Parallel Max-Tree Construction in Polylogarithmic Expected Time

Paul D. Teeninga (s2077477)
Supervisors: M. H. F. Wilkinson and J. A. Pérez

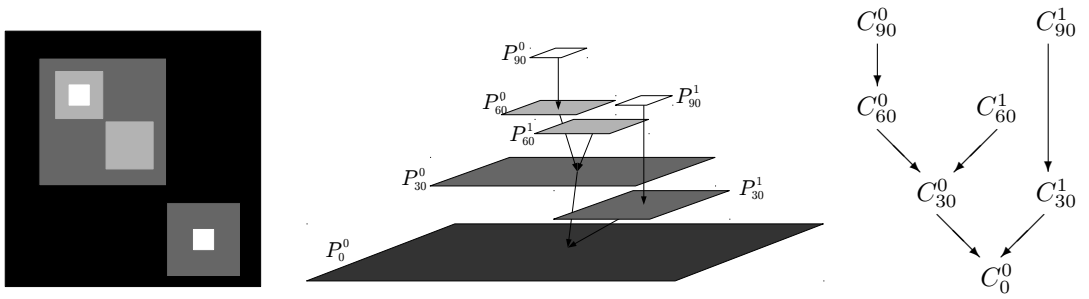
June 24, 2024

Abstract

A max-tree is an image analysis tool and represents a hierarchy of connected components in the threshold sets of a (typically) grayscale image. We introduce a computationally efficient parallel algorithm for max-tree construction of images in $\mathcal{O}(n\log(n)/p + \log^2(n)\log(p))$ expected time, for any input with length n, where n is the number of pixels and p is the number of processors. The algorithm iteratively bisects the graph representation of the image by value. A significant advantage is that the running time does not depend on bit depth, assuming constant-time comparisons. Current parallel algorithms have a linear lower bound on running time when no assumptions are made about bit depth. Results show up to 27.4× speedup (64 cores) and 44 megapixel/s throughput on worst case images with random floating-point numbers. The implementation is 3.9× to 9.5× faster than another parallel max-tree algorithm for high bit depth images.

1 Introduction

Image analysis is increasingly important in science and daily life. In our context, examples include galaxy and star detection in sky surveys [Teeninga et al., 2015][Haigh et al., 2021] (including nesting, for example a galaxy as parent object of a star), remote sensing to detect rubble [Ouzounis et al., 2011] and ships [Salembier et al., 2018], medical images to detect blood clots [Westenberg et al., 2007] and blood vessels [Xu et al., 2015], and diatom classification [Urbach et al., 2007]. With larger images and the availability of multicore processors and graphics processing units (GPUs), there is a need for efficient parallel algorithms [Merigot and Petrosino, 2008]. Many methods in image analysis are trivial to parallelize by dividing an image into tiles and processing each tile independently [Mighell, 2010]. However, connected morphological filters are not easy to parallelize [Salembier and Wilkinson, 2009], because of nontrivial dependencies between tiles. Connected morphological filters are image analysis tools based on lattice theory [Serra, 1982][Serra, 1988]. *Flat zones* in images are connected pixels with a constant value, and



(a) (b) (c)

Figure 1: Grayscale input image (a) and corresponding max-trees (b and c). P^k_h and C^k_h are pixel sets and both represent connected components k at threshold h, and contain respectively all component pixels (peak components) and only component pixels with values equal to the threshold. From [Gazagnes and Wilkinson, 2021].

a *connected filter* only merges flat zones. A connected filter is anti-extensive if image values can only decrease (or stay the same). Max-trees are used to simplify a family of anti-extensive connected filters, and we will explain the max-tree structure next.

In (discrete) images, pixels are directly connected to neighboring pixels, typically to up to 4 or 8 neighbors in 2 dimensions. To reconnect disconnected objects (e.g. caused by noise or damage), in second-generation connectivity [Ouzounis and Wilkinson, 2007a][Ouzounis and Wilkinson, 2007b], a max-tree is constructed from a mask image, while values of the original image are used in attribute computation (dual-input max-tree). Connected maximal subsets of pixels (no more can be added) form (connected) components, and an image typically has a single component. A threshold set of an image contains exactly the pixels with a value at or above some threshold. Threshold sets can contain multiple components, and each component is contained in a super component at a lesser threshold. A max-tree (figure 1) is the result of organizing components as nodes in a graph, and, for each node except the root, adding a directed edge to the smallest super component. The link with connected filters is that each node represents one or more flat zones with the same value, and contracting edges (deleting nodes) merges flat zones. Components have defined attributes [Breen and Jones, 1996], which are functions from the set of components to some attribute space. A simple attribute example is the *area*, which is the number of pixels in a component. Figure 2 shows an area opening, where objects with a small area are removed, and contour information of the galaxy is preserved.

With advances in technology, larger images (giga and terapixel) with more bits per pixel are produced. Examples include Synthetic Aperture Radar (SAR) images in remote sensing and Low Frequency Array (LOFAR) images in astronomy, where Fast Fourier Transforms are applied to raw data, producing floating-point images. Floating-point values typically use 32 or 64 bits. Stacked images produced with charge-coupled devices (CCDs) in astronomy also commonly use 32-bit floating-point or integer values. To construct max-trees of these images in a reasonable amount of time, algorithms are required to be scalable. When no assumption is made about the number of distinct values in an image, time complexities of current parallel algorithms have a linear lower bound, which we will discuss in more detail in the next section. We have developed a parallel algorithm that improves the average time complexity to polylogaritmic time, resulting in better scalability when more processors are added. To give a brief overview of the algorithm applied to images: starting with the set of all pixels, sets are iteratively bisected (or cut) by value, parent relations between nodes in different sets are known at the end of the iteration, and edges are modified to maintain paths inside components.

The rest of the paper is organized as follows. First, applications are listed in more detail and an overview is given of current algorithms. Second, in the theory section we prove that our algorithm is correct and meets complexity expectations. Third, an implementation is given to show that the algorithm is feasible to implement in practice. Fourth, results of our implementation are shown, including a comparison to other sequential and parallel max-tree construction algorithms. Fifth, future work is listed to discuss any shortcomings and possible improvements. Finally, we conclude the paper.

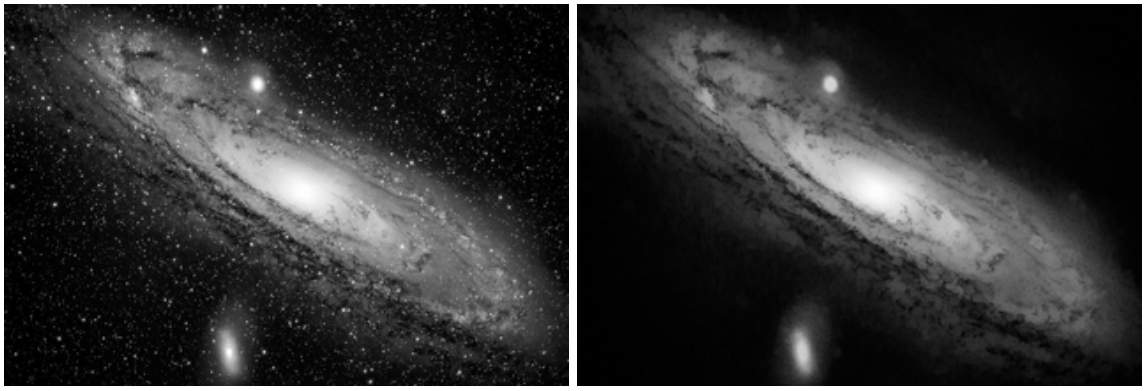
2 Applications and current algorithms

In this section we give an overview of max-tree applications and list relevant sequential and parallel max-trees algorithms, used as part of the software implementation or in comparisons.

2.1 Applications

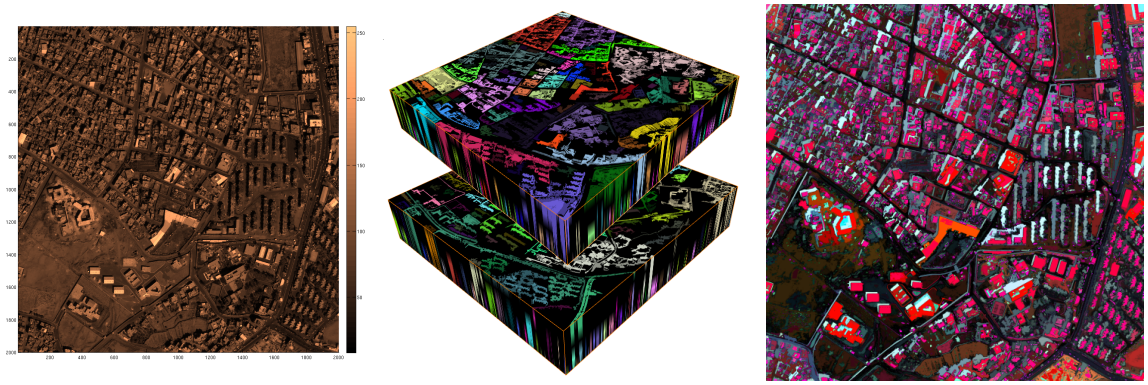
A max-tree representation of a grayscale image is a convenient structure to represent the hierarchy of nested components at all thresholds. Anti-extensive connected operators [Salembier et al., 1998] applied to a max-tree remove (nonroot) nodes if they fail a criterion. An *area opening* on a max-tree is an example of an anti-extensive operation, where (nonroot) components are removed if the area attribute (number of pixels) is less than some threshold λ. Pixels in removed components take the value of the first parent component. Figure 2 shows an example. Small objects have been removed and galaxy contours have been preserved. If an attribute is not increasing (for ancestor components), there are other choices for pixel values as described in [Salembier et al., 1998]. The dual operation of area opening is *area closing*. The operation is equivalent to area opening the negated image, and again negating the result. In contrast to area openings, pixel values are nondecreasing.

In remote sensing (geography), a *differential area profile* (DAP)[Ouzounis and Soille, 2010] of an image is a per pixel vector pair of value differences for respectively area openings and closings, given a vector of increasing scales (area minima) as criteria. Formally, let $\gamma_\lambda^{\text{area}}(x)$ map to the value of pixel x after an area



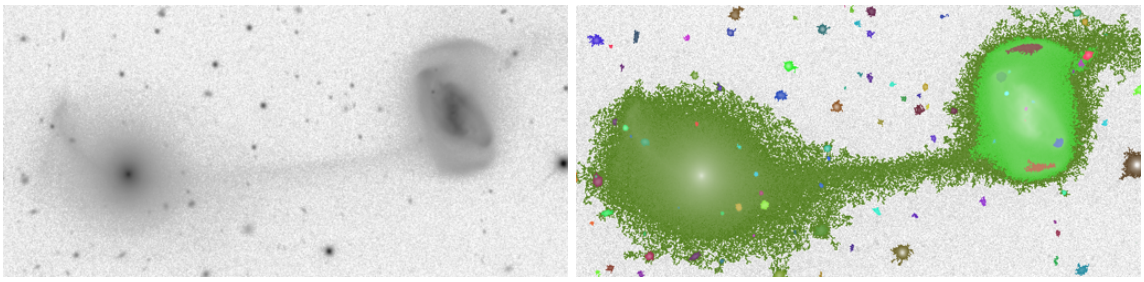
(a) (b)

Figure 2: Andromeda galaxy (a) and area opening (b). Photo by Torben Hansen, from https://www.flickr.com/photos/torbenh/6105409913.



(a) (b) (c)

Figure 3: CSL example. Original image (a), differential profiles for area openings above area closings, colored by 3D connected components (b), and CSL triplets represented as colors (c). Images from [Wilkinson et al., 2016].



(a) (b)

Figure 4: Merging galaxies. Original image (left) and hierarchical structure of segmented objects (right).

opening, where λ is the minimum area. Let $\varphi_\lambda^{\text{area}}(x)$ be the result after an area closing. Let the vector of increasing scales be $\boldsymbol{\lambda}$. The length of $\boldsymbol{\lambda}$ is k and $\lambda_0 = 0$. The differential profile for area openings is vector $\Delta^{\Pi,\gamma_\lambda^{\text{area}},\boldsymbol{\lambda}}(x)$, defined per element as negated finite difference

$$\Delta_i^{\Pi,\gamma_\lambda^{\text{area}},\boldsymbol{\lambda}}(x) = -(\gamma_{\lambda_{i+1}}^{\text{area}}(x) - \gamma_{\lambda_i}^{\text{area}}(x)), \text{for } 0 \leq i < k-1.$$

The values are nonnegative. For area closings the definition is analogous and the values are non-positive. The DAP of a pixel is defined as the concentation of the two profiles. The *CSL model* [Pesaresi and Kanellopoulos, 1999][Wilkinson et al., 2016] is a per pixel feature triplet derived from the DAP. Element C is the smallest scale that has the largest absolute value difference of both area openings and closings. Element S is that value difference. If both area opening and closing at scale C have the same absolute difference, the value of S is set to 0. The pixel is implicitly labeled as part of a convex, concave or flat structure for respectively a positive, negative or 0 value. Element L is the filtered pixel value at scale C. Figure 3 shows an example. Since the number of pixels on a boundary are increasing at larger scales, and an image has some noise, the value difference becomes smaller at larger scales, on average. This suggests that the CSL triplet gives some information about a steep edge in the neighborhood of a pixel. The model has been applied to produce a global human settlement layer [Pesaresi et al., 2013].

In the context of max-trees, a pattern spectrum is any histogram-like computed result using connected operators with different scales applied to an image (or node weighted graph). A set of connected operators with ordered scales is called a granulometry. In [Urbach et al., 2007], to classify diatoms, the pattern spectrum is a 2D histogram of elongation and area pairs, for each component. Histogram density for each component additionally depends on the height difference with the parent component and the area. The elongation attribute of a component is defined as the moment of inertia devided by the area squared. To reduce feature vector size, the histogram is transformed using a limited number of moments. Compared to other methods, advantages are scale and rotation invariance, and less susceptibility to noise.

To counter sparseness of higher dimensional attribute vectors, [Gan et al.,] applies self-organizing maps (SOMs), an unsupervised machine learning method to cluster data using a neural network. Each neuron is an attribute vector and cluster prototype. The method has been applied to fluorodeoxyglucose positron emission tomography (FDG-PET) scans for detecting lung tumors. After training, some neurons are more sensitive to tumors, i.e. the distance is closer to tumor attribute vectors.

In [Teeninga et al., 2015], statistical tests are used based on the power attribute (sum of squared values) to segment objects in sky surveys. By deleting nodes in the max-tree that are likely to be noise, and contracting nodes that have only one incoming edge (same object), the result is a nested structure of detected objects. We refer to the method as MTObjects. Figure 4 shows an example. MTobjects had the highest scores in a comparison of source-extraction tools [Haigh et al., 2021], and is one of the only methods capable of finding nested sources.

2.2 Current algorithms

We explain how a max-tree is represented in a computer, and give an overview of current sequential and parallel construction algorithms for node weighted graphs. Rigorous definitions are given in the theory section. See [Carlinet and Géraud, 2013] for an extensive comparison of algorithms.

So far, construction algorithms have been designed for images, and we talk about current algorithms in the context of image graphs. To represent an image as graph: pixels are nodes, values are node weights

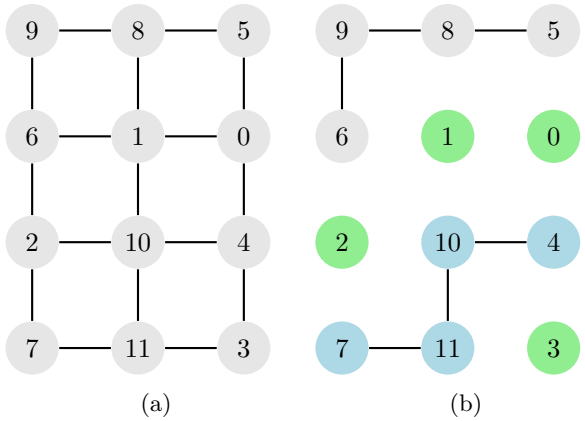


Figure 5: Injective undirected graph with node weights shown as labels (a) and a connected component C at threshold $t = w(\text{root}(C)) = 4$ in light blue (b). Parent candidates of root(C) are light green.

and connections between pixels (neighbor relations) are edges. When thresholding a graph G, all nodes with a weight less than the threshold are removed in the resulting graph. The largest connected subsets of nodes in a thresholded graph G' are (connected) components. Given any component C of G', when decreasing the threshold applied to G, the first larger component containing C is the parent of C. A max-tree of G contains components as nodes and parent relations as directed edges (figure 1).

Towards encoding a max-tree as a data structure, every component C is represented by a node (of C) with minimal weight. Figure 5 shows an example, where root(C) is the node with minimal weight. Every node is given a distinct index (a natural number) less than the total number of nodes. For any node in C with equal weight to the representative node of C, a parent reference is added either directly to the representative node, or to another node with equal weight assuming it does not introduce a cycle. Every representing node has a parent reference to another node in the parent component with lesser weight. Nodes representing root components (at minimal treshold) have a special reference to indicate a root component. One option is to refer to itself. Every node in C has a parent path to the representative node, and every ancestor component is in the root path. This allows the max-tree to be encoded as only an array of parent references, which are stored as (unsigned) integers. To reconstruct an image, some trivial bijective mapping between pixel positions and node indices is required.

2.2.1 Sequential algorithms

Current max-tree construction algorithms can be divided into two groups: flooding and set merging. The original algorithm by Salembier is given in [Salembier et al., 1998], which recursively floods components using a hierarchical queue. A modification that uses a priority queue is introduced in [Wilkinson, 2011], which performs better on images with high dynamic range. Recent optimizations from [Teeninga and Wilkinson, 2020] includes using a trie priority queue, shown in algorithm 1. Another approach to max-tree construction is iteratively merging components in nonincreasing order by weight, with a union-find data structure [Tarjan, 1975]. A derivation from the algorithm in [Berger et al., 2007] is shown in algorithm 2. When applied to images with a fixed number of dimensions, both construction with a priority queue and construction with union-find run in $\mathcal{O}(n \log n)$ time, where n is the number of pixels. This is optimal because a max-tree can be used to sort cn values from a totally ordered set, for some constant $c \leq 1$, and sorting is in $\Theta(n \log n)$ time (assuming constant time comparisons).

2.2.2 Parallel algorithms

Current parallel algorithms divide an image into tiles, construct tile max-trees sequentially (in parallel), and iteratively merges tile pairs on boundaries by merging root paths [Wilkinson et al., 2008] [Kazemier et al., 2017][Gazagnes and Wilkinson, 2021][Blin et al., 2022]. The path to the image root (component), through parents, is called a root path. Two root paths are merged in a similar way to merging sorted linked lists, and non-increasing ordering by weight is maintained. The referenced algorithms contain various optimizations to reduce the amount work. While these algorithms scale well with

Algorithm 1: Sequential max-tree construction with a priority queue

input : Image graph $G = (V, E, w)$
output: Max-tree of G

```
1  forall nodes x in V do
2  |  visited[x] ← false
3  end
4  q ← empty (trie) priority queue
5  x ← arbitrary node in G
6  visited[x] ← true
7  while true do
8  |  forall unvisited neighbors y of x do
9  |  |  visited[y] ← true
10 |  |  if w(y) < w(x) then
11 |  |  |  q.add(y)
12 |  |  else
13 |  |  |  q.add(x)
14 |  |  |  x ← y
15 |  |  |  continue line 7
16 |  |  end
17 |  end
18 |  if q is empty then
19 |  |  break
20 |  end
21 |  parent[x] ← q.top()
22 |  x ← q.top()
23 |  remove the top element of q
24 end
25 parent[x] ← x
26 return parent array
```

Algorithm 2: Sequential max-tree construction with union-find

input : Image graph $G = (V, E, w)$
output: Max-tree of G

```
1  forall nodes x in V do
2  |  make_set(x)
3  end
4  forall nodes x in V in nonincreasing order by weight do
5  |  forall neighbors y of x, where w(y) ≥ w(x) do
6  |  |  root_x ← find_root(x)
7  |  |  root_y ← find_root(y)
8  |  |  if root_x = root_y then
9  |  |  |  continue
10 |  |  end
11 |  |  parent[root_y] ← root_x
12 |  |  union(root_x, root_y)  // merge sets
13 |  end
14 end
15 parent[x] ← x, where x is the last processed node in the while loop
16 return parent array
```

more processors if the number of grey levels are fixed, the lower bound of the running time appears to be linear if no assumption is made of the number of grey levels. Because in the worst case, root path lengths can be in the order of n, the number of pixels.

Another parallel algorithm in [Moschini et al., 2017] constructs a pilot max-tree of the image with a reduced number of grey levels, in the order of the number of processors, using a distributed algorithm. The max-tree is later refined with the union-find method on each value partition in parallel, defined by the grey levels in the pilot tree. When iterating over neighbors with greater values, traversal through the pilot tree is required to determine relevant component (set) roots, which increases in cost when more processors are added. Our algorithm avoids this, as the work complexity does not depend on the number of processors.

3 Theory

The max-tree of a graph is constructed from nested connected component relations (component tree), obtained from thresholding the image at all values. A definition for injective graphs (no shared node weights) is given below. Any node weighted graph can be converted into an injective graph by defining a new weight function $u(x) = (w(x), \text{enum}(x))$, with lexical ordering, where enum is an enumerative function (bijective). The max-tree of the new graph is a max-tree of the non-injective graph by the definition in [Wilkinson et al., 2008]. In injective graphs, there is a bijection between components and nodes, which is a convenient property. Therefore, we restrict ourselves to injective graphs.

3.1 Notation

A *node-weighted graph* G in the context of max-trees is a tuple (V, E, w), where V is the finite set of *nodes*, $E \subseteq V \times V$ is the set of *directional edges* and $w : V \to W$ maps nodes to *weights* from some totally ordered set W. The graph is *injective* if w is injective. In an *undirected graph*, E is symmetric: $(x, y) \in E \equiv (y, x) \in E$. If G is *thresholded* at some weight t, the result is a graph $G_t = (V_t, E_t)$, where $V_t = \{x \in V : w(x) \geq t\}$ and $E_t = E \cap V_t \times V_t$. A *path* between nodes π_0 and π_n is a node sequence $(\pi_0, \ldots, \pi_n)$, where for every consecutive pair: $(\pi_i, \pi_{i+1}) \in E$. In a *simple path*, nodes appear at most once. A subset $V' \subseteq V$ is *connected* if a path exists in G between any pair of distinct nodes in V'. A graph is connected if all nodes are connected. *Connected components* or simply *components* of a graph are the largest connected subsets of V (no more nodes can be added). A *root* in a directed graph has no outgoing edges.

3.2 Max-tree

In this subsection we give a number of definitions to formalize the max-tree. Let $G = (V, E, w)$ be an injective undirected graph.

Definition 1 (component root)**.** For any component $C \subseteq V_t$ of a thresholded G, $\text{root}(C) = \arg\min\{w(x) : x \in C\}$. In other words, the root of a component is the node with minimal weight. Every node x can represent a unique C, in particular one that includes x at threshold $t = w(x)$, where $\text{root}(C) = x$. Therefore, the component root function is bijective.

Definition 2 (parent candidate path)**.** A *parent candidate path* of node x is any simple path $(\pi_0, \ldots, \pi_n)$, where $\pi_0 = x$ and $w(\pi_i) > w(\pi_0) > w(\pi_n)$, for $0 < i < n$.

Definition 3 (parent candidates)**.** For every node in G

$$\text{candidates}_G(x) = \{y \in V \mid \exists \text{ parent candidate path between } x \text{ and } y \text{ in } G\}$$

Visually, for any x and threshold $t = w(x)$, candidates surround component C containing $x = root(C)$ (figure 5).

Definition 4 (parent)**.** For every node in G,

$$\text{parent}_G(x) = \begin{cases} \arg\max\{w(y) : y \in \text{candidates}_G(x)\} & \text{if candidates}_G(x) \neq \emptyset \\ x & \text{otherwise} \end{cases}$$

When the parent of x equals x, it is a component root of unthresholded G.

Definition 5 (root path)**.** For every node, $\text{rootpath}_G(x) = (x, \text{parent}_G(x), \ldots)$. The root path represents a nested component structure, where every node corresponds to a component, given by the component root inverse.

In lemma 1 we show the equivalency between a node x in a component C and the component root of C in the root path of x. This result is used in the correctness proof of our max-tree algorithm.

Lemma 1. *Let C be a component of $G = (V, E, w)$ at any threshold. Then* $\forall x \in V : x \in C \equiv \text{root}(C) \in \text{rootpath}_G(x)$.

Proof. Given $x \in C$, we show $\text{root}(C) \in \text{rootpath}_G(x)$ by contradiction. Suppose $\text{root}(C) \notin \text{rootpath}_G(x)$. Since $\text{root}(C) \neq x$, $w(\text{root}(C)) < w(x)$. There exists a parent candidate path as prefix of a simple path between x and $\text{root}(C)$ in C, therefore $\text{candidates}_G(x)$ is not empty and $\text{parent}_G(x) \neq x$. Since we know that $\text{parent}_G(x)$ cannot equal $\text{root}(C)$, these steps can be repeated with ancestors of x to construct a root path with infinite length, which is not possible since there are a finite number of nodes. Therefore, $\text{root}(c)$ must be in $\text{rootpath}(x)$. Conversely, we want to show $\text{root}(C) \in \text{rootpath}_G(x) \rightarrow x \in C$. A path in G can be constructed between x and $\text{root}(C)$ via ancestors in $\text{rootpath}_G(x)$, using parent candidate paths, where the weights are all $\geq w(\text{root}(C))$, therefore $x \in C$. □

Definition 6 (max-tree)**.**

$$\text{maxtree}(G) = (V, \{(x, y = \text{parent}_G(x)) : x \in V \wedge x \neq y\})$$

Definition 6 reuses nodes from the original graph instead of components, since it is closer to the computer encoding. Any component can be reconstructed via root paths (lemma 1). To obtain a component tree from the max-tree, every node in the node set and edge relations can be replaced with the component root inverse. The definition also allows for multiple roots, if G is not connected. Therefore, the max-tree can be a forest.

When applied to a converted non-injective graph, with new weight function $u(x) = (w(x), \text{enum}(x))$, the max-tree can contain dummy nodes where parent nodes have the same original weight. A benefit of dummy nodes is that node sets (of components) are exactly encoded in the max-tree. In anti-extensive connected filter operations, nodes are implicitly deleted (and turned into dummy nodes).

3.3 Sequential algorithm

Algorithm 3 shows our novel sequential algorithm, where connected components in G' are iteratively bisected. Let $n = \#V$ (number of elements in V) and $m = \#E$. An array A is a finite vector. The bracket notation $A[i]$ is used to access element i of A. When a node x is used as index in an array, it is assumed that an enumerative function is implicitly applied, with indices less than n. Correctness of the algorithm is shown by proving $\text{parent}_G(x) = \text{parent}[x]$. Replacing the weight function by w' at line 6 gives the same ordering of values, and thus the same max-tree. The algorithm includes the following loop invariants:

1. $\forall (x, y) \in E' : w'(x) \text{ div } 2^k = w'(y) \text{ div } 2^k$. True before the loop: equal to 0 on both sides. Edges are cut between nodes in V_0 and V_1 at line 24, therefore this holds at the end of the loop.
2. $\forall x \in V : (\text{parent}_G(x) = \text{parent}_{G'}(x) \wedge x \text{ is not a root in } G) \vee (x \text{ is a root in } G' \wedge \text{parent}_G(x) = \text{parent}[x])$. Holds before the loop: right side for all roots in G and left side for all other nodes. If the right side is true, then $\text{candidates}_{G'}(x) = \emptyset$ and $\text{parent}[x]$ is unchanged at line 16. The right side remains true. If the left side is true, nodes can be split into three cases:
   (a) $x \in V_0$. We show that $\text{candidates}_{G'}(x)$ is unchanged. For every path between x and any $y \in V_0$ in G' (at the start of the loop), a path can be constructed in G' (at the end of the loop): replace consecutive subsequences through nodes in V_1 (part of a connected component in G_1) by the parent $z \in V_0$ of the connected component root. Since $w'(z) \geq w'(y)$ for every candidate path between x and y, the old candidate set is a subset of the new candidate set. The converse also applies, by reversing the steps. The left side of the invariant remains true.
   (b) $x \in V_1$ and is a root in G_1. Every parent candidate is a node in V_0. Since the left side of the invariant is true, $\text{candidates}_{G'}(x) \neq \emptyset$ and an edge (y, z) exists in E_{01}, where $y = \text{parent}_{G'}(x)$ and $root[z] = x$. This edge is iterated over at line 16. Either $\text{parent}[x] = x$ (initialization) or $\text{parent}[x]$ is some other candidate with lesser weight. In both cases, $\text{parent}[x]$ is correctly set at line 19 to $y = \text{parent}_{G'}(x)$. Paths from x to nodes in V_0 are cut at line 24 in G'. After, $\text{candidates}_{G'}(x)$ is empty and the right side of the invariant is true.

Algorithm 3: Sequential max-tree algorithm

input : Injective undirected graph $G = (V, E, w)$
output: Max-tree of G encoded as parent array

```
1  // Initializations
2  forall x ∈ V do
3  |  parent[x] ← x
4  end
5  G' = (V, E') ← G
6  w'(x) ← #{y ∈ V : w(x) > w(y)}  // Number of nodes with a lesser weight
7
8  k ← ⌈log_2(n)⌉
9  while k > 0 do
10 |  V_0 ← {x ∈ V : (w'(x) mod 2^k) div 2^(k−1) = 0}  // Current most significant bit is 0
11 |  V_1 ← V \ V_0
12 |  E_ab ← E' ∩ V_a × V_b
13 |  forall x ∈ V_1 do
14 |  |  // Add reference to the connected component root
   |  |  root[x] ← arg min{w'(y) : y ∈ V_1 ∧ ∃ path between x and y in G_1 = (V_1, E_11)}
15 |  end
16 |  forall (y, x) ∈ E_01 do
17 |  |  // Find neighbor y ∈ V_0 with maximal weight in G' for connected components in G_1
18 |  |  if parent[root[x]] = root[x] ∨ w'(y) > w'(parent[root[x]]) then
19 |  |  |  parent[root[x]] ← y
20 |  |  end
21 |  end
22 |  R ← {(y, parent[root[x]]) : (y, x) ∈ E_01}  // Fix candidate paths, R ⊆ V_0 × V_0
23 |  R ← R ∪ {(x, y) : (y, x) ∈ R}  // Symmetric closure, to ensure an undirected graph
24 |  E' ← E_00 ∪ E_11 ∪ R // Cut edges between nodes in V_0 and V_1
25 |  k ← k − 1
26 end
27 return parent array
```

(c) $x \in V_1$ and is not a root in G_1. Let C be the connected component containing x in G_1. From lemma 1, root(C) is in the root path of x in G'. The parent of x is either root$[x]$ or some other node in C. Since only paths to nodes in V_0 are cut, parent$_{G'}(x)$ is unchanged and the left side of the invariant remains true.

After the loop, only self edges remain in G' (invariant 1). For all nodes $x \in V$, the left side of invariant 3 is false and parent$_G(x) =$ parent$[x]$.

The time complexity of algorithm 3 is $\mathcal{O}((n+m)\log n)$. An explanation follows. Let S be a sorted array of $(w(x), \text{enum}(x))$ pairs (increasing), for all nodes in V. Each pair in S is denoted as $S[j] = (w_j, \text{enum}_j)$, for $j \in [0..n)$. Change each ununitialized element $w'[\text{enum}_j]$ to value (or rank) j. Array w' maps enumerated nodes to ranks. Due to sorting, these steps can be done in $\mathcal{O}(n \log n)$. Set E' is assumed to be stored in an array of enumerated node pairs (a, b), where $w'[a] \leq w'[b]$, and is sorted by $w'[a]$ (increasing). It is unnecessary to store two directed edges for each undirected edge. Inside the main loop, there are three explicit operations on arrays: 1) Component labeling (by root references), 2) maximum neighbor (parent) of components, and 3) updating edges. Root references can be determined in $\mathcal{O}(n+m)$, by initializing $root[j]$ to j, for $j \in [0..n)$ and iterating over (already sorted) edges (a, b) of E_{11} and changing $root[b]$ to $root[a]$. The other operations are similarly trivial and linear in n or m. After updating edges (a, b) in E', $w'[a]$ remains $\leq w'[b]$. The outer loop has $\mathcal{O}(\log n)$ iterations. Combined, the loop has $\mathcal{O}((n + m)\log n)$ steps. The lower bound of max-tree construction is $o(n \log n)$ time, because the max-tree of a complete graph is a sorted list. In the case of images, the algorithm is optimal, since $m = \mathcal{O}(n)$ and an image can be constructed to sort $\mathcal{O}(n)$ values. For example, values surrounding a grid with greater values.

3.4 Parallelization

First we give some computer model preliminaries. In the following, the CRCW-arbitrary PRAM model [Immerman, 1989] is used with an additional condition, which we explain. Pseudocode is extended by a parallel operation, which divides n elements between p processors, such that each processor has $\Theta(n/p)$ elements. Outside parallel operations, code is executed on one arbitrary processor p_0. Memory is shared between processors and is synchronized before and after parallel operations. If processors write to the same memory address in a parallel operation, an arbitrary processor succeeds. Our condition is that the distribution of the value being written is unchanged, which has importance for time complexities. For example, if two (uniformly) random real numbers in the range [0..1) are written to the same address, selecting the minimal value would change the written value distribution. It is allowed to prioritize certain processors.

Fundamental operations on a processor in $\mathcal{O}(1)$ time are *primitive* operations. The *work* complexity of an algorithm is the sum of primitive operations over all processors. The *depth* complexity (in our context) is the worst-case sum of primitive operations by p_0, if p is large enough such that each processor has $\mathcal{O}(1)$ elements in all parallel operations. If an algorithm has a complexity dependent on a random process, where each case has a probability, the *expected* complexity of an algorithm is the sum of probabilities multiplied by complexity. The time complexity of a parallel algorithm is $\mathcal{O}(\text{worst-case work}/p + \text{worst-case depth})$ [Blelloch, 1996]. A frequently used parallel algorithm is the *pack* (or filter) algorithm, to select and pack items from an array, using a boolean function f on each element. The work and depth complexities for packing are $\mathcal{O}(n)$ and $\mathcal{O}(\log p)$ respectively [Blelloch, 1990].

Lemma 2. *Let n and m be positive integers with $m \leq n$. Then $(m+1)(1-(1-m/n)^n) \geq m + m/n$.*

Proof. Trivially true for $n = 1$ or $m = n$. Consider $n \geq 2$ and $m < n$.

$$
\begin{aligned}
& (m+1)(1-(1-m/n)^n) \geq m + m/n \\
\Leftrightarrow \quad & 1 - m/n - (m+1)(1-m/n)^n \geq 0 \\
& \text{Let } c = 1 - m/n, \text{ where } 0 < c < 1 \\
\Leftrightarrow \quad & c - (n - cn + 1)c^n \geq 0 \\
\Leftrightarrow \quad & c(1 - (n - cn + 1)c^{n-1}) \geq 0 \\
\Leftrightarrow \quad & 1 - (n - cn + 1)c^{n-1} \geq 0 \qquad (1)
\end{aligned}
$$

Let f be the left hand side of inequality 1. The derivative of f with respect to c is $c^{n-2}(1 + (c-1)n^2)$, which has zeros at 0 and $1 - 1/n^2$, and is negative for $c \in (0, 1 - 1/n^2)$. So f is decreasing in that range. At $c = 1 - 1/n < 1 - 1/n^2$, f simplifies to $1 - 2((n-1)/n)^{n-1}$, which is ≥ 0 for every $n \geq 2$. This implies that $f \geq 0$ for every $c \in [0, 1/n]$, and by working backwards, $(m+1)(1-(1-m/n)^n) \geq m + m/n$. □

Algorithm 4: Parallel minima

input : Array $A = (a_0, \ldots, a_{n-1}), a_i \in X$
Set membership function set : $X \to [0..k), k \leq n$
Injective weight function $w : X \to \mathbb{R}$
output: Minimum weight of each nonempty set

```
1  parallel forall i ∈ [0..k)
2  |  min[i] ← ∞
3  end
4  while #A > 0 do
5  |  parallel forall i ∈ [0..#A)
6  |  |  j ← random integer in [0..#A) with equal probability
7  |  |  min[set[a_j]] ← w(a_j)  // Possible concurrent write
8  |  end
9  |  // Remove elements that have a value ≥ the currently assigned minimum
10 |  f(a_i) ← boolean function that returns true iff w(a_i) < min[set[a_i]]
11 |  parallel pack A with f
12 end
13 return min
```

Algorithm 4 determines in parallel the minimum of each set, which will be used in the parallel max-tree algorithm. In each iteration of the loop, a fraction of elements are removed, independent from the previous iteration. The expected fraction is $\geq$ some constant $c < 1$. The chance that at least one element in a set S is selected at line 7 is $1 - (1 - \#S/\#A)^{\#A} \geq 1 - e^{-m}$. If at least one element is selected (with equal probability), $(\#S+1)/2$ of the elements are expected to be removed during packing. We assume that the elements are unique. If not, more elements are removed on average. Consider the concatenation of every array $S_\cup$, where $\#S_\cup = n$. If the algorithm is applied to only $S_\cup$, the expected number of removed elements in each iteration is $\leq$ the sum of expected removed elements for each set: $(\#A + 1)/2 \leq \sum_{k=1}^{m}(\#S_k + \#S_k/\#A)/2$ (lemma 2). Therefore, for the worst-case work and depth complexity we only have to consider the concatenation of arrays. The amount of work in an iteration is $c\#A$, for some constant c. The expected work complexity $W(n)$ with $W(0) = 0$ is the recurrence relation

$$
\begin{aligned}
W(n) &= cn + \frac{1}{n}\sum_{k=1}^{n} W(n-k) \\
&= cn + \frac{1}{n}\left(W(n-1) + \sum_{k=1}^{n-1} W(n-1-k)\right) \\
&= cn + \frac{n-1}{n}(W(n-1) - c(n-1)) + \frac{1}{n}W(n-1) \\
&= cn + c\frac{(1-n)(n-1)}{n} + W(n-1) \\
&= cn + c\frac{-n^2 + 2n - 1}{n} + W(n-1) \\
&\leq 2c + W(n-1) \\
&= 2cn
\end{aligned}
$$

The expected number of loop iterations is the recurrence relation

$$\begin{aligned} R(n) &= 1 + \frac{1}{n}\sum_{k=1}^{n} R(n-k) \\ &= 1 + \frac{n-1}{n}(R(n-1)-1) + \frac{1}{n}R(n-1) \\ &= \frac{1}{n} + R(n-1) \\ &= \sum_{k=1}^{n} \frac{1}{k} \\ &= \mathcal{O}(\log n) \end{aligned}$$

Since the depth of packing is $\mathcal{O}(\log p)$, the expected depth of algorithm 4 is $\mathcal{O}(\log n \log p)$.

Parallel algorithms are known for most operations in algorithm 3. Parallel sorting has expected work and depth complexities $\mathcal{O}(n \log n)$ and $\mathcal{O}(\log n)$ respectively [Blelloch, 1996]. Root references of connected components are split into two steps: component labeling and root finding. Component labeling has expected work and depth complexities $\mathcal{O}(n)$ and $\mathcal{O}(\log n)$ respectively [Gazit, 1991]. Each component C is represented by some arbitary node in C. For any node x in C, let $\mathrm{cc}[x]$ contain a reference to the arbitrary node. Component root references are determined by applying parallel minima, where $A = \{e = (x,y) : (x,y) \in E_{11} \wedge w'[x] \leq w'[y]\}, \mathrm{set}(e) = cc[x]$ and $w(e) = w'[x]$. Component parents are determined by applying parallel maxima, where $A = \{e = (x,y) : (x,y) \in E_{01}\}, \mathrm{set}(edge) = cc[y]$ and $w(e) = w'[x]$. Both applications have expected work and depth complexities of $\mathcal{O}(n+m)$ and $\mathcal{O}(\log m \log p)$ respectively. Combined with the outer loop, the expected time complexity is $\mathcal{O}((n+m)/p \log n + \log^2(n+m) \log p)$.

4 Implementation

Algorithm 5: Overview of the implementation

input : Image $I = (V, E, w)$, a weighted undirected connected graph
output: Max-tree of I represented by array parent

```
1 reduce_edges() // Reduce edge count
2 estimate_quantiles() // Of node weights
3 partition_graph() // By quantiles
4 export_edges()
5 sort_exported_edges()
6 maxtrees_of_partitions()
```

While the theory is aimed at a general algorithm, this section gives an overview of a software implementation and challenges that were encountered. Algorithm 5 shows the main segments, which will be explained in more detail in the following subsections. There are four main optimizations:

1. The parallel algorithm requires storage of edges in memory. By reducing the number of edges, memory usage is less and run times are faster.
2. Initially, edges are highly structured due to the image grid. Structured edges can be stored more efficiently. Rewired edges, linking nodes anywhere in image, are stored separately.
3. In practice, the number of processing cores are far fewer than the number of image elements. When the graph has a number of partitions approximately equal to the number of cores, a sequential max-tree algorithm is applied to each partition.
4. The graph is partitioned by quantiles of (remaining) image values instead of most significant bits, for more balanced partitions.

Figure 6 shows an edge reduction example and partition steps. Two parallel algorithms are assumed to be already implemented: parallel for and parallel filter/pack, to select and pack items from an array.

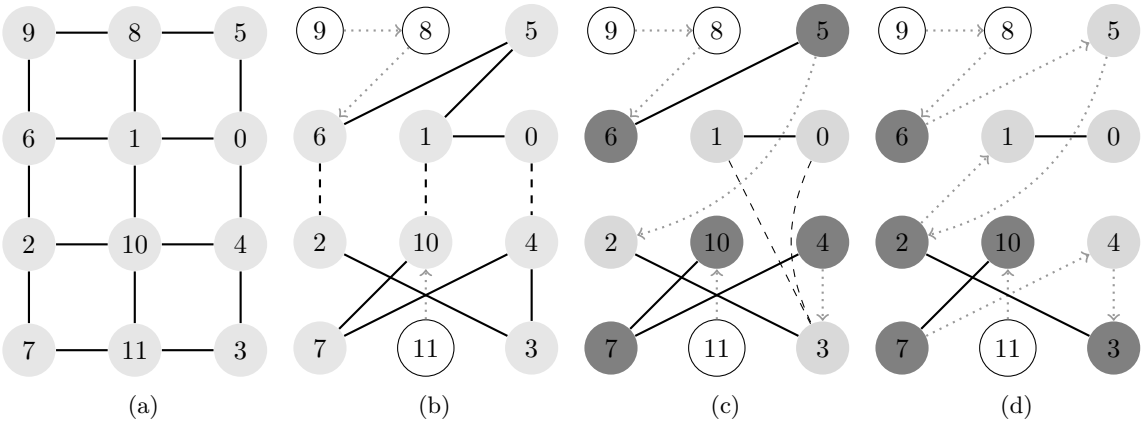


Figure 6: Image graph (a), reduced graph when horizontally dividing the image into 2 blocks (b), after one partition step (c) and after two partition steps (d). Solid lines are local edges, dashed lines are global edges and dotted directed edges are known parent relations. Dark gray nodes are in upper halves of partitions (above partition median), light gray nodes in lower halves and unfilled nodes are unused in partition steps and quantile estimation. Median estimates are 4 and (2, 6) in partition steps 1 and 2 respectively.

4.1 Edge reduction

A 2-dimensional image typically contains approximately $2N$ (4 neighbors) or $4N$ (8 neighbors) undirected edges, where N is the number of pixels. This number can quickly increase for higher dimensional images. We use two optimizations to reduce the edge count.

The image is divided into blocks for parallel processing. The first optimization modifies edges locally per block. A distinction is made between local edges, linking nodes in a block, and global edges (which require more space). Local edges can, for example, be stored as 16-bit index or 2×8-bit position pairs. Local max-trees are constructed for each block in parallel, using the trie priority queue algorithm from [Teeninga and Wilkinson, 2020]. Let E' be the symmetric closure of the union of all local max-tree edges and undirected edges between blocks. Boundary nodes are defined as having (global) edges to nodes in other blocks. Graph $G' = (V, E', w)$ has an equivalent max-tree compared to $G = (V, E, w)$, because for every node a path to the parent is maintained, potentially through boundary nodes (unequal to the parent), where weights (of nodes on the path) are $\geq$ the initial weight $w(\text{root}(C))$. This step alone reduces the number of edges to approximately N, if the block size is not too small.

Edges are stored as ordered pair (by weight) per block, and local and global edges are stored separately in arrays. The arrays are sorted using radix sort to make the next segments of the algorithm more efficient. A block initially contains all global edges that link the current block to previous blocks.

The second optimization is to remove isolated peak components. If a threshold set of image elements (peak component) does not contain boundary nodes (is isolated), parents of nodes in the threshold set can already be determined. Edges part of isolated peak components can be removed after setting the parent.

Figure 6b shows an example where both optimizations are applied.

4.2 Quantile estimation

To obtain q (power of two) graph partitions of approximately equal size in edges, quantiles are used instead of the most significant bits. Remaining nodes are sampled (in parallel) and $(\mathsf{value}, \mathsf{index})$ pairs are stored in an array S, which is then sorted lexically (LSB radix, in parallel). Quantile $p = k/q$ is estimated as $S[Mp]$, where $0 < k < q$ and M is the number of samples, dividable by q. Partition k includes quantile k. To determine the number of samples needed, sample quantile p is distributed as $(1/M)B(M, p)$, where B is the binomial distribution. The distribution can be approximated by a normal distribution with variance $(1/M^2)p(1-p)M = p(1-p)/M$, which is maximal at $p = 0.5$. When allowing for a maximum 5% error in 95% of the cases in the standard deviation at $p = 0.5$, and accounting for the number of partitions, $(0.05/1.95996/q)^2 \approx 0.25/M$. Solving for M gives $M \approx 384q^2$. This value quickly

increases with more partitions. If the number of samples are greater than the number of graph nodes, it is more efficient to determine quantiles from all (remaining) graph nodes instead of sampling. Figure 7 shows a simulation of the error to verify the formula, which seems valid.

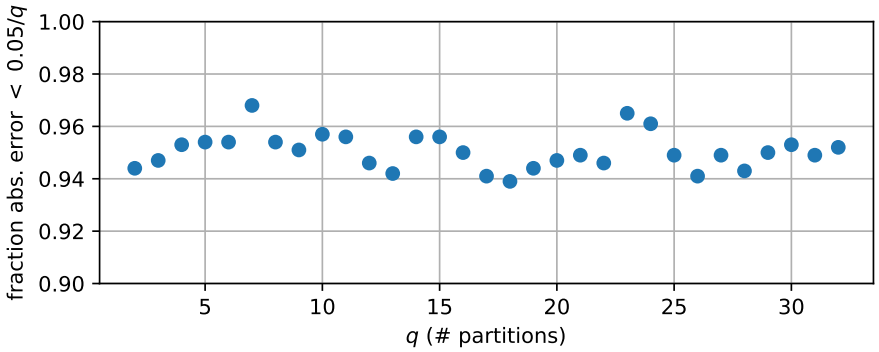


Figure 7: Fraction of absolute errors below $0.05/q$ versus q. Quantile p is estimated by the median of $384q^2$ samples with a random value in $[0, 1)$. This is repeated 1000 times per q.

4.3 Graph partitioning

Algorithm 6: Graph partitioning overview

```
n_partitions ← 1
while n_partitions < max_partitions do
    connected_components() // Of 11-edges
    find_minimal_roots() // Of connected components
    find_parents() // Of connected component roots using 01-edges
    update_edges() // Rewire 01-edges and move to global edges if they were local
    n_partitions ← 2 n_partitions
end
```

Each partition in the graph is iteratively bisected. Algorithm 6 gives as overview and is based on the parallel maxtree algorithm in the theory section. We will describe how each function processes image blocks in parallel. Ordered edges are partitioned into three subsets: 1) both endpoints have a value $<$ the partition median (00-edges), 2) only the first endpoint is $<$ the median (01-edges) and 3) both endpoints are $\geq$ the median (11-edges).

Function `connected_components()` determines component roots using 11-edges, and adds a root reference to nodes with a weight $\geq$ partition median. Initially, $\mathsf{roots}[a] \leftarrow a$. Connected components are partially determined for local 11-edges (a, b) per block. This is accomplished by iterating over sorted edges and setting $\mathsf{roots}[b] \leftarrow \mathsf{roots}[a]$. These references can be updated later (in constant time) once the global roots are known. Edges (a, b) from the (copied) global 11-edges set are updated to $(\mathsf{roots}[a], \mathsf{roots}[b])$. The connected components algorithm for (copied) global 11-edges is based on random mate from [Reif, 1985]. Edges are iteratively contracted in parallel. In each iteration, nodes are randomly and implicitly labeled with 0 or 1 using a universal integer hash function (from [Wölfel, 2003]) on the index. The hash function changes after each iteration. Nodes with a 0-label can merge into any node with a 1-label by updating the root reference, assuming an edge exists between them. After the merge step, edges (a, b) are updated to $(\mathsf{roots}[a], \mathsf{roots}[b])$ and self-loops are removed. The number of nodes are approximately reduced by a constant fraction. This is repeated until all edges are contracted. By updating references $\mathsf{roots}[a] \leftarrow \mathsf{roots}[\mathsf{roots}[a]]$, for any node a, in reverse order of node merges, paths to connected component roots are efficiently compressed (in parallel). Theoretical guarantees for linear work complexity are only given if the graph is planar and duplicate edges are removed, which is not done here. Nevertheless, work complexity appears to be linear (or close to) for 2-D and 3-D images (figure 8).

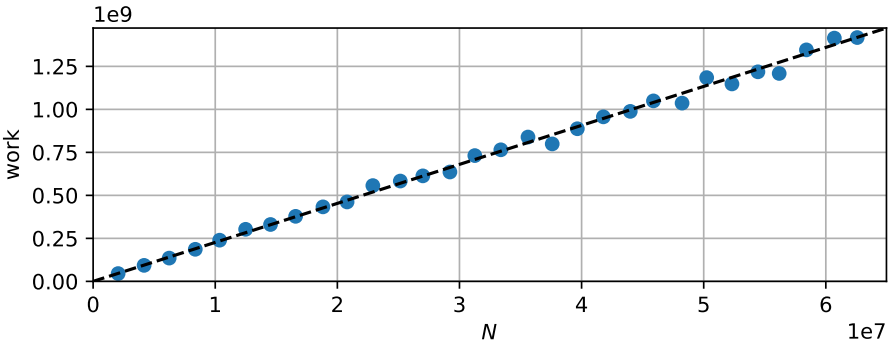


Figure 8: Random mate connected components algorithm: Work vs N. Work is the total number of elements iterated over and N is the image element count. The graph is a 3-D image with 6-neighbor connectivity.

Function `find_minimal_roots()` finds minimal connected component roots and `find_parents()` uses 01-edges to find parents of connected component roots. In both cases the parallel minima/maxima of sets algorithm is applied. Array access is not randomized. The assumption is that any performance decrease from write conflicts is less than from random array accesses. This may be different on GPUs.

Function `update_edges()` updates 01-edges (a, b) to $(a, \mathsf{parent}[\mathsf{roots}[b]])$. Local 01-edges are moved to the global edge array (in the same block) by merging two sorted arrays. Partitions are bisected at the end of the iteration. Figure 6 shows examples.

4.4 Exporting edges and sorting

Before constructing max-trees of partitions, edges are exported to a single array. Offsets are calculated with a prefix sum, using known edge counts per partition in each block. Exported edges (a, b) are sorted by weight of a in parallel with least significant bit (LSB) radix sort. The choice for number of bits per digit depends on CPU cache size and number of entries for output streams in the Translation Lookaside Buffer (TLB). On current hardware, 11-bit digits is a good choice to sort 32-bit values in 3 write iterations.

A potential problem is address infix aliasing on some architectures, which reduces throughput. If 11-bit digits are chosen, up to 2048 output streams are possible. Ideally, CPU cache entries for output streams are evenly spread to reduce cache misses. This is not always the case. For example, when sorting an array of 2^{26} items (2×32-bit value and index pair), where value = index, on an Intel(R) Core(TM) i5-4460, the throughput reduces to 10 Mi/s (megaitem/s) from 120 Mi/s (random unsigned 32-bit integers).

To reduce aliasing, a small enough buffer is used to locally sort a block of the array. Items are copied from the buffer per digit value to the output array. There are 3 further considerations for the buffer size: 1) each block of items has an associated histogram, so a smaller buffer increases overhead, 2) less throughput if the buffer is too large and cannot be kept in cache, and 3) ideally an even spread over data cache entries, which is data dependent.

Figure 9 shows the throughput of randomly picked block lengths in a range. Data points for powers of two are added separately. Decreased throughput is assumed to be because of aliasing. In this case, a buffer size close to 500k bytes seems optimal. When testing on an AMD Ryzen 9 5900X 12-Core Processor, a buffer seems unnecessary.

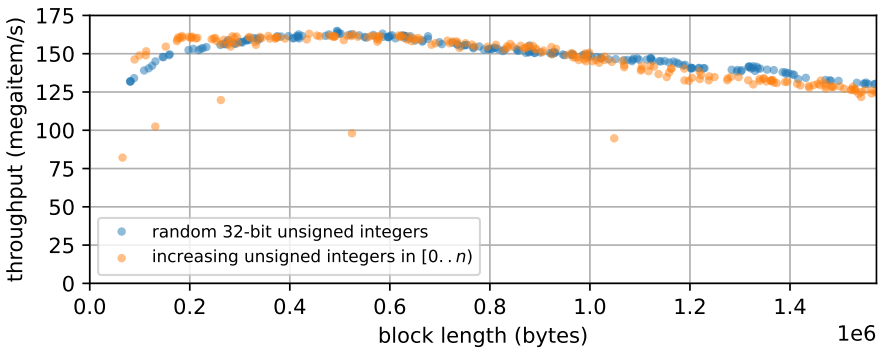


Figure 9: Sorting throughput versus randomly picked block lengths, 11-bit digits. Array of 2^{16} items: 2×32-bit (value, index) pairs.

4.5 Max-trees of partitions

Since only an edge array is available, a flooding maxtree algorithm cannot be applied efficiently to a partition. Finding neighbors of a neighbor cannot be done in constant time in this case. Union-find max-tree construction [Berger et al., 2007] can be done efficiently. We modify the union-find algorithm to include ranks [Tarjan, 1975], which has better time complexity.

Our implementation uses one byte more memory per node to store the rank, and a boolean flag to determine if a node is a set root. If a node is a root, it has a reference to the connected component root it is representing. Otherwise, it has a reference to the next node on the path to the set root. The order of edges (a, b) with the same endpoint a is arbitrary, so care has to be taken that partition roots are equal to connected component roots in graph partitioning. Otherwise, root paths of the entire image can be disconnected.

5 Results

Our algorithm (single threaded) is compared to state of the art sequential algorithms, and the parallel performance is tested. Performance on images with random values is assumed to be similar to real worse cases, or worse. The reasoning is that for lesser image values, components span the entire image and parents are at random locations in the image, which results in more CPU cache misses. The Linux test system contains 4 AMD Opteron(TM) Processor 6276 processors (released in 2011) with 64 cores total. The implementation is written in C++ and compiled with g++ 7.5.0. The code is available at https://github.com/pauldavt/max-tree-parallel.

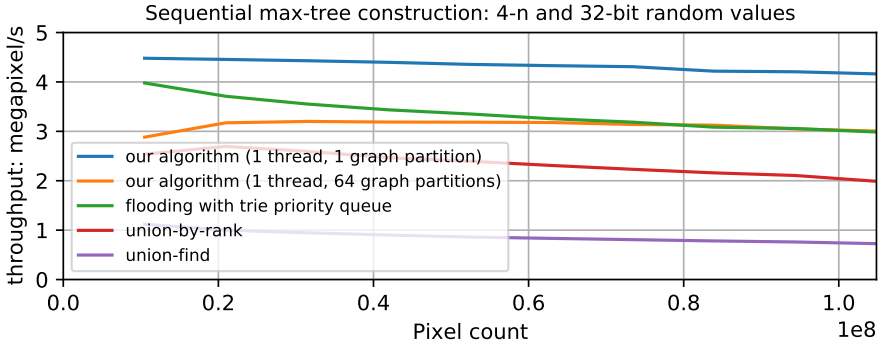


Figure 10: Comparison between our max-tree construction algorithm with 1 thread and sequential algorithms. Images have 2 dimensions and non-boundary pixels have 4 neighbors. Greater throughput is better.

Sequential results are shown in figure 10. The timing of our algorithm with 64 partitions (needed in parallel results) shows overhead compared to a single partition. Max-tree construction with a trie priority queue has been used before on tiles to construct boundary trees. Here it is applied to the entire image. Union-by-rank is implemented similarly as union-by-rank on exported edges in the parallel algorithm, except that pixels are iterated over instead of an edge array. Memory overhead compared to only path compression is 8 bits per image element. Union-find only uses path compression. Optimizations for our parallel algorithm also help in the sequential case with 1 graph partition, as the throughput is better than current sequential algorithms. When images are large enough, the throughput with 64 partitions is as good as current sequential algorithms. Any speedup achieved when adding more processors would also be at least that speed-up compared to current sequential algorithms.

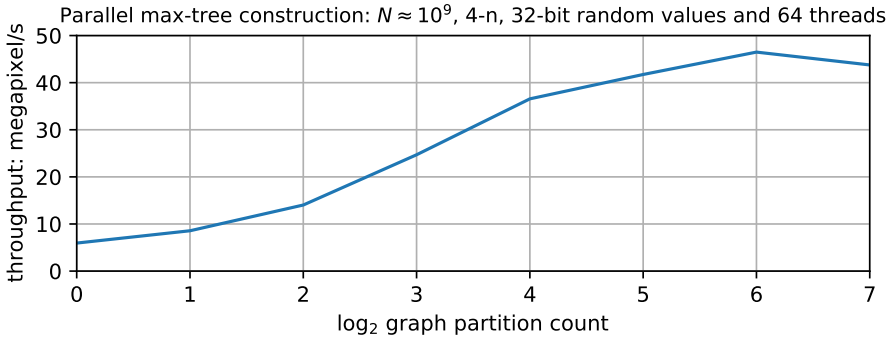


Figure 11: Throughput versus number of graph partitions ($\log_2$ scale) for a gigapixel image. Images have 2 dimensions, non-boundary pixels have 4 neighbors and the pixel count is N. Greater throughput is better.

Figure 11 shows the throughput when changing the number of graph partitions (a power of two). On this system, the greatest throughput is achieved at 64 graph partitions, equal to the number of cores. When testing images with a smaller size, the optimal number of partitions were lower in some cases.

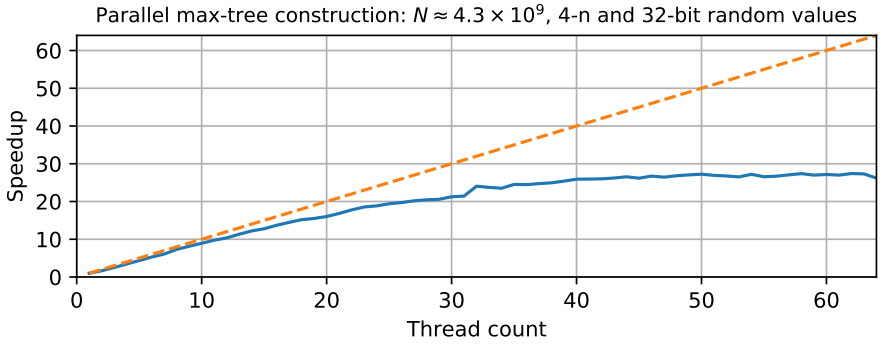


Figure 12: Speedup of our parallel algorithm with 64 threads versus 1 thread. Images have 2 dimensions, non-boundary pixels have 4 neighbors and the pixel count is N.

Figure 12 shows the achieved speedup for a 4.3 gigapixel image. The speed-up is near linear up to 10 processors. The maximum speedup is 27.4 at 63 threads, a significant improvement compared to a single thread. A possible reason for the speedup drop are bottlenecks in memory transfers to the CPU.

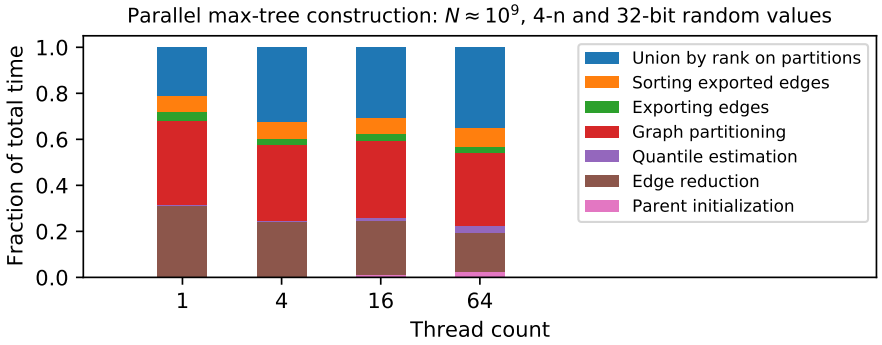


Figure 13: Pipeline timings as fraction of the total time. Images have 2 dimensions, non-boundary pixels have 4 neighbors and the pixel count is N.

Parallel pipeline timings are shown in Figure 13. When comparing 64 threads to 1 thread, the speedup is greatest for edge reduction at 46, and least for parent initialization and quantile estimation (1.6 and 2.5 respectively). Speedups for other sections are between 15.7 and 18.2. Edge reduction is highly parallelizable (multiple operations on cached image blocks), while other sections are more memory bound: non-local random memory accesses and/or simple large array operations. Memory management of the operating system might also contribute (physical allocations). Quantile estimation has a noticeable increase in fraction of the total time, since it increases quadratically with the number of threads. However, since the algorithm sorts (in parallel) when the number of samples approaches the population size, the worst case time complexity is still acceptable.

The throughput for 8-neighbors on a 4.3 gigapixel image is 32.4 megapixel/s, compared to 44 megapixel/s for 4-neighbors. When applying the parallel algorithm to a real world example, an astronomical image (0.442 gigapixel, 32-bit floating point and 4-neighbors), the throughput is 47 megapixel/s. Comparable to the result of random 32-bit values.

When comparing throughput to the refined pilot tree method in [Moschini et al., 2017] for images Float4 (random), ESO, PRAGUE and LOFAR (3D), the speed-up in our implementation is between 3.9× (LOFAR) and 9.5× (Float4), although the area attribute is not calculated.

6 Future work

The main missing feature in our algorithm is calculating attributes in parallel. Current max-tree algorithms merge known attributes of pixel subsets when assigning a parent. In our algorithm the attributes of pixel subsets are not known yet, since the current connected components algorithm does not support parallel attribute calculation. Another option is to calculate attributes in parallel after max-tree construction. In literature this is a leaffix operation [Joseph et al., 1992], which runs in $\mathcal{O}(\log n)$ time (with enough processors) and has $\mathcal{O}(n)$ work complexity. The leaffix operation requires a sorting step beforehand with $\mathcal{O}(n \log n)$ work complexity.

Dual-input max-trees [Ouzounis and Wilkinson, 2007a] that use a mask as secondary image can be constructed with our parallel algorithm. The max-tree can be constructed from the mask image and attributes of the max-tree can be computed afterwards by using values of the original image. Alpha-trees [Ouzounis and Soille, 2012] are similar to max-trees. *Quasi-flat zones* are regions of pixels where each pair of neighboring nodes has a maximum weight difference. Quasi-flat zones with lesser maximum differences are nested in quasi-flat zones with greater maximum differences, and a min-tree (minima at leaves) can be constructed. A method is given in [Ouzounis and Soille, 2012] to construct a graph that can be used in max-tree algorithms (including our parallel algorithm), and fast algorithms are given in [You et al., 2019][You et al., 2023], which could be used to add alpha tree construction in our implementation.

GPUs typically have more memory bandwidth and more parallel processing power compared to CPUs, and a GPU implementation could have better throughput. A GPU implementation of the theoretical algorithm seems straightforward, assuming fast parallel sort and filter operations are available in a software framework. Edge reduction is an important optimization to reduce memory usage of edges, which could be challenging to implement efficiently.

When images are too large to be processed on shared memory systems, and they cannot be split into smaller images, distributed computing is the only solution. The challenge here is to minimize communication between different computers, the likely limiting factor in throughput. The image representation goes from tiles in a grid to pixel sets per range of greyvalues, and a high communication overhead seems difficult to avoid, assuming images with high bit depth.

7 Conclusion

We have introduced a novel max-tree construction algorithm with optimal sequential time complexity for images with high dynamic range, and improved parallel time complexity compared to state of the art methods. The implementation with a single thread has at least the same performance as other sequential algorithms. Parallel throughput is up to 9.5× greater compared to the pilot tree method. While the latter method includes area attributes, it does not greatly impact throughput. Scalability on the tested system is good: 27.4x speedup with 64 processors on a 4.3 gigapixel image with random 32-bit values.

8 Acknowledgements

We want to thank Wim Hesselink for providing a rigorous proof of our sequential max-tree algorithm. We included elements in our proof.

We made use of a computer which was funded by the Netherlands Organisation for Scientific Research (NWO) under project number 612.001.110.

References

[Berger et al., 2007] Berger, C., Géraud, T., Levillain, R., Widynski, N., Baillard, A., and Bertin, E. (2007). Effective component tree computation with application to pattern recognition in astronomical imaging. In *2007 IEEE International Conference on Image Processing*, volume 4, pages IV–41. IEEE.

[Blelloch, 1990] Blelloch, G. E. (1990). Prefix sums and their applications.

[Blelloch, 1996] Blelloch, G. E. (1996). Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97.

[Blin et al., 2022] Blin, N., Carlinet, E., Lemaitre, F., Lacassagne, L., and Géraud, T. (2022). Max-tree computation on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3520–3531.

[Breen and Jones, 1996] Breen, E. J. and Jones, R. (1996). Attribute openings, thinnings, and granulometries. *Computer vision and image understanding*, 64(3):377–389.

[Carlinet and Géraud, 2013] Carlinet, E. and Géraud, T. (2013). A comparison of many max-tree computation algorithms. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 73–85. Springer.

[Gan et al.,] Gan, H., Gazagnes, S., Babai, M., and Wilkinson, M. H. F. *Self-Organising Attribute Maps and Pattern Spectra: Novel Explorative Data Analysis Tools for High-Dimensional Vector-Attribute Filtering*. PhD thesis, University of Groningen, Netherlands.

[Gazagnes and Wilkinson, 2021] Gazagnes, S. and Wilkinson, M. H. (2021). Distributed connected component filtering and analysis in 2d and 3d tera-scale data sets. *IEEE Transactions on Image Processing*, 30:3664–3675.

[Gazit, 1991] Gazit, H. (1991). An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal on Computing*, 20(6):1046–1067.

[Haigh et al., 2021] Haigh, C., Chamba, N., Venhola, A., Peletier, R., Doorenbos, L., Watkins, M., and Wilkinson, M. H. (2021). Optimising and comparing source-extraction tools using objective segmentation quality criteria. *Astronomy & Astrophysics*, 645:A107.

[Immerman, 1989] Immerman, N. (1989). Expressibility and parallel complexity. *SIAM Journal on Computing*, 18(3):625–638.

[Joseph et al., 1992] Joseph, J. et al. (1992). An introduction to parallel algorithms.

[Kazemier et al., 2017] Kazemier, J. J., Ouzounis, G. K., and Wilkinson, M. H. (2017). Connected morphological attribute filters on distributed memory parallel machines. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 357–368. Springer.

[Merigot and Petrosino, 2008] Merigot, A. and Petrosino, A. (2008). Parallel processing for image and video processing: Issues and challenges. *Parallel Computing*, 34(12):694–699.

[Mighell, 2010] Mighell, K. J. (2010). Crblaster: a parallel-processing computational framework for embarrassingly parallel image-analysis algorithms. *Publications of the Astronomical Society of the Pacific*, 122(896):1236.

[Moschini et al., 2017] Moschini, U., Meijster, A., and Wilkinson, M. H. (2017). A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images. *IEEE transactions on pattern analysis and machine intelligence*, 40(3):513–526.

[Ouzounis et al., 2011] Ouzounis, G., Soille, P., and Pesaresi, M. (2011). Rubble detection from vhr aerial imagery data using differential morphological profiles. In *34th Int. Symp. Remote Sensing of the Environment*.

[Ouzounis and Soille, 2010] Ouzounis, G. K. and Soille, P. (2010). Differential area profiles. In *2010 20th International Conference on Pattern Recognition*, pages 4085–4088. IEEE.

[Ouzounis and Soille, 2012] Ouzounis, G. K. and Soille, P. (2012). The alpha-tree algorithm. *JRC Scientific and Policy Report*.

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

[Ouzounis and Wilkinson, 2007b] Ouzounis, G. K. and Wilkinson, M. H. (2007b). A parallel implementation of the dual-input max-tree algorithm for attribute filtering. In *ISMM (1)*, pages 449–460.

[Pesaresi et al., 2013] Pesaresi, M., Huadong, G., Blaes, X., Ehrlich, D., Ferri, S., Gueguen, L., Halkia, M., Kauffmann, M., Kemper, T., Lu, L., et al. (2013). A global human settlement layer from optical hr/vhr rs data: Concept and first results. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 6(5):2102–2131.

[Pesaresi and Kanellopoulos, 1999] Pesaresi, M. and Kanellopoulos, I. (1999). Detection of urban features using morphological based segmentation and very high resolution remotely sensed data. In *Machine Vision and Advanced Image Processing in Remote Sensing*, pages 271–284. Springer.

[Reif, 1985] Reif, J. H. (1985). Optimal parallel algorithms for integer sorting and graph connectivity. Technical report, Harvard University, Cambridge MA, Aiken Computation Lab.

[Salembier et al., 2018] Salembier, P., Liesegang, S., and López-Martínez, C. (2018). Ship detection in sar images based on maxtree representation and graph signal processing. *IEEE Transactions on Geoscience and Remote Sensing*, 57(5):2709–2724.

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

[Salembier and Wilkinson, 2009] Salembier, P. and Wilkinson, M. H. F. (2009). Connected operators: A review of region-based morphological image processing techniques. *IEEE Signal Processing Magazine*, 26(6):136–157.

[Serra, 1982] Serra, J. (1982). *Image Analysis and Mathematical Morphology*, volume 1. Academic Press, New York.

[Serra, 1988] Serra, J., editor (1988). *Image Analysis and Mathematical Morphology, Vol. 2: Theoretical Advances*, volume 2. Academic Press, New York.

[Tarjan, 1975] Tarjan, R. E. (1975). Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225.

[Teeninga et al., 2015] Teeninga, P., Moschini, U., Trager, S. C., and Wilkinson, M. H. (2015). Improved detection of faint extended astronomical objects through statistical attribute filtering. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 157–168. Springer.

[Teeninga and Wilkinson, 2020] Teeninga, P. and Wilkinson, M. H. (2020). *Submitted to Pattern Recognition Letters*.

[Urbach et al., 2007] Urbach, E. R., Roerdink, J. B., and Wilkinson, M. H. (2007). Connected shape-size pattern spectra for rotation and scale-invariant classification of gray-scale images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(2):272–285.

[Westenberg et al., 2007] Westenberg, M. A., Roerdink, J. B., and Wilkinson, M. H. (2007). Volumetric attribute filtering and interactive visualization using the max-tree representation. *IEEE Transactions on Image Processing*, 16(12):2943–2952.

[Wilkinson et al., 2008] Wilkinson, M. H., Gao, H., Hesselink, W. H., Jonker, J.-E., and Meijster, A. (2008). Concurrent computation of attribute filters on shared memory parallel machines. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(10):1800–1813.

[Wilkinson et al., 2016] Wilkinson, M. H., Pesaresi, M., and Ouzounis, G. K. (2016). An efficient parallel algorithm for multi-scale analysis of connected components in gigapixel images. *ISPRS International Journal of Geo-Information*, 5(3):22.

[Wilkinson, 2011] Wilkinson, M. H. F. (2011). A fast component-tree algorithm for high dynamic-range images and second generation connectivity. In *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pages 1021–1024. IEEE.

[Wölfel, 2003] Wölfel, P. (2003). *Über die Komplexität der Multiplikation in eingeschränkten Branching-programmmodellen*. PhD thesis, Technical University of Dortmund, Germany.

[Xu et al., 2015] Xu, Y., Géraud, T., and Najman, L. (2015). Connected filtering on tree-based shape-spaces. *IEEE transactions on pattern analysis and machine intelligence*, 38(6):1126–1140.

[You et al., 2019] You, J., Trager, S. C., and Wilkinson, M. H. (2019). A fast, memory-efficient alpha-tree algorithm using flooding and tree size estimation. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 256–267. Springer.

[You et al., 2023] You, J., Trager, S. C., and Wilkinson, M. H. (2023). A fast alpha-tree algorithm for extreme dynamic range pixel dissimilarities. *Submitted to IEEE Transactions on Pattern Analysis and Machine Intelligence*.