A Fast, Memory-Efficient Alpha-Tree Algorithm using Flooding and Tree Size Estimation

Jiwoo You[1,2], Scott C. Trager[1], and Michael H.F. Wilkinson[2]

[1] Kapteyn Astronomical Institute, University of Groningen, Groningen, The Netherlands
{j.you,sctrager}@astro.rug.nl

[2] Bernoulli Institute of Mathematics, Computer Science and Artificial Intelligence, University of Groningen, Groningen, The Netherlands
{j.you,m.h.f.wilkinson}@rug.nl

Abstract. The α–tree represents an image as hierarchical set of α-connected components. Computation of α–trees suffers from high computational and memory requirements compared with similar component tree algorithms such as max–tree. Here we introduce a novel α–tree algorithm using 1) a flooding algorithm for computational efficiency and 2) tree size estimation (TSE) for memory efficiency. In TSE, an exponential decay model was fitted to normalized tree sizes as a function of the normalized root mean squared deviation (NRMSD) of edge-dissimilarity distributions, and the model was used to estimate the optimum memory allocation size for α–tree construction. An experiment on 1256 images shows that our algorithm runs 2.27 times faster than Ouzounis and Soille's thanks to the flooding algorithm, and TSE reduced the average memory allocation of the proposed algorithm by 40.4%, eliminating unused allocated memory by 86.0% with a negligible computational cost.

Keywords: Alpha–Tree · Mathematical Morphology · Connected Operator · Tree Size Estimation · Efficient Algorithm

1 Introduction

Connected operators are morphological filters that process input images as a set of inter-connected elementary regions called flat-zones [1, 2]. Connected operators can modify, merge or remove regions of interest without changing or moving object contours, because they filter pixels based on attributes of connected components that the pixels belong to, instead of some fixed surroundings [3]. The advent of component trees (max– and min–tree) enabled new types of filtering strategies for connected operators by providing efficient data structure to represent hierarchy of image regions [4]. In max–trees (min–trees), connected components with local maxima (minima) become leaf nodes, and those leaf nodes are successively merged to form larger connected components with lower (higher) levels, eventually becoming a root node representing the whole image [4]. This creates component-tree hierarchies of level sets, which assumes all connected components are nested around local peaks [6].

The α–tree is a more recent hierarchical image representation, where dissimilarities between local pixels define quasi-flat zones [5–8]. The α–tree has shown its usefulness on segmentation of e.g. remote sensing images [6]. Because both the pixels and dissimilarities between neighbouring pixels must be taken into account, the α–tree has multiple times more data elements to process compared to the component tree. Algorithms for component trees have been thoroughly studied and reviewed [9], which showed that flooding algorithms have lower computational cost for low-dynamic-range images than algorithms based on Tarjan's union-find algorithm [10]. On the other hand, there have only been a few studies of the α–tree algorithm, and all used algorithms based on the union-find algorithm (which will be referred as Ouzounis–Soille's algorithm throughout this paper) [6, 7, 11].

Memory requirements also play an important role in the α–tree algorithm, because a huge amount of memory is needed to store the α–tree nodes of large images. A large portion of allocated memory is wasted, because the number of α–tree nodes is usually only 40–70% of its maximum (further explained in Section 3) [7]. However, previous studies of α– and (even) component tree algorithms usually choose to allocate an oversized amount of memory, because there has been no way to estimate tree sizes before construction. In this paper we introduce a novel α–tree algorithm using 1) the flooding algorithm for computation speed and 2) tree size estimation (TSE) for memory efficiency. We show that TSE accurately estimates the tree size beforehand using the normalized root mean square deviation (NRMSD) of dissimilarity histograms, thus enabling significant reduction in memory usage at small additional computational cost.

2 A Fast, Memory-Efficient α–Tree Algorithm

2.1 Definitions

We represent an image as undirected weighted graph $G = (V, E)$, where V is the set of pixels of an image, and $E = \{\{v_i, v_j\} | i \neq j\}$ is a set of unordered neighbouring pairs (i.e. edges) of V. We define the edge weight $w : E \rightarrow Y$ of an edge $e \in E$ as a symmetric dissimilarity $d(v_i, v_j) = d(v_j, v_i)$ between pixels, where Y is the codomain of w and $W = \{w(e) | e \in E\}$ is the image of w ($W \subseteq Y$). A path $\pi(v \rightsquigarrow u)$ from $v \in V$ to $u \in V$ is a sequence of successively adjacent vertices:

$$\pi(u \rightsquigarrow v) \equiv (v = v_0, v_1, ..., v_{|\pi(u \rightsquigarrow v)|-1} = u). \tag{1}$$

with every pair $\{v_i, v_{i+1}\} \in E$. An α–connected component (α–CC) containing a vertex $v \in V$, or α–CC(v) is defined as

$$\alpha\text{–CC}(v) = \{v\} \cup \{u \mid \exists \pi(v \rightsquigarrow u) \forall (v_i \in \pi(v \rightsquigarrow u), v_i \neq u) d(v_i, v_{i+1}) \leq \alpha\}. \tag{2}$$

In words, α–CC(v) is a set of vertices containing v, where for every distant pair (v_i, v_j) of α–CC(v) their exists a path from v_i to v_j in which every successive pair has an incident edge with a weight less than or equal to α. An image can be represented as a partition of α–CCs [6]:

$$\mathbf{P}^{\alpha} = \{\alpha\text{–CC}(v) | 0 \leq v \leq |V| - 1\}, \cup_v \alpha\text{–CC}(v) = V. \tag{3}$$

For $\alpha = 0$, $\mathbf{P}^0$ forms the finest partition of flat–zones, and as α increases the partition become coarser, eventually becoming a single α–CC that encompasses the whole image ($\mathbf{P}^{\alpha_{max}} = V$). A set of all α–CCs marked by the same vertex (i.e. $\cup_\alpha \alpha$–CC(v)) form a total order:

$$\forall(\alpha_i \leq \alpha_j)\ \alpha_i\text{–CC}(v) \subseteq \alpha_j\text{–CC}(v). \tag{4}$$

We label α_j–CC(v) as redundant if there exists α_i–CC(v) such that $\alpha_i < \alpha_j$ and α_i–CC$(v) = \alpha_j$–CC(v).

2.2 Data Structure

In our algorithm, the α–tree data structure contains the following two arrays:

- *pAry*: Array of indices to leaf nodes for each pixel ($|V|$)
- *pNode*: Array of α–tree nodes ($|V|$)

Note that numbers inside vertical bars indicate array size. A typical representation of an α–tree node is to use *pNode* with size $2|V|$, where the first $|V|$ nodes correspond to leaf nodes representing a single pixel, and the rest correspond to non-leaf nodes. Here we replace single-pixel nodes with indices to parent nodes (*pAry*) to reduce memory usage. *pNode* is an array of structs containing:

- *alpha*: α value of the associated α–CC
- *parent*: An index to the parent node
- $*$*attributes*: Attributes of the associated α–CC

Since we used 8-bit images and the L_∞ norm as a dissimilarity metric, *alpha* can be stored in one byte. We used a 32-bit unsigned integer to store *parent*. The *attribute* field represents all attributes stored in a node. In our implementation we used the maximum, the minimum, and the sum of pixels inside the associated α–CC. The size of a single struct including *alpha*, *parent* and all attributes is 19 bytes.

In addition to the α–tree data structure, the α–tree flooding algorithm also requires the following temporary data structures:

- *hqueue*: Hierarchical queue in a one-dimensional array ($|E|$)
- *levelroot*: Array of parent and ancestor nodes ($|W|$)
- *dhist*: Histogram of pixel dissimilarities ($|W|$)
- *dimg*: Array of dissimilarities of all neighbouring pixel pairs ($|E|$)
- *isVisited*: Array that keeps track of visited pixels ($|V|$)

The first two arrays (*hqueue* and *levelroot*) are data structures used in the flooding algorithm which are further explained in the next section. The dissimilarity histogram (*dhist*) is used to allocate queue sizes in each hierarchy of *hqueue* and also to calculate NRMSD in TSE. Dissimilarities of all neighbouring pairs are stored in *dimg* during the computation of *dhist* to avoid computing dissimilarities of the pairs multiple times. The array $isVisited$ keeps track of visited pixels, because every pixel in the image needs only a single visit.

2.3 The α–Tree Flooding Algorithm

Flooding is a tree construction algorithm with a top-down design that constructs a tree in depth-first traverse [4]. The α–tree can be interpreted as a min-tree of a hypergraph $V' = (E, V)$ where two edges $e1$ and $e2$ in V' are neighbours if and only if there is a pixel v such that $v \in e1$ and $v \in e2$. Thus building an α–tree of V is equivalent to building a min–tree of V', and therefore any max–tree construction algorithms including the flooding algorithm can be applied to α–tree [11–13].

Our implementation of the flooding algorithm is similar to the max–tree flooding algorithm in [4], but we replace recursive calls with iterative loops as in [3], use a different neighbours-queuing process, and other optimization techniques suitable for low-dynamic-range images. Algorithm 1 shows a pseudocode of our C–implementation of the proposed α–tree algorithm (C–code available at [14, 15]). The overall design of the algorithm is similar to that of a max–tree. The main loop of the algorithm (line 10–42) creates levelroots, connects them to the α–tree, and removes redundant nodes. The inner loop (line 11–32) visits pixels in the hierarchical queue in increasing order of α, queues up neighbouring pixels, and connects the visited pixels to the levelroots. For depth-first traversal of α–tree we used a hierarchical FIFO queue [4], which is a hierarchy of queues, each of which is reserved for queuing pixels with the same α value. During the traversal, *levelroot* stores potential parent and ancestors of the current α–node being processed [3]. Nodes of the α–tree are stored in a one-dimensional array that is pre-allocated with a size estimated by TSE (explained in Section 3).

An α–tree is represented as an array of nodes, where a node is a structure storing node information (as defined in Section 2.2). An α–tree node is always associated with only a single α–CC, but an α–CC can be represented as a sub-tree of multiple nodes with the same α, where the node that has a parent with higher α is called a levelroot [3] or a canonical element [16]. In Ouzounis–Soille's algorithm a procedure called levelroot-fix or canonicalization is used as a post-processing to merge non-levelroot nodes and its levelroot parents to reduce memory usage and simplify tree structure [9]. However, since the flooding algorithm never creates non-levelroot nodes, our α–tree algorithm does not need an extra procedure to canonicalize the tree [4].

Algorithm 1 The α–tree flooding algorithm

```
void Flood(Pixel *$img$){
1       Compute dissimilarity histogram and store it in $dhist$
2       Initialize $\alpha$-node array using TSE
3       Initialize $hqueue$ using $dhist$
4       Initialize $levelroot[i]$ as $empty$ for all $i$
5       $alpha\_max$ = the maximum possible $\alpha$; // 255 for 8-bit images
6       $levelroot[alpha\_max]$ = new $\alpha$–node;
7       $current\_alpha = alpha\_max$;
8       $curSize = 0$; //current size of $\alpha$-tree
9       $hqueue.push(0, alpha\_max)$;

10      while($current\_alpha \leq alpha\_max$){
11          while($hqueue$ is not empty for all $alpha$ no higher than $current\_alpha$){
12              $p$ = front of queue;
13              $hqueue.pop()$;
14              if($p$ is $visited$){
15                  $hqueue.find\_min\_alpha()$;
16                  continue;
                }
                // visit all neighbours
17              mark $p$ as $visited$;
18              for(neighbours $q$ of $p$){
19                 if($q$ is not $visited$){
20                     $d = L_\infty(img[p], img[q])$;
21                     $hqueue.push(q, d)$
22                     if($levelroot[d]$ is $empty$)
23                          mark $levelroot[d]$ as $node\_candidate$;
                   }
                }
24              if($current\_alpha > hqueue.min\_alpha$)
25                 $current\_alpha = hqueue.min\_alpha$; //go to lower $\alpha$
26              else
27                 $hqueue.find\_min\_alpha()$; //delayed minimum $\alpha$ set
28              //connect a pixel to $levelroot$
29              if($levelroot[current\_alpha]$ is $node\_candidate$)
30                 $levelroot[current\_alpha]$ = &$pNode[curSize$++$]$ //new $\alpha$-node
31              connect $p$ to $levelroot[current\_alpha]$
32              $pAry[p] = levelroot[current\_alpha]$
            }
33          if($levelroot[current\_alpha]$ is a redundant node)
34             replace $levelroot[current\_alpha]$ with its redundant child node;
35          $next\_alpha = current\_alpha + 1$;
36          while($next\_alpha \leq alpha\_max$ && $levelroot[next\_alpha]$ is $empty$)
37             $next\_alpha = next\_alpha + 1$;
38          if($levelroot[next\_alpha]$ is $node\_candidate$)
39             $levelroot[next\_alpha]$ = &$pNode[curSize$++$]$ //new $\alpha$-node
40          connect $levelroot[current\_alpha]$ to $levelroot[next\_alpha]$
41          mark levelroot[current_alpha] as $empty$;
42          $current\_alpha = next\_alpha$;
        }
  }
```

3 The Tree Size Estimation

Both α–trees and component trees have high memory requirements. Given an image with $|V|$ pixels, the amount of memory allocated to store α–tree nodes is typically larger than $40|V|$ bytes, and it also needs temporary memory space of up to $6|V| + O(|V|)$ bytes [9]. Most algorithms use a large pre-allocated array to store tree nodes [3, 6, 7, 9], which can store up to $2|V|$ nodes. However this is rarely the case for low-dynamic large images. In [7] only 40–70% of the maximum number of nodes (i.e. $2|V|$ nodes) were created, which means a large memory space is wasted during the construction of α–tree.

There are alternative data structures that could eliminate unused memory space, such as dynamically allocated structures linked by pointers [7]. These data structures allocate each node dynamically on demand and are also more flexible compared with one-dimensional arrays, because it is easier to insert or remove nodes from the tree. However a single node takes more memory space because of the link pointer, and it runs much slower than the array implementation because of the frequent memory allocations. In [7] the dynamically allocated data structure took 48% more time than the array on average in building α–tree from images with 0.78 megapixels, and 109% more time on images with 431.3 megapixels. Memory allocation is also locking, preventing use in parallel.

Another way to deal with the problem of memory usage is to use one-dimensional arrays with a smaller array size and increase it on demand using the *realloc*() function whenever the array is full [17]. In this way the average memory usage can be reduced, but several problems arise from the memory re-allocation. If there is no memory space to extend the array in place, *realloc*() relocates the array by allocating a new array with a new size and copying the entire data from the old to the new array. Such a cumbersome reallocation procedure significantly increases computational cost, and the memory usage might even exceed the maximum memory space needed to store $2|V|$ nodes, because of the use of temporary memory space in reallocation.

The best way to reduce the excessive memory usage is to accurately estimate the tree size before building the tree. The size of the tree is generally proportional to the size of the image, but it is also heavily dependent on the image content. We have found a simple measure that can be used to accurately estimate the tree size, thus significantly reducing the memory usage at almost zero computational cost. Let the normalized tree size (NTS) be defined as follows:

$$\textit{Normalized tree size}\ (\mathrm{NTS}) = \frac{\textit{number of nodes in tree}}{2|V|}. \tag{5}$$

If the tree size is at its maximum, $NTS \approx 1$ for large $|V|$. If all dissimilarities of neighbouring pixel pairs in the image have distinct values, the α–tree built from that image will become a complete binary tree with $2|V|$ nodes, and thus NTS of that tree becomes its maximum. In such a case the pixel dissimilarity histogram should be flat, that is, $dhist[w] = 1$ for all $w \in W$. On the other extreme, the smallest tree size can be observed when all the pixels have the same intensity, and thus the entire image becomes a single α–CC. In such a case, NTS $= 1/(2|V|)$ and $dhist[w_0] = |Y|$ for some w_0 and $dhist[w] = 0$ for $w \neq w_0$, where Y is the codomain of the edge weighting as defined in Section 2.1($|Y| = |E|$). Based on these observations, we hypothesize that the NTS of an image can be estimated by the root mean square deviation (RMSD) between *dhist* and a flat distribution as follows:

$$\text{RMSD} = \frac{\sqrt{\sum_{i \in Y} (dhist[i] - 1)^2}}{|Y|}. \tag{6}$$

The RMSD is zero when the *dhist* is a flat histogram, which means all neighboring pairs have unique edge weights ($|W| = |Y|$). For low-dynamic-range images with high-resolution, $|W| << |Y|$ ($|W|$ is typically bounded by image bit-depth), thus the RMSD of those images cannot be zero. We define the normalized RMSD (NRMSD) as the RMSD divided by its maximum:

$$\text{NRMSD} = \sqrt{\frac{\sum_{i \in Y} (dhist[i] - 1)^2}{(|Y| - 1)|Y|}}, \tag{7}$$

and some rearranging yields more easily calculable form,

$$\text{NRMSD} = \sqrt{\frac{\sum_{i \in W} dhist[i]^2 - |Y|}{(|Y| - 1)|Y|}}. \tag{8}$$

which gives a normalized value between 0 and 1. The NRMSD requires little more than $|W|$ multiplications, where $|W| \leq 256$ for 8-bit greyscale images. We find that if the NTS is plotted as a function of the NRMSD, it is very close to the exponential decay function shown in Figure 1(a). Thus, we define the estimate of the NTS as an exponential decaying model as follows:

$$TSE(x) = Ae^{-\sigma x} + B \tag{9}$$

The parameters A, B, and σ are optimized using the test datasets in the next section.

4 Experiments

We use 1256 pictures including 639 greyscale aerial pictures selected from the database of the Netherlands Institute of Military History [18], with sizes ranging from 2430×3500 to 3289×3500 (*Holland* dataset). The other pictures include 617 more diverse colour images (*General* dataset), including pictures of natural scenery, artificial structures, maps, arts, and synthetic images with size ranging from 366×550 to 13394×10119. Images from the *General* dataset were also converted to greyscale images, and thus we have 1256 greyscale images and 617 colour images overall. The experiment was conducted on an Intel i7–6700HQ processor with 16GB memory.

Table 1 shows processing speed and memory usage of the proposed and the conventional α–tree algorithm by Ouzounis and Soille [6]. The experiments were conducted on the full 1256 greyscale dataset, which includes the 639 *Holland* dataset images and 617 *General* dataset (greyscale–converted), and the 617 colour dataset. The proposed algorithm runs 2.27 (2.84) times faster than Ouzounis–Soille's for greyscale (colour) images. There is more recent study on the fast single-thread α–tree algorithm by Havel *et al.* which achieved 5.56Mpixel/s on colour images using different hardware (i7–2670QM with 8GB memory) and a different dataset [7]. Full comparison of the processing speed and the memory usage between the proposed algorithm and the Havel's algorithm should be conducted in the future study.

We applied TSE to both flooding and Ouzounis–Soille's algorithm, and TSE reduced the memory usage of flooding and Ouzounis–Soille's algorithm by 40.4% and 14.1%, while decreasing the processing speed by only 1.7% and 0.3%, respectively. We have not implemented TSE on colour α–tree algorithms here because dissimilarities between colour pixels have higher dynamic range than their greylevel counterparts, which is out of the scope of this paper. Here we used L_∞ dissimilarity for colour images to reduce the dynamic range of dissimilarities. We found that the exponential decay modeling of tree sizes does not work on colour image α–trees with L_∞ dissimilarities, possibly because of the loss of information in the calculation of dissimilarity.

Table 1. The processing speed and the memory usage of the proposed and the previous α–tree algorithm. The proposed algorithm runs 2.27 (2.84) times faster than Ouzounis–Soille's in greyscale (colour) images. Tree size estimation (TSE) reduced memory usage of flooding and Ouzounis–Soille's algorithm by 40.4% and 14.1%, respectively.

Image dataset	α–tree algorithm	Processing speed (Mpixel/s)	Memory usage (byte/pixel)
1256 greyscale images (*Holland* + *General*)	Flooding	14.00	62.12
	*Flooding + TSE	13.76	37.00
	Ouzounis-Sollie	6.18	90.00
	Ouzounis-Soille + TSE	6.16	77.31
617 colour images (*General*)	*Flooding	6.72	110.12
	Ouzounis-Sollie	2.37	122.00

*proposed

4.1 TSE Performance

We compare the time and memory usage of TSE for different memory management methods. We model the NTS as an exponential decay function of the NRMSD as in (8), and optimize parameters using the full dataset. The optimal values were $A = 1.3901$, $B = -0.1906$ and $\sigma = 2.1989$. The upper bound of tree sizes is computed as

$$\text{TS}_{max}(\text{NRMSD}) = 2|V|(\text{TSE}(\text{NRMSD}) + M), \tag{10}$$

with the estimate of TSE as in (9), and M a constant determining the upper bound.

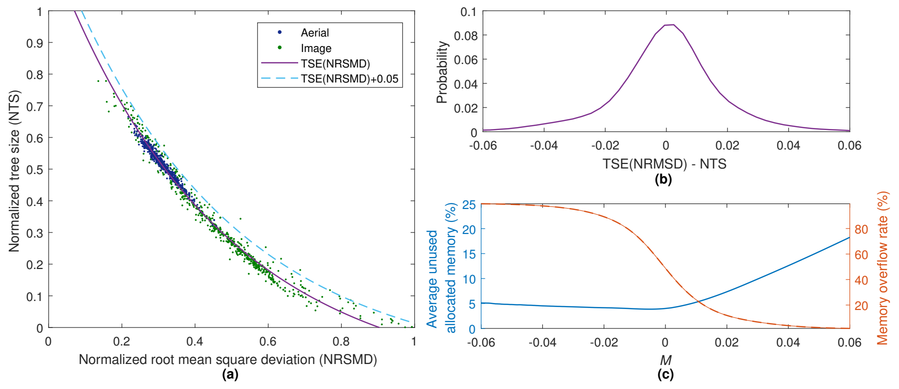

Fig. 1. **(a)** The exponential decay modeling of the normalized tree size (NTS) as a function of the normalized root mean square deviation (NRMSD).The solid line shows the least square model fit, and the dashed line shows the upper bound of NTS used as memory allocation sizes.**(b)** The probability mass function of TSE estimate error from leave-p-out cross validation with $p = 1244$. **(c)** The rate of unused allocated memory and memory overflow as a function of M.

Figure 1(a) shows the NTS as a function of the NRMSD using the test images. The solid line shows the least square fit of the exponential decay model, and the dashed line shows the upper bound of the NTS, which is used as the memory allocation size of TSE for a given NRSMD (which should be multiplied by $2|V|$ as in Eq. 10). The *General* dataset is more widely distributed than the *Holland* dataset, especially near the tail of the model, because the former contains more images with large (quasi–)flat zones, which results in high NRSMD. Figure 1(b) shows the probability mass function of TSE estimate error from leave-p-out cross validation (CV) with $p = 1244$. In the CV 12 randomly sampled images from the dataset are used to fit the TSE model, and the remaining 1244 images are used to validate the model. The training and validating are repeated 1000 times.

Despite the small size of training set, we find that approximately 80% of TSE estimates differs from the true NTS by no more than 0.02. Figure 1(c) shows the rate of unused allocated memory and memory overflow, as a function of M. As M increases, the memory overflow rate decreases, but too-high M will lead to a high unused allocated memory rate. We choose $M = 0.05$ to minimize unused allocated memory rate with zero memory overflow.

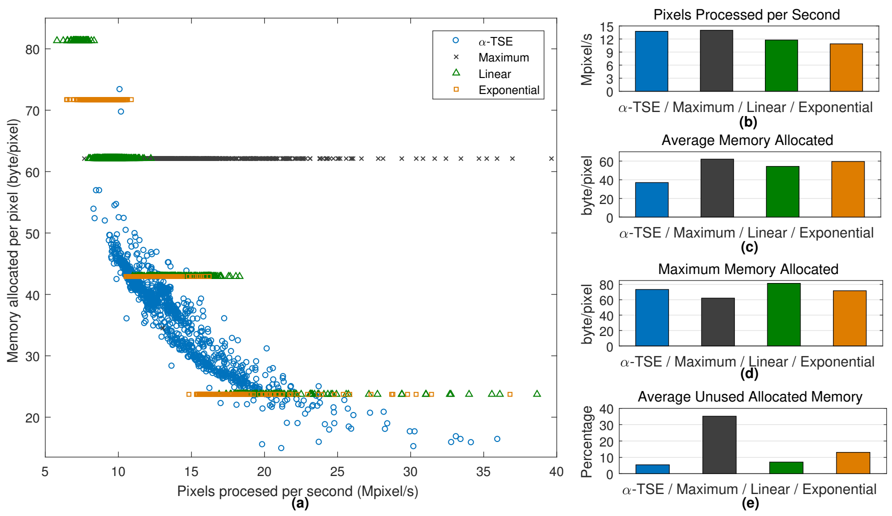

Fig. 2. (a) Execution speed and memory usage of TSE, *Maximum*, *Linear*, and *Exponential*. TSE clearly allocates adaptive memory sizes for different images. Bar plots on the right side show **(b)** pixels processed per second, **(c)** the average memory allocated, **(d)** the maximum memory allocated, and **(e)** the average unused allocated memory.

We analyze the computational and memory costs of TSE and other conventional memory management methods to verify the performance of TSE. Conventional methods typically allocate the maximum amount of memory for $2|V|$ nodes (*Maximum*). Alternatively we could either first allocate a small initial memory space of nodes and increase it linearly every time the memory overflows (*Linear*), or exponentially by multiplying it (*Exponential*). In our implementation, both *Linear* and *Exponential* use initial space of $0.4|V|$; increment in *Linear* is $0.4|V|$ and the multiplication factor in *Exponential* is 2. All methods use clipping to prevent them from allocating memory for more than $2|V|$ nodes.
Figure 2(a) shows the execution speed and memory usage of TSE, *Maximum*, *Linear*, and *Exponential*. To compute memory allocated per pixel, we count all of the memory space of data structures used to build the α–tree defined in Section 2.2. This figure shows that TSE allocates memory sizes adapted to the image sizes. The right column of the figure shows pixels processed per second (Fig. 2(b)), the average memory allocated (Fig. 2(c)), the maximum memory

allocated (Fig. 2(d)), and the average unused allocated memory (Fig. 2(e)). For TSE, these measures are 14.00 Mpixel/s, 37.00 bytes/pixel, 73.36 bytes/pixel and 5.4%, respectively. Compared to *Maximum*, TSE reduces the average memory (−40.4%) and average unused allocated memory (−84.6%) with a negligible decrease in the execution speed (−1.7%). The maximum memory is increased by 18.1%, but this is due to a couple of outlying images (on the top side of Fig. 2(a)). *Linear* and *Exponential* also show decreases in the average memory (−12.5% and −4.1%) and the unused allocated memory (−79.9% and −62.9%), but they also show a significant increase in the maximum memory allocated (+30.9% and +15.5%) and a decrease in the execution speed (−12.7% and −12.9%).
We find a high correlation (0.943) between NRSMD and the processing speed; this is because, as NRSMD increases, the number of nodes to be processed decreases, which leads to an increase in the processing speed. An accurate estimation of the processing speed can be useful in the parallel construction of an α–tree in future works.

5 Conclusion

We have introduced a novel α–tree algorithm using a flooding algorithm and TSE. We have modified and optimized a flooding algorithm for α–tree construction and find that our α–tree flooding algorithm runs 2.27 times faster than the conventional Ouzounis–Soille algorithm. Our algorithm also significantly reduces the memory usage using TSE: TSE reduced the average allocated memory by 40.4% and reduced unused allocated memory from 35.2% to 5.4%, with a negligible execution speed reduction (−1.7%). In future work, the new α–tree algorithm can be applied to existing parallel algorithms to further increase the speed or in a hybrid parallel α–tree algorithm for high-dynamic-range images using pilot tree as in [19], where TSE can be used to estimate computation and memory requirement of pilot tree nodes to maximize load balance. We can also apply TSE to component trees and investigate if we can obtain similar results as in α–trees.

References

1. Salembier, P. and Serra, J.: Flat zones filtering, connected operators, and filters by reconstruction. IEEE Transactions on Image Processing **7**(4), 1153–1160 (1995)
2. Salembier, P., and Wilkinson, M.H.F.: Connected operators: A review of region-based morphological image processing techniques. IEEE Signal Processing Magazine **26**(6), 136–157 (2009)
3. Wilkinson, M.H.F.: A fast component-tree algorithm for high dynamic-range images and second generation connectivity. In: 2011 18th IEEE International Conference on Image Processing (ICIP), pp. 1021–1024. Brussels, Belgium (2011).
4. Salembier, P., Oliveras, A, and Garido, L.: Antiextensive connected operators for image and sequence processing. IEEE Transactions on Image Processing **7**(4), 555–570 (1998)

5. Soille, P.: Constrained connectivity for hierarchical image partitioning and simplification. IEEE transactions on pattern analysis and machine intelligence **30**(7), 1132–1145 (2008)
6. Ouzounis, G.K., and Soille, P.: The alpha-tree algorithm, theory, algorithms, and applications. JRC Technical Reports, Joint Research Centre, European Commission (2012)
7. Havel, J., Merciol F., and Lefèvre S.: Efficient tree construction for multiscale image representation and processing. Journal of Real-Time Image Processing **2016**, 1–18, (2016)
8. Merciol, F., Lefèvre, S.: Fast image and video segmentation based on α–tree multiscale representation. In: International Conference on Signal Image Technology Internet Systems. pp. 336–342. Naples, Italy (November 2012)
9. Carlinet, E., and Géraud, T.: A comparative review of component tree computation algorithms. IEEE Transactions on Image Processing **23**(9), 3885–3895 (2014)
10. Berger, C., Géraud, T., Levillain, R., Widynski, N., Baillard, A. and Bertin, E.: Effective component tree computation with application to pattern recognition in astronomical imaging. In: IEEE International Conference on Image Processing (ICIP). vol. 4, pp. 41–44. San Antonio, TX, USA (September 2007)
11. Najman, L., Cousty, J., and Perret, B.: Playing with Kruskal: algorithms for morphological trees in edge-weighted graphs. In: International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing, pp. 135–146. Springer, Berlin, Heidelberg (2013)
12. Najman, L.: On the equivalence between hierarchical segmentations and ultrametric watersheds. Journal of Mathematical Imaging and Vision **40**(3), 231–247 (2011)
13. Soille, P., Najman, L.: On morphological hierarchical representations for image processing and spatial data clustering. In: International Workshop on Applications of Discrete Geometry and Mathematical Morphology, pp. 43–67. Springer, Berlin, Heidelberg (2010)
14. You, J., Alpha–tree algorithm for greyscale images. GitHub repository, https://github.com/jwRyu/AlphaTreeGrey (2019)
15. You, J., Alpha–tree algorithm for 3–channel colour images. GitHub repository, https://github.com/jwRyu/AlphaTreeRGB (2019)
16. Najman, L., and Couprie, M., Building the component tree in quasi-linear time. IEEE Transactions on Image Processing **15**(11), 3531–3539 (2006)
17. International Organization for Standardization, ISO/IEC 9899:TC3: Programming Languages — C. September 2007
18. Nederlands Instituut voor Militaire Historie, https://www.flickr.com/people/nimhimages. Last accessed 29 March 2019
19. Moschini, U., Meijster, A., and Wilkinson, M.H.F.: 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 (2018)