# Algorithms and Data Structures
for Component-Hypertrees of Gray-Level Images

Alexandre Morimitsu

Text presented
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
as a
requirement
to
obtain the title
of
PhD in Sciences

Program: Computer Science
Advisor: Prof. Ronaldo Fumio Hashimoto

During the development of this study the author received financial support from CAPES

São Paulo, March 2021

Algorithms and Data Structures for Component-Hypertrees of Gray-Level Images

This version of the thesis contains corrections and modifications suggested by the Judging Committee during the defense of the original version of this manuscript, performed on January 22nd, 2021. A copy of the original version is available in the Institute of Mathematics and Statistics of the University of São Paulo.

Jugding Committee:

- Prof. Dr. Ronaldo Fumio Hashimoto - IME-USP
- Prof. Dr. Thierry Géraud - EPITA
- Prof. Dr. Nicolas Passat - URCA
- Prof. Dr. Silvio Jamil Ferzoli Guimarães - PUC-MG
- Prof. Dr. Paulo André Vechiatto de Miranda - IME-USP

Resumo

MORIMITSU, A. **Algoritmos e Estruturas de Dados para Hiperárvores de Componentes de Imagens em Níveis de Cinza**. 2021. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2021.

Esta tese tem como foco o estudo de hiperárvores de componentes, que consistem em grafos utilizados para armazenar imagens em níveis de cinza de forma hierárquica. Neste grafos, nós representam componentes conexos de uma imagem extraídos a partir de um conjunto de vizinhanças crescentes, enquanto arcos são utilizados para organizar os nós de acordo com relações de inclusão. Neste texto, o objetivo principal consiste na elaboração de algoritmos e estruturas de dados eficientes para a construção, o armazenamento e a manipulação de hiperárvores de componentes. Mais especificamente, as principais contribuições podem ser resumidas nos seguintes itens: (i) a teoria por trás de hiperárvores de componentes é revisada e expandida, e as propriedades mais importantes são destacadas e provadas. Estas propriedades são então usadas no desenvolvimento de algoritmos e estruturas de dados otimizados, que reduzem consideravelmente o consumo de tempo e memória comparados com abordagens anteriores; (ii) o impacto da escolha das vizinhanças é analisada e uma nova família de vizinhanças baseadas em hierarquia de partições é proposta, resultando em algoritmos ainda mais rápidos; (iii) uma forma eficiente de computar variações de atributos é fornecida, possibilitando a elaboração de aplicações que focam na extração de objetos compostos de um conjunto de objetos menores; (iv) uma análise experimental é realizada, mostrando que a estratégia proposta é mais rápida e eficiente do que outras abordagens e (v) uma abordagem para segmentação de palavras é desenvolvida, mostrando um exemplo de aplicação onde variação de atributos pode ser particularmente útil.

Palavras-chave: Morfologia Matemática, operadores conexos, componentes conexos, árvores de componentes, hiperárvores de componentes, conectividade baseada em dilatação, hierarquia de partições.

Abstract

MORIMITSU, A. **Algorithms and Data Structure for Component-Hypertrees of Gray-Level Images**. 2021. Thesis (Ph.D.) - Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2021.

This thesis focuses on the study of component-hypertrees, which are graphs that store gray-level images in a hierarchical way. In such graphs, nodes represent connected components of an image extracted from multiple increasing connectivities, while arcs are used to organize these nodes according to an inclusion relation. In this research, the main goal is to develop efficient algorithms and data structures for component-hypertree construction, storage and manipulation. More specifically, our main contributions can be summarized as follows: (i) the theory behind component-hypertrees is reviewed and expanded, with some important properties being highlighted and proved. Using these properties, optimized algorithms and data structures are developed, resulting in implementations that considerably decrease time consumption and memory usage when compared to previously existing strategies; (ii) the impact of the choice of connectivities used to extract connected component is studied and a new family of neighborhoods based on a hierarchy of partitions is proposed, leading to the development of even faster algorithms; (iii) an efficient way of computing attribute variation is explained, allowing the development of applications that extract nodes comprised of clusters of smaller objects; (iv) an experimental analysis is conducted, to show that the proposed strategy is faster and more efficient than previously existing approaches and (v) a word segmentation tool is developed, to showcase an example of an application where attribute variation is particularly suitable.

Keywords: Mathematical Morphology, connected operators, connected components, component-trees, component-hypertrees, dilation-based connectivity, hierarchy of partitions.

Contents

List of Abbreviations

CC	Connected component
DAG	Directed Acyclic Graph
pixel	Picture element
SE	Structuring element

List of Symbols

α	Threshold (gray-level)
$\mathcal{A}$	Neighborhood relation
$\mathbb{A}$	Sequence of neighborhood relations
β	Threshold (gray-level)
$\mathcal{B}$	Structuring element of a generating sequence
C	Connected component
d	Dimension of an image
D_f	Domain of a gray-level image f
$\mathcal{E}$	Set of arcs of a graph
$\mathbb{E}$	Sequence of sets of arcs
f	Gray-scale image
g	Generic function
$\mathcal{G}$	Graph
η	Number of calls of parentUpdate
$\mathcal{H}$	Partition
$\mathbb{H}$	Sequence of partitions
$\mathfrak{H}$	Function that returns the element with highest depth in L
θ	Composition of downsamplings
Θ	Computation complexity
i	Neighborhood index
K	Number of gray-levels
κ	Attribute
$\mathbb{K}$	Set of gray-levels
L	Set of pixels used in dilation-generated neighborhoods
λ	Threshold (gray-level)
ℓ	Generic index
M	Node
n	Size of a sequence $\mathbb{A}$
N	Node
ν	Enumeration of a set
$\mathbb{N}$	Set of natural numbers

o	Origin of a structuring element
$\mathcal{O}$	Computational complexity (worst case)
π	Path in a graph
p	Pixel
P	Set of pixels
$\mathcal{P}$	Power set
q	Pixel
r	Representative/canonical element
R	Region
ρ	Downsampling
$\mathcal{R}$	Binary relation
$\mathbb{R}$	Set of real numbers
s	Element of a set
S	Set
$\mathcal{S}$	Structuring element
t	Scale of downsampling
v	Vertex of a graph
V	Set of vertices of a graph
ω	Weight (graph)
$\mathcal{W}$	Dilation of elements of a generating sequence
x	Coordinate
X	Binary image
y	Coordinate
$\mathbb{Z}$	Set of integer numbers
$\perp$	Parent of root nodes in arrays representing trees
$\emptyset$	Empty set

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In the field of Computer Vision and Image Processing, there are multiple ways of storing and manipulating digital images. Many of them rely on processing pixels of the image individually or in small windows, but these strategies have the downside of not taking into consideration the content of the image.

To avoid the drawback of these approaches, in the last two decades, connected operators have received increasing attention in the Mathematical Morphology community. One of the main contributing factors was the introduction of the component-tree [SOG98, Jon99], a structure that organizes connected components (CCs) of the level sets of an image hierarchically according to their inclusion relation.

Since the introduction of component-trees, numerous related works were published and, in particular, Passat and Naegel [PN11] proposed in 2011 a structure they called the component-hypertree. It consists of a graph that combines CCs from component-trees built using increasing connectivities, generating a structure that not only takes into consideration inclusion relation between CCs obtained from the same component-tree, but also relates CCs extracted using different connectivities. This property is useful in applications where objects of interest consist not only of individual CCs, but also of clusters of smaller objects, since both of them can be treated as nodes of component-hypertees. For instance, in text extraction applications, letters, words, and lines of text all become nodes of component-hypertrees, and computation of attributes in these nodes is sufficient to categorize them into each of these classes of objects.

Hence, component-hypertree is a rich representation that keeps information about both the content and the connectivity of the pixels of an image. However, contrary to component-trees, it has not been widely studied. For this reason, our main goal is to expand the theory about component-hypertrees. In particular, in this thesis, we mainly focus on the following issues related to construction of component-hypertrees:

- Construction of component-hypertrees can be time-consuming, since it needs to extract connected components of the input image according to multiple connectivities. Hence, we investigate algorithms used for component-tree construction and analyze how can they be generalized to build component-hypertrees efficiently.

- Although many connected components need to be stored in a component-hypertree, many of them are repeated. Therefore, we investigate properties of these connected components and investigate how we can design an optimized data structure to store component-hypertrees without loss of information.

- Finally, we investigate the advantages of using component-hypertrees instead of component-trees. For that, we present some attributes that analyzes how connected components merge as the connectivity increases that can only be obtained in component-hypertrees.

In the final sections of this manuscript, we show some theoretical and experimental results to show the efficiency of the proposed approaches to tackle the issues above mentioned.

1.2 Related Works

The main focus of this text is about theory and algorithms for component-trees and component-hypertrees, but it also includes themes such as shape analysis using attribute computation and different types of connectivities. Thus, in this section, we briefly recall some publications related to these topics, since they are the foundations of our study.

Historically, acquisition, analysis and characterization of shapes have been studied for a long time. One of the earliest related work consisted of the study proposed by Gray [Gra71], which focused on studying local properties of discrete binary images. These properties can be used to obtain efficient ways of computing metrics like area and perimeter of binary shapes. Additionally, Gray's paper also empathizes the importance of the connectivity used, showing some topological problems that arise with the choice of the connectivity.

As the theory evolved, analysis of shapes in more complex scenarios, such as gray-scale images or color images, were performed. In particular, this led to the development of connected filtering and, in the late 1990s, to the concept of component trees [SOG98, Jon99], in which shapes (CCs) of a grayscale image are hierarchically represented in the form of a tree.

Component trees introduced an efficient way of computing metrics in CCs and performing connected filtering. Thus, with the introduction of component trees, numerous related works were proposed. Among these works, some problems that were dealt with include, but are not limited to:

1. Strategies to build the component tree;

2. Efficient computation of attributes;

3. Connectivity used to obtain the CCs;

4. Other types of trees used to store an image hierarchically.

Below, we review some of the most relevant works related to these topics.

1.2.1 Strategies to Build the Component Tree

Different strategies to build the component-tree were proposed. Salembier et al. [SOG98] originally proposed the usage of a flooding strategy. In their approach, a component-tree is built from its root, and new nodes are created as we flood the image.

An alternative strategy was proposed by other authors later by using a merge strategy instead [NC06, BGL$^+$]. Under this approach, pixels are ordered according to their gray-levels before construction of the tree begins. Then, the pixels are processed following this order and merged together into CCs according to a union-find [Tar75] based algorithm.

A different approach was used in order to allow parallel construction of component-trees. This requires an algorithm to merge two disjoint component-trees and was first presented by Wilkinson et al. [WGH$^+$08]. An alternative version of this algorithm using merges of trees 1-D images was proposed by Matas et al. [MDA$^+$08].

A detailed comparison of the performance of these various component tree algorithms can be found in the paper published by Carlinet and Géraud [CG14].

1.2.2 Efficient Computation of Attributes

One of the biggest advantages of using component-trees is that it allows fast computation of some attributes, thanks to the inclusion order the nodes satisfy. While some attributes are straightforward to compute, there are some incremental attributes that require more complex strategies.

For instance, Passat et al. [PNR$^+$11] used the component-trees to compute, for each node, a pseudo-distance that analyzed the similarity of each node to a given binary shape in order to perform segmentation. Neumann and Matas [NM12] were able to compute attributes like perimeter (using 4-connectivity) and horizontal and vertical crossings. Climent and Oliveira [CO15] developed, based on Gray's metrics, patterns to compute the number of holes in the nodes of the tree. Silva et al. [SAMH16] later extended this idea to compute Gray's bit-quads in component-trees, allowing efficient computation of all attributes proposed by Gray.

In general, any attribute that can be efficiently computed in a component-tree can also be efficiently computed in component-hypertrees. For instance, in the original paper that introduced component-hypertrees [PN11], Passat and Naegel showed how to generalize their pseudo-distance to perform image segmentation. Additionally, in a previous publication [MAH15], we showed an implementation of a way of computing variation of attributes in a non-optimized component-hypertree.

1.2.3 Connectivity Used to Obtain the CCs

When extracting connected components of an image, it is common to use simple connectivities such as 4-connectivity or 8-connectivity. However, it is worth noting that there exist other types of connectivities [Ser98, BNG02, SW09], that can be used to define more general types of connectivities. They include concepts such as dilation-based connectivities and mask-based connectivities.

Regarding component-trees, in 2007, Ouzounis and Wilkinson [OW07a] proposed an algorithm to build a component tree using mask-based connectivity. A mask is another image that is used to both join originally disjoint CCs (according to a default connectivity) or divide a node into smaller subnodes. This algorithm builds the tree of a d dimensional image by running a typical algorithm construction in a $d + 1$-dimensional image consisting of the mask image stacked on the original image.

Mask-based connectivity was also the choice of Passat and Naegel [PN11] for their component-hypertree. By contrast, for our algorithm published in ISMM 2015 [MAH15], a specific family of dilation-based connectivity was chosen instead, in order to have desired properties that allow the development of faster algorithms. It is worthy of note, however, that mask-based connectivities are more general than our dilation-based connectivity.

1.2.4 Other Types of Graphs Used to Represent Images

As stated before, in this text we focus mainly on using component-hypertrees to represent gray-level images, but there are many other different ways of storing images using graphs.

A type of tree closely related to component-trees is the tree of shapes [MG00]. Nodes of a tree of shapes are connected components of both lower and upper level sets of the input image with their holes filled. These nodes are usually called shapes, and they are also organized according to their inclusion relation. However, although tree of shapes store nodes representing shapes from both upper and lower level sets, shapes obtained from opposite sets must have different connectivities (e.g., 4-connectivity and 8-connectivity) to avoid topological inconsistencies.

To avoid this problem, Song proposed the level lines tree [Son07], which is essentially a tree of shapes built using 6-connectivity (for 2-dimensional images). Contrary to 4 and 8-connectivities, this choice allows both upper and lower shapes to use the same connectivity, in spite of the fact that 6-connectivity is not commonly as used as 4 and 8-connectivities.

One more example closely related to component-trees is the α-tree [Soi07]. In this representation, adjacent pixels of a given image are merged into the same region if the difference between their gray-levels is lower than a given threshold α. Thus, coarser partitions are then obtained by using increasing values of α, and the regions of these partitions can be stored in a tree.

Another graph-based structure consists of the binary partition tree [SG00]. In this structure, an initial partition of the image is given and neighboring regions are then merged together based on a predefined criterion, progressively defining coarser partitions. Hence, this structure can be modeled as a tree, where leaves give the initial partition whereas the intermediate nodes show how the elements of these partitions are merged.

Finally, when gray-level images are replaced by multivalued images, the concept of component-trees can be generalized into component-graphs [PN14, PNK19]. This strategy also led to new results on tree of shapes for multivalued images, proposed by Carlinet [CG15]. It is important to note that, since values of pixels in multivalued images may not follow a total order, the underlying graph may consist of a directed acyclic graph instead of a tree.

In the field of hierarchical models, recent efforts were geared towards the design of strategies allowing to gather information provided by several trees, such as component-hypertrees. However, contrary to component-trees, component-hypertrees were not as widely adopted. In fact, since the original work proposed by Passat and Naegel [PN11], not many other studies were published in this regard, aside from our own paper published in 2015 [MAH15] that aimed at developing an efficient algorithm for component-hypertree construction for dilation-based connectivities.

To fill this gap, in the years that led to the writing of this thesis, we published three papers aiming at further developing the theory behind efficient computation of component-hypertrees. The first paper [MAS+19b] presents an efficient way of storing component-hypertrees, using a graph that avoids storing repeated nodes and reduces redundancy of arcs. The second one [MAS+19a] explains how to perform attribute computation in this optimized structure in a fast way and, finally, the last published paper [MPAH20] explains how the choice of the neighborhood affects computational complexity of the component-hypertree building algorithms and shows how properties of some types of connectivities can be used to design neighborhoods particularly suited for fast component-hypertree computation.

1.3 Text Organization

This text is organized as follows. The background is divided into two parts, where Chap. 2 explains the theoretical background and Chap. 3 contains the algorithmic background. Our main contributions start in Chap. 4, where we explain in more details the concepts behind the papers published during the writing of this thesis, that are used for efficient component-hypertree construction, storage and manipulation. Additionally, we also present some properties and proofs that did not fit in the published papers, to give some insights on the choice of the data structure used and how the underlying algorithms work. To validate the efficiency of the proposed methods, complexity analysis and experimental results are presented in Chap. 5. A conclusion and topics for future researches are proposed in Chap. 6.

Chapter 2

Theoretical Background

In this chapter, we start introducing the background that will be later used to present the proposed method. The background is divided into two chapters and, in this first one, we introduce the theoretical background, presenting concepts and properties that are used to formally define component-hypertrees, the main object of study of this thesis.

2.1 Sets and Partitions

In mathematics, a set represents a collection of elements. Given a set S, the number of elements of a set S is denoted by $|S|$ and an empty set is denoted by $\emptyset$.

If we are interested in listing all elements of a set S, then the concept of enumerated sets can be used. An enumerated set consists of a pair (S, ν), in which S is a set and $\nu : \{1, \dots, |S|\} \subset \mathbb{N}^* \to S$ is a bijective function. This function ν is called an enumeration of S. The list $\big(\nu(1), \dots, \nu(|S|)\big)$ contains all elements of S and will be called a sequence. A sequence composed of two objects is called an ordered pair.

Given two sets S and S', a binary relation $\mathcal{R}$ over S and S' is a set of ordered pairs (s, s') where $s \in S$ and $s' \in S'$. In other words, $\mathcal{R} \subset S \times S'$, where $S \times S' = \{(s, s') : s \in S, s' \in S'\}$ denotes the Cartesian product of S by S'.

Given a set S and a binary relation $\mathcal{R}$ defined as $\mathcal{R} \subset S \times S$, we say that $\mathcal{R}$ is a symmetric relation if, for any $(s, s') \in \mathcal{R}$, we have $(s', s) \in \mathcal{R}$. In this case, we also represent the elements of $\mathcal{R}$ as unordered pairs $\{s, s'\}$. In this text, when the elements of a binary relation $\mathcal{R}$ are unordered pairs, it is clear by context that $\mathcal{R}$ is a symmetric relation.

Let S and S' be two sets. If for any $s \in S'$ we have that $s \in S$, then we say that S' is a subset of S and we write $S' \subseteq S$. If S' is a subset of S and $|S'| < |S|$, then we write $S' \subset S$. In particular, if S and S' are sequences and $S' \subseteq S$, we say that S' is a sub-sequence of S and, in this case, the elements of S' must appear in the same order they appear in S.

The set containing all possible combinations of subsets of a given set S is called the power set of S and denoted by $\mathcal{P}(S)$. Finally, given a set S, a partition of S consists of a collection $\mathcal{H}$, where:

1. For any $R \in \mathcal{H}, R \neq \emptyset$.
2. For any $R, R' \in \mathcal{H}$ such that $R \neq R'$, $R \cap R' = \emptyset$.
3. $\bigcup\limits_{R \in \mathcal{H}} R = S$.

2.2 Images

For the purposes of this text, gray-levels images are used. A gray-level image is defined as a mapping $f : D_f \to \mathbb{K}$, in which $D_f \subset \mathbb{Z}^d$ and $\mathbb{K} = \{0, \dots, K-1\}$. In particular, $d > 0$ and $K > 0$

are integer numbers indicating the number of dimensions and gray-levels of f, respectively. The elements of D_f are called pixels and the gray-level of a pixel p is the value returned by $f(p)$.

If $K = 2$, then f is a binary image. In binary images, pixels satisfying $f(p) = 1$ are called foreground pixels, while the remaining ones are called background pixels. Any binary image can be represented by its set of foreground pixels $X = \{p \in D_f : f(p) = 1\}$. From now on, we use the set representation when referring to binary images.

Gray-level images can be transformed into binary images using a technique known as thresholding. Let f be a gray-level image and λ an integer number. Then, the binary image $X_\lambda(f)$ obtained from thresholding f using λ is defined as follows:

$$X_\lambda(f) = \{p \in D_f : f(p) \geq \lambda\} \tag{2.1}$$

The binary image returned by Eq. (2.1) is known as the upper level set of f at level λ. An example showing the upper level sets of a gray-level image is given in Fig. 2.1.

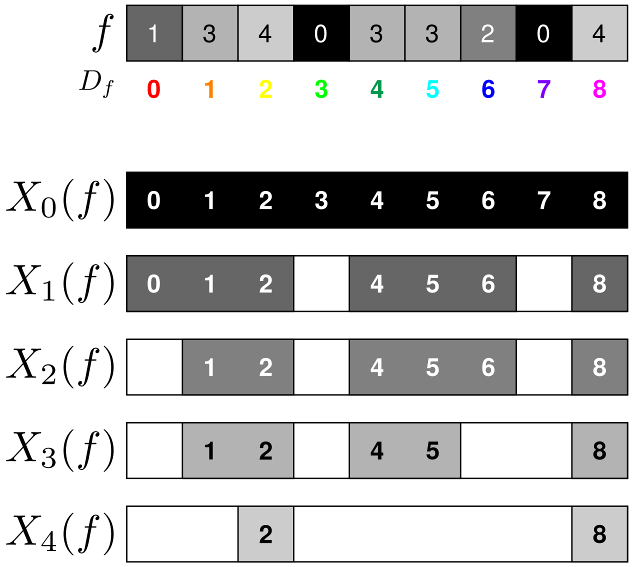


Figure 2.1: *Top: an example of an 1d image with $D_f = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ and $\mathbb{K} = \{0, 1, 2, 3, 4\}$. The numbers inside the squares represent the gray-levels of each pixel of f. Bottom: the upper level sets of $X_\lambda(f)$, where non-white pixels represent foreground pixels and white pixels represent background pixels. In this case, the numbers inside each level set represent the foreground pixels of $X_\lambda(f)$.*

Naturally, there is a dual way of obtaining binary images consisting of changing the operation to $\leq$ that generates the lower level set of f at level λ. Since these definitions are dual, in this text, only upper level sets are taken into consideration, but all properties valid in this text are also valid for lower level sets in its dual form.

2.3 Neighborhood Relations

To relate pixels of an image, the concept of neighborhood relation is used. Given an image with domain D_f, a neighborhood relation (or simply neighborhood) is a binary relation over $D_f \times D_f$.

Thus, let $\mathcal{A} \subset D_f \times D_f$ denote a neighborhood. If p is related to q, then we write $(p, q) \in \mathcal{A}$ and say that p is a $\mathcal{A}$-neighbor of q. If $\mathcal{A}$ is a symmetric neighborhood (in other words, $(p, q) \in \mathcal{A} \Leftrightarrow (q, p) \in \mathcal{A}$), then we say that p and q are $\mathcal{A}$-neighbors.

There are different ways of defining neighborhoods of digital images. Most classical neighborhoods are defined using distance measures. For example, a commonly used distance measure for digital images consists of the taxicab distance d_1, defined as:

$$d_1(p = (p_1, \ldots, p_d), q = (q_1, \ldots, q_d)) = \sum_{\ell=1}^{d} |p_\ell - q_\ell| \tag{2.2}$$

Another distance commonly used is the Chebyschev distance, which is defined as:

$$d_{Chev}(p = (p_1, \ldots, p_d), q = (q_1, \ldots, q_d)) = \max_{\ell=1}^{d} |p_\ell - q_\ell| \tag{2.3}$$

Using these definitions, one can define neighborhood relations $\mathcal{A}(d_1)$ and $\mathcal{A}(d_{Chev})$, as follows:

$$\mathcal{A}(d_1) {=} \{\{p, q\} : p \in D_f, q \in D_f, d_1(p, q) = 1\} \tag{2.4}$$

$$\mathcal{A}(d_{Chev}) {=} \{\{p, q\} : p \in D_f, q \in D_f, d_{Chev}(p, q) = 1\} \tag{2.5}$$

The neighborhood $\mathcal{A}(d_1)$ defines the classical $2d$-connected neighborhoods and $\mathcal{A}(d_{Chev})$ defines $(3^d - 1)$-connected neighborhoods in d-dimensional images. For example, for $d = 2$, $\mathcal{A}(d_1)$ defines 4-connected neighborhood and $\mathcal{A}(d_{Chev})$ defines 8-connected neighborhood.

Alternatively, the dilation operator from Mathematical Morphology can be used to define neighborhoods. In this case, the neighborhood relations are referred as dilation-based neighborhoods. These neighborhoods belong to the class of dilation-based connectivity class [BNG02].

To define this type of neighborhood, let f be a gray-level image with domain D_f, $P \subseteq D_f$ and $\mathcal{S} \subset \mathbb{Z}^d$. Then, the dilation of P by $\mathcal{S}$ is denoted by $P \oplus \mathcal{S}$ and defined as the Minkowski sum of P by $\mathcal{S}$:

$$P \oplus \mathcal{S} = \{p + s : p \in P, s \in \mathcal{S}\} \tag{2.6}$$

The set $\mathcal{S}$ in Eq. (2.6) is known as a structuring element (SE). In this text, given a SE $\mathcal{S}$, we assume that the element $(0, \ldots, 0) = 0^d$ is always in $\mathcal{S}$. We call this element the origin and denote it by o.

Given any structuring element $\mathcal{S}$, a neighborhood relation $\mathcal{A}(\mathcal{S})$ can be defined, as follows:

$$\mathcal{A}(\mathcal{S}) = \{(p, q) : p \in D_f, q \in \{p\} \oplus \mathcal{S}\} \tag{2.7}$$

It is not difficult to prove that classical neighborhoods can be defined using dilation-based neighborhoods by choosing the proper structuring element. For the examples provided, given a pixel $p \in \mathcal{D}_f$, if $\mathcal{S}$ is defined as $\mathcal{S} = \{s \in \mathbb{Z}^d : d_1(s, p) = 1\}$, then $\mathcal{A}(\mathcal{S})$ defines the neighbors of p using $(2d)$-connected neighborhoods. If $\mathcal{S} = \{s \in \mathbb{Z}^d : d_{Chev}(s, p) = 1\}$, then $\mathcal{A}(\mathcal{S})$ defines the neighbors of p using $(3^d - 1)$-connected neighborhoods.

Given a structuring element $\mathcal{S}$, the reflection of $\mathcal{S}$ is denoted by $\breve{\mathcal{S}}$ and defined as follows:

$$\breve{\mathcal{S}} = \{-s : s \in \mathcal{S}\} \tag{2.8}$$

If a structuring element $\mathcal{S}$ satisfies $\mathcal{S} = \breve{\mathcal{S}}$, we say that $\mathcal{S}$ is a symmetric SE. In this case, the following property holds:

Proposition 2.1. *Let $\mathcal{S}$ be a symmetric SE and $\mathcal{A}(\mathcal{S})$ be the neighborhood relation defined using Eq. (2.7). Then, $\mathcal{A}(\mathcal{S})$ is a symmetric neighborhood.*

2.4 Graphs

2.4.1 General Definitions

A graph $\mathcal{G}$ is a pair $\mathcal{G} = (V, \mathcal{E})$, where V is the set of elements (or vertices) and $\mathcal{E} \subseteq V \times V$ is a binary relation on $V \times V$. The graph $\mathcal{G}$ is called a directed graph if $\mathcal{E}$ is not a symmetric relation, and a undirected graph otherwise. In the directed case, the elements of $\mathcal{E}$ are called arcs, while in the undirected case, these elements are called edges.

A path in a graph $\mathcal{G} = (V, \mathcal{E})$ is a sequence of vertices $(v_1, \ldots, v_{LP})$ such that every $v_\ell \in V$ (for $1 \leq \ell \leq LP$) and every pair $(v_\ell, v_{\ell+1}) \in \mathcal{E}$ (for $1 \leq \ell < LP$). A path between two elements $v, v' \in V$ of a graph $\mathcal{G}$ is denoted by $\pi(\mathcal{G}, v, v')$.

If there is path from v to v' in $\mathcal{G}$, we say that v is connected to v' in $\mathcal{G}$ and, if $\mathcal{E}$ is symmetric, then v' is also connected to v, and we simply say that v and v' are connected. Given an undirected graph $\mathcal{G} = (V, \mathcal{E})$, a connected component (CC) C of $\mathcal{G}$ is a maximal set of connected pixels, that is, for any $v, v' \in C$, v and v' are connected in $\mathcal{G}$. In this text, defining CCs only for undirected graphs is sufficient.

If a path starts and ends at the same vertex, this path is called a cycle, and a graph without cycles is called an acyclic graph. A cycle in a directed graph is commonly called a directed cycle, while a directed graph without cycles is called a directed acyclic graph, or DAG.

Given a graph $\mathcal{G}$, a subgraph $\mathcal{G}'$ of $\mathcal{G}$ is a graph $\mathcal{G}' = (V', \mathcal{E}')$ such that $V' \subseteq V$ and $\mathcal{E}' \subseteq \mathcal{E}$. Given $\mathcal{G} = (V, \mathcal{E})$ and a set $V' \subset V$, the vertex-induced subgraph $\mathcal{G}' = (V', \mathcal{E}')$ is the subgraph of $\mathcal{G}$ such that $\mathcal{E}' \subseteq \mathcal{E}$ is composed of all the arcs $(v, v') \in \mathcal{E}$ satisfying $v, v' \in V'$. In this case, the notation $\mathcal{G}' = \mathcal{G}[V]$ is used.

2.4.2 Trees and Directed Acyclic Graphs

Let $\mathcal{G} = (V, \mathcal{E})$ be a DAG. In this case, vertices of V are also called nodes.

If $(v, v') \in \mathcal{E}$, then we say that v is a child of v' and v' is a parent of v. Given a node $v \in V$, we denote its set of children (in $\mathcal{G}$) as $\mathit{child}(\mathcal{G}, v)$ and if v has no child, $(\mathit{child}(\mathcal{G}, v) = \emptyset)$, then v is called a leaf. The set of parents of a node v is given by the mapping $\mathit{par}(\mathcal{G}, v)$ and, if v does not have any parent (that is, $\mathit{par}(\mathcal{G}, v) = \emptyset$), then v is a root node.

Given two nodes $v, v' \in V$ such that v is connected to v' in $\mathcal{G}$, then v is a descendant of v' and v' is an ancestor of v. In this way, set of descendants of v in $\mathcal{G}$ is denoted by $\mathit{desc}(\mathcal{G}, v)$ and its set of ancestors is denoted by $\mathit{anc}(\mathcal{G}, v)$.

If a DAG $\mathcal{G} = (V, \mathcal{E})$ satisfies $|V| = |\mathcal{E}| + 1$ and there is a node v_r such that any other $v \in V$ is connected to v_r, then $\mathcal{G}$ is a tree. In particular, v_r is the root of $\mathcal{G}$. Given a tree $\mathcal{G} = (V, \mathcal{E})$ and a node $v \in V$, the subtree (of $\mathcal{G}$) rooted in v consists of the subgraph $\mathcal{G}'$ induced by the set $V' = \{v\} \cup \mathit{desc}(\mathcal{G}, v)$. Note that $\{v_r\} \cup \mathit{desc}(\mathcal{G}, v_r) = V$.

In a tree, the depth of a node v is denoted by $\mathit{depth}(\mathcal{G}, v)$ and defined as the number of arcs in the path from v to the root node. If v is a root, then $\mathit{depth}(\mathcal{G}, v) = 0$. A graph composed of one or more trees is called a forest.

2.4.3 Images as Graphs

Graphs can be used to represent binary images: given a binary image X and a neighborhood $\mathcal{A}$, it is possible to represent X and $\mathcal{A}$ using the graph $\mathcal{G} = (X, \mathcal{A})$.

To represent gray-level images, we introduce the concept of weighted graphs. A graph $\mathcal{G}$ is a vertex-weighted graph if it is composed of a triple $\mathcal{G} = (V, \mathcal{E}, \omega)$, where $\omega : V \to \mathbb{R}$ is a function that assigns a numerical weight to each vertex. If a gray-level image f and a neighborhood $\mathcal{A}$ are given, then they can be represented as the weighted graph $(D_f, \mathcal{A}, f)$. An example is provided in Fig. 2.2.

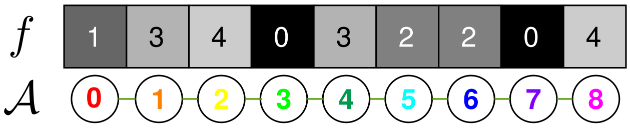


Figure 2.2: *A graphic illustration of an image represented as a vertex-weighted graph $\mathcal{G} = (V = D_f, \mathcal{E} = \mathcal{A}, \omega = f)$. The top row represents the gray-levels and the bottom row shows the domain of the image and the neighborhood relations, with pixels represented as vertices and neighboring pixels presented as arcs of the graph.*

2.5 Connectedness in Images

With the concepts of images and graphs defined, we now introduce notions of connectedness in images.

2.5.1 Binary Images

Let X be a binary image and $\mathcal{A}$ a symmetric neighborhood. We say that two pixels $p, q \in X$ are $\mathcal{A}$-connected (or simply connected when $\mathcal{A}$ is clear from context) if they are connected in the graph $\mathcal{G} = (X, \mathcal{A})$. Additionally, the set of $\mathcal{A}$-connected components ($\mathcal{A}$-CCs) of X, denoted by $CC(X, \mathcal{A})$, is defined as the set of CCs of the graph $\mathcal{G} = (X, \mathcal{A})$:

$$CC(X, \mathcal{A}) = \{C : C \text{ is a CC of } \mathcal{G} = (X, \mathcal{A})\}. \tag{2.9}$$

An example is given in Fig. 2.3.

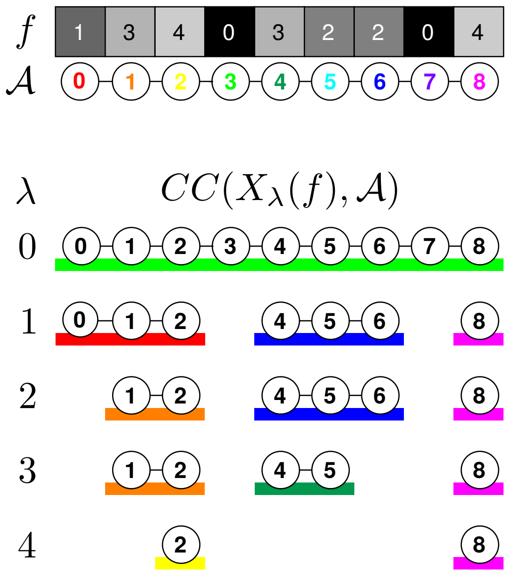


Figure 2.3: *The sets $CC(X_\lambda(f), \mathcal{A})$ extracted from the level sets of the weighted graph from Fig. 2.2. Each $\mathcal{A}$-CC is represented by a colored horizontal bar.*

The set of $\mathcal{A}$-CCs of X satisfies the following property:

Proposition 2.2. *The set $CC(X, \mathcal{A})$ forms a partition of X.*

Proposition 2.2 guarantees that, for any $p \in X$, there is exactly one $\mathcal{A}$-CC that contains p. In particular, let $CC(X, \mathcal{A}, p)$ denote the $\mathcal{A}$-CC of X that contains p. Then, it can defined as:

$$CC(X, \mathcal{A}, p) = \begin{cases} C \in CC(X, \mathcal{A}) \text{ such that } p \in C & \text{, if } p \in X; \\ \emptyset & \text{, otherwise.} \end{cases} \tag{2.10}$$

2.5.2 Gray-Level Images

Let f be a gray-level image and $\mathcal{A}$ a symmetric neighborhood. The set of $\mathcal{A}$-connected components ($\mathcal{A}$-CCs) of f is denoted by $CC(f, \mathcal{A})$ and is defined as follows:

$$CC(f, \mathcal{A}) = \bigcup_{\lambda \in \mathbb{K}} CC(X_\lambda(f), \mathcal{A}). \tag{2.11}$$

We say that two pixels p and q are $\mathcal{A}$-connected in f if and only if, for all level set $X_\lambda(f)$ where p and q are foreground pixels, p and q are $\mathcal{A}$-connected. An equivalent way of stating that two pixels are connected is given in Prop. 2.3.

Proposition 2.3. *Let f be a gray-level image, $\mathcal{A}$ a neighborhood and $p, q \in D_f$. Then, p is $\mathcal{A}$-connected to q if and only if there is a path $\pi = (p = p_1, \ldots, p_{LP} = q)$ from p to q such that, for all $1 \leq \ell \leq LP$, $f(p_\ell) \geq \min\{f(p), f(q)\}$.*

Additionally, the set of $\mathcal{A}$-CCs of f satisfies this important property:

Proposition 2.4. *Let $K > \alpha > \lambda \geq 0$. Then, for any image f and any symmetric neighborhood $\mathcal{A}$, the $\mathcal{A}$-CCs of f are decreasing, that is:*

$$CC(X_\alpha(f), \mathcal{A}, p) \subseteq CC(X_\lambda(f), \mathcal{A}, p), \forall p \in D_f \tag{2.12}$$

An implication of Prop. 2.4 is that the set of $\mathcal{A}$-CCs of any image f can be represented as a hierarchy based on inclusion relation. There are multiple ways of representing this hierarchy. One possible way consists of defining a graph using the following sets of vertices:

$$RC(f, \mathcal{A}) = \{(C, \lambda) : C \in CC(X_\lambda(f), \mathcal{A}), \lambda \in \mathbb{K}\}, \tag{2.13}$$

Then, thanks to Prop. 2.4, we can organize them according to their gray-levels. More specifically, we can define a binary relation as follows:

$$RE(f, \mathcal{A}) = \{((C, \alpha), (C', \lambda)) \in RC(f, \mathcal{A}) \times RC(f, \mathcal{A}) : C \subsetneq C' \text{ and } \alpha = \lambda + 1\} \tag{2.14}$$

Then, this hierarchy can be represented as the graph $CCT(f, \mathcal{A}) = (RC(f, \mathcal{A}), RE(f, \mathcal{A}))$. This graph CCT is a tree and will be called the complete component-tree. An example is given in the leftmost column of Fig. 2.4.

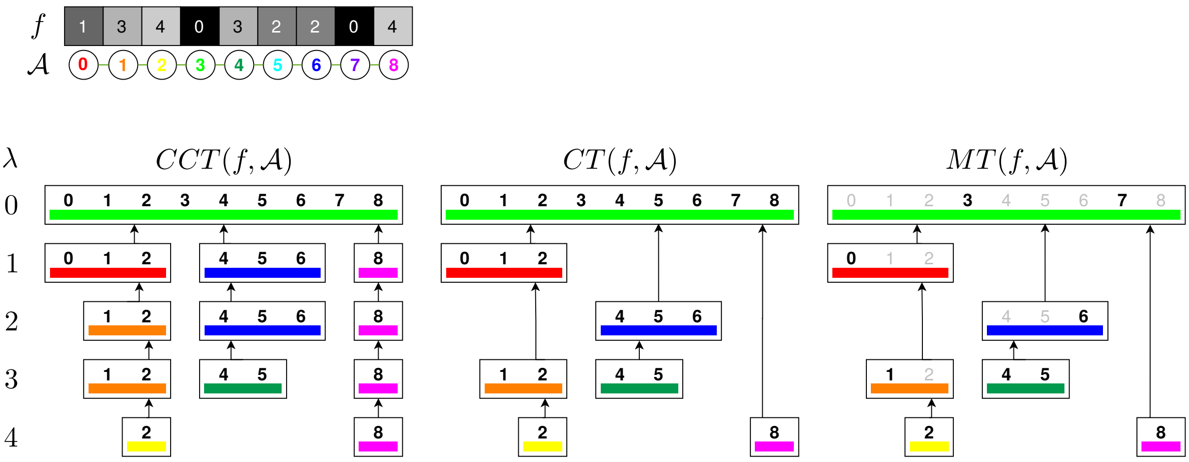


Figure 2.4: *Top: an image f with symmetric neighborhood $\mathcal{A}$. Bottom, from left to right: the complete component-tree, the component-tree and the max-tree of f using $\mathcal{A}$. The numbers inside the node indicated the elements that each node contains, and the colored bars indicate the $\mathcal{A}$-CCs that each node represents.*

An alternative way of representing the hierarchy of $\mathcal{A}$-CCs of an image f is to consider only the set of $\mathcal{A}$-CCs of f. In this case, the binary relation linking the $\mathcal{A}$-CCs is defined as follows:

$$\begin{aligned} CE(f, \mathcal{A}) = &(C, C') \in (CC(f, \mathcal{A}) \times CC(f, \mathcal{A})) : C \subset C' \text{ and} \\ &\nexists C'' \in CC(f, \mathcal{A}) \text{ such that } C \subset C'' \subset C' \end{aligned} \tag{2.15}$$

Let $CT = (CC(f, \mathcal{A}), CE(f, \mathcal{A}))$ be a graph where $CC(f, \mathcal{A})$ is the set of vertices and $CE(f, \mathcal{A})$ is the set of edges. This graph CT is a tree known as the component-tree of f (and $\mathcal{A}$). An example of component-tree is depicted in the middle column of Fig. 2.4.

Finally, a third way of representing the hierarchy of $\mathcal{A}$-CCs consists of representing the nodes of the component-tree in a compact way. For that, let $cnp : CC(f, \mathcal{A}) \to \mathcal{P}(D_f)$ be the following mapping:

$$cnp(C) = C \setminus \Big(\bigcup_{C' \in child(CT,C)} C' \Big) \tag{2.16}$$

Then, the max-tree of f and $\mathcal{A}$ is the graph $MT = (MC(f, \mathcal{A}), ME(f, \mathcal{A}))$, where:

$$MC(f, \mathcal{A}) = \{cnp(C) : C \in CC(f, \mathcal{A})\} \tag{2.17}$$

$$ME(f, \mathcal{A}) = \{(cnp(C), cnp(C')) : (C, C') \in CE(f, \mathcal{A})\} \tag{2.18}$$

An example of a max-tree is given in the rightmost column of Fig. 2.4.

Given a max-tree $MT = (MC(f, \mathcal{A}), ME(f, \mathcal{A}))$, the $\mathcal{A}$-connected components of f can be reconstructed from their respective nodes $N \in MC(f, \mathcal{A})$ using the following operation:

$$rec(MT, N) = N \cup \Big(\bigcup_{N' \in child(MT,N)} rec(MT, N') \Big) \tag{2.19}$$

In particular, using Eq. (2.19) one can prove that:

Proposition 2.5. *Let f be a gray-level image, $\mathcal{A}$ a symmetrical neighborhood, CT be the component-tree of f (using $\mathcal{A}$) and MT be the max-tree of f (using $\mathcal{A}$). Additionally, let C be a node of CT and $N = cnp(C)$ be a node of MT. Then, $rec(MT, N) = C$.*

2.6 Connectedness in Increasing Neighborhoods

With the concepts of connectedness in images defined, let us now consider the case where increasing neighborhoods are taken into consideration. First, let us restrict to the case of two increasing symmetric neighborhoods $\mathcal{A}$, $\mathcal{A}'$ where $\mathcal{A}' = \mathcal{A} \cup \{p, q\}$. Then, the following properties are valid:

Proposition 2.6. *Let X be a binary image and $\mathcal{A}, \mathcal{A}'$ be two neighborhoods satisfying $\mathcal{A}' = \mathcal{A} \cup \{p, q\}$. If p and q are $\mathcal{A}$-connected, then $CC(X, \mathcal{A}) = CC(X, \mathcal{A}')$.*

Proposition 2.7. *Let X be a binary image, $p' \in X$ and $\mathcal{A}, \mathcal{A}'$ be two neighborhoods satisfying $\mathcal{A}' = \mathcal{A} \cup \{p, q\}$. If p and q are not $\mathcal{A}$-connected, then:*

$$CC(X, \mathcal{A}', p') = \begin{cases} CC(X, \mathcal{A}, p') & \text{, if } p' \notin CC(X, \mathcal{A}, p) \text{ and } p' \notin CC(X, \mathcal{A}, q); \\ CC(X, \mathcal{A}, p) \cup CC(X, \mathcal{A}, q) & \text{, otherwise.} \end{cases} \tag{2.20}$$

An example is given in Fig. 2.5. As an implication of Prop. 2.7, one can observe that the difference between $\mathcal{A}$ and $\mathcal{A}'$, at every gray-level where p and q belong to disjoint CCs $C_p, C_q \in CC(f, \mathcal{A})$, is that they are now merged into a single component $C_p \cup C_q$ in $CC(f, \mathcal{A}')$. All other $\mathcal{A}'$-CCs remain the same as they did in $CC(f, \mathcal{A})$.

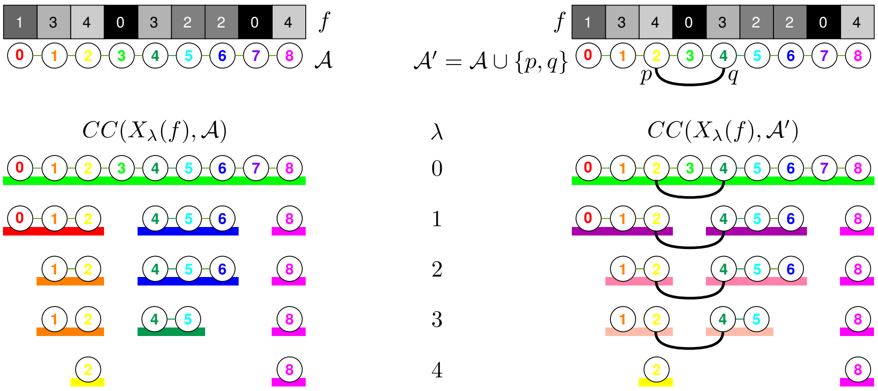

1 3 4 0 3 2 2 0 4
f
0 1 2 3 4 5 6 7 8
$\mathcal{A}$
$\mathcal{A}' = \mathcal{A} \cup \{p, q\}$
p
q
$CC(X_\lambda(f), \mathcal{A})$
λ
$CC(X_\lambda(f), \mathcal{A}')$
0
1
2
3
4

Figure 2.5: *Comparison between $CC(X_\lambda(f), \mathcal{A})$ and $CC(X_\lambda(f), \mathcal{A}')$, where $\mathcal{A}' = \mathcal{A} \cup \{p = 2, q = 4\}$. Different CCs are represented using different colors. Note that all $\mathcal{A}'$-CCs that are not $\mathcal{A}$-CCs are merges of two originally disjoint components: one containing p and the other containing q.*

Now, suppose the more general case $\mathcal{A}_1 \subset \mathcal{A}_2$. Given a binary image X and $p, q \in X$, it is plain that if p and q are $\mathcal{A}_1$-connected, than p and q are also $\mathcal{A}_2$-connected, since all arcs from $\mathcal{A}_1$ that create the path from p to q also exist in $\mathcal{A}_2$. Hence:

Proposition 2.8. *Let X be a binary image and $\mathcal{A}_1, \mathcal{A}_2$ be two neighborhoods satisfying $\mathcal{A}_1 \subset \mathcal{A}_2$. Then, for any $\mathcal{A}_1$-CC C, there exists a $\mathcal{A}_2$-CC C' such that $C \subseteq C'$.*

In other words, given a fixed binary image X, when two or more increasing neighborhoods are taken into consideration, there also exist inclusion relations between CCs of X obtained from these different neighborhoods.

2.7 Component-Hypertrees

Component-hypertrees [PN11] can be seen as extensions of component-trees for multiple increasing neighborhoods. In this sense, suppose a gray-level image f is given but, instead of a single neighborhood $\mathcal{A}$, an increasing sequence of symmetric neighborhoods $\mathbb{A} = (\mathcal{A}_1, \dots, \mathcal{A}_n)$ is given, expressly $\mathcal{A}_i \subseteq \mathcal{A}_{i+1}$ for any $1 \leq i < n$ and all of these $\mathcal{A}_i$ $(1 \ \leq i \leq n)$ are symmetric.

Then, connected components of the level sets of f can be extracted using different neighborhoods. Similar to complete component-trees, these CCs form a set of vertices $V_{CCH}(f, \mathbb{A})$, defined as follows:

$$V_{CCH}(f, \mathbb{A}) = \bigcup_{\substack{0 \leq \lambda < K \\ 1 \leq i \leq n}} \{(C, \lambda, i) : C \in CC(X_\lambda(f), \mathcal{A}_i)\}. \tag{2.21}$$

Given a node $N = (C, \lambda, i) \in V_{CCH}(f, \mathbb{A})$, we refer to the elements stored in N as follows:

- $CC(N) = C$;
- $f(N) = \lambda$;
- $adj(N) = i$.

Arcs of component-hypertrees are determined by the inclusion relation of the nodes. However, contrary to component-trees, these arcs can indicate two different types of inclusion relations: they can link two nodes from the same component-tree or link nodes from consecutive component-trees. Hence, the set of arcs is divided into two sets $\mathcal{E}^{\uparrow}_{CCH}$ and $\mathcal{E}^{\rightarrow}_{CCH}$.

The first set refers to arcs from the same component-tree, and is defined as follows:

$$\mathcal{E}^{\uparrow}_{CCH}(f, \mathbb{A}) = \{((C, \lambda, i), (C', \lambda', i')) : C \subseteq C', \lambda = \lambda' + 1, i = i'\}. \tag{2.22}$$

An arc from the set $\mathcal{E}^{\uparrow}_{CCH}(f, \mathbb{A})$ will still be called a parent arc. In this way, if $e = (N, N')$ is a parent arc, then N is a child node of N' and N' is a parent node of N.

The second set of arcs links nodes from consecutive component-trees and is defined as:

$$\mathcal{E}^{\rightarrow}_{CCH}(f, \mathbb{A}) = \{((C, \lambda, i), (C', \lambda', i')) : C \subseteq C', \lambda = \lambda', i = i' - 1\}. \tag{2.23}$$

Arcs from the set $\mathcal{E}^{\rightarrow}_{CCH}(f, \mathbb{A})$ will be called composite arcs. If $e = (N, N')$ is a composite arc, we say that N is a partial node of N' and N' is a composite node of N.

Naturally, the two sets of arcs can be combined into one:

$$\mathcal{E}_{CCH}(f, \mathbb{A}) = \mathcal{E}^{\uparrow}_{CCH}(f, \mathbb{A}) \cup \mathcal{E}^{\rightarrow}_{CCH}(f, \mathbb{A}). \tag{2.24}$$

Finally, a graph $\mathcal{G}_{CCH}(f, \mathbb{A})$ using these sets of vertices and arcs can be defined:

$$\mathcal{G}_{CCH}(f, \mathbb{A}) = (V_{CCH}(f, \mathbb{A}), \mathcal{E}_{CCH}(f, \mathbb{A})) \tag{2.25}$$

This graph $\mathcal{G}_{CCH}(f, \mathbb{A})$ defines the complete component-hypertree (or simply complete-hypertree) of f using $\mathbb{A}$. It is worth mentioning that component-hypertree is simply a denomination given by the original authors for this particular structure. This means that component-hypertrees are not hypertrees in the context of hypergraphs. In fact, component-hypertrees are actually directed acyclic graphs. An example is provided in Fig. 2.6.

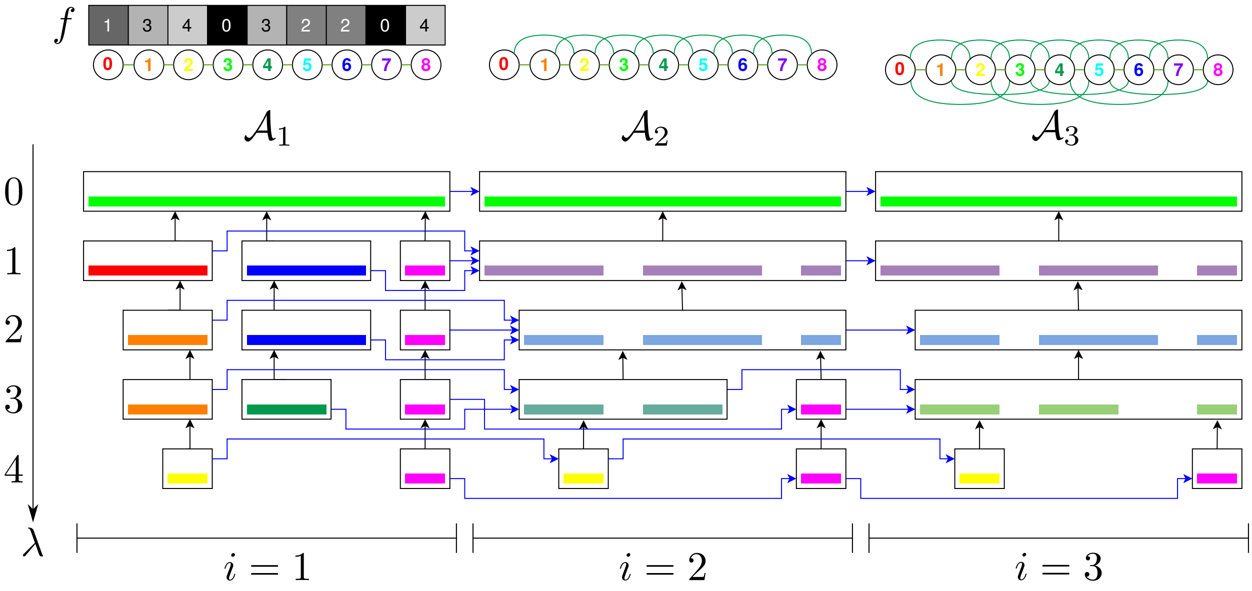


Figure 2.6: *An example of a complete component-hypertree. Parent arcs are represented using black arrows, while composite arcs are represented using blue arrows.*

To simplify the notation, we may denote a complete component-hypertree simply as $\mathcal{G}_{CCH}$ when the input image f and sequence $\mathbb{A}$ are clear from the context. Thus, given a complete hypertree $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$, let $N \in V_{CCH}$. Just like before, the parent of N is denoted by $par(\mathcal{G}_{CCH}, N)$ and its set of children by $child(\mathcal{G}_{CCH}, N)$. Analogously, the composite of N is denoted by $comp(\mathcal{G}_{CCH}, N)$ and its set of partial nodes by $part(\mathcal{G}_{CCH}, N)$. It is important to note that, using the terminology defined for general DAGs, composites arcs are also considered parent arcs, but from now on we are limiting the definition of parent arcs of component-hypertrees to arcs linking two nodes with same neighborhood index, while composite arcs link nodes with the same gray-level.

With this distinction in mind, it can be proved that the following property is valid:

Proposition 2.9. *Let $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be a complete component-hypertree and $N \in V_{CCH}$.*

Then, if $par(\mathcal{G}_{CCH}, N) \neq \emptyset$ and $comp(\mathcal{G}_{CCH}, N) \neq \emptyset$, then

$$par(\mathcal{G}_{CCH}, comp(\mathcal{G}_{CCH}, N)) = comp(\mathcal{G}_{CCH}, par(\mathcal{G}_{CCH}, N)) \tag{2.26}$$

Proposition 2.9 implies the following result:

Proposition 2.10. *Let $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be a complete component-hypertree, $N = (C, \lambda, i) \in V_{CCH}$ and λ' and i' two values satisfying one of the two conditions below:*

1. $0 \leq \lambda' < \lambda < K$ *and* $n \geq i' \geq i \geq 1$;
2. $0 \leq \lambda' \leq \lambda < K$ *and* $n \geq i' > i \geq 1$.

Then, there is exactly one node $N' = (C', \lambda', i')$ satisfying $C \subseteq C'$.

An example of Prop. 2.10 is shown in Fig. 2.7.

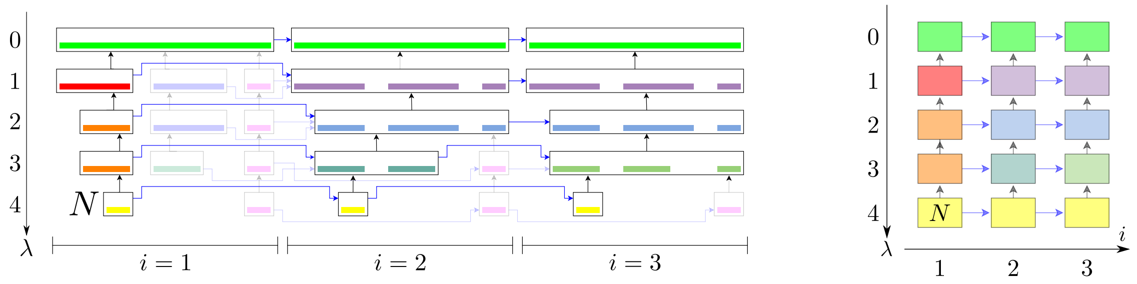


Figure 2.7: *Left: the previous complete-hypertree, with the node $N = (C = \{2\}, \lambda = 4, i = i)$ and all nodes that contain it, highlighted. Right: An alternative way of visualizing the nodes that contain N, organized in a way that show that there is exactly one node for each neighborhood index and gray-level.*

Finally, it is not difficult to show that some subgraphs of the complete-hypertree are complete component-trees. For example, given the complete-hypertree of a gray-level image f and a sequence $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, the subgraph induced by the set of vertices $V = \{N : adj(N) = i\}$ corresponds to the complete component-tree of f using $\mathcal{A}_i$.

Thus, given a component-hypertree $\mathcal{G}_{CCH}$, we denote by $\mathcal{G}_{CCH}^{([\lambda \to \lambda'],[i \to i'])}$ the subgraph of $\mathcal{G}_{CCH}$ induced by the vertices N satisfying $\lambda \leq f(N) \leq \lambda'$ and $i \leq adj(N) \leq i'$. Hence, for any $1 \leq i \leq n$, $\mathcal{G}_{CCH}^{([0 \to K-1],[i \to i])}$ consists of the complete component-tree of f using $\mathcal{A}_i$. In Fig. 2.6, this would consist of a column of the hypertree with a fixed neighborhood index i.

It is also worth noting that a structure analogous to a component-tree can be obtained from the CCs of a binary image considering a sequence of neighborhoods, namely, $\mathcal{G}_{CCH}^{([\lambda \to \lambda],[1 \to n])}$ is a forest that organizes the CCs of $X_\lambda(f)$ according to their inclusion relation. In Fig. 2.6, they consist of rows of the hypertree with a fixed gray-level λ.

With the concept of component-hypertree defined, we now presented all the theoretical definitions needed for explaining the proposed method. Now, we change focus and review known algorithms for component-tree construction. Our goal is to use properties of these algorithms to design an efficient way of computing and storing component-hypertrees.

Chapter 3

Algorithmic Background

In this chapter, we present the algorithmic background required for understanding the proposed method. Here, we recall known algorithms and data structures used for component-tree computation, showing how to implement them and also explaining some good properties that these algorithms satisfy. Understanding all these properties will be crucial for obtaining efficient ways of computing component-hypertrees, which is one of the main goals of this thesis.

To explain these concepts, we follow a bottom-up approach, starting from the simplest cases and then moving towards more advanced ones. Hence, we begin by explaining how to store and compute connected components of images in the simplest case: binary image and, at the end of the chapter, we will have a way of efficiently computing CCs for gray-level images considering any type of neighborhood.

3.1 Representing Forests as Arrays

To begin this chapter, we recall an efficient way of storing rooted trees and forests using arrays. This will be useful later so we have an efficient way of storing component-trees.

Thus, let f be a gray-level image with domain D_f and $\mathcal{G} = (V, \mathcal{E})$ be a directed forest, where $V \subseteq D_f$. Then, it is possible to create a mapping $g : V \to V$ that represents the arcs of $\mathcal{E}$, as follows:

$$(p, q) \in \mathcal{E} \Leftrightarrow g(p) = q \tag{3.1}$$

In terms of data structure, this means that any rooted tree $\mathcal{G} = (V \subseteq D_f, \mathcal{E})$ can be stored using an array ar, in which $\mathsf{ar}[p] = q$ if and only if $(p, q) \in \mathcal{E}$. In particular, given an array ar storing a forest, we denote the graph that ar represents by $\mathcal{G}(\mathsf{ar})$.

For implementation purposes, it is useful to assign a value to $\mathsf{ar}[p_r]$, in which p_r is a root node. In this text, if p_r is a root, then we write $\mathsf{ar}[p_r] = \perp$. In this way, ar can be seen as a mapping $V \to \big(V \cup \{\perp\}\big)$, where:

$$\mathsf{ar}[p] = \begin{cases} q & \text{if } (p, q) \in \mathcal{E}; \\ \perp & \text{, if } p \text{ is a root node.} \end{cases} \tag{3.2}$$

For convenience, for any gray-level image f, we also set $f(\perp) = -1$. This choice simplifies the implementation of some of the algorithms that appear later on this chapter. Additionally, if an array ar represents a forest, then we apply the nomenclature of forests directly to the array ar. For example, p connected to q in ar means that p is connected to q in $\mathcal{G}(\mathsf{ar})$.

3.2 Labeling Connected Components in Binary Images

Before moving to the algorithm for component-tree construction, we start with the simpler case of extracting CCs in binary images. This can be done by using union-find [Tar75], which is a data structure that can be used to represent disjoint sets. It can be thought as a forest, where each tree stores a disjoint set and the roots of these trees are used to represent said sets. An example is depicted in Figure 3.1.

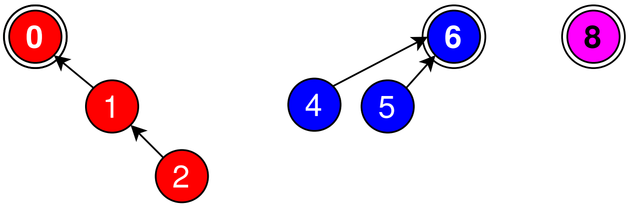


Figure 3.1: *An union-find structure represented as a forest. Roots are represented as double circles. For visualization purposes, all elements that belong to the same tree are presented using the same color. In this case, we have 3 different disjoint sets* $\{0, 1, 2\}$, $\{4, 5, 6\}$ *and* $\{8\}$, *each represented with a different color.*

Representing $\mathcal{A}$-CCs of a binary image X using forests is possible based on two properties of connectedness: the first one is the transitive property: if p is $\mathcal{A}$-connected to q and q is $\mathcal{A}$-connected to r, then the path from p to q combined to the path from q to r forms a path from p to r, and p is also $\mathcal{A}$-connected to r.

Additionally, if $\mathcal{A}$ is a symmetric neighborhood, then connectedness is also commutative, since if there is a path from p to q, then the reverse path from q to p also exists. In this case, given a binary image X and a symmetric neighborhood $\mathcal{A}$, let $C \in CC(X, \mathcal{A})$ and $r \in C$. If we know all pixels that are $\mathcal{A}$-connected to r, then C can be obtained since, for any $p, q \in X$ with $p \neq q$ such that p is $\mathcal{A}$-connected to r and q is also $\mathcal{A}$-connected to r, then p is $\mathcal{A}$-connected to q. By definition, C is a maximal set where any $p, q \in C$ is pair-wise connected, so a way of obtaining C consists simply of finding all pixels $\mathcal{A}$-connected to r.

In other words, a single element $r \in C$ can be used to represent C, if all other elements of X that are $\mathcal{A}$-connected to r are known. This relation can be represented in a forest $\mathcal{G} = (X, \mathcal{E})$, where $p \in X$ is connected to r if and only if there is a path from p to r in $\mathcal{G}$. Then, each tree of $\mathcal{G}$ represents a $\mathcal{A}$-CC, that can be reconstructed by gathering all pixels in each tree. Additionally, two pixels are $\mathcal{A}$-connected if and only if they belong to the same tree, or in other words, they have the same root. This root node is also commonly referred as the canonical element.

Given these definition and properties, now we concentrate on how to implement an algorithm to compute an union-find. First, since a union-find represents a forest, it can be stored using an array $\mathsf{uf} : X \to \left(X \cup \bot\right)$ by employing Eq. (3.2). In our implementation, $\bot$ acts as a way of representing canonical elements, namely, an element $r \in X$ is canonical if and only if $\mathsf{uf}[r] = \bot$.

To implement the union-find, 3 basic operations need to be defined: MAKESET, FIND and UNION.

The MAKESET procedure initializes uf with every element disjoint from each other (see Alg. 1).

Algorithm 1 Initialization of the union-find.

```
procedure MAKESET($X$)
    for $p \in X$ do
        $\mathsf{uf}[p] \leftarrow \bot$;
    return uf;
```

The second operation, FIND, given an element $p \in X$, returns the root of the tree that contains p. This is presented in Alg. 2.

Algorithm 2 FIND procedure, which given a pixel $p \in X$, returns its root node.

```
procedure FIND(uf, $p$)
    while $\mathsf{uf}[p] \neq \bot$ do
        $p \leftarrow \mathsf{uf}[p]$;
    return $p$;
```

Finally, the UNION procedure receives two elements $p, q \in X$ and, if they belong to disjoint trees (that is, FIND applied to p and q return different roots), then these two trees are merged. To do so, it makes the root of the tree containing p point to the root of the tree containing q (the order is not relevant in this case, it could be in the opposite direction as well, unless we want an optimized version of the UNION procedure). Informally, it can be thought as "hanging" the subtree rooted in p on the subtree rooted in q, creating a merged tree where q is the new root.

If p and q already belong to the same subtree, no changes are performed. See Alg. 3.

Algorithm 3 UNION procedure, which returns an updated uf with p and q in the same tree.

```
procedure UNION($\mathsf{uf}, p, q$)
    $rP \leftarrow$ FIND($\mathsf{uf}, p$);
    $rQ \leftarrow$ FIND($\mathsf{uf}, q$);
    if $rP \neq rQ$ then                ▷ if true, $p$ and $q$ belong to different sets
        $\mathsf{uf}[rP] \leftarrow rQ$;
    return $\mathsf{uf}$;
```

Given a binary image X and any symmetric neighborhood $\mathcal{A}$, one can make use of Alg. 4 to build an union-find that stores $CC(X, \mathcal{A})$. This is a very well-known algorithm used for labeling CCs of binary images.

Algorithm 4 Labeling $\mathcal{A}$-CCs of X using the disjoint set structure.

```
1: procedure CCLABELING($X, \mathcal{A}$)
2:     $\mathsf{uf} \leftarrow$ MAKESET($X$);
3:     for $\{p, q\} \in \mathcal{A}$ do
4:         $\mathsf{uf} \leftarrow$ UNION($\mathsf{uf}, p, q$);
```

At the end of Alg. 4, for any $\mathcal{A}$-CC C of X, there is a canonical element r_C of uf that represents C and, for any canonical element r of uf, there is a corresponding $\mathcal{A}$-CC C_r of X. When this happens, we say that uf stores the $\mathcal{A}$-CCs of X, or that uf stores $CC(X, \mathcal{A})$.

Since each disjoint set is represented by (the root of) a forest, it is important to have a mapping that reconstruct the CCs. Hence, we define a mapping rec as:

$$rec(\mathsf{uf}, p) = \{p\} \cup desc(\mathsf{uf}, p) \tag{3.3}$$

Simply put, $rec(\mathsf{uf}, p)$ can be thought as the set of nodes contained in the tree rooted in p.

Although Alg. 4 is a very well-known algorithm, for our purposes, it is useful to explain why the algorithm is correct, since the correctness of our proposed algorithms can be proven using similar approaches. Hence, our aim is now to explain why uf, at the end of Alg. 4, stores $CC(X, \mathcal{A})$. But in order to do that, some preliminary definitions are necessary.

First, we need a way of referring to the variables of the algorithm. In particular, in Line 3 of Alg. 4, the pairs of $\mathcal{A}$-neighbors are processed in a specific order, which defines an enumeration of $\mathcal{A}$. Hence, let $\mathcal{A}_{alg} = (a_1, \ldots, a_{|\mathcal{A}|})$ be the sequence generated from the enumeration of $\mathcal{A}$ defined by the processing order of the neighboring pairs in Alg. 4.

Additionally, let $\mathcal{A}^j_{alg}$ be defined as follows:

$$\mathcal{A}^j_{alg} = \begin{cases} \emptyset & \text{if } j = 0; \\ \bigcup\limits_{\ell=1}^{j} a_\ell & \text{, otherwise.} \end{cases} \tag{3.4}$$

In other words, $\mathcal{A}^j_{alg}$ denotes the set of processed neighbors after the first j loops of Alg. 4. From that, we define uf_j as the union-find resulting from processing all pairs in $\mathcal{A}^j_{alg}$. If $j = 0$, we define uf_0 as the initialized array using the MAKESET procedure.

With these definitions in mind, our final objective is to prove that uf_j stores $CC(X, \mathcal{A}_{alg}^{j})$, for any $0 \leq j \leq |\mathcal{A}|$. Then, when $j = |\mathcal{A}|$, since $\mathcal{A}_{alg}^{|\mathcal{A}|}$ contains exactly the elements of $\mathcal{A}$, then $\mathsf{uf}_{|\mathcal{A}|}$ stores $CC(X, \mathcal{A}_{alg}^{|\mathcal{A}|}) = CC(X, \mathcal{A})$.

This idea can be proved by induction in j. However, before showing the final proof, we analyze how the array uf_{j-1} is updated to produce uf_j. There are two cases to consider: we first analyze the case when $a_j = (p_j, q_j)$ is composed of elements that are $\mathcal{A}_{alg}^{j-1}$-connected.

Proposition 3.1. *Let X be a binary image, $\mathcal{A}$ a symmetric neighborhood and suppose Alg. 4 is called, where $\mathcal{A}_{alg}$ is obtained from an enumeration of $\mathcal{A}$ defined by the processing order of the neighboring pixels in Line 3. In particular, suppose that $a_j = (p_j, q_j)$, uf_{j-1} stores $CC(X, \mathcal{A}_{alg}^{j-1})$ and that p_j and q_j are $\mathcal{A}_{alg}^{j-1}$-connected. Then, $\mathsf{uf}_{j-1} = \mathsf{uf}_j$.*

Proof. If p_j and q_j are $\mathcal{A}_{alg}^{j-1}$-connected, then they belong to the same CC $C \in CC(X, \mathcal{A}_{alg}^{j-1})$. By hypothesis, this means that p_j and q_j have the same root in uf_{j-1}. In this case, the UNION procedure does not make any modifications to uf_{j-1}, and we have that $\mathsf{uf}_{j-1} = \mathsf{uf}_j$.

□

If p_j and q_j are not $\mathcal{A}_{alg}^{j-1}$-connected, then:

Proposition 3.2. *Let X be a binary image, $\mathcal{A}$ a symmetric neighborhood and suppose Alg. 4 is called, where $\mathcal{A}_{alg}$ is obtained from an enumeration of $\mathcal{A}$ defined by the processing order of the neighboring pixels in Line 3. In particular, suppose that $a_j = (p_j, q_j)$, uf_{j-1} stores $CC(X, \mathcal{A}_{alg}^{j-1})$ and that p_j and q_j are not $\mathcal{A}_{alg}^{j-1}$-connected. Then, if uf_j is the updated array after calling UNION for (p_j, q_j), then for any $r \in X$:*

1. *If r was not a canonical element in uf_{j-1}, r is not a canonical element in uf_j;*
2. *Let $rP_j = \text{FIND}(\mathsf{uf}_{j-1}, p_j)$. If $r = rP_j$, then r was a canonical element in uf_{j-1} representing the connected component $rec(\mathsf{uf}_{j-1}, rP_j)$. However, rP_j is not a canonical element in uf_j;*
3. *Let $rQ_j = \text{FIND}(\mathsf{uf}_{j-1}, q_j)$. If $r = rQ_j$, then r was a canonical element in uf_{j-1} representing the connected component $rec(\mathsf{uf}_{j-1}, rQ_j)$ and rQ_j is still a canonical element in uf_j representing the connected component $rec(\mathsf{uf}_j, rQ_j) = rec(\mathsf{uf}_{j-1}, rP_j) \cup rec(\mathsf{uf}_{j-1}, rQ_j)$;*
4. *If r is a canonical element in uf_{j-1} but $r \neq \text{FIND}(\mathsf{uf}_{j-1}, p_j)$ and $r \neq \text{FIND}(\mathsf{uf}_{j-1}, q_j)$, then r is still a canonical element of uf_j, where $rec(\mathsf{uf}_{j-1}, r) = rec(\mathsf{uf}_j, r)$.*

Proof. Note that the effect of calling UNION for (p_j, q_j) is making $\mathsf{uf}_j[rP_j] = rQ_j$, where $rP_j = \text{FIND}(\mathsf{uf}_{j-1}, p_j)$ and $rQ_j = \text{FIND}(\mathsf{uf}_{j-1}, q_j)$. For all other elements $p \in X$, $\mathsf{uf}_j[p] = \mathsf{uf}_{j-1}[p]$. Hence:

1. The only element that had its parent change in uf_j (compared to uf_{j-1}) is rP_j, which was a canonical element in uf_{j-1}. Hence, all non-canonical elements p in uf_{j-1} have the same parent in uf_j, that is, $\perp \neq \mathsf{uf}_{j-1}[p] = \mathsf{uf}_j[p]$.
2. As explained in the previous item, $\mathsf{uf}_j[rP_j] = rQ_j \neq \perp$. Hence, rP_j is not canonical in uf_j.
3. For the element rQ_j, we have that $\mathsf{uf}_j[rQ_j] = \mathsf{uf}_{j-1}[rQ_j]$. Since rQ_j was canonical in uf_{j-1}, then $\mathsf{uf}_j[rQ_j] = \perp$, implying that rQ_j is a canonical element.

   Additionally, by definition, $rec(\mathsf{uf}_j, rQ_j) = \{rQ_j\} \cup desc(\mathsf{uf}_j, rQ_j)$. All previous descendants of rQ_j in uf_{j-1} are still descendants of rQ_j in uf_j, since all paths that existed in uf_{j-1} also exist in uf_j. Additionally, there was no path from rP_j to rQ_j in uf_{j-1} (because $rP_j \neq rQ_j$ and they were both roots of different trees), but there is one in uf_j. Hence, rP_j and its descendants of are now descendants of rQ_j as well, so $desc(\mathsf{uf}_j, rQ_j) \supseteq desc(\mathsf{uf}_{j-1}, rQ_j) \cup rec(\mathsf{uf}_{j-1}, rP_j)$.

   Finally, to prove that $desc(\mathsf{uf}_j, rQ_j) = desc(\mathsf{uf}_{j-1}, rQ_j) \cup rec(\mathsf{uf}_{j-1}, rP_j)$, suppose by contradiction that there is an element $p' \in desc(\mathsf{uf}_j, rQ_j)$ that is neither in $desc(\mathsf{uf}_{j-1}, rQ_j)$ nor

$desc(\mathsf{uf}_{j-1}, rQ_j)$. Then, there is path from p' to rQ_j in uf_j and, there are two possibilities: this path does not use the arc (rP_j, rQ_j) or it does.

Remember that all paths that do not use the arc (rP_j, rQ_j) exist in uf_{j-1} as well. Hence, if the path does not use (rP_j, rQ_j), then the same path from p' to rQ_j already existed in uf_{j-1}, implying $p' \in desc(\mathsf{uf}_{j-1}, rQ_j)$, which is a contradiction. If the path uses the arc (rP_j, rQ_j), by the same argument, the path from p' to rP_j exists in uf_{j-1}, implying $p' \in desc(\mathsf{uf}_{j-1}, rP_j)$, which is also a contradiction. Thus, we conclude that such p' does not exist and $desc(\mathsf{uf}_j, rQ_j) = desc(\mathsf{uf}_{j-1}, rQ_j) \cup rec(\mathsf{uf}_{j-1}, rP_j)$.

4. As explained before, the only canonical element of uf_{j-1} that is not canonical in uf_j is rP_j, for any canonical element $r \neq rP_j$, r is canonical in uf_j. Additionally, since $r \neq rP_j$ nor $r \neq rQ_j$, then $desc(\mathsf{uf}_j, r) = desc(\mathsf{uf}_{j-1}, r)$, since no arcs were added or removed in the subtree of r. Hence, if the descendants of r are the same, $rec(\mathsf{uf}_j, r) = rec(\mathsf{uf}_{j-1}, r)$.

□

With these two propositions proved, we can now prove the correctness of Alg. 4:

Proposition 3.3. *Let X a binary image, $\mathcal{A}$ a symmetric neighborhood and suppose Alg. 4 is called, where $\mathcal{A}_{alg}$ is obtained from an enumeration of $\mathcal{A}$ defined by the processing order of the neighboring pixels in Line 3. Finally, let uf_j be the array uf after calling* UNION *to the first j elements of $\mathcal{A}_{alg}$. Then, for any $0 \leq j \leq |\mathcal{A}|$, uf_j contains the $CC(X, \mathcal{A}^j_{alg})$.*

Proof. This proposition can be proved using induction in j.

The base case is $j = 0$. In this case, $\mathcal{A}^0_{alg} = \emptyset$, which means there are no pairs of neighboring pixels and uf is the array returned by the MAKESET procedure, where each pixel represent a disjoint set, which is correct by construction.

Now, suppose by induction hypothesis that, for any indices less than j, the statement in the proposition is correct. Let us prove that the proposition will still be true when UNION is called for $a_j = (p_j, q_j)$.

Let rP_j and rQ_j be the canonical elements of p_j and q_j in uf_{j-1}, respectively. There are two possibilities: either $rP_j = rQ_j$ or $rP_j \neq rQ_j$.

If $rP_j = rQ_j$, by induction hypothesis, p_j and q_j belong to the same CC of $CC(X, \mathcal{A}^{j-1}_{alg})$. Note that $\mathcal{A}^j_{alg} = \mathcal{A}^{j-1}_{alg} \cup \{p_j, q_j\}$, which means Prop. 2.6 can be used, implying that $CC(X, \mathcal{A}^j_{alg}) = CC(X, \mathcal{A}^{j-1}_{alg})$. On the other hand, since $rP_k = rQ_k$, then thanks to Prop. 3.1, $\mathsf{uf}_j = \mathsf{uf}_{j-1}$. Putting everything together, we have that uf_{j-1} stores $CC(X, \mathcal{A}^{j-1}_{alg})$, $\mathsf{uf}_j = \mathsf{uf}_{j-1}$ and $CC(X, \mathcal{A}^j_{alg}) = CC(X, \mathcal{A}^{j-1}_{alg})$. Hence, uf_j also stores $CC(X, \mathcal{A}^j_{alg})$.

If $rP_j \neq rQ_j$, then p_j and q_j are in different CCs of $CC(X, \mathcal{A}^{j-1}_{alg})$. In particular, Prop. 2.7 and Prop. 3.2 are valid. To prove the proposition, we need to show that any $\mathcal{A}^j_{alg}$-CC can be reconstructed by a canonical element of uf_j and, for each canonical element r, its reconstruction generates a valid $\mathcal{A}^j_{alg}$-CC. We will only prove the proposition one way, since the opposite way should follow the same ideas in reverse.

Initially, let $C \in CC(X, \mathcal{A}^j_{alg})$, where neither $rP_j \in C$ nor $rQ_j \in C$. Using Prop. 2.7, $C = CC(X, \mathcal{A}^j_{alg}) = CC(X, \mathcal{A}^{j-1}_{alg})$. By the induction hypothesis, there is a canonical element r representing C in uf_{j-1} such that $C = rec(\mathsf{uf}_{j-1}, r)$. Finally, thanks to Prop. 3.2, $rec(\mathsf{uf}_{j-1}, r) = rec(\mathsf{uf}_j, r)$, because r can not be rP_j or rQ_j. Then, $C = rec(\mathsf{uf}_j, r)$, and this proves the proposition for all components $C \in CC(X, \mathcal{A}^j_{alg})$ where neither rP_j nor rQ_j are in C.

A consequence of Prop. 2.7 is that there is only one $\mathcal{A}^j_{alg}$-CC that is not covered in the previous case: the component $C' = CC(X, \mathcal{A}^j_{alg}, rP_j) = CC(X, \mathcal{A}^j_{alg}, rQ_j) = CC(X, \mathcal{A}^{j-1}_{alg}, rP_j) \cup CC(X, \mathcal{A}^{j-1}_{alg}, rQ_j)$. Using Prop. 3.2, then $rec(\mathsf{uf}_j, rQ_j) = rec(\mathsf{uf}_{j-1}, rP_j) \cup rec(\mathsf{uf}_{j-1}, rQ_j)$ and, by inductive hypothesis, $rec(\mathsf{uf}_{j-1}, rP_j) = CC(X, \mathcal{A}^{j-1}_{alg}, rP_j)$ and $rec(\mathsf{uf}_{j-1}, rQ_j) = CC(X, \mathcal{A}^{j-1}_{alg}, rQ_j)$.

Hence, $rec(\mathsf{uf}_j, rQ_j) = C'$ and this finishes the proof, since it includes all possible cases of choices of $\mathcal{A}^j_{alg}$-CCs. □

Corollary 3.1. *For any binary image X and symmetric neighborhood $\mathcal{A}$, Alg. 4 builds a union-find uf that stores the $\mathcal{A}$-CCs of X.*

Proof. This can be easily proved by applying Prop. 3.3 for $j = |\mathcal{A}|$, since $\mathcal{A}^{|\mathcal{A}|}_{alg} = \mathcal{A}$. □

Thus, we conclude that, after finishing Alg. 4, uf stores the $\mathcal{A}$-CCs of X. In particular, this implies the following property:

Proposition 3.4. *Let X be a binary image, $\mathcal{A}$ a neighborhood and uf the union-find that stores $CC(X, \mathcal{A})$. Then, given any $p, q \in X$, p and q are $\mathcal{A}$-connected if and only if* $\text{FIND}(\mathsf{uf}, p) = \text{FIND}(\mathsf{uf}, q)$.

It is worth of note that the FIND procedure can be optimized by performing a technique known as path compression, which shortens the path between every visited element to its root.

More precisely, suppose $\text{FIND}(\mathsf{uf}, p)$ is called. Then, for every element p' on the path from p to its root rP, we set $\mathsf{uf}[p'] \leftarrow rP$. This shortens the path between p' to rP and, if the FIND procedure is called to any of these elements p' later, then this second call is faster because the path linking p' to rP only has one arc.

This idea is presented in Alg. 5.

Algorithm 5 FIND procedure with path compression.

```
procedure FINDCOMP($\mathsf{uf}, p$)
    $rP \leftarrow p$;
    while $\mathsf{uf}[rP] \neq \bot$ do
        $rP \leftarrow \mathsf{uf}[rP]$;
    $p' \leftarrow p$;
    while $\mathsf{uf}[p'] \neq rP$ do
        $pNext \leftarrow \mathsf{uf}[p']$;
        $\mathsf{uf}[p'] \leftarrow rP$;
        $p' \leftarrow pNext$;
    return $rP$;
```

There is also an optimized version of the UNION procedure that performs balanced merges of disjoint sets to ensure that the depth of each tree does not increase by more than 1. This optimization will be skipped in this text because it is not relevant to the proposed theory.

3.3 Labeling Connected Components in Gray-Level Images

Now, we consider the problem of finding an efficient way of storing CCs of gray-level images. This problem has been object of many studies in the field, such as [NC06, BGL$^+$]. There are different approaches to tackle this problem. In this text, we explain the approach based on the union-find structure, that uses a single array that can be used to efficiently store and reconstruct CCs of gray-level images.

However, understanding the properties of this array and why the algorithms proposed in these papers work is crucial for developing the proposed method. So, in this section, instead of showing the solution proposed in [NC06] and [BGL$^+$] and explaining why it works, we use a proof by construction approach, showing what properties are important and how their algorithms are built to satisfy these properties.

First off, given a gray-level image f and a symmetric neighborhood $\mathcal{A}$, a natural way of storing the $\mathcal{A}$-CCs of f would be to compute the union find uf^λ of every level set $X_\lambda(f)$, $\lambda \in \mathbb{K}$. In this sense, let $\mathcal{G} = (D_f, \mathcal{A})$. Then, each uf^λ contains the CCs of the induced subgraph $\mathcal{G}[X_\lambda(f)]$. For simplicity, we denote this subgraph as $\mathcal{G}_\lambda = (V_\lambda = X_\lambda(f), \mathcal{E}_\lambda)$.

Instead of computing each of these union-finds individually, we recall that Prop. 2.4 states that the set of $\mathcal{A}$-CCs of f is decreasing. This implies that, for any pair of pixels (p, q) where p and q are $\mathcal{A}$-connected in $X_\lambda(f)$ they are also $\mathcal{A}$-connected in $X_{\lambda-1}(f)$, for $0 < \lambda < K$. Hence, if uf^λ was already computed, there is no need to compute $\mathsf{uf}^{\lambda-1}$ from scratch, since all connectivity relations existing in uf^λ are also valid in $\mathsf{uf}^{\lambda-1}$. Thus, to compute $\mathsf{uf}^{\lambda-1}$ from uf^λ, we can call the UNION procedure only to the pairs of neighbors $(p', q') \in \big(\mathcal{E}_{\lambda-1} \setminus \mathcal{E}_\lambda\big)$, or more specifically, the pairs of neighboring pixels that exist in $\mathcal{G}_{\lambda-1}$ that do not exist in $\mathcal{G}_\lambda$.

In particular, this set of arcs $(p, q) \in (\mathcal{E}_{\lambda-1} \setminus \mathcal{E}_\lambda)$ can be characterized as follows:

Proposition 3.5. *For any $\{p, q\} \in (\mathcal{E}_{\lambda-1} \setminus \mathcal{E}_\lambda)$, at least one of the following is true:*

1. $f(p) = \lambda - 1$ *and* $f(q) \geq \lambda - 1$*;*

2. $f(p) \geq \lambda - 1$ *and* $f(q) = \lambda - 1$.

Proof. By definition, $V_\lambda = X_\lambda(f) \subseteq X_{\lambda-1}(f) = V_{\lambda-1}$. Hence, any $p \in V_{\lambda-1}$ satisfies $f(p) \geq \lambda - 1$. Then, for any $\{p, q\} \in \mathcal{E}_{\lambda-1}$ with $f(p) \neq \lambda - 1$ and $f(q) \neq \lambda - 1$, $(p, q) \in \mathcal{E}_\lambda$ and, for any $(p, q) \in (\mathcal{E}_{\lambda-1} \setminus \mathcal{E}_\lambda)$, at least one of them has gray-level $\lambda - 1$, that is, $f(p) = \lambda - 1$ or $f(q) = \lambda - 1$. □

Combining these ideas, a first idea to compute and store the union-finds of f and $\mathcal{A}$ are presented in Alg. 6.

Algorithm 6 Naive way of obtaining $CC(f, \mathcal{A})$.

```
procedure BUILDUFS($f, \mathcal{A}$)
    for $K > \lambda \geq 0$ do
        if $\lambda = K - 1$ then
            $\mathsf{uf}^\lambda \leftarrow$ MAKESET($D_f$);
        else
            $\mathsf{uf}^\lambda \leftarrow$ copy of $\mathsf{uf}^{\lambda+1}$;
        for $(p, q) \in \mathcal{A}$ where $\big(f(p) = \lambda$ and $f(q) \geq \lambda\big)$ or $\big(f(q) = \lambda$ and $f(p) \geq \lambda\big)$ do
            $\mathsf{uf}^\lambda \leftarrow$ UNION($\mathsf{uf}^\lambda, p, q$);
```

It should be simple to prove that Alg. 6 works, but it is not very efficient in terms of memory usage, since it has to store K union-finds. Additionally, the resulting union-finds can vary vastly according to how the UNION procedure link the nodes. Figure 3.2 shows two possible results when using this strategy for the same image, but varying the order in which neighboring pixels are processed.

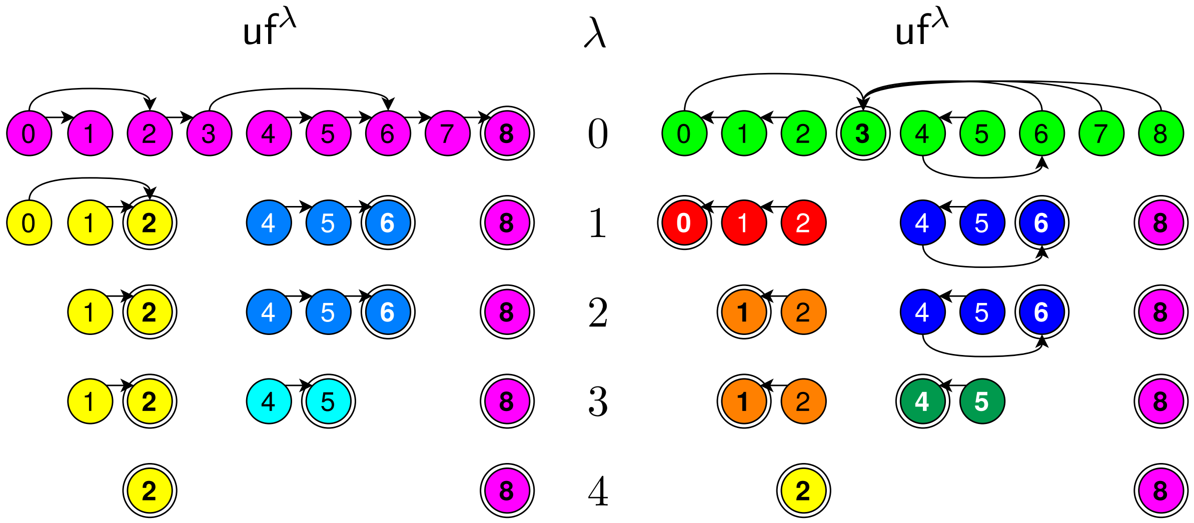


Figure 3.2: *Two possible outputs of Alg. 6, depending on how the* UNION *procedure is implemented. Colors are defined according to the root of each tree. It is easy to notice that, in the implementation on the left, different CCs can have the same root, while in the implementation on the right this does not happen.*

To optimize memory usage, we observe the following: by definition, for any $\lambda \in \mathbb{K}$, if r is canonical in uf^λ, then $C = rec(\mathsf{uf}^\lambda, r)$ is a $\mathcal{A}$-CC of $X_\lambda(f)$. If the sub-tree of r is always the same for any $0 \leq \alpha < \lambda$, then $rec(\mathsf{uf}^\alpha, r) = rec(\mathsf{uf}^\lambda, r) = C$ and, in particular, $rec(\mathsf{uf}^0, r) = C$. This means that, under these conditions, uf^0 can reconstruct any CC from any uf^λ, $\lambda \in \mathbb{K}$, as long as we know which elements were canonical at some point.

To enforce this condition, we observe that, in the original UNION procedure (Alg. 3), whether we performed $\mathsf{uf}[rP] \leftarrow rQ$ or $\mathsf{uf}[rQ] \leftarrow rP$ at Line 5 was irrelevant. However, in this case, since the roots of the trees have different gray-levels, we can modify the UNION procedure to always enforce $\mathsf{uf}[rP] = rQ$ if $f(rP) \geq f(rQ)$ or $\mathsf{uf}[rQ] = rP$, otherwise. This idea is presented in Alg. 7:

Algorithm 7 UNION procedure which forces a gray-level order between linked elements.

```
procedure UNIONORDERED($f$, uf, $p$, $q$)
    $rP \leftarrow$ FIND(uf, $p$);
    $rQ \leftarrow$ FIND(uf, $q$);
    if $rP \neq rQ$ then
        if $f(rP) \geq f(rQ)$ then
            uf[$rP$] $\leftarrow rQ$;
        else
            uf[$rQ$] $\leftarrow rP$;
    return uf;
```

If we do so, given a canonical element r from uf^λ, we guarantee that $rec(\mathsf{uf}^\lambda, r) = rec(\mathsf{uf}^{\lambda-1}, r)$. The formal proof of this idea is explained in more details later, but the basic idea is based on the fact that all arcs added to $\mathsf{uf}^{\lambda-1}$ point towards an element with gray-level $\lambda - 1$. Since all elements $p \in rec(\mathsf{uf}^\lambda, r)$ satisfy $f(p) \geq \lambda$, none of them can have a new child node and we conclude that $rec(\mathsf{uf}^\lambda, r) = rec(\mathsf{uf}^{\lambda-1}, r)$.

When using Alg. 7 to compute the union-finds, we obtain the results shown in Fig. 3.3 (left). Because all CCs can be recovered from uf^0, we only need to store uf^0, or to put it another way, only a single array is needed to store all CCs of a gray-level image. In general, instead of uf^0, this array is usually called parent in the literature. An example is shown in Fig. 3.3 (right).

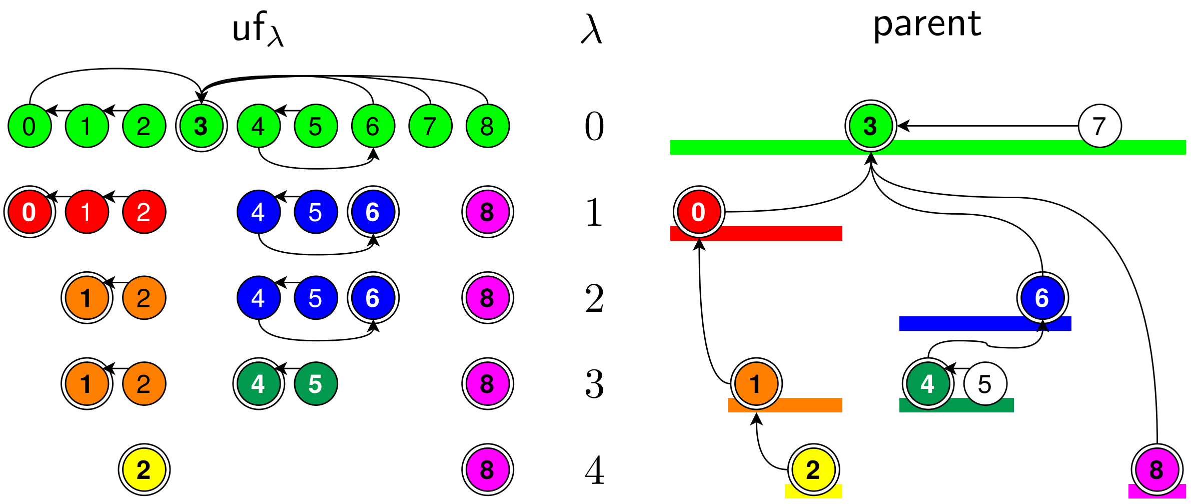


Figure 3.3: *Left: the union-finds obtained using Alg. 6 but forcing unions according to Alg. 7. Right: representation of the* parent *array, with nodes reallocated according to their gray-levels. Nodes with double circles represent pixels that were canonical elements in at least one* uf$^\lambda$, $0 \leq \lambda < K$ *and can be used to reconstruct CCs.*

In order to prove correctness of the algorithms, we need to analyze some properties of the parent arrays. First of all, by construction, it is plain that the hierarchy of the union-finds are ordered according to the gray-levels of the elements.

Proposition 3.6. *For any $\lambda \in \mathbb{K}$, if uf$^\lambda[p] = q$, then $f(p) \geq f(q)$.*

Additionally, Alg. 7 can only change a union-find by connecting a root of a tree to another root, implying that, if the input is a forest, then the output is also a forest. Since the initial union-find (built calling MAKESET) is a forest, then:

Proposition 3.7. *For any $\lambda \in \mathbb{K}$, uf$^\lambda$ is a forest.*

In fact, since every root is pointing towards an artificial element $\perp$, any uf$^\lambda$ can be thought as a tree, where $\perp$ is an artificial root that is the parent of all canonical elements of uf$^\lambda$.

For any $p \in D_f$, uf$^\lambda[p]$ can only be modified when p is a canonical element. By definition, once uf$^\lambda[p] \neq \perp$, then p can not become a canonical element anymore. Hence:

Proposition 3.8. *For any $0 < \alpha < \lambda < K$, given a pixel $p \in \mathcal{D}_f$, if uf$^\lambda[p] \neq \perp$, then uf$^\lambda[p] =$ uf$^\alpha[p]$.*

Combining the previous proposition with Prop. 3.6, one can prove that:

Proposition 3.9. *Let $\lambda \in \mathbb{K}$ and r be a canonical element of uf$^\lambda$. Then, for any $0 \leq \alpha < \lambda < K$, rec(uf$^\lambda$, r) = rec(uf$^\alpha$, r).*

Note that, under the conditions of Prop. 3.9, if r is canonical in uf$^\lambda$, then $rec($uf$^\lambda, r)$ is a $\mathcal{A}$-CC of $X_\lambda(f)$, and this CC can still be obtained from uf$^\alpha$. The only caveat is that r may not be canonical in uf$^\alpha$, so this CC can only be obtained if it is possible to know that r was a canonical element at some point.

To tackle this problem, we note that, if r was canonical, then there exists a gray-level $\lambda \in \mathbb{K}$ such that uf$^\lambda[r] = \perp$. In particular, let $\beta \in \mathbb{K}$ be the lowest gray-level where uf$^\beta[r] = \perp$. If $\beta = 0$, then r is canonical in uf^0 and it is easy to know that r is a canonical element. If $\beta > 0$, then r is not canonical in uf^0 but uf$^0 =$ uf$^{\beta-1}[r] = q \neq \perp$. By Prop. 3.6, $f(q) \geq f(r)$, but since uf$^\beta[r] = \perp$, then $f(q) \neq f(r)$. Then:

Proposition 3.10. *Let $0 < \beta < K$ be the lowest gray-level where the pixel r is a canonical element of uf$^\lambda$. Then, $f($uf$^0[r]) > f(r)$.*

Using a similar argument, for non-canonical elements, we have that:

Proposition 3.11. *Let $q \in D_f$ be a pixel that is not a canonical element in any uf^{λ}, $\lambda \in \mathbb{K}$. Then, $f(\mathit{uf}^0[q]) = f(q)$.*

Combining the two previous propositions, we have that:

Proposition 3.12. *An element $r \in X$ is a canonical element (namely, there exists $\lambda \in \mathbb{K}$ where r is canonical in uf^{λ}) if and only if $f(r) > f(\mathit{uf}^0[r])$.*

Finally, this means that all canonical elements of all uf^{λ} ($\lambda \in \mathbb{K}$) can be obtained from uf^0 by comparing the gray-levels of the pixels. Additionally, for any of these canonical elements $r \in D_f$, $rec(\mathsf{uf}^{f(r)}, r)$ represents a $\mathcal{A}$-CC of f, and since $rec(\mathsf{uf}^{f(r)}, r) = rec(\mathsf{uf}^0, r)$, all of these CCs can also be obtained from uf^0. In other words, we can conclude that:

Proposition 3.13. *All $\mathcal{A}$-CCs of f are stored in $\mathit{parent} = \mathit{uf}^0$.*

From this point onward, the array uf^0 will be called parent.

3.4 Canonical Elements and Representatives

For union-finds of binary images, there was a clear mapping between connected pixels in binary images and having the same canonical elements in union-finds (Prop. 3.4). Following the properties of the parent array that we saw in Sec. 3.2, to define a similar concept for gray-level images and the parent array, we say that $r \in D_f$ is a canonical element of parent if $f(r) > f(\mathsf{parent}[r])$ or $\mathsf{parent}[r] = \bot$, that is, r is the root node. Thanks to Prop. 3.10, this definition is equivalent of saying that r is canonical in parent if and only if there is a $\lambda \in \mathbb{K}$ such that r is canonical in uf^{λ}.

Since parent represents multiple union-finds, it is possible that a pixel $p \in D_f$ have different roots depending on which union-find uf^{λ} is taken into consideration. Hence, we define the representative of p at level λ as the canonical element r returned by $\textsc{find}(\mathsf{uf}^{\lambda}, p)$. Thanks to Props. 3.10 and 3.11, this element can be found in the parent array using Alg 8.

Algorithm 8 Returns the representative of p in parent with level λ.

```
procedure FINDREP($f, \mathsf{parent}, p, \lambda$)
    while $\mathsf{parent}[p] \neq \bot$ and $f(\mathsf{parent}[p]) \geq \lambda$ do
        $p \leftarrow \mathsf{parent}[p]$;
    return $p$;
```

In other words, Alg. 8 returns the last element of the path $\pi(\mathsf{parent}, p, \bot)$ (where $\bot$ represents an artificial root of parent that is an ancestor of all nodes) that has gray-level higher than or equal to λ. Representatives can be used to allocate nodes of component-trees, thanks to the following property:

Proposition 3.14. *Let f be a gray-level image, $\mathcal{A}$ be a symmetric neighborhood and parent be an array storing $CC(f, \mathcal{A})$. Then, for any $\lambda \in \mathbb{K}$ and $p \in X_\lambda(f)$:*

$$CC(X_\lambda(f), \mathcal{A}, p) = rec(\mathit{parent}, \textsc{findRep}(f, \mathit{parent}, p, \lambda)) \tag{3.5}$$

Using the concept of representatives, we say that two pixels $p, q \in D_f$ are comparable in parent if and only if $\textsc{findRep}(\mathsf{parent}, f, p, \lambda) = \textsc{findRep}(\mathsf{parent}, f, q, \lambda)$ for all $0 \leq \lambda \leq \min\{f(p), f(q)\}$.

Since representatives can be thought as elements of paths, testing if two pixels $p \in D_f$ and $q \in D_f$ are comparable is equivalent to test if all representatives (of p and q) are common ancestors of both p and q. It is easy to see that, once a representative r is a common ancestor of p and q, all representatives that are ancestors of r are also common ancestors of p and q. Hence, let $\beta \in \mathbb{K}$ be the highest gray-level where both p and q are foreground pixels in $X_\beta(f)$, namely, $\beta = \min\{f(p), f(q)\}$. Then:

Proposition 3.15. *Let f be a gray-level image, $\mathcal{A}$ a symmetric neighborhood and* parent *an array representing a forest. Suppose two pixels $p, q \in D_f$ are given and let $\beta = \min\{f(p), f(q)\}$. Then, p and q are comparable in* parent *if and only if* $\textsc{findRep}(f, \textsf{parent}, p, \beta) = \textsc{findRep}(f, \textsf{parent}, q, \beta)$.

Using this property, one can prove that:

Proposition 3.16. *Let f be a gray-level image, $\mathcal{A}$ a symmetric neighborhood and* parent *be an array storing $CC(f, \mathcal{A})$. Then, p and q are connected in f if and only if p and q are comparable in* parent.

A consequence of parent containing all the information about connectedness of pixels is that, by construction, parent already contains all inclusion relations between the CCs, that is to say, given a gray-level image f and a neighborhood $\mathcal{A}$, all arcs of the component-tree of f and $\mathcal{A}$ can be obtained from the parent array.

More precisely, the following proposition is valid:

Proposition 3.17. *Let f be a gray-level image, $\mathcal{A}$ a symmetric neighborhood relation and $CT(f, \mathcal{A}) = (V_{CT}, E_{CT})$ be the component-tree of f using $\mathcal{A}$. Then, for any arc $e = (C, C') \in \mathcal{E}_{CT}$, there is an arc $(r, \textsf{parent}[r])$ where $C = rec(\textsf{parent}, r)$, $C' = rec(\textsf{parent}, r')$ and*
$r' = \textsc{findRep}(f, \textsf{parent}, \textsf{parent}[r], f(\textsf{parent}[r]))$.

Proof. If $e = (C, C')$ is an arc of the component-tree, that means that $C \subset C'$ and there is no other $\mathcal{A}'$-CC between them. We proved that all $\mathcal{A}$-CCs are stored in parent, so there are two distinct canonical elements in parent that can reconstruct C and C' which, by the assumptions of this proposition, are r and r', respectively.

To prove the proposition, we first show that r' in an ancestor of r in parent. To do that, suppose by contradiction that r' is not an ancestor. Then, either r' is a descendant of r or they are unrelated.

If r' was a descendant of r, since $rec(\textsf{parent}, r) = \{r\} \cup desc(\textsf{parent}, r)$, then $r' \cup desc(\textsf{parent}, r') = rec(\textsf{parent}, r') \subset desc(\textsf{parent}, r)$, in other words, $rec(\textsf{parent}, r) = C \supset C' = rec(\textsf{parent}, r')$, which is a contradiction.

If they are not related, that is to say, neither r is descendant of r' nor r' is a descendant of r, then $r \notin desc(r')$. This means $r \notin rec(\textsf{parent}, r') = C'$, which implies that $C \not\subset C'$, and that is also a contradiction.

Hence, r' must be an ancestor of r in parent. If $r' = \textsf{parent}[r]$ or r' is the representative of $\textsf{parent}[r]$, there is nothing to prove, since it already satisfies the conditions of the proposition. Then, suppose by contradiction that r' is not the representative of $\textsf{parent}[r]$. This implies that, in the path from r to r' in parent, there must exist at least one canonical element $r'' \neq r'$.

However, this would imply that $rec(\textsf{parent}, r'')$ represents a valid $\mathcal{A}$-CC $C'' \in V_{CT}$. Since $r \in desc(r'')$ and $r'' \in desc(r')$, then we would have $rec(\textsf{parent}, r) \subset rec(\textsf{parent}, r'') \subset rec(\textsf{parent}, r')$, or $C \subset C'' \subset C'$, which implies that $(C, C') \notin \mathcal{E}_{CT}$, which is a contradiction. Hence, such element r'' can not exist and $r' = \textsc{findRep}(f, \textsf{parent}, \textsf{parent}[r], f(\textsf{parent}[r]))$, as desired. □

Using a similar approach, it is also possible to prove that all such arcs in the parent array are present in the component-tree. In this sense, this means that there is an equivalence in terms of information retained by a component-tree and its parent array.

3.5 Ordered parent Construction

We now describe how the ideas explained in this section are adapted and optimized to obtain the union-find based algorithm for efficient component-tree construction used in the literature.

For that, given a neighborhood $\mathcal{A}$ and two pairs of neighboring pixels $a = (p, q), a' = (p', q') \in \mathcal{A}$, we define the $\prec$ relation as follows:

$$a = (p, q) \prec a' = (p', q') \Rightarrow \min\{f(p), f(q)\} \geq \min\{f(p'), f(q')\} \tag{3.6}$$

Given a gray-level image f and a neighborhood $\mathcal{A}$, let $(\mathcal{A})^{alg} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$ be a sequence obtained from an enumeration of $\mathcal{A}$, where the elements of $(\mathcal{A})^{alg}$ satisfy the following relation:

$$a_\ell = (p_\ell, q_\ell) \prec a_{\ell+1}, 1 \leq \ell < |\mathcal{A}| \tag{3.7}$$

By doing so, if $\mathcal{G} = (f, \mathcal{A})$, we guarantee the arcs of $\mathcal{G}[X_\lambda(f)]$ are processed before the arcs of $\mathcal{G}[X_{\lambda-1}(f)]$ for any $0 < \lambda < K$. Then, max-tree construction can be implemented using Alg. 9.

Algorithm 9 Computing the $\mathcal{A}$-CCs of f using a max-tree.

```
procedure BUILDPARENT($f, \mathcal{A}$)
    parent ← MAKESET($D_f$);
    $(\mathcal{A})_{alg}$ ← $\mathcal{A}$ ordered according to the $\prec$ relation;
    for $(p, q) \in (\mathcal{A})_{alg}$ do
        parent ← UNIONORDERED(f, parent, $p, q$);
```

Another optimization that can be performed is to add path compression using an additional union-find zpar. This array zpar consists simply of the parent array with the paths compressed. The idea is to use zpar to find the representatives of p and q inside the calls of the UNIONORDERED procedure quickly, but not compressing the paths of parent to keep Prop. 3.8 valid. This idea is presented in Algs. 10 and 11.

Algorithm 10 Ordered max-tree construction with path compression.

```
procedure BUILDPARENTZPAR($f, \mathcal{A}$)
    parent ← MAKESET($D_f$);
    zpar ← MAKESET($D_f$);
    $(\mathcal{A})^{alg}$ ← $\mathcal{A}$ ordered according to the $\prec$ relation;
    for $(p, q) \in (\mathcal{A})^{alg}$ do
        (parent, zpar) ← UNIONZPAR(parent, zpar, $p, q$);
```

Algorithm 11 UNION procedure with path compression for the parent array.

```
procedure UNIONZPAR(parent, zpar, $p, q$)
    $rP$ ← FINDCOMP(zpar, $p$);                    ▷ Path compression applied only in zpar
    $rQ$ ← FINDCOMP(zpar, $q$);
    if $rP \neq rQ$ then
        if $f(rP) \geq f(rQ)$ then
            parent[$rP$] ← $rQ$;
            zpar[$rP$] ← $rQ$;
        else
            parent[$rQ$] ← $rP$;
            zpar[$rQ$] ← $rP$;
    return (parent, zpar);
```

This implementation that uses path compression seems to be the most common implementation for merge-based component-tree construction. In fact, it can be shown that this strategy is very efficient and it can be performed in quasi-linear time, assuming the ordering step can be performed in linear time.

3.6 Unordered parent Construction

As stated above, the union-find based strategy that uses path compression is a very efficient way of computing component-trees when ordering the arcs of $\mathcal{A}$ according to Eq. (3.7) is possible.

However, when ordering is not possible, a different approach is required. In the literature, the unordered strategy has been used mainly for parallel construction of component-trees [OW07b], where partitions of the domain of image are processed separately, and a final step is used to join the component-trees of the partitions.

Similar to the last section, instead of presenting the algorithm, we first show the properties that we want the algorithm to have and then build the algorithm to satisfy these properties.

To begin with, we start comparing differences that may arise from an ordered and a non-ordered implementation. A good property of an ordered algorithm (e.g., Alg. 9) is that each pixel has its parent changed only once (from $\perp$ to a pixel), and the ordering guarantees that this change is always performed to the correct parent node. However, this property can not be guaranteed in an unordered construction. An example is given in Fig 3.4.

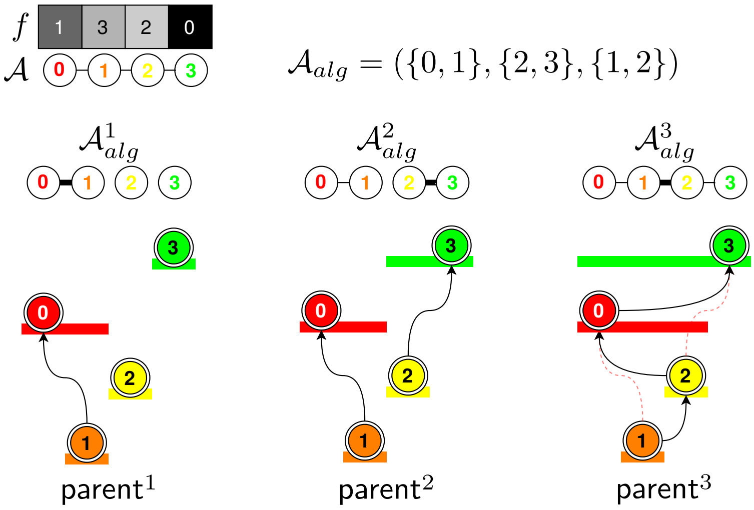


Figure 3.4: *Unordered* parent *construction, where the sequence $\mathcal{A}_{alg}$ is not ordered. In the figure,* parent^j *stores $\mathcal{A}^j_{alg}$-CCs ($1 \leq j \leq 3$). Note that, to update* parent^2 *to* parent^3*, the parents of the pixels 1 and 2 needs to be reassigned.*

As it can be seen in the figure, in a non-ordered strategy, the parent of a node may change multiple times to keep the comparability relations consistent.

3.7 Parent Change Properties

In this section, we analyze how to develop an unordered algorithm for component-tree construction that perform changes in the parent array consistently based on the processed pairs of neighbors.

For that, we first analyze how the connected components change when a new pair of neighboring pixels is processed. Suppose a gray-level image f and a symmetric neighborhood $\mathcal{A}$ are given and we want an algorithm to build parent, the array that stores $CC(f, \mathcal{A})$, without relying in an ordering of the elements of $\mathcal{A}$. Thus, let $\mathcal{A}_{alg} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$ be obtained from an enumeration of $\mathcal{A}$ and suppose $\mathcal{A}_{alg}$ defines the order that our algorithm will use to process the pairs of pixels. From $\mathcal{A}_{alg}$, we define $\mathcal{A}^j_{alg}$ as the set of the j first neighbors of $\mathcal{A}_{alg}$, or to put it another way, $= \bigcup\limits_{\ell=1}^{j} a_\ell$. Finally, suppose that the first $j-1$ pairs were processed, resulting in an array parent^{j-1} that stores $CC(f, \mathcal{A}^{j-1}_{alg})$, and that we want to update parent^{j-1} to parent^j, where parent^j stores $CC(f, \mathcal{A}^j_{alg})$. An example is shown in Fig. 3.5.

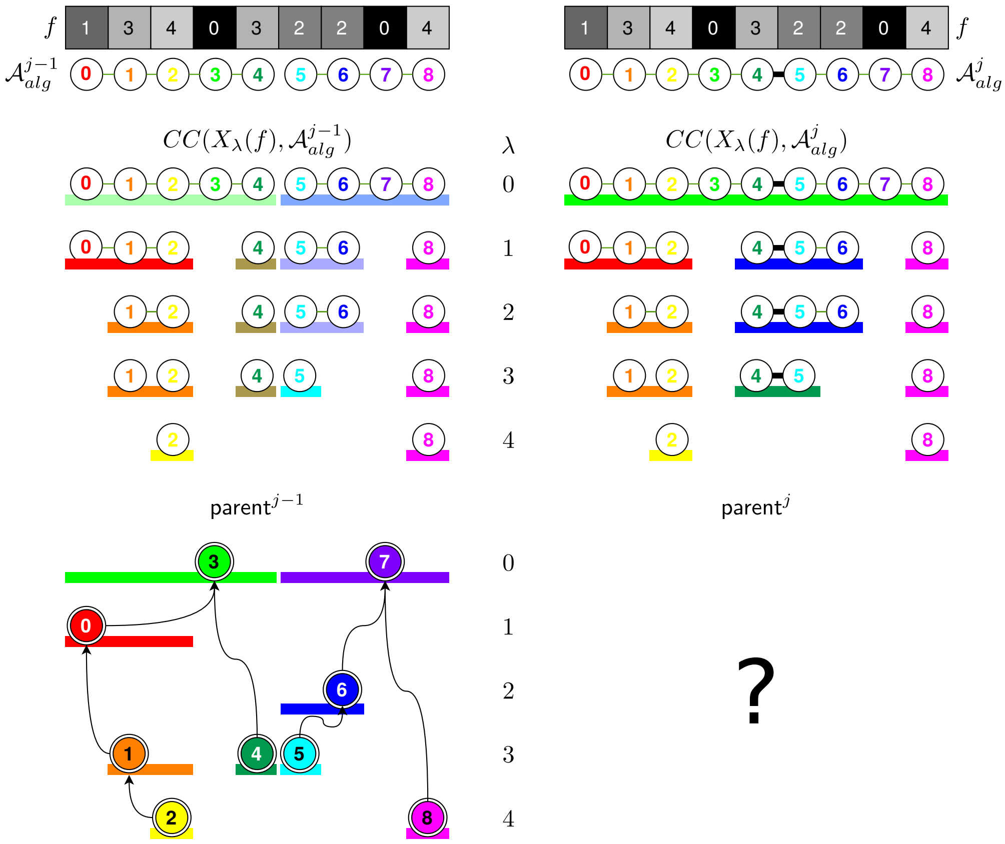

f
1 3 4 0 3 2 2 0 4
$\mathcal{A}^{j-1}_{alg}$
$\mathcal{A}^{j}_{alg}$
$CC(X_\lambda(f), \mathcal{A}^{j-1}_{alg})$
λ
$CC(X_\lambda(f), \mathcal{A}^{j}_{alg})$
0
1
2
3
4
parent^{j-1}
parent^{j}
?

Figure 3.5: *Assumptions used for this section: a gray-level f and two neighborhoods $\mathcal{A}^{j-1}_{alg}$, $\mathcal{A}^{j}_{alg}$ are given, where $\mathcal{A}^{j}_{alg} = \mathcal{A}^{j-1}_{alg} \cup a_j$ and $a_j = \{p_j = 5, q_j = 4\}$. The array* parent^{j-1} *that stores $CC(f, \mathcal{A}^{j-1}_{alg})$ is given and we want to compute* parent^{j}*, an array updated from* parent^{j-1}*, that stores $CC(f, \mathcal{A}^{j}_{alg})$.*

We recall from Sec 2.6 that, comparing $CC(f, \mathcal{A}^{j-1}_{alg})$ to $CC(f, \mathcal{A}^{j}_{alg})$, the only differences are in the CCs that were extracted from gray-levels λ where p_j and q_j are not $\mathcal{A}^{j-1}_{alg}$-connected.

Thus, let β be a gray-level where $p_j, q_j \in X_\beta(f)$ but are not $\mathcal{A}^{j-1}_{alg}$-connected. Then, from Prop 2.7, we saw that the ($\mathcal{A}^{j-1}_{alg}$-) connected components $C^\beta_p = CC(X_\beta(f), \mathcal{A}^{j-1}_{alg}, p_j)$ and $C^\beta_q = CC(X_\beta(f), \mathcal{A}^{j-1}_{alg}, q_j)$ become a single ($\mathcal{A}^{j}_{alg}$-) connected component $C^\beta = C^\beta_p \cup C^\beta_q$.

Since we want parent^j to store the $\mathcal{A}^{j}_{alg}$-CCs of f, we need a canonical element r^β in parent^j that reconstructs C^β for every β where p_j and q_j are originally disconnected in $X_\beta(f)$. As explained before, the parent array can be seen as multiple union-finds, one for each gray-level. In this sense, let uf$^\lambda_{j-1}$ denote the union-find of parent^{j-1} at level λ induced by parent^{j-1}, that is, $\mathcal{G}($uf$^\lambda_{j-1})$ is the subgraph of $\mathcal{G}($parent$^{j-1})$ induced by $X_\lambda(f)$. Since parent^{j-1} stores $CC(f, \mathcal{A}^{j-1}_{alg})$, then uf$^\lambda_{j-1}$ stores $CC(X_\lambda(f), \mathcal{A}^{j-1}_{alg})$. An example is shown in Fig. 3.6.

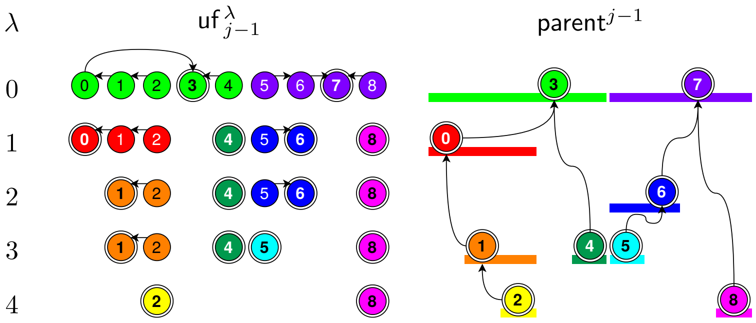


Figure 3.6: *Representing the array* parent^{j-1} *from Fig. 3.5 as multiple union-finds* uf$^\lambda_{j-1}$.

Looking at each union-find $\mathsf{uf}^{\lambda}_{j-1}$ individually, each of them stores the $\mathcal{A}^{j-1}_{alg}$-CCs of f. Hence, analyzing Prop. 2.7 in this context, what we want are updated union-finds $\mathsf{uf}^{\lambda}_{j}$ where, for each $\mathsf{uf}^{\lambda}_{j}$, the canonical elements r of $\mathsf{uf}^{\lambda}_{j}$ satisfy:

$$rec(\mathsf{uf}^{\lambda}_{j}, r) = \begin{cases} rec(\mathsf{uf}^{\lambda}_{j-1}, r) & \text{if } p_j, q_j \notin rec(\mathsf{uf}^{\lambda}_{j-1}, r) \\ rec(\mathsf{uf}^{\lambda}_{j-1}, rP_\lambda) \cup rec(\mathsf{uf}^{\lambda}_{j-1}, rQ_\lambda) & \text{otherwise,} \end{cases} \tag{3.8}$$

where rP_λ and rQ_λ are the canonical elements of p_j and q_j in $\mathsf{uf}^{\lambda}_{j-1}$, respectively.

Naturally, we still also want all these updated union-finds to be represented by a single array parent^j. Otherwise, one could simply call the UNION procedure for each union-find $\mathsf{uf}^{\lambda}_{j-1}$ individually, but this strategy does not guarantee that Prop. 3.8 would still hold. The main problem is that, in order to keep the union-finds representable by a single array, a change in an arc of $\mathsf{uf}^{\beta}_{j-1}$ must be reflected in $\mathsf{uf}^{\lambda}_{j-1}$, for all $0 \leq \lambda < \beta$.

For instance, consider the example of Fig. 3.5 and suppose $\beta = 3$ and that $rP_\beta = 5$ and $rQ_\beta = 4$ are the canonical elements of $p_j = 5$ and $q_j = 4$ in $\mathsf{uf}^{\beta}_{j-1}$ respectively. To connect p_j to q_j at level β, we add the arc (rP_β, rQ_β) to $\mathsf{uf}^{\beta}_{j-1}$, generating an updated union-find $\mathsf{uf}^{(\beta,1)}_{j-1}$. Then, for all $0 \leq \lambda < \beta$, the arc (rP_β, rQ_β) is also added, generating updated union-finds $\mathsf{uf}^{(\lambda,1)}_{j-1}$, where the previous parent of rP_j is overwritten when $\mathsf{uf}^{\lambda}_{j-1}[rP_j] \neq \perp$. As a consequence, all these updated union-finds can be represented by a single array $\mathsf{parent}^{(j-1,1)} = \mathsf{uf}^{(0,1)}_{j-1}$. An illustration of this process in shown in Fig. 3.7.

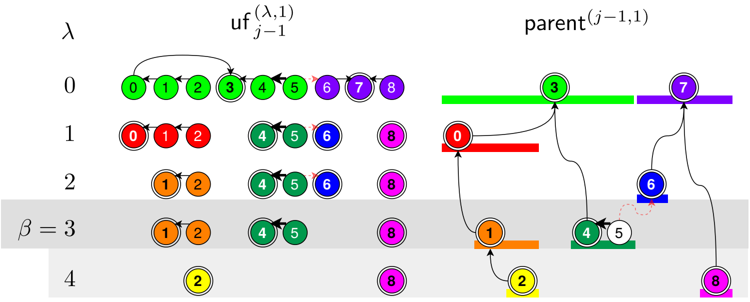


Figure 3.7: *Update from $\mathsf{uf}^{\lambda}_{j-1}$ to $\mathsf{uf}^{(\lambda,1)}_{j-1}$ for $a_j = \{p_j = 5, q_j = 4\}$. Note that, for $\beta = 3$, $rP_\beta = 5$ and $rQ_\beta = 4$ were originally disconnected, so the arc $(rP_\beta = 5, rQ_\beta = 4)$ is added to connect p_j and q_j at $\mathsf{uf}^{\beta}_{j-1}$. Then, this change is propagated to the other union-finds where rP_β is a foreground pixel to keep the* parent *array consistent.*

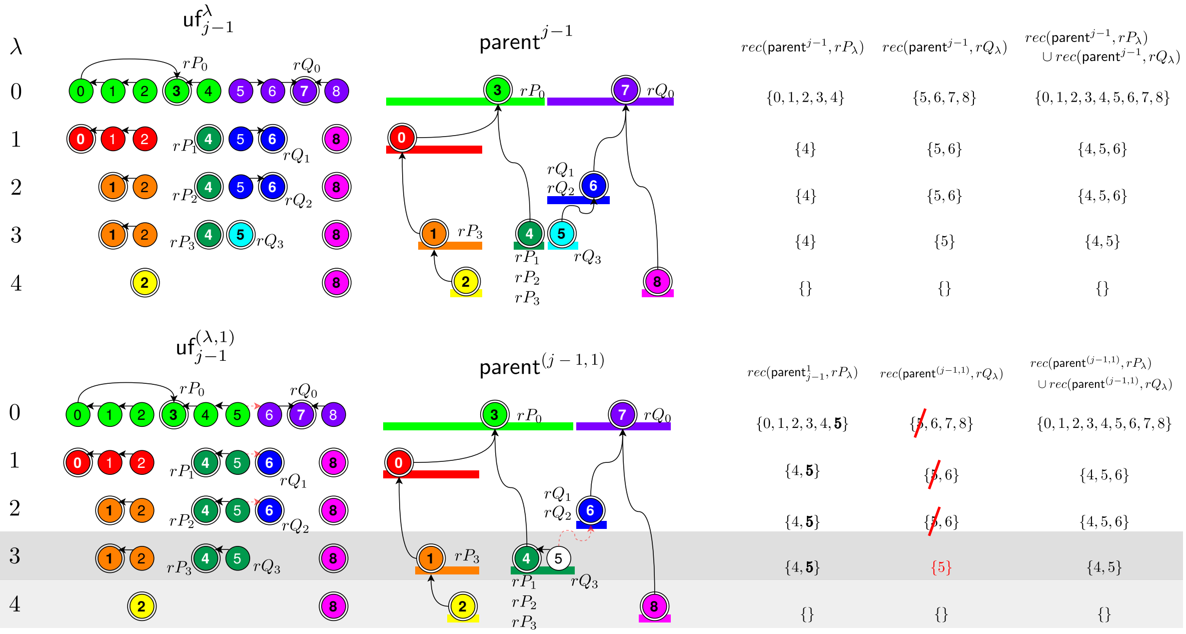
From Fig. 3.7, some properties can be observed. First of all, for any λ satisfying $\beta < \lambda < K$, $\mathsf{uf}^{(\lambda,1)}_{j-1} = \mathsf{uf}^{\lambda}_{j-1}$. This happens because, in the example, β indicates the highest gray-level where p_j is a foreground pixel, and for any $\lambda > \beta$, this arc can not be assigned because rP_λ does not exist.

As a consequence, for every gray-level where p_j is a foreground pixel, then p_j is now connected to q_j, as desired. However, a side effect is that, for every gray-level λ where $\mathsf{uf}^{\lambda}_{j-1}[rP_\lambda] \neq \perp$, we may have overwritten its previous parent p', and now this connectivity relation between rP_λ and p' is lost. For example, in Fig. 3.7, this happens between the pixels $rP_\lambda = 5$ and $p' = 6$, for all gray-levels $0 \leq \lambda \leq 2$.

On the bright side, since rP_β is a canonical element of $\mathsf{uf}^{\beta}_{j-1}$, then $\mathsf{uf}^{\beta}_{j-1}[rP_\beta] = \perp$, and no connectivity relations were lost at level β. Hence, for the gray-level β, all connectivity relations of $\mathsf{uf}^{\beta}_{j-1}$ were preserved and p_j and q_j become connected. Therefore, $\mathsf{uf}^{(\beta,1)}_{j-1}$ now stores the $\mathcal{A}^{j}_{alg}$-CCs of $X_\beta(f)$, as desired.

On the other hand, the gray-levels $0 \leq \lambda < \beta$ need to be fixed. We recall that we want, for each $\mathsf{uf}^{(\lambda,1)}_{j-1}$, to update them and have a canonical element r representing $rec(\mathsf{uf}^{\lambda}_{j-1}, rP_\lambda) \cup rec(\mathsf{uf}^{\lambda}_{j-1}, rQ_\lambda)$. It is important to emphasize that, from Eq. (3.8), rP_λ and rQ_λ need to be the canonical elements of p_j and q_j in $\mathsf{uf}^{\lambda}_{j-1}$, and not of $\mathsf{uf}^{(\lambda,1)}_{j-1}$.

Hence, let us check what happens to the reconstruction of rP_λ and rQ_λ in $\mathsf{uf}_{j-1}^{(\lambda,1)}$. For the case shown in Fig. 3.7, their reconstructions are presented in Fig. 3.8.



$\mathsf{uf}_{j-1}^{\lambda}$
λ
parent^{j-1}
$rec(\mathsf{parent}^{j-1}, rP_\lambda)$
$rec(\mathsf{parent}^{j-1}, rQ_\lambda)$
$rec(\mathsf{parent}^{j-1}, rP_\lambda) \cup rec(\mathsf{parent}^{j-1}, rQ_\lambda)$
0 $\{0,1,2,3,4\}$ $\{5,6,7,8\}$ $\{0,1,2,3,4,5,6,7,8\}$
1 $\{4\}$ $\{5,6\}$ $\{4,5,6\}$
2 $\{4\}$ $\{5,6\}$ $\{4,5,6\}$
3 $\{4\}$ $\{5\}$ $\{4,5\}$
4 $\{\}$ $\{\}$ $\{\}$
rP_0 rQ_0 rP_1 rQ_1 rP_2 rQ_2 rP_3 rQ_3
$\mathsf{uf}_{j-1}^{(\lambda,1)}$
$\mathsf{parent}^{(j-1,1)}$
$rec(\mathsf{parent}_{j-1}^{1}, rP_\lambda)$
$rec(\mathsf{parent}^{(j-1,1)}, rQ_\lambda)$
$rec(\mathsf{parent}^{(j-1,1)}, rP_\lambda) \cup rec(\mathsf{parent}^{(j-1,1)}, rQ_\lambda)$
0 $\{0,1,2,3,4,\mathbf{5}\}$ $\{\cancel{5},6,7,8\}$ $\{0,1,2,3,4,5,6,7,8\}$
1 $\{4,\mathbf{5}\}$ $\{\cancel{5},6\}$ $\{4,5,6\}$
2 $\{4,\mathbf{5}\}$ $\{\cancel{5},6\}$ $\{4,5,6\}$
3 $\{4,\mathbf{5}\}$ $\{5\}$ $\{4,5\}$
4 $\{\}$ $\{\}$ $\{\}$

Figure 3.8: *Update from $\mathsf{uf}_{j-1}^{\lambda}$ to $\mathsf{uf}_{j-1}^{(\lambda,1)}$ for $a_j = \{p_j = 5, q_j = 4\}$, where rP_λ and rQ_λ are, respectively, the canonical elements of p_j and q_j in $\mathsf{uf}_{j-1}^{\lambda}$. On the right, we compare how the reconstruction of rP_λ and rQ_λ changes from $\mathsf{uf}_{j-1}^{\lambda}$ to $\mathsf{uf}_{j-1}^{(\lambda,1)}$. Note that the union of the reconstructions of rP_λ and rQ_λ is the same before and after the update.*

The key property to observe in Fig. 3.8 is that, although the reconstructions of rP_λ and rQ_λ change in $\mathsf{uf}_{j-1}^{(\lambda,1)}$ when compared to $\mathsf{uf}_{j-1}^{\lambda}$ (for $0 \leq \lambda < \beta$), the union of these reconstructions remains unaltered. Hence, for any $\mathsf{uf}_{j-1}^{(\lambda,1)}$, adding the arc (rP_λ, rQ_λ) (or (rQ_λ, rP_λ) depending on the gray-levels of rP_λ and rQ_γ, but assume without loss of generality that (rP_λ, rQ_λ) is added) still creates an updated union-find where rQ represents the CC $rec(\mathsf{uf}_{j-1}^{\lambda}, rP_\lambda) \cup rec(\mathsf{uf}_{j-1}^{\lambda}, rQ_\lambda)$.

More formally, this idea can be stated as the following proposition:

Proposition 3.18. *Let parent be an array seen as multiple union-finds uf^λ, for $\lambda \in \mathbb{K}$. Additionally, let p, q be two pixels and suppose rP_λ (resp. rQ_λ) is the canonical element of p (resp. q) in uf^λ. Finally, let $\beta \in \mathbb{K}$ be a gray-level where p and q are disconnected in uf^β and suppose that, for all $0 \leq \alpha \leq \beta$, $\mathsf{uf}^{(\alpha,1)}$ is the union-find updated from uf^α by changing the parent of rP_β to rQ_β. Then, $rec(\mathsf{uf}^{(\alpha,1)}, rP_\alpha) \cup rec(\mathsf{uf}^{(\alpha,1)}, rQ_\alpha) = rec(\mathsf{uf}^\alpha, rP_\alpha) \cup rec(\mathsf{uf}^\alpha, rQ_\alpha)$.*

Proof. To start the proof we note that, since $\alpha < \beta$, then rP_β and rQ_β exist in uf^α. Additionally, since uf^α and uf^β are originated from the parent array, then uf^β is a subgraph of uf^α. In particular, this implies that the subtree of rP_β is the same in uf^α and uf^β. A consequence of uf^β being a subgraph of uf^α is that rP_α (resp. rQ_α) is either rP_β (resp. rQ_β) or one of its ancestors.

The update from uf^α to $\mathsf{uf}^{(\alpha,1)}$ can be seen as a two-step process, where we first remove the previous arc (rP_β, p') and then add (rP_β, rQ_β) (see Fig. 3.9). The removal of (rP_β, p') has the effect of removing the subtree of rP_β from the subtree of its previous parent p', while the addition of the arc has the effect of adding the subtree of rP_β into the subtree of rQ_β.

With this idea in mind, we analyze the possible cases from rP_α and rQ_α. First, we analyze the case when $rP_\alpha \neq rP_\beta$ and rP_β is not a descendant of rQ_α in uf^α, which is the case shown in Fig. 3.9. Then, the change of arc makes rP_β become a child of rQ_β and, consequently, a descendant of rQ_α in $\mathsf{uf}^{(\alpha,1)}$. Hence, we have $rec(\mathsf{uf}^{(\alpha,1)}, rQ_\alpha) = rec(\mathsf{uf}^\alpha, rQ_\alpha) \cup rec(\mathsf{uf}^\alpha, rP_\beta)$, while $rec(\mathsf{uf}^{(\alpha,1)}, rP_\alpha) = rec(\mathsf{uf}^\alpha, rP_\alpha) \setminus rec(\mathsf{uf}^\alpha, rP_\beta)$. Using these properties, we conclude that

$rec(\mathsf{uf}^{(\alpha,1)}, rP_\alpha) \cup rec(\mathsf{uf}^{(\alpha,1)}, rQ_\alpha) = \big(rec(\mathsf{uf}^\alpha, rP_\alpha) \setminus rec(\mathsf{uf}^\alpha, rP_\beta)\big) \cup \big(rec(\mathsf{uf}^\alpha, rQ_\alpha) \cup rec(\mathsf{uf}^\alpha, rP_\beta)\big) = rec(\mathsf{uf}^\alpha, rP_\alpha) \cup rec(\mathsf{uf}^\alpha, rQ_\alpha)$.

There are two other cases to consider: if $rP_\alpha = rP_\beta$, then rP_α becomes a descendant of rQ_α in $\mathsf{uf}^{(\alpha,1)}$. From this part, a similar proof to the last case can be used, with the difference that the subtree of rP_α in $\mathsf{uf}^{(\alpha,1)}$ is included in the subtree of rQ_α. Since the proof is similar, it will be omitted here.

The last case happens when rP_β is a descendant of rQ_α in uf^α. Then, this implies that p and q were already connected or, equivalently, that $rP_\alpha = rQ_\alpha$. Hence, in this case, $rec(\mathsf{uf}^\alpha, rP_\alpha) \cup rec(\mathsf{uf}^\alpha, rQ_\alpha) = rec(\mathsf{uf}^\alpha, rP_\alpha) = rec(\mathsf{uf}^\alpha, rQ_\alpha)$.

The removal of the arc (rP_β, p') has the effect of removing the subtree of rP_β from the subtree of rP_α, while the addition of (rP_β, rQ_β) adds it to the subtree of rQ_α. But since $rP_\alpha = rQ_\alpha$, we are essentially removing and adding the subtree of rP_β in the subtree of $rP_\alpha = rQ_\alpha$, that is, the elements of the subtree do not change, which finishes the proof of the proposition. □

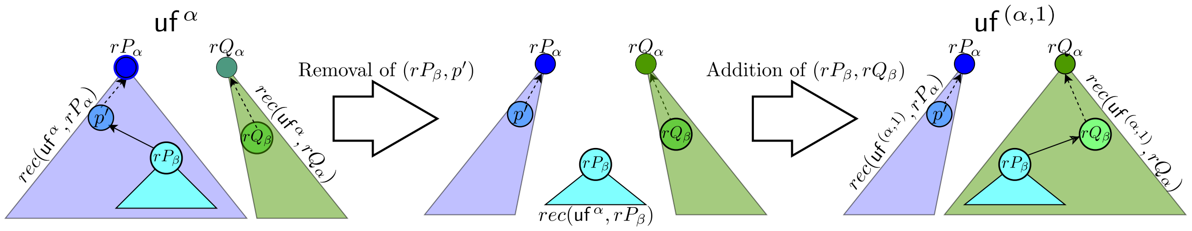

uf^α
rP_α
rQ_α
$rec(\mathsf{uf}^\alpha, rP_\alpha)$
p'
rP_β
rQ_β
$rec(\mathsf{uf}^\alpha, rQ_\alpha)$
Removal of (rP_β, p')
rP_α
rQ_α
p'
rQ_β
rP_β
$rec(\mathsf{uf}^\alpha, rP_\beta)$
Addition of (rP_β, rQ_β)
$\mathsf{uf}^{(\alpha,1)}$
rP_α
rQ_α
$rec(\mathsf{uf}^{(\alpha,1)}, rP_\alpha)$
p'
rP_β
rQ_β
$rec(\mathsf{uf}^{(\alpha,1)}, rQ_\alpha)$

Figure 3.9: *Update from uf^α to $\mathsf{uf}^{(\alpha,1)}$, seen as a two-step process where we first remove the arc pointing to the previous parent p' of rP_β and then we add the arc (rP_β, rQ_β). In this figure, the triangles indicate the subtree rooted in the nodes indicate by the circles.*

Note that this idea can be applied multiple times: if we update $\mathsf{uf}^{(\alpha,1)}$ to $\mathsf{uf}^{(\alpha,2)}$ using the same strategy or, more generally, $\mathsf{uf}^{(\alpha,k)}$ to $\mathsf{uf}^{(\alpha,k+1)}$, the same properties are valid.

In this way, an idea to finish the update from parent^{j-1} to parent^j is to repeat the same process over and over, finding the next gray-level β where p_j and q_j are disconnected (in decreasing order of gray-levels, to make sure that already correct union-finds do not change), and add the arc (rP_β, rQ_β) or (rQ_β, rP_β) to connect them. The continuation of Fig. 3.8 using this approach is presented in Fig 3.10. It is worth noting that the updates performed in the parent array may make a root node to a non-root node (for example, the node 6 pointing to the node 7 in the bottom right in Fig 3.10).

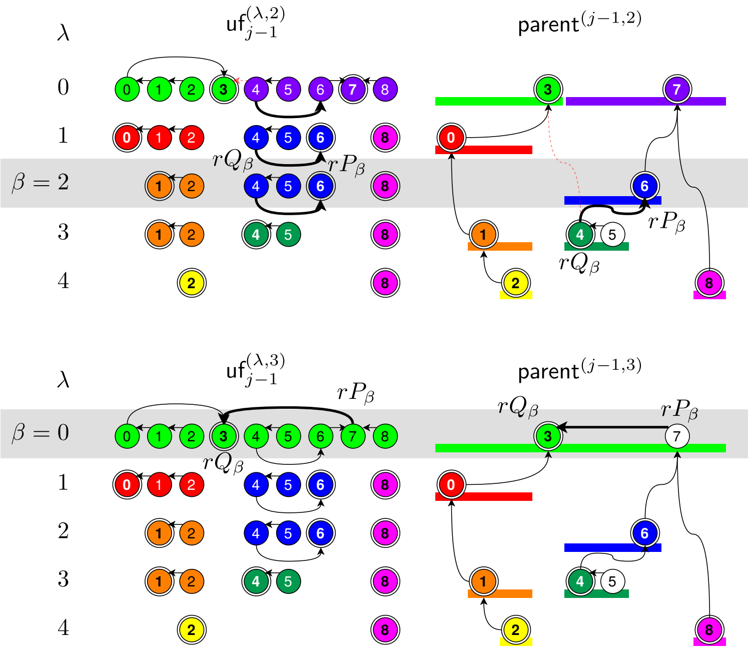


Figure 3.10: *Update of the union-finds (and its respective* parent *arrays), by connecting* rP_β *and* rQ_β *for all gray-levels* β *where they are disconnected. At the end, all canonical elements of* $\mathsf{uf}_{j-1}^{(\lambda,3)}$ *store* $\mathcal{A}_{alg}^{j}$*-CCs of* $X_\lambda(f)$*, for all* $\lambda \in \mathbb{K}$*, which means* $\mathsf{parent}^{(j-1,3)}$ *stores the* $\mathcal{A}_{alg}^{j}$*-CCs of* f*.*

Using these ideas, we can develop a first version of an algorithm that updates parent^{j-1} to parent^{j}. These ideas are presented in Alg. 12.

Algorithm 12 Algorithm that, given a pair $\{p_j, q_j\}$, updates a given array $\mathsf{parent} = \mathsf{parent}^{j-1}$ to parent^{j}, where p_j and q_j are comparable in parent^{j}.

```
1: procedure ParentUpdateNaive($f$, parent, $p_j$, $q_j$)
2:     $\mathsf{parent}^{j-1} \leftarrow$ copy of parent;
3:     for $K > \beta \geq 0$ do
4:         if findRep($f, \mathsf{parent}, p_j, \beta$) $\neq$ findRep($f, \mathsf{parent}, q_j, \beta$) then    ▷ $p_j$ and $q_j$ are disconnected
5:             $rP_\beta \leftarrow$ findRep($f, \mathsf{parent}^{j-1}, p_j, \beta$);    ▷ Canonical element of $p_j$ in $\mathsf{uf}_{j-1}^{\beta}$
6:             $rQ_\beta \leftarrow$ findRep($f, \mathsf{parent}^{j-1}, q_j, \beta$);    ▷ Canonical element of $q_j$ in $\mathsf{uf}_{j-1}^{\beta}$
7:             unionOrdered($f$, parent, $rP_\beta$, $rQ_\beta$);
```

Although Alg. 12 can be used to update parent^{j-1} to parent^{j}, there are still some optimizations that can be performed. First of all, it is possible not to store parent^{j-1}, and find the representatives rP_λ and rQ_λ in parent directly instead. Naturally, we need some additional properties to show how to perform this optimization.

Another optimization consists of not testing all gray-levels λ at Line 3 of Alg. 12. For example, in gray-levels λ where p_j and q_j are already connected in $\mathsf{uf}_{j-1}^{\lambda}$ or one of them is not a foreground pixel, there is no need to connect them.

With these ideas in mind, we now focus on finding properties to implement an optimized algorithm. We recall that, given a pixel p_j, the canonical element rP_λ of p_j in $\mathsf{uf}_{j-1}^{\lambda}$ can be found in parent^{j-1} by calling findRep($f, \mathsf{parent}^{j-1}, p, \lambda$), that is to say, rP_λ is the last canonical element in the path from p_j to the root node $\perp$ in parent^{j-1} satisfying $f(rP_\lambda) \geq \lambda$.

Hence, to obtain the canonical elements rP_λ and rQ_λ of $\mathsf{uf}_{j-1}^{\lambda}$ in $\mathsf{uf}_{j-1}^{(\lambda,k)}$ for any $\lambda \in \mathbb{K}$ and any valid value of k, we need to observe how paths change when the parent of a node is modified. For that, an illustration is provided in Fig 3.11.

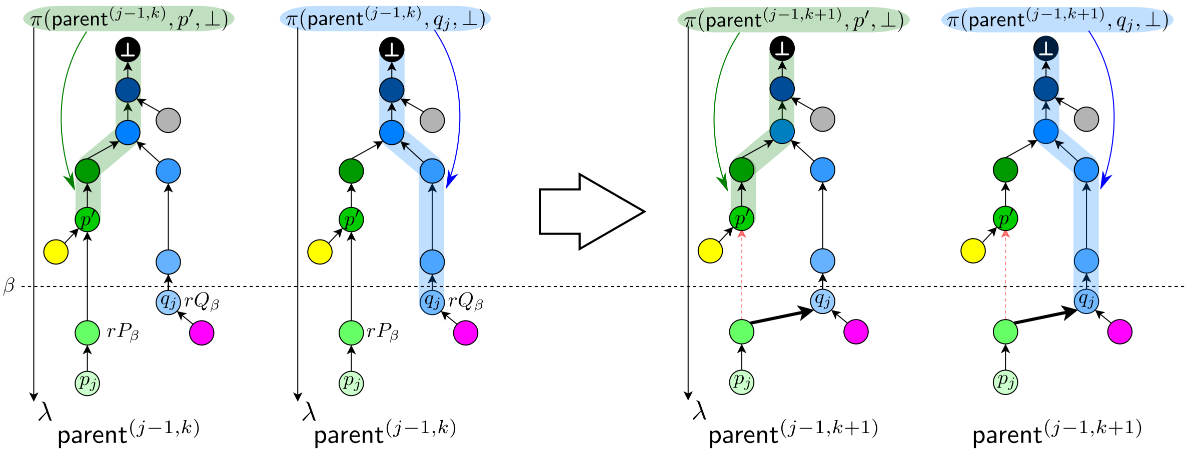

$\pi(\mathsf{parent}^{(j-1,k)}, p', \perp)$
$\pi(\mathsf{parent}^{(j-1,k)}, q_j, \perp)$
$\pi(\mathsf{parent}^{(j-1,k+1)}, p', \perp)$
$\pi(\mathsf{parent}^{(j-1,k+1)}, q_j, \perp)$
$\perp$
p'
β
q_j
rQ_β
rP_β
p_j
λ
$\mathsf{parent}^{(j-1,k)}$
$\mathsf{parent}^{(j-1,k)}$
$\mathsf{parent}^{(j-1,k+1)}$
$\mathsf{parent}^{(j-1,k+1)}$

Figure 3.11: *Update from $\mathsf{parent}^{(j-1,k)}$ to $\mathsf{parent}^{(j-1,k+1)}$ by connecting rP_β to rQ_β, where the two column in the left show the paths from $p' = \mathsf{parent}^{(j-1,k)}[rP_\beta]$ to $\perp$ and rQ_β to $\perp$ and the two columns in the right show the same paths after the update. Green nodes are ancestors of p_j and blue nodes are ancestors of q_j. Note that, before and after the change or arc, the paths from p' to $\perp$ and rQ_β to $\perp$ are unaltered.*

It should be clear from Fig. 3.11 that the following property holds:

Proposition 3.19. *Suppose Alg. 12 is running and let $\mathsf{parent}^{(j-1,k)}$ denote the state of parent array after k calls of* UNIONORDERED. *Suppose that the $(k+1)$-th call of* UNIONORDERED *updates the parent of rP_β from p' to rQ_β, namely, $\mathsf{parent}^{(j-1,k)}[rP_\beta] = q'$ but $\mathsf{parent}^{(j-1,k+1)}[rP_\beta] = rQ_\beta$. Then, $\pi(\mathsf{parent}^{(j-1,k)}, p', \perp) = \pi(\mathsf{parent}^{(j-1,k+1)}, p', \perp)$ and*
$\pi(\mathsf{parent}^{(j-1,k)}, rQ_\beta, \perp) = \pi(\mathsf{parent}^{(j-1,k+1)}, rQ_\beta, \perp)$.

From Prop. 3.19, it is easy to prove that:

Proposition 3.20. *Suppose Alg. 12 is running and let $\mathsf{parent}^{(j-1,k)}$ denote the state of parent array after k calls of* UNIONORDERED. *Suppose that the $(k+1)$-th call of* UNIONORDERED *updates the parent of rP_β from p' to rQ_β. Then, $\pi(\mathsf{parent}^{j-1}, p', \perp) = \pi(\mathsf{parent}^{(j-1,k+1)}, p', \perp)$ and $\pi(\mathsf{parent}^{j-1}, rQ_\beta, \perp) = \pi(\mathsf{parent}^{(j-1,k+1)}, rQ_\beta, \perp)$.*

Proof. By applying Prop. 3.19 multiple times, it is easy to see that
$\pi(\mathsf{parent}^{(j-1,\ell)}, p', \perp) = \pi(\mathsf{parent}^{(j-1,k+1)}, p', \perp)$ and
$\pi(\mathsf{parent}^{(j-1,\ell)}, rQ_\beta, \perp) = \pi(\mathsf{parent}^{(j-1,k+1)}, rQ_\beta, \perp)$ for any $\ell \leq k$. In particular, for $\ell = 0$, $\mathsf{parent}^{(j-1,0)} = \mathsf{parent}^{j-1}$ and the property is proved. □

Propositions 3.19 and 3.20 are defined assuming an arc from rP_β to rQ_β, but it is easy to see that the analogous properties would be valid for the case where rP_β becomes the parent of rQ_β.

Under the assumptions of Prop. 3.20, since the paths from p' to $\perp$ and rQ_β to $\perp$ are the same in $\mathsf{parent}^{(j-1,k+1)}$ and parent^{j-1}, we can obtain any canonical element of p_j or q_j with gray-level $\lambda < \beta$. For instance, by definition, rP_λ is the last canonical element in the path from p_j to $\perp$ in parent^{j-1} with gray-level higher than or equal to λ. If $\beta \leq f(p_j)$ and $\lambda \leq f(p_j)$, then both rP_λ and rP_β are in this path and, in particular, rP_λ is an ancestor of rP_β when $\lambda < \beta$. Hence, rP_λ can also be seen as the last canonical element in the path from $\mathsf{parent}^{j-1}[rP_\beta] = p'$ to $\perp$ in parent^{j-1}, and this path is the same in parent^{j-1} and $\mathsf{parent}^{(j-1,k+1)}$.

Since the values of β from Alg. 12 are obtained using a decreasing order, this strategy can be applied. Hence, we can save memory by, instead of storing the entire parent^{j-1} array, storing only the variables p' and rP_β from the previous loop and using the updated parent array to find the canonical elements of p_j and q_j.

For the other optimization, we first note the two pixels p_j and q_j are not $\mathcal{A}_{alg}^{j-1}$-connected if and only if they have different representatives at level $\min\{f(p_j), f(q_j)\}$ (Props. 3.15 and 3.16). Also, the highest possible gray-level where p_j and q_j can be disconnected is also $\min\{f(p_j), f(q_j)\}$, since one of them would not be a foreground pixel at a higher gray-level. Hence, in our previous examples, the highest possible value for β is $\beta = \min\{f(p_j), f(q_j)\}$. For our purposes, since we assume that the neighborhood $\mathcal{A}$ is symmetric, $f(p_j) \geq f(q_j)$ can always be enforced by swapping the labels of the variables p_j and q_j when that condition is not true. Hence, we now assume that $f(p_j) \geq f(q_j)$, implying that the first value for β is $\beta = f(q)$. We refer to this first value as β_1.

The lowest value for β that needs to be considered is the lowest gray-level where p_j and q_j are disconnected. Note that, given a gray-level $\lambda \in \mathbb{K}$, if $c \neq \perp$ is an ancestor of both p_j and q_j in $\mathsf{uf}^{\lambda}_{j-1}$, then c is also an ancestor of p_j and q_j in $\mathsf{uf}^{\alpha}_{j-1}$, for any $f(c) < \alpha < \lambda$, since $\mathsf{uf}^{\lambda}_{j-1}$ is a subgraph of $\mathsf{uf}^{\alpha}_{j-1}$.

If p_j and q_j have a common ancestor $c \neq \perp$, this means that they have the same canonical element in $\mathsf{uf}^{f(c)}_{j-1}$. Hence, let c' be the first common ancestor of p_j and q_j in parent. This implies that, at any level $\lambda > f(c')$, p_j and q_j are disconnected in $\mathsf{uf}^{\lambda}_{j-1}$, and the lowest possible gray-level where they are disconnected is $f(c') + 1$. Therefore, we can restrict the range of values of β to the interval $]f(c'), f(q)]$.

It is important to note that, even inside this range, some gray-levels can be skipped. This property is explained in Prop. 3.21.

Proposition 3.21. *Suppose Alg. 12 is running and let $\mathsf{parent}^{(j-1,k)}$ denote the state of parent array after k calls of* UNIONORDERED. *In particular, assume that the k-th call of* UNIONORDERED *assigned $\mathsf{parent}^{(j-1,k)}[rP_{\beta_k}] = rQ_{\beta_k}$. Then, for any $0 \leq \lambda \leq \beta_k$ such that $rP_\lambda = rP_{\beta_k}$, $rec(\mathsf{parent}^{(j-1,k)}, rQ_\lambda) = rec(\mathsf{parent}^{j-1}, rP_\lambda) \cup rec(\mathsf{parent}^{j-1}, rQ_\lambda)$.*

Proof. A side effect of assigning $\mathsf{parent}^{(j-1,k)}[rP_{\beta_k}] = rQ_{\beta_k}$ is that it makes rP_{β_k} a descendant of rQ_λ for all $0 \leq \lambda < \beta_k$ and, therefore, $rec(\mathsf{parent}^{(j-1,k)}, rQ_\lambda) = rec(\mathsf{uf}^{k}_{j-1}, rQ_\lambda) \cup rec(\mathsf{uf}^{k}_{j-1}, rP_{\beta_k})$. When $rec(\mathsf{uf}^{k}_{j-1}, rP_{\beta_k}) = rec(\mathsf{uf}^{k}_{j-1}, rP_\lambda)$, then $rec(\mathsf{parent}^{(j-1,k)}, rQ_\lambda) = rec(\mathsf{uf}^{k}_{j-1}, rQ_\lambda) \cup rec(\mathsf{uf}^{k}_{j-1}, rP_\lambda)$, namely, rQ_λ already represents the $\mathcal{A}^{j}_{alg}$-CC that merges the disjoint components containing p_j and q_j at level λ and is already correct. □

In other words, a single call of UNIONORDERED can correct multiple union-finds. For the example presented in Figs. 3.8 and 3.10, the ranges of gray-levels corrected by each update of the parent array is shown in Fig. 3.12.

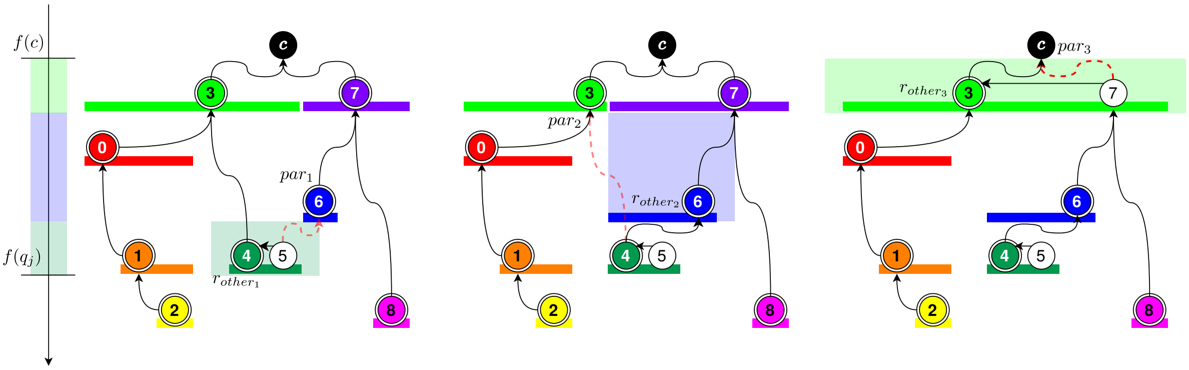


Figure 3.12: *The changes of parenthood relation from Alg. 12 applied to the examples shown in Fig. 3.8 and 3.10, with the range of gray-levels corrected by each change highlighted.*

The final step to develop our optimized algorithm is to understand which gray-levels can be skipped. For that, in the context of Prop. 3.21, we need to find, for each step k, the highest gray-level α such that $rP_\alpha \neq rP_{\beta_k}$. However, this can be easily done in the parent array: any ancestor of rP_{β_k} has gray-level higher than β_k and, therefore, $\alpha = f(\mathsf{parent}^{(j-1,k)}[rP_{\beta_k}])$. Then, when the change $\mathsf{parent}^{(j-1,k)}[rP_{\beta_k}] \leftarrow rQ_\beta$ is performed, all gray-levels from $]\alpha, \beta]$ are now correct, and we repeat this process again until we correct all gray-levels where p_j and q_j were originally disconnected.

Putting all these ideas together, one can write an optimized algorithm for constructing the parent array using an unordered strategy, using the following approach:

Algorithm 13 Updating the parent array to, given a pair (p, q), make p and q comparable. This algorithm assumes $f(p) \geq f(q)$, and the algorithm is called initially with $cur = p$ and $other = q$.

```
1: procedure PARENTUPDATE($f$, parent, $cur$, $other$)
2:     $r_{cur} \leftarrow$ FINDREP($f$, parent, $cur$, $f(other)$);                ▷ $f(other) \leq f(cur)$
3:     $r_{other} \leftarrow$ FINDREP($f$, parent, $cur$, $f(other)$);
4:     if $r_{cur} \neq r_{other}$ then                          ▷ If $cur$ and $other$ are disconnected
5:         $par \leftarrow parent[r_{cur}]$;                           ▷ Previous parent of $r_{cur}$
6:         parent$[r_{cur}] \leftarrow r_{other}$;     ▷ Corrects the interval $[f(par)+1, f(r_{other})]$
7:         PARENTUPDATE($f$, parent, $r_{other}$, $par$);  ▷ Recursion to correct gray-levels $\lambda \leq f(par)$.
```

To give some context about Alg. 13, the name of the variables were changed to be independent of p_j and q_j. In this sense, the paths in $\{\pi(\mathsf{parent}, p_j, \perp), \pi(\mathsf{parent}, q_j, \perp)\}$ are classified as the current path (cur) and the other path ($other$), and we always make a canonical element from the current path (r_{cur}) point to the canonical element of the other path (r_{other}) at Line 6. Thus, this makes all gray-levels in the interval $]f(par), f(other)]$ correct and the algorithm is recursively called to correct the gray-levels lower than or equal to $\alpha = f(par)$, as discussed above. An example is provided in Fig. 3.13.

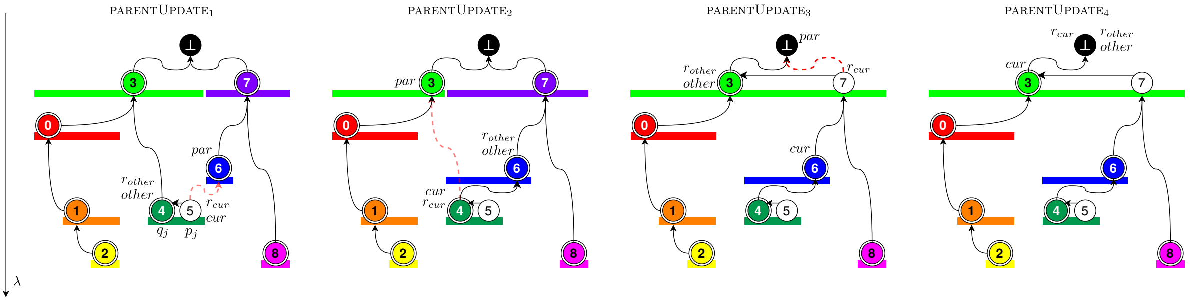


Figure 3.13: *Simulation of the optimized algorithm* PARENTUPDATE, *showing that it performs the same changes as* PARENTUPDATENAIVE, *but skips unnecessary gray-levels and does not store* parent^{j-1}.

To finish the proof of correctness of Alg. 13, we have to prove that Alg. 13 finishes and that the assignment at Line 6 is correct (in other words, it generates a decreasing parent array and links the correct canonical elements). The first part is done in Prop. 3.22.

Proposition 3.22. *Suppose Alg. 13 is called and let* PARENTUPDATE$_k$ *denote the k-th call of* PARENTUPDATE. *Additionally, let cur_k (resp. $other_k$, r_{cur_k}, r_{other_k} and par_k) denote the value of cur inside* PARENTUPDATE$_k$. *Then, there exist a $k' \in \mathbb{Z}$, $k \geq 1$ such that $r_{cur_{k'}} = r_{other_{k'}}$.*

Proof. We first observe that, for any $k \geq 1$, we have that $f(par) < f(r_{other_k})$ by construction, since par is the parent r_{cur_k} and r_{cur_k} is, by definition, the last element in the path from cur to $\perp$ with gray-level $\lambda \geq f(other_k) = f(r_{other_k})$. Since $other_k = par_{k-1}$ for any $k > 1$, this implies that $f(other_k)$ decreases as k increases.

Since the number of gray-levels is finite, then in the worst case, $f(other_k)$ has a finite amount of values until either the condition $r_{cur_k} = r_{other_k}$ is true or $f(other_k) \notin \mathbb{K}$. This can only happen if $other_k$ is not a pixel of the domain of the image and, in the parent array, this can only happen if $other_k = \perp$ (we recall that, in Sec. 3.1, we set $f(\perp) = -1$). When this happens, then we have $r_{cur_k} = \perp = r_{other_k}$, as desired, and the algorithm finishes. □

The proof that Alg. 13 generates a decreasing array is done in Prop 3.23.

Proposition 3.23. *Suppose Alg. 13 is called and let* PARENTUPDATE$_k$ *denote the k-th call of* PARENTUPDATE. *Additionally, let cur_k (resp. $other_k$, r_{cur_k}, r_{other_k} and par_k) denote the value of cur inside* PARENTUPDATE$_k$. *Then, $f(r_{cur_k}) \geq f(r_{other_k})$.*

Proof. Let $k \in \mathbb{Z}$ satisfy $k \geq 1$. Note that r_{cur_k} and r_{other_k} are the representatives of cur_k and $other_k$ at level $f(other_k)$ and, therefore, $f(r_{cur_k}) \geq f(other_k)$ and $f(r_{other_k}) \geq f(other_k)$. But since r_{other} is the representative of $other_k$ at level $f(other_k)$, then $f(r_{other_1}) \leq f(q)$. That means $f(r_{other_k}) = f(other_k)$ and we conclude that $f(r_{cur_k}) \geq f(other_k) = f(r_{other_k})$, as desired. □

To prove the last part, we need some additional properties. First, we adapt Prop. 3.20 to the context of Alg. 13:

Proposition 3.24. *Suppose Alg. 13 is running. Let* PARENTUPDATE$_k$ *denote the k-th call of* PARENTUPDATE, cur_k *(resp.* $other_k$, r_{cur_k}, r_{other_k} *and* par_k*) denote the value of cur inside* PARENTUPDATE$_k$ *and suppose that* PARENTUPDATE$_k$ *receives an array* $\mathsf{parent}^{(j-1,k-1)}$ *and updates it to* $\mathsf{parent}^{(j-1,k)}$ *at Line 6. Then,* $\pi(\mathsf{parent}^{(j-1,k)}, other_k, \perp) = \pi(\mathsf{parent}^{j-1}, other_k, \perp)$ *and* $\pi(\mathsf{parent}^{(j-1,k)}, par_k, \perp) = \pi(\mathsf{parent}^{j-1}, par_k, \perp)$.

Using this property, one can prove that the variables r_{cur_k} and r_{other_k} are always in the path $\pi(\mathsf{parent}^{j-1}, p_j, \perp)$ or $\pi(\mathsf{parent}^{j-1}, q_j, \perp)$. More specifically, the variables r_{cur_k} and r_{other_k} are always alternating and in different paths. This is proved in Prop. 3.25.

Proposition 3.25. *Suppose Alg. 13 is running. Let* $k \in \mathbb{Z}$, *let* PARENTUPDATE$_k$ *denote the k-th call of* PARENTUPDATE, cur_k *(resp.* $other_k$, r_{cur_k}, r_{other_k} *and* par_k*) denote the value of cur inside* PARENTUPDATE$_k$ *and suppose that* PARENTUPDATE$_k$ *receives an array* $\mathsf{parent}^{(j-1,k-1)}$ *and updates it to* $\mathsf{parent}^{(j-1,k)}$ *at Line 6. Then:*

1. *If k is odd:*
   - *(a)* $cur_k \in \pi(\mathsf{parent}^{j-1}, p_j, \perp)$;
   - *(b)* $r_{cur_k} \in \pi(\mathsf{parent}^{j-1}, p_j, \perp)$;
   - *(c)* $other_k \in \pi(\mathsf{parent}^{j-1}, q_j, \perp)$;
   - *(d)* $r_{other_k} \in \pi(\mathsf{parent}^{j-1}, q_j, \perp)$;
2. *If k is even:*
   - *(a)* $cur_k \in \pi(\mathsf{parent}^{j-1}, q_j, \perp)$;
   - *(b)* $r_{cur_k} \in \pi(\mathsf{parent}^{j-1}, q_j, \perp)$;
   - *(c)* $other_k \in \pi(\mathsf{parent}^{j-1}, p_j, \perp)$;
   - *(d)* $r_{other_k} \in \pi(\mathsf{parent}^{j-1}, p_j, \perp)$;

Proof. This property can be proved by induction.

For the base case $k = 1$, $cur_k = p_j$ and $other_k = q_j$, so they are in the paths $\pi(\mathsf{parent}^{j-1}, p_j, \perp)$ and $\pi(\mathsf{parent}^{j-1}, p_j, \perp)$, respectively. The variable r_{cur_k} is either cur_k or an ancestor, so $r_{cur_k} \in \pi(\mathsf{parent}^{j-1}, p_j, \perp)$. For the same reason, we have that $r_{other_k} \in \pi(\mathsf{parent}^{j-1}, q_j, \perp)$, as desired.

Now, suppose $k > 1$ and that the proposition is valid for any $k' < k$. Additionally, without loss of generality, assume that k is odd.

By definition, $cur_k = r_{other_{k-1}}$ and r_{cur_k} is the canonical element of cur_k at level $f(other_k)$ obtained from $\mathsf{parent}^{(j-1,k-1)}$. Hence, both cur_k and r_{cur_k} are in the path $\pi(parent^{k-1}_{j-1}, cur_k, \perp) = \pi(parent^{k-1}_{j-1}, r_{other_{k-1}}, \perp)$.

According to Prop. 3.24, $\pi(\mathsf{parent}^{(j-1,k-1)}, r_{other_{k-1}}, \perp) = \pi(\mathsf{parent}^{j-1}, r_{other_{k-1}}, \perp)$ and, applying the induction hypothesis, we have that $r_{other_{k-1}} \in \pi(\mathsf{parent}^{j-1}, p_j, \perp)$. Thus, $\pi(\mathsf{parent}^{j-1}, r_{other_{k-1}}, \perp) \subseteq \pi(\mathsf{parent}^{j-1}, p_j, \perp)$, and $cur_k, r_{cur_k} \in \pi(\mathsf{parent}^{j-1}, r_{other_{k-1}}, \perp)$ implies that $cur_k, r_{cur_k} \in \pi(\mathsf{parent}^{j-1}, p_j, \perp)$, as desired.

Using the fact that $other_k = par_{k-1}$, $other_k, r_{other_k} \in \pi(\mathsf{parent}^{j-1}, q_j, \perp)$ can be proved. Since the proof is similar to the one used for cur_k and r_{cur_k}, it will be skipped. □

Finally, to prove the correctness of our algorithm, we need to prove Prop. 3.26.

Proposition 3.26. *Suppose Alg. 13 is running. Let $k \in \mathbb{Z}$, let* PARENTUPDATE$_k$ *denote the k-th call of* PARENTUPDATE *and suppose that* PARENTUPDATE$_k$ *receives an array $\mathsf{parent}^{(j-1,k-1)}$ and updates it to $\mathsf{parent}^{(j-1,k)}$ at Line 6. Then, r_{cur_k} and r_{other_k} are the canonical elements of p_j and q_j in parent^{j-1} at level $f(other_k)$.*

Proof. Let $k \geq 1$ and, without loss of generality, assume that k is odd. By construction, r_{cur_k} is the canonical element of cur_k in $\mathsf{parent}^{(j-1,k-1)}$ at level $f(other_k)$, that is, $r_{cur_k} \in \pi(\mathsf{parent}^{(j-1,k)} - 1, cur_k, \perp)$. From Prop. 3.25, we have that $cur_k \in \pi(\mathsf{parent}^{j-1}, p_j, \perp)$, implying that $\pi(\mathsf{parent}^{j-1}, cur_k, \perp) \subseteq \pi(\mathsf{parent}^{j-1}, p_j, \perp)$. Therefore, r_{cur_k} can also be seen as the last canonical element in $\pi(\mathsf{parent}^{j-1}, p_j, \perp)$ with gray-level higher than or equal to $f(other_k)$, namely, the canonical element of p_j in parent^{j-1} at level $f(other_k)$. Hence, $r_{cur_k} = rP_{f(other_k)}$.

Using an analogous proof, we conclude that $r_{other_k} = rQ_{f(other_k)}$, as desired. □

Combining the properties seen in this Chapter, we can prove that Alg. 13 is correct: thanks to Props. 3.21 and 3.26, the k-th call of Alg. 13 corrects the gray-levels in the interval $]f(par_k), f(other_k)] =]f(other_{k+1}), f(other_k)]$. Supposing that η calls of PARENTUPDATE are performed and c is the first common ancestor of p_j and q_j, the pixels that we want to connect, then these calls combined correct the interval $]f(other_\eta), f(other_1)] =]f(c), f(q_j)]$, which comprises exactly the interval of gray-levels where p_j and q_j were originally disconnected.

Chapter 4

Proposed Method

In this chapter, we explain the main concepts developed in this thesis. Based on the two last chapters, we explain how to define a memory-efficient way of storing component-hypertrees and also how to develop fast algorithms to build this optimized structure. It is worth noting that the theory and algorithms presented in this chapter are the foundation of 3 papers [MAS+19b, MAS+19a, MPAH20] that were published during the development of this Ph.D.

4.1 Component-Hypertree Construction

4.1.1 General Algorithm

Based on the properties and algorithms seen in the last two chapters, one can build an algorithm for component-hypertree construction, as follows: given a gray-level image f and a sequence of increasing symmetric neighborhoods $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, the most natural idea for building the component-hypertree of f using $\mathbb{A}$ consists of computing, for each neighborhood $\mathcal{A}_i$ ($1 \leq i \leq n$), its respective array parent_i and use it to allocate the corresponding component-tree. Then, nodes from consecutive component-trees can be linked according to the inclusion relation of the nodes. A template showing this idea is shown in Alg. 14.

Algorithm 14 Complete component-hypertree construction.

```
procedure BUILDHYPERTREE($f$, $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$)
    $\mathsf{parent}_1 \leftarrow$ MAKESET($D_f$);
    Compute $\mathsf{parent}_1$ using any algorithm for component-tree construction;
    Allocate the complete component-tree of $f$ and $\mathcal{A}_1$ based on $\mathsf{parent}_1$;
    for $2 \leq i \leq n$ do
        Compute $\mathsf{parent}_i$ by updating $\mathsf{parent}_{i-1}$ calling PARENTUPDATE for all $\mathcal{A}_i$-neighbors.
        Allocate nodes based on canonical elements of $\mathsf{parent}_i$.
        Allocate parent arcs of nodes allocated at step $i$ based on the arcs of $\mathsf{parent}_i$.
        for all nodes $N$ of the complete component-tree of $\mathcal{A}_{i-1}$ do
            Find $N'$ satisfying $adj(N) = i$, $CC(N) \subseteq CC(N')$ and $f(N) = f(N')$;
            Allocate the composite arc $(N, N')$;
```

It is easy to note that, although this implementation works, it is not very efficient memory-wise because a complete component-hypertree stores multiple nodes representing the same CC. In this regard, just like there exist compact ways of representing component-trees, we want efficient ways of storing component-hypertrees that can be quickly computed and minimize the amount of redundant information stored.

Hence, in this chapter, we explain how the previous background of component-trees, combined with modified algorithms specifically designed to tackle this problem, can be used in the development of algorithms and data structures for efficient component-hypertree computation and storage.

Similar to the previous chapters, the proposed method is divided into two parts: first, we introduce the theoretical definitions and properties that we want our optimized hypertree to satisfy and later we introduce our proposed algorithms for efficient component-hypertree construction, explaining how they are related to these theoretical concepts.

4.2 Compact Component-Hypertrees

Based on the reduction from complete component-trees to (non-complete) component-trees, we define the concept of compact component-hypertrees. In the compact representation of component-hypertrees, each CC is represented only by one node, and arcs are mapped only from arcs of the complete representation that linked nodes representing different CCs. To formally define compact-hypertree, we first need to introduce the concepts of compact nodes and compact arcs.

4.2.1 Compact Nodes

Let f be a gray-level image, $\mathbb{A}$ a sequence of increasing symmetric neighborhood and $CCH = (V_{CCH}, \mathcal{E}_{CCH})$ the complete component-hypertree of f and $\mathbb{A}$. Then, given any node $N \in V_{CCH}$, the compact node of N is the node N' returned by the mapping $cn : V_{CCH} \to V_{CCH}$, defined as follows:

$$\begin{aligned} cn(N) = N' \in V_{CCH} : & \, CC(N') = CC(N) \text{ and} \\ & \forall M \in V_{CCH} \text{ such that } CC(M) = CC(N') = CC(N), \\ & f(M) \leq f(N') \text{ and } adj(M) \geq adj(N') \end{aligned} \tag{4.1}$$

In other words, given a node $N \in V_{CCH}$, the compact node $N' = cn(N)$ is the node N' of the complete-hypertree representing the same CC as N with the highest gray-level and the lowest neighborhood index.

A question that may arise from this definition is whether $cn(N)$ is well-defined, namely, if $cn(N)$ is unique. This is proved in Prop. 4.1:

Proposition 4.1. *Let f be a gray-level image, $\mathbb{A}$ a sequence of increasing neighborhoods and $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ the complete hypertree of f and $\mathbb{A}$. Additionally, let $N \in V_{CCH}$ be an arbitrary node and let $\mathbb{M}_N = \{M_N \in V_{CCH} : CC(M_N) = CC(N)\}$. Finally, suppose $M, M' \in \mathbb{M}_N$ are two nodes satisfying the following conditions:*

- *Among all nodes of $\mathbb{M}_N$ with the highest possible gray-level, M is the node with the lowest neighborhood index.*
- *Among all nodes of $\mathbb{M}_N$ with the lowest possible neighborhood index, M' is the node with the highest gray-level.*

Then, M and M' are the same node.

Proof. First, we observe that $|\mathbb{M}_N| \geq 1$ since $N \in \mathbb{M}_N$. If $|\mathbb{M}_N| = 1$, then there is only one choice for M and M', which is $M = M' = N$, which already proves the proposition.

Suppose then that $|\mathbb{M}_N| > 1$. By assumption, M has the highest possible gray-level and M' has the lowest possible neighborhood index. Hence, $adj(M') \leq adj(M)$ and $f(M) \geq f(M')$.

Now, suppose by contradiction that $M \neq M'$. Then, there are 3 possible cases:

1. $adj(M) = adj(M')$: note that, since $M \neq M'$, this implies that $f(M) > f(M')$. But then, M' is not the node with highest gray-level among all nodes with the smallest neighborhood index because M exists, satisfies $adj(M) = adj(M')$ and has higher gray-level than M', which is a contradiction. In other words, assuming $M \neq M'$ leads to a contradiction and, therefore, M and M' must be the same node.

2. $f(M) = f(M')$ and $adj(M') < adj(M)$: the proof is very similar to the last case and will be omitted.

3. $f(M) > f(M')$ and $adj(M') < adj(M)$: this case is illustrated in Fig. 4.1.

   By definition, this implies that $CC(M)$ is a $\mathcal{A}_{adj(M)}$-CC of $X_{f(M)}(f)$, or in other words, any $p \in CC(M)$ satisfies $f(p) \geq f(M)$. Using the same argument, $CC(M')$ is a $\mathcal{A}_{adj(M')}$-CC of $X_{f(M')}(f)$. Since $CC(M) = CC(M') = CC(N')$, we can combine these arguments and conclude that $CC(N')$ is also a $\mathcal{A}_{adj(M')}$-CC of $X_{f(M)}(f)$, that is, there exists a node $N' = (\mathcal{C}, f(M), adj(M')) \in V_{CCH}$ where $\mathcal{C} = CC(N')$, which is a contradiction, since we found a node N' with the same gray-level as M but with lower neighborhood index that also has a higher gray-level compared to M'. Hence, assuming $M \neq M'$ leads to a contradiction and, therefore, M and M' must be the same node.

For all possible cases, a contradiction was obtained. Therefore, we conclude that $M = M'$ and the proposition is proved. □

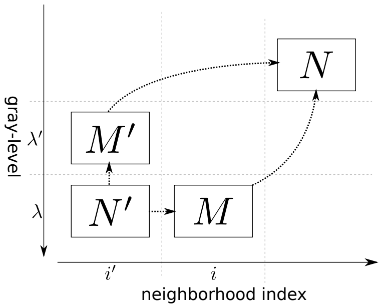


Figure 4.1: *Visualization of the nodes for the proof of the third case in Prop. 4.1.*

Since the cn mapping is defined, the set of compact nodes can be defined as:

$$V_{CH} = \bigcup_{N \in V_{CCH}} cn(N) \tag{4.2}$$

Figure 4.2 depicts an example that highlights the compact nodes of a complete hypertree.

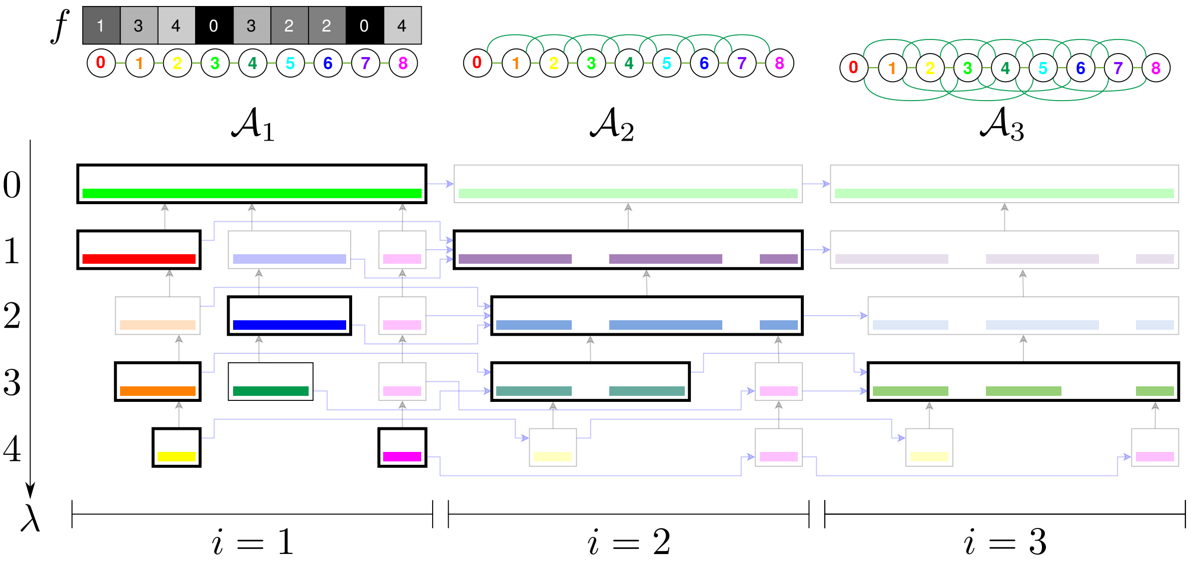


Figure 4.2: *A complete component-hypertree with the compact nodes highlighted.*

4.2.2 Compact Arcs

With the concept of compact nodes defined, given an arc $e = (N, N') \in \mathcal{E}_{CCH}$ with $CC(N) \neq CC(N')$, the compact arc of e is defined as:

$$ce(e = (N, N')) = (cn(N), cn(N')) \tag{4.3}$$

The set of compact arcs is defined as:

$$\mathcal{E}_{CH} = \bigcup_{\substack{e=(N,N')\in\mathcal{E}_{CCH} \\ CC(N)\neq CC(N')}} cn(e) \tag{4.4}$$

Finally, we define the compact component-hypertree of f and $\mathbb{A}$ as the graph $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$. An example is given in Fig. 4.3.

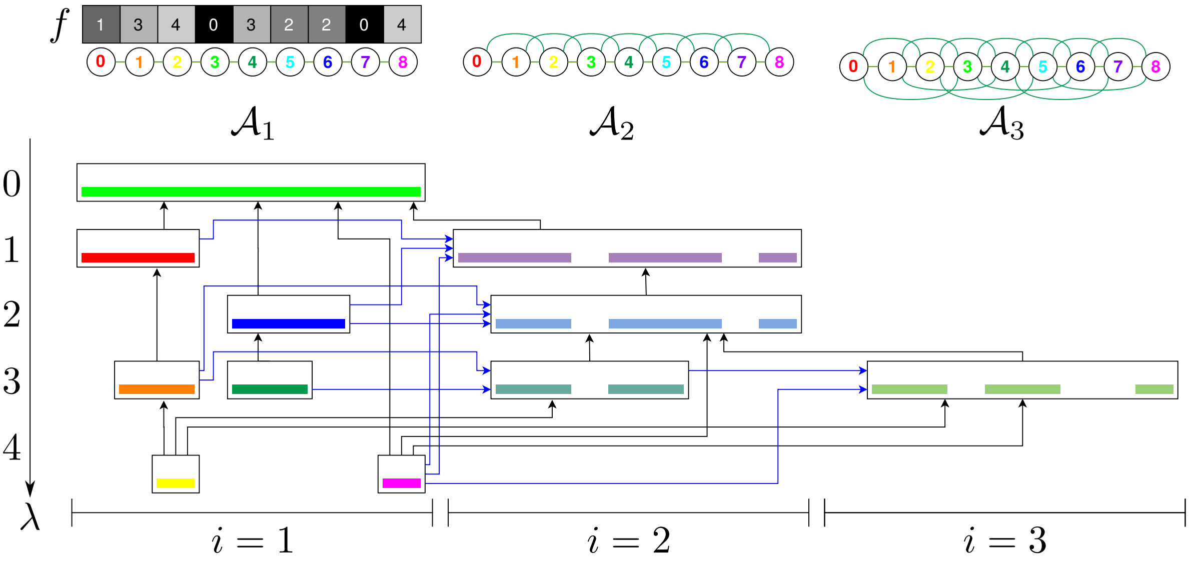


Figure 4.3: *The compact-hypertree of the complete component-hypertree from Fig. 4.2. Black arrows represent parent arcs, while blue arrows represent composite arcs.*

Given a complete hypertree $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ and its respective compact-hypertree $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$, we say that an arc $e \in \mathcal{E}_{CH}$ is a parent (resp. composite) arc if $e = ce(e')$ and $e' \in \mathcal{E}_{CCH}$ is a parent (resp. composite) arc. In Fig. 4.3, parent arcs are drawn as black arrows, while composite arcs are drawn as blue arrows.

It is not difficult to prove the following properties:

Proposition 4.2. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree and $e = (N, N') \in \mathcal{E}_{CH}$ be a parent arc. Then $f(N) > f(N')$.*

Proposition 4.3. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree and $e = (N, N') \in \mathcal{E}_{CH}$ be a composite arc. Then $adj(N) < adj(N')$.*

4.2.3 Equivalence between Complete-Hypertrees and Compact-Hypertrees

Now that compact-hypertrees have been formally defined, we show that, in terms of information stored, compact-hypertrees and complete component-hypertrees are equivalent. For that, we need to prove that all non-compact nodes and non-compact arcs can be recovered only from the information stored in the compact-hypertree.

For that, suppose a gray-level image f and a sequence of increasing symmetric neighborhoods $\mathbb{A} = (\mathcal{A}, \ldots, \mathcal{A}_n)$ is given. Let $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be the complete-hypertree of f and $\mathbb{A}$ and

$\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be the corresponding compact-hypertree. Then, given a node $N \in V_{CH}$, suppose we want to find all nodes $M \in V_{CCH}$ such that $cn(M) = N$.

In component-trees, repeated CCs can be found by comparing the gray-level of N with the gray-level of its parent N_P. In that case, for any $f(N) < \lambda < f(N_P)$, there existed a node N' satisfying $CC(N') = CC(N)$ and $f(N') = \lambda$ that was removed. To give an example, Fig. 4.4 shows the component-trees used in Sec. 2.

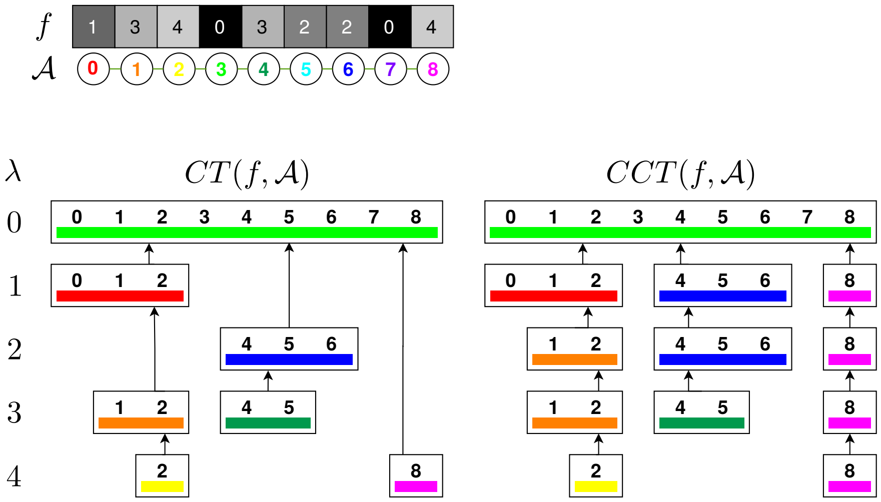


Figure 4.4: *A component-tree and its respective complete component-tree. Note that repeated CCs from the complete representation can be obtained from the (non-complete) component-tree by comparing gray-level of nodes.*

This idea can also be applied to compact-hypertrees, if we know all neighborhoods where $CC(N)$ is allocated and also the parent of N for all these neighborhoods. In this regard, let $par_i(N)$ denote the parent of N at neighborhood index i, that is, $(N, par_i(N))$ is a compact arc where $(CC(N), CC(par_i(N)))$ is a parent arc of the component-tree of f using $\mathcal{A}_i$.

Supposing we know all these values i where $CC(N)$ exist, then we know that all ancestors $N' = (C, \lambda, i)$ of N satisfying $f(N) < \lambda < f(par_i(N))$ satisfy $CC(N') = CC(N)$. An example is provided in Fig. 4.5.

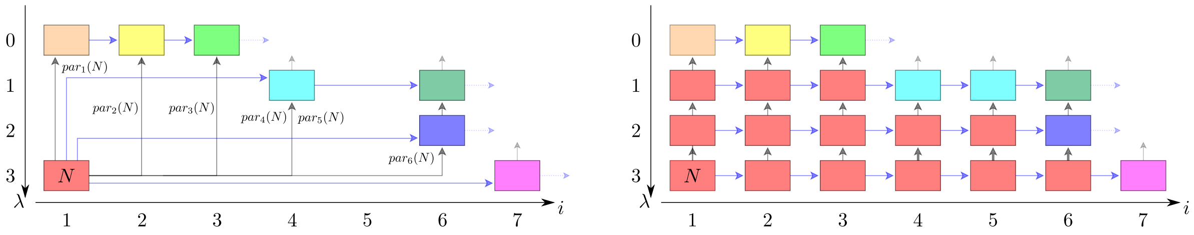


Figure 4.5: *Left: part of a compact-hypertree, that shows all arcs leaving from a node given node N. In particular, the parents of N for different neighborhoods are presented. Right: the respective part of the complete component-hypertree, showing nodes that include $CC(N)$. Nodes with the same color represent the same CC. In this regard, all red nodes represent $CC(N)$ and they can be recovered using the parents of N from the compact representation.*

To find all the values of i where $CC(N)$ is a $\mathcal{A}_i$-CC, an analogous idea can be applied. Within a fixed gray-level $\beta \in \mathbb{K}$, CCs from consecutive neighborhoods are increasing. In this sense, if we organize the CCs of $X_\beta(f)$ (obtained from $\mathbb{A}$) according to their inclusion relation, we would have a component-tree (or component-forest). An example is given in Fig. 4.6.

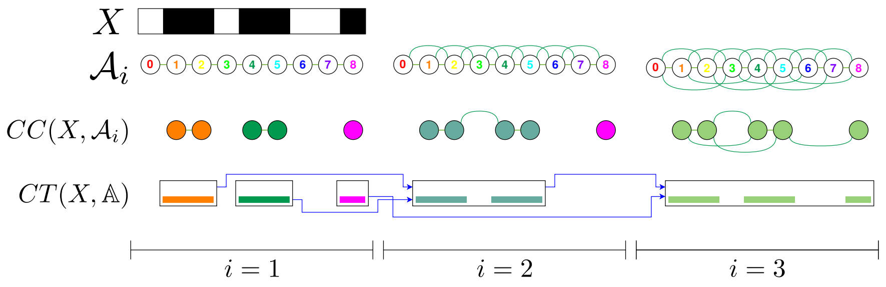


Figure 4.6: *From top to bottom: a binary image X (foreground pixels are dark); a sequence $\mathbb{A} = (\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$; the CCs of X using $\mathbb{A}$; and these CCs organized according to their inclusion relation in a sideways component-tree. Note that the pink CC is repeated, but represented only once, and the non-allocated node can be inferred from the tree by comparing the neighborhood index of the pink node with its "parent", just like a regular component-tree extracted from level sets.*

In this component-tree, given a node M and its "parent" (which is the composite of M according to our definitions) M', we know that $CC(M)$ exists for any index i' satisfying $adj(M) \leq i' < adj(M')$. Then, for all gray-levels $\lambda \in \mathbb{K}$ where $CC(N)$ is a component of $X_\lambda(f)$, we know that nodes representing $CC(N)$ with neighborhood index i' satisfying $adj(N) < i' < adj(comp_\lambda(N))$ were removed from the compact-hypertree (see Fig. 4.7), where $comp_\lambda(N)$ denotes the composite node of N at level λ.

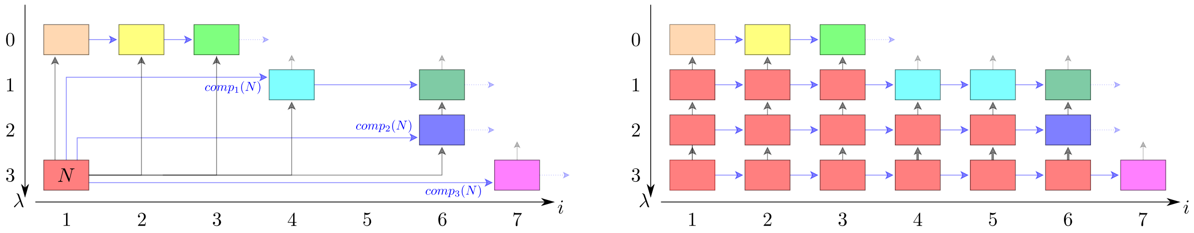


Figure 4.7: *The analogous of Fig. 4.5, but for composite nodes.*

More formally, this property can be stated as the following proposition:

Proposition 4.4. *Let f be a gray-level image, $\mathbb{A}$ a sequence of increasing symmetric neighborhoods, $\mathcal{G}_{CCH} = (\mathcal{V}_{CCH}, \mathcal{E}_{CCH})$ be the complete-hypertree of f and $\mathbb{A}$ and $\mathcal{G}_{CH} = (\mathcal{V}_{CH}, \mathcal{E}_{CH})$ be its respective compact-hypertree. Additionally, let $N \in \mathcal{V}_{CH}$ be a compact node and suppose there exists a node $N' \in \mathcal{V}_{CCH}$ satisfying $cn(N') = N$ and let $f(N') = \lambda$. Then for any $adj(N') < i < adj(comp_\lambda(N))$, there is a node $M = (CC(M), \lambda, i) \in \mathcal{V}_{CCH}$ such that $cn(M) = N$.*

From these ideas, it can be shown that the parents and composites of a compact node N satisfy some properties. First of all, it can be proved that gray-levels of the parent nodes of a fixed node N are increasing, or more specifically:

Proposition 4.5. *Let f be a gray-level image, $\mathbb{A}$ a sequence of increasing symmetric neighborhoods, $\mathcal{G}_{CCH} = (\mathcal{V}_{CCH}, \mathcal{E}_{CCH})$ be the complete-hypertree of f and $\mathbb{A}$ and $\mathcal{G}_{CH} = (\mathcal{V}_{CH}, \mathcal{E}_{CH})$ be its respective compact-hypertree. Then, given a node $N \in \mathcal{V}_{CH}$, for any $1 \leq i < i' \leq n$ such that $par_i(N)$ and $par_{i'}(N)$ exist, $f(par_i(N)) \leq f(par_{i'}(N))$.*

Proof. Suppose by contradiction that there exists $1 \leq i < i' \leq n$ such that $par_i(N)$ and $par_{i'}(N)$ both exist but $f(par_i(N)) > f(par_{i'}(N))$.

Let us analyze the nodes in the context of the complete-hypertree $\mathcal{G}_{CCH}$. Let $M \in \mathcal{V}_{CCH}$ be the node satisfying $CC(par_i(N)) \subseteq CC(M)$ at gray-level $f(par_i(N))$ and neighborhood index i', that is to say, $M = (CC(M), f(par_i(N)), i')$. Using Prop. 2.10, we know that this node exists and is unique and, in particular, this implies $CC(N) \subset CC(par_i(N)) \subseteq CC(M)$.

Since $e = (N, par_{i'}(N)) \in \mathcal{E}_{CH}$, then e is a compact arc of an arc of the complete-hypertree, implying that there exists a node $N' \in \mathcal{V}_{CCH}$ with $adj(N') = i'$ and $CC(N') = CC(N)$. Additionally, by construction, $CC(par_{i'}(N))$ is the parent of $CC(N)$ in the component-tree $CT(f, \mathcal{A}_{i'})$.

Hence, $CC(N)$, $CC(M)$ and $CC(par_{i'}(N))$ are all nodes of $CT(f, \mathcal{A}_{i'})$, and both $CC(M)$ and $CC(par_{i'}(N))$ are ancestors of $CC(N)$ in $CT(f, \mathcal{A}_{i'})$.

On the one hand, since $CC(par_{i'}(N))$ is the parent of $CC(N)$, there is no $\mathcal{A}_{i'}$-CC included between $CC(N)$ and $CC(par_{i'(N)})$, and the only possibility left is that $CC(par_{i'}(N)) \subseteq CC(M)$. On the other hand, since component-tree are decreasing, then $f(M) = f(par_i(N)) > f(par_{i'}(N))$ implies that $CC(M) \subseteq CC(par_{i'}(N))$. The condition $CC(M) = CC(par_{i'}(N))$ can not be true because $par_{i'}(N)$ is a compact node, but M represents the same CC and has higher gray-level.

Finally we conclude that assuming $f(par_i(N)) > f(par_{i'}(N))$ leads both to $CC(par_{i'}(N)) \subset CC(M)$ and $CC(par_{i'}(N)) \supset CC(M)$, which is a contradiction. Hence, $f(par_i(N)) > f(par_{i'}(N))$ can not happen and the proposition is proved. □

The analogous of Prop. 4.5 is shown in Prop. 4.6.

Proposition 4.6. *For any $0 \leq \lambda < \lambda' < K$ such that $comp_\lambda(N)$ and $comp_{\lambda'}(N)$ exist, $adj(comp_\lambda(N)) \leq adj(comp_{\lambda'}(N))$.*

Using these propositions, one can use the following strategy to recover repeated nodes not allocated by the compact-hypertree: given a node N, because of Prop. 4.6, we know $comp_\lambda(N)$ is increasing, in other words, the highest neighborhood index returned by $adj(comp_\lambda(N))$ is obtained from the highest possible value of λ. By definition, N is a compact node and is already allocated at the highest gray-level where $CC(N)$ exists, so the highest possible value for λ is $f(N)$.

Also by definition, we know that N also has the smallest possible neighborhood index where $CC(N)$ exists. Then, we know that nodes representing $CC(N)$ at level $f(N)$ on the complete hypertree exist for all $adj(N) \leq i < adj(comp_{f(N)}(N))$. From that, we have all neighborhood indices i where $CC(N)$ exist, and we know that removed nodes existed, for each of these values of i, in the interval $]f(par_i(N)), f(N)]$. These ideas can be observed in Figs. 4.5 and 4.7 for $f(N) = 3$ and $adj(N) = 1$.

To implement the ideas discussed in this section, it is important to know how to obtain $par_i(N)$ from compact-hypertrees. Let $par(\mathcal{G}_{CH}, N)$ (or simply $par(N)$ when the hypertree used is clear from context) be the set of all parent nodes of N in the compact-hypertree $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$. This means that, for any $N' \in par(N)$, $(N, N') \in \mathcal{E}_{CH}$ is a parent arc. For convenience, suppose that the elements of $par(\mathcal{G}_{CH}, N)$ are organized in an increasing order, based on their neighborhood indices.

In the simplest case, if we want to obtain $par_i(N)$ and there is a node $N' \in par(\mathcal{G}_{CH}, N)$ with $adj(N') = i$, then, $par_i(N) = N'$. However, we can not guarantee that this property is always true because we are storing compact arcs, and if the parent of N in $CT(f, \mathcal{A}_i)$ is originally a non-compact arc, then its compact arc will have neighborhood index lower than i.

To decide which of the parent nodes with neighborhood index lower than i is the correct parent, Prop 4.7 can be applied.

Proposition 4.7. *Let f be a gray-level image, $\mathbb{A}$ a sequence of increasing symmetric neighborhoods, $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be the complete-hypertree of f and $\mathbb{A}$ and $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be its respective compact-hypertree. Additionally, let $N \in V_{CH}$, $1 \leq i \leq n$ and $M, M' \in par(\mathcal{G}_{CH}, N)$ be two compact nodes satisfying $1 \leq adj(M) < adj(M') \leq i$. Then, $par_i(N) \neq M$.*

Proof. First, we note that since $M, M' \in par(\mathcal{G}_{CH}, N)$ and N, M and M' are compact nodes imply that $CC(N) \subset CC(M)$ and $CC(N) \subset CC(M')$.

Then, From Prop. 4.5, we know that $f(M) \leq f(M')$. If $f(M) = f(M')$, then $CC(M)$ and $CC(M')$ are nodes of the (sideways) component-tree of $X_{f(M)}(f)$ using $\mathbb{A}$. In particular, since they intersect at $CC(N)$ and $adj(M) < adj(M')$, then $CC(M')$ is an ancestor of $CC(M)$. We remind that $CC(M)$ can only exist in neighborhood indices i' satisfying $adj(M) \leq i' < adj(comp_{f(M)}(M))$, but since $CC(M')$ is an ancestor of $CC(M)$, then $adj(comp_{f(M)}(M)) \leq adj(M') \leq i$. Thus, $CC(M)$ is a connected-component only in the neighborhood indices in the interval $[adj(M), i[$ and can not represent a $\mathcal{A}_i$-CC, that is, $par_i(N) \neq M$, as desired.

If $f(M) < f(M')$, for $CC(M)$ to be a $adj(M')$-CC, there must exist a node N' in the complete-hypertree satisfying $CC(N') = CC(M)$ and $adj(N') = adj(M')$. In particular, since M is a compact node, the highest gray-level where $CC(M)$ can exist is $f(M)$, implying that $f(N') \leq f(M)$.

Putting everything together, $CC(M')$ and $CC(N')$ are nodes of the component-tree of f using $\mathcal{A}_{adj(M')}$ satisfying $CC(N) \subset CC(M')$ and $CC(N) \subset CC(M) = CC(N')$. Since $f(N') \leq f(M) < f(M')$, then $CC(N')$ must be an ancestor of $CC(M')$, or putting it another way, $CC(N) \subset CC(M') \subset CC(N')$. But in this case, there is no direct arc from $CC(N)$ to $CC(N')$ and $(CC(N), CC(N'))$ is not a parent arc at neighborhood index $adj(M)$, implying $par_i(N) \neq (M)$. □

From Prop. 4.7, we can prove that:

Proposition 4.8. *Let f be a gray-level image, $\mathbb{A}$ a sequence of increasing symmetric neighborhoods, $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be the complete-hypertree of f and $\mathbb{A}$ and $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be its respective compact-hypertree. Suppose $N \in V_{CH}$, $1 \leq i \leq n$ is a neighborhood index where $par_i(N)$ exists and let M be the node of $par(\mathcal{G}_{CH}, N)$ with the highest neighborhood index that satisfies $adj(M) \leq i$. Then, $par_i(N) = M$.*

Proof. On the one hand, to build the compact-hypertree, all parent arcs of the complete-hypertree were mapped into compact (parent) arcs. Hence, the set $par(\mathcal{G}_{CH}, N)$ contains all parents of N, implying $par_i(N) \in par(\mathcal{G}_{CH}, N)$.

On the other hand, a consequence of Prop. 4.7 is that, among all nodes M' of $par(\mathcal{G}_{CH}, N)$ satisfying $adj(M') \leq i$, only the node M in $par(N)$ with the highest neighborhood index can be $par_i(N)$. Additionally, any node M' of $par(\mathcal{G}_{CH}, N)$ satisfying $adj(M') > i$ can not be $par_i(N)$ because, for any of these nodes, $CC(M')$ is not a $\mathcal{A}_i$-CC.

Hence, among all nodes of $par(\mathcal{G}_{CH}, N)$, M is the only choice that does not lead to a contradiction and, therefore, M is the correct node. □

Proposition 4.8 implies that, if we want $par_i(N)$ but there is no node $N' \in par(\mathcal{G}_{CH}, N)$ satisfying $adj(N') = i$, we need to look for the node satisfying $adj(N') \leq i$ with highest neighborhood index. Using a similar approach, one can prove that $comp_\lambda(N)$ is the node $N' \in comp(N)$ with the lowest gray-level $f(N') \geq \lambda$.

Thus, with the ideas presented in this section, it is possible to recover all non-compact nodes of complete-hypertrees. From these nodes N, we can recover all non-compact arcs using $par_i(N)$ and $comp_\lambda(N)$.

Although non-compact nodes and arcs can be recovered, compact component-hypertrees already contain all inclusion relations of the connected components of their respective complete component-hypertrees. This property holds thanks to Prop. 4.9:

Proposition 4.9. *Let $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be a complete component-hypertree and $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be its respective compact-hypertree. Then, given two nodes $N, N' \in V_{CCH}$ such that there is a path from N to N' in $\mathcal{G}_{CCH}$, there is also a path from $cn(N)$ to $cn(N')$ in $\mathcal{G}_{CH}$.*

Proof. For this proposition, we only give an idea of the proof by showing how to transform a path in the complete hypertree to a path in the compact representation.

Let $\pi(\mathcal{G}_{CCH}, N, N') = (N = N_1, \ldots, N_{LP} = N')$ be a path linking N to N' in $\mathcal{G}_{CCH}$. Then, any pair $e_\ell = (N_\ell, N_{\ell+1})$ is an arc of the complete hypertree. To obtain a path from $cn(N)$ to $cn(N')$ in $\mathcal{G}_{CH}$, it suffices to concatenate the arcs $ce(\ell)$ for all $1 \leq \ell < LP$ with $N_\ell \neq N_{\ell+1}$. □

In particular, this property holds even for subgraphs of $\mathcal{G}_{CH}$, as shown by Prop. 4.10.

Proposition 4.10. *Let $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be a complete component-hypertree and $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be its respective compact-hypertree. Suppose $N, N' \in V_{CCH}$ and that there is a path from N to N' in $\mathcal{G}_{CCH}$. Then, there is a path from $cn(N)$ to $cn(N')$ in $\mathcal{G}_{CH}^{([0 \to K-1],[1 \to adj(N')])}$.*

Proof. By construction, if there is a path π from N to N' in $\mathcal{G}_{CCH}$, then $f(N) \geq f(N')$ and $adj(N) \leq adj(N')$. Additionally, any node $M \in \pi$ must satisfy $f(N) \geq f(M) \geq f(N')$ and $adj(N) \leq adj(M) \leq adj(N')$, since any arc in the complete hypertree either decreases the gray-level or increases the neighborhood index.

Hence, if we convert this path π to a path π' in the compact-hypertree, for all nodes $M' \in \pi'$, there exists a node $M \in \pi$ such that $M' = cn(M)$. Since M' is the compact node of M, then $f(M') \geq f(M)$ and $adj(M') \leq adj(M)$, that is, $f(M) \leq f(M') < K$ and $1 \leq adj(M') \leq adj(M)$.

In particular, the node of π with the lowest gray-level is N', and the node of π with highest neighborhood index is also N'. Therefore, all nodes $M' \in \pi'$ satisfy $0 \leq f(N') \leq f(M') < K$ and $1 \leq adj(M') \leq adj(N')$, and all of these nodes M' are in $\mathcal{G}_{CH}^{([0\to K-1],[1\to adj(N')])}$. □

This property is particularly useful if we want to build an algorithm to build compact component-hypertrees. We recall from Sect. 2.7 that the notation $\mathcal{G}_{CH}^{([0\to K-1],[1\to i])}$ refers to a subgraph of the component-hypertree $\mathcal{G}_{CH}$ restricted to the nodes with gray-levels in the interval $[0, K-1]$ and neighborhood indices in the interval $[1, i]$. What we want is an incremental algorithm that receives a gray-level image f and a sequence of neighborhoods $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, and constructs the compact-hypertree $\mathcal{G}_{CH}$ of f and $\mathbb{A}$ as follows: first, we construct $\mathcal{G}_{CH}^{([0\to K-1],[1\to 1])}$, which is the max-tree of f and $\mathcal{A}_1$. Then, the algorithm iterates on the neighborhood indices $2 \leq i \leq n$, where it updates $\mathcal{G}_{CH}^{([0\to K-1],[1\to i-1])}$ to $\mathcal{G}_{CH}^{([0\to K-1],[1\to i])}$. Finally, at the end of step n, we will have built $\mathcal{G}_{CH}^{([0\to K-1],[1\to n])}$, which is precisely the compact-hypertree of f and $\mathbb{A}$.

4.2.4 Arc Redundancy

Although compact-hypertrees present an efficient way of storing CCs without repetition, it turns out that there are still redundant information regarding inclusion relation of CCs in the arcs, that is, even if some of the arcs were removed, all inclusion relations between CCs of the hypertree would still be preserved. In the field of graphs, this operation of removing unneeded arcs is known as transitive reduction.

Some examples are provided in Fig. 4.8, where the dotted red, blue and black arrows represent redundant arcs. They are redundant because there are other longer paths that give the same inclusion relation or, in other words, if they were removed from the compact-hypertree, no inclusion relations between nodes would be lost.

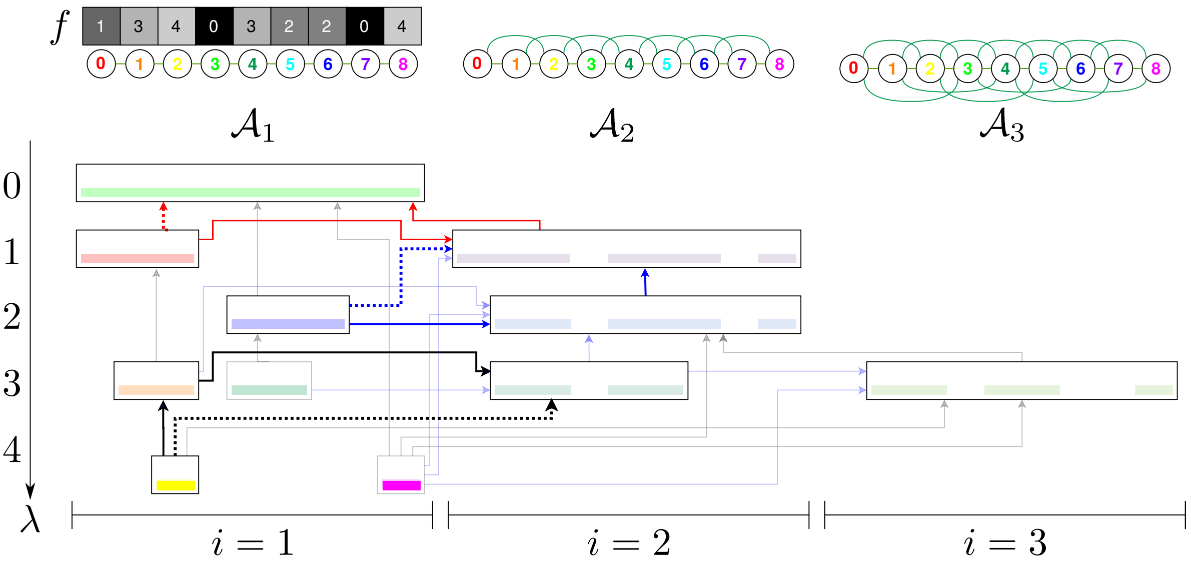


Figure 4.8: *Redundant arcs in compact-hypertrees. Dotted arcs show redundant arcs, while paths with the same color of these arcs show longer paths that gives the same information about inclusion relation. In terms of types of arcs, red arrows are arcs of type 1, blue arrows are arcs of type 2 and black arrows are arcs of type 3.*

Upon inspection, it is possible to find 3 types of redundant arcs in compact-hypertrees. These types are listed below:

- Type 1 (red dotted arc in Fig. 4.8): Redundant arcs $e = (N, N')$ where all paths from N to N' passes through at least one node M with $adj(M) > adj(N)$ and $adj(M) > adj(N')$.
- Type 2 (blue dotted arc in Fig. 4.8): Redundant composite arcs $e = (N, N')$ that are not of type 1, that is, there exists at least one path from N to N' in the compact-hypertree in which, for all nodes M in this path, $adj(M) \leq \max\{adj(N), adj(N')\}$).
- Type 3 (black dotted arc in Fig. 4.8): Redundant parent arcs $e = (N, N')$ that are not of type 1, that is, there exists at least one path from N to N' in the compact-hypertree in which, for all nodes M in this path, $adj(M) \leq \max\{adj(N), adj(N')\}$).

Given a compact-hypertree $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$, we denote by $\mathcal{E}^1_{CH}$ (resp. ^{2} and ^{3}) the set of redundant arcs of $\mathcal{G}_{CH}$ of type 1 (resp. 2 and 3).

An implication of these definitions is presented in Prop. 4.11:

Proposition 4.11. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree and $e \in \mathcal{E}_{CH}$ be an arc. Then, if $e \in \mathcal{E}^2_{CH}$ or $e \in \mathcal{E}^3_{CH}$, there is a path $\pi(\mathcal{G}_{CH}, N, N') = (N = N_1, \ldots, N_{LP} = N')$ from N to N' where $adj(N_\ell) \leq \max\{adj(N), adj(N')\}$, for all $1 \leq \ell \leq LP$.*

In other words, if an arc $e = (N, N')$ is of type 2 or 3, the longer path linking N to N' is restricted to $\mathcal{G}_{CH}^{([0 \to K-1],[1 \to i'])}$, where $i' = \max\{adj(N), adj(N')\}$. Hence, if we follow the strategy of developing an algorithm for component-hypertree construction that iterates on the neighborhood indices $1 \leq i \leq n$ and allocates only compact nodes N satisfying $adj(N) \leq i$ and arcs that link allocated nodes, then at the end of step i' it is already possible to know if the arc $e = (N, N')$ is a redundant arc of type 2 or 3. For arcs of type 1, any longer path linking N to N' passes through an arc M where $adj(M) > i'$, and it is not possible to know, at the end of step i', that e is a redundant arc. For this reason, in the proposed algorithm for component-hypertree construction that is presented later in this chapter, we opted to keep redundant arcs of type 1 in the component-hypertree, since removing them would require a post-processing step at the end of every step i, to check if arcs allocated in previous steps became redundant.

4.2.5 Properties of Compact Arcs

To better understand which arcs are redundant, we now analyze some properties of compact arcs, In particular, arcs of compact-hypertree can be divided into 4 types of arcs:

1. Vertical arcs: parent arcs $e = (N, N')$ where $adj(N) = adj(N')$.
2. Backward arcs: parent arcs $e = (N, N')$ where $adj(N) > adj(N')$.
3. Horizontal arcs: composite arcs $e = (N, N')$ where $f(N) = f(N')$.
4. Diagonal arcs: arcs $e = (N, N')$ where $adj(N) < adj(N')$ and $f(N) > f(N')$.

An illustration showing the types of arcs of compact-hypertrees is provided in Fig. 4.9

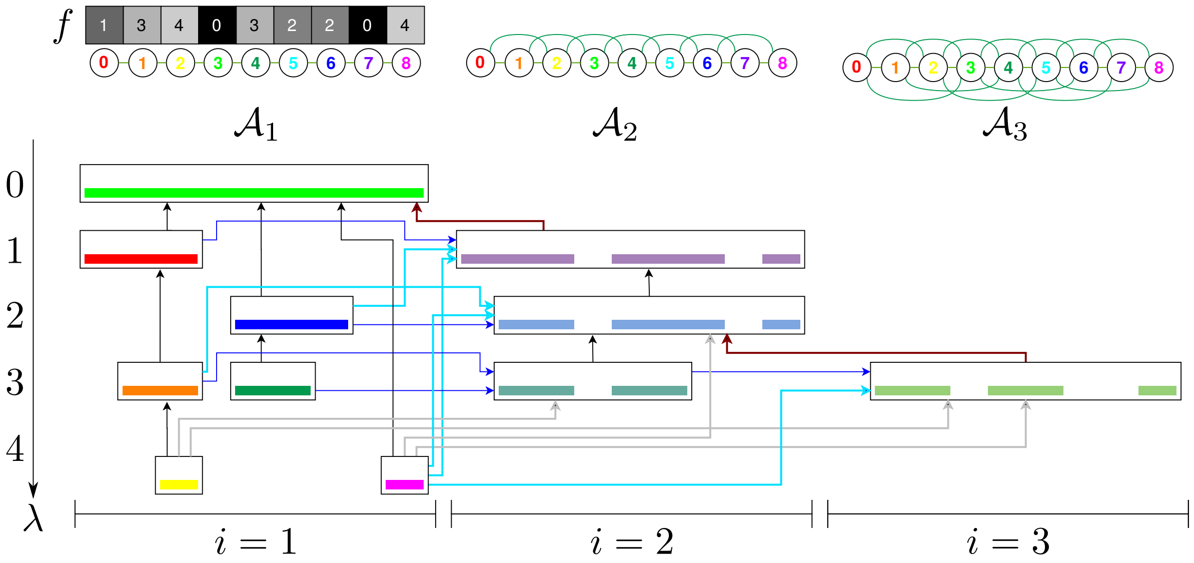


Figure 4.9: *Four types of arcs in compact-hypertrees: vertical arcs (in black), backward arcs (in brown), horizontal arcs (in dark blue) and diagonal arcs (diagonal parent arcs are displayed in gray, while diagonal composite arcs are drawn in light blue).*

There are no other types of arcs in compact-hypertrees. To prove that, we show below that the existence of any other type of arc would lead to a contradiction.

Proposition 4.12. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree. Then, there is no arc $e = (N, N') \in \mathcal{E}_{CH}$ with $f(N) = f(N')$ and $adj(N) = adj(N')$.*

Proof. Suppose by contradiction that e exists. Then, $CC(N) \subset CC(N')$, $f(N) = f(N')$ and $adj(N) = adj(N')$. However, this implies that both $CC(N)$ and $CC(N')$ are $\mathcal{A}_{adj(N)}$-CCs of $X_{f(N)}(f)$ with $CC(N) \subset CC(N')$. Since the elements of any set of CCs of any level sets are disjoint, this is not possible (in fact, this implies that $CC(N)$ is a not even a valid connected component, since it does not include elements in $CC(N') \setminus CC(N)$ and, therefore, is not a maximal set of connected pixels). Hence, this arc e can not exist. □

Proposition 4.13. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree. Then, there is no arc $e = (N, N') \in \mathcal{E}_{CH}$ with $f(N) = f(N')$ and $adj(N) > adj(N')$.*

Proof. Suppose by contradiction that the arc e exists. On the one hand, this implies that $CC(N) \subset CC(N')$.

On the other hand, let us consider the inclusion relations between $CC(N)$ and $CC(N')$ without considering the arc e. There are two possibilities for $CC(N)$ and $CC(N')$: either $CC(N) \cap CC(N') = \emptyset$ or $CC(N) \cap CC(N') \neq \emptyset$. Note that $CC(N) \cap CC(N') = \emptyset$ contradicts $CC(N) \subset CC(N')$ immediately, and there is nothing more to prove in this case.

If $CC(N) \cap CC(N') \neq \emptyset$, then there is a pixel $p \in CC(N) \cap CC(N')$. By definition, every pixel $q \neq p$ in $CC(N')$ is $\mathcal{A}_{adj(N')}$-connected to p. However, since $adj(N') < adj(N)$ and the neighborhoods are increasing, this also implies that all of them are also $\mathcal{A}_{adj(N)}$-connected to p, or more specifically, $CC(N') \subseteq CC(N)$. Note that this also contradicts $CC(N) \subset CC(N')$.

Hence, in either cases, we have a contradiction because $CC(N) \subset CC(N')$ can not be satisfied and, therefore, this arc e can not exist. □

Proposition 4.14. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree. Then, there is no arc $e = (N, N') \in \mathcal{E}_{CH}$ with $f(N) < f(N')$.*

Proof. Suppose by contradiction that the arc e exists. On the one hand, this implies that $CC(N) \subset CC(N')$.

On the other hand, by definition, $CC(N)$ is a $\mathcal{A}_{adj(N)}$-CC of $X_{f(N)}(f)$ and N is a compact node. This means that $f(N)$ is the highest gray-level where $CC(N)$ exists or, in other words, there exists at least one pixel $p \in CC(N)$ satisfying $f(p) = f(N)$.

Using the same argument, $CC(N')$ must be a $\mathcal{A}_{adj(N')}$-CC of $X_{f(N')}(f)$. By definition, this implies that any $p' \in CC(N')$ satisfies $f(p') \geq f(N')$. Since $f(N) < f(N')$, any pixel $p \in CC(N)$ with $f(p) = f(N)$ is not in $CC(N')$, which implies that $CC(N) \not\subseteq CC(N')$, which contradicts the property $CC(N) \subset CC(N')$.

What led to this contradiction was the assumption that the arc e existed. Hence, there are no such arcs e in the compact-hypertree. □

Propositions 4.12 to 4.14 cover all other possible cases of compact arcs and, therefore, the only types of compact arcs existing in compact-hypertrees are the ones presented in Fig. 4.9.

With that in mind, let us show some important properties of the arcs that actually exist in compact-hypertrees. These properties are important in the development of the algorithm for hypertree-construction.

Proposition 4.15. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree. Then, any node $N \in V_{CH}$ has at most one vertical arc leaving from N.*

Proof. Note that vertical arcs are parent arcs that link two nodes with the same neighborhood index. Hence, if there were two vertical arcs (N, N') and (N, N''), that would imply that N has two different parents at neighborhood index $adj(N)$, which is not possible because that would imply that N has two different parents in the component-tree using the neighborhood $\mathcal{A}_i$. □

Proposition 4.16. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree. Then, any node $N \in V_{CH}$ has at most one horizontal arc leaving from N.*

Proof. The proof is similar to the last proposition and will be omitted. □

Proposition 4.17. *Let $\mathcal{G}_{CH}$ be a compact-hypertree. Then, in $\mathcal{G}_{CH}^{([0 \to K-1],[1 \to i])} = (V_{CH}^{1 \to i}, \mathcal{E}_{CH}^{1 \to i})$, any node N that is not a root and satisfies $adj(N) = i$ has exactly one parent node and no composite nodes.*

Proof. It is clear that N has no composite nodes in $\mathcal{G}_{CH}^{([0 \to K-1],[1 \to i])}$ because for all composite arcs $e = (N, N')$, $adj(N') > adj(N) = i$, which implies that $N' \notin V_{CH}^{1 \to i}$.

In the case of the parent node, we first note that, if N is a compact node and is not a root, then there is always a node N' with $adj(N') = adj(N)$ such that $CC(N) \subset CC(N')$ (in the worst case scenario, N' is a root itself). Hence, N has at least one parent.

Suppose that N has more than one parent. Since N is a compact node, then N does not belong to $\mathcal{G}_{CH}^{([0 \to K-1],[1 \to i-1])}$ or, in other words, there are no nodes M in the complete hypertree $\mathcal{G}_{CCH}^{([0 \to K-1],[1 \to i-1])}$ where $CC(M) = CC(N)$. Hence, any (compact) parent arc $e \in E$ leaving from N in the compact-hypertree must be obtained from an parent arc from $\mathcal{G}_{CCT}^{i} = \mathcal{G}_{CCH}^{([0 \to K-1],[i \to i])}$, which is the complete component-tree using neighborhood $\mathcal{A}_i$.

If there were two or more parent arcs leaving from N, this means that there are multiple paths linking N to the root node in $\mathcal{G}_{CCT}^{i}$, that is, $\mathcal{G}_{CCT}^{i}$ would not be a (component) tree, which is a contradiction. Therefore, N has exactly one parent node. □

4.3 Minimal-Hypertrees

Ideally, in an implementation focused on saving the most possible amount of memory, we would like to remove all types of redundant arcs. However, this optimization would not be as useful, neither in time consumption nor in memory allocation, if we first allocate the arcs and then remove them later. The problem with this approach is that the removal step would take time and we would still use extra memory to allocate redundant arcs before removing them.

To have an efficient algorithm, we need an implementation that can efficiently allocate arcs only if they are not redundant. In practice, due to the way the hypertree construction algorithm is designed, it is not possible to predict all types of redundant arcs. This happens because the proposed algorithm is designed to iterate on the neighborhood indices and never remove nodes or arcs. We recall that, for every redundant arc there is a longer path that gives the same information regarding the inclusion relation. In particular, if $e = (N, N')$ is a redundant arc of type 1, any longer path linking N to N' will pass through a node M with $adj(M) > \max\{adj(N), adj(N')\}$. In other words, we can only know that e is a redundant arc at the step $i = adj(M)$, but the arc e needs to be allocated at step $i' = \max\{adj(N), adj(N')\} < adj(M)$, otherwise, the partially constructed hypertree allocated at step i' would have this inclusion relation missing. For this reason, redundant arcs of type 1 are allocated by our algorithm, but since redundant arcs of types 2 and 3 do not have this problem, it is possible to not allocate redundant arcs of types 2 and 3. Thus, we say that an arc $e \in \mathcal{E}_{CH}$ is a minimal arc if and only if $e \notin \mathcal{E}^2_{CH} \cup \mathcal{E}^3_{CH}$.

The graph $\mathcal{G}_{MH} = (V_{CH}, \mathcal{E}_{MH})$, where $\mathcal{E}_{MH} = \mathcal{E}_{CH} \setminus \left(\mathcal{E}^2_{CH} \cup E^3_{CH}\right)$ is called the minimal-hypertree of $\mathcal{G}_{CH}$. An example is provided in Fig. 4.10.

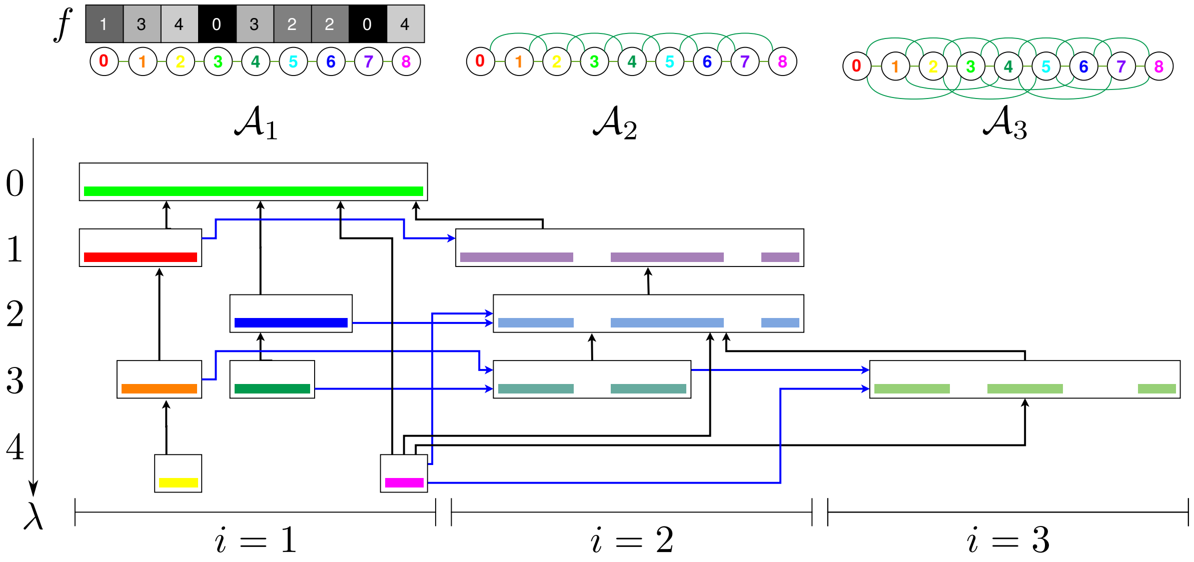


Figure 4.10: *The minimal-hypertree of f and $\mathbb{A}$.*

It is plain that a minimal-hypertree keeps all inclusion relations between nodes of the compact-hypertree, since only arcs that give redundant information about inclusion relation are removed. In turn, this implies the inclusion relations of the CCs of complete hypertrees are also preserved. Hence, minimal-hypertrees are **??**

To efficiently compute these minimal representations, we need a way of characterizing minimal and non-minimal arcs. According to the properties that non-minimal arcs satisfy, it is possible to detect them and create an efficient algorithm for minimal-hypertree construction that does not allocate them.

4.4 Which Compact Arcs are Minimal Arcs

Using the properties from the previous sections, we can categorize each type of compact arc as a minimal or a non-minimal arc.

4.4.1 Vertical Arcs

All vertical arcs of compact-hypertrees are minimal arcs. We prove this property below:

Proposition 4.18. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ and $e = (N, N') \in \mathcal{E}_{CH}$ be a vertical arc (that is to say, $adj(N) = adj(N')$). Then, e is a minimal arc.*

Proof. Suppose by contradiction that e is not minimal. Then, using Prop. 4.11, there should be a path $\pi = \pi(\mathcal{G}_{CH}, N, N') = (N = N_1, \ldots, N_{LP} = N')$ from N to N' where $\pi \neq e$ and, for all nodes N_ℓ of this path π, $adj(N_\ell) \leq \max\{adj(N), adj(N')\} = adj(N) = adj(N')$.

From this property, we know that (N_1, N_2), the first arc in the path π, can not be a composite arc, since that would imply $adj(N_2) > adj(N_1) = adj(N)$. For the same reason, it can not be a diagonal parent arc either.

Hence, the first arc (N_1, N_2) of the path must be a parent node where $adj(N_2) \leq adj(N_1)$. However, thanks to Prop. 4.17, N has only one parent node with neighborhood index less than or equal to $adj(N)$ which, in this case, can only be the arc e.

Hence, every path that connects N to N' must start with the arc e. Since $\mathcal{G}_{CH}$ is a DAG, there is no possible way to continue this path and return back to N'. Hence, we conclude that $\pi = e$, which is a contradiction. Therefore, e must be a minimal arc. □

4.4.2 Backward Arcs

All backwards arcs of compact-hypertrees are minimal arcs. More formally:

Proposition 4.19. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ and $e = (N, N') \in \mathcal{E}_{CH}$ be a backward arc (that is to say, $adj(N) > adj(N')$ and $f(N) > f(N')$). Then, e is a minimal arc.*

Proof. Analogous to the previous case. □

4.4.3 Horizontal Arcs

All horizontal arcs of hypertrees are minimal arcs. We prove this property below.

Proposition 4.20. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ and $e = (N, N') \in \mathcal{E}_{CH}$ be a horizontal arc (that is to say, $f(N) = f(N')$). Then, e is a minimal arc.*

Proof. Suppose that e is not minimal. Then, using Prop. 4.11, there should be a path $\pi = \pi(\mathcal{G}_{CH}, N, N') = (N = N_1, \ldots, N_{LP} = N')$ from N to N' where $\pi \neq e$ and, for all nodes N_ℓ of this path π, $adj(N_\ell) \leq \max\{adj(N), adj(N')\} = adj(N')$.

From this property, we know that (N_1, N_2), the first arc in the path π, can not be a parent arc, since that would imply $f(N_2) < f(N_1) = f(N) = f(N')$, and there is path from N_2 to N' because there are no arcs in the compact-hypertree that increase the gray-level (Prop. 4.14). For the same reason, it can not be a diagonal arc either.

Hence, the first arc (N_1, N_2) of the path must be a composite arc where $f(N_1) = f(N_2)$. However, thanks to Prop. 4.16, N has at most one horizontal arc which, in this case, can only be the arc e.

Hence, every path that connects N to N' must start with the arc e. From this point, since $\mathcal{G}_{CH}$ is a DAG, there is no path from N' back to N, and we conclude that $\pi = e$, which is a contradiction. Therefore, e must be a minimal arc. □

4.4.4 Diagonal Arcs

In this case, whether or not a diagonal arc is a minimal arc depends on some additional properties.

First, we investigate when a parent diagonal arc exists in compact-hypertrees.

Proposition 4.21. *Let $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be a complete hypertree and $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be its compact representation. Suppose $e = (N, N') \in \mathcal{E}_{CH}$ is a parent diagonal arc. Then, there is a parent arc $e' = (M, N') \in \mathcal{E}_{CCH}$ where $cn(M) = N$.*

Proof. Let $\pi = \pi(\mathcal{G}_{CCH}, N, N') = (N = N_1, \ldots, N_{LP} = N')$ be a path from N to N' in the complete hypertree that ends with a parent arc, or more specifically, $e_{LP-1} = (N_{LP-1}, N_{LP}) \in \mathcal{E}_{CCH}$ is a parent arc. If $CC(N_{LP-1}) = CC(N)$, then $cn(N_{LP-1}) = N$, and $e' = e_{LP-1}$ is the desired arc.

Suppose that $CC(N_{LP-1}) \neq CC(N)$. Then, since paths in the complete hypertrees follow an inclusion relation, we have that $CC(N) \subseteq CC(N_{LP-1}) \subseteq CC(N')$, but we assumed $CC(N_{LP-1}) \neq CC(N)$ and $CC(N_{LP-1}) \neq CC(N')$ because assuming $CC(N_{LP-1}) = CC(N')$ would contradict the fact that N' is a compact node. Hence, $CC(N) \subset CC(N_{LP-1}) \subset CC(N')$. An implication of this result is that $CC(N)$ is not a $\mathcal{A}_i$-CC of $X_\lambda(f)$ for any $i \geq adj(N_{LP-1}) = adj(N')$ and any $0 \leq \lambda \leq f(N_{LP-1}) = f(N') + 1$ (if it was, then we would have $CC(N) \subset CC(N_{LP-1}) \subseteq CC(N)$, which is impossible).

On the other hand, the existence of e implies that there should exist a parent arc $e'' = (M, M') \in \mathcal{E}_{CCH}$ such that $ce(e'') = e$ which, in turn, implies $cn(M') = N'$. Since N' is a compact node, then $f(M') \leq f(N')$ and $adj(M') \geq adj(N')$ and, since e'' is a parent arc, then $f(M) \leq f(N') + 1$ and $adj(M) = adj(M') \geq adj(N')$. However, as explained above, $CC(N)$ is not a valid CC under these parameters, so $cn(M) \neq N$. Hence, there is no parent arc $e'' \in \mathcal{E}_{CCH}$ that satisfies $ce(e'') = e$.

In summary, assuming $CC(N_{LP-1}) \neq CC(N)$ implies that $e \notin \mathcal{E}_{CH}$, which is a contradiction. Therefore, we conclude that $CC(N_{LP-1}) = CC(N)$, and $e' = (N_{LP-1}, N') = (M, N')$. □

Using a similar proof, one can show the analogous property from diagonal composite arcs:

Proposition 4.22. *Let $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be a complete hypertree and $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be its compact representation. Suppose $e = (N, N') \in \mathcal{E}_{CH}$ is a composite diagonal arc. Then, there is a composite arc $e = (M, N') \in \mathcal{E}_{CCH}$ where $cn(M) = N$.*

An example showing a configuration of the complete-hypertree that generates a diagonal composite arc is given in Fig. 4.11.

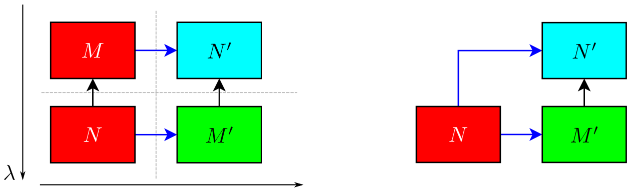


Figure 4.11: *Occurence of diagonal composite arcs. On the left, we show the configuration of a complete-hypertree that generates a diagonal composite arc, where nodes with the same color represent the same CC. The respective compact-hypertree is shown on the right.*

Note that these two propositions are not exclusive, so it is possible to have a diagonal arc in compact-hypertrees that is both a parent and a composite arc. When this happens, we call it a double arc. An example is given in Fig. 4.12.

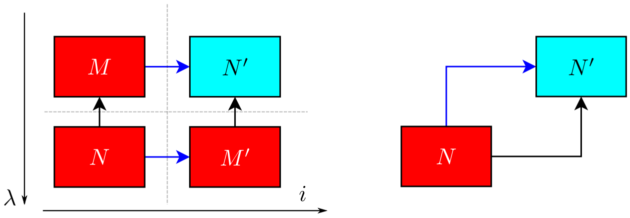


Figure 4.12: *Occurence of diagonal double arcs. On the left, we show the configuration of a complete-hypertree that generates a diagonal double arc, where nodes with the same color represent the same CC. The respective compact-hypertree is shown on the right.*

A direct consequence of the definition of double arcs is stated in the proposition below:

Proposition 4.23. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree and $e = (N, N') \in \mathcal{E}_{CH}$ a double arc. Then, for any other arc $e' = (N, M) \in \mathcal{E}_{CH}$, $f(M) < f(N')$ or $adj(M) > adj(N')$.*

Double arcs are very important because of the following property:

Proposition 4.24. *Let $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ be a compact-hypertree and $e = (N, N') \in \mathcal{E}_{CH}$ a diagonal arc. Then, e is minimal if and only if e is a double arc.*

Proof. There are two parts to this proof:

1. If e is not a double arc, then e is not minimal.

   For this proof, let $\mathcal{G}_{CCH} = (V_{CCH}, \mathcal{E}_{CCH})$ be the complete hypertree of $\mathcal{G}_{CH}$ and suppose e is a diagonal parent arc but not a diagonal composite arc. Then, there is no composite arc $e' = (M, N') \in \mathcal{E}_{CCH}$ such that $ce(e') = e$, or, in other words, for any path $\pi = \pi(\mathcal{G}_{CCH}, N, N') = (N = M_1, \ldots, M_{LQ} = N')$ from N to N' in the complete hypertree that ends with a composite arc, $CC(M_{LQ-1}) \neq CC(N)$, since $CC(M_{LQ-1}) = CC(N)$ would imply $e'' = (M_{LQ-1}, M_{LQ} = N')$ is a composite arc of the complete-hypertree where $ce(e'') = e$. Additionally, since N' is a compact node, we also know that $CC(M_{LQ-1}) \neq CC(N')$.

   The fact that the elements N, M_{LQ-1}, N' are in a path implies that $CC(N) \subset CC(M_{LQ-1}) \subset CC(N')$ or, in other words, there is a path from N to M_{LQ-1} and from M_{LQ-1} to N' in the complete hypertree. As explained before, inclusion relations in compact-hypertree are preserved, so this implies that there is also a path from N to $cn(M_{LQ-1})$ and a path from $cn(M_{LQ-1})$ to N' in $\mathcal{G}_{CH}$ and, therefore, the combination of these two paths makes e a non-minimal arc.

   The arguments when e is only a diagonal composite arc are similar and will be omitted.

2. If e is a double arc, then e is minimal.

   Thanks to Prop. 4.23, any path starting from N that does not use the arc e either goes to a node M with $adj(M) > adj(N')$ or $f(M) < f(N')$. In the first case, a node with $adj(M) > \max\{adj(N), adj(N')\}$ was reached and, even if there is a path from M to N' now, this only implies that e is a redundant node of type 1, but that still means that e is a minimal arc.

   In the second case, if $f(M) < f(N')$, then there is no arc that goes to a node with higher gray-level (Prop. 4.14), so there is no possible path from M to N', and e is also a minimal arc in this case.

   Hence, we conclude that e is minimal.

□

4.5 Algorithm for Minimal Hypertree Construction

Now that the definitions and properties of minimal arcs are presented, we focus on developing an efficient algorithm for minimal-hypertree construction.

As explained before, the subgraph $\mathcal{G}_{MH}^{([0\to K-1],[1\to i])}$ contains all inclusion relations of CCs existing in $\mathcal{G}_{CCH}^{([0\to K-1],[1\to i])}$, for any $1 \leq i \leq n$. Thus, given a gray-level image f and a sequence of increasing symmetric neighborhoods $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, a way of building the minimal-hypertree consists of updating, for each $1 \leq i \leq n$, the subgraph $\mathcal{G}_{MH}^{([0\to K-1],[1\to i-1])}$ to $\mathcal{G}_{MH}^{([0\to K-1],[1\to i])}$ by allocating the corresponding new compact nodes and minimal arcs. Then, at the end of step n, $\mathcal{G}_{MH}^{[0\to K-1],[1\to n]} = \mathcal{G}_{MH}$ has all inclusion relations of $\mathcal{G}_{CCH}$, as desired.

Fast ways of computing parent_i were already presented at Chap. 3. Hence, we now focus on allocating the nodes and arcs of the minimal-hypertree. First, we focus on efficient allocation of compact nodes.

4.5.1 Algorithm for Compact Node Allocation

To allocate compact nodes efficiently, we need a fast algorithm that can detect "new" nodes, or more specifically, for every step $1 \leq i \leq n$, we want to find nodes that represent $\mathcal{A}_i$-CC that are not $\mathcal{A}_{i-1}$-CC.

Allocating nodes at step $i = 1$ is easy since, at the end of the step, the minimal-hypertree is simply a component-tree, and all nodes are "new" nodes, since there were no nodes allocated before that. Any algorithm for component-tree allocation can be used here and, in essence, we simply allocate one node for each canonical element of $\mathsf{parent} = \mathsf{parent}_1$.

For steps $i > 1$, allocating a new node for each canonical element of $\mathsf{parent} = \mathsf{parent}_i$ is not a good strategy, because there could exist a node with neighborhood index $i - 1$ that already represents the same CC. Hence, we need an efficient way of detecting repeated and new nodes in the parent array. In this text, the proposed approach is to change the PARENTUPDATE procedure slightly to mark occurrences of new nodes. For reference, the PARENTUPDATE procedure is presented again below as Alg. 15.

Algorithm 15 The PARENTUPDATE procedure from Chap. 3.

```
procedure PARENTUPDATE($f$, parent, $cur$, $other$)
    $r_{cur} \leftarrow$ FINDREP($f$, parent, $cur$, $f(other)$);          ▷ $f(other) \leq f(cur)$
    $r_{other} \leftarrow$ FINDREP($f$, parent, $cur$, $f(other)$);
    if $r_{cur} \neq r_{other}$ then                                ▷ If $cur$ and $other$ are disconnected
        $par \leftarrow parent[r_{cur}]$;                              ▷ Previous parent of $r_{cur}$
        parent$[r_{cur}] \leftarrow r_{other}$;              ▷ Corrects the interval $[f(par) + 1, f(r_{other})]$
        PARENTUPDATE($f$, parent, $r_{other}$, $par$);   ▷ Recursion to correct gray-levels $\lambda \leq f(par)$.
```

The first modification that we need to do is to change PARENTUPDATE to visit all canonical elements that can possibly represent new nodes. This first modification is shown in Alg. 16.

Algorithm 16 Connect procedure, a modification of PARENTUPDATE that visits all canonical elements that can represent new nodes.

```
 1: procedure CONNECT($f$, parent, $cur$, $other$)                  ▷ Assumes $f(cur) \geq f(other)$
 2:     $r_{cur} \leftarrow$ FINDREP($f$, parent, $cur$, $f(cur)$);
 3:     $r_{other} \leftarrow$ FINDREP($f$, parent, $other$, $f(other)$);
 4:     if $r_{cur} \neq r_{other}$ then
 5:         $par \leftarrow$ parent$[r_{cur}]$;
 6:         if $par \neq r_{other}$ then          ▷ If $parR = otherR$, $curR$ and $otherR$ are connected
 7:             if $f(par) \geq f(r_{other})$ then                          ▷ Assumes $f(\bot) = -1$
 8:                 CONNECT($f$, parent, $par$, $r_{other}$);
 9:             else
10:                 parent$[r_{cur}] \leftarrow r_{other}$;
11:                 CONNECT($f$, parent, $r_{other}$, $par$);               ▷ Skips one recursive call
```

The main difference of Alg. 16 compared to PARENTUPDATE is that this version sets r_{cur_k} as the representative of cur_k at level $f(cur_k)$ instead of $f(other_k)$. In this sense, the r_{cur_k} that PARENTUPDATE returns is found by using multiple calls of CONNECT at Line 8: while we do not have the correct element, we make recursive calls of CONNECT setting $cur_{k+1} = par_k$, until we have $f(\mathsf{parent}[r_{cur_k}]) < f(other_r)$, which is the element returned by PARENTUPDATE. At this point, the algorithm behaves exactly like Alg. 15, and it updates the parenthood relationship. An example is given in Fig. 4.13.

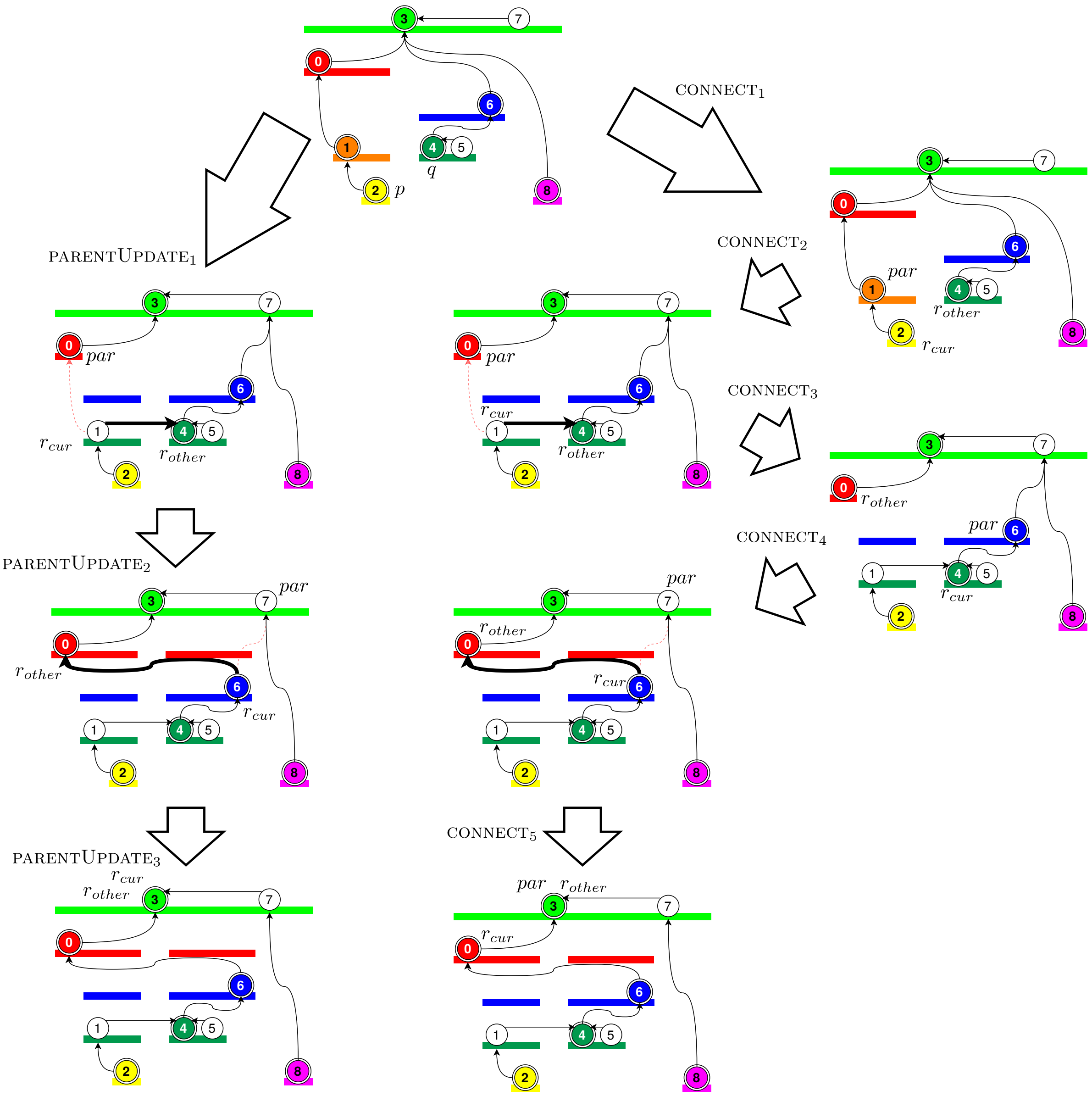


Figure 4.13: *Comparison between the* PARENTUPDATE *procedure and the* CONNECT *procedure.*

Suppose that Alg. 16 receives an array $(\mathsf{parent}_i)^{j-1}$ and updates it to an array $(\mathsf{parent}_i)^j$. Then, compared to PARENTUPDATE, we have the additional property that we visit all canonical elements in the paths $\pi((\mathsf{parent}_i)^{j-1}, p_j, c)$ and $\pi((\mathsf{parent}_i)^{j-1}, q_j, c)$, with the exception of c, the first common ancestor of p_j and q_j. Note that this does not happen in Alg. 13, where we skip directly to the correct representative.

In other words, for any canonical element r in these paths, there is always a step k where $r_{cur_k} = r$. For component-tree computation purposes, this property is not needed and may slow down the algorithm. However, it can be useful for finding all the $\mathcal{A}^j_{alg}$-CCs that were not $\mathcal{A}^{j-1}_{alg}$-CCs.

We recall from Prop. 2.7 that all new CCs are merges of two disjoint CCs, one containing p_j, and the other containing q_j. From the properties of the algorithms, we know that the new CCs are restricted to gray-levels $f(q_j) \leq \lambda < f(c)$, where $f(p_j) \geq f(q_j)$ and c is the first common ancestor of p_j and q_j. Using these properties, we obtain Prop. 4.25.

Proposition 4.25. *Let $(\mathsf{parent}_i)^{j-1}$ and $(\mathsf{parent}_i)^j$ be two arrays where $(\mathsf{parent}_i)^j$ is obtained by updating $(\mathsf{parent}_i)^{j-1}$ calling* CONNECT *for the pair $\{p_j, q_j\} \in \mathcal{A}_i$. Additionally, assume that p_j and q_j are not comparable in $(\mathsf{parent}_i)^{j-1}$ and let c be the first common ancestor of p_j and q_j. Then, all nodes of $(\mathsf{parent}_i)^j$ that represent CCs that did not exist in $(\mathsf{parent}_i)^{j-1}$ are canonical elements r of $(parent_i)^j$ satisfying $f(q_j) \leq f(r) < f(c)$ and $p, q \in rec((\mathsf{parent}_i)^j, r)$.*

In fact, it can be proved that all of these elements represent new CCs.

Proposition 4.26. *Let $(\mathsf{parent}_i)^{j-1}$ and $(\mathsf{parent}_i)^j$ be two arrays where $(\mathsf{parent}_i)^j$ is obtained by updating $(\mathsf{parent}_i)^{j-1}$ calling* CONNECT *for the pair $\{p_j, q_j\} \in \mathcal{A}_i$. Additionally, assume that p_j and q_j are not comparable in $(\mathsf{parent}_i)^{j-1}$ and let c be the first common ancestor of p_j and q_j. Then, all canonical elements r of $(\mathsf{parent}_i)^j$ satisfying $f(q_j) \leq f(r) < f(c)$ and $p_j, q_j \in rec((\mathsf{parent}_i)^j, r)$ represent new nodes.*

Proof. In $(\mathsf{parent}_i)^{j-1}$, canonical elements that represented nodes containing both p_j and q_j need to contain both p_j and q_j in their subtrees or, in other words, they need to be common ancestors of p_j and q_j. Since c is the first common ancestor, all other canonical element r' of $(\mathsf{parent}_i)^{j-1}$ satisfying $p_j, q_j \in rec((\mathsf{parent}_i)^{j-1}, r')$ are ancestors of c, which implies that $f(r') \leq f(c)$. Thus, for all canonical elements r of $(\mathsf{parent}_i)^j$ that satisfies $f(q_j) \leq f(r) < f(c)$ and $p, q \in rec((\mathsf{parent}_i)^j, r)$ represents a new CC, since there were no nodes in this range of gray-levels that contained both p_j and q_j in $(\mathsf{parent}_i)^{j-1}$. □

To find canonical elements of $(\mathsf{parent}_i)^j$ that satisfies the conditions of Prop. 4.26, we can actually look at the canonical elements of $(\mathsf{parent}_i)^{j-1}$ that are visited during a call of the CONNECT procedure. But before proving that, we need an additional property about the CONNECT procedure:

Proposition 4.27. *Let $(\mathsf{parent}_i)^{j-1}$ and $(\mathsf{parent}_i)^j$ be two arrays where $(\mathsf{parent}_i)^j$ is obtained by updating $(\mathsf{parent}_i)^{j-1}$ calling* CONNECT *for the pair $\{p_j, q_j\} \in \mathcal{A}_i$. If $(\mathsf{parent}_i)^j[p] \neq (\mathsf{parent}_i)^{j-1}[p]$ for a certain pixel p, then $f((\mathsf{parent}_i)^j[p]) > f((\mathsf{parent}_i)^{j-1}[p])$.*

Proof. Suppose $(\mathsf{parent}_i)^j[p] \neq (\mathsf{parent}_i)^{j-1}[p]$. Then, the change of parenthood relation can only happen at Line 10 and, at that point, we have that $(\mathsf{parent}_i)^{j-1}[p] = par$ and was modified to $(\mathsf{parent}_i)^j[p] = r_{other}$. By construction, Line 10 can only be executed when $f(par) > f(r_{other})$ and, therefore, $f((\mathsf{parent}_i)^{j-1}[p]) > f((\mathsf{parent}_i)^j[p])$. □

Now, we can prove that any canonical element of $(\mathsf{parent}_i)^j$ is also canonical in $(\mathsf{parent}_i)^{j-1}$:

Proposition 4.28. *Let $(\mathsf{parent}_i)^{j-1}$ and $(\mathsf{parent}_i)^j$ be two arrays where $(\mathsf{parent}_i)^j$ is obtained by updating $(\mathsf{parent}_i)^{j-1}$ calling* CONNECT *for the pair $\{p_j, q_j\}$. Then, any canonical element r of $(\mathsf{parent}_i)^j$ is also a canonical element in $(\mathsf{parent}_i)^{j-1}$.*

Proof. Let r be a canonical element of $(\mathsf{parent}_i)^j$, namely, $f((\mathsf{parent}_i)^j[r]) < f(r)$ and suppose by contradiction that r is not canonical in $(\mathsf{parent}_i)^{j-1}$, that is, $f((\mathsf{parent}_i)^{j-1}[r]) = f(r)$. Consequently, we have that $f((\mathsf{parent}_i)^{j-1}[r]) > f((\mathsf{parent}_i)^j[r])$, implying that $(\mathsf{parent}_i)^{j-1}[r] \neq (\mathsf{parent}_i)^j[r]$ and a change of parenthood relation happened.

However, if a change of parenthood occurred, then according to Prop. 4.27, we have $f((\mathsf{parent}_i)^j[r]) > f((\mathsf{parent}_i)^{j-1}[r])$, which is a contradiction. Thus, r is a canonical element in $(\mathsf{parent}_i)^{j-1}$ and the proposition is proved. □

Using Prop. 4.28, a possible way of finding all canonical elements that represent new nodes is to find all canonical elements r of $(\mathsf{parent}_i)^{j-1}$ that represented nodes containing either p_j or q_j that are still canonical in $(\mathsf{parent}_i)^j$ and satisfy $f(c) < f(r) \leq f(q_j)$.

Hence, the proposed way of allocating compact arcs is to mark all canonical elements visited during each call of the CONNECT procedure that represent new CCs. If the marked element is still a representative in parent_i, then it represents a new CC and is allocated by our proposed algorithm.

The modified version of CONNECT that marks new nodes is presented in Alg. 17.

Algorithm 17 Updated CONNECT that marks new nodes in *newNodesList*.

```
procedure CONNECT($f$, parent, $cur$, $other$, $changed$, $newNodesList$)
    $r_{cur} \leftarrow$ FINDREP($f, cur, f(cur)$, parent);
    $r_{other} \leftarrow$ FINDREP($f, other, f(other)$, parent);
    if $r_{cur} \neq r_{other}$ then
        $par \leftarrow$ FINDREP($f$, parent[$r_{cur}$], $f$(parent[$r_{cur}$]), parent);
        if $changed$ then
            $newNodesList = newNodesList \cup \{r_{cur}\}$;                ▷ Set of new nodes
        if $par \neq r_{other}$ then
            if $f(par) \geq f(r_{other})$ then
                CONNECT($f$, parent, $par$, $r_{other}$, $changed$, $newNodesList$);
            else
                parent[$r_{cur}$] $\leftarrow r_{other}$;
                CONNECT($f$, parent, $r_{other}$, $par$, $True$, $newNodesList$);
```

Allocation of new nodes based on Alg. 17 is given in Alg. 18. An example is provided in Fig. 4.14.

Algorithm 18 Allocation of new nodes using *newNodesList*.

```
procedure ALLOCATENODES($f$, parent, $newNodesList$, $i$)
    for $p \in newNodesList$ do
        if $p =$FINDREP($f$, parent, $p, f(p)$) then
            Allocate node $N = (p, f(p), i)$;
```

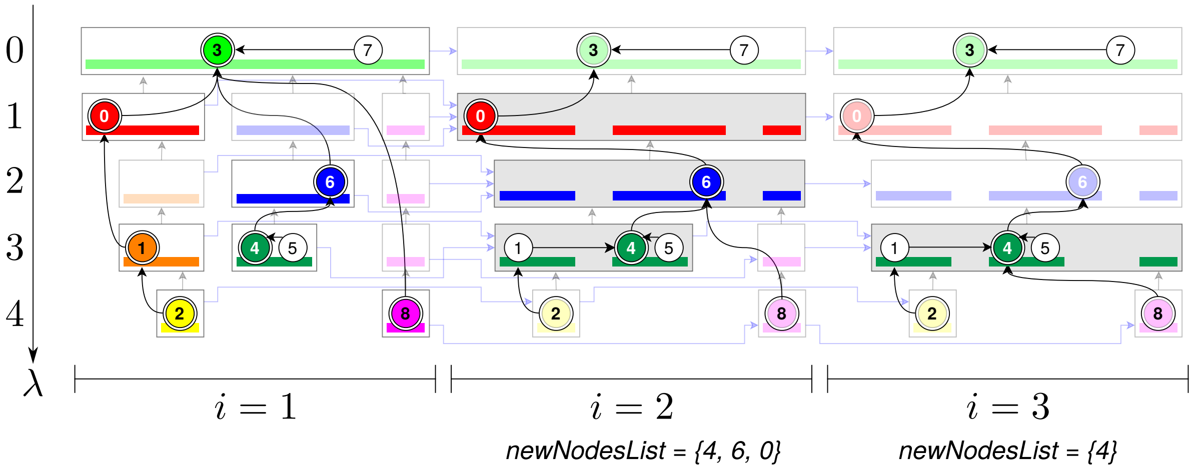


Figure 4.14: *Marking new nodes using the updated* CONNECT *procedure from Alg. 17. In this figure, the complete component-hypertree is shown overlayed with the arrays* parent$_i$*, to highlight that the elements in* $newNodesList$ *represent* $\mathcal{A}_i$*-CCs (marked in gray) that are not* $\mathcal{A}_{i-1}$*-CCs. These marked elements are then used to allocate new nodes using Alg. 18.*

4.5.2 Algorithm for Minimal Arc Allocation

Now that allocation of nodes has been explained, we focus on allocation of minimal arcs. In this sense, let $\mathcal{G}^i_{alg} = (V^i_{alg}, \mathcal{E}^i_{alg})$ be the graph built by the algorithm up to step i, namely, V^i_{alg} is the set allocated nodes and $\mathcal{E}^i_{alg}$ is the set allocated arcs by the algorithm from steps 1 to i. Our goal is to build $\mathcal{G}^i_{alg}$ in a way that it matches $\mathcal{G}^{([0 \to K-1],[1 \to i])}_{MH}$, for any $1 \leq i \leq n$.

From Sec. 4.4, we concluded that the following arcs of compact-hypertrees are minimal arcs:

- Vertical arcs;

- Backward arcs;
- Horizontal arcs;
- Double arcs.

On the other hand, the following arcs were redundant arcs:

- Diagonal parent arcs that are not composite arcs;
- Diagonal composite arcs that are not parent arcs.

The ideas behind the algorithm for minimal arc allocation are explained below. In the following, any compact node N satisfying $adj(N) = i$ is called a new node, while nodes with $adj(N) < i$ are called old nodes.

Regarding vertical and backward arcs, we have the following properties:

Proposition 4.29. *Let $1 < i \leq n$ and suppose that $\mathcal{G}_{MH}^{([0\to K-1],[1\to i])}$ is given. Then, the new vertical arcs and backward arcs that did not exist in $\mathcal{G}_{MH}^{([0\to K-1],[1\to i-1])}$ are all parent arcs of new nodes.*

Proposition 4.29 should be easy to prove, since if there existed any vertical or backward arc that was not a parent arc of a new node, then that arc would link two old nodes, implying that an inclusion relation was missing in $\mathcal{G}_{MH}^{([0\to K-1],[1\to i-1])}$.

As a consequence of Prop. 4.29, a way of making sure that $\mathcal{G}_{alg}^{i}$ has all the new vertical and backward arcs consists of allocating, for each new node N, the arc $(N, par_i(N))$. Following this approach, it is still not clear if we are allocating more arcs than needed, but we can be sure that all new vertical and backwards are allocated.

The other types of minimal arcs (horizontal and double arcs) are composite arcs, that is, they link old nodes to new nodes. In particular, all minimal arcs existing in $\mathcal{G}_{MH}^{[0\to K-1],[1\to i]}$ are mapped from arcs of $\mathcal{G}_{CCH}^{[0\to K-1],[1\to i]}$, in other words, any new minimal arc $e' = (N, N')$ linking and old node to a new node necessarily comes from an arc $e = (M, M')$ of the complete hypertree satisfying $e' = ce(e)$ and, since N' is a new compact node, we have that $adj(N') = i$, implying that $adj(M') = i$ and, by consequence, $adj(M') = i - 1$.

Hence, to allocate any of these minimal arcs, we need an efficient way of finding these old nodes that store $\mathcal{A}_{i-1}$-CCs. For that, we make use of an additional structure $\mathsf{lastNode}_i : D_f \to V_{alg}^{i}$, that maps each pixel $p \in D_f$ to the last allocated node (in other words, the node N satisfying $adj(N) \leq i$ with highest neighborhood index) that had p as its representative. In this way, at the end of step i, all nodes storing $\mathcal{A}_i$-CCs can be obtained by accessing $\mathsf{lastNode}_i[r]$, where r is any canonical element of parent_i.

As a reminder, in the parent array, composite arcs can be found using the $\textsc{findRep}$ function. In the complete hypertree, a node $N = (\mathcal{C}, \lambda, i-1)$ (where $\mathcal{C} = rec(\mathsf{parent}_{i-1}, r)$ and r is the representative of N) has a composite arc to the node $N' = (\mathcal{C}', \lambda, i)$ (where $\mathcal{C}' = rec(\mathsf{parent}_i, r')$ and $r' = \textsc{findRep}(f, \mathsf{parent}_i, r, \lambda)$). From now on, given a node N, its representative is denoted by $rep(N)$.

In this context, let (N, N') be a new horizontal arc with $rep(N) = r$ and $rep(N') = r'$. Then, the following properties need to be satisfied: $adj(N) < i$, $adj(N') = i$, $f(N) = f(N')$, r is a canonical element of parent_{i-1} and r' is a canonical element of parent_i such that $r' = \textsc{findRep}(f, \mathsf{parent}_i, r, f(N))$.

There are two possibilities here: either $r = r'$ or $r \neq r'$. If $r = r'$, then this arc does not exist in any parent array, since there are no arcs (r, r) in the arrays. This means that these horizontal arcs need to be allocated manually. So, in order to guarantee that all of these arcs are added to $\mathcal{E}_{alg}^{i}$, for all new nodes M', we find the respective node $M = \mathsf{lastNode}[rep(N)]$ and allocate all arcs (M, M') manually. Because of Prop 4.28, that states that any canonical element of parent_i is also a canonical element of parent_{i-1}, we can be assured that all of these nodes M' actually exist.

If $r \neq r'$, then this implies that $f(\mathsf{parent}_{i-1}[r]) < f(r)$ but $f(\mathsf{parent}_i[r]) = f(r)$, which means that r is not a canonical element of parent_i. This implies that the parent of r changed at step i

or, in other words, $\mathsf{parent}_{i-1}[r] \neq \mathsf{parent}_i[r]$. In this case, to ensure that all of these horizontal arcs are added, we can add to $\mathcal{E}^i_{alg}$ all arcs (M, M') where $rep(M) = r$ and the parent of r changed in parent_i. Again, we may be adding more arcs than needed, but we can be sure that we included all horizontal arcs.

Finally, let us analyze the case of double arcs. Let $e = (N, N')$ be a new double arc with $r = rep(N)$ and $r' = rep(N')$. Since $f(N) > f(N')$, then $r \neq r'$.

Now, we remind that e is both a parent and a composite diagonal arc. For a parent arc to be a diagonal arc, it must link an old node to a new node and, for a composite arc to be a diagonal arc, we need to have $f(\mathsf{parent}_{i-1})[r] < f(\mathsf{parent}_i r)$ or, in other words, the parent of r changed at step i. Since we already included all arcs pointing from old nodes to new nodes that had a change in parenthood relation, then all double arcs are included in $\mathcal{E}^i_{alg}$.

4.5.3 Algorithm for Arc Allocation

From the previous section, we showed that all minimal arcs in $\mathcal{E}^i_{alg} \setminus \mathcal{E}^{i-1}_{alg}$ are included in the following set of arcs:

1. Parent arcs of new nodes;
2. Composite arcs (N, N') from old to new nodes where $rep(N) = rep(N')$;
3. Parent arcs (N, N') where $rep(N) = r$ and $\mathsf{parent}_{i-1}[r] \neq \mathsf{parent}_i[r]$.

Now, we analyze how we can check those conditions using the data structures and algorithms that were already presented. For that, we will modify the CONNECT procedure to include a second set *parentUpdateList* that marks nodes that need to have their parent updated and, later, we will modify the node allocation step to include the composite arcs that can not be obtained from the parent array.

These ideas are detailed below:

1. Parent arcs of new nodes: since these nodes are represented by canonical elements in the set *newNodesList* of the CONNECT procedure, marking these nodes for parent update is easy, we just add the same elements to the set *parentUpdateList*.
2. Case 2 above: for this case, for every node N' allocated at step i with $r = rep(N)$, we find the last allocated node N that had $adj(N') < i$ and $r = rep(N')$ using $\mathsf{lastNode}_i[r]$ and manually allocate the arc (N, N').
3. Case 3 above: a change in parenthood relation can only happen in the CONNECT procedure. In particular, it happens at Line 12 in Alg. 16, so a way of keeping track of nodes that had their parent changed is to add to the set *parentUpdateList* the element r_{cur} every time that line is called in the CONNECT procedure.

A way of efficiently implementing these ideas is to allocate the new nodes, adding the composite arcs with same representatives as stated in case (2) above. At this point, $\mathsf{lastNode}_i$ can be updated, since all new nodes are already allocated, and we can use it to map the representatives to their respective newly allocated nodes.

Then, all pixels in the set *parentUpdateList* can now be used to allocate the new remaining arcs from cases (1) and (3) above. In theory, for all $r \in parentUpdateList$ from case (1), the arcs $(\mathsf{lastNode}_i[r], \mathsf{lastNode}_i[r'])$ can be allocated as parent arcs, where r' is the representative of parent_i, or more specifically, $r' = \text{FINDREP}(f, \mathsf{parent}_i, \mathsf{parent}_i[r], f(\mathsf{parent}_i[r]))$. For for all pixels $r \in parentUpdateList$ satisfying case (3) above, the arcs $(\mathsf{lastNode}_{i-1}[r], \mathsf{lastNode}_i[r'])$ can be allocated as double arcs, where r' is the representative of $\mathsf{parent}_i[r]$.

However, we do not need to differentiate, in the set *parentUpdateList*, which pixels come from which case. It turns out that, for all pixels $r \in parentUpdateList$ from case (3) above, $\mathsf{lastNode}_{i-1}[r] =$

$\mathsf{lastNode}[r]$, and there is no need to differentiate $\mathsf{lastNode}_i$ and $\mathsf{lastNode}_{i-1}$. In both cases, the arc that needs to be allocated is the arc $(N, N') = (\mathsf{lastNode}_i[r], \mathsf{lastNode}_i[r'])$. To differentiate parent arcs from double arcs, it suffices to check if $adj(N) = i$ (vertical or backward arc) or $adj(N) < i$ (double arc). Additionally, because there is no need to differentiate $\mathsf{lastNode}_i$ from $\mathsf{lastNode}_{i-1}$, a single mapping $\mathsf{lastNode}$, representing $\mathsf{lastNode}_i$ at step i, can be used to save memory.

Using these ideas, the updated algorithms are shown below as Algs. 19, 20 and 21.

Algorithm 19 Updated CONNECT that marks new nodes in *newNodesList* and new arcs in *parentUpdateList*. To simplify the algorithm, it assumes that these two lists are global variables.

```
procedure CONNECT($f$, parent, $cur$, $other$, $changed$)
    $r_{cur} \leftarrow$ FINDREP($f, cur, f(cur)$, parent);
    $r_{other} \leftarrow$ FINDREP($f, other, f(other)$, parent);
    if $r_{cur} \neq r_{other}$ then
        $par \leftarrow$ FINDREP($f$, parent$[r_{cur}]$, $f($parent$[r_{cur}])$, parent);
        if $changed$ then
            $newNodesList = newNodesList \cup \{r_{cur}\}$                ▷ Global variable
            $parentUpdateList = parentUpdateList \cup \{r_{cur}\}$        ▷ Global variable
        if $par \neq r_{other}$ then
            if $f(par) \geq f(r_{other})$ then
                CONNECT($f$, parent, $par$, $r_{other}$, $changed$);
            else
                parent$[r_{cur}] \leftarrow r_{other}$;
                $parentUpdateList = parentUpdateList \cup \{r_{cur}\}$    ▷ Global variable
                CONNECT($f$, parent, $r_{other}$, $par$, $True$);
```

Algorithm 20 Updated ALLOCATENODES. First, it allocates new nodes based on the elements of *newNodesList*. Then, for each new node N, it allocates horizontal arcs (N', N), where $rep(N') = rep(N)$. Finally, $\mathsf{lastNode}$ is updated.

```
procedure ALLOCATENODES($f$, parent, $newNodesList$, $i$)
    for $p \in newNodesList$ do
        if $p =$FINDREP($f$, parent, $p, f(p)$) then
            Allocate node $N = (p, f(p), i)$;
            if lastNode$[p] \neq \emptyset$ then                      ▷ equivalent to $i > 1$
                Add composite arc (lastNode$[p], N$);
            lastNode$[p] \leftarrow N$;
```

Algorithm 21 Allocation of minimal arcs based on *parentUpdateList*.

```
procedure UPDATENEWARCS($f$, parent, lastNode, $parentUpdateList$);
    for $p \in updateParentList$ do
        $N \leftarrow$ lastNode$[p]$, $N' \leftarrow$ lastNode[parent$[p]$];
        if $f(N) < f(N')$ then
            Add parent arc $(N, N')$;
        if $adjIndex(N) < adjIndex(N')$ then
            Add composite arc $(N, N')$;
```

Finally, using these algorithms, we can build the minimal-hypertree using Alg 22.

Algorithm 22 Component-hypertree construction template.

```
procedure BUILDHYPERTREE($f$, $\mathbb{A} = (\mathcal{A}_1, \dots, \mathcal{A}_n)$)
    parent ← MAKESET($D_f$);
    for $p \in \mathcal{D}_f$ do
        lastNode[$p$] ← $\emptyset$;
    for $1 \leq i \leq n$ do
        $newNodesList = \emptyset$;                ▷ Assumes $newNodesList$ is a global variable
        $parentUpdateList = \emptyset$;            ▷ Assumes $parentUpdateList$ is a global variable
        for $\{p, q\} \in \mathcal{A}_i$ do
            if $f(p) \geq f(q)$ then
                CONNECT($f$, parent, $p$, $p'$, $False$);
            else
                CONNECT($f$, parent, $p'$, $p$, $False$);
        ALLOCATENODES($f$, parent, $newNodesList$, $i$);
        UPDATENEWARCS($f$, parent, lastNode, $parentUpdateList$);
```

Examples running these algorithms are presented in Figs. 4.15, 4.16 and 4.17.

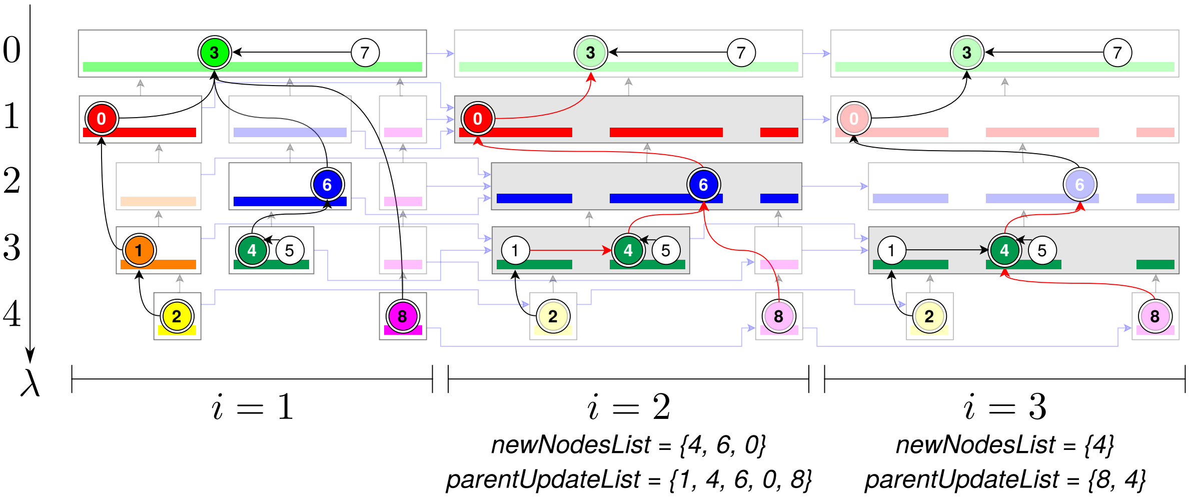
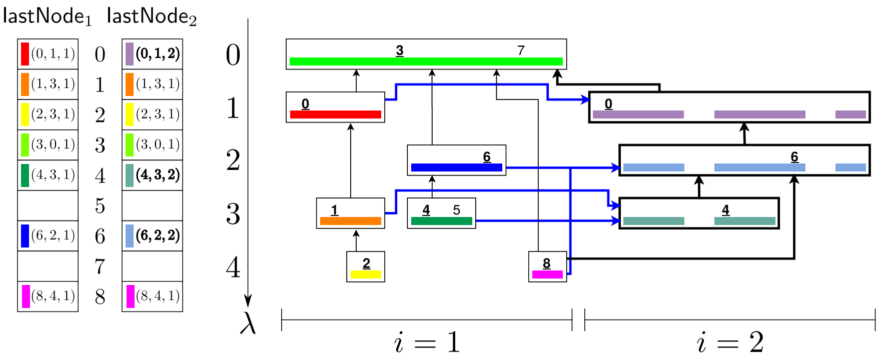


Figure 4.15: *Marking nodes and arcs using the* CONNECT *procedure from Alg. 19. In this figure, the complete component-hypertree is shown overlayed with the arrays* parent$_i$*, highlighting the new nodes (in gray) and the parent arcs (in red) that need to be allocated in the minimal-hypertree. New nodes are represented by canonical elements in newNodesList and red arcs are arcs* $(r, \text{parent}[r])$*, where* $r \in parentUpdateList$*.*



lastNode$_1$		lastNode$_2$
🟥 $(0,1,1)$	0	**$(0,1,2)$**
$(1,3,1)$	1	$(1,3,1)$
$(2,3,1)$	2	$(2,3,1)$
$(3,0,1)$	3	$(3,0,1)$
$(4,3,1)$	4	**$(4,3,2)$**
	5	
🟦 $(6,2,1)$	6	**$(6,2,2)$**
	7	
$(8,4,1)$	8	$(8,4,1)$

Figure 4.16: *Allocation of new nodes and arcs for the step* $i = 2$ *running Alg. 22.*

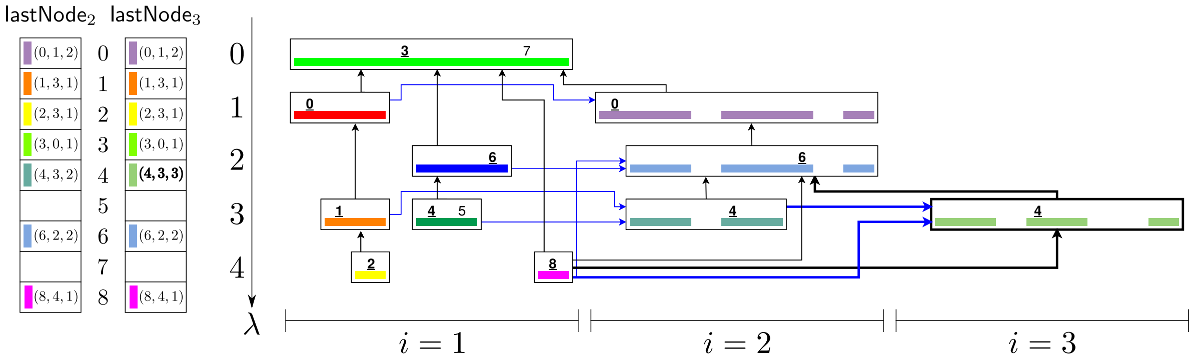


Figure 4.17: *Allocation of new nodes and arcs for the step $i = 3$ running Alg. 22.*

It is worth noting that, in Figs. 4.16 and 4.17, the nodes N satisfying $adj(N) = 1$ store non-canonical pixels as well. This is not done explicitly in the algorithms, but is needed for reconstruction and attribute computation. To do so, we add all pixels $p \in D_f$ to the node $N_p = \mathsf{lastNode}_1[r]$, where r is the representative of p in parent_1, that is, $r = \textsc{findRep}(f, \mathsf{parent}_1, p, f(p))$. For nodes N with $adj(N) > 1$, this is not necessary, because all of these nodes are merges of nodes from $\mathcal{A}_1$, and pixels of their CCs can be recovered using the inclusion relations of the component-hypertree.

4.5.4 Arcs Allocated by the Algorithm are Minimal Arcs

In Sec. 4.4, we showed some properties that minimal arcs satisfy, and allocated in our algorithm all arcs that satisfied said properties. In other words, we can guarantee that every minimal arc is allocated by the algorithm. Now, in this section, we prove that all allocated arcs are actually minimal arcs, or more formally, that $\mathcal{E}^i_{alg} \subseteq \mathcal{E}^{([0 \to K-1],[1 \to i])}_{MH}$.

As a reminder, the array parent_i represents the component-tree of an image f using the neighborhood $\mathcal{A}_i$. This means that, for every arc $(r, \mathsf{parent}_i[r])$ where r is a canonical element (and not the root), there is an arc (N, N') in the component-tree $CT(f, \mathcal{A}_i)$ where $CC(N) = rec(\mathsf{parent}_i, r)$ and $CC(N') = rec(\mathsf{parent}_i, \mathsf{parent}_i[r])$. Since all arcs of each individual component-tree are in the compact-hypertree, then for all arcs in parent_i, there is a corresponding parent arc in $\mathcal{G}^{([0 \to K-1],[1 \to i])}_{CH}$. Hence, we can be sure that any parent arc allocated by the algorithm is a parent arc of the compact-hypertree of f and $\mathbb{A}$.

Additionally, most arcs are allocated using the *parentUpdateList* set of the CONNECT procedure. Using Prop. 4.28, it is easy to see that:

Proposition 4.30. *Suppose Alg. 22 is running at step i and let $parentUpdateList_i$ denote parentUpdateList before the call of* UPDATENEWARCS *at step i. If $r \in parentUpdateList_i$, then r is a canonical element of parent_{i-1}.*

Now, we need to prove that the parent arcs allocated using *parentUpdateList* are, indeed, minimal arcs.

For that, suppose Alg. 22 is running at step i. There are two cases that allocate parent arcs in the algorithm. In the first case, any new node N satisfying $adj(N) = i$ has a parent arc allocated, where the parent node N' of N satisfies either $adj(N') = i$ or $adj(N') < i$. In other words, this parent arc is either a vertical arc or a backward arc and, therefore, the arc (N, N') allocated by the algorithm is a minimal arc.

The second case consists of allocation of parent arcs of nodes that had their parent changed. In this case, they consist of arcs $(r, \mathsf{parent}_i[r])$ where r represents an old compact node N and $r' = \textsc{findRep}(f, \mathsf{parent}_i, \mathsf{parent}_i[r], f(\mathsf{parent}_i[r]))$ represents a compact node N'.

From Prop. 4.27, we know that a change in parent implies that $f(\mathsf{parent}_{i-1}[r]) < f(\mathsf{parent}_i[r])$. This property has two implications: first, N' represents a new node, since a change in parent implies that the new parent represents a new CC. Second, $\textsc{findRep}(f, \mathsf{parent}_{i-1}, r, f(\mathsf{parent}_i[r])) = \textsc{findRep}(f, \mathsf{parent}_{i-1}, r, f(r))$, that is to say that, in the complete-hypertree, the nodes $N =$

$(\mathcal{C}, f(r), i-1)$ and $M = (\mathcal{C}', f(\mathsf{parent}_i[r]), i-1)$ with $r \in \mathcal{C}$ and $r \in \mathcal{C}'$ satisfy $rep(N) = rep(M)$ and $\mathcal{C} = \mathcal{C}'$. Hence, this node M is a partial node of $N' = (rec(\mathsf{parent}_i, \mathsf{parent}_i[r]), f(\mathsf{parent}_i[r]), i)$ in the complete hypertree, implying that the arc $(cn(M), cn(N')) = (N, N')$ is a composite arc of the compact-hypertree. However, the existence of the arc $(r, \mathsf{parent}_i[r])$ in parent_i also implies that $(cn(N), cn(N')) = (N, N')$ is also a parent arc. Hence, (N, N') is a double arc and, by consequence, a minimal arc.

For composite arcs, they can be either allocated manually at the node allocation step or added later if they are in the set *parentUpdateList* of the CONNECT procedure. In the first case, for any newly allocated node N' with $rep(N') = r$, we find the last allocated node N with $rep(N) = r$ in $\mathsf{lastNode}[r]$ which, at that moment, still stores $\mathsf{lastNode}_{i-1}[r]$. In $\mathsf{lastNode}_{i-1}$, for any canonical element r' of parent_{i-1}, $\mathsf{lastNode}_{i-1}[r']$ stores a node representing a $\mathcal{A}_{i-1}$-CC. Hence, N is a compact node satisfying $f(N) = f(r)$ and $CC(N)$ is a $\mathcal{A}_{i-1}$-CC, or to put it another way, there exists a node $M = (\mathcal{C}, f(r), i-1)$ in the complete hypertree satisfying $\mathcal{C} = CC(N)$. Meanwhile, N' is a new compact node satisfying $f(N') = f(r)$ and $adj(N) = i$. Since they both include r and N' is a compact node, then $CC(N) = CC(M) \subset CC(N')$, implying that (M, N') is a composite arc of the complete hypertree. Hence, $(cn(M), cn(N')) = (N, N')$ is a composite arc of the compact-hypertree satisfying $f(r) = f(N) = f(N')$, that is, it is a horizontal arc and, by consequence, a minimal arc.

In the second case, for composite arcs that are parent arcs (in other words, they are double arcs), this case was already covered before. The remaining case is when the arc $(r, \mathsf{parent}_i[r])$ is only mapped to a composite of the compact-hypertree. This can only happen if r had its parent changed but is not a canonical element of parent_i.

Using Prop. 4.30, we know that r was a canonical element of parent_{i-1}. In this case, the arc $(r, \mathsf{parent}_i[r])$ is mapped into the arc $(\mathsf{lastNode}[r], \mathsf{lastNode}[r'])$, where r' is the representative of $\mathsf{parent}_i[r]$ in parent_i. Again, r' represents a new node because a change in parenthood relation happened.

Since r is not canonical in parent_i, $N = \mathsf{lastNode}_i[r] = \mathsf{lastNode}_{i-1}[r]$, while $N' = \mathsf{lastNode}_i[r']$ represents a new node. Since $N = \mathsf{lastNode}_{i-1}[r]$, then $CC(N)$ is a $\mathcal{A}_{i-1}$-CC of $X_{f(r)}(f)$. This means that, in the complete-hypertree, there exists a node $M = (CC(N), f(r), i-1)$.

On the other hand, $CC(N')$ is a $\mathcal{A}_i$-CC of $X_{f(r')}(f)$, and $N' = (CC(N'), f(r'), i)$ is a node of the complete-hypertree. In particular, since r is not canonical in parent_i, then $f(r) = f(\mathsf{parent}_i[r]) = f(r')$.

In other words, both $M = (CC(N), f(r), i-1)$ and $N' = (CC(N'), f(r), i)$ are nodes of the complete-hypertree. Since $r \in CC(N)$ and $r \in CC(N')$, then any pixel $\mathcal{A}_{i-1}$-connected to r is also $\mathcal{A}_i$-connected to r, implying that $CC(N) \subset CC(N')$. Hence, (M, N') is a composite arc of the complete-hypertree and the arc $(cn(M), cn(N') = (N, N')$ is a compact arc with $f(N) = f(N')$ and, finally, we can conclude that (N, N') is a horizontal arc. By consequence, it is a minimal arc.

With that, we conclude that all arcs allocated by the algorithm are minimal arcs.

4.6 Parent and Composite Nodes in Minimal-Hypertrees

As explained before, minimal-hypertrees provide an efficient way of representing component-hypertrees by removing repeated CCs and avoiding storing most redundant arcs. However, it is important to observe that minimal-hypertrees, compact-hypertrees and complete-hypertrees are equivalent, in the sense that all information stored in a complete component-hypertree can be inferred from its respective minimal-hypertrees.

Thus, this means that, given a compact node N of a minimal-hypertree, it is possible to infer the location of all non-compact nodes M of the complete-hypertree such that $cn(M) = N$, as well as all parent, composite, children and partial nodes of N.

4.6.1 Parent Nodes in Minimal-Hypertrees

For this section, we first recall that a redundant parent arc $e = (N, N')$ is an arc of a compact-hypertree that is a diagonal parent arc but not a diagonal composite arc. A diagonal parent arc (N, N') is redundant if there exists another parent arc (N, M) where $f(M) = f(N')$ and $adj(M) < adj(N')$. In this case, there is path from N to M and a path from M to N', making the arc (N, N') redundant. An example is shown in Fig 4.18.

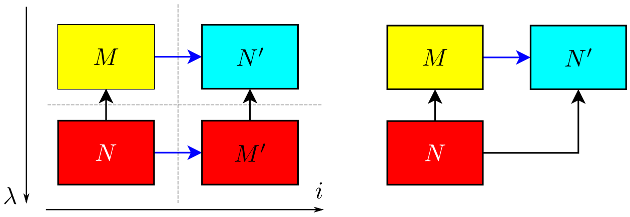


Figure 4.18: *Occurence of diagonal parent arcs. On the left, we show the configuration of a complete-hypertree that generates a diagonal parent arc, where nodes with the same color represent the same CC. The respective compact-hypertree is shown on the right. In the minimal-hypertree, the diagonal parent arc is not allocated.*

Let $par(\mathcal{G}_{MH}, N)$ be the set of all parents of N in the minimal-hypertree $\mathcal{G}_{MH} = (V_{MH}, \mathcal{E}_{MH})$, that is, for any $N_P \in par(\mathcal{G}_{MH}, N)$, $(N, N_P) \in \mathcal{E}_{MH}$ is a parent arc. If we compare $par(\mathcal{G}_{MH}, N)$ to $par(\mathcal{G}_{CH}, N)$, where $\mathcal{G}_{CH} = (V_{CH}, \mathcal{E}_{CH})$ is the corresponding compact-hypertree of $\mathcal{G}_{MH}$, then it is clear that $par(\mathcal{G}_{MH}, N) \subseteq par(\mathcal{G}_{CH}, N)$, where any $N' \in par(\mathcal{G}_{CH}, N) \setminus par(\mathcal{G}_{MH}, N)$ has the property that (N, N') is a redundant parent arc.

Suppose we want to find $par_i(\mathcal{G}_{MH}, N)$, the parent of N at neighborhood index i in the minimal-hypertree $\mathcal{G}_{MH}$. For convenience, suppose also that the elements of $par(\mathcal{G}_{MH}, N)$ are ordered according to their neighborhood indices.

If there exists a node $N_P \in par(\mathcal{G}_{MH}, N)$ where $adj(N_P) = i$, then $N_P = par_i(\mathcal{G}_{MH}, N)$. If this node does not exist, then we find the element $N'_P \in par(\mathcal{G}_{MH}, N)$ with the highest neighborhood index $adj(N'_P) < i$. In the compact-hypertree, that would be the correct parent already. However, since some parent arcs are not allocated in minimal-hypertrees, it is possible that the parent arc that points to the correct node is not allocated.

In that case, suppose that N' is the node we are searching for, in other words, $N' = par_i(\mathcal{G}_{MH}, N)$. If $(N, N') \notin \mathcal{E}_{MH}$, then this arc is redundant, implying that there exists a parent $N'_P \in par(\mathcal{G}_{MH}, N)$ satisfying $CC(N'_P) \subset CC(N')$, $f(N'_P) = f(N')$ and $adj(N'_P) < adj(N')$. Hence, either N' is a composite of N'_P or there is a path from N'_P to N' composed of horizontal arcs, and the correct node N' is the last element in this path that has neighborhood index lower than or equal to i.

Then, supposing $par(\mathcal{G}_{MH}, N) = (N_1, \ldots, N_{|par(\mathcal{G}_{MH},N)|})$ is the set of parent nodes of N in $\mathcal{G}_{MH}$ ordered by neighborhood indices (more formally, $adj(N_\ell) < adj(N_{\ell+1})$ for any $1 \leq \ell < |par(\mathcal{G}_{MH}, N)|$), then Alg. 23 can be used to obtain the parent of N at neighborhood index $1 \leq i \leq n$. An example is shown in Fig 4.19.

Algorithm 23 Algorithm that, given two nodes N, N_C with N_C a composite of N, finds nodes N' such that $e = (N, N')$ is a missing redundant composite arc in a minimal-hypertree.

```
procedure GETPARENT($N$, $i$)
    $N_C \leftarrow comp(N)$ with $f(N) = f(N_C)$;
    $N_P \leftarrow \emptyset$;
    if $i < adj(N_C)$ then                              ▷ in other words, if $par_i(N)$ exists
        $\ell \leftarrow 1$
        $N_P = N_1$;      ▷ Suppose $par(N) = (N_1, \ldots, N_{|par(N)|})$, ordered by neighborhood index
        while $adj(N_{\ell+1}) \leq i$ do
            $\ell = \ell + 1$
            $N_P = N_\ell$
        $N_{PC} \leftarrow comp(N_P)$ with $f(N_{PC}) = f(N_P)$;
        while $N_{PC} \neq \emptyset$ and $adj(N_{PC}) \leq i$ do
            $N_P \leftarrow N_{PC}$
            $N_{PC} \leftarrow comp(N_P)$ with $f(N_{PC}) = f(N_P)$;
    return $N_P$
```

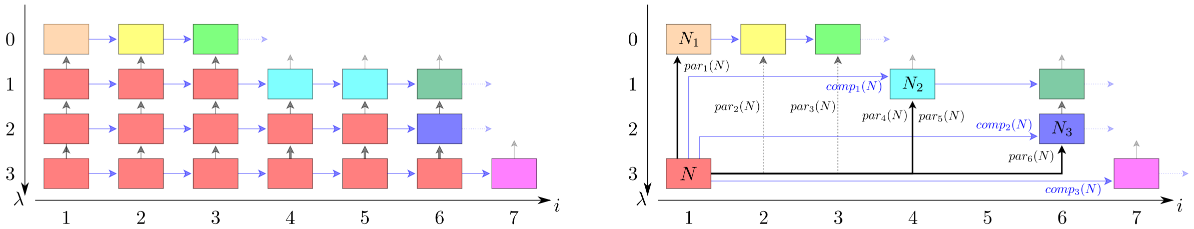


Figure 4.19: *Finding the parent of a node N at neighborhood index i in minimal-hypertrees. Dotted arcs indicate compact redundant arcs that are not allocated in the minimal representation.*

4.6.2 Composite Nodes in Minimal-Hypertrees

For some applications, having all the redundant arcs can be useful. In particular, some attributes that will be presented in Sec 4.8 take advantage of the number of partial nodes of a node and, for that, we need to take redundant arcs into consideration.

For these particular cases where redundant composite arcs are useful, a question that may arise is if it is not better to simply allocate the compact component-hypertree instead of the minimal-hypertree. However, it turns out that designing an efficient algorithm for compact-hypertree construction seems to be harder than designing an algorithm for minimal-hypertree construction. The biggest problem is that redundant composite arcs are not arcs of the parent arrays, and to detect them we need to either further modify the CONNECT procedure to detect occurrences of redundant arcs, which does not seem to be an easy task, or find these composite arcs by checking all possible gray-levels where non-compact nodes were removed to check for potential missing redundant arcs, which is not efficient.

Hence, in these cases, we chose to build the minimal-hypertree instead and, if needed, using a post-processing algorithm to search for redundant arcs. The algorithm that we present is more efficient than checking all gray-levels for each node and has the additional advantage that we do not need to explicitly allocate the nodes: knowing where they are is enough to compute the attributes, so memory can also be saved.

To explain how redundant composite arcs can be detected in minimal-hypertrees, we first recall that redundant composite arcs are diagonal arcs that are composite arcs but not parent arcs. This happens when the configuration shown in Fig. 4.20 situation exists in a complete-hypertree:

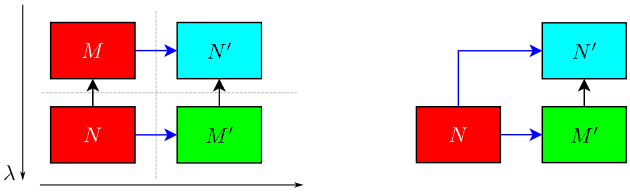


Figure 4.20: *Patterns that indicate the occurrence of diagonal composite arcs in complete-hypertrees and their respective representation in compact-hypertrees.*

For every occurrence of this scenario, there is a redundant composite arc. Figure 4.21 depicts a more general example, showing some cases of this pattern happening in a complete hypertree and how they are mapped into its corresponding minimal-hypertree.

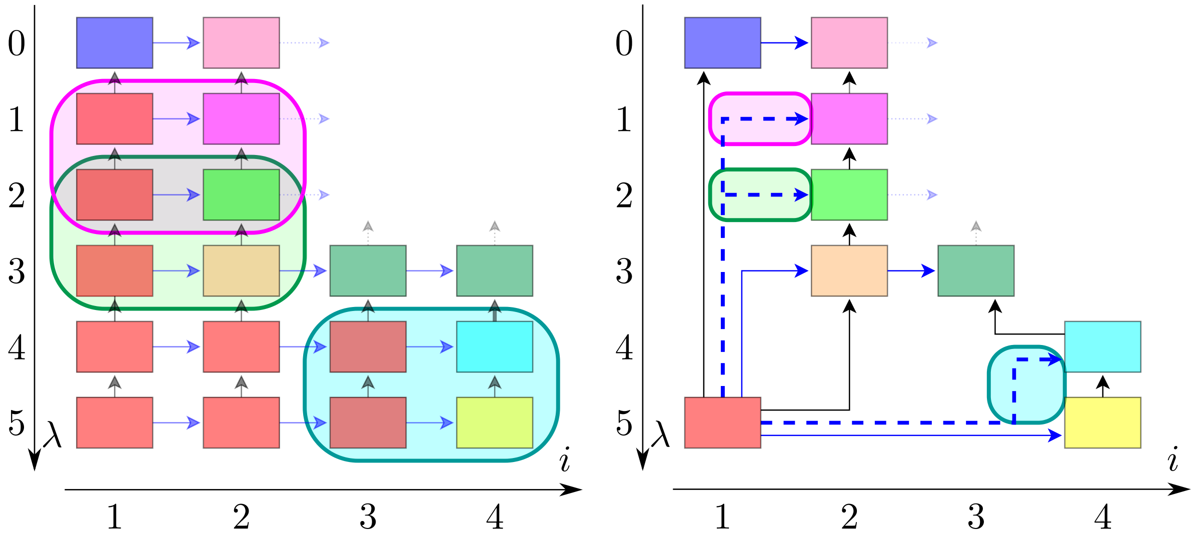


Figure 4.21: *Left: a generic example showing patterns where diagonal composite arcs occur. Right: the respective minimal-hypertree, with the redundant composite arcs represented as dashed arrow.*

According to these observations, to find redundant composite arcs, we need to detect the situation shown in Fig. 4.20 in minimal-hypertrees, that is, we need to find nodes N, M' and N' where M' is a composite node of N' and N' is a parent of M'. Under this circumstance, if M, the original parent of N in the complete hypertree, satisfied $cn(M) = cn(N)$, then we have a redundant composite arc from N to N'.

Naturally, since M is not a compact node, M does not exist in the minimal-hypertree. However, its existence can be inferred from the parent arcs of the minimal-hypertree: from the definition of a compact arc, if a compact parent arc (N_1, N_2) exists, then for all gray-levels between $f(N_1)$ and $f(N_2)$, there was a non-compact node that stored the same CC as N_1 that was removed. This can be observed, for instance, in Fig 4.21, for all the red nodes with $i = 1, 2$ and 3.

Hence, redundant composite arcs can be detected using the following idea: suppose we want to find all missing redundant composite arcs that had a node N as their origin. Then, we find a composite node N_C and, for all ancestors N_C^ℓ of N_C, it is possible that a redundant composite arc from N is missing. To know if these arcs actually exist, we need to search for non-compact nodes that stored $CC(N)$ that were removed, and that can be done by looking at all gray-levels $\lambda \in \mathbb{K}$ such that $f(N_P) < \lambda < f(N)$, where N_P is a parent of N. For all of these gray-levels, there is a missing redundant arc from N to the correspondent ancestor of N_C. An example is shown in Fig 4.22.

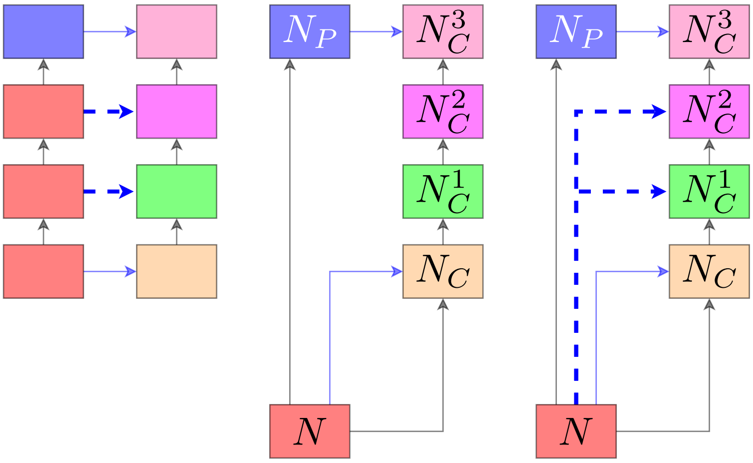


Figure 4.22: *Left: part of a complete-hypertree, extracted from Fig. 4.21, where dashed arcs will become redundant composite arcs. Middle: the respective part in the minimal-hypertree, where the redundant arcs are not allocated. Right: finding missing redundant composite arcs in the minimal-hypertree by looking for ancestors of N_C with gray-level higher than $f(N_P)$.*

In the simplest cases, where N has only one composite and one parent, this idea works. To deal with multiple composite nodes, the idea is simple: we just repeat the process for every pair (N, N_C), where N_C is a composite node of N.

When multiple parents exist, we must choose the proper parent node when choosing N_P and the ancestors of N_C. To do that, we remind that we are always looking at a situation where we are analyzing two consecutive neighborhood indices $adj(N_C)-1$ and $adj(N_C)$ of the complete-hypertree. Hence, in our algorithm, we want N_P to be the parent of N at neighborhood index $adj(N_C) - 1$ and the ancestors of N_C to be obtained from neighborhood index $adj(N_C)$. Using these ideas, the algorithm for obtaining missing redundant composite arcs is presented in Alg. 24.

Algorithm 24 Algorithm that, given two nodes N, N_C with N_C a composite of N, finds nodes N' such that $e = (N, N')$ is a missing redundant composite arc in a minimal-hypertree.

```
procedure GETCOMPOSITES($N$, $N_C$)
    $composites \leftarrow \emptyset$;
    $k \leftarrow N_P.\mathsf{level}$, where $N_P = par_{adj(N_C)-1}(N)$;
    $N' \leftarrow par_{adj(N_C)}(N_C)$;
    while $N'.\mathsf{level} > k$ do
        $composites \leftarrow composites \cup \{N'\}$;
        $N' \leftarrow par_{adj(N_C)}(N')$;
    return $composites$;
```

With Alg. 24, one can recover all missing redundant composite arcs if needed. In particular, this algorithm will be useful later to obtain statistical measures that analyze attributes of merged nodes in component-hypertrees.

4.7 Choice of Neighborhoods

In this chapter, up to this point, we presented an efficient algorithm for minimal-hypertree construction for any choice of increasing symmetric neighborhoods. Now, we change focus and consider the problem of how the choice of neighborhoods used to build $\mathbb{A}$ affects the complexity of the minimal-hypertree construction algorithm.

It can be observed that the complexity of Alg. 22 depends on the number of neighboring pixels. In particular, its complexity is linked to the number of calls of Line 8. This implies that CONNECT is called $\sum_{i=1}^{n} |\mathcal{A}_i|$ times, which may be prohibitive, depending on the choice of the neighborhoods. In other words, Alg. 22 can run faster if the size of the neighborhoods is reduced. Hence, an idea to make the proposed algorithms faster is to process, at every step $1 \leq i \leq n$, only a subset of elements of $\mathcal{A}_i$, if we can guarantee that processing this subset has the same effect of processing all pairs of $\mathcal{A}_i$-neighboring pixels.

For this purpose, given a gray-level image f and an increasing sequence $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, two neighborhoods $\mathcal{A}_i$ and a $\mathcal{A}'_i$ are equivalent if and only if, given an array parent_{i-1} storing $CC(f, \mathcal{A}_{i-1})$, running Alg. 22 at step i with $\mathcal{A}_i$ or $\mathcal{A}'_i$ both produce an array parent_i that stores $CC(f, \mathcal{A}_i)$. Given a sequence $\mathbb{A}' = (\mathcal{A}'_1, \ldots, \mathcal{A}'_n)$, $\mathbb{A}$ and $\mathbb{A}'$ are equivalent iff, for any $1 \leq i \leq n$, $\mathcal{A}_i$ and $\mathcal{A}'_i$ are equivalent, and this implies that running Alg. 22 with $\mathbb{A}'$ and $\mathbb{A}$ produces the same hypertrees. For some well-chosen types of neighborhoods, it is possible to obtain optimized equivalent sequences $\mathbb{A}'$ that significantly reduce the complexity of Alg. 22.

A possible strategy for obtaining those neighborhoods is to build a neighborhood $\mathcal{A}'_i \subset \mathcal{A}_i$ without including arcs $(p, q) \in \mathcal{A}_i$ when p and q are comparable in parent_{i-1} (since they do not change the array when CONNECT is called), or when it is known that there exists another pair $(p', q') \in \mathcal{A}_i$ that will make p and q become comparable as a side effect of calling CONNECT for (p', q').

In particular, a call of the CONNECT procedure has the effect of merging two disjoint paths of the parent array into one. In other words, Prop. 4.31 is valid.

Proposition 4.31. *Let f be a gray-level image and $\mathcal{A}_1$, $\mathcal{A}_2$ be two symmetric neighborhoods such that $\mathcal{A}_2 = \mathcal{A}_1 \cup \{p, q\}$. Suppose that parent_1 stores $CC(f, \mathcal{A}_1)$ and we call* CONNECT$(f, \mathsf{parent}_1, p, q, False)$*, generating an updated array parent_2 storing $CC(f, \mathcal{A}_2)$. Then:*

- *For any pair of pixels (p', q') such that p' and q' are comparable in parent_1, p' and q' are also comparable in parent_2.*

- *For any p' such that $p' = p$ or $p' \in anc(\mathsf{parent}_1, p)$ and any q' such that $q' = q$ or $q' \in anc(\mathsf{parent}_1, q)$, p' and q' are comparable in parent_2.*

More formally, suppose parent_{i-1} is given and there exist pixels p, p', q and $q' \in \mathcal{D}_f$ such that, in parent_{i-1}, the following conditions are valid: (1) p and q are not comparable; (2) $p' \in desc(rep(p))$; and (3) $q' \in desc(rep(q))$ (note that this implies that p' and q' are also not comparable in parent_{i-1}). Additionally, suppose (p, q) and $(p', q') \in \mathcal{A}_i$. Thus, calling CONNECT to (p', q') at step i of Alg. 22 also makes p and q comparable in parent_i. Hence, if CONNECT is called for (p', q') (that is to say, the pair with greater depths in parent_{i-1}, which is also the pair with higher gray-levels in a max-tree), then calling CONNECT for (p, q) is not needed, implying that $\mathcal{A}_i$ and $\mathcal{A}'_i = \mathcal{A}_i \setminus \{(p, q)\}$ are equivalent. Besides, all pairs in $P \times Q \subseteq \mathcal{A}_i$, where P (resp. Q) consists of pixels in the path from p' to the root node (resp. q' to the root) are not needed, with the exception of pair (p', q'). Formally, this idea is presented in Prop. 4.32.

Proposition 4.32. *Suppose that:* (1) *$\mathcal{A}_i$ is a neighborhood;* (2) *parent_{i-1} is given, with $i > 1$;* (3) *$P \times Q \subseteq \mathcal{A}_i$ such that P (resp. Q) is a subset of a path in parent_{i-1} linking a pixel to the root node. If $(h_p, h_q) \in P \times Q$ such that h_p (resp. h_q) is the element of P (resp. Q) with the greatest depth in parent_{i-1}, then $\mathcal{A}_i$ and $\mathcal{A}'_i = (\mathcal{A}_i \setminus (P \times Q)) \cup \{(h_p, h_q)\}$ are equivalent.*

Note that, for any pair of sets (P, Q) satisfying the conditions of Prop. 4.32, $\mathcal{A}'_i$ has up to $|P \times Q| - 1$ fewer elements than $\mathcal{A}_i$. Hence, running Alg. 22 using $\mathcal{A}'_i$ instead of $\mathcal{A}_i$ generates the same parent_i but processing less neighboring pixels.

Although Prop. 4.32 seems to suggest a reduction of $\mathcal{A}_i$ to $\mathcal{A}'_i$, some choices of neighborhoods provide efficient ways of computing $\mathcal{A}'_i$ directly. In the context of Prop. 4.32, for any pair (P, Q) satisfying the specified conditions, only the pair (h_p, h_q) needs to be added to $\mathcal{A}'$. In the following, we present some sequences of neighborhoods that can benefit from this strategy.

4.7.1 Dilation-generated Neighborhoods

In this section, we recall the notions of sequences of dilation-generated neighborhoods [MAH15]. The main idea consists of having multiple increasing neighborhoods, where the next neighborhoods are slightly bigger than the previous, and this growth in size is defined using small SEs.

A dilation-generated neighborhood sequence $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ is built from a sequence of increasing SEs $\mathbb{S} = (\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n)$. We recall that, given a SE $\mathcal{S}$, the neighborhood $\mathcal{A}(\mathcal{S})$ can be defined using Eq. (2.7). Applying Eq. (2.7), we have that:

$$\mathcal{A}(\mathcal{S}_i) = \{(p, q) : p \in \{q\} \oplus \mathcal{S}_i\} \tag{4.5}$$

The sequence $\mathbb{S}$, in turn, is built from another sequence of SEs $\mathbb{B} = (\mathcal{B}_1, \ldots, \mathcal{B}_n)$, as follows:

$$\mathcal{S}_i = \begin{cases} \mathcal{B}_1 \oplus \breve{\mathcal{B}}_1 & i = 1 \\ \mathcal{S}_{i-1} \oplus \mathcal{B}_i \oplus \breve{\mathcal{B}}_i & 2 \leq i \leq n \end{cases} \tag{4.6}$$

From now on, we will refer to the sequences $\mathbb{S}$ and $\mathbb{B}$, as generated sequence and generating sequence, respectively. Examples are provided in Fig. 4.23.

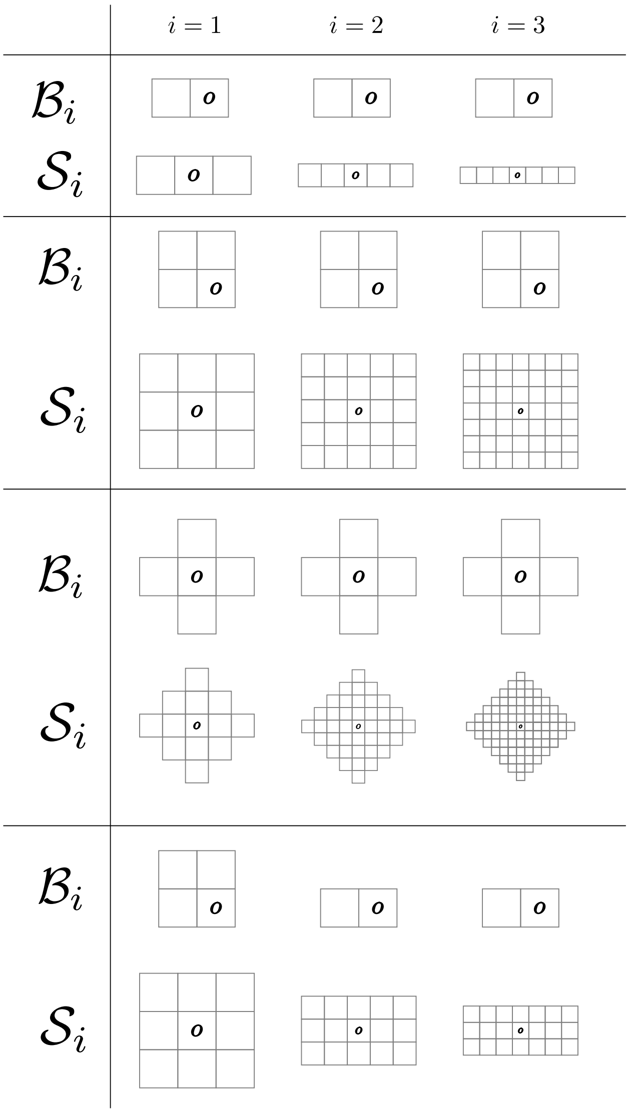


Figure 4.23: *Example of generating sequences and their respective generated sequences.*

Dilation-generated neighborhoods follow some properties that allow the development of efficient algorithms for hypertree construction. For instance:

Proposition 4.33. *For all $1 \leq i \leq n$, $\mathcal{S}_i$ is symmetric.*

To help enunciating other properties, we make use of the following notation:

$$\mathcal{W}_i = \mathcal{B}_1 \oplus \ldots \oplus \mathcal{B}_i \tag{4.7}$$

Using this definition, an alternative way of defining the generating sequence $\mathbb{S}$ consists of the following:

Proposition 4.34. *For all $1 \leq i \leq n$, $\mathcal{S}_i = \mathcal{W}_i \oplus \breve{\mathcal{W}}_i$.*

Additionally, we proved in [MAH15] that Prop. 4.34 implies the following result:

Proposition 4.35. *Let f be a gray-level image, p and $q \in \mathcal{D}_f$ and $\mathbb{S}$ a generated sequence of SEs. Then, for any $1 \leq i \leq n$,*

$$p \in (q \oplus \mathcal{S}_i) \Leftrightarrow ((p \oplus \mathcal{W}_i) \cap (q \oplus \mathcal{W}_i)) \neq \emptyset \tag{4.8}$$

Two examples are provided in Fig. 4.24.

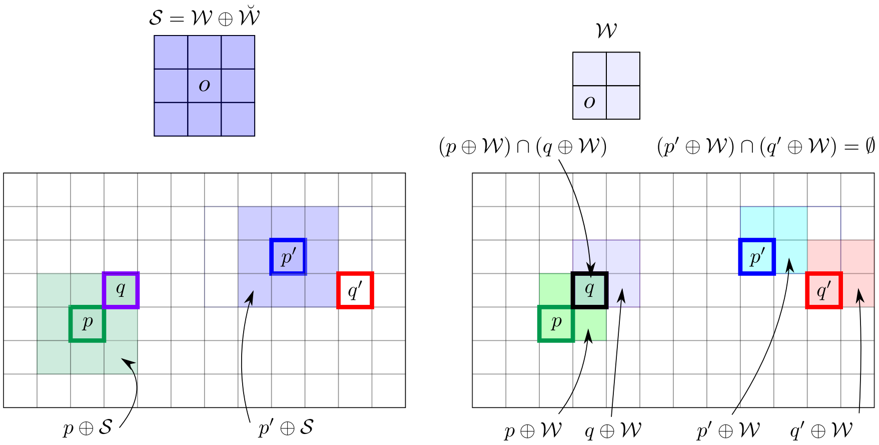


Figure 4.24: *Left: a generic example showing pixels p, q, p', q' satisfying $\{p, q\} \in \mathcal{A}(\mathcal{S})$ but $\{p', q'\} \notin \mathcal{A}(\mathcal{S})$. Right: the same pixels p, q, p', q' and their respective dilations by $\mathcal{W}$. Note that the dilations of p and q by $\mathcal{W}$ intersect, indicating they are $\mathcal{A}(\mathcal{S})$-neighbors, while the dilations of p' and q' do not, indicating they are not $\mathcal{A}(\mathcal{S})$-neighbors.*

It is worth noting that the intersection of two neighboring pixels can happen outside of the domain of the image. An example is shown in Fig. 4.25.

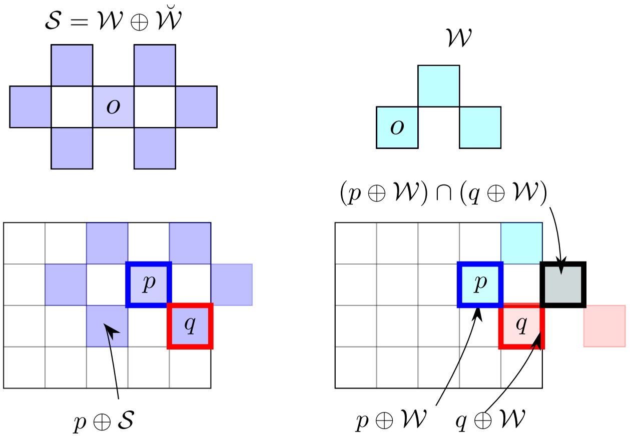

$\mathcal{S} = \mathcal{W} \oplus \check{\mathcal{W}}$
o
$\mathcal{W}$
o
$(p \oplus \mathcal{W}) \cap (q \oplus \mathcal{W})$
p
q
p
q
$p \oplus \mathcal{S}$
$p \oplus \mathcal{W}$
$q \oplus \mathcal{W}$

Figure 4.25: *An example of SE $\mathcal{S}$ where, given two $\mathcal{A}(\mathcal{S})$-neighboring pixels p, q, the intersection $(p \oplus \mathcal{W}) \cap (q \oplus \mathcal{W}) \neq \emptyset$, but happens outside of the domain of the image.*

However, for the neighborhoods presented in Fig. 4.23, it can be proved that there is always at least one pixel inside the domain in the intersections of the dilation of the pixels. If $\mathcal{S}_i$ is generated from a symmetric SE $\mathcal{W}_i$, then it is easy to prove that, given two $\mathcal{A}(\mathcal{S}_i)$-neighboring pixels $p = (x_p, y_p)$ and $q = (x_q, y_q)$, that $(\frac{x_p+x_q}{2}, \frac{y_p+y_q}{2})$ is in $(p \oplus \mathcal{W}_i) \cap (q \oplus \mathcal{W}_i)$. An example is shown in Fig. 4.26.

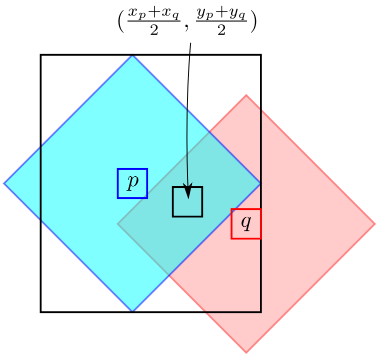


Figure 4.26: *A schematic example showing that, for symmetric SEs $\mathcal{W}$, if the intersection $(p \oplus \mathcal{W}) \oplus (q \oplus \mathcal{S}) \neq \emptyset$, then the middle point of p and q is always in the intersected area.*

For the specific case of a non-symmetric square SE in $2d$, it can also be shown that the intersection happens inside the domain. We use Fig. 4.27 to explain how to prove this fact. Basically, given two $\mathcal{A}(\mathcal{S})$-neighboring pixels $p = (x_p, y_p), q = (x_q, y_q) \in D_f$, there are four possible configurations for the relative position of p and q: they are vertically, horizontally or diagonally disposed (but there are two possible configuration for the diagonal case) and, for any of these cases, it can be proved that there is one point of the domain that is in the intersection $(p \oplus \mathcal{W}) \cap (q \oplus \mathcal{W})$.

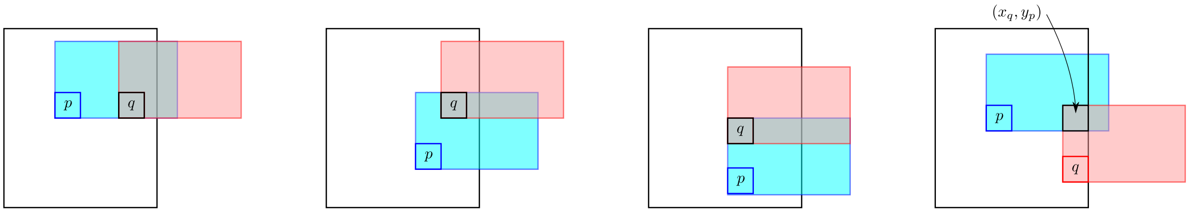


Figure 4.27: *Illustration showing that the intersection $(p \oplus \mathcal{W}) \cap (q \oplus \mathcal{W})$ happens inside the domain for a rectangular non-symmetric SE. For the first three cases, the point q is always in the intersection, while in the last case, the point (x_q, y_p) is in the intersection, and should be in the domain if we consider a rectangular image.*

It is worth noting that the proofs for the symmetric and the rectangular SEs can be easily extended for d-dimensional images, in other words, if an intersection occurs, there is at least one pixel of the domain in the intersected area.

Hence, Prop. 4.35 gives an alternative way of obtaining $\mathcal{A}(\mathcal{S}_i)$-neighbors using intersections of dilated sets. Under the conditions of the proposition, $(p \oplus \mathcal{W}_i) \cap (q \oplus \mathcal{W}_i) \neq \emptyset$ implies that there exists an element $r \in D_f \oplus \mathcal{W}_i$ such that $r \in (p \oplus \mathcal{W}_i) \cap (q \oplus \mathcal{W}_i)$.

Hence, let $L_i : \mathcal{D}_f \oplus \mathcal{W}_i \to \mathcal{P}(\mathcal{D}_f)$ (or $L_i : \mathcal{D}_f \to \mathcal{P}(\mathcal{D}_f)$ if symmetric or squares SEs are being used) be a function that returns, for each pixel p, $L_i(p) = \{q : p \in (q \oplus (\mathcal{W}_i))\}$. Then, all pairs of elements in $L_i(p)$ are $\mathcal{S}_i$-neighbors among themselves.

Because of the way $L_i(p)$ is defined, it is easy to prove that:

Proposition 4.36. *Let $r \in \mathcal{D}_f \oplus \mathcal{W}_i$. Then:*

$$L_i(r) = \bigcup_{x \in \{r\} \oplus \breve{\mathcal{B}}_i} L_{i-1}(x) \tag{4.9}$$

Note that Prop. 4.36 implies that $p, q \in L_i(r)$ iff $\exists p', q' \in \{r\} \oplus \breve{\mathcal{B}}_i$ such that $p \in L_{i-1}(p')$ and $q \in L_{i-1}(q')$.

Using this proposition, the main idea to obtain a neighborhood $\mathbb{A}'$ equivalent to $\mathbb{A} = (\mathcal{A}(\mathcal{S}_1), \ldots, \mathcal{A}(\mathcal{S}_n))$ consists of building mappings, for $2 \leq i \leq n$, $L_{i-1} : \mathcal{D}_f \oplus \mathcal{W}_{i-1} \to \mathcal{P}(\mathcal{D}_f)$ in such a way that, for each $r \in \mathcal{D}_f \oplus \mathcal{W}_{i-1}$, any two distinct pixels in $L_{i-1}(r)$ are comparable in

parent_{i-1} (that is, $L_{i-1}(r)$ is a subset of a path in parent_{i-1}). Supposing that a pair $(p, q) \in \mathcal{A}_i$ with $L_{i-1}(p) \times L_{i-1}(q) \subseteq \mathcal{A}_i$ is given, based on Prop. 4.32, we define:

$$\mathcal{A}'_i = (\mathcal{A}_i \setminus (L_{i-1}(p) \times L_{i-1}(q))) \cup \{h_p^{i-1}, h_q^{i-1}\}, \tag{4.10}$$

where $h_p^{i-1} \in L_{i-1}(p)$ and $h_q^{i-1} \in L_{i-1}(q)$ have the greatest depth in parent_{i-1}. Then $\mathcal{A}_i$ and $\mathcal{A}'_i$ are equivalent.

Note that obtaining these elements h^{i-1} is easy: let $\mathfrak{H}_i : \mathcal{D}_f \to \mathcal{D}_f$ be a function that returns, for each element $p \in \mathcal{D}_f$, the element h_p^i with highest graylevel in $L_i(p)$. Then, $\mathfrak{H}_i$ can be efficiently updated using the following property:

$$\mathfrak{H}_i(p) = \underset{p \in (\{q\} \oplus \mathcal{B}_i)}{\arg\max}\; f(\mathfrak{H}_{i-1}(q)) \tag{4.11}$$

Using this property, one can use Alg. 25 to build the sets of neighboring pixels:

Algorithm 25 Algorithm for building the sets of neighboring pixels using dilation-generated neighborhoods.

```
procedure CREATENEIGHBORS($f$, $\mathbb{B} = (\mathcal{B}_1, \ldots, \mathcal{B}_n)$);
    for $p \in \mathcal{D}_f$ do
        $\mathfrak{H}_0(p) \leftarrow p$;
    $\mathcal{D}_1 \leftarrow \mathcal{D}_f$
    for $1 \leq i \leq n$ do
        $\mathcal{A}_i \leftarrow \emptyset$;
        $\mathfrak{H}_i \leftarrow$ copy of $\mathfrak{H}_{i-1}$;
        if $i < n$ and $\mathcal{B}_{i+1} \subseteq \mathcal{B}_i$ then
            $\mathcal{D}_{i+1} \leftarrow \emptyset$;
        else
            $\mathcal{D}_{i+1} \leftarrow \mathcal{D}_f$;
        for $q \in \mathcal{D}_i$ do
            for $p \in (\{q\} \oplus (\mathcal{B}_i))$ do
                $\mathcal{A}_i \leftarrow \mathcal{A}_i \cup (\mathfrak{H}_{i-1}(q), \mathfrak{H}_i(p))$;
                if $f(\mathfrak{H}_{i-1}(q)) > f(\mathfrak{H}_i(p))$ then
                    $\mathfrak{H}_i(p) \leftarrow \mathfrak{H}_{i-1}(q)$;
                    if $p \notin \mathcal{D}_{i+1}$ then
                        $\mathcal{D}_{i+1} \leftarrow \mathcal{D}_{i+1} \cup \{p\}$;
```

In general, this strategy of building $\mathcal{A}'_i$ and calling CONNECT to its pairs is faster than processing all $\mathcal{A}_i$-neighboring pixels. Furthermore, there are still other optimizations that can be performed.

For instance, in Alg. 25, $\mathcal{D}_i$ is the set of pixels to be used in step i. Using a generic sequence $\mathbb{B} = (\mathcal{B}_1, \ldots, \mathcal{B}_n)$, the simplest way of computing the sets of neighbors is to always choose $\mathcal{D}_i = \mathcal{D}_f$ for any $1 \leq i \leq n$. However, when $\mathcal{B}_{i+1} \subseteq \mathcal{B}_i$ (for $1 \leq i < n$), then $(\{q\} \oplus (\mathcal{B}_{i+1})) \subseteq (\{q\} \oplus (\mathcal{B}_i))$, for any $q \in \mathcal{D}_f$. In particular, at step i, if $\mathfrak{H}_i(q) = \mathfrak{H}_{i-1}(q)$, then q does not need to be added to $\mathcal{D}_{i+1}$ because it can not change any $\mathfrak{H}_{i+1}(p)$ for any $p \in (\{q\} \oplus \mathcal{B}_{i+1})$ at step $i + 1$. Additionally, it can not add any relevant pair of neighbors: for any $p \in (\{q\} \oplus \mathcal{B}_{i+1})$ with $\mathfrak{H}_{i+1}(p) = \mathfrak{H}_i(p)$, the pair $(\mathfrak{H}_i(q), \mathfrak{H}_{i+1}(p))$ was already added in step i as the pair $(\mathfrak{H}_{i-1}(q), \mathfrak{H}_i(p))$. If $\mathfrak{H}_{i+1}(p) \neq \mathfrak{H}_i(p)$, then the new relevant pair of neighbors was already added at the moment $\mathfrak{H}_{i+1}(p)$ changed. This strategy reduces the number of pixels to process, leading to a even more efficient algorithm for hypertree construction based on dilation-generated neighborhoods.

A more detailed complexity analysis is provided in Chapter 5.1. But before that, we analyze other types of neighborhoods that can take advantage of Prop. 4.32.

4.7.2 Neighborhoods Based on Hierarchies of Partitions

In this section, we present the strategy published on [MPAH20] to obtain a sequence of neighboring pixels $\mathbb{A}'$ based on hierarchy of partitions. This strategy can significantly reduce the complexity of the hypertree building algorithm by employing Prop. 4.32.

Let $\mathbb{H} = (\mathcal{H}_1, \ldots, \mathcal{H}_n)$ be a hierarchy of partitions of $\mathcal{D}_f$, that is, each $\mathcal{H}_i$ is a partition of $\mathcal{D}_f$ for $1 \leq i \leq n$ and, for every element R of $\mathcal{H}_i$, there exists an element R' of $\mathcal{H}_{i+1}$ such that $R \subseteq R'$, or to put it another way, the partition $\mathcal{H}_i$ refines the partition $\mathcal{H}_{i+1}$ for $1 \leq i < n$.

For each partition $\mathcal{H}_i$, we consider an undirected graph $\mathcal{G}_i = (\mathcal{H}_i, \mathcal{E}_i)$, where we say $R, R' \in \mathcal{H}_i$ are adjacent if and only if $\{R, R'\} \in \mathcal{E}_i$. Thus, the graph $\mathcal{G}_i = (\mathcal{H}_i, \mathcal{E}_i)$ induces a neighborhood $\mathcal{A}_i$ in such way that, for any two distinct pixels $p, p' \in \mathcal{D}_f$, we have $(p, p') \in \mathcal{A}_i$ iff there exist $R, R' \in \mathcal{H}_i$ such that $p \in R, p' \in R'$ and either $R = R'$ or $(R, R') \in \mathcal{E}_i$.

Depending on the choices of $\mathbb{H}$ and $\mathbb{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_n)$, it is possible to design efficient algorithms for hypertree construction. In particular, let $\mathbb{H}$ and $\mathbb{E}$ be defined in such a way that the following condition holds:

$$\begin{aligned} \forall R \in \mathcal{H}_i, \forall R_1, R_2 \in \mathcal{H}_{i-1} \text{ with } R_1 \neq R_2, R_1 \cup R_2 \subseteq R \\ \Rightarrow (R_1, R_2) \in \mathcal{E}_{i-1} \end{aligned} \tag{4.12}$$

Suppose that $\mathbb{H}$ and $\mathbb{E}$ are given and they satisfy the conditions in Eq. (4.12). Since any region $R \in \mathcal{H}_i$ consists of a set of regions of $\mathcal{H}_{i-1}$, then for any $(p, q) \in \mathcal{A}_i$ with $p, q \in R$, either p and q both belong to the same region of $\mathcal{H}_{i-1}$ or they belong to adjacent regions of $\mathcal{G}_{i-1}$. In both cases, by definition, this implies that p and q are $\mathcal{A}_{i-1}$-neighbors.

Hence, for any pair $(p, q) \in \big(\mathcal{A}_i \setminus \mathcal{A}_{i-1}\big)$, p and q are in adjacent regions of $\mathcal{G}_i$. Thus, there exist two adjacent regions $R_p, R_q \in \mathcal{H}_i$ such that $p \in R_p$, $q \in R_q$. As explained above, any two distinct elements of R_p (resp. R_q) are $\mathcal{A}_{i-1}$-neighbors, which means, at the start of step i in Alg. 22, they are comparable in parent_{i-1}. Then, Prop 4.32 applies and we can define:

$$\mathcal{A}'_i = (\mathcal{A}_i \setminus (R_p \times R_q)) \cup \{h^i_p, h^i_q\}, \tag{4.13}$$

where $h^i_p \in R_p$ and $h^i_q \in R_q$ have the greatest depth in parent_{i-1}. From Prop 4.32, we conclude that $\mathcal{A}_i$ and $\mathcal{A}'_i$ are equivalent.

4.7.3 Pyramidal Hierarchy

One particular strategy that can be used to efficiently build a hierarchy of partitions is to design a pyramidal hierarchy. For that, consider a sequence $(\mathcal{D}_0, \mathcal{D}_1 \ldots, \mathcal{D}_n)$ such that $\mathcal{D}_f = \mathcal{D}_0 \supset \mathcal{D}_1 \supset \ldots, \supset \mathcal{D}_n$ and, for $i = 1, \ldots, n$, let $\rho_i : \mathcal{D}_{i-1} \to \mathcal{D}_i$ be a mapping, called downsampling, that assigns each pixel $p \in \mathcal{D}_{i-1}$ to a pixel $u \in \mathcal{D}_i$. In this way, we define a pyramidal hierarchy as a sequence of downsamplings $(\rho_1, \ldots, \rho_n)$. An interesting property is that the composition of i downsamplings $\theta_i = \rho_i \rho_{i-1} \cdots \rho_1 : \mathcal{D}_f \to \mathcal{D}_i$ induces an equivalence relation on $\mathcal{D}_f$, denoted by $\equiv_i$, as follows:

$$\forall p, q \in \mathcal{D}_f, (p \equiv_i q) \Leftrightarrow \theta_i(p) = \theta_i(q) \tag{4.14}$$

This equivalence relation leads us to a partition $\mathcal{H}_i$ of $\mathcal{D}_f$ in a such way that two distinct pixels $p, q \in \mathcal{D}_f$ are in the same region in $\mathcal{H}_i$ iff $\theta_i(p) = \theta_i(q) = u \in \mathcal{D}_i$. More formally, $\mathcal{H}_i = \{R^u_i : u \in \mathcal{D}_i\}$ where $R^u_i = \{x \in \mathcal{D}_f : u = \theta_i(x)\}$. In this way, a pyramidal hierarchy $(\rho_1, \ldots, \rho_n)$ defines a hierarchy of partitions $\mathbb{H} = (\mathcal{H}_1, \ldots, \mathcal{H}_n)$.

Given these definitions, we now focus on obtaining a specialized and suitable pyramidal hierarchy for efficient component-hypertree construction. One possible choice consists of the following: given $(t_1, \ldots, t_d) \in \mathbb{Z}^d$ with $t_j > 0$, we define the downsampling $\rho_i : \mathcal{D}_{i-1} \to \mathcal{D}_i$ as, for any $x = (x_1, \ldots, x_d) \in \mathcal{D}_{i-1}$,

$$\rho_i(x = (x_1, \ldots, x_d)) = \begin{cases} (x_1, \ldots, x_d), & i = 1; \\ (\lfloor x_1/t_1 \rfloor, \ldots, \lfloor x_d/t_d \rfloor), & i > 1, \end{cases} \tag{4.15}$$

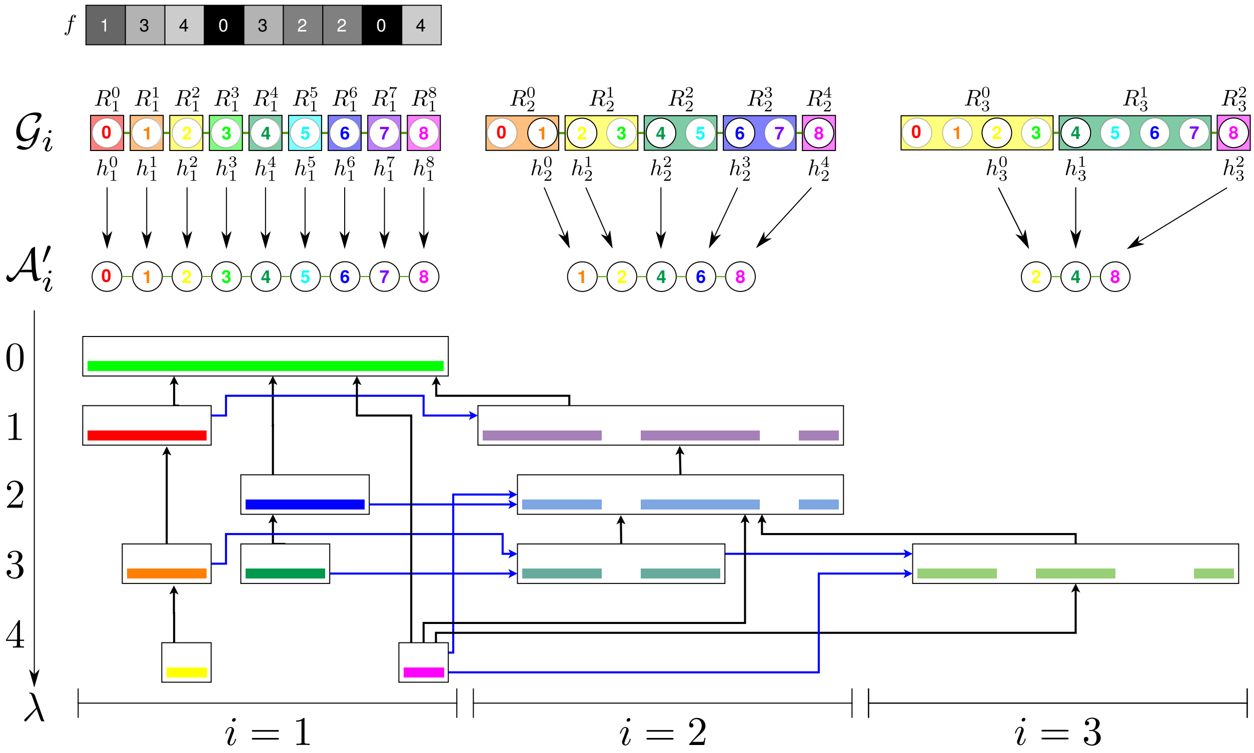
where $\lfloor a \rfloor$ indicates the greatest integer less than or equal to a. In this way, $\mathcal{H}_i$ consists of hyper-rectangles of $\mathcal{D}_f$ of size $((t_1)^{i-1}, (t_2)^{i-1}, \ldots, (t_d)^{i-1})$.

Now, we define the sequence $\mathbb{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_n)$. In order to ensure the validity of condition given in Eq. (4.12), we make use of the following graph $\mathcal{G}_i = (\mathcal{H}_i, \mathcal{E}_i)$, where

$$\mathcal{E}_i = \{(R_i^v, R_i^u) \in \mathcal{H}_i \times \mathcal{H}_i : v, u \in \mathcal{D}_i, |v_j - u_j| < t_j, 1 \leq j \leq d\} \tag{4.16}$$

Finally, we analyze this problem in the context of Alg. 22 and Prop. 4.32. Suppose that Alg. 22 is running at the beginning of step i. Then, since the choice of $\mathbb{E}$ satisfies the conditions given in Eq. (4.12), we can make use of Eq. (4.13) to compute a neighborhood $\mathcal{A}'_i$ equivalent to $\mathcal{A}_i$, using the following strategy: for all $p \in \mathcal{D}_i$, find all $q \in \mathcal{D}_i$ such that $p \neq q$ and $(R_i^p, R_i^q) \in \mathcal{E}_i$; and, then, add the pair (h_p^i, h_q^i) into $\mathcal{A}'_i$ (see Eq. (4.13)). An example is provided in Fig. 4.28.

A detailed analysis showing the efficiency of this strategy using a pyramidal hierarchy of hyper-rectangles for component-hypertree construction, compared to other choices of neighborhoods, is given in Chap. 5.



f
1 3 4 0 3 2 2 0 4
$\mathcal{G}_i$
R_1^0 R_1^1 R_1^2 R_1^3 R_1^4 R_1^5 R_1^6 R_1^7 R_1^8
h_1^0 h_1^1 h_1^2 h_1^3 h_1^4 h_1^5 h_1^6 h_1^7 h_1^8
R_2^0 R_2^1 R_2^2 R_2^3 R_2^4
h_2^0 h_2^1 h_2^2 h_2^3 h_2^4
R_3^0 R_3^1 R_3^2
h_3^0 h_3^1 h_3^2
$\mathcal{A}'_i$
0 1 2 3 4 5 6 7 8
1 2 4 6 8
2 4 8
0
1
2
3
4
λ
$i = 1$
$i = 2$
$i = 3$

Figure 4.28: *From top to bottom: the input image f; the sequence of graphs of the hierarchy of partitions $\mathbb{G} = (\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3)$ that defines the neighboring pixels of f; the sequence of neighborhoods $\mathbb{A}' = (\mathcal{A}'_1, \mathcal{A}'_2, \mathcal{A}'_3)$ equivalent to the neighborhoods defined by $\mathbb{G}$ and the resulting component-hypertree of f using $\mathbb{A}' = (\mathcal{A}'_1, \mathcal{A}'_2, \mathcal{A}'_3)$.*

4.8 Attributes

With the theory presented up to this point, one can build an efficient structure to store component-hypertrees. In this section, we focus on fast ways of computing attributes, in order to carry out image processing tasks using the algorithms and structures presented so far.

Attributes are functions $\kappa : CC(f, \mathbb{A}) \to \mathbb{R}$ that, given a CC C, return a numerical value that gives quantitative information about the shape of C. Classic examples of attributes include metrics such as area (number of pixels) or perimeter (number of pixels in the border of the component) of a shape.

For a given attribute att, we denote its value in C as $\kappa_{att}(C)$. For example, the area of a CC C is denoted by $\kappa_{area}(C)$. Some examples of attributes and their definitions are presented below:

- $\kappa_{area}(C) = |C|$, that is, the number of pixels $p \in C$;
- $\kappa_{minX}(C)$: the lowest x coordinate of all pixels $p = (x, y) \in C$;
- $\kappa_{maxX}(C)$: the highest x coordinate of all pixels $p = (x, y) \in C$;
- $\kappa_{width}(C) = \kappa_{maxX}(C) - \kappa_{minX}(C) + 1$.

There are attributes that can take advantage of the inherently hierarchical structure of component-trees and component-hypertrees to be efficiently computed. These attributes are known as *incremental attributes*. In this case, given a node N of a max-tree, an incremental attribute can be computed using only pixels stored in N and the attribute values stored in all children of N. Hence, let $pixels(N)$ denote the pixels stored in the node N. For instance, the attributes presented above are incremental and can be computed in component-trees using the formulas below:

- $\kappa_{area}(N) = |pixels(N)| + \sum\limits_{N_C \in child(N)} \kappa_{area}(N_C)$;
- $\kappa_{minX}(N) = \min\{x : p = (x, y) \in pixels(N), \min\limits_{N_C \in child(N)} \kappa_{minX}(N_C)\}$.

For component-hypertrees, analogous strategies can be used for some attributes. Given a node N with $adj(N) > 1$ (if $adj(N) = 1$, the same strategies used in max-tree can be used, since $part(N) = \emptyset$), for attributes such as $minX$, $maxX$ and $width$, these attributes can be efficiently computed as follows:

- $\kappa_{minX}(N) = \min\limits_{N' \in (child(N) \cup part(N))} \{\kappa_{minX}(N')\}$.

For other attributes, a better strategy consists of taking into consideration all the partial nodes. Since every node is a cluster of nodes obtained from a smaller neighborhood, attributes of nodes of hypertrees can be efficiently computed by only looking at their partial nodes. For example, assuming that all partial nodes of N are stored in $part(N)$, an alternative way of incrementally computing the values of some attributes for nodes N of component-hypertrees with neighborhood index higher than 1 is shown below:

- $\kappa_{area}(N) = \sum\limits_{N_P \in part(N)} \kappa_{area}(N_P)$;
- $\kappa_{minX}(N) = \min\limits_{N_P \in part(N)} \kappa_{minX}(N_P)$.

Due to the storage optimization of minimal hypertrees, some of these partial nodes N_P may not be in $part(N)$, in case the arc that links N_P to N is redundant. For these cases, it is necessary to first find the missing redundant arcs using Alg. 24. From now on, the set of partial nodes of N that include redundant composite nodes will be denoted by $part^*(N)$.

4.8.1 Statistical Measures of Clusters of Nodes

Since nodes with index $i > 1$ are clusters of nodes of previous trees, we can compute additional statistics that measure how the values of an attribute vary in partial nodes. Some of these statistical measurements, like average and variance, can also be incrementally computed by storing some additional information: let N be a node of a component-hypertree with $adj(N) > 1$ and att be any attribute. Then, we will define the accumulated and squared values of the attribute att in N as, respectively:

$$\kappa_{att}^{acc}(N) = \sum_{N_P \in part^*(N)} \kappa_{att}(N_P); \tag{4.17}$$

$$\kappa_{att}^{sq}(N) = \sum_{N_P \in part^*(N)} (\kappa_{att}(N_P)^2). \tag{4.18}$$

Then, we can compute the mean and variance of the values of κ_{att} in the partial nodes of N using the following statistical properties:

$$\overline{\kappa}_{att}(N) = \frac{\kappa_{att}^{acc}(N)}{|part^*(N)|}; \tag{4.19}$$

$$\kappa_{att}^{var}(N) = \frac{\kappa_{att}^{sq}(N)}{|part^*(N)|} - \overline{\kappa}_{att}(N)^2. \tag{4.20}$$

It is important to note that we can change the denominator of these function to compute mean and variance according to some other rules. For example, given a node N, instead of computing the mean of an attribute in N according to their direct partial nodes, we can compute the mean according to the number of CCs that are parts of N in any given neighborhood index. For example, in Fig. 4.29 (right), the node N' has two direct partial nodes, but it is composed of 4 nodes with neighborhood index 1 (N_1 to N_4).

In particular, the number of CCs with neighborhood index 1 will be referred as $partials_1$. It can be computed as an incremental attribute: given a node N of a component-hypertree, we can efficiently compute $\kappa_{partials_1}(N)$ according to the following rules:

$$\kappa_{partials1}(N) = \begin{cases} 1, & \text{if } adj(N) = 1; \\ \sum\limits_{N_C \in part^*(N)} \kappa_{partials_1}(N_C), & \text{otherwise.} \end{cases} \tag{4.21}$$

4.8.2 Attributes Between Nodes

For nodes of component-hypertrees that have more than one partial node, we can also compute attributes that give an insight of how nodes are merged or some values quantifying information between merged nodes.

For example, assuming $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ is defined such that $\mathcal{A}_i = \mathcal{A}(\mathcal{S}_i)$ and $\mathcal{S}_i$ is a $(2i+1) \times 3$ SE for $1 \leq i \leq n$, the neighborhood index will give the size of the horizontal gap between partial nodes. We define this attribute as *horizontal distance*. Analogously, we can obtain the *vertical distance* by changing the SEs to a sequence of $3 \times (2i + 1)$ rectangles; and the *Chebyschev distance* by changing SEs to a sequence of squares with side $2i + 1$, for $1 \leq i \leq n$.

All of these distances can be computed as incremental attributes. For example, let N be a node and assume $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, where $\mathcal{A}_i = (\mathcal{A}(\mathcal{S}_i))$ and $\mathcal{S}_i$ is a $(2i+1) \times 3$ SE, for $1 \leq i \leq n$. Then, we can compute $\kappa_{hdist}^{acc}(N)$, the value of the accumulated horizontal distance in the partial nodes, as follows:

$$\kappa_{hdist}^{acc}(N) = \begin{cases} 0, & \text{if } adj(N) = 1; \\ (adj(N)-1)(|part^*(N)|-1) + \sum\limits_{N_C \in part^*(N)} \kappa_{hdist}^{acc}(N_C), & \text{otherwise}; \end{cases} \tag{4.22}$$

An example is provided in Fig.4.29 (right).

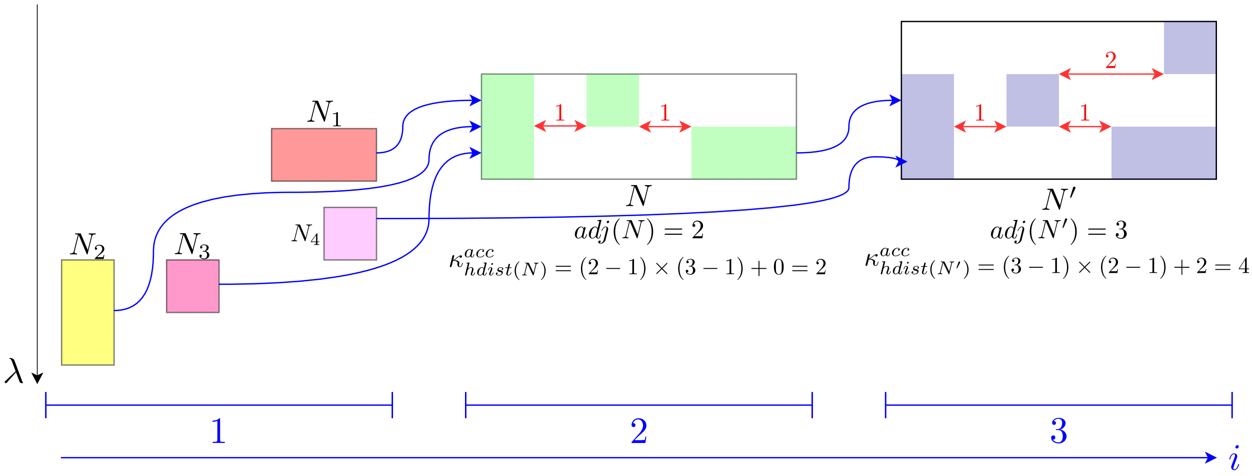

N_1
N_2
N_3
N_4
1
1
N
$adj(N) = 2$
$\kappa_{hdist(N)}^{acc} = (2-1) \times (3-1) + 0 = 2$
2
1
1
N'
$adj(N') = 3$
$\kappa_{hdist(N')}^{acc} = (3-1) \times (2-1) + 2 = 4$
λ
1
2
3
i

Figure 4.29: *Computation of horizontal distance. The computed value is equivalent to the sum of all horizontal spacing (in red) between the isolated components.*

In the next chapter, we provide some examples showing how these attributes can be used to perform text segmentation and classification.

Chapter 5

Experiments

In this chapter, we present theoretical and experimental results regarding computational complexity, time and memory consumption of the proposed algorithms and data structures. Additionally, we also present some text-related applications where computation of attributes can be applied. Finally, we use this application to show how the choice of neighborhoods can affect not only performance, but also how accurate are the computed attributes.

5.1 Analysis

5.1.1 Complexity Analysis

For the complexity analysis, suppose that a sequence of increasing sets of neighborhoods $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ is given and let $\mathbb{A}' = (\mathcal{A}'_1, \ldots, \mathcal{A}'_n)$ be a sequence equivalent to $\mathbb{A}$. As discussed above, time complexity of Alg. 22 is proportional to the number of calls of CONNECT, or to put it another way, it is proportional to $\sum_{i=1}^{n} |\mathcal{A}'_i|$.

This implies that the choice of neighborhoods affects the complexity of the algorithm. In particular, the complexity can be greatly reduced if the neighborhoods of $\mathbb{A}'$ have significantly less elements than the neighborhoods of $\mathbb{A}$.

As seen in the last chapter, for dilation-generated neighborhoods, the neighborhoods $\mathcal{A}'_i \in \mathbb{A}'$ have sizes proportional to the size of the SEs used to generate the neighborhoods. In the general case, this implies $|\mathcal{A}'_i| = \Theta(|\mathcal{D}_f| \cdot |\mathcal{B}_i|)$. By using the specific example of the d-dimensional cubic SEs, each $\mathcal{B}_i$ would have size 2^d, leading to $\sum_{i=1}^{n} |\mathcal{A}'_i| = \Theta(|\mathcal{D}_f| \cdot 2^d \cdot n)$.

However, for this particular example, when every $\mathcal{B}_i$ has size 2^d, then the optimization of Alg. 25 can be used and $|\mathcal{A}_i|$ is approximately $|\mathcal{D}_i| \cdot |\mathcal{B}_i|$. Finally, $|\mathcal{D}_i|$ depends on how many times Line 17 is true at step $i - 1$.

By construction, Line 17 is true if the maximum element of $q \oplus \mathcal{B}_1^{i-1}$ is greater than the maximum element of $p \oplus \mathcal{B}_1^{i-1}$. As a consequence, the number of elements of $\mathcal{D}_i$ depends on the input image and a precise complexity analysis can not be performed.

For this reason, to obtain an estimation of the expected complexity, we use a probabilistic approach. To estimate the expected number of elements in $\mathcal{D}_i$, we assume that the probability of the maximum being in $q \oplus \mathcal{B}_1^{i-1}$ (and not in $p \oplus \mathcal{B}_1^{i-1}$) is proportional to the size of the sets. Note that the size of $\mathcal{B}_1^{i-1}$ in this case is i^d, but since $p \in \{q\} \oplus \mathcal{B}_i$, then $\left(p \oplus \mathcal{B}_1^{i-1}\right)$ and $\left(q \oplus \mathcal{B}_1^{i-1}\right)$ overlap, and the only possibility of Line 17 being true is if the maximum of $\left(p \oplus \mathcal{B}_1^{i-1}\right) \cup \left(q \oplus \mathcal{B}_1^{i-1}\right)$ is in $\left(q \oplus \mathcal{B}_1^{i-1}\right) \setminus \left(p \oplus \mathcal{B}_1^{i-1}\right)$.

Since the size of $\left(q \oplus \mathcal{B}_1^{i-1}\right) \setminus \left(p \oplus \mathcal{B}_1^{i-1}\right)$ is approximately $d \times i$ and the size of $\left(p \oplus \mathcal{B}_1^{i-1}\right) \cup \left(q \oplus \mathcal{B}_1^{i-1}\right)$ is approximately i^d, we have that:

$$|\mathcal{D}_i| \approx \frac{d \cdot i}{i^d} \cdot |D_f| = \frac{d}{i^{d-1}} \cdot |D_f|$$

By choosing a fixed d, we can estimate $\sum\limits_{i=1}^{n} |\mathcal{D}_i|$. For example, for $d = 2$, then $\sum\limits_{i=1}^{n} \frac{1}{i}$ can be approximated using the fact that $\int_1^n \frac{1}{i} di \approx \ln n$. If $d > 2$, then it grows even slower. Thus, for this particular example, the expected number of calls of CONNECT is proportional to $\Theta(|D_f| \cdot |\mathcal{B}_i| \cdot \ln n)$, that is to say, when the number of neighborhoods is increased linearly, the complexity of the construction algorithm only increases by a logarithmic factor of n.

This computational complexity already ensures that component-hypertrees can be built in a reasonable time even for a large number of neighborhoods. However, this complexity can be improved even further when neighborhoods based on a hierarchy of partitions are taken into consideration.

For the specific case of a pyramidal hierarchy, we have $|\mathcal{A}'_i| = \Theta(|\mathcal{H}_i| \cdot |\mathcal{S}_i|)$. In a simple case where $t = (2, \dots, 2)$ and $\mathcal{S}_i$ defines $(3^d - 1)$-adjacency, we have $\sum_{i=1}^{n} |\mathcal{A}'_i| = \mathcal{O}(|\mathcal{D}_f| \cdot (1 + \frac{1}{2^d - 1}) \cdot 3^d) = \mathcal{O}(|\mathcal{D}_f| \cdot 3^d)$. For a small constant d, this implies that $\sum_{i=1}^{n} |\mathcal{A}'_i|$ is linear on the size of the domain of f. It is important to emphasize, however, that this gain in performance has an impact in the precision of the computed attributes, as we will see later in this chapter.

To conclude our analysis in terms of computational complexity, the exact cost of the CONNECT procedure is difficult to calculate. When CONNECT is called to two pixels p, q, the number of recursive calls is proportional to the paths linking p to c and q to c in the parent array, where c is the first common ancestor of c. It is difficult to compute, on average, the size of these paths, but we can be sure that both of them are bounded by K. Hence, in the worst case, CONNECT runs in $\mathcal{O}(K)$, but on average the procedure is faster than that because, after the updates performed in the array by the procedure, a subsequent call of CONNECT with the same parameters would be faster because the common ancestor of p and q in the second call would now be either p or q.

Finally, the computational complexity of the node and arc allocation procedures are proportional to the sizes of *newNodesList* and *parentUpdateList*. These sets are composed of canonical elements, and the number of canonical elements is bounded by $|D_f|$. Hence, in the worst case, these algorithms run in $\mathcal{O}(|D_f|)$ for each step $1 \leq i \leq n$. However, on average, the number of elements in these sets is usually much lower.

5.1.2 Experimental Results

As emphasized by the last section, both the dilation-generated strategy and the pyramidal strategy are more efficient than a general approach for building component-hypertrees, with the counterpart of a more restricted choice of neighborhoods.

For the dilation-based strategy, when the property $\mathcal{B}_i \subseteq \mathcal{B}_{i-1}$ is satisfied, the size of the neighborhoods can grow, at most linearly, for each dimension; this implies that a large number of SEs is required to connect distant nodes. By contrast, the pyramidal approach reduces the domain in each dimension geometrically; this implies that a much lower n can be used. This observation, added to the fact that the pyramidal strategy already has a better computational complexity, indicates that building the component-hypertree using this approach is also much faster than the dilation-based strategy.

In order to experimentally corroborate these observations, a set of about 500 random images was taken from the ICDAR 2017 Robust Reading Challenge Dataset [NYB+17] for time consumption experiments. These images were divided into 10 subsets $I_1, \dots, I_{10}$, where I_j consists of images with size between $j - 1$ and j megapixels (MP). For each image, component-hypertrees using both the pyramidal hierarchy strategy and the dilation-based strategy were computed. Component-hypertree computation includes the update of the max-tree, allocation of nodes and arcs and computation of the attributes number of CCs, height, width and spacing [MAS+19b]. For pyramidal hierarchies, $t = (2, 2)$ was used and, for dilation-based neighborhoods, $\mathcal{B}_i$ was the 2×2 SE for all $1 \leq i \leq n$.

For the dilation-based strategy, 3 configurations were used: the first with $n = 11$, to match the n used for the pyramidal strategy, and the other two with $n = 512$. One of these configurations with $n = 512$ updates the component-hypertree only at steps $1 \leq i \leq n$ that were powers of 2, to try to extract similar nodes to those obtained using the hierarchical approach, whereas the other one updates it at every step.

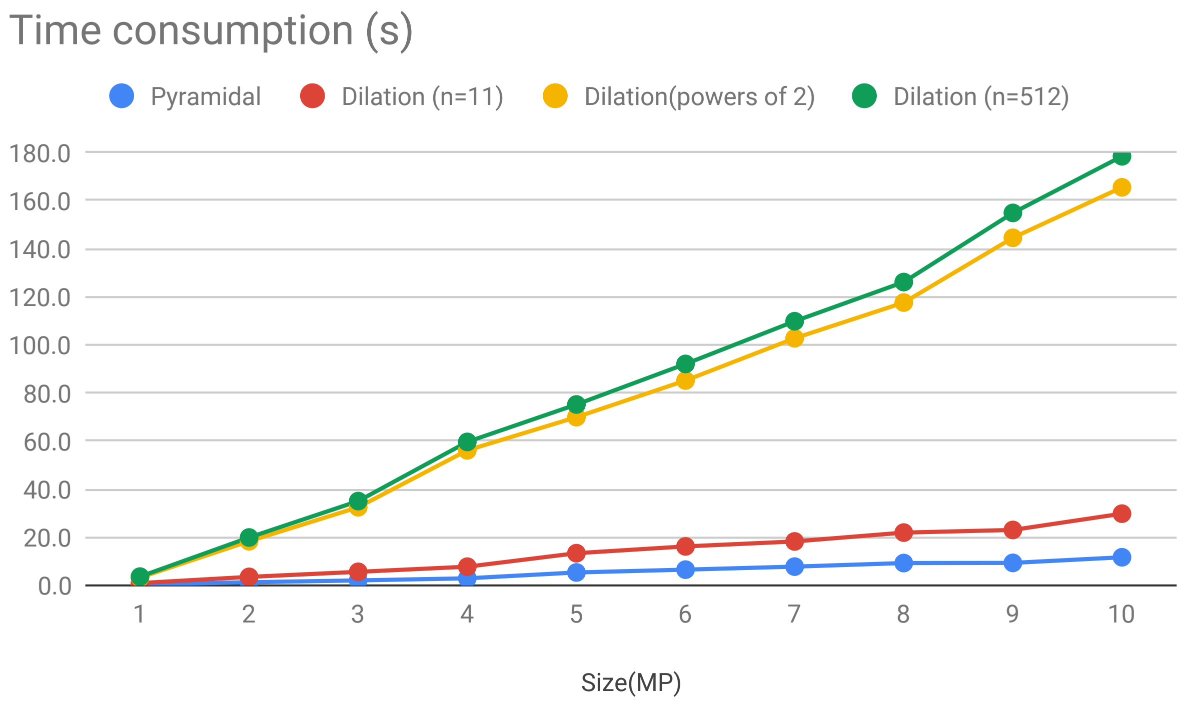


Figure 5.1: *Comparison of times between the pyramidal hierarchy strategy (blue line, $n = 11$) and the dilation-based strategy using 3 different configurations: $n = 11$ (red), $n = 512$ allocating updating the hypertree at powers of 2 (yellow) and $n = 512$ allocating all nodes (green).*

For each subset of images, the average time, in seconds, was computed. The implementation was written in Java and was tested in a i7 2.6GHz processor with 16GB of RAM. The results are presented in Fig. 5.1.

From the graph, it is possible to see that the pyramidal strategy is about 2.5 times faster than the dilation-based approach, for this fixed $n = 11$. However, extraction of nodes with similar distances to the ones obtained by the pyramidal approach requires the usage of $n = 512$ when using the dilation-based strategy, which drastically increases processing time, as shown by the yellow and green lines in the graph. In fact, for the 10MP images tested, the pyramidal strategy was more than 14 faster than those approaches, and the difference would only increase as the size of the images increases as well.

A drawback of a geometric growth in the size of the neighborhoods is that many neighborhoods are skipped, which impacts in loss of precision in terms of distance of merged nodes. In fact, while the dilation-based strategy can compute the exact Chebyshev distance between two pixels [MAS+19a], the Chebyshev distance of two pixels that connect at step i using the pyramidal approach can be any value between $(t')^{i-2} + 1$ up to $(t')^i - 1$, assuming $t = (t', \dots, t')$.

Even so, it is still possible to obtain an estimate of the distance between merged nodes. Experiments performed in about 15 000 nodes extracted from images of the ICDAR database showed that, for $t = (2, 2)$, the value $2^{(i-1)}$ is a good estimate of the distance of nodes merged at step i, $2 \leq i \leq n$. Comparing to the exact values, the estimated distances differed by about 8% on average, with more of half of the nodes differing by less than 5%, more than 90% of the nodes differing by less than 25% and about 0.1% of the nodes with a difference of more than 50% compared to their real value. Moreover, the difference to the exact value tends to decrease as the number of merged nodes increases.

Even though the computed distance is not exact, this degree of precision can still be useful for some applications that use distance as a feature. For instance, it is still possible to extract different scales of objects only by restricting the values of the variance of the distance of the merged nodes.

One example is shown in Fig. 5.2.

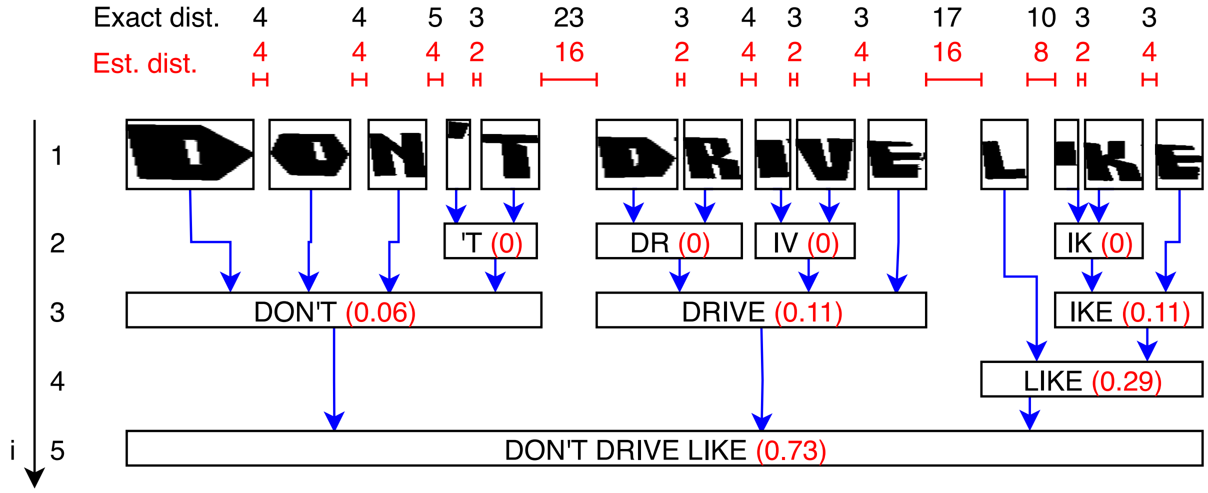


Figure 5.2: *Computation of horizontal spacing using the pyramidal approach. In the top row, the exact horizontal distance between CCs, in pixels, is shown. In the row below, in red, the estimated values are displayed. For this example, $t = (2, 1)$ is used to merge nodes aligned horizontally, so the estimated distance is 2^{i-1}. At the bottom, some nodes of the hypertree are depicted, showing how letters are merged as i increases. Each node is indicated by its content (in black) and its variance of the horizontal spacing between merged nodes (in red). The displayed value is obtained by dividing the variance of the horizontal spacing by the average height of the merged nodes, to add scale invariance. Note that the variance of horizontal spacing can be useful to differentiate words from lines of texts.*

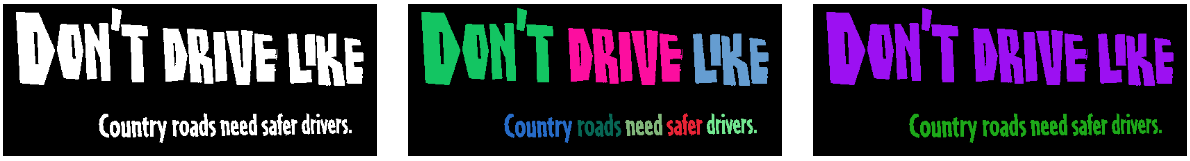

DON'T DRIVE LIKE
Country roads need safer drivers.
DON'T DRIVE LIKE
Country roads need safer drivers.
DON'T DRIVE LIKE
Country roads need safer drivers.

Figure 5.3: *Extraction of letters, words and lines of text using different thresholds for the variance of the distance of merged nodes, based on the component-hypertree from Fig. 5.2. For example, by selecting nodes with variance less than* 0.3*, words can be extracted (middle column, where elements with the same color belong to the same node) and, if no threshold is given, then entire lines of text can be obtained (right).*

5.1.3 Memory Consumption

In this section, we analyze how much memory is saved by using our minimal representation of component-hypertrees, compared to the complete representation and a naive strategy that only removes repeated CCs if they have the same neighborhood index. In other words, in the complete representation, all CCs of all complete component-trees from $\mathcal{A}_1$ to $\mathcal{A}_n$ are stored in memory and, in this naive representation, only nodes of (non-complete) component-trees are stored.

For these tests, images from the Born Digital set of the ICDAR 2011 [KMM$^+$11] were used, which consists of a set of 410 images of webpages and e-mail attachments. For each image of this set, we used a sequence of square neighborhoods $\mathbb{A} = (\mathcal{A}_1, \dots, \mathcal{A}_n)$, with $n = 50$, where $\mathcal{A}_i$ is defined using a SE of size $(2i + 1) \times (2i + 1)$. Then, we computed the average of the number of nodes and arcs for each i for the 3 structures: the complete hypertree, the naive implementation and our minimal representation. The results in Fig 5.4 shows that the minimal representation can save a considerable amount of memory compared to the other ones. On average, for the cases tested, we have a saving of about 50% compared to the naive implementation and 80% compared to the complete hypertree for $n = 10$. Additionally, it can be seen from the graph that the number of nodes and arcs remains almost constant in the minimal representation, implying that the numbers of redundant nodes and arcs tend to increase as n increases. Since these nodes and arcs are still allocated in the non-minimal hypertrees, the gains in terms of percentage is even greater for higher values of n.

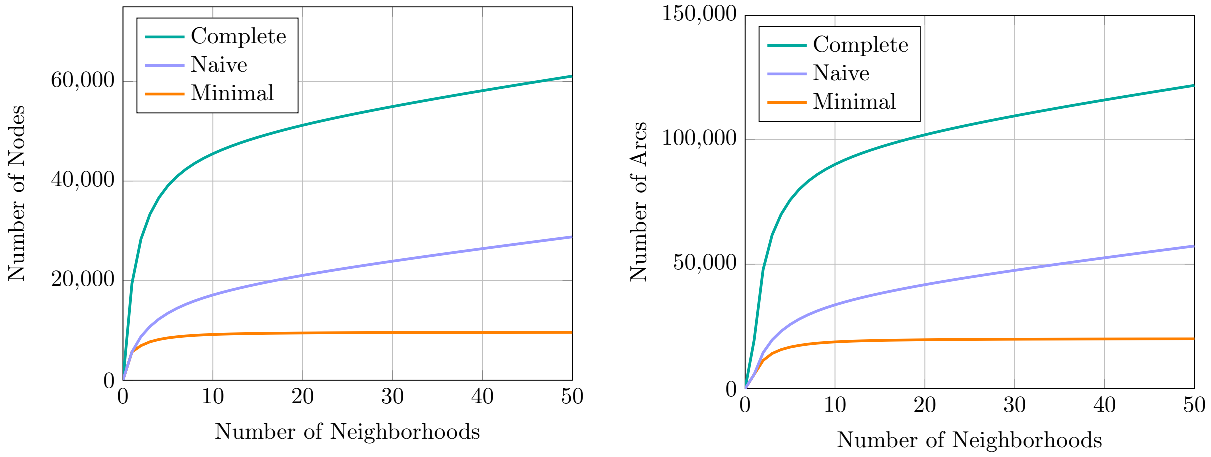


Figure 5.4: *Left: average number of nodes for each representation for up to 50 neighborhoods. Right: the same experiments but for the average number of arcs.*

5.2 Applications

5.2.1 Segmentation of Words in an Image Containing Text

As shown in Fig. 5.3, based on the assumption that spaces between words are wider than spaces between letters of a same word, we can develop a strategy to extract words from an image containing text based on how far apart the CCs are.

Naturally, the results can be optimized with the usage of more attributes. Hence, we chose images from the ICDAR Born-Digital database, which consists of small images extracted from e-mail and web pages that contain texts, and the main goal is to segment the words of these images.

For this proof of concept, the following assumptions about the texts were made:

1. Texts have a visually good contrast with the background, and are not very noisy;
2. Letters of the same word have a similar color;
3. Letters are composed of a single CC when using 8-connected neighborhood;
4. Words are aligned on the horizontal axis;
5. Letters of the same word have similar heights;
6. Letters of the same word have a consistent distance among themselves, which should be smaller than the distance between two letters from different words.

For each of these suppositions, the following design choices were made, respectively:

1. Images with low contrast or that suffered from effects of lightning, noise or uneven background were not used;
2. Using the component-hypertree. This is an essential assumption when using component-hypertrees, in order to have nodes that contain words without noise;
3. $\mathcal{A}_1$ is the SE that defines 8-connected neighborhood;

4. $\mathcal{A}_i$ for $i > 1$ only grows horizontally;

5. For each node N of the component-hypertree, we computed their height, the average and the variance of the heights of their parts;

6. For each node N of the component-hypertree, the following attributes were computed:

   (a) $\kappa_{avgdist}(N)$, the average of the distances of all nodes N' that are parts of N with $i(N') = 1$, in other words, they are descendants of N with $f(N') = f(N)$ and $i(N') = 1$;

   (b) $\kappa_{vardist}(N)$, the variance of the distances of all nodes N' from the previous item.

Since average and variance are not scale invariant, we decided to normalize these values by using the following formula:

$$\kappa_{normdist}(N) = \frac{\kappa_{vardist}(N)}{\kappa_{avgdist}(N)^2} \tag{5.1}$$

Clearly, $\kappa_{normdist}(N)$ is directly proportional to $\kappa_{vardist}(N)$. In other words, the smaller the value of $\kappa_{normdist}(N)$, the better the consistency between the distance of the latters N.

Empirical tests suggest $\kappa_{normdist}(N) \leq 0.08$ is a good threshold to segment words. For normalized height variation, we used a threshold of 0.4, because of the difference of heights within non-capitalized letters. The normalization was done the same way: we divide variance of the heights by the average of the heights squared.

Table 5.1 shows same examples of results obtained with this simple strategy. For these tests, we reduced the number of gray-levels to $K = 4$ in order to improve performance and reduce noise from gray-level variation.

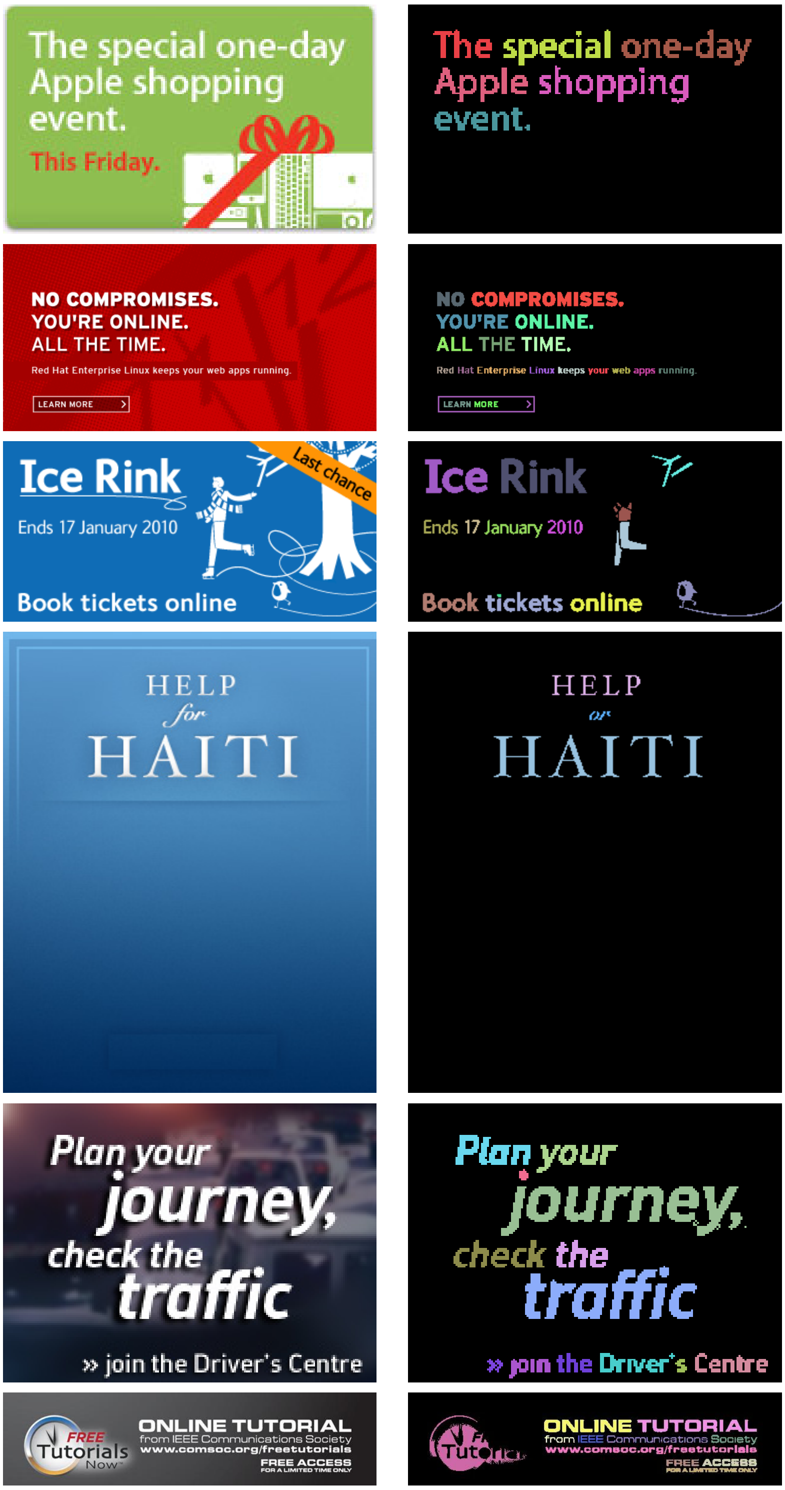

The special one-day Apple shopping event.
This Friday.
The special one-day Apple shopping event.
NO COMPROMISES. YOU'RE ONLINE. ALL THE TIME.
Red Hat Enterprise Linux keeps your web apps running.
LEARN MORE
NO COMPROMISES. YOU'RE ONLINE. ALL THE TIME.
Red Hat Enterprise Linux keeps your web apps running.
LEARN MORE
Ice Rink
Ends 17 January 2010
Book tickets online
Last chance
Ice Rink
Ends 17 January 2010
Book tickets online
HELP for HAITI
HELP or HAITI
Plan your journey, check the traffic
» join the Driver's Centre
Plan your journey, check the traffic
» join the Driver's Centre
FREE Tutorials Now
ONLINE TUTORIAL
from IEEE Communications Society
www.comsoc.org/freetutorials
FREE ACCESS
FOR A LIMITED TIME ONLY
ONLINE TUTORIAL
from IEEE Communications Society
www.comsoc.org/freetutorials
FREE ACCESS
FOR A LIMITED TIME ONLY

Table 5.1: *Segmentation of words using our method. In the left, we show the original images and, in the right, the result of our word segmentation algorithm. Letters with the same color belong to the same node.*

5.2.2 Segmentation of Oriental Ideograms

Another application that can take advantage of component-hypertrees is segmentation of Oriental ideograms, such as Chinese, Japanese or Korean characters. Contrary to Latin alphabet, these ideograms can be composed of small parts that are not connected using 8-connected neighborhood, but the use of a bigger connectivity may connect these parts in a single node.

Additionally, the following suppositions were made about these types of text:

1. Texts have a visually good contrast with the background, and are not very noisy;
2. Letters of the same word have a similar color;
3. As stated above, letters are not composed of a single CC when using 8-connected neighborhood;
4. Words can be aligned in both the horizontal axis or the vertical axis;
5. Each letter can be contained in a square shape, and letters from the same word or line of text fit within squares with similar sides.

Based on these suppositions, we made the following design choices to our application:

1. Images with low contrast or that suffered from effects of lightning, noise or uneven background were discarded;
2. Using the component-hypertree;
3. $\mathcal{A}_1$ is chosen as the SE that defines 8-connected neighborhood, in order to obtain the parts of a letter, but we do not assume each node is already a complete letter;
4. $\mathcal{A}_i$ for $i > 1$ grows both horizontally and vertically. This is important to both join parts of the same letter but also because we do not know the orientation of the text;
5. For each node N, we compute the following attribute:

$$\kappa_{square}(N) = \begin{cases} \text{true}, & \text{if } \frac{10}{14} \leq \frac{\kappa_{height}(N)}{\kappa_{width}(N)} \leq \frac{14}{10} \\ \text{false}, & \text{otherwise} \end{cases}$$

Using these assumptions, we obtain nodes with $adj(N) = n$ and decompose them into smaller parts that fit within a square, using the κ_{square} attribute.

The results for image with horizontal text is shown in Table 5.2. Extracted characters have their bounding box drawn. Letters without their bounding boxes were selected as candidates but discarded because they did not fit a square shape.

The images were obtained from the Robust Reading Competition from the training database for ICDAR 2017, and the regions with text were manually extracted.

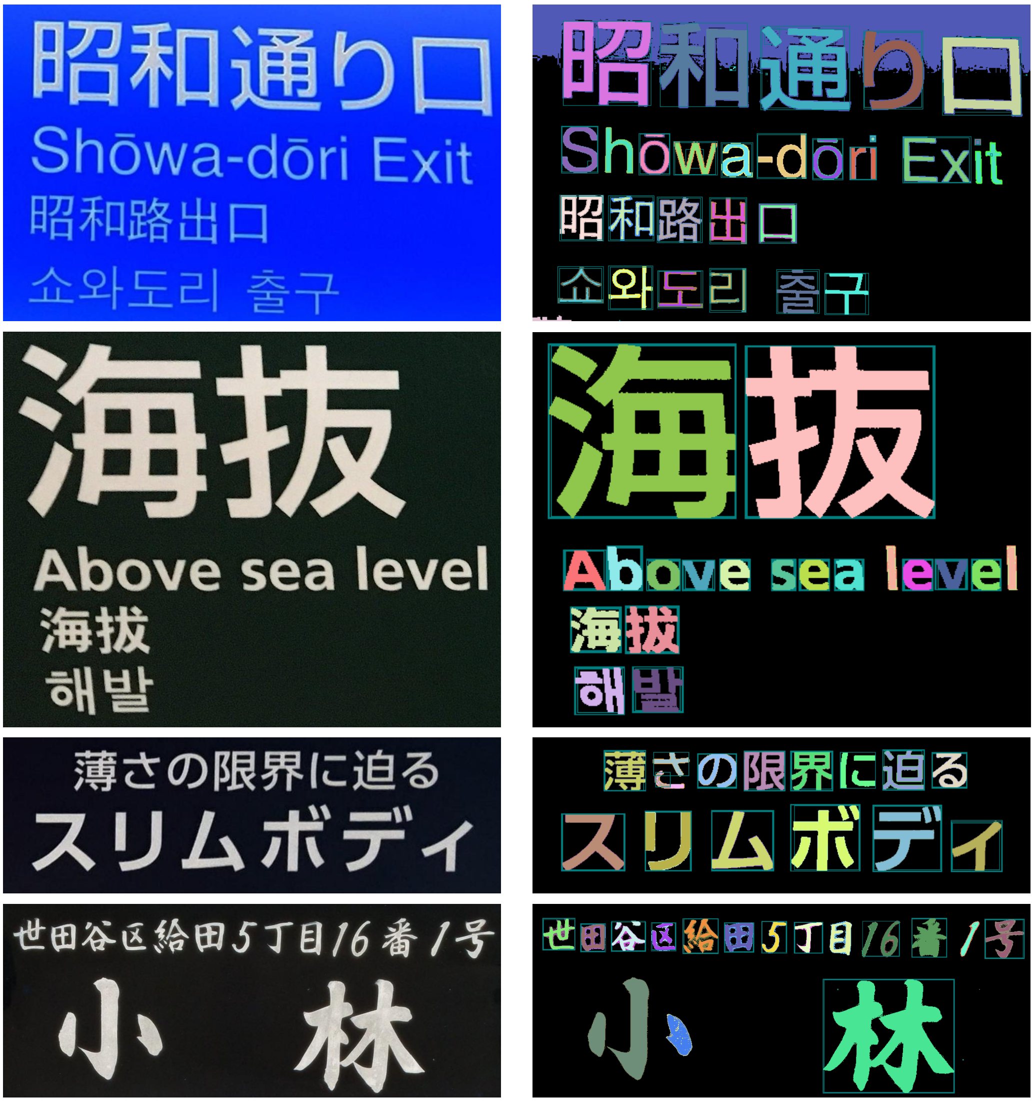


Table 5.2: *Extraction of Oriental ideograms using our method for texts with horizontal orientation. In the left we show the original image and in the right we show our results, with each selected node represented by a color and its bounding box.*

The same application, without any modification, can be applied to text with a vertical orientation, as we show in Table 5.3.

Since we only we use the κ_{square} attribute, the results are not as robust as our word segmentation application. The results are noisier, and in some cases, some parts of some ideograms are missing or merged (for example, the letter 小 was divided in two in the last image from Fig 5.2 and the letters 銀座 , although recognized individually as candidates for letters, became a single node in the last image from Fig 5.3 since our threshold is too lenient). Nevertheless, the results obtained have an acceptable rate of success and can be applied to multiple languages: the first two images from Table 5.2 have texts in Japanese, Chinese and Korean, all correctly segmented.

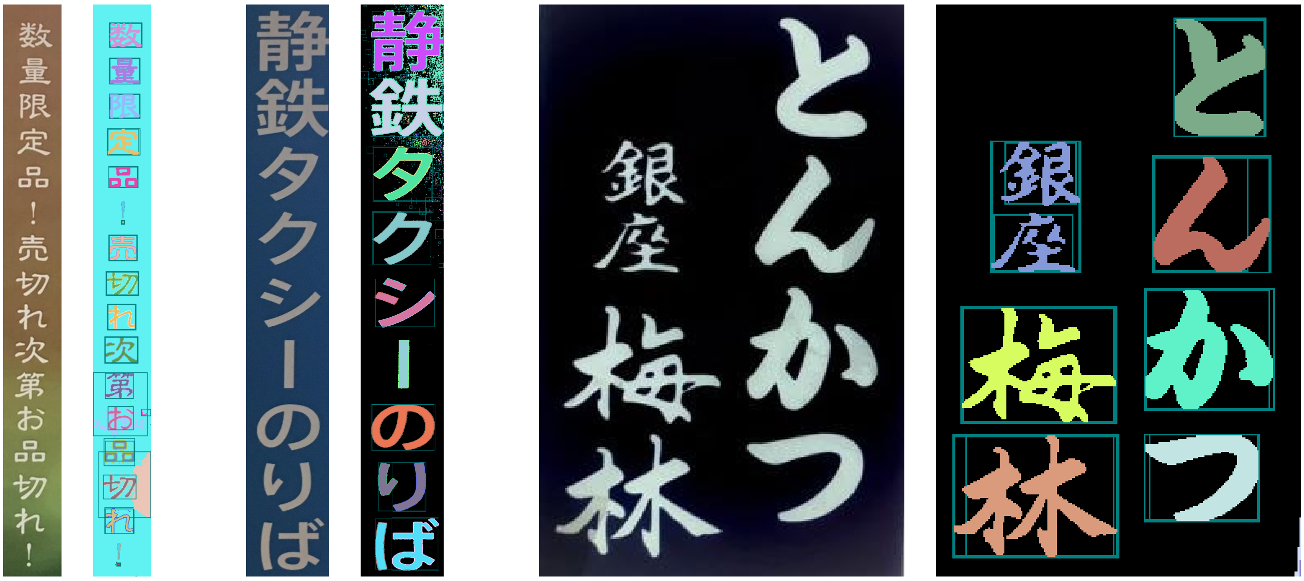


Table 5.3: *Extraction of Oriental ideograms using our method for texts with vertical orientation.*

Chapter 6

Conclusion

In this chapter, we finish this thesis by presenting perspective works and our conclusions.

6.1 Perspective Works

In terms of perspective works, there are some topics that need to be further investigated. One such example consists of analyzing other types of neighborhoods similar to hierarchy of partition, but that uses a downsampling function with a non-integer scale. In this case, the obtained neighborhood does not consist of a hierarchy of partitions, but the same computational complexity is kept and the error in terms of computed distance can be reduced. Some experimental results suggest this approach may work, but further studies would be necessary to understand if important properties are lost.

Another topic related to the choice of neighborhoods consists of further investigating which neighborhoods can be generated using the dilation-based approach. In this sense one could, for example, choose directly which neighborhoods to use when extracting connected components and the generated sequence would be computed based on the sequence chosen by the user.

Still related to the topic of neighborhoods, it may also be worthy to investigate connections between different types of neighborhoods. For instance, it is likely that dilation-generated neighborhoods are as a subset of mask-based neighborhoods, but with additional properties. In this sense, if a more efficient algorithm for mask-based neighborhoods is developed, it could also benefit the algorithm for dilation-based neighborhoods.

Attribute computation and applications were also not explored in-depth in this manuscript. In particular, many topics related to component-trees can be extended to component-hypertrees, such as: development of new attributes that can be efficiently computed, pruning or removal of nodes and strategies for node selection based on computed attributes. This last problem seems particularly important because, since component-hypertrees are directed acyclic graphs, two nodes can intersect without one being included in the other. This issue does not happen in usual component-trees, so new strategies need to be developed to deal with this case.

Finally, a final topic that is worth considering is the extension of the developed theory for high-dimensional images or general graphs. The first case seems straightforward, but the latter does not seem to be an easy task, since many properties of gray-level images and the neighborhoods were used to develop the algorithms. However, it may be possible to extend the theory for other domains that share similar properties to the ones used in the proposed algorithms.

6.2 Final Words

This thesis dealt with the problem of building component-hypertrees of gray-level images according to a sequence of increasing neighborhoods. In essence, component-hypertrees can be seen as an extension of component-trees for multiple neighborhoods. This means that, if many neighborhoods are used, a component-hypertree can take a long time to be built and use too much memory to be stored. Therefore, in this manuscript, we focused on designing efficient algorithms and data structures for component-hypertree computation, storage and manipulation.

In order to do that, we first presented a theoretical and algorithmic background in component-trees and explained how to modify and extend these concepts to obtain an approach that is optimized for component-hypertree construction. To optimize our algorithm even further, it was noted that the computational complexity was directly tied to the neighborhoods used to extract the nodes, so an additional section was dedicated to explore different choices of neighborhoods and explain how they can be optimized for this particular problem. This led to the development of dilation-generated neighborhoods and neighborhoods based on hierarchy of partitions. Then, to reduce memory usage, we proposed the minimal-hypertrees, an optimized data structure that avoids storing repeated nodes and also removes most arcs that give redundant information regarding inclusion relation between nodes.

To corroborate the efficiency of the proposed approaches, we presented analyses of computation complexity, memory usage and showed some experimental results, proving that our algorithms can save a considerable amount of time and memory when compared to strategies that do not use the optimizations proposed in this manuscript. It is also worth mentioning that the proposed methods comprise three publications accepted in some of the most important conferences and journals [MAS+19b, MAS+19a, MPAH20] in the field of Mathematical Morphology, which further endorses the effectiveness of our strategies.

Finally, at the end of this thesis, a text-extraction application was presented. This is an example of a problem where component-hypertrees are particularly suitable, since texts are usually clustering of smaller objects that can be merged together by considering multiple increasing neighborhoods. Then, we conclude this manuscript by presenting some perspective works and with these final words.

Bibliography

[BGL$^+$] Christophe Berger, Thierry Géraud, Roland Levillain, Nicolas Widynski, Anthony Baillard and Emmanuel Bertin. Effective component tree computation with application to pattern recognition in astronomical imaging. In *2007 IEEE International Conference on Image Processing*, volume 4. IEEE. 2, 20

[BNG02] Ulisses Braga-Neto and John Goutsias. Connectivity on complete lattices: New results. *Computer Vision and Image Understanding*, 85(1):22–53, 2002. 3, 7

[CG14] Edwin Carlinet and Thierry Géraud. A comparative review of component tree computation algorithms. *IEEE Transactions on Image Processing*, 23(9):3885–3895, 2014. 2

[CG15] Edwin Carlinet and Thierry Géraud. MToS: A tree of shapes for multivariate images. *IEEE Transactions on Image Processing*, 24(12):5330–5342, 2015. 4

[CO15] Juan Climent and Luiz S. Oliveira. A new algorithm for number of holes attribute filtering of grey-level images. *Pattern Recognition Letters*, 53:24–30, 2015. 3

[Gra71] Stephen B. Gray. Local properties of binary images in two dimensions. *IEEE Transactions on Computers*, 100(5):551–561, 1971. 2

[Jon99] Ronald Jones. Connected filtering and segmentation using component trees. *Computer Vision and Image Understanding*, 75(3):215–228, 1999. 1, 2

[KMM$^+$11] Dimosthenis Karatzas, Sergi R. Mestre, Joan Mas, Farshad Nourbakhsh and Partha P. Roy. ICDAR 2011 robust reading competition - challenge 1: Reading text in born-digital images (web and email). In *2011 International Conference on Document Analysis and Recognition*, pages 1485–1490, Sep. 2011. 84

[MAH15] Alexandre Morimitsu, Wonder A. L. Alves and Ronaldo F. Hashimoto. Incremental and efficient computation of families of component trees. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 681–692. Springer, 2015. 3, 4, 70, 71

[MAS$^+$19a] Alexandre Morimitsu, Wonder A. L. Alves, Dennis J. Silva, Charles F. Gobber and Ronaldo F. Hashimoto. Incremental attribute computation in component-hypertrees. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 150–161, 2019. 4, 39, 83, 92

[MAS$^+$19b] Alexandre Morimitsu, Wonder A. L. Alves, Dennis J. Silva, Charles F. Gobber and Ronaldo F. Hashimoto. Minimal component-hypertrees. In *International Conference on Discrete Geometry for Computer Imagery*, pages 276–287, 2019. 4, 39, 82, 92

[MDA$^+$08] Petr Matas, Eva Dokladalova, Mohamed Akil, Thierry Grandpierre, Laurent Najman, Martin Poupa and Vjaceslav Georgiev. Parallel algorithm for concurrent computation of connected component tree. In *International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 230–241. Springer, 2008. 2

[MG00] Pascal Monasse and Frédéric Guichard. Fast computation of a contrast-invariant image representation. *IEEE Transactions on Image Processing*, 9(5):860–872, 2000. 3

[MPAH20] Alexandre Morimitsu, Nicolas Passat, Wonder A.L. Alves and Ronaldo F. Hashimoto. Efficient component-hypertree construction based on hierarchy of partitions. *Pattern Recognition Letters*, 135:30–37, 2020. 4, 39, 75, 92

[NC06] Laurent Najman and Michel Couprie. Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15(11):3531–3539, 2006. 2, 20

[NM12] Lukáš Neumann and Jiří Matas. Real-time scene text localization and recognition. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3538–3545. IEEE, 2012. 3

[NYB$^+$17] Nibal Nayef, Fei Yin, Imen Bizid, Hyunsoo Choi, Yuan Feng, Dimosthenis Karatzas, Zhenbo Luo, Umapada Pal, Christophe Rigaud, Joseph Chazalon et al. Icdar2017 robust reading challenge on multi-lingual scene text detection and script identification-rrc-mlt. In *2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR)*, volume 1, pages 1454–1459. IEEE, 2017. 82

[OW07a] Georgios K. Ouzounis and Michael H. F. Wilkinson. Mask-based second-generation connectivity and attribute filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):990–1004, 2007. 3

[OW07b] Georgios K. Ouzounis and Michael H. F. Wilkinson. A parallel implementation of the dual-input max-tree algorithm for attribute filtering. In *ISMM*, pages 449–460, 2007. 27

[PN11] Nicolas Passat and Benoît Naegel. Component-hypertrees for image segmentation. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, volume 6671, pages 284–295. Springer, 2011. 1, 3, 4, 12

[PN14] Nicolas Passat and Benoît Naegel. Component-trees and multivalued images: Structural properties. *Journal of Mathematical Imaging and Vision*, 49(1):37–50, 2014. 4

[PNK19] Nicolas Passat, Benoît Naegel and Camille Kurtz. Component-graph construction. *Journal of Mathematical Imaging and Vision*, 61(6):798–823, 2019. 4

[PNR$^+$11] Nicolas Passat, Benoît Naegel, François Rousseau, Mériam Koob and Jean-Louis Dietemann. Interactive segmentation based on component-trees. *Pattern Recognition*, 44(10):2539–2554, 2011. 3

[SAMH16] Dennis J. Silva, Wonder A. L. Alves, Alexandre Morimitsu and Ronaldo F. Hashimoto. Efficient incremental computation of attributes based on locally countable patterns in component trees. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 3738–3742. IEEE, 2016. 3

[Ser98] Jean Serra. Connectivity on complete lattices. *Journal of Mathematical Imaging and Vision*, 9(3):231–251, 1998. 3

[SG00] Philippe Salembier and Luis Garrido. Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. *IEEE Transactions on Image Processing*, 9(4):561–576, 2000. 3

[SOG98] Philippe Salembier, Albert Oliveras and Luis Garrido. Antiextensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998. 1, 2

[Soi07] P. Soille. On genuine connectivity relations based on logical predicates. In *14th International Conference on Image Analysis and Processing*, pages 487–492, 2007. 3

[Son07] Yuqing Song. A topdown algorithm for computation of level line trees. *IEEE Transactions on Image Processing*, 16(8):2107–2116, 2007. 3

[SW09] Philippe Salembier and Michael H. F. Wilkinson. Connected operators. *IEEE Signal Processing Magazine*, 26(6), 2009. 3

[Tar75] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975. 2, 15

[WGH+08] Michael H. F. Wilkinson, Hui Gao, Wim H. Hesselink, Jan-Eppo Jonker and Arnold Meijster. Concurrent computation of attribute filters on shared memory parallel machines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(10):1800–1813, 2008. 2