HAL
open science

Join, Select, and Insert: Efficient Out-of-core Algorithms for Hierarchical Segmentation Trees

Josselin Lefèvre, Jean Cousty, Benjamin Perret, Harold Phelippeau

To cite this version:

Josselin Lefèvre, Jean Cousty, Benjamin Perret, Harold Phelippeau. Join, Select, and Insert: Efficient Out-of-core Algorithms for Hierarchical Segmentation Trees. IAPR International Conference on Discrete Geometry and Mathematical Morphology (DGMM), Oct 2022, Strasbourg, France. pp.274-286, 10.1007/978-3-031-19897-7_22 . hal-03794772

HAL Id: hal-03794772

https://hal.science/hal-03794772v1

Submitted on 3 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Join, select, and insert: efficient out-of-core algorithms for hierarchical segmentation trees

Josselin Lefèvre[1,2], Jean Cousty[1], Benjamin Perret[1], Harold Phelippeau[2]

[1] LIGM, Univ Gustave Eiffel, CNRS, ESIEE Paris, F-77454 Marne-la-Vallée, France
[2] Thermo Fisher Scientific, Bordeaux, France

Abstract. Binary Partition Hierarchies (BPH) and minimum spanning trees are fundamental data structures involved in hierarchical analysis such as quasi-flat zones or watershed. However, classical BPH construction algorithms require to have the whole data in memory, which prevent the processing of large images that cannot fit entirely in the main memory of the computer. To cope with this problem, an algebraic framework leading to a high level calculus was introduced allowing an out-of-core computation of BPHs. This calculus relies on three operations: *select*, *join*, and *insert*. In this article, we introduce three efficient algorithms to perform these operations providing pseudo-code and complexity analysis.

1 Introduction

Hierarchies of partitions are versatile representations that have proven useful in many image analysis and processing problems. In this context, binary partition hierarchies [14] (BPH) built from altitude ordering and associated minimum spanning trees are key structures for several (hierarchical) segmentation methods: in particular it has been shown [2,11] that such hierarchies can be used to efficiently compute quasi-flat zone (also referred as α-trees) hierarchies [10,2] and watershed hierarchies [9,2]. Efficient algorithms for building BPHs on standard size images are well established, but, with the constant improvement of acquisition systems comes a dramatic increase in image resolutions, which can reach several terabytes in size. In such case, it becomes impossible to put a single image in the main memory of a standard workstation and classical algorithms for BPHs stop working. This creates the need for scalable algorithms to construct BPHs in an out-of-core manner to handle images that cannot fit in memory.

In [4,6,8], the authors investigate distributed memory algorithms to compute min and max trees for terabytes images. In [5], computation of minimum spanning trees of streaming images is considered. A parallel algorithm for the computation of quasi-flat zones hierarchies has been proposed in [7]. Finally, the authors of [1] recently proposed massively parallel algorithms for the computation of max-trees on GPUs. All these work rely on a common idea which is to work independently on small pieces of the space, "join" the information found on adjacent pieces, and "insert" this joint information into other pieces.

In a previous work [3], the authors specifically addressed the problem of computing a BPH under the out-of-core constraint, *i.e.*, when the objective is

to minimize the amount of memory required by the algorithms. To do so, they introduced an algebraic framework formalizing the distribution of a hierarchy over a partition of the space together with three algebraic operations acting on BPHs: *select*, *join*, and *insert*. They showed that, when a causal partition of the space is considered, it is possible to compute the distribution of a BPH using these three operations by browsing the different regions of the partition only twice (once in a forward pass and once in a backward pass) and by requiring to have only the information about two adjacent regions in the main memory at any step of the algorithm. However, no efficient algorithm has been proposed for the three operations *select*, *join*, and *insert*.

In this work, we propose efficient implementations for these operations. The proposed algorithms rely on a particular data structure to represent *local* hierarchies which is designed to efficiently search and browse the nodes of the hierarchy and to store only the necessary and sufficient information required locally to compute the distribution of the BPH. We give algorithms, with their pseudo-code, for the three operations whose time complexity is either linear or linearithmic. In order to ease the presentation, we consider the particular case of 2d images, modelled as 4-adjacency graphs, but the method can be easily extended to any regular graph. The implementation of the method in C++ and Python based on the hierarchical graph processing library Higra [12] is available online `https://github.com/PerretB/Higra-distributed`.

This article is organized as follows. Section 2 gives the definition of BPH. Section 3 recalls the notion of the distribution of a hierarchy and the calculus method that can be used to compute such distribution over a causal partition of the space. Section 4 explains the proposed data structures. Section 5 presents the algorithms for the three operations *select*, *join*, and *insert*. Finally, Section 6 concludes the work and gives some perspectives.

2 Binary partition hierarchy by altitude ordering

In this section, we first remind the definitions of hierarchy of partitions. Then we define the binary partition hierarchy by altitude ordering using the edge-addition operator [3] and we recall the bijection existing between the regions of this hierarchy and the edges of a minimum spanning tree of the graph.

Let V be a set. A *partition of* V is a set of pairwise disjoint subsets of V. Any element of a partition is called a *region* of this partition. The *ground* of a partition $\mathbf{P}$, denoted by $gr(\mathbf{P})$, is the union of the regions of $\mathbf{P}$. A partition whose ground is V is called a *complete partition of* V. Let $\mathbf{P}$ and $\mathbf{Q}$ be two partitions of V. We say that $\mathbf{Q}$ is a *refinement of* $\mathbf{P}$ if any region of $\mathbf{Q}$ is included in a region of $\mathbf{P}$. A *hierarchy on* V is a sequence $(\mathbf{P}_0, \ldots, \mathbf{P}_\ell)$ of partitions of V such that, for any λ in $\{0, \ldots, \ell - 1\}$, the partition $\mathbf{P}_\lambda$ is a refinement of $\mathbf{P}_{\lambda+1}$. Let $\mathcal{H} = (\mathbf{P}_0, \ldots, \mathbf{P}_\ell)$ be a hierarchy. The integer ℓ is called the *depth of* $\mathcal{H}$ and, for any λ in $\{0, \ldots, \ell\}$, the partition $\mathbf{P}_\lambda$ is called the λ*-scale of* $\mathcal{H}$. In the following, if λ is an integer in $\{0, \ldots, \ell\}$, we denote by $\mathcal{H}[\lambda]$ the λ-scale of $\mathcal{H}$. For any λ in $\{0, \ldots, \ell\}$, any region of the λ-scale of $\mathcal{H}$ is also called a *region of* $\mathcal{H}$.

The hierarchy $\mathcal{H}$ is *complete* if $\mathcal{H}[0] = \{\{x\} \mid x \in V\}$ and if $\mathcal{H}[\ell] = \{V\}$. We denote by $\mathcal{H}_\ell(V)$ the set of all hierarchies on V of depth ℓ, by $\mathcal{P}(V)$ the set of all partitions on V, and by $2^{|V|}$ the set of all subsets of V.

In the following, the symbol ℓ stands for any strictly positive integer.

We define a graph as a pair $G = (V, E)$ where V is a finite set and E is composed of unordered pairs of distinct elements in V. Each element of V is called a vertex of G, and each element of E is called an edge of G. The Binary Partition Hierarchy (BPH) by altitude ordering relies on a total order on E, denoted by $\prec$. Let k in $\{1, \ldots, \ell\}$, we denote by $u_k^\prec$ the k-th element of E for the order $\prec$. Let u be an edge in E, the *rank of u for $\prec$*, denoted by $r^\prec(u)$, is the unique integer k such that $u = u_k^\prec$. We then define the *update of a hierarchy $\mathcal{H}$ with respect to an edge $\{x, y\}$*, denoted by $\mathcal{H} \oplus \{x, y\}$: with k the rank of $\{x, y\}$, $\mathcal{H} \oplus \{x, y\}[\lambda]$ remains unchanged for any λ in$\{0, k-1\}$ while, for any λ in $\{k, \ldots, \ell\}$, we have $(\mathcal{H} \oplus \{x, y\})[\lambda] = \mathcal{H}[\lambda] \setminus \{R_x, R_y\} \cup \{R_x \cup R_y\}$ where R_x (resp. R_y) denotes the region of $\mathcal{H}[\lambda]$ containing x (resp. y). Let $E' \subseteq E$ and let $\mathcal{H}$ be a hierarchy. We set $\mathcal{H} \boxplus E' = \mathcal{H} \oplus u_1 \oplus \ldots \oplus u_{|E'|}$ where $E' = \{u_1, \ldots, u_{|E'|}\}$. The binary operation $\boxplus$ is called the *edge-addition*. Thanks to this operation, we can define formally the BPH for $\prec$. Let X be a set, we denote by $\perp_X$ the hierarchy defined by $\perp_X [\lambda] = \{\{x\} \mid x \in X\}$, for any λ in $\{0, \ldots \ell\}$. The BPH for $\prec$, denoted by $\mathcal{B}^\prec$ is the hierarchy $\perp_X \boxplus E$.

Let $\mathcal{B}^\prec$ be a binary partition by altitude ordering, $\mathcal{R}$ be a region of $\mathcal{B}^\prec$ and $\mathcal{R}^\star$ be the set of non-leaf regions of $\mathcal{B}^\prec$. The *rank* of $\mathcal{R}$, denoted by $r(\mathcal{R})$, is the lowest integer λ such that $\mathcal{R}$ is a region of $\mathcal{B}^\prec[\lambda]$. We consider the map μ from $\mathcal{R}^\star$ in E such that, for any non-leaf region $\mathcal{R}$ of $\mathcal{B}^\prec$, we have $\mu^\prec(\mathcal{R}) = u_{r(\mathcal{R})}^\prec$. We say that $\mu^\prec(\mathcal{R})$ is the *building edge* of $\mathcal{R}$. Building edges of the binary partitions hierarchy defines a minimum spanning tree of an edge-weighted graph. In Figure 1, $\mathcal{Y}$ is the BPH built on the 4-adjacency graph B. Non-leaf nodes of $\mathcal{Y}$ correspond to the edges of the minimum spanning tree of B (dashed edges).

3 Distributed hierarchies of partitions on causal partition

In this section, we recall the definition of the distribution of a BPH on a sliced graph and the principle of its calculus in an out-of-core manner. Intuitively, distributing a hierarchy consists in splitting it into a set of smaller trees such that: 1) each smaller tree corresponds to a *selection* of a sub part of whole tree that intersects a slice of the graph and 2) the initial hierarchy can be reconstructed by "gluing" those smaller trees.

Let V be a set. The operation *sel* is the map from $2^{|V|} \times \mathcal{P}(V)$ to $\mathcal{P}(V)$ which associates to any subset X of V and to any partition $\mathbf{P}$ of V the subset $\mathrm{sel}(X, \mathbf{P})$ of $\mathbf{P}$ which contains every region of $\mathbf{P}$ that contains an element of X. The operation *select* is the map from $2^{|V|} \times \mathcal{H}_\ell(V)$ in $\mathcal{H}_\ell(V)$ which associates to any subset X of V and to any hierarchy $\mathcal{H}$ on V the hierarchy $\mathrm{select}\,(X, \mathcal{H}) = (\mathrm{sel}(X, \mathcal{H}[0]), \ldots, \mathrm{sel}(X, \mathcal{H}[\ell]))$.

We are then able to define the distribution of a hierarchy thanks to *select*. Let V a set, let $\mathbf{P}$ be *a complete partition on* V and let $\mathcal{H}$ be a hierarchy on V.

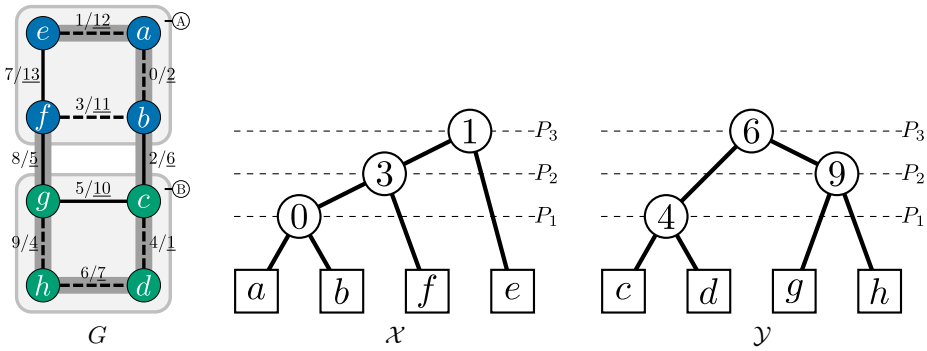

Fig. 1. G a 4-adjacency graph divided into two slices A and B (respectively blue and green). Each edge of G is associated with a pair (*index, weight*). The two slices are separated by their common neighborhood *i.e.* edges 8 and 2. Hierarchy $\mathcal{X}$ (respectively $\mathcal{Y}$) is the BPH built on A (respectively B). Indices associated with non-leaf nodes of the BPHs correspond to the indices of their corresponding building edges represented by dashed edges in G. We can note that MSTs built on slices (dashed edges) are not sub-trees of the complete MST (shown as shadow). In consequence, edge 3 is part of the hierarchy $\mathcal{X}$ when it should not be.

The distribution of $\mathcal{H}$ over $\mathbf{P}$ is the set $\{\text{select}\,(R, \mathcal{H}) \mid R \in \mathbf{P}\}$ and for any region R of $\mathbf{P}$, select $(R, \mathcal{H})$ is called a *local hierarchy (of $\mathcal{H}$ on R)*.

The calculus introduced in [3] aims to compute the distribution of a BPH over a partition of the space. In this article, we consider the special case of a 4 adjacency graph representing a 2d image that can be divided into slices, and we are interested in computing a distribution of the BPH over those slices. It should be noted that this is not a limiting factor, and the method can easily be adapted to any regular grid graph.

Let h and w be two integers representing the height and the width of an image. In the following, the set V is the Cartesian product of $\{0, \cdots, h-1\} \times \{0, \cdots, w-1\}$. Thus, any element x of V is a pair $x = (x_i, x_j)$ such that x_i and x_j are the coordinates of x. In the 4-adjacency grid, the set of all edges E is equal to $\{\{x, y\} \in V \mid |x_i - y_i| + |x_j - y_j| \leq 1\}$. Let k be a positive integer, the causal partition of V is the sequence $(S_0, \ldots, S_k)$ such that for any t in $\{0, \cdots, k\}$, $S_t = \{(i, j) \in V \mid t \times \frac{w}{k} \leq i < (t+1) \times \frac{w}{k}\}$. Each element of this partition is called a *slice*. The set of vertices at the interface between two neighbor slices A and B and belonging to A is noted $\gamma_B^\bullet(A)$. The major advantage of considering this partition over a regular graph is that each subset of V or E can be computed on the fly efficiently from a computational and memory point of view.

Given this causal partition, Algorithm 1 allows computing the local hierarchies of the BPH of the complete graph on each slice. This algorithm can be divided in two parts: causal and anti-causal traversal of the slices. Each of these parts relies on the same idea. First, start with the causal traversal. Given a causal partition of V into $k+1$ slices, for any i in $\{1, \cdots, k\}$ compute the BPH on S_i

Algorithm 1: Out-of-core binary partition hierarchy [3].

Data: A graph (V, E), a total order $\prec$ on E, and a causal partition $(S_0, \ldots, S_k)$ of V

Result: $\{\mathcal{B}_0^{\downarrow}, \ldots, \mathcal{B}_k^{\downarrow}\}$: the distribution of the BPH $\mathcal{B}_V^{\prec}$ over $\{S_0, \ldots, S_k\}$.

1 $\mathcal{B}_0^{\uparrow} := \mathcal{B}_{S_0}^{\prec}$ `// call PlayingWithKruskal algorithm`

2 **foreach** i ***from*** 1 ***to*** k **do** `// Causal traversal of the slices`

3 Call PlayingWithKruskal algorithm to compute $\mathcal{B}_{S_i}^{\prec}$

4 $\mathcal{M}_i^{\uparrow} := \text{join}\left(\text{select}\left(\gamma_{S_i}^{\bullet}(S_{i-1}), \mathcal{B}_{i-1}^{\uparrow}\right), \text{select}\left(\gamma_{S_{i-1}}^{\bullet}(S_i), \mathcal{B}_{S_i}^{\prec}\right)\right)$

5 $\mathcal{B}_i^{\uparrow} := \text{insert}(\text{select}\left(\gamma_{S_{i-1}}^{\bullet}(S_i), \mathcal{M}_i^{\uparrow}\right), \mathcal{B}_{S_i}^{\prec})$

6 $\mathcal{B}_k^{\downarrow} := \mathcal{B}_k^{\uparrow}$; $\mathcal{M}_k^{\downarrow} := \mathcal{M}_k^{\uparrow}$

7 **foreach** i ***from*** $k-1$ ***to*** 0 **do** `// Anticausal traversal of the slices`

8 $\mathcal{B}_i^{\downarrow} := \text{insert}(\text{select}\left(\gamma_{S_{i+1}}^{\bullet}(S_i), \mathcal{M}_{i+1}^{\downarrow}\right), \mathcal{B}_i^{\uparrow})$

9 **if** $i > 0$ **then** $\mathcal{M}_i^{\downarrow} := \text{insert}(\text{select}\left(\gamma_{S_{i-1}}^{\bullet}(S_i), \mathcal{B}_i^{\downarrow}\right), \mathcal{M}_i^{\uparrow})$

with a call to the algorithm presented in [11] hereafter called PlayingWithKruskal (line 3). Then, select the part of this hierarchy containing the vertices adjacent to the previous slice and join it with the part of the hierarchy associated to the previous slice containing the vertices adjacent to the current slice, leading to the "merged" hierarchy denoted by $\mathcal{M}_i^{\uparrow}$ (line 4). The merged hierarchy is then inserted in the BPH which gives $\mathcal{B}_i^{\uparrow}$ (line 5). The hierarchies $\mathcal{B}_i^{\uparrow}$ associated to slice i misses the information located in slices of higher indices, and consequently only the last local hierarchy $\mathcal{B}_k^{\uparrow}$ is correct *i.e.* $\mathcal{B}_k^{\uparrow} = \text{select}\,(S_k, \mathcal{H})$. In order to compute the valid distribution, and after having spread information in the causal direction, information must be back propagated in the reverse anti-causal direction so that each local hierarchy is enriched with the global context (lines 7 to 9).

4 Data structures

In this section, we present the data structures used in the algorithms defined in the following sections. These data structures are designed to contain only the necessary and sufficient information so that we never need to have all the data in the main memory at once. The data structure representing a local hierarchy assumes that the nodes of the hierarchy are indexed in a particular order and relies on three "attributes": 1) a mapping of the indices from the local context (a given slice) to the global one (the whole graph) noted $\mathcal{H}$`.map`, 2) the parent array denoted by $\mathcal{H}$`.par` encoding the parent relation between the tree nodes, and 3) an array $\mathcal{H}$`.weights` giving, for each non-leaf-node of the tree, the weight of its corresponding building edge.

More precisely, given a binary partition hierarchy $\mathcal{H}$ with n regions, every integer between 0 and $n-1$ is associated to a unique region of $\mathcal{H}$. Moreover, this

indexing of the regions of $\mathcal{H}$ follows a topological order such that: 1) any leaf region is indexed before any non-leaf region; 2) two leaf regions $\{x\}$ and $\{y\}$ are sorted with respect to an arbitrary order on the element V, called the *raster scan order* of V. Thus $\{x\}$ has an index lower than $\{y\}$ if x is before y with respect to the raster scan order; and 3) two non-leaf regions are sorted according to their rank, *i.e.*, the order of their building edges for $\prec$. This order can be seen as an extension of the order $\prec$ on E to the set $V \cup E$ that enables 1) to efficiently browse the nodes of a hierarchy according to their scale of appearance in the hierarchy and 2) to efficiently match regions of V with the leaves of the hierarchy. By abuse of notation, this extended order is also denoted by $\prec$ in the following.

To keep track of the global context, a link between the indices in the local tree and the global indices in the whole graph is stored in the form of an array `map` which associates: 1) to the index i of any leaf region R, the vertex x of the graph G such that $R = \{x\}$, *i.e.* `map[i]=x`; and 2) to the index i of any non-leaf region R, its building edge, *i.e.* `map[i]=`$\mu^{\prec}(R)$.

The parent relation of the hierarchy is stored thanks to an array `par` such that `par[i]=j` if the region of index j is the parent of the region of index i.

The binary partition hierarchy is built for a particular ordering $\prec$ of the edges of G. In practice, this ordering is induced by weights computed over the edges of G. To this end, we store an array `weights` of $|\mathcal{R}^{\star}(\mathcal{H})|$, *i.e.* the number of non-leaf regions, elements such that, for every region R in $\mathcal{R}^{\star}(\mathcal{H})$ of index i, `weights[i]` is the weight of the building edge $\mu^{\prec}(R)$ of region R. The edges can then be compared according to the following total order induced by the weights: we set $u \prec v$ if the weight of u is less than the one of v or if u and v have equal weights but u comes before v with respect to the raster scan order.

5 Algorithms

Select. In this part, we give an algorithm to compute the result of the *select* operation. This operation consists in "selecting" the part of a given hierarchy intersecting a subset of the space. In Algorithm 1, *select* takes as input a set of vertices located at the "border" of a slice and a hierarchy in order to obtain a smaller "border hierarchy".

Select algorithm proceeds in 3 steps:

1. *Lines 3-7.* Mark any leaf-node of $\mathcal{H}$ that corresponds to an element of X, *i.e.* any leaf-region $\{x\}$ with $x \in X$;
2. *Lines 8-9.* Traverse the hierarchy from leaves to root and mark any node that is a parent of a marked node;
3. *Lines 11-17.* Build the hierarchy $\mathcal{S}$ whose nodes are only marked nodes of $\mathcal{H}$.

In Algorithm 2 we assume that X is sorted and that $X \subset gr(\mathcal{H})$, which is always the case in Algorithm 1. For each element $X[i]$ of X, we search for the index j of a leaf of $\mathcal{H}$ mapped to $X[i]$, *i.e.* such that $\mathcal{H}$.`map[j]` $= X[i]$. To this end, it is necessary to make a traversal of the leaves of $\mathcal{H}$. As mentioned before, the leaves correspond to the first indices by construction. The first step can then

Algorithm 2: SELECT

```
Data: H: a hierarchy, X: set of selected nodes st. X ⊂ gr(H)
Result: S: the hierarchy select (X, H)
Initialize an array mark to false for every region of H
i := 0; j := 0            // i iterates over X and j over the leaves of H
while i < |X| and j < |H.leaves| do
    if X[i] = H.map[j] then
        mark[j] := true
        i := i + 1
    j := j + 1
foreach n from 0 to |H| − 1 do
    if mark[n] = true then mark[H.par[n]] := true
n_S := 0
foreach n from 0 to |H| − 1 do
    if mark[n] = true then
        S.par[n_S] := H.par[n]
        S.map[n_S] := H.map[n]
        if n ∈ R*(H) then
            S.weight[n_S − |X|] := H.weight[n_S − |H.leaves|]
    n_S := n_S + 1
return S
```

be performed in linear time with respect to the number of leaf-regions of $\mathcal{H}$. The second step consist in traversing the whole hierarchy from leaves to root in order to mark every region of $\mathcal{H}$ which belongs to select $(X, \mathcal{H})$ *i.e.* regions parent of a marked one. The complexity of this step is therefore linear with respect to the number of regions of $\mathcal{H}$. Finally, the last step boils down to extracting the hierarchy select $(X, \mathcal{H})$ from the marked nodes. For this a new hierarchy is created by traversing $\mathcal{H}$ again. As the traversal is done by increasing order of index, the properties relating to the weights of the building edges and order of appearance of regions are preserved. The complexity of this last step is linear with respect to the number of regions of $\mathcal{H}$. Thus, Algorithm 2 has a linear $O(n)$ complexity, where n is the number of regions of $\mathcal{H}$.

Join. Formally the *join of* $\mathcal{X}$ *and* $\mathcal{Y}$, denoted by $join(\mathcal{X}, \mathcal{Y})$, is the hierarchy defined by $join(\mathcal{X}, \mathcal{Y}) = (\mathcal{X} \sqcup \mathcal{Y}) \boxplus F$, where F is the common neighborhood of the grounds of $\mathcal{X}$ and of $\mathcal{Y}$, and $\sqcup$ denotes the supremum on hierarchies (see [13]). Intuitively, this operation merges two hierarchies according to their common neighborhood, that is the set of edges linking their grounds. In [7], the authors proposed an algorithm that can be used to successively add edges of the common neighborhood. Intuitively, to add an edge, the hierarchy is updated while climbing the branches associated with the edge extremities. The worst-case complexity is then linear with respect to the size of the hierarchy for adding a single edge. Thus the overall complexity of such join procedure would be $O(k \times n)$

where n is the size of the hierarchies and k is the number of edges in the common neighborhood. In this section, we drop the multiplicative dependency in the size of the neighborhood at the cost of introducing a sorting of F and we present an algorithm whose complexity is quasi-linear with respect to the size n of the hierarchies and linearithmic with respect to the number k of edges in F.

Algorithm 3: JOIN

Data: $\mathcal{X}$*and* $\mathcal{Y}$: two hierarchies, F common neighborhood of $gr(\mathcal{X})$ and $gr(\mathcal{Y})$.
Result: A collection $Q_D = \text{join}\,(\mathcal{X},\ \mathcal{Y})$

foreach *node* n_i *of* $\mathcal{X}$ **do** Q_D.MakeSet(i)
foreach *node* n_i *of* $\mathcal{Y}$ **do** Q_D.MakeSet($i + |\mathcal{X}.\text{leaves}|$)
aDescendent($\mathcal{X}, 0$)
aDescendent($\mathcal{Y}, |\mathcal{X}.leaves|$)
F :=sort(F)
$i_1 := |\mathcal{X}.leaves|;\ i_2 := |\mathcal{Y}.leaves|;\ i_3 := 0$
while $i_1 < |\mathcal{X}|$ ***or*** $i_2 < |\mathcal{Y}|$ ***or*** $i_3 < |F|$ **do**
  if $F[i_3] \prec \mathcal{X}.map[i_1]$ ***and*** $F[i_3] \prec \mathcal{Y}.map[i_2]$ **then**
    $(x, y) := F[i_3];\ m := F[i_3];\ w := weight(F[i_3]);\ i_3 \mathrel{+}= 1$
  else if $\mathcal{X}.map[i_1] \prec \mathcal{Y}.map[i_2]$ **then**
    $(x, y) := \mathcal{X}.\text{desc}[i_1];\ m := \mathcal{X}.\text{map}[i_1];\ w := \mathcal{X}.\text{weight}[i_1];\ i_1 \mathrel{+}= 1$
  else
    $(x, y) := \mathcal{Y}.\text{desc}[i_2];\ m := \mathcal{Y}.\text{map}[i_2];\ w := \mathcal{Y}.\text{weight}[i_2];\ i_2 \mathrel{+}= 1$
  $c_x := Q_D.\text{FindCanonical}(x);\ c_y := Q_D.\text{FindCanonical}(y)$
  if $c_x! = c_y$ **then**
    $n := Q_D.\text{Union}(c_x, c_y);\ Q_D.\text{map}[n] := m$
    $Q_D.\text{weight}[n - (|\mathcal{X}.leaves| + |\mathcal{Y}.leaves|)] := w$

A detailed presentation of the proposed algorithm is given in Algorithm 3 which calls auxiliary functions presented in Algorithm 4. Intuitively, in order to compute the join of two hierarchies $\mathcal{X}$ and $\mathcal{Y}$, Algorithm 3 consists in "emulating" PlayingWithKruskal algorithm on the graph obtained from (i) the edges associated to the non-leaf nodes of $\mathcal{X}$ and of $\mathcal{Y}$ and (ii) the edges in the common neighborhood F of $\mathcal{X}$ and $\mathcal{Y}$. Therefore, all these edges are considered in increasing order with respect to $\prec$ and, for each edge, it is decided if this edge must be considered or not in the creation process of the join hierarchy. The decision is made based on the potential creation of a cycle if this edge were added during the minimum spanning tree creation process. We can thus see on the Figure 2 that the node 3 has been added to $\mathcal{X}$ by construction but that it is then discarded during the construction of the joined hierarchy. Potential-cycles creation is efficiently checked with Tarjan Union-Find data structures as in Kruskal's algorithm. A main observation can be made to highlight the difference between the situation encountered in the contexts of join algorithm and PlayingWithKruskal algorithms: in the context of join, some edges, which

are associated to the nodes of the hierarchies $\mathcal{X}$ and $\mathcal{Y}$, are made of vertices that do not belong to the underlying space (*i.e.*, the common neighborhood of the slices supporting the grounds of $\mathcal{X}$ and $\mathcal{Y}$). When such edge is found, the standard algorithm can be shortcut leading to a modified version of the PlayingWithKruskal auxiliary functions presented in Algorithm 4. Compared to original functions, the only change is the insertion of the if test at line 9. This test detects the edges for which a shortcut must occur based on an attribute called `desc`. This attribute is pre-computed for every node of $\mathcal{X}$ and of $\mathcal{Y}$ by the auxiliary function `aDescendent`. Overall, the following steps are performed in Algorithm 3:

- *Lines 1-2.* Initialize the Union-find data structures;
- *Lines 3-4.* Compute the attribute `desc` for both $\mathcal{X}$ and $\mathcal{Y}$;
- *Lines 5 to 13.* Browse the edges in increasing order. Observe that it implies sorting the edges in the common neighborhood F of $\mathcal{X}$ and $\mathcal{Y}$ in increasing order for $\prec$ (non-leaf-nodes of $\mathcal{X}$ and $\mathcal{Y}$ are already sorted by construction);
- *Lines 15-17* Apply PlayingWithKruskal steps, calling the modified version of the auxiliary functions.

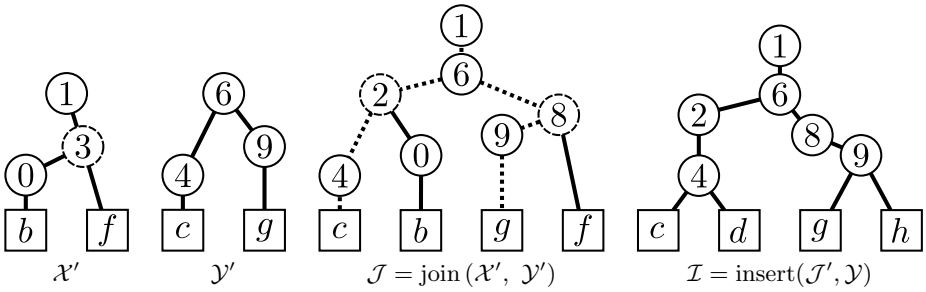

Fig. 2. The hierarchy $\mathcal{J}$ is build by computing the join over $\mathcal{X}' = \text{select}\,(\{c, h\}, \mathcal{X})$ and $\mathcal{Y}' = \text{select}\,(\{d, i\}, \mathcal{Y})$ (border trees computed from BPHs of Figure 1). We can see that the node 3 of $\mathcal{X}'$ not longer appear in the joint tree in the favor of nodes corresponding to the common neighborhood of the grounds of $\mathcal{X}$ and $\mathcal{Y}$ *i.e.* nodes 2 and 8. That is to say, that by taking into account the topological order on the edges of the MSTs associated with the border trees and the common neighborhood, 3 does not belongs to the BPH. $\mathcal{I}$ is then build by inserting the hierarchy $\mathcal{J}' = \text{select}\,(\{c, g\}, \mathcal{J})$ into the hierarchy $\mathcal{Y}$. Given the Definition 16 of 1, $\mathcal{I}$ is a "correct" local hierarchy for the tile B.

The first step complexity is linear with respect to the number of elements of $gr(\mathcal{X}) \cup gr(\mathcal{Y})$. The second step uses the auxiliary function `aDescendent` to compute attributes `desc` for both $\mathcal{X}$ and $\mathcal{Y}$. It should be noted that this last function takes a parameter *shift* which allows to index the leaves of the second hierarchy after those of the first. For each node of the two hierarchies,

Algorithm 4: AUXILIARY FUNCTIONS FOR *join* ALGORITHM

```
   // The functions called hereafter on Q_T and Q_BT are those described in [11]
1  Procedure Q_D.MakeSet(q)
2  |  Q_D.Root[q] := q; Q_BT.MakeSet(q); Q_T.MakeSet(q)
3
4  Function Q_D.FindCanonical(q)
5  |  return Q_T.FindCanonical(q)
6
7  Function Q_D.Union(c_x, c_y)
8  |  t_u := Q_D.Root[c_x]; Q_BT.par[t_u] := Q_BT.size
9  |  if c_y = −1 then Q_D.Root[c_x] := Q_BT.size
10 |  else
11 |  |  t_v := Q_D.Root[c_y]; Q_BT.par[t_v] := Q_D.size
12 |  |  c := Q_T.Union(c_x, c_y); Q_D.Root[c] := Q_BT.size
13 |  end
14 |  Q_BT.MakeSet(Q_BT.size)
15 |  return Q_D.size − 1
16
17 Function aDescendent(H: a hierarchy, s: shift)
   |  // set the attribute H.desc: an array that maps to each node n of H two leaves
   |     that are descendants of the two children of n.
18 |  foreach node n of H do H.desc[n]:= (−1, −1)
19 |  foreach leaf node n of H do H.desc[n].first:= n + s
20 |  foreach non-root non-leaf node n of H in increasing order for ≺ do
21 |  |  p := H.par[n]
22 |  |  if H.desc[p].first= −1 then H.desc[p].first:= H.desc[n].first
23 |  |  else H.desc[p].second:= H.desc[n].first
24 |  end
```

an attribute is computed during a leaves to root traversal which gives a linear complexity with respect to the number of regions of each hierarchy. Third step requires to sort the edges of F with respect to $\prec$ before browsing the edges which implies a complexity of $O(k \times log(k) + |\mathcal{X}| + |\mathcal{Y}|)$ with k the number of edges in F. The fourth step is equivalent to PlayingWithKruskal algorithm in terms of complexity. Then, its complexity is $O(m \times \alpha(n))$ where m is sum of the number of edges in F and the number of non-leaf nodes of $\mathcal{X}$ and $\mathcal{Y}$, where n is the number of leaf nodes in $\mathcal{X}$ and $\mathcal{Y}$ and where $\alpha()$ is the inverse Ackermann function which grows sub-logarithmically.

Insert. In this part, we present an algorithm to compute the hierarchy $\mathcal{Z} = \text{insert}(\mathcal{X}, \mathcal{Y})$. We assume that $\mathcal{X}$ is *insertable* in $\mathcal{Y}$ *i.e.* for any λ in $\{0, \ldots, \ell\}$, for any region Y of $\mathcal{Y}[\lambda]$, Y is either included in a region of $\mathcal{X}[\lambda]$ or is included in $V \setminus gr(\mathcal{X}[\lambda])$. This assumption holds true at each call to *insert* in Algorithm 1. The *insertion of* $\mathcal{X}$ *into* $\mathcal{Y}$ is the hierarchy $\mathcal{Z}$, such that, for any λ in $\{0, \ldots, \ell\}$, $\mathcal{Z}[\lambda] = \mathcal{X}[\lambda] \cup \{R \in \mathcal{Y}[\lambda] \mid R \cap gr(\mathcal{X}[\lambda]) = \emptyset\}$. Algorithm 5, presented hereafter, computes the insertion $\mathcal{Z}$ of $\mathcal{X}$ into $\mathcal{Y}$. From a high level point of view, it proceeds in two main steps:

Algorithm 5: INSERT

Data: $\mathcal{X}$ and $\mathcal{Y}$: two hierarchies such that $\mathcal{X}$ insertable in $\mathcal{Y}$.
Result: $\mathcal{Z}$: the hierarchy insert$(\mathcal{X}, \mathcal{Y})$

1 $x := 0;\ y := 0;\ z := 0;$ `// indices for the nodes/regions of` $\mathcal{X}$`,` $\mathcal{Y}$`, and` $\mathcal{Z}$
2 Initialize an array InZ of $|\mathcal{Y}|$ Booleans to true (resp. to false) for every leaf (resp. non-leaf) region of $\mathcal{Y}$
3 **while** $x < |\mathcal{X}|$ ***or*** $y < |\mathcal{Y}|$ **do**
4 **if** $x < |\mathcal{X}|$ ***and*** $y < |\mathcal{Y}|$ ***and*** $\mathcal{X}.map[x] = \mathcal{Y}.map[y]$ **then**
 `// Duplicate region` (x, y) `found in` $\mathcal{X}$ `and` $\mathcal{Y}$`, keep (and renumber) it in` $\mathcal{Z}$
5 $\mathrm{C}_{\mathcal{X}\to\mathcal{Z}}[x] := z;\ \mathrm{C}_{\mathcal{Y}\to\mathcal{Z}}[y] := z;\ \mathrm{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}[z] := (x, y)$
6 $x \mathrel{+}= 1;\ y \mathrel{+}= 1;\ z \mathrel{+}= 1$
7 **else if** $\mathcal{Y}.map[y] \prec \mathcal{X}.map[x]$ **then**
8 **if** $InZ[y] = true$ **then** `// Keep (en renumber) region` y `in` $\mathcal{Z}$
9 $InZ[\mathcal{Y}.\mathrm{par}[y]] := true$
10 $\mathrm{C}_{\mathcal{Y}\to\mathcal{Z}}[y] := z;\ \mathrm{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}[z] := (-1, y)$
11 $y \mathrel{+}= 1;\ z \mathrel{+}= 1$
12 **else** $y \mathrel{+}= 1$ `// Discard region` y `from` $\mathcal{Z}$
13 **else** `// Keep (and renumber) region` x `in` $\mathcal{Z}$
14 $\mathrm{C}_{\mathcal{X}\to\mathcal{Z}}[x] := z;\ \mathrm{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}[z] := (x, -1)$
15 $x \mathrel{+}= 1;\ z \mathrel{+}= 1$
16 $\mathcal{Z} :=$ initialize a tree structure with $n_{\mathcal{Z}} = z$ nodes
17 **foreach** z ***from*** *0* ***to*** $n_{\mathcal{Z}}$ **do**
18 $(x, y) := \mathrm{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}[r]$
19 **if** $x \neq -1$ **then**
20 $\mathcal{Z}.\mathrm{map}[z] := \mathcal{X}.\mathrm{map}[x]$
21 **if** $[x] = \mathcal{X}.root$ **then** $\mathcal{Z}.\mathrm{par}[z] := x$
22 **else** $\mathcal{Z}.\mathrm{par}[z] := \mathrm{C}_{\mathcal{X}\to\mathcal{Z}}[\mathcal{X}.\mathrm{par}[x]]$
23 **if** $z \geq |\mathcal{X}.leaves|$ **then**
 $\mathcal{Z}.\mathrm{weight}[z - |\mathcal{X}.leaves|] := \mathcal{X}.\mathrm{weight}[x - |\mathcal{X}.leaves|]$
24 **else**
25 $\mathcal{Z}.\mathrm{map}[z] := \mathcal{Y}.\mathrm{map}[y]$
26 **if** $y = \mathcal{Y}.root$ **then** $\mathcal{Z}.\mathrm{par}[z] := y$
27 **else** $\mathcal{Z}.\mathrm{par}[z] := \mathrm{C}_{\mathcal{Y}\to\mathcal{Z}}[\mathcal{Y}.\mathrm{par}[y]]$
28 **if** $z \geq |\mathcal{Y}.leaves|$ **then**
 $\mathcal{Z}.\mathrm{weight}[z - |\mathcal{Y}.leaves|] := \mathcal{Y}.\mathrm{weight}[x - |\mathcal{Y}.leaves|]$

1. *Lines 3-15.* Identify and renumber the regions of $\mathcal{X}$ and $\mathcal{Y}$ that belong to $\mathcal{Z}$ and store the correspondences between the new number of the regions in $\mathcal{Z}$ and the indices of the initial regions in $\mathcal{X}$ and $\mathcal{Y}$. It can be observed that this step is necessary since a region of $\mathcal{Z}$ can be duplicated in both $\mathcal{X}$ and $\mathcal{Y}$ and that some regions of $\mathcal{Y}$ are discarded from $\mathcal{Z}$. In order to perform this step, the regions of $\mathcal{X}$ and $\mathcal{Y}$ are simultaneously browsed in increasing order

for $\prec$. The correspondences between the regions of the hierarchies are stored in three arrays: $\mathrm{C}_{\mathcal{X}\to\mathcal{Z}}$, $\mathrm{C}_{\mathcal{Y}\to\mathcal{Z}}$, and $\mathrm{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}$;
2. *Lines 17-28.* Build the parenthood relation (`par`) of the hierarchy $\mathcal{Z}$ using the parenthood relation of the hierarchies $\mathcal{X}$ and $\mathcal{Y}$ and the correspondences between the regions of the hierarchies. At the same time, we also build the attributes `map` and `weight` associated to $\mathcal{Z}$.

During the first step, each region of the two hierarchies $\mathcal{X}$ and $\mathcal{Y}$ is considered once and processed with a limited number of constant-time instructions. Thus, the overall time complexity of Lines 3-15 is linear with respect to the number of nodes of $\mathcal{X}$ and $\mathcal{Y}$. The worst-case complexity of the second step is also linear with respect to the number of nodes of $\mathcal{X}$ and $\mathcal{Y}$ since $\mathcal{Z}$ contains at most all regions of each of hierarchy. Thus, the overall complexity of Algorithm 5 is $O(|\mathcal{X}| + |\mathcal{Y}|)$.

6 Conclusion

In this article, we proposed efficient and easily implementable algorithms for the three algebraic operations on hierarchies *select*, *join*, and *insert*. These algorithms rely on a particular data structure to represent local hierarchies in order to achieve linear or linearithmic time complexity while limiting the amount of information required in the main memory. Thanks to these contributions it is now possible to efficiently implement the calculus scheme proposed in [3] for the out-of-core computation of BPHs. In future works, we plan to study the time and memory consumption of the proposed algorithms in practice and to develop efficient algorithms to process the distribution of a BPH in order to obtain a completely out-of-core pipeline for seeded watershed segmentation.

Acknowledgements This work was supported by the French ANR grant ANR-20-CE23-0019.

References

1. Carlinet, E., Blin, N., Lemaitre, F., Lacassagne, L., Geraud, T.: Max-tree computation on GPUs. IEEE TPDS (2022)
2. Cousty, J., Najman, L., Perret, B.: Constructive links between some morphological hierarchies on edge-weighted graphs. In: ISMM. pp. 86–97 (2013)
3. Cousty, J., Perret, B., Phelippeau, H., Carneiro, S., Kamlay, P., Buzer, L.: An algebraic framework for out-of-core hierarchical segmentation algorithms. In: DGMM. pp. 378–390 (2021)
4. Gazagnes, S., Wilkinson, M.H.F.: Distributed connected component filtering and analysis in 2d and 3d tera-scale data sets. IEEE Transactions on Image Processing **30**, 3664–3675 (2021)
5. Gigli, L., Velasco-Forero, S., Marcotegui, B.: On minimum spanning tree streaming for hierarchical segmentation. PRL **138**, 155–162 (2020)

6. Götz, M., Cavallaro, G., Geraud, T., Book, M., Riedel, M.: Parallel computation of component trees on distributed memory machines. TPDS **29**(11), 2582–2598 (2018)
7. Havel, J., Merciol, F., Lefèvre, S.: Efficient tree construction for multiscale image representation and processing. JRTIP **16**(4), 1129–1146 (2019)
8. Kazemier, J.J., Ouzounis, G.K., Wilkinson, M.H.: Connected morphological attribute filters on distributed memory parallel machines. In: ISMM. pp. 357–368 (2017)
9. Meyer, F.: The dynamics of minima and contours. In: ISMM. pp. 329–336 (1996)
10. Meyer, F., Maragos, P.: Morphological scale-space representation with levelings. In: International Conference on Scale-Space Theories in Computer Vision. pp. 187–198 (1999)
11. Najman, L., Cousty, J., Perret, B.: Playing with Kruskal: algorithms for morphological trees in edge-weighted graphs. In: ISMM. pp. 135–146 (2013)
12. Perret, B., Chierchia, G., Cousty, J., Guimarães, S.J.F., Kenmochi, Y., Najman, L.: Higra: Hierarchical graph analysis. SoftwareX **10**, 100335 (2019)
13. Ronse, C.: Partial partitions, partial connections and connective segmentation. JMIV **32**(2), 97–125 (2008)
14. Salembier, P., Garrido, L.: Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. TIP **9**(4), 561–576 (2000)