Minimal Component-Hypertrees

Alexandre Morimitsu[1(✉)], Wonder Alexandre Luz Alves[2], Dennis Jose Silva[1], Charles Ferreira Gobber[2], and Ronaldo Fumio Hashimoto[1(✉)]

[1] Department of Computer Science, Institute of Mathematics and Statistics, Universidade de São Paulo, São Paulo, Brazil
{alexandre.morimitsu,ronaldo}@usp.br

[2] Informatics and Knowledge Management Graduate Program, Universidade Nove de Julho, São Paulo, Brazil

Abstract. Component trees are interesting structures of nested connected components, efficiently represented by max-trees, used to implement fast algorithms in Image Processing. In these structures, connected components are constructed using a single neighborhood. In recent years, an extension of component trees, called component-hypertrees, was introduced. It consists of a sequence of component trees, generated from a sequence of increasing neighborhoods, in which their connected components are also hierarchically organized. Although this structure could be useful in applications dealing with clusters of objects, not much attention has been given to component-hypertrees. A naive implementation can be costly both in terms of time and memory. So, in this paper, we present algorithms and data structures to efficiently compute and store these structures without redundancy obtaining a minimal representation of component-hypertrees. Experimental results using our efficient algorithm show that the number of nodes is reduced by approximately 70% in comparison to a naive implementation.

Keywords: Mathematical Morphology · Component-hypertree · Component tree · Connected component · Connected operators

1 Introduction

In recent years, component trees have received increasing attention in Image Processing, particularly in Mathematical Morphology, since they can be used for many tasks such as image filtering [9] and shape recognition [11]. A component tree represents a gray-level image by storing connected components of its level sets for a given connectivity and relating them by the inclusion relation. Although multiple types of connectivity can be used, like 4 and 8-connectivity, mask-based [6] and hyper-connectivity [7], only a single tree can be extracted, and thereby limiting the number of problems that can be solved by this approach.

Recently, Passat and Naegel [8] proposed a structure which represents a gray-level image by a set of component trees built with increasing neighborhoods called component-hypertree. Due to increaseness of the neighborhoods,

https://doi.org/10.1007/978-3-030-14085-4_22

component-hypertrees relate connected components of different neighborhoods according to their inclusion. For example, in text extraction problem, a word and its letters are nodes of a component-hypertree and they are related by the inclusion relation.

In the original paper [8], Passat and Naegel provided a theoretical background for mask-based connectivities. Although the structure is explained, no explicit optimized algorithm for component-hypertrees construction is provided. Also, some theoretical explanations are given about how to perform simplification in the graph, but many repeated nodes still remain. In [3], algorithms are provided for a specific type of dilation-based connectivities. That work showed an efficient way of computing hypertrees for multiple neighborhoods but it did not focus on the problem of storing hypertrees efficiently.

Using an approach that stores the CCs of all component trees of all neighborhoods, the component-hypertree would have a prohibitive memory usage. However, most of the CCs in different trees are repeated, meaning a considerable amount of memory can be saved by smartly allocating only relevant nodes and arcs of the hypertree. So in this paper, we present the minimal component-hypertree: a data structure which efficiently stores the component-hypertree by keeping only the minimum number of nodes and arcs needed to efficiently recover all CCs. In this structure, repeated CCs obtained from different level sets and neighborhoods are discarded, meaning that each CC is stored only once.

For that, we adopt the following strategy: before explaining our data structure and algorithms, we first recall the theory behind component trees and component-hypertrees in Sect. 2. We present our main contributions in Sect. 3 where we show the theory and the algorithms used to build minimal component-hypertrees. In Sect. 4, we show some experiments that quantify the obtained memory saving using our strategy. Finally, we conclude our paper in Sect. 5.

2 Background

2.1 Images and Connectivity

In this study, we consider an image f as a mapping from a subset of a rectangular grid $\mathcal{D} \subset \mathbb{Z}^2$ to a set of gray-levels $\mathbb{K} = \{0, 1, \dots, K-1\}$. An element $p \in \mathcal{D}$ is called pixel and the gray-level associated with p in image f is denoted by $f(p)$. In addition, a pixel $p \in \mathcal{D}$ can be a neighbor of others pixels of $\mathcal{D}$ in their surroundings. This notion of neighborhood can be defined by a structuring element (SE), that is, a subset $\mathcal{S} \subset \mathbb{Z}^2$. Thus, given a set of pixels $P \subseteq \mathcal{D}$ and a SE $\mathcal{S}$, we define $\mathcal{P} \oplus \mathcal{S}$, the *dilation* of $\mathcal{P}$ by $\mathcal{S}$ as $\mathcal{P} \oplus \mathcal{S} = \{p + s : p \in \mathcal{P}, s \in \mathcal{S} \text{ and } (p+s) \in \mathcal{D}\}$. In this sense, we say that two pixels p and q are $\mathcal{S}$-neighbors if $q \in (\{p\} \oplus \mathcal{S})$. In this paper, we only take into consideration symmetric SEs (i.e., if $s \in \mathcal{S}$, so does $-s$), so $q \in (\{p\} \oplus \mathcal{S})$ is equivalent to $p \in (\{q\} \oplus \mathcal{S})$. Given a SE $\mathcal{S}$, we define the set of neighboring pixels defined by $\mathcal{S}$, or simply *neighborhood*, as $\mathcal{A}(\mathcal{S}) = \{(p, q) : p \in \{q\} \oplus \mathcal{S}, p \in \mathcal{D}, q \in \mathcal{D}\}$. If the SE $\mathcal{S}$ is not relevant, we may simply denote a neighborhood by $\mathcal{A}$. In addition, considering a set $X \subset \mathcal{D}$ and a set of neighborhood $\mathcal{A}$, we say that two pixels $p, q \in X$ are

$\mathcal{A}$-connected if and only if there exists a sequence of pixels $(p_1, p_2, \ldots, p_m)$, with $p_j \in X$, such that all the following conditions hold: (a) $p_1 = p$; (b) $p_m = q$; (c) $\{p_j, p_{j+1}\} \in \mathcal{A}$ for all $1 \leq j < m$. *Connectedness* can be used to define *$\mathcal{A}$-connected components* ($\mathcal{A}$-CCs) or simply *connected components* (CCs). In this way, $\mathcal{A}$-CCs are defined as maximal sets of $\mathcal{A}$-connected pixels in X.

2.2 Component Trees

From an image f we define, for any $\lambda \in \mathbb{K}$, the set $X_\lambda(f) = \{p \in \mathcal{D} : f(p) \geq \lambda\}$ as the *upper level set* at value λ of the image f. Level sets are nested, i.e., $X_0(f) \supseteq X_1(f) \supseteq \ldots \supseteq X_{K-1}(f)$. By defining $CC(f, \mathcal{A}) = \{(\mathcal{C}, \lambda) : \mathcal{C}$ is a $\mathcal{A}$-CC of $X_\lambda(f), \lambda \in \mathbb{K}\}$, we can denote an order relation between these elements: given two elements $(\mathcal{C}, \alpha)$, $(\mathcal{C}', \beta) \in CC(f, \mathcal{A})$, we define that $(\mathcal{C}, \alpha) \sqsubset (\mathcal{C}', \beta) \Leftrightarrow \mathcal{C} \subseteq \mathcal{C}'$ and $\alpha > \beta$. This means these elements can be organized hierarchically in a tree structure according to this order relation. This tree is commonly referred as *component tree*. It is easy to note that, in a component tree, a single CC $\mathcal{C}$ can be represented by multiple nodes, if $\mathcal{C}$ exists in multiple level sets. In order to simplify the structure and save memory, an option is to store only one node per CC. It is easy to show that, in order to recover the complete tree, storing only the highest gray-level that generates each CC suffices. Using this property, we can create a simplified component tree without repeated CCs. This simplified structure is usually called *max-tree*. An example is shown in Fig. 1.

2.3 Algorithmic Background

There are many efficient ways of building max-trees [1]. One of the most efficient strategies consists in a generalization of the classic CC-labeling algorithm based on disjoint set structure (union-find) [4,12]. In this generalized structure, each CC has a representative pixel, and links between representative pixels give information about the inclusion relation of the CCs. In this way, each CC is composed of a representative and all other elements that point towards it. Disjoint sets can be efficiently stored using an array, which is called parent, so that parent[p] refers to the element that p is pointing to. Algorithm 1 shows the union-find algorithm.

Algorithm 1. Max-tree building based on disjoint sets.

```
1: procedure UNIONFIND($f$, $\mathcal{A}$, parent)
2:     for $\{p, q\} \in \mathcal{A}$ do
3:         $rP \leftarrow$ FIND(parent, $f$, $p$);
4:         $rQ \leftarrow$ FIND(parent, $f$, $q$);
5:         UNION(parent, $rP$, $rQ$);
6:     return parent;

1: procedure MAKESET(parent, $f$)
2:     for $p \in \mathcal{D}$ do
3:         parent[p] $\leftarrow \emptyset$;

1: procedure FIND(parent, $f$, $p$)
2:     if $p = \emptyset$ then
3:         return $p$;
4:     if $f(p) \neq f$(parent[p]) then
5:         return $p$;
6:     else
7:         parent[p] $\leftarrow$
8:                 FIND(parent, $f$, parent[p]);
9:         return parent[p];
```

In this algorithm, there are three basic operations: (1) MAKESET, which initializes the array with each pixel as a disjoint CC; (2) FIND, which obtains the representative of a given pixel p; and (3) UNION, which given two representatives p and q, makes the adjustments to merge CCs containing p to CCs containing q. The basic idea of the algorithm to build max-trees using disjoint sets is simply to connect all neighboring pixels by calling the UNION function when they do not belong to the same CC. Figure 1 shows a graphical example of union-find (rightmost column).

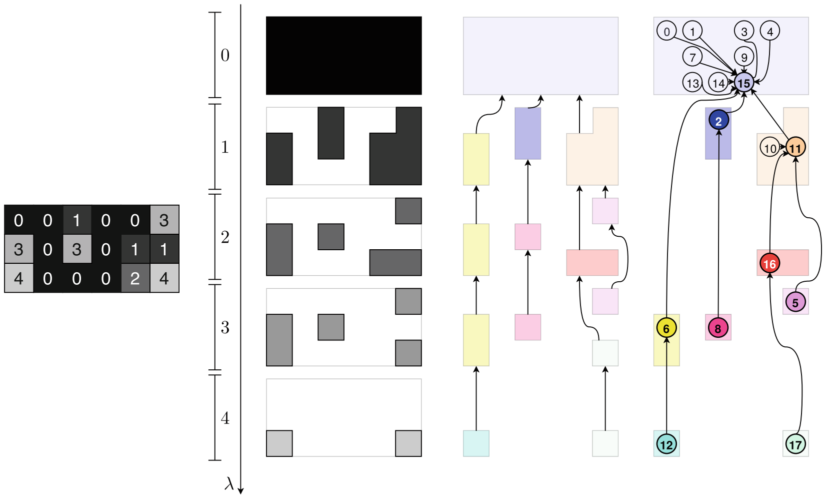


Fig. 1. From left to right: a gray-level image f; the upper level sets of f; the respective component tree with each CC as a node and arcs indicating the order relation; a graphical representation of the parent array that represents the max-tree. Each element p is represented by a circle and parent[p] is given by the arc leaving from p.

2.4 Component-Hypertrees

An extension of component trees can be obtained by considering another way of organizing the CCs of upper level sets in a hierarchical way using multiple neighborhoods. Let $\mathbb{A}$ be a sequence of neighborhood sets, i.e., $\mathbb{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$. If these neighborhoods are increasing (i.e., $\mathcal{A}_i \subset \mathcal{A}_{i+1}$ with $1 \leq i < n$), then for any $\mathcal{A}_i$-CC $\mathcal{C}$, there exists an $\mathcal{A}_{i+1}$-CC $\mathcal{C}'$ that contains $\mathcal{C}$, that is, $\mathcal{C} \subseteq \mathcal{C}'$. An easy way of obtaining increasing neighborhoods is to consider a sequence of SEs $\mathbb{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ such that $\mathcal{S}_i \subset \mathcal{S}_{i+1}, 1 \leq i < n$ (an example is provided in Fig. 2). Then, if we define $\mathcal{A}_i = \mathcal{A}(\mathcal{S}_i)$ and $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, $\mathbb{A}$ will be a sequence of increasing neighborhoods. By combining the CCs of upper level sets and increasing neighborhoods, we can build a hierarchical graph known as *component-hypertree*. More formally, given an image f and a sequence of increasing neighborhoods $\mathbb{A}$, we define the set of $\mathbb{A}$-CCs, denoted by $N(f, \mathbb{A})$, as

$$N(f, \mathbb{A}) = \{(\mathcal{C}, \lambda, i) : \mathcal{C} \in CC(f, \mathcal{A}_i), \lambda \in \mathbb{K}, i \in \mathbb{I}\}, \quad (1)$$

where $\mathbb{I} = \{1, 2, \ldots, n\}$. Considering this set, two distinct elements $(\mathcal{C}, \alpha, i)$ and $(\mathcal{C}', \beta, j)$ of $N(f, \mathbb{A})$ are said to be nested if and only if $\mathcal{C} \subseteq \mathcal{C}', \alpha \geq \beta$ and $i \leq j$ with $(\mathcal{C}, \alpha, i) \neq (\mathcal{C}', \beta, j)$. In this case, we write $(\mathcal{C}, \alpha, i) \sqsubset (\mathcal{C}', \beta, j)$. Given two elements $\mathcal{N}, \mathcal{N}' \in N(f, \mathbb{A})$ such that $\mathcal{N} \sqsubset \mathcal{N}'$, if there is no $\mathcal{N}'' \in N(f, \mathbb{A})$ satisfying $\mathcal{N} \sqsubset \mathcal{N}'' \sqsubset \mathcal{N}'$, then we write $\mathcal{N} \prec \mathcal{N}'$.

The *hypertree* of an image f using a sequence of neighborhoods $\mathbb{A}$, denoted by $HT(f, \mathbb{A})$, is simply the directed graph $(V(f, \mathbb{A}), E(f, \mathbb{A}))$ where its vertices $V(f, \mathbb{A})$ are the $\mathbb{A}$-CCs, i.e., $V(f, \mathbb{A}) = \{v \in N(f, \mathbb{A})\}$ and its arcs $E(f, \mathbb{A})$ is defined as $E(f, \mathbb{A}) = \{(\mathcal{N}, \mathcal{N}') \in N(f, \mathbb{A}) \times N(f, \mathbb{A}) : \mathcal{N} \prec \mathcal{N}'\}$.

Each arc $e = (\mathcal{N}, \mathcal{N}')$ is named by either *parent arc* or *composite arc*. On one hand, if e is such that $\mathcal{N}$ and $\mathcal{N}'$ are nodes representing CCs from consecutive level sets but with the same neighborhood, we say that e is a *parent arc*; in addition, we also say that $\mathcal{N}$ is a *child node* of $\mathcal{N}'$, or $\mathcal{N}'$ is a *parent node* of $\mathcal{N}$. On the other hand, if $\mathcal{N}$ and $\mathcal{N}'$ have the same gray-level but consecutive neighborhood indices, we say that e is a *composite arc*; in addition, we also say that $\mathcal{N}'$ is a *composite node* of $\mathcal{N}$, or $\mathcal{N}$ being a *partial node* of $\mathcal{N}'$. An example is provided at the left side of Fig. 3, where parent and composite arcs are respectively represented by black and blue colors.

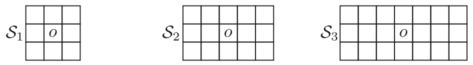


Fig. 2. A sequence $\mathbb{S} = (\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3)$ with 3 increasing SEs.

3 Minimal Component-Hypertree

As explained, component trees can be efficiently stored using max-trees. Similarly, we want to define a minimal structure for storing hypertrees. For that, it suffices to keep in memory the CCs along with their highest threshold and their lowest adjacency index. Thus, we define the compact node of a node $\mathcal{N} = (\mathcal{C}, \lambda, i)$, denoted by $cn((\mathcal{C}, \lambda, i))$, of a hypertree $HT(f, \mathbb{A})$ as

$$\begin{aligned} &cn((\mathcal{C}, \lambda, i)) = (\mathcal{C}', \beta, \ell) \in N(f, \mathbb{A}) \text{ such that } \mathcal{C}' = \mathcal{C} \text{ and} \\ &\forall (\mathcal{C}'', \alpha, j) \in N(f, \mathbb{A}) \text{ with } \mathcal{C}'' = \mathcal{C}, \beta \geq \alpha \text{ and } \ell \leq j. \end{aligned} \tag{2}$$

In the same way, given an arc $(\mathcal{N}, \mathcal{N}') \in E(f, \mathbb{A})$, we define its *compact arc* as

$$ca((\mathcal{N}, \mathcal{N}')) = (cn(\mathcal{N}), cn(\mathcal{N}')). \tag{3}$$

So, from the above definitions, we can derive the *compact representation* of a hypertree $HT(f, \mathbb{A})$ as the directed graph $(CN(f, \mathbb{A}), CE(f, \mathbb{A}))$ where

$$\begin{aligned} &CN(f, \mathbb{A}) = \{cn(\mathcal{N}) : \mathcal{N} \in N(f, \mathbb{A})\} \text{ and} \\ &CE(f, \mathbb{A}) = \{ca((\mathcal{N}, \mathcal{N}')) : (\mathcal{N}, \mathcal{N}') \in E(f, \mathbb{A})\} \end{aligned} \tag{4}$$

are, respectively, the sets of compact nodes and compact arcs. Figure 3 shows an example of a hypertree (left) and its compact representation (right).

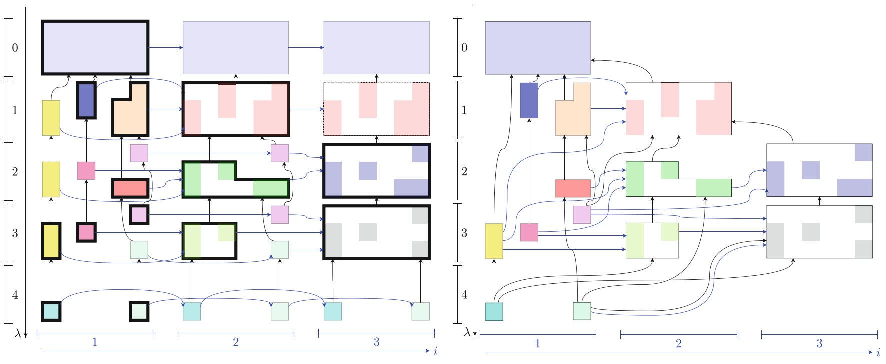


Fig. 3. Left: the complete hypertree of f from Fig. 1 using $\mathbb{A} = (\mathcal{A}(\mathcal{S}_1), \mathcal{A}(\mathcal{S}_2), \mathcal{A}(\mathcal{S}_3))$, where $\mathcal{S}_i, i = \{1, 2, 3\}$ is in Fig. 2. Each shape represents a node, and the colored squares represent their respective CCs. Parent arcs are colored black, while composite arcs are blue. Compact nodes are drawn with thick border. Right: the simplified hypertree with only compact arcs and compact nodes. (Color figure online)

From the compact representation, we still can eliminate some redundant arcs (see an example in Fig. 4). We say that a compact arc $e = (\mathcal{N}, \mathcal{N}') \in CE(f, \mathbb{A})$ is a *redundant arc* if and only if there exists a directed path $\pi = (\mathcal{N} = \mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_m = \mathcal{N}')$ in the compact hypertree such that

1. $\mathcal{N}_1 = \mathcal{N} = (\mathcal{C}, \alpha, i_1)$;
2. $\mathcal{N}_m = \mathcal{N}' = (\mathcal{C}', \beta, i_m)$;
3. For any k, with $1 \leq k < m$, $\mathcal{N}_k = (\mathcal{C}_k, \gamma_k, i_k)$, $i_k \leq i_{k+1}$.

We say that these arcs are redundant because, in terms of inclusion relation of CCs, they give the same information of the alternative path π. In addition, the above Condition (3) enforces that any path does not go "backward", i.e., we cannot have a path π that goes from a node with bigger neighborhood to a node with smaller neighborhood.

If a compact arc is not redundant, it is called a *minimal arc*. Let $MA(f, \mathbb{A})$ be the set of all minimal arcs of a compact hypertree built from an image f and using all neighborhoods in $\mathbb{A}$. Then, we call the *minimal component-hypertree* of f using $\mathbb{A}$, the directed graph $(CN(f, \mathbb{A}), MA(f, \mathbb{A}))$. Since all arcs that are not present in the minimal hypertree are redundant, we can easily see that this minimal representation preserves all the inclusion relation of CCs of the hypertree. An example of a minimal component-hypertree is shown in Fig. 4.

3.1 Data Structure

Since a hypertree is a DAG (which could be neither a tree nor a forest), only an array is not enough to store it. In this way, we explicitly allocate nodes and arcs in order to keep hypertrees in memory. Besides that, since nodes from trees with indices higher than 1 are composed of cluster of nodes from the first tree,

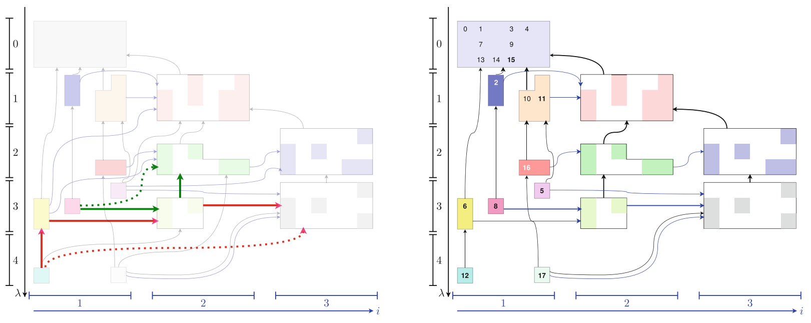


Fig. 4. Left: two examples of redundant arcs in the compact hypertree. For each color (red, green), the dotted arcs are redundant since they are equivalent to the paths with the same color. Right: the minimal component-hypertree i.e., the compact hypertree shown in the left without redundant arcs. Number inside nodes indicate the pixels stored in each node. (Color figure online)

only pixels of the first tree need to be stored. This strategy saves a considerable amount of memory, since each pixel is stored only once in the entire hypertree.

In terms of the data stored in each allocated node $\mathcal{N}$ of a hypertree, we keep the following information: (a) $rep(\mathcal{N})$ is its representative pixel; (b) $level(\mathcal{N})$ is the (highest) gray-level that generates its CC; (c) $index(\mathcal{N})$ is the (smallest) neighborhood index that generates its CC; (d) $pixels(\mathcal{N})$ is the set of pixels used to reconstruct its CC; (e) $par(\mathcal{N})$ is its set of parent nodes; (f) $child(\mathcal{N})$ is its set of children nodes; (g) $comp(\mathcal{N})$ is its set of composite nodes and (h) $part(\mathcal{N})$ is its set of partial nodes. An allocated node will be initialized as a triple $N = (rep(\mathcal{N}), level(\mathcal{N}), index(\mathcal{N}))$, with other fields empty.

3.2 Algorithm Template

The naive way of building hypertrees is to compute each max-tree from $\mathcal{A}_1$ to $\mathcal{A}_n$ by allocating its nodes without checking for repetition and then linking the corresponding nodes between two consecutive trees. In terms of memory usage, this is not efficient even if the repeated nodes are removed later, since it is needed to first allocate the whole hypertree in memory (which is a procedure that we want to keep away).

To avoid this problem, we build up the minimal hypertree by using a variant of the unordered version of the UNION procedure. The original algorithm was first presented in [13]. It has the property that it only changes a parent relation of a node if it really needs, i.e., a new neighborhood makes two originally disjoint CCs become connected. By keeping track of these changes, we can predict when new nodes need to be allocated. So our minimal hypertree construction strategy will follow the template of Algorithm 2.

In this sense, our template can be divided into two phases: one that updates the parent array (at Line 10) and other that updates the graph itself (at Lines

Algorithm 2. Minimal hypertree construction template. Underlined parameters are modified during the function call.

```
1: procedure BUILDHYPERTREE($f$, $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$)
2:   $nodes \leftarrow \emptyset$;
3:   for $p \in \mathcal{D}$ do
4:     parent[p] $\leftarrow \emptyset$;
5:     compactNode[p] $\leftarrow \emptyset$;
6:   for $1 \leq i \leq n$ do
7:     $updateNode \leftarrow \emptyset$;
8:     $updateArc \leftarrow \emptyset$;
9:     for $(p, q) \in \mathcal{A}_i$ do
10:      UNION($\underline{\text{parent}}$, $f$, $p$, $q$, $i$, $False$, $\underline{updateNode}$, $\underline{updateArc}$);
11:    ALLOCATENODES($\underline{nodes}$, $\underline{\text{compactNode}}$, $updateNode$, parent, $f$, $i$);
12:    UPDATENEWARCS($\underline{\text{compactNode}}$, $updateArc$, parent);
```

11 and 12). After Line 10, we have the parent array representing the max-tree of f using $\mathcal{A}_i$, and it is used to allocate new nodes and update inclusion arcs.

The time complexity of this algorithm highly depends on the number of neighbors in each step i (at Line 9). We do not focus on the choice of the sequence of neighborhoods in this paper, but the usage of richer connectivities, such as mask-based connectivities [6,8,13] and dilation-based connectivities [3,10], can considerably reduce the number of neighbors we have to process, especially because all elements of $\mathcal{A}_{i-1}$ also belong to $\mathcal{A}_i$.

3.3 Detection of New Nodes Needed to Be Allocated

A key property needed for efficient node allocation is to quickly detect the emergence of new nodes in the parent array for the next component tree. For that, we need to change the UNION (called at Line 10) to detect these new nodes.

Given a pair of neighboring pixels (p, q), the UNION procedure makes the necessary changes so that the parent array reflects the next updated component tree with p and q as neighbors. In a nutshell, these changes consist of merging all disjoint CCs containing either p or q. This means that these changes are limited to the branches linking p to r and q to r, where r is the first common ancestor of p and q in the parent array.

In particular, the CCs represented in the branch from p to r are all CCs containing p that do not contain q and, likewise, the CCs from q to r do not contain p. So, the changes we need to do in the UNION procedure is to mark elements of the array when a change in the parent array occurs. Let $\pi_p = (p = p_1, p_2, \ldots, p_j = r)$ and $\pi_q = (q = q_1, q_2, \ldots, q_k = r)$ be the paths in the parent array, respectively, from p to r and from q to r. If a pixel q' of π_q becomes the new parent of p' from π_p (since they become connected in the adjacency $\mathcal{A}_i$), then we have a change in the parent array so that p' has its parent changed to q' and all elements from q' to r will have at least a new pixel p' included, and will produce new CCs. Analogously, the same property holds if a pixel from π_p

becomes a parent of an element of π_q. Once a change happens, we mark all arcs and nodes until reaching the common ancestor. These marks are used later to allocate nodes and update the inclusion relation of new nodes (Lines 11 and 12).

An example is provided in Fig. 5. In the left, the original state of the union-find representation is shown. Suppose p_8 and p_{16} become neighbors (p_j refers to the pixel in the figure with label j). Notice that p_{15} is their first common ancestor and all changes are limited to the branches from p_8 to p_{15} and p_{16} to p_{15}. When parent of p_8 is updated to p_{16}, all elements from p_{16} up to p_{15} now have the new pixel p_8. Likewise, parent of pixel p_{11} is now pixel p_2, and this triggers the creation of a new node. This change makes p_{11} not be a representative anymore.

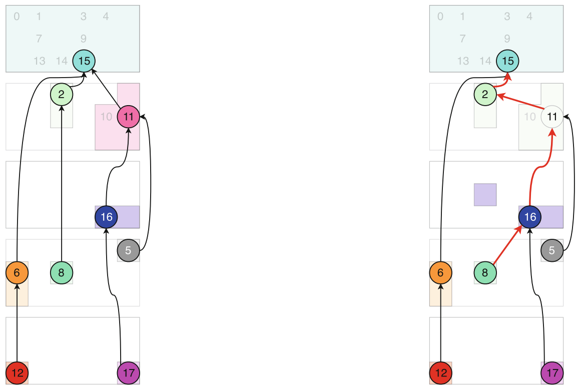


Fig. 5. Left: representation of the parent array from Fig. 1, with representatives highlighted. Right: updating the array when pixels p_8 and p_{16} are neighbors. In Algorithm 3, the red arcs will be added in the set *updateArc* and nodes p_{16}, p_{11} and p_2 will be added to the set *updateNode*, since they all have a new descendant (p_8). Node p_{15} is not added because it is the first common ancestor of p_8 and p_{16}.

In this way, we obtain a version of UNION procedure that detects when new nodes need to be created by marking changes in the parent array (see Algorithm 3), where marked elements are added to the set *updateNode*. Likewise, to create new arcs, whenever a node is created, the path leading that node to the common ancestor is added to the set *updateArc*.

3.4 Graph Update: Node Allocation and Arc Addition

Node allocation procedure is shown in Algorithm 4. For each marked pixel (i.e., the one that is in *updateNode*) which is representative, we allocate a node. We still need to deal with some arcs from different component trees, since it is possible to have arcs linking nodes with same representative. If the newly allocated node has one partial node (the node from the previous tree with the same representative), then we add an arc linking them. In addition, we update compactNode to always point to the lastly allocated node for each representative (i.e. the last and smallest compact node that contains the representative). Arc addition is

Algorithm 3. Modified union-find to mark changes in the parent array.

```
1: procedure UNION(parent, $f$, $p$, $q$, $i$, $changed$, $updateNode$, $updateArc$)
2:   if $f(p) < f(q)$ then                                    ▷ we consider $f(\emptyset) = -1$
3:     UNION(parent, $f$, $q$, $p$, $i$, $changed$, $updateNode$, $updateArc$);
4:   else
5:     if $p \neq q$ then
6:       $parP \leftarrow$ FIND(parent, $f$, parent[p]);
7:       if $changed$ and $i > 1$ then
8:         $updateNode \leftarrow updateNode \cup \{p\}$; $updateArc \leftarrow updateArc \cup \{p\}$;
9:       if $parP \neq q$ then
10:        if $f(parP) \geq f(q)$ then
11:          UNION(parent, $f$, $parP$, $q$, $i$, $changed$, $updateNode$, $updateArc$);
12:        else
13:          parent[p] $\leftarrow q$;
14:          if $i > 1$ then
15:            $updateArc \leftarrow updateArc \cup \{p\}$;
16:          UNION(parent, $f$, $q$, $parP$, $i$, True, $updateNode$, $updateArc$);
```

straightforward: for all marked arcs (the ones in *updateArc*), we add them (linking compact nodes) in the graph (see Algorithm 5). In both Algorithms 4 and 5, arc addition depends on the gray-levels and neighborhood indices of their nodes. If they have different gray-levels (resp., indices), a parent (resp., composite) arc is added. Note that these two conditions are not exclusive: it is possible to add both arcs, like the ones from the green node (with pixel p_{17}) to the gray node in Fig. 4 (right).

Algorithm 4. Allocation of new nodes from the set *updateNode*.

```
1: procedure ALLOCATENODES($nodes$, compactNode, $updateNode$, parent, $f$, $i$)
2:   for $p \in updateNode$ do
3:     $rP \leftarrow$ FIND(parent, $f$, $p$);
4:     if $p = rP$ then
5:       Allocate node $N' \leftarrow (rP, f(rP), i)$;
6:       if compactNode[rP] $\neq \emptyset$ then
7:         $N \leftarrow$ compactNode[rP];
8:         Add arc $(N, N')$;
9:       compactNode[rP] $\leftarrow N'$;
10:      $nodes \leftarrow nodes \cup \{N'\}$;
```

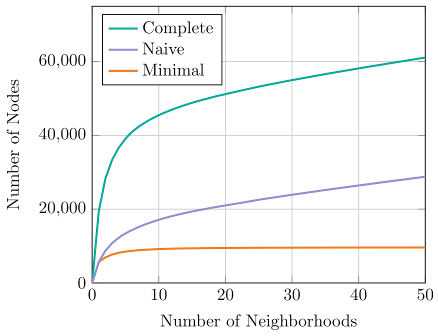
Algorithm 5. Updating arcs involving new nodes.

```
1: procedure UPDATENEWARCS(compactNode, $updateArc$, parent);
2:   for $p \in updateArc$ do
3:     Add arc (compactNode[p], compactNode[parent[p]]);
```

4 Experiments

In this section, we analyze how much memory is saved by using our minimal representation of component-hypertrees, compared to the complete representation and the naive strategy explained in Sect. 3.2. On one hand, in the complete representation, all CCs for all component trees from $\mathcal{A}_1$ to $\mathcal{A}_n$ are stored in memory. On the other hand, in the naive representation, max-trees (the compact representation for component trees) are used.

For our tests, we used images from ICDAR 2011 [2]. For each image of this set, we used a sequence of square neighborhoods $\mathbb{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, 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. 6 shows that the minimal representation can save a considerable amount of memory compared to the other ones. For example, in average, for 10 neighborhoods, we have a saving of about 50% compared to the naive implementation and 80% compared to the complete hypertree, both in terms of number of nodes and number of arcs.



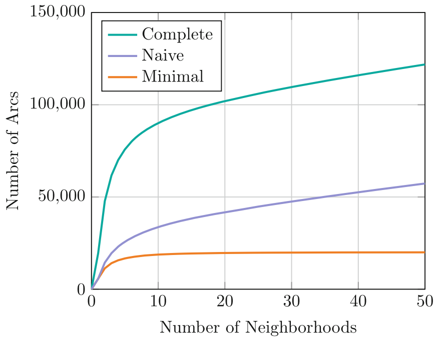


Fig. 6. Left: average number of nodes for each representation for up to 50 neighborhoods. Right: the same experiments but for the average number of arcs.

For more complex images from ICDAR 2017 [5], our method still saves between 50% and 80% of memory when compared to the naive approach and about 60% to 85% when compared to the complete representation (considering $n = 50$). In terms of time consumption, using the optimized approach given in [3] to update the array and our approach for node allocation resulted in total time ranging from 1 s (for images with 0.25 mega-pixels) to about a minute (for images with 8 mega-pixels).

5 Conclusion

In this paper, we presented algorithms and data structures behind the construction of minimal component-hypertrees that efficiently store them without redundancy and without loss of information about the inclusion relation of CCs.

Experiments show that our approach saves a considerable amount of memory compared to the complete representation and the strategy of independently building the max-trees for each neighborhood. As a future work, we plan to study how to efficiently compute attributes in these structures.

Acknowledgements. This study was financed in part by the CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (Finance Code 001); FAPESP - Fundação de Amparo a Pesquisa do Estado de São Paulo (Proc. 2018/15652-7); CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico (Proc. 428720/2018-8).

References

1. Carlinet, E., Géraud, T.: A comparative review of component tree computation algorithms. IEEE Trans. Image Process. **23**(9), 3885–3895 (2014)
2. Karatzas, D., Mestre, S.R., Mas, J., Nourbakhsh, F., Roy, P.P.: 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 (ICDAR), pp. 1485–1490. IEEE (2011)
3. Morimitsu, A., Alves, W.A.L., Hashimoto, R.F.: Incremental and efficient computation of families of component trees. In: Benediktsson, J.A., Chanussot, J., Najman, L., Talbot, H. (eds.) ISMM 2015. LNCS, vol. 9082, pp. 681–692. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18720-4_57
4. Najman, L., Couprie, M.: Building the component tree in quasi-linear time. IEEE Trans. Image Process. **15**(11), 3531–3539 (2006)
5. Nayef, N., et al.: ICDAR 2017 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), vol. 1, pp. 1454–1459. IEEE (2017)
6. Ouzounis, G.K., Wilkinson, M.H.: Mask-based second-generation connectivity and attribute filters. IEEE Trans. Pattern Anal. Mach. Intell. **29**(6), 990–1004 (2007)
7. Ouzounis, G.K., Wilkinson, M.H.: Hyperconnected attribute filters based on k-flat zones. IEEE Trans. Pattern Anal. Mach. Intell. **33**(2), 224–239 (2011)
8. Passat, N., Naegel, B.: Component-hypertrees for image segmentation. In: Soille, P., Pesaresi, M., Ouzounis, G.K. (eds.) ISMM 2011. LNCS, vol. 6671, pp. 284–295. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21569-8_25
9. Salembier, P., Oliveras, A., Garrido, L.: Antiextensive connected operators for image and sequence processing. IEEE Trans. Image Process. **7**(4), 555–570 (1998)
10. Serra, J.: Connectivity on complete lattices. J. Math. Imaging Vis. **9**(3), 231–251 (1998)
11. Silva, D.J., Alves, W.A., Morimitsu, A., Hashimoto, R.F.: Efficient incremental computation of attributes based on locally countable patterns in component trees. In: 2016 IEEE International Conference on Image Processing (ICIP), pp. 3738–3742. IEEE (2016)
12. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM (JACM) **22**(2), 215–225 (1975)
13. Wilkinson, M.H., Gao, H., Hesselink, W.H., Jonker, J.E., Meijster, A.: Concurrent computation of attribute filters on shared memory parallel machines. IEEE Trans. Pattern Anal. Mach. Intell. **30**(10), 1800–1813 (2008)