordered	+0%	**+30%**	**+31%**	**+38%**
Rust	Raster	+0%	+388%	+251%	+574%
	Local	+0%	+61%	**-14%**	+66%
	Ordered	+0%	**+9%**	+6%	**+15%**

Table 5.3: Execution time overhead (in percentage) of the algorithmic schemes compared to the statically-typed with direct access image of side size of 4096

benchmarks, except for the dilation implemented in Rust (Figure 5.6b), the image structure with direct access to the implementation whose values are statically typed is the fastest. This is particularly true for the raster algorithmic pattern: in fact, it benefits from the fact that the different values are read in the same order as they are stored in memory, and thus chunks of values are kept in the memory cache of the processor, such that it allows a faster access compared with the access to the computer memory. Conversely, the ordered algorithmic scheme does not traverse the image in such order, except in the case the given ordering of the value is the raster order, which never happens for natural images, and thus for a large amount of access operation, *cache misses*, which are accesses to a value which is not stored in the processor memory cache, are very likely to happen.

Furthermore, in addition to the importance of the algorithmic scheme in the choice of the image structure to use, the difference in performance between the different image structures when applied to these algorithmic schemes is explained by the fact that some compiler optimizations cannot be performed due to the lack of knowledge at compile-time. First of all, in the context that the values or the implementation of the image are stored dynamically, the operations given as input arguments for the different algorithms are only known at runtime, and thus inlining cannot be performed. This results in indirections at runtime, as summarized in Table 5.2, and thus impacts the execution speed. In addition to such consequences to the dynamism, when the value type is unknown at compile-time, the compiler is unable to generate SIMD instructions. For the raster pattern, these are widely used for the statically typed buffer since the algorithmic scheme is convenient for such operations on contiguous values, which is not the case for the ordered pattern since the values are not read in raster order.

Table 5.3 shows the percentage of the overhead of the different algorithms

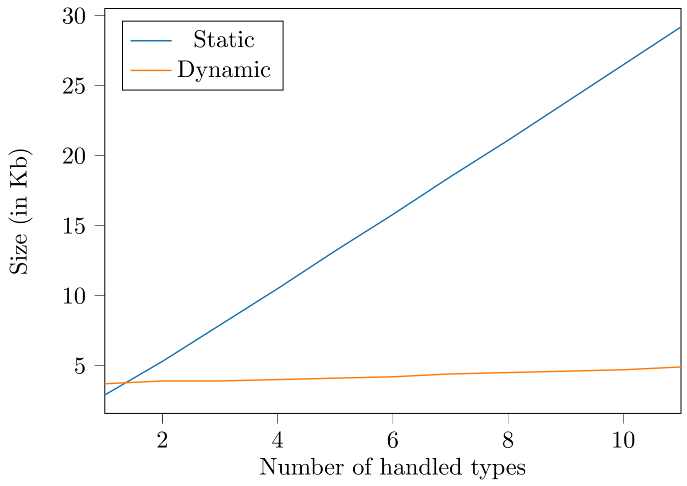


Figure 5.7: Evolution of the generated machine code amount related to the max-tree algorithm for statically and dynamically typed values

applied to the different image structures compared with the usage of a statically typed buffer of value with direct access to the implementation. To this aim, a large image is used (4096×4096) in order to have significant results. The values highlighted in red are the lowest overhead for an image model for each language. Whether in C++ or in Rust, the overhead of the ordered algorithmic scheme, and thus for the max-tree algorithm, is negligible compared with the other algorithmic schemes, except for the dilation in Rust. So for the max-tree algorithm, the knowledge of some static information is not of prime importance and it may be interesting to use dynamism to allow interactivity with a low impact on the performance of the construction algorithm.

Finally, a last experiment measures the amount of generated machine code depending on the number of handled types for one algorithm. C++ and Rust are languages whose genericity mechanism relies on a monomorphization process, such that for each combination of type parameters, specialized machine code is generated. The measurements are performed on the max-tree algorithm for the buffer image structure whose access is direct with statically and dynamically typed values. To this aim, the Bloaty [21] profiler is used, and the results are displayed in Figure 5.7. Both versions exhibit a linear increase with the addition of new image types. However, the quantity of new code generated in the dynamic version (100b/type) is 26 times lower than in the static version (2.6Kb/type) where a new full algorithm is instantiated. Therefore, the dynamic version prevents code bloat.

5.6 Conclusion

In this chapter, an approach to handle both static and dynamic genericity is presented for generic image processing algorithms. To this aim, four image structures are presented, with either static or dynamic information for the image pixel values type or the underlying implementation of the image. This approach is first presented in C++ and then extended to the Rust programming language. It is evaluated using different algorithmic schemes by the means of image processing algorithms implemented in C++ and Rust, and the results show that the importance of the knowledge of the type and implementation information depends on the algorithmic scheme.

From the results presented in this chapter, the loss of performance for the max-tree construction algorithm is negligible compared with the other algorithms. Thus, the knowledge at compile-time of the type and implementation information is not important. From this conclusion, in the next chapter, this construction algorithm for the max-tree representation is studied in depth, and its performance in the case of static or dynamic value types is compared with different existing implementations.

The pixel value type and the implementation of an image are either static or dynamic in this chapter. However, this is not the case for the domain of the image: it is only valid for a 2D image. The extension to nD images is not straightforward: the list of values for the coordinates can easily be extended to be dynamic, as well as the domain dimensions. Furthermore, to improve the runtime performance of such a process, a simple optimization, called Small Buffer Optimization (SBO), can be performed to allow the use of the stack for a small number of dimensions, or allocate on the heap for larger values. In practice, nD images whose number of dimensions n is larger than 4 ($3\text{D}+t$) are very rare such that dynamic allocations may be performed with $n > 4$. Finally, a unique representation for indexing will be studied to extend the image model to any image domain.

Chapter 6

Static-Dynamic Hierarchy Construction

In the previous chapter, an approach for the implementation of generic image processing algorithms is proposed with the particularity that the image pixel value types and its implementations may be either known at compile-time or runtime without the necessity to rewrite the whole algorithm. It is then evaluated on several algorithmic schemes, and we concluded that the gap between the knowledge at runtime or compile-time of the type information is negligible for the max-tree construction. We propose here an in-depth study of the implementation of this algorithm, and we compare this approach with other existing implementations.

6.1 Max-tree construction algorithms

Max-tree construction algorithms are widely studied in [34]. Their different algorithmic patterns are examined and their runtime speed are compared to result in a decision tree about the choice of the algorithm to use according to the nature of the environment and the quantization of the input image. In this section, these different algorithmic schemes are briefly reminded, and the choice of the construction algorithm used along this chapter is explained.

Max-tree construction algorithms may be divided into three categories: flooding, immersion, and merge-based algorithms, the latest category building several max-trees using either a flooding or an immersion algorithm, and then merging these different trees to obtain the whole max-tree. The flooding algorithm category is a two steps method consisting of first obtaining the pixel with the lowest value, used as the root of the tree, and then performing a flooding propagation to obtain the max-tree. Such an algorithm is first proposed by Salembier *et al.* [183], and later improved in [91] to remove the recursive call to the flooding procedure. Finally, Wilkinson [234] removed the constraint imposed by the hierarchical queue limiting the quantization of the image pixel values by replacing it with a priority queue and thus improving the generic capabilities of such max-tree algorithms.

Immersion algorithms are interesting for our purposes. From the decision tree in [34], they are recommended in different situations concerning the memory limitation of the system and the quantization of the image. They are based on the union-find algorithm [207] to obtain the upper connected components and model their inclusion relationship as a tree by a leaves to root construction. The first max-tree algorithm using this structure is proposed in [150] and is later improved by Berger *et al.* in [16]. The latest is the algorithm studied in this chapter since the union-find data structure is at the basis of several other tree construction algorithms, such as the α-tree or the hierarchical watersheds which can be built using the Kruskal algorithm [152].

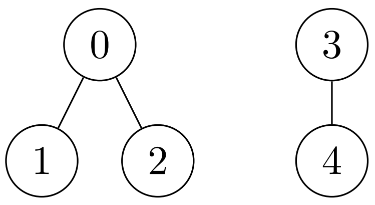
6.2 Overview of the Berger's max-tree algorithm

As explained previously, the max-tree construction algorithm proposed by Berger *et al.* [16] relies on a union-find data structure Q. This data structure represents a set of disjoint sets and is endowed with three operations:

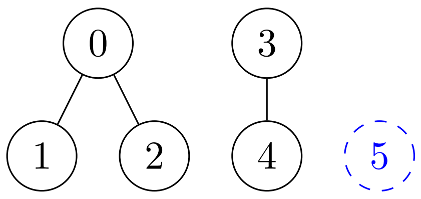
- **make-set**(Q, p): this operation creates a new set containing the element p.
- **find**(Q, p): this operation returns the representative element of the set containing p.
- **union**(Q, p, q): this operation merges the two sets represented by p and q.

There exists several ways to represent the set of disjoint sets, and a common means to do that is the use of a forest, which is a set of trees. Each tree represents a set: its leaves are singleton such that they represent a set composed of one element, and each internal node of a tree represents the merging (or union) of two sets to create a new set. The root node of a tree contains the representative element of a set and this value is returned by the **find** operation.

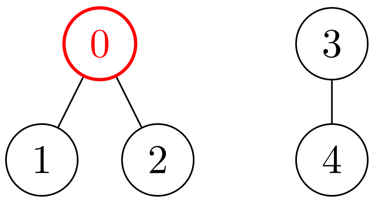
These concepts are illustrated in Figure 6.1. In this figure, the set of sets is illustrated by the forest containing two trees in Figure 6.1a. The **make-set** operation in Figure 6.1b creates a new tree in the internal forest structure, composed of only one canonical element. The **find** operation in Figure 6.1c



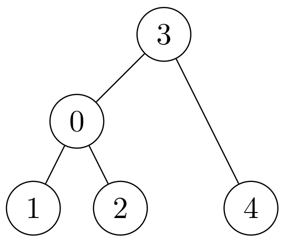
(a) Internal structure



(b) **make-set**$(Q, 5)$ operation



(c) **find**$(Q, 1)$ operation



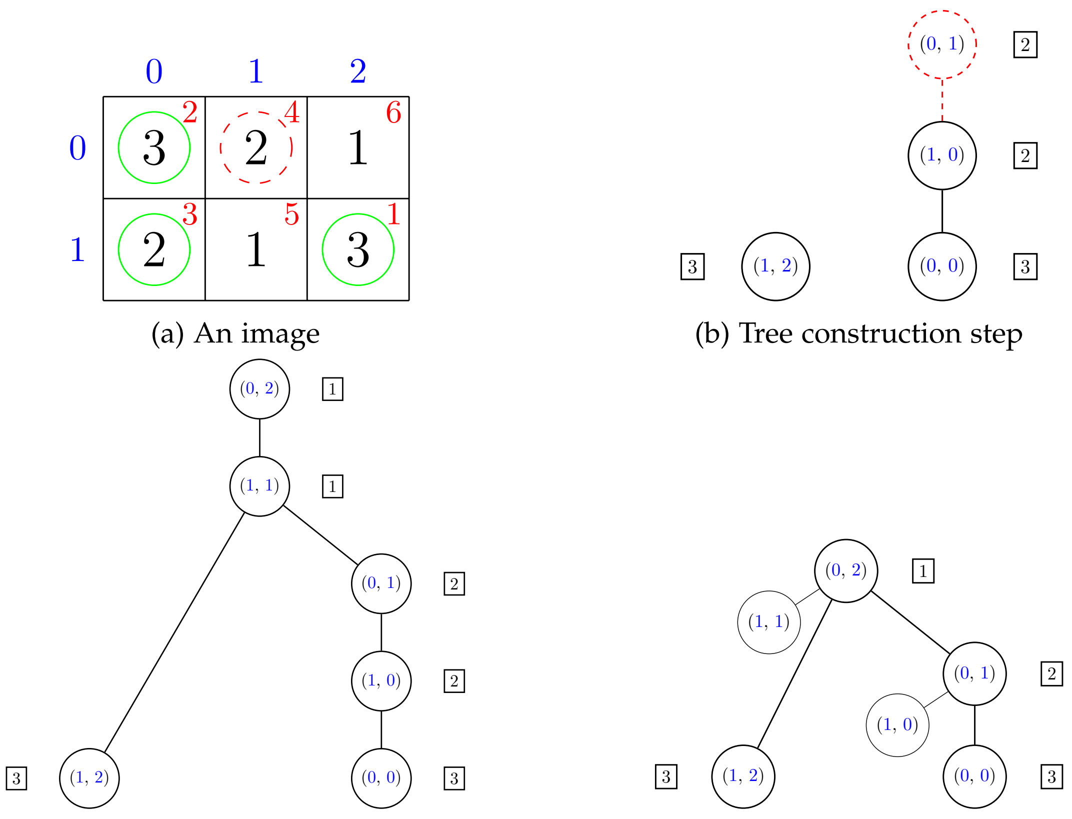
(d) **union**$(Q, 0, 3)$

Figure 6.1: Union-find operations

returns the representative element of the set a given element belongs to. This element is the root of the tree in the figure. Thus, if the element given as an argument is a leaf, the whole tree has to be traversed until the root. An optimization, named *path compression*, reduces the complexity of this operation and is discussed later in this chapter. Finally, the **union** operation in Figure 6.1d merges two trees to form a new tree. In the figure, the root node of one of the two trees becomes the root node of the new tree forming the new set.

Berger's algorithm uses this structure to compute the max-tree. First of all, it sorts decreasingly the image pixels by their value. Then, it traverses these sorted pixels and creates a new node of the tree for each of them (**make-set** operation). The adjacent pixels are then looked at, and if some of them are in a set of the union-find structure and not in the same set as the current pixel, then the two sets are merged (**union** operation). When all the image pixels are processed, the resulting forest is composed of one tree which is the max-tree. This tree is simplified by a post-processing operation called *canonicalization* resulting in one representative node per component.

This algorithm is illustrated in Figure 6.2. In this figure, the max-tree is built on the image in Figure 6.2a in which the blue numbers denote the coordinates (row, column), the red numbers are the rank of an image pixel after the sorting operation and the black numbers are the pixel intensities. To build the max-tree, a 4-adjacency relationship is used. In Figure 6.2b, a step from the max-tree construction algorithm is displayed. The current pixel at this step is enclosed by a red dashed circle in Figure 6.2a, and the pixels circled in green are the pixels



(a) An image (b) Tree construction step

(c) Max-tree (before canonicalization) (d) Max-tree (canonized)

Figure 6.2: Max-tree construction

that are already visited, and thus already in the internal forest of the union-find structure. In this construction step, the current pixel is merged with an existing set. The resulting tree from the traversal of the sorted pixels is displayed in Figure 6.2c. In this tree, some consecutive nodes in their path to the root belong to the same upper connected component. In that case, the tree is less simple to manipulate. The canonicalization removes such consecutive nodes such that only one node per level is representative in its path to the root, as illustrated in Figure 6.2d, the other node at the same level just pointing to the pixels of the component represented by such a representative node.

The max-tree algorithm is displayed formally in Algorithm 6.1. In this algorithm, the union-find operations are highlighted and labeled. The union-find internal forest structure is represented as a function named *zpar* which is implemented as an image. However, this function does not represent the max-tree structure due to the path compression optimization, which flattens the trees in the forest in order to have fast access to the representative element of a set. To this aim, the value of $zpar$ at each node of the forest is updated to the root node of the corresponding tree when necessary during the **find** operation, implemented in the FIND_ROOT function in Algorithm 6.1. Thus, another function is required to encode the whole max-tree, and it corresponds to the $parent$ function in the algorithm.

6.3 Existing implementations of Berger's max-tree

There exist several implementations of the max-tree construction using Berger's algorithm. In this section, we study some of them and we provide a comparison between all of them. The first max-tree construction is the one provided by the authors of [16] which is implemented in the Milena library [122]. As it is the reference implementation, it strictly follows the algorithm given in Algorithm 6.1. It is implemented in C++ and is highly generic: it accepts any kind of image, whatever its value set or its domain, and builds its max-tree.

The second implementation is provided by the *trees-lib* library [23], which contains the implementations of several hierarchical representations of images by the use of the OpenCV library [27] and is used as a reference implementation for [24]. The max-tree algorithm is implemented as proposed in [16]. However, in this library, an inclusion tree is represented by a linked list whose nodes are dynamically allocated and linked to each other according to the parenthood relationship. Such a data structure is costly in terms of runtime performance as each node is allocated one by one compared to the *parent* mapping which requires only one memory allocation. Furthermore, this mapping is implemented for the need of the max-tree construction algorithm as a vector of vectors. Consequently, the elements are not contiguous in memory and an allocation is performed for each line of the *parent* mapping in addition to one allocation of the vector storing all the lines. Finally, an overhead is added by the conversion from the *parent* mapping to the linked list representation.

The max-tree construction is implemented in Scikit-Image [230] using the Cython transpiler [14] which transforms code implemented in Cython, a language mixing Python and C functionalities, into an implementation in C and makes it available in Python by means of the CPython API. Unlike the proposed algorithm in [16] and the previously discussed implementations, the construction provided by Scikit-Image does not make use of the path compression optimization. This is justified in the source code comments by the fact that the recursion performed by the FIND_ROOT function adds an overhead in terms of performance which is not the case with an iterative version despite its higher complexity. The provided max-tree construction function takes as input any image defined on a hyperrectangle, implemented as a NumPy array [85], and returns the *parent* mapping as well as the list of ordered pixel positions.

The latest implementation discussed in this section is the one provided by the Higra [164] library. The function constructs the max-tree of any vertex-weighted graph. As in [16], this implementation uses path compression as an optimization for the **find** operation but also uses a union-by-rank, which avoids the creation of a degenerated tree and guaranties an $O(n \log n)$ complexity for the union-find when used along path compression. As graphs may be used to represent the adjacency relationship between the image pixels, such an implementation can be used on rectangular images, but also on simplicial complexes as discussed in Chapter 3.

Scikit-Image and Higra provide a Python interface to their functionalities. They make their respective max-tree construction usable with a limited set of types (real numeric value types with a quantization lower than or equal to 64 bits), but this one covers a whole majority of cases when the image is univariate. To this aim, Higra specializes its templated functions building the max-tree and Scikit-Image uses *fused types* in Cython which are set of accepted types for a function used for generic programming in Cython. The latter generates specialized C code for each type. On the other hand, trees-lib is based on the OpenCV C++ API and does not provide any interface in a dynamic language but allows some interactivity due to the image structure with dynamically typed values provided by OpenCV. However, with the default max-tree construction function being specialized with a comparison of integer values, floating points are difficult to handle without using the templated version on the comparator. Finally, Milena is fully static, with a templated implementation of the max-tree, and does not provide any dynamic version of its construction.

6.4 Static-dynamic implementation

In this section, the implementation of the max-tree using the methodology explained in the previous chapter is described. In Algorithm 6.1, the parts of the algorithm requiring the knowledge of the image pixel value type are highlighted in red. Precisely, these two parts are the sorting of the pixels and the canonicalization of the tree. These two steps from the max-tree construction algorithm require knowing the ordering relationship between the image pixel values. Conversely, the union-find operations do not require manipulating the values of the image, and thus their implementation can be fully static, meaning that the type of every expression is known at compile time, and underlying optimizations can be performed by the compiler. Finally, some other information, not related to the value type of an image, may be known either at runtime or at compile-time. This is for example the case for the

Listing 6.1: Max-tree function prototypes C++

```cpp
// Generic version
template <class I, class N, class C>
std::pair<buffer2d<point2d>, std::vector<point2d>>
maxtree(I img, N nbh, C comp);

// Interfaces
template <class T, class N, class C>
std::pair<buffer2d<point2d>, std::vector<point2d>>
maxtree_static(buffer2d<T> img, N nbh, C comp);

std::pair<buffer2d<point2d>, std::vector<point2d>>
maxtree_dynamic(buffer2d_any img, c4c8_t nbh,
                std::function<int(const void*, const void*)> comp);
```

adjacency relationship of the image pixels whose usage is highlighted in blue in Algorithm 6.1: for 2D images, this latest is not unique, as explained in Chapter 3, and the number of neighbors may varies depending on the chosen one.

The identification of the steps of the algorithm that require knowing the pixel value type of the image enables to implement the max-tree construction with the static-dynamic genericity methodology. In order to enable the max-tree to be implemented, we consider a 2D image as input for the algorithm. However, the case the image may have different implementations is handled. The prototypes of the max-tree functions are displayed in Listing 6.1. The `maxtree` function is a templated entity, but it only affects the arguments of the function. These three arguments are the following: the input image, the adjacency relationship of the image pixels implemented as a window as explained in Chapter 3, and a comparison operation. Indeed, this comparison function is a three-way comparison defined for two values a and b by:

$$C(a,b) = \begin{cases} -1 & \text{if } a < b \\ 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases}$$

The two interfaces displayed below the generic `maxtree` function handle the different cases of typing for the image pixel values and call the generic max-tree construction function. Furthermore, it takes into account the fact that the knowledge of the adjacency relationship is known at compile-time or runtime: the window stores a set of offsets that are applied to a given point and returns the set of adjacent points. When this information is known at runtime, the window is endowed with additional dynamic information indicating which kind of adjacency is used. For example, the `c4c8_t` type in the dynamic version of the max-tree stores in an array the offsets of the 8-adjacency relationship such that the first four offsets are the horizontal and vertical ones and the last four ones are the diagonal offsets. Then, if the dynamic information are related to the 4-adjacency relationship, only the first four offsets are applied to the current point. Finally, some operations may be performed before the usage of the generic version and are described in the following.

Using a three-way operator for the max-tree algorithm allows using it both for the sorting step and the canonicalization step, which requires checking the equality of two image pixel values to simplify the tree. This operation is known at compile-time in the case the image pixel values are statically typed, or at runtime by means of a `std::function` in the case the type of the pixel values is dynamic. The resulting output type of the max-tree construction function corresponds to the sorted pixel sequence and the parent image from Algorithm 6.1. Indeed, these are not templated: they do not depend on the input image implementation or values type.

In the previous chapter, four image structures are presented for static-dynamic genericity. By taking into account the knowledge acquired previously, two strategies based on the previous models are presented in this chapter. These rely on the necessity to go from dynamic typing to static typing to reduce the gap between the different structures while keeping generic abilities. Furthermore, due to the design of generic algorithms presented in the previous chapter, their adaptation requires few modifications of the max-tree construction implementation.

The first strategy is based on *projections*. From the C++ 20 standard [96], a projection is a *transformation that an algorithm applies before inspecting the values of the elements* of a container. Such operations are widely used with range algorithms in C++, which may apply such a transformation before the operation is performed on a value. It has to be noted that the default projection is the *identity* transformation that, for an element value, returns the same value. Applied to the max-tree, projections are added in two parts of the max-tree construction algorithm: the sorting step and the canonicalization. For these two parts, the projection is applied on the image pixel values in order to convert the values from type-erased ones to statically typed ones before performing the three-way comparison operation. In this case, the conversion has not to be performed in the comparison operation but in a dedicated operation such that the argument types of the comparison may be known at compile-time.

The second strategy consists in creating a new image with a new pixel

Value type	**Access policy**	**Strategy**	**# Indirection**
Static Type	Direct		0
		Projection	0
		Conversion	0
	Indirect		2
		Projection	2
		Conversion	1
Dynamic Type	Direct		2
		Projection	2
		Conversion	1
	Indirect		4
		Projection	4
		Conversion	2

Table 6.1: Indirection amount for each image model

value type, eventually with a quantization large enough to handle several value types, and then copying all the elements of the current image in this new one, performing a *conversion* of the pixel values if required. This strategy has several advantages, but also some drawbacks: first, the pixel value type and implementation of the image given as input to the max-tree construction algorithm are known at compile-time, allowing optimizations to be performed by the compiler. Then, only one version of the templated function is generated as it takes only one kind of image encompassing a large majority of value types. However, it requires allocating a new image and copying all of its values so that this process involves a complete traversal of the input image. Consequently, this adds an overhead in terms of runtime speed and memory. This memory overhead may be reduced by applying a merge-based algorithm in conjunction with Berger's construction on blocks of image whose size is fixed.

To summarize the different strategies to build the max-tree with the knowledge of some information either at runtime or compile-time, Table 6.1 lists the number of indirections according to the image model. These are counted such that an indirection is required if an operation cannot be inlined by the compiler, with a `std::function` object for example, and the fact that the sorting and the canonicalization steps require to look for the image pixel values is taken into account. First of all, it has to be reminded that the access to the implementation when unknown at compile-time requires one indirection per pixel value access, and another one when this value has its type only known at runtime. Furthermore, in the max-tree construction algorithm, each pixel value is looked at twice. For the *conversion* strategy, as the image is copied in a new one with a static value type and direct access to the buffer, there is no indirection in the max-tree construction algorithm, but these are counted for the *convert and copy* procedure, which traverses the image once. The *projection* strategy has the same amount of indirection as the strategies studied in Chapter 5: the access to the implementation of the image does not change, and instead of handling the pixel value type in the operation, this is performed in the projection, which cannot be known at compile-time in the case of dynamically typed pixel values.

6.5 Evaluation

To perform the different experiments, the experimental protocol is the same as in Chapter 5. Images have a square domain of dimensions $s \times s$ with $s \in \{2^n | n \in [\![4..12]\!]\}$. The values are randomly generated and are encoded as unsigned integers on 8 bits such that $\mathcal{V} \subseteq [\![0..255]\!]$. Except when using the Scikit-Image [230] library, the different benchmarks are performed with binaries compiled using GCC 10.2.1 with the following flags *-o3*, *-ftree-vectorize*, *-mavx*, and *-unroll-loops*. To perform the measurements, the Google Benchmark [79] is used. Finally, the experiments have been run on a Linux Debian 11 machine equipped with a processor Intel i7-3770, 3.40GHz. The benchmarks of the Scikit-Image's max-tree construction were performed within the CPython 3.9.2 interpreter with Scikit-Image 0.19.2. To this aim, the `time` module is used.

We have implemented the Berger's max-tree construction algorithm in C++ as a templated function and we provide interface to the correct image structure in the same way as in Listing 6.1. It has to be noted that our implementation of the max-tree construction utilizes a union-by-rank [207] in addition to the path compression used by Berger *et al.* [16] in the union-find data structure as it guaranties an $O(n \log n)$ complexity.

First, the results from Table 6.2 correspond to the overhead of max-tree construction when applied to the different structures compared with the application of this algorithm on the `buffer2d<T>` image structure for each side size ($2^n \times 2^n$) of the image. When using the *projection* and the *conversion* strategies, the values are transformed into unsigned values encoded on 64 bits. Thus, in the case the image values are typed-erased, they are first converted into their static type and then projected into the type used by these two strategies. The first part of the table describes the overhead obtained by the use of the image structures defined in Chapter 5. The second part displays the results of the usage of the projection strategy. We can observe in this table that this strategy has a greater overhead than the usage of the conversion in the comparison operation except in the case where the image value type and implementation are statically known. This is because the projections perform two conversions in

n		4	5	6	7	8	9	10	11	12
`buffer2d<T>`		0	0	0	0	0	0	0	0	0
`indirect2d<T>`		48	33	24	22	22	15	24	23	30
`buffer2d_any`		52	32	27	27	27	20	27	25	31
`indirect2d_any`		60	40	36	36	37	30	35	33	38
`buffer2d<T>`	Projection	24	7	2	-3	-4	-19	3	3	3
`indirect2d<T>`	Projection	58	40	31	29	29	20	29	28	34
`buffer2d_any`	Projection	63	46	38	36	37	29	35	36	40
`indirect2d_any`	Projection	68	51	45	43	44	39	42	40	45
`buffer2d<T>`	Conversion	5	1	-6	-10	-9	-15	6	6	9
`indirect2d<T>`	Conversion	10	-1	-6	-10	-8	-14	7	7	9
`buffer2d_any`	Conversion	18	2	-1	-7	-6	-17	8	8	10
`indirect2d_any`	Conversion	12	3	-2	-6	-6	-13	10	8	10

Table 6.2: Overhead (in percentage) of each image structure of dimension $2^n \times 2^n$ applied to the max-tree computation algorithm compared with the statically typed buffer with direct access.

our experiments at each call that are difficult to optimize in the case the image value type is dynamically known as the projector is stored in a `std::function` object. Furthermore, the usage of this projection in a dynamic context results in an indirection. Finally, the conversion strategy has a low overhead compared to the other strategies and for the image side sizes ranging from 64 to 512, this overhead is negative. These negative overheads have been investigated but there were no causes that explained this issue. Further investigations will be conducted on the analysis of the assembly code, profiling, and caching in the CPU. Finally, the benchmarks will be run on different computer configurations to observe if it has an impact on the results.

Figure 6.3 illustrates the outcome of the comparison of our implementation with the existing ones described in Section 6.3. As observed in Table 6.2, the application of the different image structures proposed in Chapter 5 to our max-tree implementation produces similar results. Furthermore, these performance are likewise akin to the ones from Higra. This is explained by the fact that the implementations are similar: they both use path compression and union-by-rank. On the other hand, the construction algorithm is implemented in *trees-lib* without union-by-rank and its tree structure is represented as a linked list, which requires an allocation for each node. These two factors are the underlying cause of the runtime performance overhead compared with our implementation. Finally, the implementation of the construction in Scikit-Image

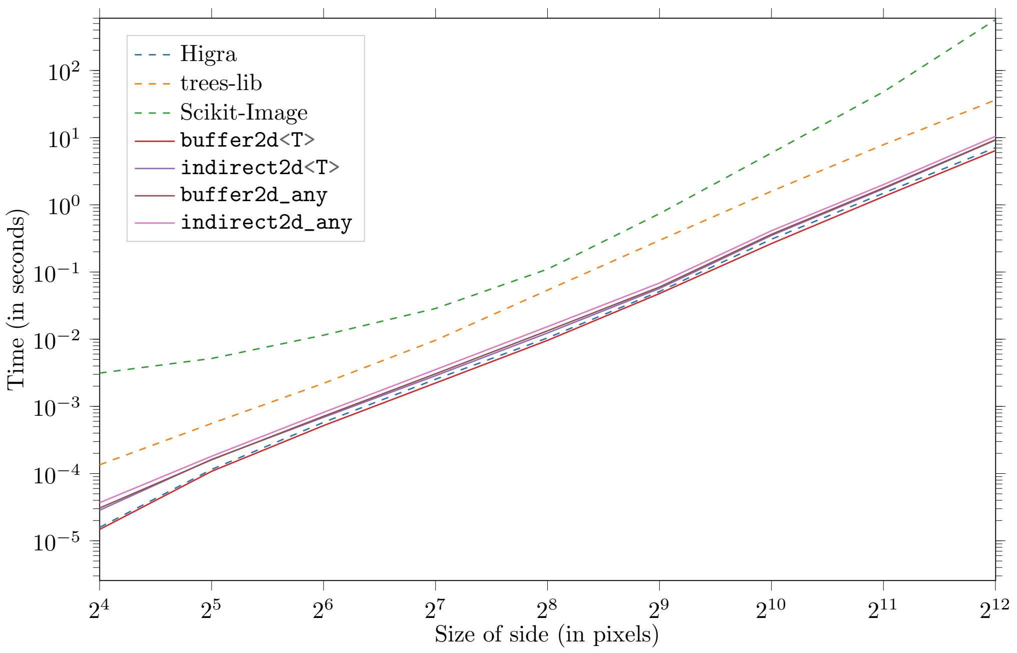


Figure 6.3: Benchmark on different max-tree implementations

uses none of the union-find optimizations cited previously such that it results in the worst outcomes in terms of runtime speed.

6.6 Conclusion and perspectives

In this chapter, we studied different possibilities to implement the max-tree algorithm using static-dynamic genericity. We used the results obtained in the previous chapter to analyze and extend our methodology in order to efficiently build the max-tree. We thus evaluated the cost of indirections for this particular case. Furthermore, we proposed to use two new strategies based on the image structures studied in the previous chapter relying on the coercion of the image pixel values, and we show that one of them reduces the cost of the dynamism. In order to improve the efficiency and reduce the gap between the application of this algorithm on the static and dynamic image structures. To this aim, it would be interesting to parallelize the algorithm, and more particularly to use a merging-based construction based on an immersion algorithm.

We studied an immersion-based max-tree algorithm relying on the union-find data structure. However, the max-tree algorithm is not the only hierarchical representation based on such an algorithmic pattern. For example, the α-tree or hierarchical watersheds may be built using the Kruskal algorithm [152], which is also based on union-find and has a similar pattern, but performs its computation on edges from undirected graphs instead of the pixel values. Indeed, the max-tree is linked with those hierarchies [49] as the α-tree is the max-tree of the minimum spanning tree (MST) of a graph. As a result, the same conclusion holds for the MST computation as well as other hierarchies based on the union-find algorithm.

Algorithm 6.1 Berger's max-tree algorithm

```
function FIND_ROOT($x, zpar$)
    $r \leftarrow x$
    while $zpar(r) \neq r$ do
        $r \leftarrow zpar(r)$
    end while
    while $zpar(x) \neq x$ do
        $t \leftarrow x$
        $x \leftarrow zpar(x)$
        $zpar(t) \leftarrow r$
    end while
    return $r$
end function

function CANONIZE($f, parent, R$)
    for $p \in R$ (reverse order) do
        $q \leftarrow parent(p)$
        if $f(parent(q)) = f(q)$ then
            $parent(p) \leftarrow parent(q)$
        end if
    end for
end function

function COMPUTE_MAXTREE($f$)
    for $p \in \Omega$ do
        $zpar(p) \leftarrow undef$
    end for
    $R \leftarrow$ REVERSE_SORT(f)                                  ▷ Requires image value type
    for $p \in R$ do
        $parent(p) \leftarrow p$                                       ▷ make-set
        $zpar(p) \leftarrow p$
        for $n \in \mathcal{N}(p)$ such that $zpar(n) \neq undef$ do   ▷ Requires adjacency knowledge
            $r \leftarrow$ FIND_ROOT($n, zpar$)                          ▷ find
            if $r \neq p$ then
                $parent(r) \leftarrow p$                                 ▷ union
                $zpar(r) \leftarrow p$
            end if
        end for
    end for
    $parent \leftarrow$ CANONIZE($f, parent, R$)                   ▷ Requires image value type
    return $R, parent$
end function
```

Part III

Applications in Image Processing

Chapter 7

Noise Level Estimation using Hierarchical Representations

In this chapter, hierarchical representations are used to estimate the noise level function of an image, in order to detect the statistical nature of the noise and estimate the parameters of such noise. In the past, we proposed to use the tree of shapes to adapt to the content of an image a method using square patches statistics to estimate this function. However, this method is limited to grayscale images due to the tools it uses. In this chapter, we propose to extend this method to color images, and secondly, we propose some modifications to improve its results. The results demonstrate that some modifications improve the whole pipeline while others, such as the use of the whole hierarchy, are similar in terms of the precision of the estimation.

7.1 Context and motivation

Noise level estimation is an important step of several image processing pipelines. The knowledge of the noise parameters as well as its statistical nature improves the efficiency of the treatments of an image. Denoising is the most obvious process requiring such information and is a crucial step in several applications such as the image acquisition step [98]. Denoising algorithms can be divided into two categories: non-blind denoising algorithms, in which the noise level is assumed to be known, and blind denoising algorithms, in which this parameter is unknown. In the case of blind denoising, the noise level is usually estimated first, and given as information to a non-blind denoising algorithm.

In [128], the authors first estimate the noise and then provide the results of the estimation to the BM3D denoising algorithm [52]. Sutour *et al.* [204] propose a similar methodology using the non-local means algorithm [28] (NL-means), but unlike the method proposed in [128], they take into account the fact that the noise may be dependent to the image content. The usage of convolutional neural networks (CNN) also relies on such pipelines, as is the case for FFDNet [244], a non-blind denoising network, which results in better denoised images in terms of PSNR when a noise level estimation method [236] is used beforehand.

Denoising is not the only application whose performance is improved by the knowledge of the noise level. In [127], the authors use the noise level information to design an adaptive Canny edge detector. In [62], the noise information is used to obtain a denoising method that preserves the edges of the image for segmentation purposes. In [70], it is required as a parameter in a Markov network used for super-resolution. Finally, in [229], the noise parameters are used in a compression-denoising process.

With that number of applications, many noise level estimation methods were proposed. While some estimation methods are based on several images [88], estimating the noise level from a single image remains a difficult problem. In [128], the authors propose to estimate the noise level in an image from noisy patches. Variance stabilization, which approximates a Gaussian distribution from a Poisson distribution, is used in [147] to estimate a Poissonian-Gaussian noise model. Finally, the method described in [236] uses a CNN to separate the noise from the image content in order to estimate the noise parameters.

The previously listed noise estimation methods suffer from two major drawbacks: they have a strong hypothesis on the statistical nature of the noise, and they consider the noise parameters are constant all over the image, even when the noise is signal dependent. In [127], the authors propose to model the noise level as a function related to the pixel intensities from an image. A similar model is used in [65] to a signal-dependent noise (Poisson noise) with a signal-independent noise (Gaussian noise). Deep learning-based noise level estimation methods have also been developed for signal-dependent noise by setting the noise level map values to pixelwise parameters [206, 243] instead of using only one value [236]. Furthermore, Beaurepaire *et al.* first identify the statistical nature of the noise by analyzing statistics on patches of the image before estimating its parameters. However, their method is not able to represent a mixed noise. Sutour *et al.* [204] combine these two advantages: they estimate the noise level as a quadratic function, in which each coefficient is related to the noise parameter of a particular noise statistic, and thus allows to represent a mixed noise, but also allowing to identify its nature. Finally, Esteban *et al.* [63] extend their method to take into account the content of an image, the method from [204] being based on squared patches.

The latter two methods to estimate the noise level fulfill the need to identify the nature of the noise and to represent correctly signal-dependent noise simultaneously. Furthermore, they allow to represent a mixed noise due to the use of a quadratic function as a noise level function. However, they have a major drawback: they are limited to grayscale images because of the rank-based correlation coefficient and the tree of shapes that rely on the rank of the pixel values, which is not natural where these are multivariate, which is a major issue in terms of genericity. Thus, in this chapter, the cause of these constraints is identified, and an extension to color image is presented.

7.2 Grayscale noise level function estimation

In this section, the noise level estimation procedure we proposed in [63] is recalled. As a reminder, this noise level estimation adapts the method proposed in [204] to the content of an image.

7.2.1 The noise level function

To model the noise in an image, the estimation is relying on a function, called the noise level function (NLF) [127]. The application of such function to any pixel value of the image returns the variance of the noise corrupting this pixel, such that $\sigma^2 = \text{NLF}(x)$, for $x \in \mathcal{V}$ a pixel value in the image, and σ^2 the associated noise variance. From the state of the art provided in the previous section, the NLF must provide two major pieces of information: the statistical nature of the noise, and the parameters associated with this distribution. The NLF provided by Sutour *et al.* [204] fulfills these requirements but also enables the modelization of a mixed noise. It is defined as a quadratic function by:

$$\sigma^2 = \text{NLF}_{(a,b,c)}(x) = ax^2 + bx + c \tag{7.1}$$

with $(a, b, c) \in \mathbb{R}^3$ being three parameters modeling different statistical distributions of the noise, and $x \in \mathcal{V}$ a pixel value. In this function, the parameters a, b and c are respectively related to the multiplicative noise, the Poisson noise, and the additive white Gaussian noise. Thus, this function represents an additive white Gaussian noise with $(a, b) = (0, 0)$, a Poisson noise with $(a, c) = (0, 0)$, or a multiplicative noise with $(b, c) = (0, 0)$. Furthermore, it can also represent a mixed noise such as the Poisson-Gaussian noise with $a = 0$ and $(b, c) \neq (0, 0)$. The objective of the noise level estimation is thus the estimation of the three parameters a, b and c.

7.2.2 Mumford-Shah minimization using the tree of shapes

The adaptation to the content of the image of the method presented in [204] consists in replacing the square patches with some regions of the noisy image. In Chapter 4, different hierarchical representations of images are proposed for different purposes. Among them, the tree of shapes, which encodes the inclusion relationship of the connected components of an image, and by definition the inclusion relationship of its level lines, is a contrast invariant representation. This representation is used for segmentation in our noise level estimation pipeline.

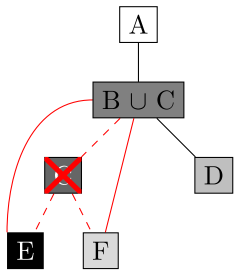
Many segmentation algorithms rely on an energy minimization procedure. An energy that is well-established for segmentation purposes is the Mumford-Shah functional [142]. This energy is used in accordance with various hierarchical representations for segmentation purposes. In [83], it serves as a criterion to obtain optimal cuts from partitioning hierarchies according to the energy, and a new partitioning hierarchy, called *persistent hierarchy*, is defined as the stack of the different optimal cuts. In [9], the minimization procedure is applied to the tree of shapes in an iterative procedure but requires to compute the shapes that are removed in the current iteration as well as the shapes that are candidate for the next one. In [240], Xu *et al.* propose an efficient algorithm based on the work of [9] to select meaningful level lines from the tree of shapes. This algorithm is the one used to select regions of interest in the noise level estimation process since its authors demonstrate its robustness to noise in addition to its efficient execution speed.

Since it is hard to minimize the classical Mumford-Shah functional, the methods used in [83, 9, 240] are based on a simplified version of this functional, the piecewise-constant Mumford-Shah functional. Applied on the tree of shapes, it is defined by:

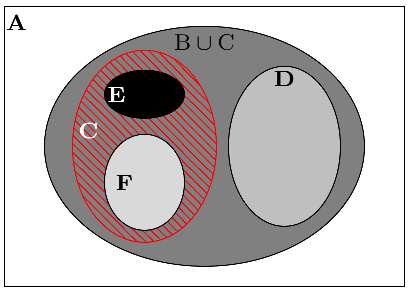
$$E(\mathcal{T}_\circ') = \sum_{\mathcal{C}_i \in \mathcal{T}_\circ'} \sum_{\substack{p \in \mathcal{C} \\ p \notin Children(\mathcal{C})}} \|f(p) - \tilde{f}(p)\|_2^2 + \lambda |\partial \mathcal{C}_i| \tag{7.2}$$

with $\mathcal{T}_\circ'$ the tree of shapes constrained by the functional, $\tilde{f}$ a piecewise constant function resulting in the mean value of the pixels from the region containing a point p, $|\partial \mathcal{C}_i|$ is the length of the level lines surrounding a shape $\mathcal{C}_i$, and λ is a parameter of this functional.

The algorithm proposed in [240] follows the following steps which are repeated until no shapes are removed: first, it sorts the shapes of the tree in ascending order of the average gradient of their contours, then it traverses these sorted shapes and removes the shapes whose variational criterion is invalidated. Let $R_{\mathcal{C}_i}$ be the connected component represented by a shape $\mathcal{C}_i$ from $\mathcal{T}_\circ'$ without its holes filled, $\text{par}(\mathcal{C}_i)$ be the parent of $\mathcal{C}_i$ in $\mathcal{T}_\circ'$, and $\xi(R) = \sum_{p \in R} \|f(p) - \tilde{f}\|_2^2$.

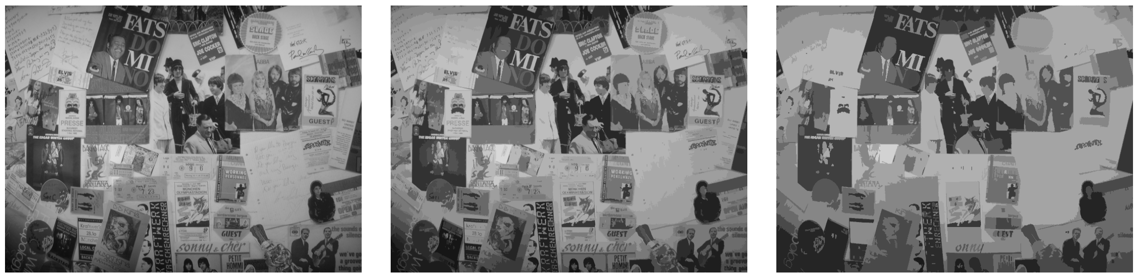


(a) Simplification $\mathcal{T}_\circ'$ of $\mathcal{T}_\circ$



(b) Reconstruction of $\mathcal{T}_\circ'$

Figure 7.1: Simplification of the ToS $\mathcal{T}_\circ$ from Figure 4.5b



(a) An image (b) $\lambda = 500$ (c) $\lambda = 5000$

Figure 7.2: Simplifications on a natural image with different parameter values

The variational criterion is defined for a shape $\mathcal{C}_i$ by:

$$\xi(R_{\mathcal{C}_i}) + \xi(R_{\text{par}(\mathcal{C}_i)}) - \xi(R_{\mathcal{C}_i} \cup R_{\text{par}(\mathcal{C}_i)}) - \lambda|\partial\mathcal{C}_i| \geq 0 \tag{7.3}$$

The case where the variational criterion is invalidated means that the removal of a shape decreases the value of the functional on the whole tree. In practice, one iteration is sufficient. This process is illustrated in Figure 7.1, in which the component C is removed, thus the components B and C form a new component as the union of both.

Finally, the illustrations in Figure 7.2 display the application of the simplification procedure on a tree built from a natural image using different values of the parameter λ. In the first simplification in Figure 7.2b, the parameter value is small: the image is simplified a little, compared to the image in Figure 7.2c, in which most of the texture of the image has disappeared. Thus, it has to be noted that the parameter λ controls the level of simplification: the greater its value, the most simplified the tree and thus its reconstruction.

7.2.3 The Kendall's τ correlation coefficient

In [204], authors propose a detector of noisy patches by the mean of a statistical test. To this aim, two hypotheses are stated: the null hypothesis H_0 is related to the fact that the content of an image patch does not vary and is only composed of noise, and the alternative hypothesis H_1 states that the patch content exhibits variation due to the presence of texture. As the image is noisy, this statistical test is based on the correlation of two random variables X and Y, and if these are not correlated, so $\text{Corr}(x, y) = 0$, then the image content is constant and the image patch is only composed of noise.

The correlation coefficient used for the statistical test is the Kendall's τ. It is a rank correlation coefficient, which means its computation is not based on the values of the observation of X and Y, respectively denoted by x and y, but on their rank. The choice of using the rank instead of the values relies on the fact that the estimation aims at detecting the statistical nature of the noise, and such statistic parameters are required to use a value-based correlation coefficient.

The Kendall's τ rank correlation coefficient is based on the notion of *concordant pairs* and *discordant pairs*. Let x and y be two sequences of observations of size n of the two random variables X and Y. Let $x_i, x_j \in x$ and $y_i, y_j \in y$ with $i \in [\![0..n-1]\!]$. Two pairs (x_i, y_i) and (x_j, y_j) are said to be concordant if $x_i < x_j$ and $y_i < y_j$ (or $x_i > x_j$ and $y_i > y_j$). They are discordant if $x_i < x_j$ and $y_i > y_j$ (or $x_i > x_j$ and $y_i < y_j$).

With these definitions, the Kendall's τ rank correlation for two sequences of n pixels values x and y is defined by [103]:

$$\tau(x, y) = \frac{1}{n(n-1)} \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \text{sign}(x_i - x_j)\text{sign}(y_i - y_j) \tag{7.4}$$

The two sequences are perfectly correlated if there are only discordant or concordant pairs. However, it does not take into account the case of *tied pairs*, which are pairs being neither concordant nor discordant, with $x_i = x_j$ or $y_i = y_j$. Such pairs can exist in image processing applications since two pixels can have the same value. Thus, the Kendall's τ considering tied pairs is defined by [104]:

$$\tau(x, y) = \frac{n_c - n_d}{\sqrt{(n - n_x)(n - n_y)}} \tag{7.5}$$

with n_c the number of concordant pairs, n_d the number of discordant pairs, n_x the number of tied pairs in x and n_y the number of tied pairs in y. This version of the Kendall's τ is thus used in the following.

7.2.4 NLF estimation for grayscale images

This section describes the noise level function estimation using the previously introduced tools. The estimation process is divided into three main steps: the segmentation, the detection of the homogeneous regions from the simplifications of the image minimizing the Mumford-Shah functional, and the estimation of the NLF from those homogeneous regions.

The segmentation process relies on the tree of shapes. First, the image is blurred with a Gaussian kernel. This filtering is performed to remove the level lines introduced by the noise, thus providing clear level lines for the segmentation. Then, a tree of shapes $\mathcal{T}_\circ$ is built on this blurred image. The tree $\mathcal{T}_\circ$ is then simplified into several trees $\mathcal{T}_\circ'^{\lambda}$, with λ the Mumford-Shah functional parameter used for the simplification. Then, all these trees are reconstructed, providing partitions whose regions are used for the statistical test. This creates different partitions, whose regions have different levels of coarseness according to the value given to λ. The variation of this parameter allows to have a set of regions with different morphologies.

The second step of the estimation is the statistical test on the regions of the simplified images but with the values of the original image. To ensure the statistical test is significant, a restriction on the size of the regions is imposed, and all regions having a size smaller than this threshold (set to 250 pixels in practice) are not tested, and thus considered non-homogeneous. Then, a statistical test associated to the Kendall's τ rank correlation coefficient, presented in the previous subsection, is performed. Several sequences of observations of X and Y are obtained from random divisions of a region into two sequences of pixel values of size n, and for each pair of sequences x and y, the p-value $p = \mathbb{P}(\tau(X,Y) > \tau(x,y|H_0))$ is computed. Among all the p-values computed, the second smallest is compared to a predefined level of detection α_d, and if this p-value is greater, the region is considered as homogeneous. In practice, ten divisions are performed.

Finally, the NLF is estimated from all the homogeneous regions. First, the first and second order statistics, denoted respectively by $\hat{\mu}$ and $\hat{\sigma}^2$, are computed from all the homogeneous regions. From Equation (7.1), the NLF for a given image intensity results in the variance of noise. Thus, estimating the NLF is performed by minimizing the residual error between the empirical variance $\hat{\sigma}_i^2$ and the variance predicted by $\text{NLF}_{(a,b,c)}(\hat{\mu}_i)$ for all homogeneous regions such that:

$$\widehat{(a,b,c)} = \underset{(a,b,c)\in(\mathbb{R}^+)^3}{\operatorname{argmin}} \|\text{NLF}_{(a,b,c)}(\hat{\mu}_i) - \hat{\sigma}_i^2\|_1 \tag{7.6}$$

The minimization of this problem is performed using the primal-dual algorithm of Chambolle-Pock [40], relying on an L_1 minimization since it is more robust to

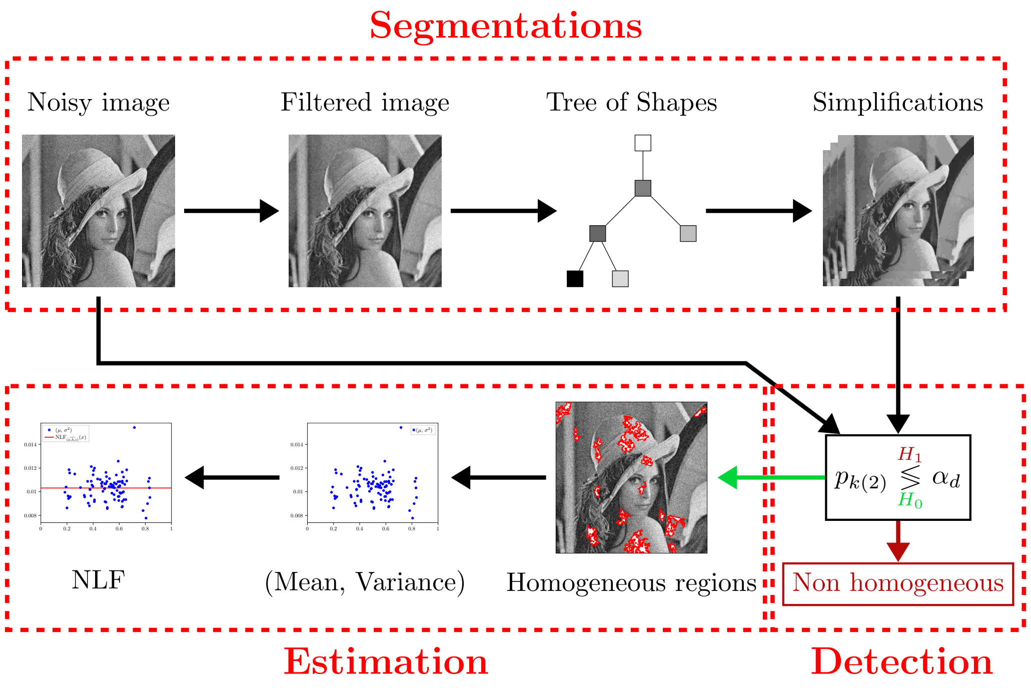


Figure 7.3: NLF estimation for grayscale images

outliers than an L_2 minimization.

The estimation process is summarized in Figure 7.3. For the results of the evaluation of this method, either in terms of the precision of the estimation or for the quality of the identification of the statistical nature of the noise, the reader is referred to [63].

7.3 Extension to color images

In this section, the extension of the NLF estimation to color images is presented.

7.3.1 Issues with multivariate values

The grayscale noise level estimation presented previously is based on the tree of shapes to get the set of regions to be submitted to the statistical test, itself relying on the Kendall's τ correlation coefficient. The tree of shapes relies on the order of its input image values and requires the set of values $\mathcal{V}$ to be endowed with an ordering relationship $\leq$ such that $(\mathcal{V}, \leq)$ forms a complete lattice [176]. Furthermore, the Kendall's τ correlation coefficient is based on the rank of its input sequences values and also requires them to be ordered. Thus, the extension to color images is not straightforward since color values are multivariate and are not endowed with a natural ordering relationship.

Research on multivariate mathematical morphology has been intensively performed, and reviews can be found in [5, 124]. The principle to handle multivariate values for mathematical morphology consists in endowing the set of image values with an ordering relationship such that it forms a complete lattice. Let $a, b, c \in \mathcal{V}$ be three values of an image and $\leq$ be a binary relation on $\mathcal{V}$. This binary relation is

- *reflexive* if $a \leq a$
- *antisymmetric* if $a \leq b$ and $b \leq a \Rightarrow a = b$
- *transitive* if $a \leq b$ and $b \leq c \Rightarrow a \leq c$
- *total* if $a \leq b$ **or** $b \leq a$

The binary relation $\leq$ is a *pre-ordering* if it is reflexive and transitive. It is a *partial ordering* if it is an antisymmetric pre-ordering. It is a *total ordering* if it is a total partial ordering.

In [125], the authors propose a convenient representation to process multivariate images whose value set is endowed with a total ordering relationship based on the rank of these values. Let m be the number of multivariate values in the image value set $\mathcal{V} \subset \mathbb{R}^n$ such that $m = |\mathcal{V}|$. A rank transform $\mathcal{R}$ is a bijective mapping $\mathcal{R} : \mathcal{V} \to [\![1..m]\!]$ such that it returns the rank of a value according to a total ordering relationship $\leq$. Applied on all the values of an image f, the rank transform results in a *rank map* $\mathcal{R}_f : \Omega \to [\![1..m]\!]$ such that at every position of the image, the rank of the value is returned. Thus, for morphological operations, instead of processing the pixel values, the rank map is used. As $\mathcal{R}$ is bijective, the multivariate values can be easily retrieved from their rank.

7.3.2 Complete lattice learning

To obtain a rank map $\mathcal{R}_f$, the set of image values $\mathcal{V}$ requires to be endowed with a total ordering relationship $\leq$. Since multivariate values have no natural total ordering relationship, one has to be imposed artificially. In [124], Lezoray proposes an unsupervised method named *complete lattice learning* based on machine learning to obtain such total ordering from the framework of h-ordering. A h-ordering $\leq_h$ is an ordering relying on a projection h such that for $a, b \in \mathcal{V}$,

$$a \leq_h b \iff h(a) \leq h(b) \tag{7.7}$$

The objective of the complete lattice learning is to learn the projection h by the mean of the Laplacian Eigenmaps [15].

The complete lattice learning is a three steps process. First, a dictionary $\mathcal{D}$ of $p \ll m$ elements of $\mathcal{V}$ is built by a vector quantization algorithm. For the noise level estimation, the vector quantization is performed by using the K-means clustering algorithm. This step is necessary to reduce the amount of sample to be given to the Laplacian Eigenmaps as its complexity is quadratic with the number of input samples.

The Laplacian Eigenmaps [15] is a manifold learning algorithm based on the Laplacian matrix of the pairwise similarity graph of the element of a set. In this case, this similarity graph is built on the elements of $\mathcal{D}$ and results in an adjacency matrix $S_{\mathcal{D}}$ which encodes the similarity between two values such that

$$(S_{\mathcal{D}})_{ij} = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{s}\right) \tag{7.8}$$

with $\mathbf{x}_i$ and $\mathbf{x}_j$ respectively the i^{th} and j^{th} element of $\mathcal{D}$ and s being the maximum distance between two elements $\mathbf{x}_i$ and $\mathbf{x}_j$ such that $s = \max_{\mathbf{x}_i,\mathbf{x}_j \in \mathcal{D}} \|\mathbf{x}_i - \mathbf{x}_j\|_2^2$. From this matrix, the diagonal degree matrix $(D_{\mathcal{D}})_{ii} = \sum_j (S_{\mathcal{D}})_{ij}$. The Laplacian matrix is then defined by $L_{\mathcal{D}} = D_{\mathcal{D}} - S_{\mathcal{D}}$. The projection $h_{\mathcal{D}}$ used to compute the h-ordering, is obtained by computing the eigenvectors of the normalized Laplacian matrix defined as:

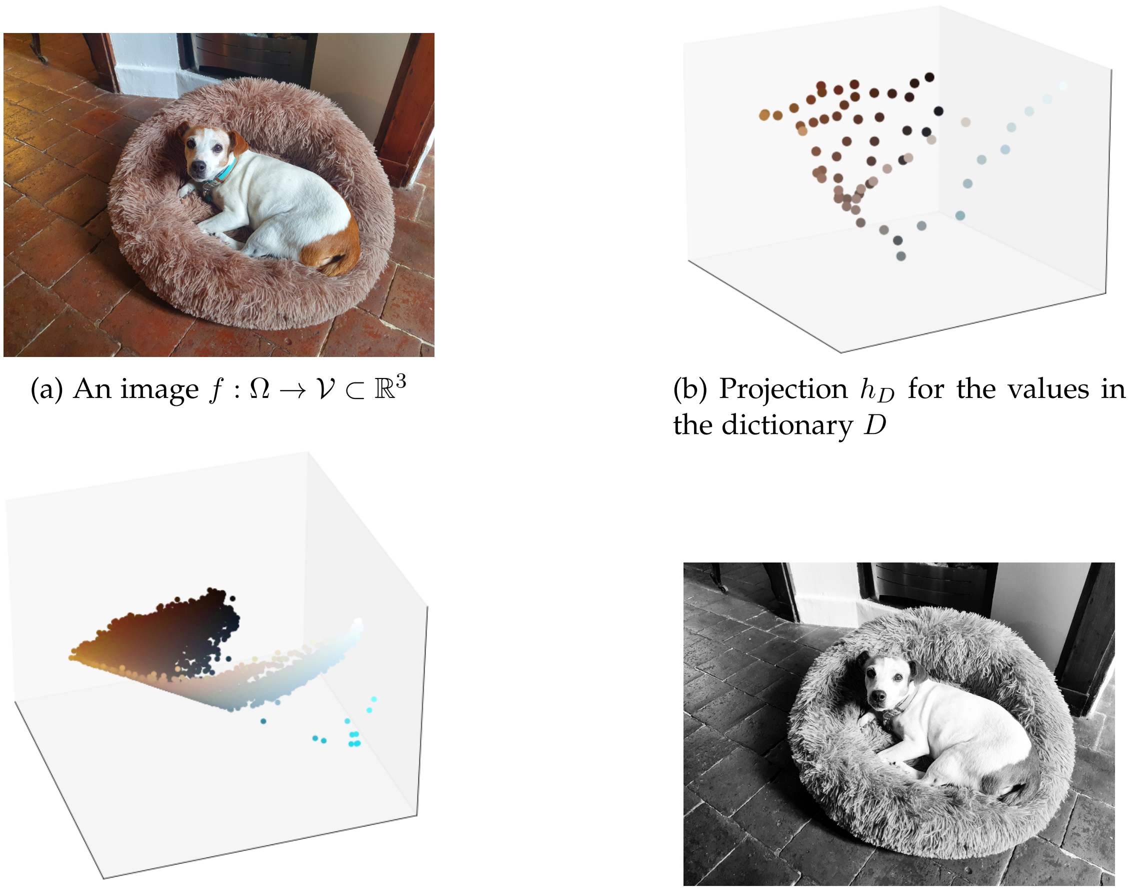
$$\overline{L}_{\mathcal{D}} = D_{\mathcal{D}}^{-\frac{1}{2}} L_{\mathcal{D}} D_{\mathcal{D}}^{-\frac{1}{2}} \tag{7.9}$$

The projection $h_{\mathcal{D}}$ is defined by $h_{\mathcal{D}}(\mathbf{x}_i) = (\phi_{\mathcal{D}}^1(\mathbf{x}_i), ..., \phi_{\mathcal{D}}^p(\mathbf{x}_i))$ where $\mathbf{x}_i$ is the i^{th} element of $\mathcal{D}$ and $\phi_{\mathcal{D}}^j$ is the j^{th} eigenvector of $\overline{L}_{\mathcal{D}}$, with $\phi_{\mathcal{D}}^j(\mathbf{x}_i)$ its i^{th} value.

This projection $h_{\mathcal{D}}$ being computed on the dictionary $\mathcal{D}$, it is only valid for the values of $\mathcal{D}$. Thus, the projection $h_{\mathcal{D}}$ has to be extended to all the values contained in the set of image values $\mathcal{V}$. This operation is performed by the use of the Nyström extension [205]. Let $\Phi_{\mathcal{D}}$ the matrix of eigenvectors of $\overline{L}_{\mathcal{D}}$, $\Phi_{\mathcal{V}}$ the matrix of eigenvectors for the image values set $\mathcal{V}$, $\Lambda_{\mathcal{D}}$ the diagonal matrix of eigenvalues obtained by the eigen-decomposition of $\overline{L}_{\mathcal{D}}$, and $D_{\mathcal{V}}$ the diagonal degree matrix computed from the values of $\mathcal{V}$. The Nyström extension is defined by

$$\Phi_{\mathcal{V}} = D_{\mathcal{V}}^{-\frac{1}{2}} S_{\mathcal{D}}^{\mathrm{T}} D_{\mathcal{D}}^{-\frac{1}{2}} \Phi_{\mathcal{D}} \Lambda_{\mathcal{D}}^{-1} \tag{7.10}$$

The projection h is thus defined by the eigenvectors contained in the matrix $\Phi_{\mathcal{V}}$ such that for $\mathbf{v} \in \mathcal{V}$, $h(\mathbf{v}) = (\phi_{\mathcal{V}}^1(\mathbf{v}), ..., \phi_{\mathcal{V}}^p(\mathbf{v}))$ with $\phi_{\mathcal{V}}^j(\mathbf{v})$ the j^{th} extrapolated eigenvector of the normalized Laplacian matrix for all the values in $\mathcal{V}$. The vectors returned by the projection h are ordered lexicographically so that the h-ordering is well-defined.



(a) An image $f : \Omega \to \mathcal{V} \subset \mathbb{R}^3$

(b) Projection h_D for the values in the dictionary D

(c) Projection h for the values of f

(d) The associated rank map $\mathcal{R}_f$

Figure 7.4: Illustration of the complete lattice learning framework

Figure 7.4 illustrates the complete lattice learning framework. In Figure 7.4a, a color image f is given as input. It is quantized and the Laplacian Eigenmaps are applied to the dictionary $\mathcal{D}$ of values, resulting in the projection $h_{\mathcal{D}}$ in Figure 7.4b. The projection $h_{\mathcal{D}}$ is then extended to all the values of $\mathcal{V}$, resulting in the projection h displayed in Figure 7.4c. Finally, the rank transform is applied to the image f by using the projection h, resulting in the rank map $\mathcal{R}_f$ in Figure 7.4d.

7.3.3 The multivariate tree of shapes

The extension of the tree of shapes to color images has been studied and different solutions were proposed. In [35], a first attempt to build a tree of shapes on color images is presented. The authors propose to build one tree per channel and to

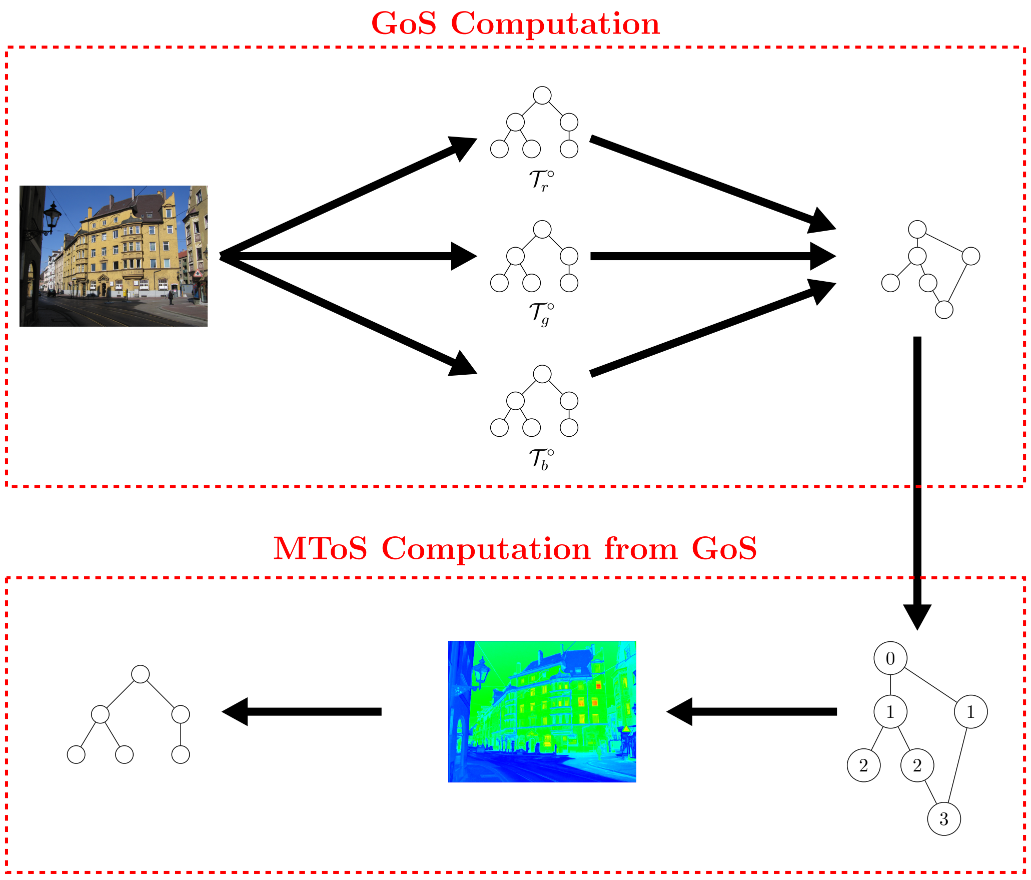
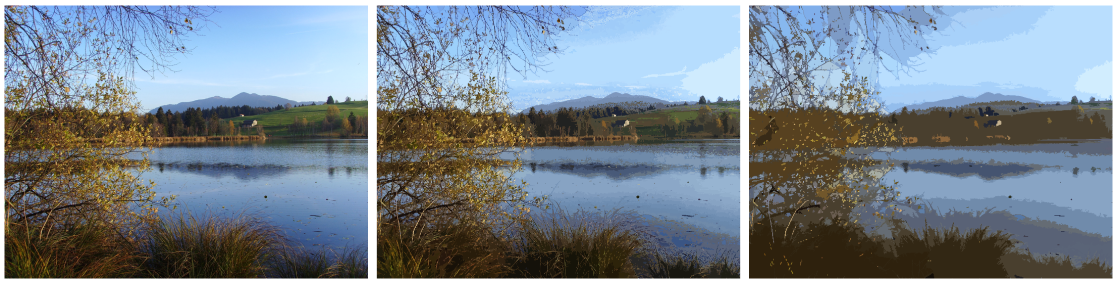


Figure 7.5: MToS computation process

merge them by computing a map using the area attribute obtained from the channelwise trees, and then compute a tree of shapes from this map. In [36], the authors explore three approaches: building the tree using a total ordering imposed on the image value set, using a distance between the level lines of the tree to build it with an algorithm similar to the one proposed in [73] and build a graph of shapes representing the inclusion relationship between the different shapes from the different channels, this latest not building a tree. The graph-based approach has the best results in terms of PSNR for color image denoising purposes using filtering. Finally, in [33], the authors build a tree from the graph of shapes, resulting in the Multivariate Tree of Shapes (MToS) [38]. Thus, this tree relies on the inclusion relationship of the shapes instead of the ordering relationship of the color values.

The different elements for the MToS computation are discussed. This construction process is illustrated in Figure 7.5. For more details about the algorithms used to obtain the different components, the reader is referred to [31].



(a) An image (b) $\lambda = 1000$ (c) $\lambda = 10000$

Figure 7.6: Illustrations of the Mumford-Shah simplification on the MToS

The MTos computation is divided into two steps: the computation of the graph of shapes (GoS), and the retrieval of the MToS from the information embedded by the GoS. In the first step, the channelwise trees of shapes $\mathcal{T}_\circ^c$ are constructed from each channel f_c of the image. Then, these trees are merged into one graph $\mathfrak{G} = (V_\mathfrak{G}, E_\mathfrak{G})$ which is the GoS. Each vertex of $V_\mathfrak{G}$ represents a node of a ToS $\mathcal{T}_\circ^c$, and the edges from $E_\mathfrak{G}$ encode the inclusion of the shapes in one channel, but also the inclusion between the shapes of the channelwise ToSs. The roots of the channelwise ToSs represent a unique vertex in the GoS, which is considered as its root. Thus, the GoS is a directed acyclic graph. Finally, for the set of shapes $\mathfrak{C} = \cup_c \mathfrak{C}_\circ^c$, $(\mathfrak{C}, \subseteq)$ forms the cover of $\mathfrak{G}$.

The second step of the MToS computation is the extraction of the MToS from $\mathfrak{G}$. To this aim, the depth of each node of $\mathfrak{G}$ is computed. Let $v_r \in V_\mathfrak{G}$ be the root of $\mathfrak{G}$. For any node $v \in V_\mathfrak{G}$, the depth $\rho(v)$ is defined by

$$\rho(v) = \max_{\pi(v_r, v)} |\pi(v_r, v)| \tag{7.11}$$

This depth attribute is thus the longest path from a node v to the root v_r. From the depth attribute related to each vertex of the graph, a distance map $m : \Omega \to \mathbb{N}$ is computed, such that for all $p \in \Omega$, the value $m(p)$ is the maximum value $\rho(v)$ such that p is contained in the shape represented by v. Finally, a max-tree is built on the depth map, ensuring that the components of set of upper connected components $\mathfrak{C}^\geq$ are valid shapes by applying the hole-filling operator. This max-tree is thus called *hole-filled max-tree* and its construction on the depth map m results in the MToS.

Finally, the simplification of the MToS by minimizing the Mumford-Shah functional is demonstrated in [38], and illustrated in Figure 7.6, where an image is simplified using two different Mumford-Shah parameters.

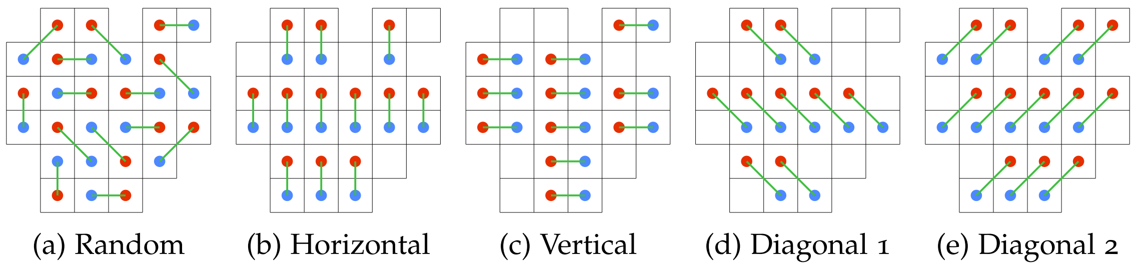


Figure 7.7: Division schemes for two sequences x and y

7.3.4 Multivariate NLF estimation

The complete lattice learning and the MToS are two tools that allow to extend the NLF estimation to color images. However, the NLF formulated in Equation 7.1 is valid only for grayscale images and thus requires to be adapted. In this document, the Gaussian, Poisson, and multiplicative noise are still the three main statistical natures of the noise, and the correlation between the channels is not taken into account. This leads to a simple extension of the NLF, called the Multivariate Noise Level Function (MNLF), defined for a multivariate image with n channels by:

$$\mathrm{MNLF}_{(\mathbf{a},\mathbf{b},\mathbf{c})}(\mathbf{x}) = \begin{pmatrix} a_1 x_1^2 + b_1 x_1 + c_1 \\ a_2 x_2^2 + b_2 x_2 + c_2 \\ \dots \\ a_n x_n^2 + b_n x_n + c_n \end{pmatrix} \tag{7.12}$$

with $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^n$, a_i, b_i and c_i being respectively the i^{th} element of $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, and $\mathbf{x} \in \mathcal{V}$ being a pixel value of the image whose i^{th} element is x_i. Thus, the objective of the extension to color images is to estimate this MNLF.

The MNLF estimation is quite similar to the NLF estimation: it still has its three main parts (segmentation, detection of homogeneous regions, and estimation). However, the main differences are the tools used to perform these steps. First, the segmentation step is almost the same as for the grayscale parameters estimation: the image is blurred using a Gaussian filter to reduce the number of level lines in the image, and the MToS is built on this filtered image. This tree is simplified several times with different values for the parameter λ of the Mumford-Shah functional, resulting in different segmentations of the image.

The homogeneous region detection step is the part of the estimation that changes the most. First, instead of using the values of the original image for the computation of the Kendall's τ correlation coefficient and its associated p-value, the rank map, obtained from the application of the rank transform on the image using the h-ordering whose projection h is the result of the complete lattice

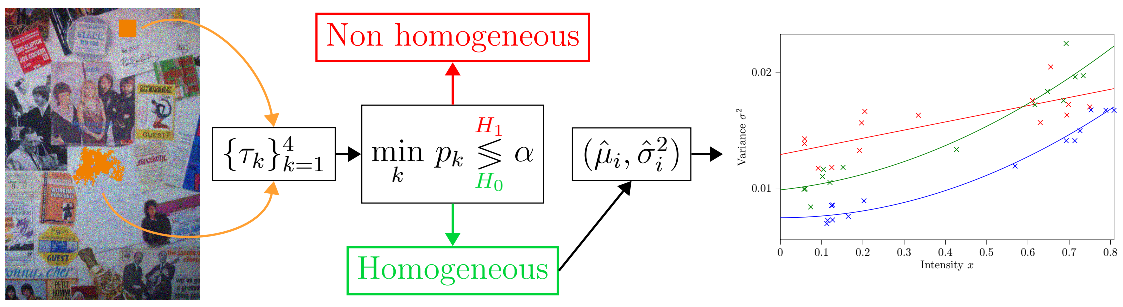

Non homogeneous
$\{\tau_k\}_{k=1}^4$
$\min_k p_k \overset{H_1}{\underset{H_0}{\lessgtr}} \alpha$
Homogeneous
$(\hat{\mu}_i, \hat{\sigma}_i^2)$
Variance σ^2
Intensity x

Figure 7.8: Summary of the multivariate estimation

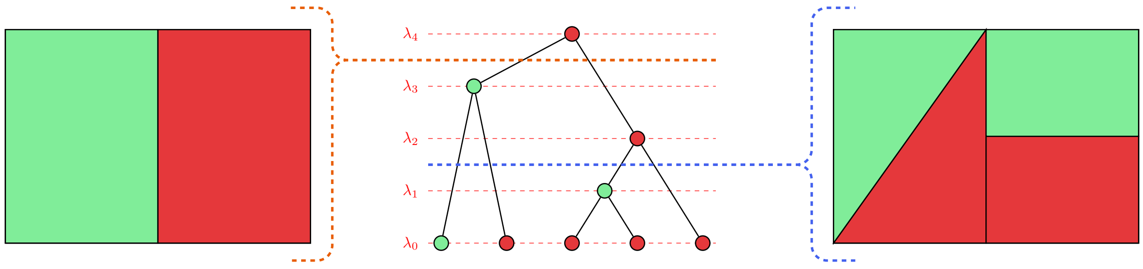


Figure 7.9: Homogeneous nodes detection

learning, is used instead. Since the Kendall's τ relies on the ranks instead of the values of the pixels, this representation is well-fitted for its computation and allows a fast correspondence between the points of the regions obtained from the segmentation and the associated rank. The second difference is the division of a given region into two sequences of rank values. In [63], a random division is performed. However, using an adjacency-based division, as it is performed by Sutour *et al.* [204], increases the efficiency of the estimation. These different division schemes are illustrated in Figure 7.7. Figure 7.7a shows the division used for the grayscale estimation, and Figures 7.7b to 7.7e display the divisions used for the multivariate estimation. Thus, instead of using the second smallest p-value for the detection of the region, the smallest p-value is compared to detection level α_d, and if it is greater, the region is considered homogeneous.

Finally, the estimation process is the same as for the grayscale estimation but performs the estimation channelwise, each NLF obtained resulting in the MNLF. The multivariate estimation pipeline is summarized in Figure 7.8.

7.4 Extension to the whole hierarchy

The minimization process to obtain a partition from the tree of shapes simplification used in the previous section has several disadvantages. First, only a subset of regions are tested for the homogeneous region detection: some regions may be homogeneous and provide new statistical sample for the estimation but are not tested since they do not belong to the segmentation. Then, the same region may appear in two segmentations and be detected as homogeneous, leading to the same statistical sample appearing twice for the minimization of Equation 7.6. If this region is a false positive, meaning that it is detected as homogeneous whereas it is not, this increases the number of outliers and affects the precision of the estimation. The issue of recurrent regions could be solved by computing the similarity of the detected regions using for example the Dice score to perform an unification of the tested regions, but such a process would have a significant impact on the speed of the estimation.

These issues are solved by using all the nodes of the hierarchy whose size is greater in terms of area than a given threshold. In practice, the hierarchy is first filtered. Thus, as each node of the filtered hierarchy represents a region, the statistical test is applied to all of them as for the regions from the segmentation and the resulting homogeneous regions are qualified by the term *homogeneous nodes* due to their relation with the tree. This is illustrated in Figure 7.9. In this figure, the homogeneous nodes are displayed in green and non-homogeneous nodes are in red, with their relation with their respective partition for two horizontal cuts. This demonstrates the resolution of the two previously cited issues: as the nodes are used instead of the regions from each segmentation, they are tested only once. Furthermore, all the regions are tested. Thus, the detection of homogeneous regions is performed by a simple tree traversal.

7.5 Comparative results

In this section, the different approaches presented in this chapter are evaluated.

7.5.1 Experimental set-up

To compare the different estimation processes, the whole image database of Laurent Condat [42] is used. This database is composed of 150 natural images whose dimension is either 720×540 or 540×720. In those high-definition images, the acquisition noise can be neglected, and thus this dataset is ideal for experimentation on noise estimation tasks. These images are corrupted by

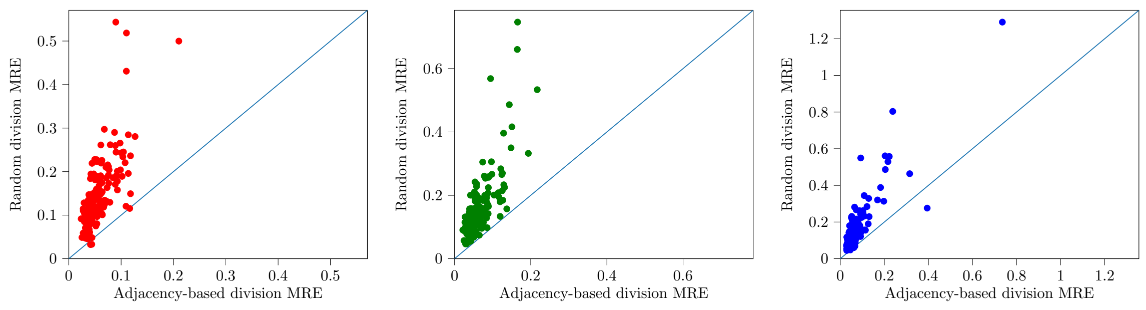


Figure 7.10: Comparison between random division and the adjacent division

a mixed noise with a given MNLF whose coefficient elements in $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ are randomly picked from a standard normal law $\mathcal{N}(0.01, 0.003)$.

Following what has been done in [63] and [204], the different MNLF are compared channelwise using the mean relative error (MRE) measure between two NLF. Let $\text{NLF}_{(a_i,b_i,c_i)}$ be a reference NLF and $\text{NLF}_{\widehat{(a_i,b_i,c_i)}}$ be the NLF resulting from a noise level estimation. The MRE is defined by:

$$\text{MRE}(\widehat{(a_i, b_i, c_i)}) = \frac{1}{|\mathcal{V}_i|} \sum_{v \in \mathcal{V}_i} \frac{|\text{NLF}_{(a_i,b_i,c_i)}(v) - \text{NLF}_{\widehat{(a_i,b_i,c_i)}}(v)|}{\text{NLF}_{(a_i,b_i,c_i)}(v)} \tag{7.13}$$

with $\mathcal{V}_i$ the set of unique value of the i^{th} channel. From this mathematical definition, the lower the MRE, the more efficient the estimation.

The different estimations are parameterized as follows. First, the level of detection α_d is set to 0.4 and the minimum size for a region to be tested is 250. For the region-based estimation, the Mumford-Shah parameter λ is set to 200, 500 and 700, leading to three segmentations for the estimation.

For the different evaluations, the evaluation protocol consists in adding noise to an image, running the different estimations on the noisy image, and then computing the MRE. This process is performed 20 times and the mean MRE is computed. For the visualization of the results, three plots are displayed by comparison, one for each channel, and the two axes correspond to the mean MRE for an estimation. Thus, for each image, a point is drawn on the plot. A line is drawn following the identity function, and a point above the line means the estimation related to the x-axis MRE is more efficient than the estimation related to the y-axis MRE.

7.5.2 Region division schemes comparison

Figure 7.10 displays the comparison of the random division scheme used in [63] with the usage of an adjacent division scheme such as the one used for square

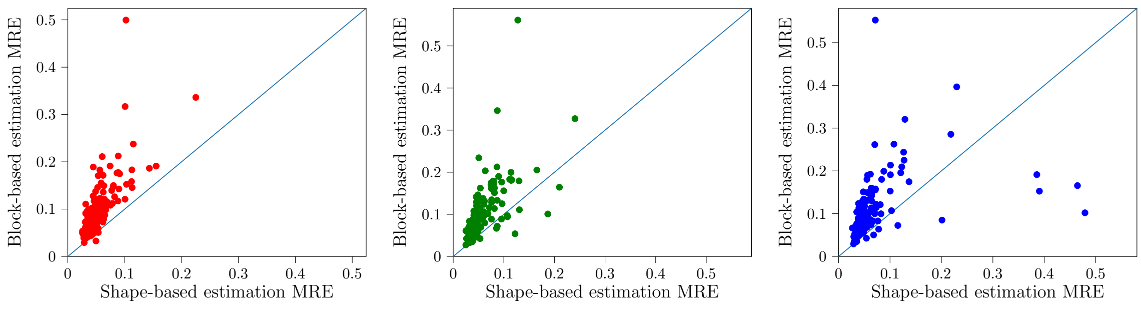


Figure 7.11: Comparison between shape-based and block-based estimations

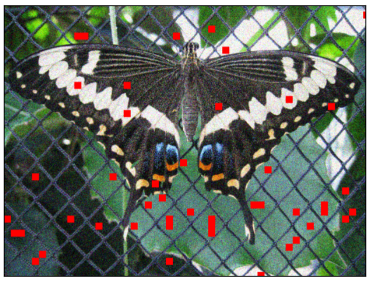
patches in [204] but adapted to the irregular morphology of the regions. A large majority of the estimation for the adjacency-based division has a lower MRE than the usage of a random division. Thus, the adjacency-based division is more efficient in the red, green, and blue channels for respectively 146, 149, and 147 images out of the 150 images of the database. This experiment validates the usage of an adjacency-based division instead of a random division.

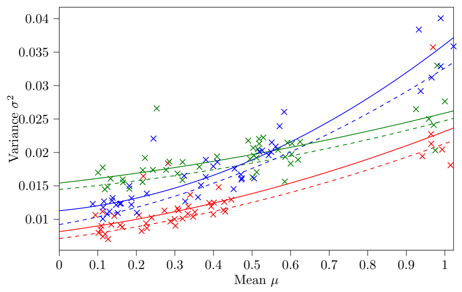
7.5.3 Precision of content adaptation

To evaluate the precision of the extension of the grayscale estimation to color images, to be concordant with the experiments run in [63] and [204], the block-based and the region-based estimation are compared. To this aim, the block-based estimation is extended to color images, but instead of using the rank map to order the color, the statistical test is achieved channelwise, and the p-value is compared to a detection level lowered to $\alpha_d/3$. The usage of such statistical tests comes from the fact that it is more efficient than the usage of the rank map for block-based color estimation, which is not the case for the region-based estimation. Furthermore, the image is divided into square blocks whose initial side size is set to 16. This size is iteratively decreased by 2 until a repartition criterion on the valuespace of the image is respected or if a minimum side size, set to 4, is reached.

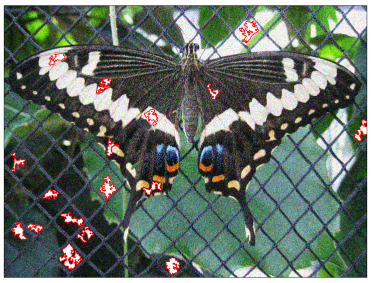
The results of the comparison are displayed in Figure 7.11. These plots show that the extension of the estimation to color images yields better results in terms of precision for the region-based estimation than the block-based one. In fact, for the red, green, and blue channels, the amount of images with better results using the region-based estimation is respectively 147, 137, and 139 images. The results obtained in [63] for grayscale images are thus validated for color images.

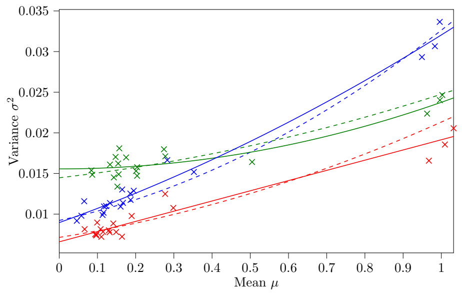
To analyze the results obtained previously, block-based and region-based estimations are performed on the same noisy image and displayed in Figure 7.12.


(a) Square patches



(b) Square patches MNLF


(c) Regions



(d) Regions MNLF

Figure 7.12: Visual comparison of the patch selection

Figures 7.12a and 7.12c show respectively the homogeneous blocks and regions from the estimations, and Figures 7.12b and 7.12d show their respective resulting MNLF. In Figures 7.12b and 7.12d, dashed plots represent the known MNLF of the image, and plain ones show the estimated MNLF. From those figures, many blocks are detected as homogeneous in areas of the image where no homogeneous regions are detected. Furthermore, there is a larger amount of homogeneous blocks, conversely to homogeneous regions. However, several homogeneous blocks result in several outliers for the pairs $(\hat{\mu}, \hat{\sigma}^2)$. Thus, the block-based estimation results in a large number of homogeneous blocks, but the region-based estimation has qualitative homogeneous regions, leading to a better MRE despite its amount.

7.5.4 Comparison with the whole hierarchy

In this part, the extension of the estimation to the whole hierarchical structure of the tree is investigated. To this aim, two hierarchical representations are used: the MToS, which is used for the extension of the grayscale estimation, and the ω-tree, which belongs to the partitioning hierarchies and has shown robustness to the noise when used for attribute profiles construction [113]. As a constrained criterion ω for the construction of the tree, the length of the bounding box

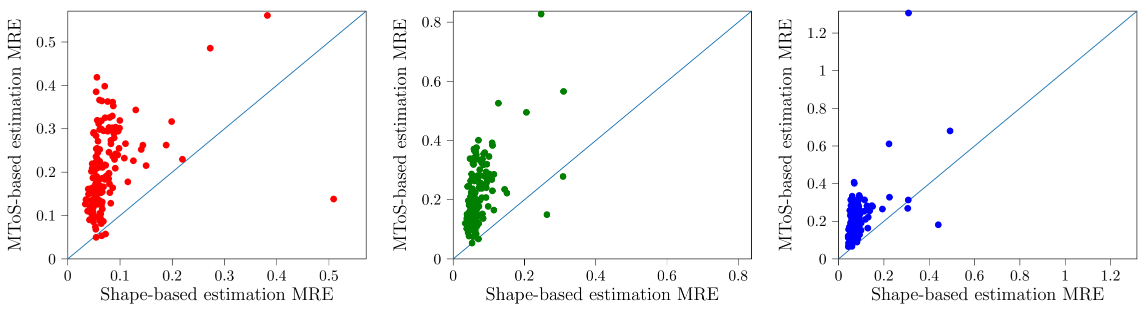


Figure 7.13: Comparison between shape-based and MToS-based estimation MRE

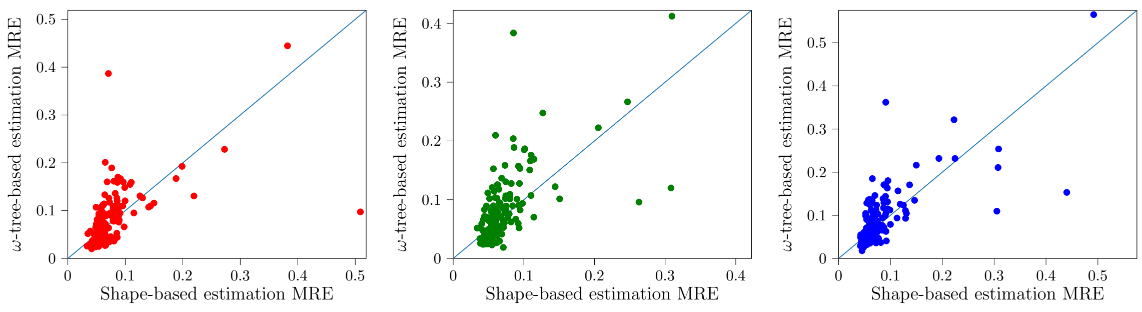


Figure 7.14: Comparison between shape-based and ω-tree-based estimation MRE

diagonal from the three-dimensional space of the color values distribution of a region is used.

The resulting MRE for the estimation using the MToS and the ω-tree are displayed respectively in Figures 7.13 and 7.14. In the first case, the usage of a subset of shapes using the simplified tree outperforms the results when the whole MToS is used. Thus, even though all the regions from the MToS are tested to detect homogeneous regions, the precision of the estimation is greatly impacted compared to the usage of a subset obtained from the minimization of the Mumford-Shah functional. In the second case, a different hierarchy is used as it belongs to partitioning hierarchies. This hierarchy, based on a global constraint in addition to the usage of a local dissimilarity, results in no improvement when used for the estimation of the MNLF. In fact, the shapes-based estimation is better than the ω-tree-based estimation for the red, green, and blue channels for respectively 78, 73, and 77 images.

7.6 Conclusion

To conclude this chapter, the noise level estimation using the tree of shapes was first extended to multivariate images, improving the genericity of the method in [63] on the values of an image. In terms of noise level estimation, the results demonstrated by the region-based method validated the previous results for grayscale images compared to the block-based estimation. However, this method does not handle the inter-channel correlation that usually results from the demosaicking when this one is performed before the denoising in the image acquisition pipeline [98]. Taking into account such parameters would not modify a lot the estimation pipeline since the segmentation is performed using the MToS, which is one representation of the whole multivariate image. Furthermore, the multivariate values for the statistical test are transformed into a single value by means of the complete lattice learning. This work would require a new MNLF and adapting the estimation pipeline. Future works include the modelization of a noise level function taking into account such a correlation and thus an adaptation to the current estimation to be able to estimate this parameter.

In the second part of the contribution presented in this chapter, the estimation was extended to the whole hierarchy and not only a subset of its regions. However, this does not improves the results, and leads to a new issue: what is the behavior of a hierarchical representation when built on a noisy image? This question is investigated in the next chapter.

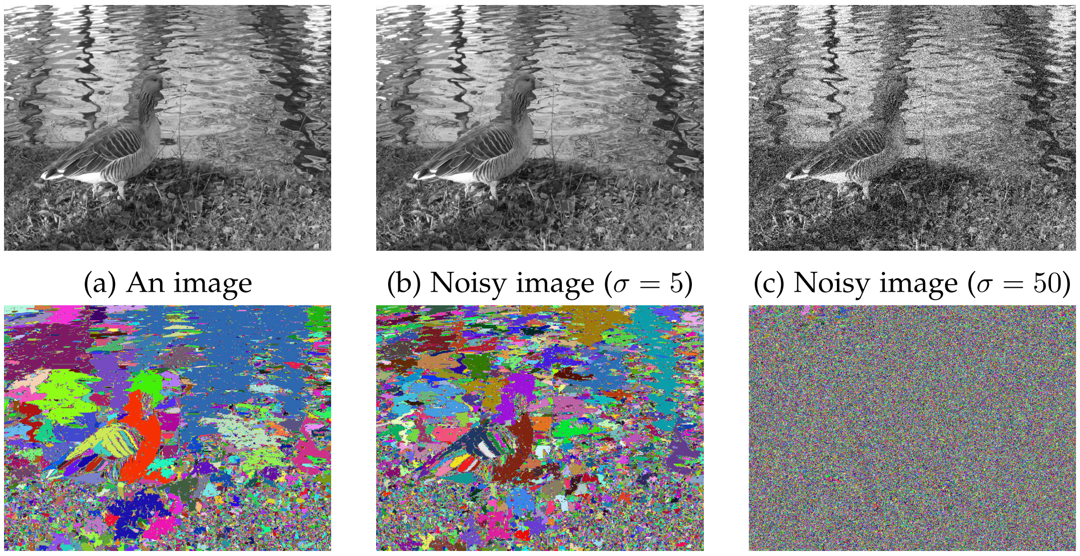
Chapter 8

Impact of the Noise on Hierarchical Representations

In the previous chapter, hierarchical representations of images were investigated in order to estimate the noise level function of an image. They were used to get a set of regions on which a statistical test was applied to obtain homogeneous regions, whose texture information is negligible enough to only have noise. However, using all the regions of the hierarchy did not improve the result of the estimation compared to the usage of a subset of regions, which raised an interrogation about the impact of the noise in an image on its hierarchical representation. Thus, in this chapter, such an impact is examined. More specifically, the structure variation of the hierarchical representation is studied related to different noise levels. This study is then extended to the case where the hierarchy is constrained by an energy, and it is shown that the choice of the energy influences the variation of the structure in the presence of noise.

8.1 Motivations

In the previous chapter, hierarchical representations were used to provide a set of regions to be tested in order to obtain image segments only containing noise. However, the results were similar in terms of precision to the method only using a subset of regions from the hierarchy. This led to the questioning about the impact of the noise in an image when such a hierarchical representation is built from this noisy image. This chapter focuses on the α-tree [157] due to its relation with several hierarchical representations [49]. In Figure 8.1, an α-tree is built



(a) An image (b) Noisy image ($\sigma = 5$) (c) Noisy image ($\sigma = 50$)

(d) (10)-partition of (a) (e) 10-partition of (b) (f) 10-partition of (c)

Figure 8.1: Different horizontal cuts from an α-tree built from noisy images

from the images in Figures 8.1a, 8.1b, and 8.1c and their ($\alpha = 10$)-partition is displayed respectively in Figures 8.1d, 8.1e, and 8.1f. The last two images are corrupted with an additive Gaussian noise with a low noise level ($\sigma = 5$) and a high noise level ($\sigma = 50$). By comparing the different partitions at the same level α, we can observe that the noise has an impact on the α-tree: the partition from the tree built on the clean image has 62305 regions while this amount increases with the noise level (74473 for $\sigma = 5$ and 290803 for $\sigma = 50$). Furthermore, the content is still retrievable when the noise parameter is small, but the partition obtained from the tree built on a highly noisy image is entirely corrupted by the noise.

Partitioning hierarchies have been evaluated in the context of segmentation many times. To this aim, different measures [170], either based on regions or on boundaries, were used to compare one or several partitions of the hierarchies with a ground-truth segmentation. In [6], a binary partition tree built from different contour detector outputs is evaluated by such measures on regions and boundaries by combining the results of these measures on different partitions of the hierarchy. In [171, 172], authors propose an algorithm that selects the best partition from a partitioning hierarchy according to a given measure and a ground truth. Thus, the resulting partition is the best segmentation according to the measure the hierarchy can provide. In [163], the authors extend the previous evaluation procedures for hierarchical segmentation and apply them to hierarchical watersheds. This methodology also serves to evaluate combinations of hierarchical watersheds [186]. However, among all these evaluation methods and experiments, the case where a noisy image is used as an input is not taken into account.

The impact of the noise in hierarchical representations has been little investigated. In the context of the tree of shapes [26], the stability of the tree of shapes is observed on a toy image by means of different measures on the tree. These measures are either geometrical, topological, or spectral. In [113], a comparison of the impact of the noise is performed for attribute profile construction on inclusion and partitioning hierarchies, and it resulted in greater performance for the ω-tree than the others in terms of noise robustness. However, this work is limited to attribute profiles, while the objective of this chapter is to evaluate the impact of the noise on the whole hierarchy.

Thus, in this chapter, the impact of the noise in an image on the α-tree built on this image is studied. More specifically, we look for a relationship between the noise level and an attribute measured on the tree. To this aim, we observe the variation of two measures computed on the tree structure: the distribution of the depth attribute for each node of the tree, and the amount of non-persistent nodes when the tree is constrained to an energy. Furthermore, in order to provide a consistent evaluation, natural images with various textures in their content and a constant image, which contains only noise when corrupted, are compared. We then use the knowledge acquired in the structural study of the tree to design a dissimilarity measure between two trees based on the geometry of their regions to measure the impact of the noise by comparing two hierarchies built on a clean and a noisy image.

8.2 Structural study of the impact of the noise in the α-tree

8.2.1 Study set-up

Let $f : \Omega \to \mathcal{V} \subseteq [\![0..255]\!]$ be a grayscale image whose values are encoded as unsigned values on 8 bits. A special case of this image is the constant image f_c with $c \in [\![0..255]\!]$ such that $\forall p \in \Omega, f_c(p) = c$. For those images, their noisy versions are denoted by f_σ and $f_{c,\sigma}$ for respectively a noisy image and a noisy constant image, which means an image that only contains noise. Noisy images are defined by $f_\sigma = f + n_\sigma$ with n_σ a sample of values drawn from a normal law $\mathcal{N}(0, \sigma^2)$ and defined on the domain Ω. When artificially corrupted, the values of f_σ not belonging to $\mathcal{V}$ are clipped to the nearest value between 0 and 255. Finally, the α-tree $\mathcal{T}_\alpha$ is built on such noisy images using an L_1 dissimilarity between the

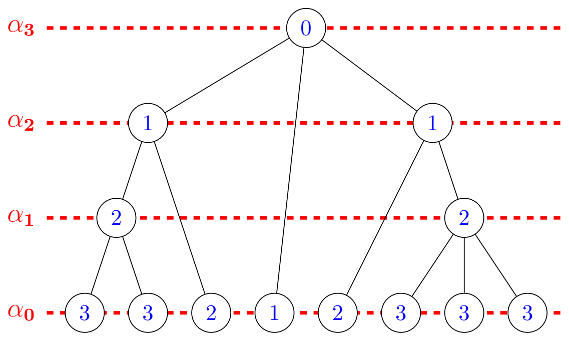


Figure 8.2: Illustration of the depth $\mathcal{P}_{\mathcal{T}_\alpha}$ on a tree $\mathcal{T}_\alpha$

pixels with the 4-adjacency relationship.

To study the structure of the tree, the depth of every node is used, and their distribution is observed. This attribute is the number of nodes on the path from the current node to the root, the latter being excluded from the count. The depth $\mathcal{P}_{\mathcal{T}_\alpha}(n)$ for a node n representing a region $R_\alpha \in \mathcal{T}_\alpha$ is defined by

$$\mathcal{P}_{\mathcal{T}_\alpha}(n) = \begin{cases} 0 & \text{if } n = \text{root}(\mathcal{T}_\alpha) \\ \mathcal{P}_{\mathcal{T}_\alpha}(\text{parent}(n)) + 1 & \text{else} \end{cases} \tag{8.1}$$

The depth attribute is illustrated in Figure 8.2. The numbers in blue correspond to the depth values of each node of the α-tree $\mathcal{T}_\alpha$. This attribute relies entirely on the structure of the tree and not on its regional properties.

This attribute is studied by means of several histograms $h(d) = |\{n \in \mathcal{T}_\alpha | \mathcal{P}_{\mathcal{T}_\alpha}(n) = d\}|$. When computed on a tree built from a noisy image f_σ (respectively a noisy constant image $f_{c,\sigma}$), these histograms are denoted by h_σ (respectively $h_{c,\sigma}$). The average depth $\mu(h)$ and the mode of the distribution $m(h)$ are computed from these histograms such that

$$\mu(h) = \frac{1}{|h|} \sum_{d \in \mathcal{P}_{\mathcal{T}_\alpha}} d \times h(d) \tag{8.2}$$

and

$$m(h) = \operatorname*{argmax}_{d \in h} h(d) \tag{8.3}$$

with $|h|$ the total number of element in h. These statistics on the histograms are used in the following to analyze the distribution and investigate a potential relationship between the noise level σ and the structure of the tree.

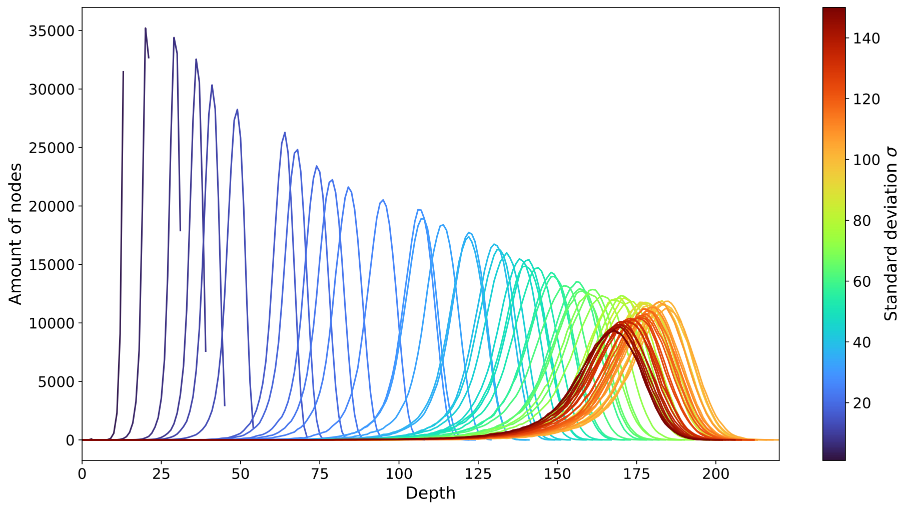


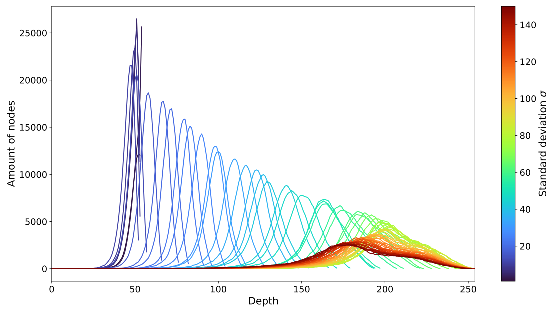
Figure 8.3: Depth histograms $h_{c,\sigma}$ obtained from the α-trees built on $f_{c,\sigma}$.

8.2.2 Impact of the noise on the tree structure

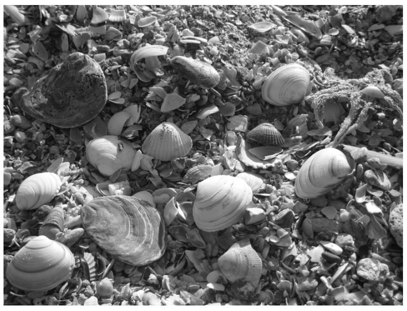
To study the impact of the noise corrupting the image on which $\mathcal{T}_\alpha$ is built, the noisy constant image $f_{c,\sigma}$ is used. In practice, the value c is set to 127, which is the median value of its value space. This image has one advantage: as it is constant, no texture is merged with the noise, and thus the α-tree is built on a pure noise realization. This process enables the comparison of noise impact on diverse natural images, each with varying content.

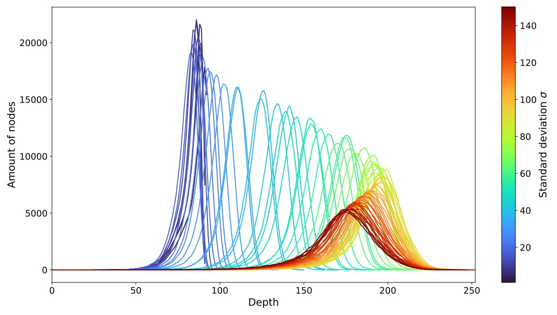
First of all, the impact of the noise in the constant image from which the α-tree is built is observed. To this aim, several $f_{c,\sigma}$ are used, with a noise level σ varying from 1 to 150. The histograms of the depth obtained from the trees built on these images are computed and displayed in 8.3. By observing the histograms, a common behavior is observed: the depth distribution tends to become a tailed distribution when σ increases. The tail is composed of nodes with a low depth, which are small components, usually of size 1, which have an intensity significantly different from their surrounding pixel values. Secondly, the mode of the distributions increases while the noise level increases, until a given high noise level, after which it then decreases. This comes from the clipping of the values during the corruption process of the image, in which new flat zones appear due to the saturation. Furthermore, the variance of the histograms increases while the noise level increases. It may be concluded from these observations that there indeed exists a strong relationship between the noise level σ and the structure of the tree, and particularly the distribution of the depth obtained from the tree nodes, but until a certain level in which case the image becomes largely saturated.

(a) Low brightness image



(b) Depth histogram of (a)


(c) Textured image



(d) Depth histogram of (c)

Figure 8.4: Depth histograms h_σ obtained from α-trees computed on noisy natural images f_σ

From the previous conclusion, the impact of the noise is evaluated on trees built from natural images f_σ corrupted by Gaussian noise. Depth distributions of the α-tree built on two natural images from the Laurent Condat image database [42] are displayed in Figure 8.4. These two images are chosen due to their particular characteristics such as a low brightness (Figure 8.4a) or a high texture (Figure 8.4c). Their depth distributions are respectively displayed in Figures 8.4b and 8.4d.

From these distributions h_σ, it is observed that they have a similar behavior as $h_{c,\sigma}$: their mode increases up to some noise level, and then decreases slowly. Moreover, the variance of each distribution increases proportionally with the noise level, similar to the case of the noisy constant image. However, there are some differences between the distributions $h_{c,\sigma}$ and h_σ: at low noise levels ($\sigma \leq 20$), the modes of the distributions h_σ are closer to each other than the modes of the distributions $h_{c,\sigma}$: this is due to the fact that the natural images are textured, and thus impact the α-tree structure, unlike constant images which only have one region covering the single partition of the hierarchy when there is no noise. Furthermore, there are differences between the distributions from the two natural images: the variance is larger for the low-brightness image than

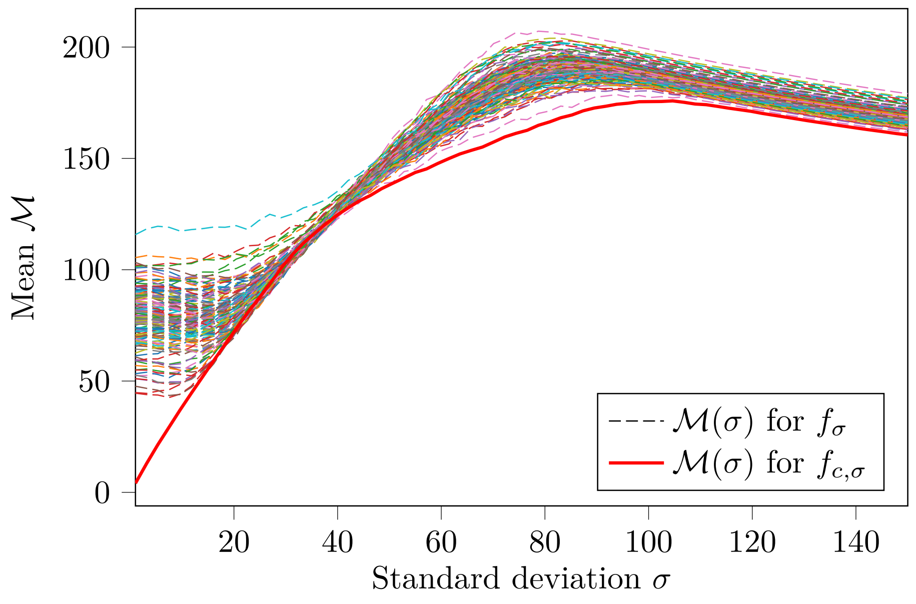


Figure 8.5: Comparison between the evolution of the average mean $\mathcal{M}(\sigma)$ from α-trees built on all the 150 natural images and the constant image

for the textured image, and the mode of the distributions in Figure 8.4b has a greater interval than for the textured image. Thus, even if the distributions h_σ are similar to the distributions $h_{c,\sigma}$, they differ according to the content of the image they are built upon.

Finally, these observations of the distributions are extended to the whole 150 images. To this aim, each image of the database is corrupted several times, and for each f_σ, a depth distribution is computed. The average mean of the distribution $\mathcal{M}(\sigma)$ at a given noise level σ is calculated such that

$$\mathcal{M}(\sigma) = \frac{1}{N}\sum_{i=1}^{N}\mu(h_i) \tag{8.4}$$

with N the number of times each image is corrupted and h_i the distribution computed for the i^{th} noisy image. For the evolution of $\mathcal{M}(\sigma)$ displayed in Figure 8.5, N is fixed to 20 in order to have enough samples to give robust results. In addition to using all the natural images, $\mathcal{M}(\sigma)$ is also computed for the constant noisy images $f_{c,\sigma}$ to provide a comparison between the depth distribution computed from natural images and pure noise.

The average $\mathcal{M}(\sigma)$ is lower for $f_{c,\sigma}$ at low noise levels, and this is almost true for all the levels σ. Furthermore, it is observed that, at low noise levels, the average mean depth is higher but their distribution over the 150 images of the database is also wide, the majority ranging from 50 to 100. This comes from the fact that the content of the image is still prevailing over the noise for the construction of the tree. However, while the noise level increases, the average mean plots become narrower. Meanwhile, the average means increase up to a certain noise level, which varies depending on the image, after which they all start to decrease. This may come from the clipping of the values at high noise levels, as it has been observed previously. Finally, it can be concluded from these plots that the noise in an image impacts the structure of the tree starting from a given noise level (ranging between 20 and 50 for all the images): $\mathcal{M}(\sigma)$ has a similar evolution for f_σ and $f_{c,\sigma}$, meaning that the noise takes precedence over the image content.

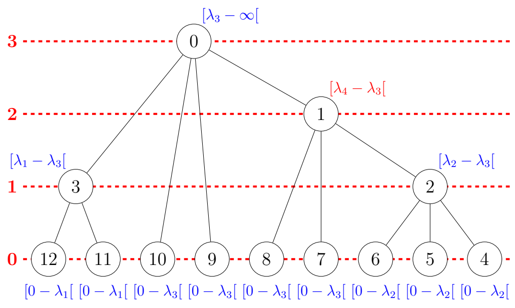
8.2.3 Extension to the context of persistent hierarchies

In this part, the previous study is extended to persistent hierarchies. More precisely, the amount of non-persistent nodes according to a given energy is studied. Thus, we first start by recalling the notion of persistent hierarchy according to an energy.

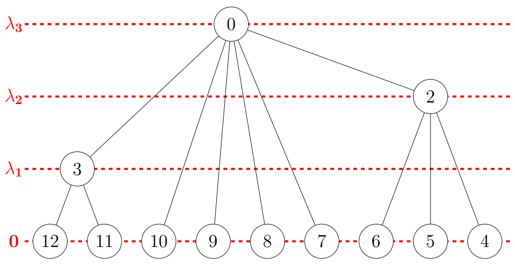
Persistent hierarchy

As explained in Chapter 4, each node n of a partitioning hierarchy is associated with a given index. This index is the value related to the lowest partition P to which a region R represented by n in the tree belongs, and this index is named the *scale of appearance*. In the case of the α-tree, the value associated with each node is the value α of its component. In a similar manner, each node is associated with a scale of disappearance, which is the lowest value associated with the partition to which R does not belong anymore. Thus, the set of values for a region to belong to a partition of the hierarchy is continuous and it defines the *interval of persistence*, denoted by $[\lambda^+(R), \lambda^-(R)[$ for a region R with $\lambda^+(R)$ the scale of appearance and $\lambda^-(R)$ the scale of disappearance. The root node of the tree, representing the region covering the whole domain Ω of the image is a special case where $\lambda^-(\Omega) = +\infty$.

Several image processing tasks such as segmentation rely on an energy minimization procedure, and these work efficiently with hierarchical representations of images. An application using the Mumford-Shah functional on the tree of shapes, an inclusion hierarchy, is demonstrated in Chapter 7 for segmentation purposes. On partitioning hierarchies, Guigues *et al.* [83] propose to use such energy to obtain optimal cuts $\mathbf{C}^*$ according to this energy. They a


$[\lambda_3 - \infty[$
3
$[\lambda_4 - \lambda_3[$
2
$[\lambda_1 - \lambda_3[$
$[\lambda_2 - \lambda_3[$
1
0
0
1
3
2
12
11
10
9
8
7
6
5
4
$[0 - \lambda_1[$ $[0 - \lambda_1[$ $[0 - \lambda_3[$ $[0 - \lambda_3[$ $[0 - \lambda_3[$ $[0 - \lambda_3[$ $[0 - \lambda_2[$ $[0 - \lambda_2[$ $[0 - \lambda_2[$

(a) Persistence interval on a tree



(b) Persistence hierarchy of (a)

Figure 8.6: Tree to persistent hierarchy process

separable energy of the form:

$$E_\lambda(\mathbf{C}^*) = \sum_{R_i \in \mathbf{C}^*} D(R_i) + \lambda \sum_{R_i \in \mathbf{C}^*} C(R_i) \tag{8.5}$$

with $D(R_i)$ a data-fidelity term to R_i, $C(R_i)$ a regularization term and λ a parameter of this energy. Such energies include the piecewise-constant Mumford-Shah functional [142] used in the previous chapter. When the parameter λ varies from low values to high values, the optimal cut $\mathbf{C}^*$ evolves from partitions with fine regions to partitions with coarse regions. Thus, this parameter acts as a scale parameter. By the means of a functional dynamic problem [83], the scale of appearance $\lambda^+(R^*)$ and the scale of disappearance $\lambda^-(R^*)$ for all regions R^* in all the optimal cuts $\mathbf{C}^*_\lambda$ of a hierarchy constrained by a separable energy E_λ can be computed. Some regions R of the hierarchy do not belong to any optimal cut $\mathbf{C}^*_\lambda$, with thus $\lambda^+(R) > \lambda^-(R)$. These regions are said to be *non-persistent* and are removed from the hierarchy. Finally, the filtered hierarchy whose regions are related to the scale of appearance according to E_λ is named the persistent hierarchy according to E_λ and is the hierarchy of optimal cut according to E_λ.

An example of the computation of a persistent hierarchy is displayed in Figure 8.6. In Figure 8.6a, the persistence interval according to some given separable energy is computed. The scale values are ordered such that $0 < \lambda_1 < \lambda_2 < \lambda_3 < \lambda_4 < +\infty$. Thus, the node 1 is non-persistent due to the fact that $\lambda^+(R_1) > \lambda^-(R_1)$. It is removed from the hierarchy when the latter is transformed into a persistent hierarchy in Figure 8.6b.

Evolution of the amount of non-persistent nodes

To observe the impact of the noise on the amount of non-persistent nodes, two energies are considered. These energies use two major informations about a

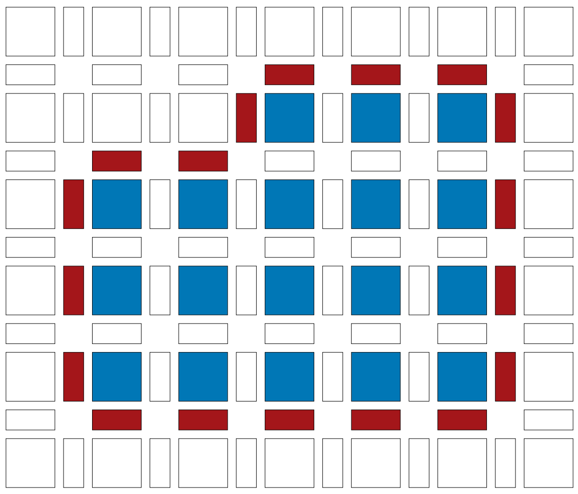


Figure 8.7: Representation of a region R with its contour

region R: its pixel values, and its contours information. This is illustrated in Figure 8.7, in which a region R is displayed by taking into account the inter-pixel space. It has to be noted that this illustration is valid in the context of this chapter as the α-trees are built using a 4-adjacency relationship. In this illustration, the pixels of the region are displayed as blue squares and their contour as red rectangle. The values of the region are denoted by f as they from the image pixel values, and the values of the contours are denoted by ∂f. In this chapter, the value associated to ∂f is the L_1 distance between two adjacent pixels.

The first energy that we consider is the piecewise-constant Mumford-Shah functional [142], defined on each region R_α of the tree $\mathcal{T}_\alpha$ by

$$E_{ms,\lambda}(R_\alpha) = \sum_{p \in R_\alpha} \|f(p) - \tilde{f}\|_2^2 + \lambda |\partial R_\alpha| \tag{8.6}$$

with $\tilde{f}$ the mean value of the region R_α, λ the parameter of the functional and ∂R_α the contours of the region R_α. For the second functional, we propose to adapt the Mumford-Shah functional such that it takes into account the dissimilarity on the contours of a region. This new functional is defined by

$$E_{cs,\lambda}(R_\alpha) = \sum_{p \in R_\alpha} \|f(p) - \tilde{f}\|_2^2 + \lambda \sum_{p \in \partial R_\alpha} \partial f(p) \tag{8.7}$$

with ∂f the set of gradient values of two adjacent pixels of the image forming the contours of the regions of the α-tree. This change of regularization term is proposed because a region with a small variance and a high gradient along its contours is most likely to be constrasted relatively to its adjacent regions, and

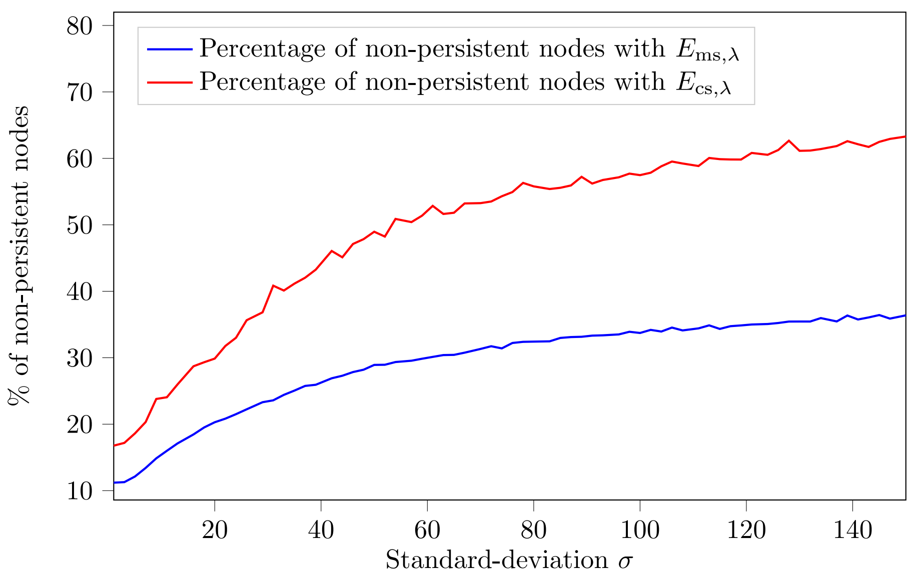
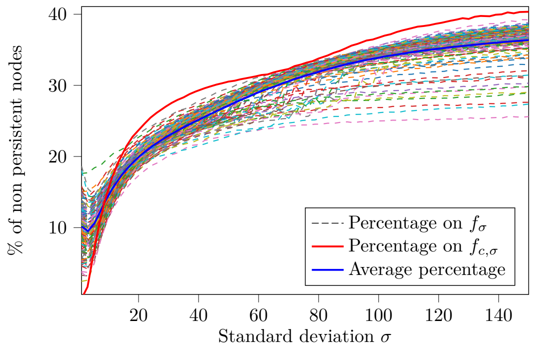


Figure 8.8: Comparison of the percentage of non-persistent nodes in the image displayed on Figure 8.4c

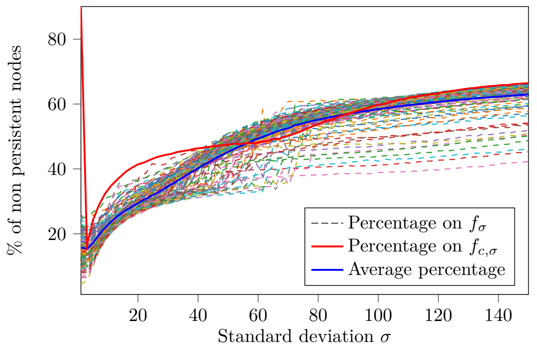
therefore prone to be less affected by the noise in the image on which the α-tree is built.

Figure 8.8 provides a comparison of the impact of these two functionals on an α-tree built on a noisy version of the image displayed in Figure 8.4c according to different noise levels. These two plots display the percentage of non-persistent nodes in the α-tree such that they are removed when the persistent hierarchy is computed according to a given energy. They have a similar behavior: the percentage of non-persistent nodes increases with the noise level σ. Additionally, this amount is greater when $E_{cs,\lambda}$ is used as an energy criterion than $E_{ms,\lambda}$, and this difference is twice larger for $E_{cs,\lambda}$ at high noise levels.

This comparison is then extended to all 150 images from the Laurent Condat dataset in a similar manner to the previous part, but instead of using the average mean, the average percentage of non-persistent nodes is computed. The results are displayed in Figure 8.9. In this figure, the two energies $E_{ms,\lambda}$ and $E_{cs,\lambda}$ are used respectively in Figures 8.9a and 8.9b. All individual images are plotted in dashed lines, and their average is the blue line. Furthermore, the evolution in the case of a pure noise image is also observed (red plot). For the two energies, the percentage of non-persistent nodes is increasing similarly for the natural images. Furthermore, this percentage is evolving similarly for the pure noise, except in the case $\sigma = 1$ for $E_{cs,\lambda}$, in which more than 80% of the nodes are removed while it is close to 0% for $E_{ms,\lambda}$. However, the amount of removed nodes is not the same for the two plots: at high noise levels, the average percentage is around



(a) Percentage of non-persistent nodes for $E_{ms,\lambda}$



(b) Percentage of non-persistent nodes for $E_{cs,\lambda}$

Figure 8.9: Evolution of the percentage of non-persistent nodes under $E_{ms,\lambda}$ and $E_{cs,\lambda}$ related to the noise level σ

30% for $E_{ms,\lambda}$ while it is almost 60% for $E_{cs,\lambda}$. Finally, this average percentage is closer to the one of non-persistent nodes from the tree built on $f_{c,\sigma}$ when $E_{cs,\lambda}$ constrains the hierarchy compared with the usage of $E_{ms,\lambda}$. These observations suggest that using $E_{cs,\lambda}$ in the presence of noise is more relevant than using $E_{ms,\lambda}$: a large amount of regions that are not robust to the noise according to the definition of the proposed functional is removed from the persistent hierarchy such that only robust regions are kept.

8.3 Applicability of the study

In the previous section, a structural study of the noise was performed and it resulted in some information such as the fact that the choice of the functional is important in terms of non-persistent nodes. In this section, the knowledge acquired previously is used to design a pipeline to compute a dissimilarity measure between two trees from their geometry. This measure is then used to look for a relation between the noise level of an image and the dissimilarity between a tree built on a clean image and a tree built on a noisy image. It has to be noted that this section describes the preliminary results of a work in progress.

8.3.1 Tree dissimilarity

Designing a measure of dissimilarity between two hierarchical representations is challenging as several parameters may be taken into account. First, the structure of the trees differs depending on the image content they originate.

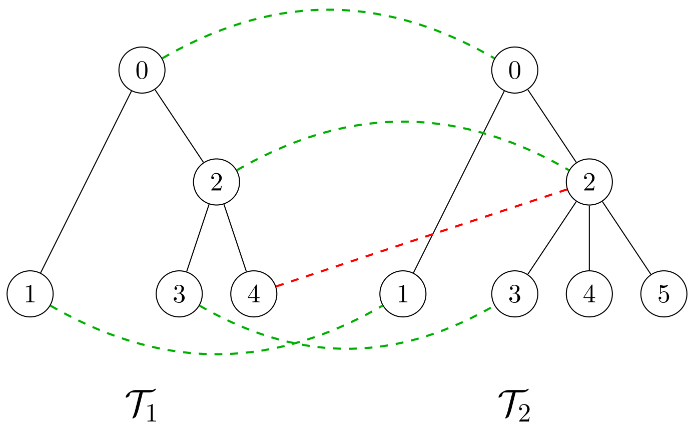


Figure 8.10: Pairing nodes of the tree with invalid pairing

The previous section takes into account this structure to study the evolution of a tree related to the noise level. Another property of the tree is the geometrical information provided by each node as they represent a region from a partition of the hierarchy in the context of partitioning hierarchies.

Several measures exist to evaluate the quality of object detection or image segmentation tasks [170]. Among them, the Jaccard index, which is a similarity measure for two regions R_1 and R_2 from two segmentations, is defined by

$$J(R_1, R_2) = \frac{|R_1 \cap R_2|}{|R_1 \cup R_2|} \tag{8.8}$$

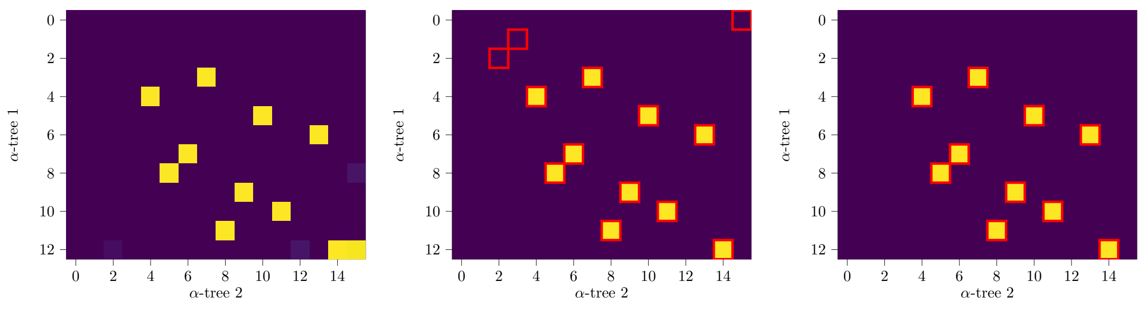
and its formulation as a distance is defined by

$$d_J(R_1, R_2) = 1 - J(R_1, R_2) \tag{8.9}$$

Such a distance may be used to compute the pairwise dissimilarity between each region of two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ that results in a matrix D such that:

$$(D)_{i,j} = d_J(R_1^i, R_2^j) \tag{8.10}$$

for two regions R_1^i and R_2^j respectively represented by nodes in the trees $\mathcal{T}_1$ and $\mathcal{T}_2$. The pairing of a region related to a node of the tree $\mathcal{T}_1$ to a region related to a node of the tree $\mathcal{T}_2$ may be performed by selecting the lowest distance for each line of the dissimilarity matrix. This is illustrated in Figure 8.10 where the nodes from $\mathcal{T}_1$ are paired with nodes from $\mathcal{T}_2$. To compute a dissimilarity between two trees, this pairing may be unique such that the relation between a region $R_1 \in \mathcal{T}_1$ and a region $R_2 \in \mathcal{T}_2$ is bijective. Thus, the dissimilarity measure is not biased by the pairing of two regions from $R_1^i, R_1^j \in \mathcal{T}_1$ with the same regions $R_2 \in \mathcal{T}_2$ such



(a) Dissimilarity matrix (b) LSAP (c) Filtering

Figure 8.11: Pairing selection methodology

that the difference $|R_1^i| - |R_1^2|$ is too small to be significant. This is illustrated by the red dashed link in Figure 8.10. Such a case may include the merging of a region with a unique pixel in the context of the study of the noise impact in partitioning hierarchies.

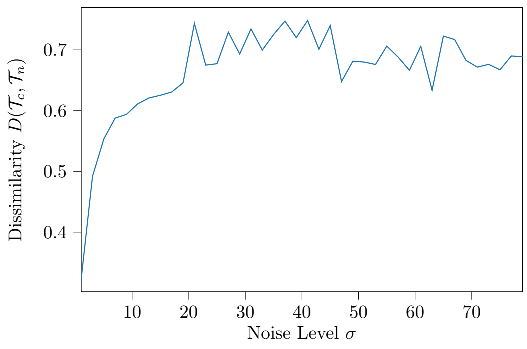
To avoid the case where several nodes from $\mathcal{T}_1$ are paired with the same node in $\mathcal{T}_2$, a linear sum assigment problem (LSAP) is used. It is formulated by:

$$\min_{i,j} \sum_i \sum_j (D)_{i,j} \tag{8.11}$$

such that i and j are the indices of respectively a region of $\mathcal{T}_1$ and $\mathcal{T}_2$, and their relation in the pairing is bijective. This problem is solved by the usage of the *Hungarian algorithm* [114]. This is illustrated in Figure 8.11 where the dissimilarity matrix from Figure 8.11a, whose values go from yellow ($(D)_{i,j} = 0$) to purple ($(D)_{i,j} = 1$), is given as input to the Hungarian algorithm such that the red square in Figure 8.11b are the selected dissimilarity measures for the bijective pair of nodes from two trees. Finally, it may be observed in Figure 8.11b that some nodes are paired with a dissimilarity of 1, meaning that $|R_1^i \cap R_2^j| = 0$. To avoid that these pairs add a bias to the dissimilarity measure, these are filtered out from the set of pairs, resulting in the paired values in the dissimilarity matrix in Figure 8.11c.

From the set of pairs obtained by the pairing process, a dissimilarity value can be computed. There exist several means to combine the Jaccard distances into one value. First of all, the sum of the dissimilarity values from all the pairs results in a measure where the number of pairs is not taken into account. On the other hand, to compute the dissimilarity $\mathcal{D}(\mathcal{T}_1, \mathcal{T}_2)$ between two trees, the empirical mean is used instead. It has to be noted that the root node of a tree is never taken into account in the computation of the mean since the root node of $\mathcal{T}_1$ is always paired with the root node of $\mathcal{T}_2$, whatever the content of the image due to the fact that it represents a region covering the image domain Ω.

(a) An image



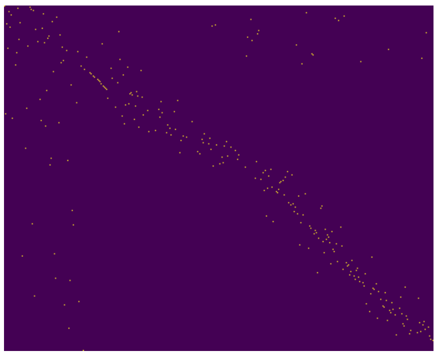
(b) Relationship between the noise level and our dissimilarity $\mathcal{D}(\mathcal{T}_c, \mathcal{T}_n)$

Figure 8.12: Impact of the noise on the geometrical information of the tree

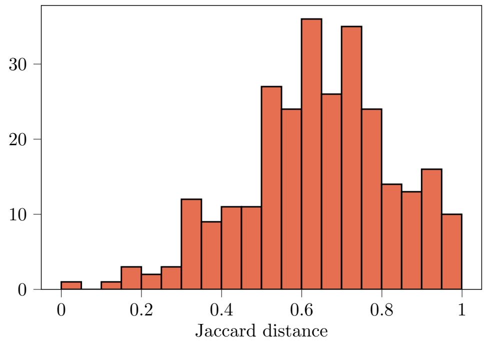
In terms of computationability, the cost of performing such a process is expensive. Furthermore, if the input tree is binary, it has been proven that such a process is computationally unfeasible [172]. To this aim, two pre-processing steps are performed: first, the input tree is filtered with an area criterion [165] whose threshold is usually set to 200. Thus, only significant regions for many image processing applications are kept. Then, from the knowledge acquired in the previous section, the non-persistent nodes from the tree according to the energy $E_{cs,\lambda}$ are removed, allowing to only keep the nodes that are robust to the noise. The computation of $\mathcal{D}(\mathcal{T}_1, \mathcal{T}_2)$ is performed on this filtered tree.

8.3.2 Observation and preliminary results

In this section, the process computing $\mathcal{D}(\mathcal{T}_1, \mathcal{T}_2)$ is evaluated. In order to compute this measure, the image on Figure 8.12a is corrupted by a Gaussian noise whose standard deviation σ goes from 0 to 80. Two α-trees, denoted by $\mathcal{T}_c$ and $\mathcal{T}_n$, are respectively built on the clean and noisy versions of the image. Then, the distance $\mathcal{D}(\mathcal{T}_c, \mathcal{T}_n)$ is computed on these two trees. Figure 8.12b displays the value $\mathcal{D}(\mathcal{T}_c, \mathcal{T}_n)$ according to the noise level σ of the noisy image. The plot may be divided two parts: first, the value $\mathcal{D}(\mathcal{T}_c, \mathcal{T}_n)$ is increasing when σ ranges from 0 to 20. This part of the plot, when put in relation with section 8.2, may be compared with the results from Figure 8.5, where the average mean values of the depth distributions of an image are not correlated from the ones of the pure noise for the same noise level range. As concluded in section 8.2.2, the content of the image still prevails for this range of noise levels, but the impact of the noise is growing up in the image and this is justified by the evolution of the plot. The second part of the plot varies with a behavior which does not permit to find



(a) Selected pairs from the dissimilarity matrix



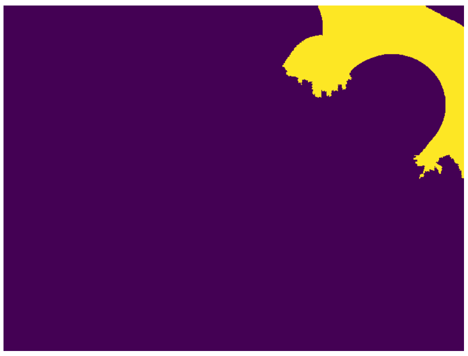
(b) Histogram of dissimilarity values

Figure 8.13: Observation of the dissimilarity computation process on the image in Figure 8.12a corrupted with $\sigma = 20$

any relationship between the noise level and the geometrical dissimilarity. This correspond to the part of the plots in Figure 8.5 in which the evolution of the average means of the depth distributions computed from the natural images and the pure noise is similar.

To analyze the plot obtained previously, the results of the pairing process in the computation of $\mathcal{D}(\mathcal{T}_c, \mathcal{T}_n)$ are displayed in Figure 8.13. These elements of the dissimilarity computation are obtained with a noisy version of the image displayed in Figure 8.12a whose noise level is set to 20. The selected pairs of regions have a dissimilarity located, for most of them, on the diagonal of the matrix, as illustrated in Figure 8.13a. However, the distribution of the Jaccard distance values, displayed in Figure 8.13b, informs us that the first quartile is located at 0.527, indicating that only 25% of the values fall below this threshold, and only 2.5% of the these values are below 0.2. Thus, the majority of the paired regions have a large distance between them such that the noise impacts the geometry of the regions and reduces the amount of similar regions from the trees built on the clean and noisy images.

Finally, we observe two selected pairs in Figure 8.13a. They are chosen to analyze the cases the Jaccard distance is low and high. In Figures 8.14 and 8.15, the selected regions are highlighted in yellow. Regions R_c and R_n respectively belongs to $\mathcal{T}_c$ and $\mathcal{T}_n$. The first pair, displayed in Figure 8.14, is selected with $d_J(R_\text{c}, R_\text{n}) = 0.1015$. These regions are similar in terms of distance, but we can also observe that they have the same shape. However, the contour of R_n in Figure 8.14b has irregularities compared with the one of R_c in Figure 8.14b. The second pair displayed in Figure 8.15 is composed of two regions with different shapes: that explains the fact that the Jaccard distance $d_J(R_\text{c}, R_\text{n}) = 0.9496$. This


(a) Region R_c from clean image

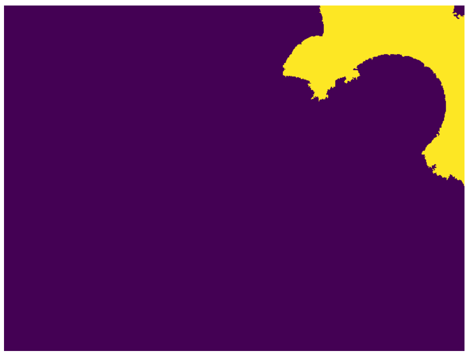
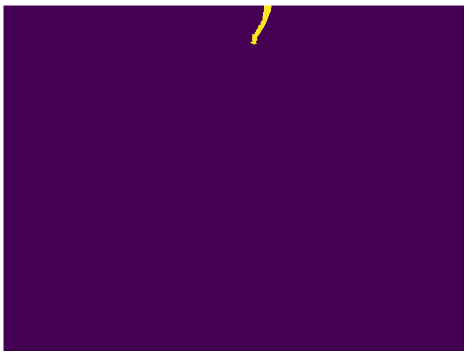

(b) Region R_n from noisy image

Figure 8.14: Paired regions with a low Jaccard distance ($d_J(R_c, R_n) = 0.1015$)


(a) Region R_c from clean image

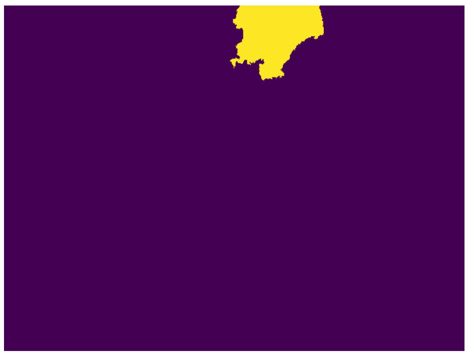

(b) Region R_n from noisy image

Figure 8.15: Paired regions with a high Jaccard distance ($d_J(R_c, R_n) = 0.9496$)

illustrates the main default of our method: it takes into account pairs which have a single pixel in common.

8.4 Conclusion and perspectives

In this chapter, we performed a study of the impact of the noise in an image on the hierarchy built on it. We looked for a relationship between the noise level of an image and some attributes computed from the tree, using either some information from its structure or taking into account the geometry of the regions represented by the hierarchy. In the first part, we made some observations on the distribution of the depth attribute of every node, and we showed that the content of an image prevailed until a given noise level for the tree structure. Then, we extended our study to the context of persistent hierarchies according to a given energy. We proposed to extend the Mumford-Shah functional by taking into account the contour values of a region and we compared both when the noise level of an image varies. We observed that the usage of this new functional removed more nodes in the hierarchy that are not robust to the noise according to our definition than the Mumford-Shah functional. Finally, we applied the knowledge from the structural study to design a dissimilarity measure between two trees by the use of the geometrical information of each node. We used this measure to compute the geometrical difference for a wide range of noise parameter between the nodes from a tree built from a clean image and another tree from a noisy one.

This study is still a work in progress but some preliminary results were obtained. First, we observed a correlation between the resulting relationship and the study of the depth distributions of the whole database: when the evolution of the empirical means of the depth distributions obtained from the natural images is different from the ones of the pure noise, the dissimilarity is increasing. However, such an observation has only been performed on one image. As a next step, the dissimilarity computation will be applied on the whole image database and further observations will be performed. Furthermore, we observed two cases in which regions were paired. In the first case, the Jaccard distance between the two regions was low, and the regions were similar. In the second case, the regions had different shapes. An interesting improvement for the computation of the dissimilarity matrix would be to insert the contour information in the computation of its values.

Chapter 9

Conclusion and Perspectives

This thesis investigated three criteria that an image processing library should respect (genericity, performance, and interactivity), and more specifically the different solutions to succeed at most their combination in a library specialized in Mathematical Morphology. Such a library is then used to build hierarchical representations from images corrupted by noise for two applications. In this chapter, we briefly recall the conclusion from these two contribution axes and we study their perspectives.

9.1 Genericity, efficiency and interactivity for an image processing library

The combination of genericity, performance, and interactivity for data structures and algorithms in an image processing library is a challenging issue. In Chapter 5, a methodology is presented to perform *static-dynamic* genericity in static languages for image processing such that an algorithm may be implemented to handle any kind of 2D images, whatever their value type and implementation are known at compile-time or runtime. This method, first presented in C++, is then compared to the Rust programming language, the latter having some facilities for this purpose such as boxing for runtime polymorphism. Finally, the different image structures proposed are applied to different algorithmic schemes, implemented by means of static-dynamic genericity, and compared in terms of performance. From these comparisons, we concluded that the knowledge of type information at compile-time is important when the algorithmic pattern follows the order in memory in which the values of an image are stored due to some compiler optimization, but it is negligible when this order is not known in advance, as is the case for the max-tree construction.

A max-tree construction algorithm is then studied and its implementation following the static-dynamic genericity methodology is then described in Chapter 6. Three strategies are described for the image structures to be used in the max-tree algorithm implementation: the structures from Chapter 5, the *projection*, and the *conversion*. These have advantages and drawbacks: for example, the conversion creates a new image whose value type and implementation information are known at compile-time and then copies the values from the input image to the new one, but requires allocating a new image, resulting in an overhead in the runtime speed and memory. Finally, these different strategies are compared with other implementations of the max-tree construction algorithm and result in similar performance as the max-tree construction implementation from the Higra library.

Various perspectives may be considered for this first axis:

Extension to various domains - The work presented in this chapter is limited to n-dimensional images defined on hyperrectangles. However, as explained in Chapter 3, various domains may be considered such as the edge set or vertex set of a graph, or the triangle set of a 2-simplicial complex. An efficient and generic way to represent the elements of an image domain is the usage of a univariate index value so that only one type of point may be taken into account for n-dimensional points but also to locate edges, vertices, or triangles. In that case, some optimization such as small buffer optimization (SBO), which would dynamically allocate the table storing the coordinates of each point of a domain for a large number of dimensions, becomes useless and can be avoided by the index representation coordinates.

Parallelization - Static-dynamic genericity has an overhead for every algorithm studied in Chapter 5 but it is more or less important depending on the underlying pattern. For all of them, it is possible to divide the computation into several blocks and perform the operations on each of these blocks. Such a methodology is called tiling. Merge-based max-tree construction algorithms rely on tiling to build one max-tree per block and then merge them to obtain the max-tree of the whole image. The usage of tiling may be used on dynamic images, and thus provide highly performant implementations of image processing algorithms while keeping interactivity and genericity.

External modules and Just-In-Time (JIT) compilation - Static-dynamic genericity relies on operations performed on the image values which may be selected either at runtime or at compile-time depending on the knowledge of type information. These still require to be defined and implemented before the usage of an algorithm, and thus to be compiled. Two solutions may be investigated to reduce the amount of operations contained in the main library: external modules and Just-In-Time (JIT) compilation. External modules are compiled libraries, independent of the main library, containing the operations covering a large set of type combinations for each operation. Such libraries may be linked to the main one on demand when an operation is not available. Furthermore, these modules may not be contained in the package of the library, and thus downloaded when needed. The second solution is the usage of JIT compilation requiring the library to embed a compiler and generate the machine code for each operation when it is necessary. Thus, compared with external modules, operations are not required to be defined but use the dynamic type information to be generated. These two solutions have several advantages: they result in a lightweight main library, but may also provide some optimized operations with SIMD instructions, difficult for the compiler to generate when the type information of the pixel values is dynamic. Furthermore, the usage of JIT compilation to obtain vectorized machine code is a benefit since SIMD instructions are usually platform-dependent and add a parameter to the combinatorial of an operation.

Construction of hierarchical representations of images - In Chapter 6, a max-tree construction algorithm based on the union-find data structure is studied as an application of static-dynamic genericity. The union-find data structure is used for different construction algorithms such as the Kruskal minimum spanning tree, which may be adapted to build the α-tree or the hierarchical watersheds of an image. Compared with the max-tree construction, which uses the pixel values to build the hierarchy, the Kruskal algorithm is based on an edge-weighted graph obtained from the adjacency graph of an image whose edges are weighted using the gradient of an image. Thus, in conjunction with the extension of static-dynamic genericity to different domains, it may be interesting to study the construction of other hierarchies with this methodology.

9.2 Hierarchical representations of images in the presence of noise

As an application, hierarchical representations are used for noise level estimation, and more specifically for segmentation purposes to obtain homogeneous regions from which the noise can be estimated. In Chapter 7, a method to estimate the noise level function of a grayscale image, returning the noise variance according to the intensity of an image pixel, is extended to multivariate images by incorporating two tools from mathematical morphology: the multivariate tree of shapes and a rank map obtained from a complete lattice learning. This estimation is then compared to a block-based estimation, and the precision of the estimation using such a content-adapted estimation shows better results than using blocks. However, this method only uses a subset of regions from the hierarchy by computing an optimal segmentation according to the Mumford-Shah functional, so it is extended to the whole set of regions of this hierarchy. The resulting estimation does not result in any improvements, such that the impact of the noise contained in an image on which a hierarchy is built is questioned.

To this aim, the evolution of the structure of the α-tree is observed by the mean of the depth attribute of each node, which is the number of nodes on the path from the current one to the root of the tree. By varying the noise level corrupting an image, different distributions of the depth computed as histograms are compared, and the impact of the noise contained in an image on the structure of the tree is highlighted by comparing the empirical means of each distribution with the ones obtained from pure noise without any content. Furthermore, this impact is studied in the context of an optimal hierarchy according to a given energy, and it is concluded that the energy chosen influences the amount of non-persistent nodes, which are nodes that are not in an optimal cut of the tree.

For this axis, several open problems may be tackled:

Improvements and applicability of the noise level estimation - The noise level estimation presented in Chapter 7 is performed channelwise, meaning that the correlation between the channels is not taken into account. In practice, the demosaicking step of the image acquisition pipeline may mix the noise information in the different channels such that a noise correlation between the channels of an image may exist. In that case, the MNLF should take into account such information and the estimation has to be adapted to this improved MNLF. Furthermore, the choice of the usage of a noise level estimation on multivariate images or grayscale images depends on the well-known problem of the choice of the processing order for demosaicking and denoising in the acquisition pipeline. Thus, the estimation will be integrated into a denoising method and the two cases considered previously will be tested to assess the best use case of our estimation.

Trees dissimilarity in the presence of noise - The study of the impact of the noise contained in an image on the structure a hierarchy has interesting results but may not be exploitable directly to improve the performance of an application such as segmentation or denoising using hierarchical representations of images. In Chapter 8, a sketch of a method to compute the dissimilarity between two hierarchies built from images in the presence of noise is presented. The computation of this dissimilarity measure selects nodes using the information acquired from the study on the structure of the tree as the usage of all the nodes for pairwise dissimilarity makes the computation unfeasible. Indeed, the objective of this dissimilarity is to create a new measure that highlights the relationship between the noise level of an image and its influence on a hierarchy. To this aim, this measure may be computed from two trees, one built from a clean image and another one from a noisy version of this image. In that way, further investigations will be performed to improve this dissimilarity measure and propose different applications relying on it.

Applications of the trees dissimilarity - The usage of such a measure may be useful for different applications. First of all, the diverse use cases relying on hierarchical representations of images such as segmentation, object detection, or filtering may be improved by taking into account the noise impact on the hierarchy. Furthermore, new applications may be developed: denoising has been little investigated when performed using these representations. Furthermore, the development of new loss functions for neural networks leads to the questioning about the usefulness of the hierarchies in the presence of noise. Recently, the max-tree has been used to create a new loss function based on attribute values and it has been applied for image filtering purposes. Thus, a new denoising method should be investigated using the dissimilarity measure between the two trees with the condition that the relationship ensuing from this measure between the hierarchies of the clean and noisy versions of an image is differentiable.

List of publications

International conferences

- **Baptiste Esteban**, Guillaume Tochon, Edwin Carlinet, and Didier Verna. "Estimation of the noise level function for color images using mathematical morphology and non-parametric statistics". In: *2022 26th International Conference on Pattern Recognition (ICPR)*. 2022, pp. 428–434. DOI: 10.1109/ICPR56361.2022.9956218

- **Baptiste Esteban**, Edwin Carlinet, Guillaume Tochon, and Didier Verna. "The Cost of Dynamism in Static Languages for Image Processing". In: *Proceedings of the 21st International Conference on Generative Programming: Concepts & Experiences (GPCE 2022)*. Auckland, New Zealand, Dec. 2022. DOI: 10.1145/3564719.3568693

- **Baptiste Esteban**, Guillaume Tochon, Edwin Carlinet, and Didier Verna. "Structural Analysis of the Additive Noise Impact on the α-tree". In: *Proceedings of the 20th International Conference on Computer Analysis of Images and Patterns (CAIP 2023)*. Vol. 14185. Lecture Notes in Computer Science Series. To Appear. Limassol, Cyprus: Springer, Sept. 2023

National conferences

- **Baptiste Esteban**, Guillaume Tochon, Edwin Carlinet, and Didier Verna. "Estimation de la fonction de niveau de bruit pour des images couleurs en utilisant la morphologie mathématique". In: *28e Colloque sur le traitement du signal et des images*. Nancy, France: GRETSI - Groupe de Recherche en Traitement du Signal et des Images, Sept. 2022, pp. 953–956

- **Baptiste Esteban**, Edwin Carlinet, Guillaume Tochon, and Didier Verna. "Généricité dynamique pour des algorithmes morphologiques". In: *28e Colloque sur le traitement du signal et des images*. Nancy, France: GRETSI - Groupe de Recherche en Traitement du Signal et des Images, Sept. 2022, pp. 477–480
- **Baptiste Esteban**, Guillaume Tochon, Edwin Carlinet, and Didier Verna. "Analyse structurelle de l'influence du bruit sur l'arbre alpha". In: *29e Colloque sur le traitement du signal et des images*. Grenoble, France: GRETSI - Groupe de Recherche en Traitement du Signal et des Images, Aug. 2023

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/ (visited on 10/13/2023).

[2] David Abrahams and Stefan Seefeld. *Boost.Python*. URL: http://www.boost.org/libs/python/ (visited on 12/06/2023).

[3] Radhakrishna Achanta et al. *Slic superpixels*. Tech. rep. 2010.

[4] Theodore Aouad and Hugues Talbot. "Binary Morphological Neural Network". In: *2022 IEEE International Conference on Image Processing (ICIP)*. 2022, pp. 3276–3280. DOI: 10.1109/ICIP46576.2022.9897474.

[5] E. Aptoula and S. Lefèvre. "A comparative study on multivariate mathematical morphology". In: *Pattern Recognition* 40.11 (2007), pp. 2914–2929. DOI: 10.1016/j.patcog.2007.02.004.

[6] Pablo Arbeláez, Michael Maire, Charless Fowlkes, and Jitendra Malik. "Contour Detection and Hierarchical Image Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.5 (2011), pp. 898–916. DOI: 10.1109/TPAMI.2010.161.

[7] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[8] Asra Aslam, Ekram Khan, Mohammad Samar Ansari, and M. M. Sufyan Beg. *A Novel Falling-Ball Algorithm for Image Segmentation*. 2021. DOI: 10.48550/arXiv.2105.02615.

[9] Coloma Ballester, Vicent Caselles, Laura Igual, and Luis Garrido. "Level Lines Selection with Variational Models for Segmentation and Encoding". In: *Journal of Mathematical Imaging and Vision* 27.1 (2007), pp. 5–27. ISSN: 1573-7683. DOI: 10.1007/s10851-006-7252-0.

[10] John Barnes. *Programming in Ada 2012 with a Preview of Ada 2022*. 2nd ed. Cambridge University Press, 2022. DOI: 10.1017/9781009181358.

[11] L. Beaurepaire, K. Chehdi, and B. Vozel. "Identification of the nature of noise and estimation of its statistical parameters by analysis of local histograms". In: *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 4. 1997, 2805–2808 vol.4. DOI: 10.1109/ICASSP.1997.595372.

[12] David M Beazley. "Automated scientific software scripting with SWIG". In: *Future Generation Computer Systems* 19.5 (2003), pp. 599–609. DOI: 10.1016/S0167-739X(02)00171-1.

[13] David M Beazley et al. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++." In: *Tcl/Tk Workshop*. Vol. 43. 1996, p. 74.

[14] Stefan Behnel et al. "Cython: The Best of Both Worlds". In: *Computing in Science & Engineering* 13.2 (2011), pp. 31–39. DOI: 10.1109/MCSE.2010.118.

[15] Mikhail Belkin and Partha Niyogi. "Laplacian Eigenmaps for Dimensionality Reduction and Data Representation". In: *Neural Computation* 15.6 (2003), pp. 1373–1396. DOI: 10.1162/089976603321780317.

[16] Ch. Berger et al. "Effective Component Tree Computation with Application to Pattern Recognition in Astronomical Imaging". In: *IEEE International Conference on Image Processing*. Vol. 4. 2007, pp. IV –41–IV –44. DOI: 10.1109/ICIP.2007.4379949.

[17] Nicolas Beucher and Serge Beucher. "Mamba Image User Manual". In: *Mamba library website* (2017).

[18] Serge Beucher. "Watershed, Hierarchical Segmentation and Waterfall Algorithm". In: *Mathematical Morphology and Its Applications to Image Processing*. Ed. by Jean Serra and Pierre Soille. Dordrecht: Springer Netherlands, 1994, pp. 69–76. ISBN: 978-94-011-1040-2. DOI: 10.1007/978-94-011-1040-2_10.

[19] Serge Beucher and Fernand Meyer. "The morphological approach to segmentation: the watershed transformation". In: *Mathematical morphology in image processing* 34.1993 (1993), p. 49.

[20] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671.

[21] *Bloaty: a size profiler for binaries*. URL: https://github.com/google/bloaty (visited on 09/06/2023).

[22] Isabelle Bloch et al. “On Some Associations Between Mathematical Morphology and Artificial Intelligence”. In: *Discrete Geometry and Mathematical Morphology*. Ed. by Joakim Lindblad, Filip Malmberg, and Nataša Sladoje. Cham: Springer International Publishing, 2021, pp. 457–469. ISBN: 978-3-030-76657-3. DOI: 10.1007/978-3-030-76657-3_33.

[23] Petra Bosilj. *trees-lib*. URL: https://github.com/pbosilj/trees-lib (visited on 09/11/2023).

[24] Petra Bosilj, Ewa Kijak, and Sébastien Lefèvre. “Partition and inclusion hierarchies of images: A comprehensive survey”. In: *Journal of Imaging* 4.2 (2018), p. 33.

[25] Roger Bourne. “ImageJ”. In: *Fundamentals of Digital Imaging in Medicine*. London: Springer London, 2010, pp. 185–188. ISBN: 978-1-84882-087-6. DOI: 10.1007/978-1-84882-087-6_9.

[26] Nicolas Boutry and Guillaume Tochon. “Stability of the Tree of Shapes to Additive Noise”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2021, pp. 365–377. DOI: 10.1007/978-3-030-76657-3_26.

[27] Gary Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).

[28] A. Buades, B. Coll, and J. M. Morel. “A Review of Image Denoising Algorithms, with a New One”. In: *Multiscale Modeling & Simulation* 4.2 (2005), pp. 490–530. DOI: 10.1137/040616024.

[29] Nicolas Burrus et al. “A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming”. In: *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*. Anaheim, CA, USA, Oct. 2003.

[30] Luca Cardelli and Peter Wegner. “On Understanding Types, Data Abstraction, and Polymorphism”. In: *ACM Comput. Surv.* 17.4 (1985), pp. 471–523. ISSN: 0360-0300. DOI: 10.1145/6041.6042.

[31] Edwin Carlinet. “A Tree of shapes for multivariate images”. PhD thesis. Université Paris-Est, Nov. 2015.

[32] Edwin Carlinet, Sébastien Crozet, and Thierry Géraud. "The Tree of Shapes Turned into a Max-Tree: A Simple and Efficient Linear Algorithm". In: *2018 25th IEEE International Conference on Image Processing (ICIP)*. 2018, pp. 1488–1492. DOI: 10.1109/ICIP.2018.8451180.

[33] Edwin Carlinet and Thierry Géraud. "A Color Tree of Shapes with Illustrations on Filtering, Simplification, and Segmentation". In: *Mathematical Morphology and Its Applications to Signal and Image Processing*. Ed. by Jón Atli Benediktsson, Jocelyn Chanussot, Laurent Najman, and Hugues Talbot. Cham: Springer International Publishing, 2015, pp. 363–374. ISBN: 978-3-319-18720-4. DOI: 10.1007/978-3-319-18720-4_31.

[34] Edwin Carlinet and Thierry Géraud. "A comparative review of component tree computation algorithms". In: *IEEE Transactions on Image Processing* 23.9 (2014), pp. 3885–3895. DOI: 10.1109/TIP.2014.2336551.

[35] Edwin Carlinet and Thierry Geraud. "A Morphological Tree of Shapes for Color Images". In: *2014 22nd International Conference on Pattern Recognition*. IEEE, 2014. DOI: 10.1109/icpr.2014.204.

[36] Edwin Carlinet and Thierry Geraud. "Getting a morphological tree of shapes for multivariate images: Paths, traps, and pitfalls". In: *2014 IEEE International Conference on Image Processing (ICIP)*. IEEE, 2014. DOI: 10.1109/icip.2014.7025123.

[37] Edwin Carlinet and Thierry Géraud. "Intervertebral Disc Segmentation Using Mathematical Morphology—A CNN-Free Approach". In: *Computational Methods and Clinical Applications for Spine Imaging*. Ed. by Guoyan Zheng, Daniel Belavy, Yunliang Cai, and Shuo Li. Cham: Springer International Publishing, 2019, pp. 105–118. ISBN: 978-3-030-13736-6. DOI: 10.1007/978-3-030-13736-6_9.

[38] Edwin Carlinet and Thierry Geraud. "MToS: A Tree of Shapes for Multivariate Images". In: *IEEE Transactions on Image Processing* 24.12 (2015), pp. 5330–5342. DOI: 10.1109/tip.2015.2480599.

[39] Edwin Carlinet and Thierry Géraud. "Morphological object picking based on the color tree of shapes". In: *2015 International Conference on Image Processing Theory, Tools and Applications (IPTA)*. 2015, pp. 125–130. DOI: 10.1109/IPTA.2015.7367111.

[40] Antonin Chambolle and Thomas Pock. "A First-Order Primal-Dual Algorithm for Convex Problems with Applications to Imaging". In: *Journal of Mathematical Imaging and Vision* 40.1 (2010), pp. 120–145. DOI: 10.1007/s10851-010-0251-1.

[41] Ruey-Feng Chang, Chii-Jen Chen, and Chen-Hao Liao. “Region-based image retrieval using edgeflow segmentation and region adjacency graph”. In: *2004 IEEE International Conference on Multimedia and Expo (ICME)*. Vol. 3. 2004, 1883–1886 Vol.3. DOI: 10.1109/ICME.2004.1394626.

[42] Laurent Condat. *Laurent Condat's Image base*. URL: https://lcondat.github.io/imagebase.html.

[43] James O Coplien. “Curiously recurring template patterns”. In: *C++ gems*. 1996, pp. 135–144.

[44] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[45] Michel Couprie, Laszlo Marak, Hugues Talbot, et al. “Pink image processing library”. In: *Poster European Python Scientific Conference*. 2011.

[46] Jean Cousty, Gilles Bertrand, Laurent Najman, and Michel Couprie. “Watershed Cuts: Minimum Spanning Forests and the Drop of Water Principle”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31.8 (2009), pp. 1362–1374. DOI: 10.1109/TPAMI.2008.173.

[47] Jean Cousty and Laurent Najman. “Incremental Algorithm for Hierarchical Minimum Spanning Forests and Saliency of Watershed Cuts”. In: *Mathematical Morphology and Its Applications to Image and Signal Processing*. Ed. by Pierre Soille, Martino Pesaresi, and Georgios K. Ouzounis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 272–283. ISBN: 978-3-642-21569-8. DOI: 10.1007/978-3-642-21569-8_24.

[48] Jean Cousty, Laurent Najman, Yukiko Kenmochi, and Silvio Guimarães. “Hierarchical Segmentations with Graphs: Quasi-flat Zones, Minimum Spanning Trees, and Saliency Maps”. In: *Journal of Mathematical Imaging and Vision* 60.4 (2018), pp. 479–502. ISSN: 1573-7683. DOI: 10.1007/s10851-017-0768-7.

[49] Jean Cousty, Laurent Najman, and Benjamin Perret. “Constructive Links between Some Morphological Hierarchies on Edge-Weighted Graphs”. In: *Mathematical Morphology and Its Applications to Signal and Image Processing*. Ed. by Cris L. Luengo Hendriks, Gunilla Borgefors, and Robin Strand. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 86–97. ISBN: 978-3-642-38294-9. DOI: 10.1007/978-3-642-38294-9_8.

[50] Jean Cousty, Laurent Najman, and Jean Serra. “Raising in watershed lattices”. In: *2008 15th IEEE International Conference on Image Processing*. 2008, pp. 2196–2199. DOI: 10.1109/ICIP.2008.4712225.

[51] *Criterion.rs*. URL: https://bheisler.github.io/criterion.rs/book/index.html (visited on 09/06/2023).

[52] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. "Image Denoising by Sparse 3-D Transform-Domain Collaborative Filtering". In: *IEEE Transactions on Image Processing* 16.8 (2007), pp. 2080–2095. DOI: 10.1109/TIP.2007.901238.

[53] Beman Dawes and Alisdair Meredith. *Adopt Library Fundamentals V1 TS Components for C++17 (P0220R1)*. 2016. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0220r1.html (visited on 03/09/2023).

[54] James C. Dehnert and Alexander Stepanov. "Fundamentals of Generic Programming". In: *Generic Programming*. Ed. by Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–11. ISBN: 978-3-540-39953-7. DOI: 10.1007/3-540-39953-4_1.

[55] Akim Demaille, Alexandre Duret-Lutz, Sylvain Lombardy, and Jacques Sakarovitch. "Implementation Concepts in Vaucanson 2". In: *Proceedings of Implementation and Application of Automata, 18th International Conference (CIAA'13)*. Ed. by Stavros Konstantinidis. Vol. 7982. Lecture Notes in Computer Science. Halifax, NS, Canada: Springer, July 2013, pp. 122–133. ISBN: 978-3-642-39274-0. DOI: 10.1007/978-3-642-39274-0_12.

[56] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[57] Louis Dionne. *Dyno: Runtime polymorphism done right*. URL: https://github.com/ldionne/dyno (visited on 03/09/2023).

[58] Piotr Dollar and C. L. Zitnick. "Structured Forests for Fast Edge Detection". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2013.

[59] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[60] Marcos Cordeiro d'Ornellas. "Algorithmic Patterns for Morphological Image Processing". PhD thesis. Universiteit van Amsterdam, 2001.

[61] Marcos Cordeiro d'Ornellas and Rein Van Den Boomgaard. "The state of art and future development of morphological software towards generic algorithms". In: *International Journal of Pattern Recognition and Artificial Intelligence* 17.02 (2003), pp. 231–255. DOI: 10.1142/S0218001403002344.

[62] Marc Droske and Martin Rumpf. "Multiscale Joint Segmentation and Registration of Image Morphology". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29.12 (2007), pp. 2181–2194. DOI: 10.1109/TPAMI.2007.1120.

[63] Baptiste Esteban, Guillaume Tochon, and Thierry Géraud. "Estimating the Noise Level Function with the Tree of Shapes and Non-parametric Statistics". In: *Computer Analysis of Images and Patterns*. Ed. by Mario Vento and Gennaro Percannella. Cham: Springer International Publishing, 2019, pp. 377–388. ISBN: 978-3-030-29891-3. DOI: 10.1007/978-3-030-29891-3_33.

[64] Matthieu Faessel and Michel Bilodeau. "SMIL: Simple morphological image library". In: *Séminaire Performance et Généricité, LRDE* (2014).

[65] Alessandro Foi, Mejdi Trimeche, Vladimir Katkovnik, and Karen Egiazarian. "Practical Poissonian-Gaussian Noise Modeling and Fitting for Single-Image Raw-Data". In: *IEEE Transactions on Image Processing* 17.10 (2008), pp. 1737–1754. DOI: 10.1109/TIP.2008.2001399.

[66] *Folly: Facebook Open-source Library*. URL: https://github.com/facebook/folly (visited on 04/09/2023).

[67] Gabriel Barbosa da Fonseca et al. "New hierarchy-based segmentation layer: towards automatic marker proposal". In: *2021 34th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*. 2021, pp. 354–361. DOI: 10.1109/SIBGRAPI54419.2021.00055.

[68] Python Software Foundation. *CPython*. URL: https://github.com/python/cpython (visited on 06/07/2023).

[69] Martin Fowler. *Domain-Specific Languages*. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN: 978-0-321-71294-3.

[70] W.T. Freeman, T.R. Jones, and E.C. Pasztor. "Example-based super-resolution". In: *IEEE Computer Graphics and Applications* 22.2 (2002), pp. 56–65. DOI: 10.1109/38.988747.

[71] Thierry Géraud. "Outil logiciel pour le traitement d'images: Bibliothèque, paradigmes, types et algorithmes". In French. Habilitation Thesis. Université Paris-Est, June 2012.

[72] Thierry Géraud and Edwin Carlinet. *A Modern C++ Library for Generic and Efficient Image Processing*. Journée du Groupe de Travail sur la Géometrie Discrète et la Morphologie Mathématique. June 2018.

[73] Thierry Géraud, Edwin Carlinet, Sébastien Crozet, and Laurent Najman. "A quasi-linear algorithm to compute the tree of shapes of n-D images". In: *International symposium on mathematical morphology and its applications to signal and image processing*. Springer. 2013, pp. 98–110. DOI: 10.1007/978-3-642-38294-9_9.

[74] Thierry Géraud and Roland Levillain. "Semantics-Driven Genericity: A Sequel to the Static C++ Object-Oriented Programming Paradigm (SCOOP 2)". In: *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*. Paphos, Cyprus, July 2008.

[75] Thierry Géraud, Hugues Talbot, and Marc Van Droogenbroeck. "Algorithms for Mathematical Morphology". In: *Mathematical Morphology—From Theory to Applications*. Ed. by Laurent Najman and Hugues Talbot. Wiley-ISTE, July 2010, pp. 323–353. ISBN: 978-1-84821-215-2. DOI: https://doi.org/10.1002/9781118600788.ch12.

[76] Debasish Ghosh. *DSLs in Action*. 1st. USA: Manning Publications Co., 2010. ISBN: 9781935182450. DOI: 10.5555/1965333.

[77] Robert Glück, Ryo Nakashige, and Robert Zöchling. "Binding-time analysis applied to mathematical algorithms". In: *System Modelling and Optimization: Proceedings of the Seventeenth IFIP TC7 Conference on System Modelling and Optimization, 1995*. Ed. by Jaroslav Doležal and Jiří Fidler. Boston, MA: Springer US, 1996, pp. 137–146. DOI: 10.1007/978-0-387-34897-1_14.

[78] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[79] *Google Benchmark*. URL: https://github.com/google/benchmark (visited on 09/06/2023).

[80] Douglas Gregor et al. "Concepts: Linguistic support for generic programming in C++". In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2006, pp. 291–310. DOI: 10.1145/1167515.1167499.

[81] Michel Grimaud. "New measure of contrast: the dynamics". In: *Optics & Photonics*. 1992. URL: https://api.semanticscholar.org/CorpusID:64320621.

[82] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[83] Laurent Guigues, Jean Pierre Cocquerez, and Hervé Le Men. "Scale-Sets Image Analysis". In: *International Journal of Computer Vision* 68.3 (2006), pp. 289–317. ISSN: 1573-1405. DOI: 10.1007/s11263-005-6299-0.

[84] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11 –15.

[85] Charles R Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[86] Jiří Havel, François Merciol, and Sébastien Lefèvre. "Efficient schemes for computing α-tree representations". In: *Mathematical Morphology and Its Applications to Signal and Image Processing: 11th International Symposium, ISMM 2013, Uppsala, Sweden, May 27-29, 2013. Proceedings 11*. Springer. 2013, pp. 111–122. DOI: 10.1007/978-3-642-38294-9_10.

[87] Jiří Havel, François Merciol, and Sébastien Lefèvre. "Efficient tree construction for multiscale image representation and processing". In: *Journal of Real-Time Image Processing* 16.4 (2019), pp. 1129–1146. ISSN: 1861-8219. DOI: 10.1007/s11554-016-0604-0.

[88] G.E. Healey and R. Kondepudy. "Radiometric CCD camera calibration and noise estimation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16.3 (1994), pp. 267–276. DOI: 10.1109/34.276126.

[89] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. 3rd. Addison-Wesley Professional, 2008. ISBN: 0321562992. DOI: 10.5555/1502323.

[90] Romain Hermary et al. "Learning Grayscale Mathematical Morphology with Smooth Morphological Layers". In: *Journal of Mathematical Imaging and Vision* 64.7 (2022), pp. 736–753. ISSN: 1573-7683. DOI: 10.1007/s10851-022-01091-1.

[91] Wim H. Hesselink. "Salembier's Min-tree algorithm turned into breadth first search". In: *Information Processing Letters* 88.5 (2003), pp. 225–229. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2003.08.003.

[92] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[93] International Organization for Standardization / International Commission on Illumination. *Colorimetry - Part 4: CIE 1976 L*a*b* colour space*. ISO/CIE 11664-4:2019. 2019.

[94] Apple Inc. *The Swift Programming Language*. 2014. URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/ (visited on 06/06/2023).

[95] The MathWorks Inc. *MATLAB version: 9.13.0 (R2022b)*. Natick, Massachusetts, United States, 2022. URL: https://www.mathworks.com.

[96] ISO. *ISO/IEC 14882:2020: Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, 2020, p. 1853. URL: https://www.iso.org/standard/79358.html.

[97] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. 2017. URL: https://github.com/pybind/pybind11 (visited on 12/06/2023).

[98] Qiyu Jin, Gabriele Facciolo, and Jean-Michel Morel. "A Review of an Old Dilemma: Demosaicking First, or Denoising First?" In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2020, pp. 2169–2179. DOI: 10.1109/CVPRW50498.2020.00265.

[99] Neil D. Jones. "An Introduction to Partial Evaluation". In: *ACM Comput. Surv.* 28.3 (1996), 480–503. ISSN: 0360-0300. DOI: 10.1145/243439.243447.

[100] Ronald Jones. "Component trees for image filtering and segmentation". In: *IEEE Workshop on Nonlinear Signal and Image Processing*. Vol. 9. Mackinac Island. 1997.

[101] Ronald Jones. "Connected Filtering and Segmentation Using Component Trees". In: *Computer Vision and Image Understanding* 75.3 (1999), pp. 215–228. ISSN: 1077-3142. DOI: 10.1006/cviu.1999.0777.

[102] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. "Progressive Growing of GANs for Improved Quality, Stability, and Variation". In: *CoRR* abs/1710.10196 (2017). URL: http://arxiv.org/abs/1710.10196.

[103] M. G. Kendall. "A New Measure of Rank Correlation". In: *Biometrika* 30.1/2 (1938), p. 81. DOI: 10.2307/2332226.

[104] M. G. Kendall. "The Treatment Of Ties In Ranking Problems". In: *Biometrika* 33.3 (1945), pp. 239–251. DOI: 10.1093/biomet/33.3.239.

[105] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0-13-110370-9. DOI: 10.5555/576122.

[106] Efim Khalimsky, Ralph Kopperman, and Paul R Meyer. "Computer graphics and connected topologies on finite ordered sets". In: *Topology and its Applications* 36.1 (1990), pp. 1–17. DOI: 10.1016/0166-8641(90)90031-V.

[107] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284. DOI: 10.5555/3271463. URL: https://doc.rust-lang.org/book/.

[108] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0201896834.

[109] T Yung Kong and Azriel Rosenfeld. "Digital topology: Introduction and survey". In: *Computer Vision, Graphics, and Image Processing* 48.3 (1989), pp. 357–393. DOI: 10.1016/0734-189X(89)90147-3.

[110] Ullrich Köthe. "Reusable software in computer vision". In: *Handbook of computer vision and applications* 3 (1999), pp. 103–132.

[111] Ullrich Köthe and Karsten Weihe. "The STL model in the geometric domain". In: *Generic Programming*. Springer. 2000, pp. 232–248.

[112] Serge Koudoro, Matthieu Faessel, and Michel Bilodeau. "Morph-M: Image Processing Library Specialized in Mathematical Morphology". In: *IPOL*. Meeting on Image Processing Libraries (2012).

[113] Ssafak Güner Koç et al. "A comparative noise robustness study of tree representations for attribute profile construction". In: *2017 25th Signal Processing and Communications Applications Conference (SIU)*. 2017, pp. 1–4. DOI: 10.1109/SIU.2017.7960159.

[114] H. W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97. DOI: 10.1002/nav.3800020109.

[115] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. "Numba: A LLVM-Based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162.

[116] Raphael Lapertot, Giovanni Chierchia, and Benjamin Perret. "Supervised Learning of Hierarchical Image Segmentation". working paper or preprint. July 2023. URL: https://hal.science/hal-04205711.

[117] Jukka Lehtosalo. *Mypy: Static Typing for Python*. URL: https://mypy-lang.org/ (visited on 09/26/2023).

[118] Roland Levillain. "Towards a software architecture for generic image processing". PhD thesis. Université Paris-Est, 2011.

[119] Roland Levillain, Thierry Géraud, and Laurent Najman. “Milena: Write Generic Morphological Algorithms Once, Run on Many Kinds of Images”. In: *Mathematical Morphology and Its Application to Signal and Image Processing*. Ed. by Michael H. F. Wilkinson and Jos B. T. M. Roerdink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 295–306. ISBN: 978-3-642-03613-2. DOI: 10.1007/978-3-642-03613-2_27.

[120] Roland Levillain, Thierry Géraud, and Laurent Najman. “Writing Reusable Digital Topology Algorithms in a Generic Image Processing Framework”. In: *Applications of Discrete Geometry and Mathematical Morphology*. Ed. by Ullrich Köthe, Annick Montanvert, and Pierre Soille. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 140–153. ISBN: 978-3-642-32313-3. DOI: 10.1007/978-3-642-32313-3_10.

[121] Roland Levillain, Thierry Géraud, Laurent Najman, and Edwin Carlinet. “Practical Genericity: Writing Image Processing Algorithms Both Reusable and Efficient”. In: *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*. Vol. 8827. Lecture Notes in Computer Science. Bayro-Corrochano, Eduardo and Hancock, Edwin. Puerto Vallarta, Mexico: Springer, Nov. 2014, pp. 70 –79. DOI: 10.1007/978-3-319-12568-8_9. URL: https://hal.science/hal-01082349.

[122] Roland Levillain, Thierry Géraud, and Laurent Najman. “Why and howto design a generic and efficient image processing framework: The case of the Milena library”. In: *2010 IEEE International Conference on Image Processing*. 2010, pp. 1941–1944. DOI: 10.1109/ICIP.2010.5649620.

[123] O. Lezoray and L. Grady. *Image Processing and Analysis with Graphs: Theory and Practice*. Digital Imaging and Computer Vision. CRC Press, 2017. ISBN: 9781439855089. URL: https://books.google.fr/books?id=NoPRBQAAQBAJ.

[124] Olivier Lézoray. “Complete lattice learning for multivariate mathematical morphology”. In: *Journal of Visual Communication and Image Representation* 35 (2016), pp. 220–235. DOI: 10.1016/j.jvcir.2015.12.017.

[125] Olivier Lezoray, Christophe Charrier, and Abderrahim Elmoataz. “Rank transformation and manifold learning for multivariate mathematical morphology”. In: *2009 17th European Signal Processing Conference*. 2009, pp. 35–39.

[126] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. “Abstraction Mechanisms in CLU”. In: *Commun. ACM* 20.8 (1977), 564–576. ISSN: 0001-0782. DOI: 10.1145/359763.359789.

[127] Ce Liu, W.T. Freeman, R. Szeliski, and Sing Bing Kang. "Noise Estimation from a Single Image". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 1. 2006, pp. 901–908. DOI: 10.1109/CVPR.2006.207.

[128] Xinhao Liu, Masayuki Tanaka, and Masatoshi Okutomi. "Single-Image Noise Level Estimation for Blind Denoising". In: *IEEE Transactions on Image Processing* 22.12 (2013), pp. 5226–5237. DOI: 10.1109/TIP.2013.2283400.

[129] Le Duy Lê Duy Huỳnh, Yongchao Xu, and Thierry Géraud. "Morphology-based hierarchical representation with application to text segmentation in natural images". In: *2016 23rd International Conference on Pattern Recognition (ICPR)*. 2016, pp. 4029–4034. DOI: 10.1109/ICPR.2016.7900264.

[130] David MacQueen. "Modules for Standard ML". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: Association for Computing Machinery, 1984, 198–207. ISBN: 0897911423. DOI: 10.1145/800055.802036. URL: https://doi.org/10.1145/800055.802036.

[131] Simon Marlow et al. "Haskell 2010 language report". In: (2010).

[132] Jonathan Masci, Jesús Angulo, and Jürgen Schmidhuber. "A Learning Framework for Morphological Operators Using Counter–Harmonic Mean". In: *Mathematical Morphology and Its Applications to Signal and Image Processing*. Ed. by Cris L. Luengo Hendriks, Gunilla Borgefors, and Robin Strand. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 329–340. ISBN: 978-3-642-38294-9. DOI: 10.1007/978-3-642-38294-9_28.

[133] J Matas, O Chum, M Urban, and T Pajdla. "Robust wide-baseline stereo from maximally stable extremal regions". In: *Image and Vision Computing* 22.10 (2004). British Machine Vision Computing 2002, pp. 761–767. ISSN: 0262-8856. DOI: 10.1016/j.imavis.2004.02.006.

[134] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

[135] Arnold Meijster and Jos B. T. M. Roerdink. "A disjoint set algorithm for the watershed transform". In: *9th European Signal Processing Conference (EUSIPCO 1998)*. 1998, pp. 1–4.

[136] Fernand Meyer. "Stochastic watershed hierarchies". In: *2015 Eighth International Conference on Advances in Pattern Recognition (ICAPR)*. 2015, pp. 1–8. DOI: 10.1109/ICAPR.2015.7050646.

[137] Fernand Meyer. “Un algorithme optimal de ligne de partage des eaux”. In: *Actes du 8e congrès reconnaissance des formes et intelligence artificielle* 2 (1991), pp. 847–859.

[138] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[139] Robin Milner and Mads Tofte. *Commentary on Standard ML*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0262631377. DOI: 10.5555/103021.

[140] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. " O’Reilly Media, Inc.", 2013.

[141] Pascal Monasse and Frederic Guichard. “Fast computation of a contrast-invariant image representation”. In: *IEEE transactions on image processing* 9.5 (2000), pp. 860–872. DOI: 10.1109/83.841532.

[142] David Mumford and Jayant Shah. “Optimal approximations by piecewise smooth functions and associated variational problems”. In: *Communications on Pure and Applied Mathematics* 42.5 (1989), pp. 577–685. DOI: 10.1002/cpa.3160420503.

[143] David Musser, Sibylle Schupp, and Rüdiger Loos. “Requirement Oriented Programming”. In: *Generic Programming*. Ed. by Mehdi Jazayeri, Rüdiger G. K. Loos, and David R. Musser. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 12–24. ISBN: 978-3-540-39953-7. DOI: 10.1007/3-540-39953-4_2.

[144] David R Musser, Gilmer J Derge, and Atul Saini. *STL tutorial and reference guide: C++ programming with the standard template library*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[145] David R. Musser and Alexander A. Stepanov. “A Library of Generic Algorithms in Ada”. In: *Proceedings of the 1987 Annual ACM SIGAda International Conference on Ada*. SIGAda ’87. Boston, Massachusetts, USA: Association for Computing Machinery, 1987, 216–225. ISBN: 0897912438. DOI: 10.1145/317500.317529.

[146] David R Musser and Alexander A Stepanov. “Generic programming”. In: *International Symposium on Symbolic and Algebraic Computation (ISSAC)*. Springer, 1988, pp. 13–25. DOI: 10.1007/3-540-51084-2_2.

[147] Markku Mäkitalo and Alessandro Foi. “Noise Parameter Mismatch in Variance Stabilization, With an Application to Poisson–Gaussian Noise Estimation”. In: *IEEE Transactions on Image Processing* 23.12 (2014), pp. 5348–5359. DOI: 10.1109/TIP.2014.2363735.

[148] Makoto Nagao, Takashi Matsuyama, and Yoshio Ikeda. “Region extraction and shape analysis in aerial photographs”. In: *Computer Graphics and Image Processing* 10.3 (1979), pp. 195–223. ISSN: 0146-664X. DOI: 10.1016/0146-664X(79)90001-7.

[149] L. Najman and M. Schmitt. “Geodesic saliency of watershed contours and hierarchical segmentation”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18.12 (1996), pp. 1163–1173. DOI: 10.1109/34.546254.

[150] Laurent Najman and Michel Couprie. “Building the component tree in quasi-linear time”. In: *IEEE Transactions on image processing* 15.11 (2006), pp. 3531–3539. DOI: 10.1109/TIP.2006.877518.

[151] Laurent Najman and Jean Cousty. “A graph-based mathematical morphology reader”. In: *Pattern Recognition Letters* 47 (2014), pp. 3–17.

[152] Laurent Najman, Jean Cousty, and Benjamin Perret. “Playing with kruskal: algorithms for morphological trees in edge-weighted graphs”. In: *Mathematical Morphology and Its Applications to Signal and Image Processing: 11th International Symposium, ISMM 2013, Uppsala, Sweden, May 27-29, 2013. Proceedings 11*. Springer. 2013, pp. 135–146. DOI: 10.1007/978-3-642-38294-9_12.

[153] Laurent Najman and Hugues Talbot. *Mathematical Morphology: from theory to applications*. Ed. by Najman Laurent and Talbot Hugues. ISTE-Wiley, June 2010, p. 507. DOI: 10.1002/9781118600788.

[154] Axel Naumann. *Variant: a type-safe union for C++17 (PR0088R3)*. 2016. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0088r3.html (visited on 03/09/2023).

[155] Eric Niebler. *Boost Accumulators*. 2005. URL: https://www.boost.org/doc/libs/1_82_0/doc/html/accumulators.html (visited on 11/06/2023).

[156] David Nistér and Henrik Stewénius. “Linear Time Maximally Stable Extremal Regions”. In: *Computer Vision – ECCV 2008*. Ed. by David Forsyth, Philip Torr, and Andrew Zisserman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 183–196. ISBN: 978-3-540-88688-4. DOI: 10.1007/978-3-540-88688-4_14.

[157] Georgios K. Ouzounis and Pierre Soille. “Pattern Spectra from Partition Pyramids and Hierarchies”. In: *Mathematical Morphology and Its Applications to Image and Signal Processing*. Ed. by Pierre Soille, Martino Pesaresi, and Georgios K. Ouzounis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 108–119. ISBN: 978-3-642-21569-8. DOI: 10.1007/978-3-642-21569-8_10.

[158] Georgios K Ouzounis and Pierre Soille. *The alpha-tree algorithm*. Tech. rep. 2012.

[159] Georgios K. Ouzounis and Michael H.F. Wilkinson. “Partition-induced connections and operators for pattern analysis”. In: *Pattern Recognition* 43.10 (2010), pp. 3193–3207. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2009.10.002.

[160] Adam Paszke et al. “Automatic differentiation in pytorch”. In: (2017).

[161] Fernando Perez and Brian E. Granger. “IPython: A System for Interactive Scientific Computing”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 21–29. DOI: 10.1109/MCSE.2007.53.

[162] Benjamin Perret and Jean Cousty. “Component Tree Loss Function: Definition and Optimization”. In: *Discrete Geometry and Mathematical Morphology*. Ed. by Étienne Baudrier, Benoît Naegel, Adrien Krähenbühl, and Mohamed Tajine. Cham: Springer International Publishing, 2022, pp. 248–260. ISBN: 978-3-031-19897-7. DOI: 10.1007/978-3-031-19897-7_20.

[163] Benjamin Perret, Jean Cousty, Silvio Jamil F. Guimarães, and Deise S. Maia. “Evaluation of Hierarchical Watersheds”. In: *IEEE Transactions on Image Processing* 27.4 (2018), pp. 1676–1688. DOI: 10.1109/TIP.2017.2779604.

[164] Benjamin Perret et al. “Higra: Hierarchical graph analysis”. In: *SoftwareX* 10 (2019), p. 100335.

[165] Benjamin Perret et al. “Removing non-significant regions in hierarchical clustering and segmentation”. In: *Pattern Recognition Letters* 128 (2019), pp. 433–439. DOI: 10.1016/j.patrec.2019.10.008.

[166] Sylvie Philipp-Foliguet, Michel Jordan, Laurent Najman, and Jean Cousty. “Artwork 3D model database indexing and classification”. In: *Pattern Recognition* 44.3 (2011), pp. 588–597. DOI: 10.1016/j.patcog.2010.09.016.

[167] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091. DOI: 10.5555/509043.

[168] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. “A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 507–517. DOI: 10.1109/MSR.2019.00077.

[169] Fons van der Plas and Mikołaj Bochenski. *Pluto.jl*. URL: https://plutojl.org/ (visited on 05/07/2023).

[170] Jordi Pont-Tuset and Ferran Marques. “Measures and Meta-Measures for the Supervised Evaluation of Image Segmentation”. In: *2013 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2013. DOI: 10.1109/cvpr.2013.277.

[171] Jordi Pont-Tuset and Ferran Marques. “Supervised Assessment of Segmentation Hierarchies”. In: *Computer Vision – ECCV 2012*. Springer Berlin Heidelberg, 2012, pp. 814–827. DOI: 10.1007/978-3-642-33765-9_58.

[172] Jordi Pont-Tuset and Ferran Marques. “Upper-bound assessment of the spatial accuracy of hierarchical region-based image representations”. In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2012. DOI: 10.1109/icassp.2012.6288021.

[173] Jonathan Ragan-Kelley et al. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Seattle, Washington, USA: Association for Computing Machinery, 2013, 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176.

[174] V Robert et al. “The NAROO digitization center-Overview and scientific program”. In: *Astronomy & Astrophysics* 652 (2021), A3. DOI: 10.1051/0004-6361/202140472.

[175] Christian Ronse. “Partial Partitions, Partial Connections and Connective Segmentation”. In: *Journal of Mathematical Imaging and Vision* 32.2 (2008), pp. 97–125. ISSN: 1573-7683. DOI: 10.1007/s10851-008-0090-5.

[176] Christian Ronse. “Why mathematical morphology needs complete lattices”. In: *Signal processing* 21.2 (1990), pp. 129–154.

[177] Guido van Rossum, Jukka Lehtosalo, and Lukasz Langa. *PEP 484 – Type Hints*. URL: https://peps.python.org/pep-0484/.

[178] Michaël Roynard. “Generic programming in modern C++ for Image Processing”. PhD thesis. Sorbonne université, 2022.

[179] Michaël Roynard, Edwin Carlinet, and Thierry Géraud. “A Modern C++ Point of View of Programming in Image Processing”. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. 2022, pp. 164–171.

[180] Michaël Roynard, Edwin Carlinet, and Thierry Géraud. "An Image Processing Library in Modern C++: Getting Simplicity and Efficiency with Generic Programming". In: *Reproducible Research in Pattern Recognition*. Ed. by Bertrand Kerautret et al. Cham: Springer International Publishing, 2018, pp. 121–137. ISBN: 978-3-030-23987-9.

[181] Szymon Rusinkiewicz. "Estimating curvatures and their derivatives on triangle meshes". In: *2nd International Symposium on 3D Data Processing, Visualization and Transmission (3DPVT)*. IEEE. 2004, pp. 486–493. DOI: 10.1109/TDPVT.2004.1335277.

[182] P. Salembier and L. Garrido. "Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval". In: *IEEE Transactions on Image Processing* 9.4 (2000), pp. 561–576. DOI: 10.1109/83.841934.

[183] P. Salembier, A. Oliveras, and L. Garrido. "Antiextensive connected operators for image and sequence processing". In: *IEEE Transactions on Image Processing* 7.4 (1998), pp. 555–570. DOI: 10.1109/83.663500.

[184] Deise Santana Maia, Jean Cousty, Laurent Najman, and Benjamin Perret. "Properties of combinations of hierarchical watersheds". In: *Pattern Recognition Letters* 128 (2019), pp. 513–520. ISSN: 0167-8655. DOI: 10.1016/j.patrec.2019.10.009.

[185] Deise Santana Maia, Jean Cousty, Laurent Najman, and Benjamin Perret. "Watersheding Hierarchies". In: *Mathematical Morphology and Its Applications to Signal and Image Processing*. Ed. by Bernhard Burgeth et al. Cham: Springer International Publishing, 2019, pp. 124–136. ISBN: 978-3-030-20867-7. DOI: 10.1007/978-3-030-20867-7_10.

[186] Deise Santana Maia et al. "Evaluation of Combinations of Watershed Hierarchies". In: *Mathematical Morphology and Its Applications to Signal and Image Processing*. Ed. by Jesús Angulo, Santiago Velasco-Forero, and Fernand Meyer. Cham: Springer International Publishing, 2017, pp. 133–145. ISBN: 978-3-319-57240-6. DOI: 10.1007/978-3-319-57240-6_11.

[187] Jean Serra. "A lattice approach to image segmentation". In: *Journal of Mathematical Imaging and Vision* 24 (2006), pp. 83–130. DOI: 10.1007/s10851-005-3616-0.

[188] Jean Serra. *Image Analysis and Mathematical Morphology*. Image Analysis and Mathematical Morphology. Academic Press, 1982. ISBN: 9780126372410.

[189] Jean Serra. “Morphological filtering: An overview”. In: *Signal Processing* 38.1 (1994). Mathematical Morphology and its Applications to Signal Processing, pp. 3–11. ISSN: 0165-1684. DOI: 10.1016/0165-1684(94)90052-3.

[190] Jeremy Siek and Andrew Lumsdaine. *Boost Concept Checking Library (BCCL)*. 2000. URL: https://www.boost.org/libs/concept_check/concept_check.htm (visited on 02/06/2023).

[191] Jeremy Siek and Andrew Lumsdaine. “Concept checking: Binding parametric polymorphism in C++”. In: *First Workshop on C++ Template Programming*. 2000, p. 71.

[192] Jeremy Siek and Walid Taha. “Gradual typing for objects”. In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 2–27. DOI: 10.1007/978-3-540-73589-2_2.

[193] Jeremy G. Siek and Walid Taha. “Gradual Typing for Functional Languages”. In: *Scheme and Functional Programming Workshop*. 2006, pp. 81–92.

[194] J.G. Siek, L.Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual, The*. C++ In-Depth Series. Pearson Education, 2001. ISBN: 9780321601612.

[195] Pierre Soille. “Constrained connectivity for hierarchical image partitioning and simplification”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.7 (2008), pp. 1132–1145. DOI: 10.1109/TPAMI.2007.70817.

[196] Pierre Soille. *Morphological Image Analysis*. Springer Berlin Heidelberg, 1999. DOI: 10.1007/978-3-662-03939-7. URL: https://doi.org/10.1007%2F978-3-662-03939-7.

[197] *Spyder: The Scientific Python Development Environment*. URL: https://www.spyder-ide.org/ (visited on 05/07/2023).

[198] Alexander Stepanov and Meng Lee. *The standard template library*. Vol. 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995.

[199] Alexander Stepanov and Paul McJones. *Elements of Programming*. 1st. Addison-Wesley Professional, 2009. ISBN: 032163537X. DOI: 10.5555/1614221.

[200] Alexander A Stepanov, Aaron Kershenbaum, and David R Musser. *Higher order programming*. 1987.

[201] Michael Still. *The definitive guide to ImageMagick*. Apress, 2006.

[202] Christopher Strachey. "Fundamental Concepts in Programming Languages". In: *Higher-Order and Symbolic Computation* 13.1 (2000), pp. 11–49. ISSN: 1573-0557. DOI: 10.1023/A:1010000313106.

[203] Bjarne Stroustrup. *The C++ programming language*. en. 4th ed. Boston, MA: Addison-Wesley Educational, Jan. 2013.

[204] Camille Sutour, Charles-Alban Deledalle, and Jean-François Aujol. "Estimation of the Noise Level Function Based on a Nonparametric Detection of Homogeneous Image Regions". In: *SIAM Journal on Imaging Sciences* 8.4 (2015), pp. 2622–2661. DOI: 10.1137/15M1012682.

[205] Ameet Talwalkar, Sanjiv Kumar, Mehryar Mohri, and Henry Rowley. "Large-scale SVD and Manifold Learning". In: *Journal of Machine Learning Research* 14.96 (2013), pp. 3129–3152.

[206] Hanlin Tan et al. "Pixelwise Estimation of Signal-Dependent Image Noise Using Deep Residual Learning". In: *Computational Intelligence and Neuroscience* 2019 (2019), p. 4970508. ISSN: 1687-5265. DOI: 10.1155/2019/4970508.

[207] Robert Endre Tarjan. "Efficiency of a Good But Not Linear Set Union Algorithm". In: 22.2 (1975), pp. 215–225. ISSN: 0004-5411. DOI: 10.1145/321879.321884.

[208] EPITA Research Laboratory Image Team. *Pylene: A Generic and Efficient Image Processing Library*. URL: https://gitlab.lre.epita.fr/olena/pylene (visited on 04/07/2023).

[209] GIMP Development Team. *GNU Image Manipulation Program (GIMP)*. URL: https://www.gimp.org (visited on 05/07/2023).

[210] The PyPy Team. *PyPy: A fast, compliant alternative implementation of Python*. URL: https://www.pypy.org (visited on 06/07/2023).

[217] ANSI. *American National Standard: Programming Language – Common Lisp*. ANSI X3.226:1994 (R1999). 1994.

[218] Mads Tofte. "Essentials of Standard ML Modules". In: *Advanced Functional Programming*. Ed. by John Launchbury, Erik Meijer, and Tim Sheard. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 208–229. DOI: 10.1007/3-540-61628-4_8.

[219] *Trait `std::ops::Add`*. URL: https://doc.rust-lang.org/std/ops/trait.Add.html (visited on 08/06/2023).

[220] Alain Trémeau and Philippe Colantoni. "Regions adjacency graph applied to color image segmentation". In: *IEEE Transactions on image processing* 9.4 (2000), pp. 735–744. DOI: 10.1109/83.841950.

[221] Florence Tushabe and MHF Wilkinson. “Image preprocessing for compression: Attribute filtering”. In: *Proceedings of the World Congress on Engineering and Computer Science (WCECS)*. 2007.

[222] Erik R. Urbach. “Intelligent Object Detection Using Trees”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 289–300. DOI: 10.1007/978-3-319-18720-4_25.

[223] Erik R Urbach and Michael HF Wilkinson. “Shape-only granulometries and grey-scale shape filters”. In: *Proceedings of the International Symposum on Mathematical Morphology (ISMM)*. Vol. 2002. 2002, pp. 305–314.

[224] Corinne Vachier and Fernand Meyer. “Extinction value: a new measurement of persistence”. In: *IEEE Workshop on nonlinear signal and image processing*. Vol. 1. Neos Marmaras Greece. 1995, pp. 254–257.

[225] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*. Pearson Education, 2002. ISBN: 9780672334054.

[226] Luc Vincent. “Grayscale area openings and closings, their efficient implementation and applications”. In: *First Workshop on Mathematical Morphology and its Applications to Signal Processing*. 1993, pp. 22–27.

[227] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature methods* 17.3 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[228] Andrew J. Viterbi and James K. Omura. *Principles of Digital Communication and Coding*. 1st. USA: McGraw-Hill, Inc., 1979. ISBN: 0070675163. DOI: 10.5555/578474.

[229] James S. Walker. “Combined image compressor and denoiser based on tree-adapted wavelet shrinkage”. In: *Optical Engineering* 41.7 (2002), pp. 1520 –1527. DOI: 10.1117/1.1483086.

[230] Stéfan van der Walt et al. “scikit-image: image processing in Python”. In: *PeerJ* 2 (June 2014), e453. ISSN: 2167-8359. DOI: 10.7717/peerj.453. URL: https://doi.org/10.7717/peerj.453.

[231] Steven Watanabe. *Boost.TypeErasure*. URL: https://www.boost.org/doc/libs/1_83_0/doc/html/boost_typeerasure.html (visited on 03/09/2023).

[232] A.I. Watson. “A new method of classification for Landsat data using the ‘watershed’ algorithm”. In: *Pattern Recognition Letters* 6.1 (1987), pp. 15–19. DOI: 10.1016/0167-8655(87)90044-4.

[233] Michael H. F. Wilkinson and Jos B. T. M. Roerdink. "Fast Morphological Attribute Operations Using Tarjan's Union-Find Algorithm". In: *Mathematical Morphology and its Applications to Image and Signal Processing*. Ed. by John Goutsias, Luc Vincent, and Dan S. Bloomberg. Boston, MA: Springer US, 2000, pp. 311–320. ISBN: 978-0-306-47025-7. DOI: 10.1007/0-306-47025-X_34.

[234] Michael H.F. Wilkinson. "A fast component-tree algorithm for high dynamic-range images and second generation connectivity". In: *2011 18th IEEE International Conference on Image Processing*. IEEE, 2011. DOI: 10.1109/icip.2011.6115597.

[235] Niklaus Wirth. "Modula: A language for modular multiprogramming". In: *Software: Practice and Experience* 7.1 (1977), pp. 1–35.

[236] Shaoping Xu et al. "A fast yet reliable noise level estimation algorithm using shallow CNN-based noise separator and BP network". In: *Signal, Image and Video Processing* 14.4 (2020), pp. 763–770. ISSN: 1863-1711. DOI: 10.1007/s11760-019-01608-z.

[237] Yongchao Xu, Thierry Géraud, and Laurent Najman. "Connected Filtering on Tree-Based Shape-Spaces". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.6 (2016), pp. 1126–1140. DOI: 10.1109/TPAMI.2015.2441070.

[238] Yongchao Xu, Thierry Géraud, and Laurent Najman. "Context-based energy estimator: Application to object segmentation on the tree of shapes". In: *2012 19th IEEE International Conference on Image Processing*. 2012, pp. 1577–1580. DOI: 10.1109/ICIP.2012.6467175.

[239] Yongchao Xu, Thierry Géraud, and Laurent Najman. "Morphological filtering in shape spaces: Applications using tree-based image representations". In: *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*. 2012, pp. 485–488.

[240] Yongchao Xu, Thierry Géraud, and Laurent Najman. "Salient level lines selection using the Mumford-Shah functional". In: *2013 IEEE International Conference on Image Processing*. 2013, pp. 1227–1231. DOI: 10.1109/ICIP.2013.6738253.

[241] Yongchao Xu, Pascal Monasse, Thierry Géraud, and Laurent Najman. "Tree-Based Morse Regions: A Topological Approach to Local Feature Detection". In: *IEEE Transactions on Image Processing* 23.12 (2014), pp. 5612–5625. DOI: 10.1109/TIP.2014.2364127.

[242] Jiwoo You, Scott C Trager, and Michael HF Wilkinson. "A fast, memory-efficient alpha-tree algorithm using flooding and tree size estimation". In: *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*. Springer. 2019, pp. 256–267. DOI: 10.1007/978-3-030-20867-7_20.

[243] Zongsheng Yue et al. "Variational Denoising Network: Toward Blind Noise Modeling and Removal". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019.

[244] Kai Zhang, Wangmeng Zuo, and Lei Zhang. "FFDNet: Toward a Fast and Flexible Solution for CNN-Based Image Denoising". In: *IEEE Transactions on Image Processing* 27.9 (2018), pp. 4608–4622. DOI: 10.1109/TIP.2018.2839891.