A Fast Alpha-tree Algorithm for Extreme Dynamic Range Pixel Dissimilarities

Ryu, Jiwoo; Trager, Scott; Wilkinson, M.H.F.

A Fast Alpha-tree Algorithm for Extreme Dynamic Range Pixel Dissimilarities

Jiwoo Ryu, Scott C. Trager, and Michael H.F. Wilkinson *Senior Member, IEEE*

Abstract—The α-tree algorithm is a useful hierarchical representation technique which facilitates comprehension of images such as remote sensing and medical images. Most α-tree algorithms make use of priority queues to process image edges in a correct order, but because traditional priority queues are inefficient in α-tree algorithms using extreme-dynamic-range pixel dissimilarities, they run slower compared with other related algorithms such as component tree. In this paper, we propose a novel hierarchical heap priority queue algorithm that can process α-tree edges much more efficiently than other state-of-the-art priority queues. Experimental results using 48-bit Sentinel-2A remotely sensed images and randomly generated images have shown that the proposed hierarchical heap priority queue improved the timings of the flooding α-tree algorithm by replacing the heap priority queue with the proposed queue: 1.68 times in 4-N and 2.41 times in 8-N on Sentinel-2A images, and 2.56 times and 4.43 times on randomly generated images.

Index Terms—alpha-tree, efficient priority queue, redundant node redu, high-dynamic-range image, multi-spectral image

I. INTRODUCTION

UNDERSTANDING an image requires recognizing regions and objects, based on features such as sizes, brightness, and frequency on different scales. A hierarchical image representation provides an efficient way to achieve this goal by segmenting the image from finer to coarser scales [1]–[5]. Many of those applications use tree data structures to construct the hierarchical representations, because the tree data structure provides a natural and an efficient way to represent a hierarchy of image partitions at different scales [6]–[9].

Alpha-trees are used in applications such as built-up urban area detection, image/video segmentation, human anatomical image segmentation, traffic sign recognition, or plant disease identification [6], [7], [9], [10]. Segmentation, detection or classification using α-trees can be done by filtering the α-tree nodes using either hand-coded rules [6], or machine learning [10]. Deep neural networks (DNNs) could be trained on α-tree nodes just as in superpixel-based DNN algorithms [11], [12], although we know of no published work on this topic. The alpha-tree algorithm partitions an image by merging areas with homogeneous gray levels or color, i.e. flat zones. A flat-zone is a connected component (CC) where neighboring pixels in it have low dissimilarities. The choice of the pixel dissimilarity measure depends on the application and the computational

J. Ryu and M.H.F. Wilkinson are with the Bernoulli Institute for Mathematics, Computer Science, and Artificial Intelligence, University of Groningen. J. Ryu and S.C. Trager are with the Kapteyn Astronomical Institute, University of Groningen.
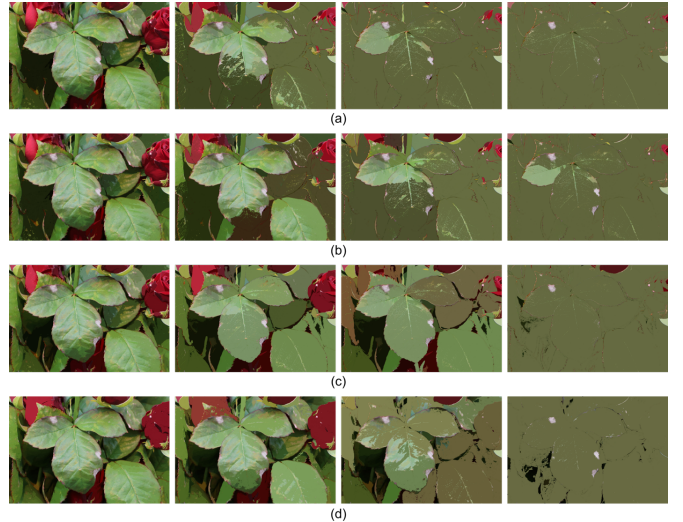
Fig. 1. Illustration of α-tree representations built from an image of a rose plant, using different pixel dissimilarity metrics of (a) L^∞ norm, (b) L^2 norm, (c) L^2 norm in CIELAB color space and (d) modified L^2 norm in HSV color space.

complexity. Previous studies used the L^2 or L^∞ norms in gray scale or RGB color space [6], [7], [9]. It is also possible to use other color spaces such as HSV, CIELAB, or to use hyper-spectral images [13], [14], all of which lead to extreme dynamic ranges in the dissimilarity measure.

Figure 1 shows α-tree scale spaces from finer (left) to coarser (right) scale, using (a) L^∞ norm in RGB color space, (b) L^2 in RGB, (c) L^2 in CIELAB, and (d) L^2 in HSV. L*, a* and b* in CIELAB and H, S and V in HSV are normalized before computing L^2 norm. Note that the dissimilarity in H is computed in a circular fashion, requiring a minor modification to the L^2 norm. Salient objects such as red roses and white fungal leaf spots are preserved better in coarse scales in CIELAB or HSV, making their identification easier. Using higher dynamic range dissimilarity using L^2 norm instead of L^∞ also provides much more information in resulting α-tree. The four images shown in each of Fig. 1(b), (c) and (d) are sampled from millions of scales in their practically continuous scale spaces, while the four images in Fig. 1(a) are the only images in this range in a scale space with only few hundred scales. Silla compared RGB, CIELAB and HSV color spaces in an α-tree application for disease detection on plants, and found that using L^2 norm in HSV color space provides the best segmentation [10].

Use of higher-dynamic-range pixel-dissimilarity measures

significantly improves the performance of α-tree, but it comes with a high computational cost. The dynamic range of pixel dissimilarities tends to become very high when dealing with color or multichannel images. Even if each channel of a color image is in low-dynamic-range (LDR) or high-dynamic-range (HDR), its pixel dissimilarities tend to be in extreme-dynamic-range (XDR), unless L^∞ norm is used or dissimilarity values are quantized.[1] It is also possible to have pixel dissimilarities in XDR on grayscale images in LDR. Zhang and Wilkinson [15] introduced a 2-D Gabor filter to smooth out edges along the direction of edge strength to reduce the chaining effect, resulting in higher-dynamic-range pixel dissimilarities. However, the α-tree algorithm has not made much progress since [6], and to the best of our knowledge, there has been no published study on the α-tree algorithm that works on XDR images. In [6] an algorithm based on Berger's union-find algorithm using a hierarchical queue was proposed [16], which works only on LDR or HDR pixel dissimilarities, due to the use of the hierarchical queue. In [17] a parallel α-tree algorithm based on the union-find algorithm was proposed, which also works only on LDR or HDR because the computational cost of merging subtrees of sub-image blocks increases exponentially as the bit depth increases. In [18] an application of Kruskal algorithm to build α-tree was proposed, which is essentially a union-find-like algorithm which uses union by rank and path compression to keep the tree depth low. A new approach using homological-based tools was introduced in [19] which proposed a parallel algorithm to build α-trees in 8-bit images. Our previous work on α-tree algorithm used flooding algorithm and hierarchical queue for LDR images, where the size of the α-tree was estimated in prior to save the memory footprint of α-trees [20].

Flooding (i.e. merge-based algorithm) is another type of α-tree algorithm based on the original flooding algorithm for max-tree construction [21]. Whereas union-find algorithms construct the α-tree by merging neighboring CCs in the order of increasing pixel dissimilarity, the flooding algorithm constructs a CC at certain area by "flooding" the area at certain level before moving on to another CC. Whereas union-find algorithms process edges in an increasing order but constructs CCs at seemingly arbitrary positions in an image, the flooding algorithm newly created CCs are always adjacent to CCs that already has been constructed but does not guarantee edges to be processed in an increasing order.

Redundant edges (REs) are the biggest challenge in developing an efficient α-tree algorithm. REs are image edges that create redundant and residual nodes in α-trees, which are α-tree nodes that carry the same information as their child node [6], [17]. They cause the major computational bottleneck in most α-tree algorithms because more than approximately 75% of edges are either redundant in α-tree construction [6]. Because of REs, most conventional α-tree algorithms using union-find or similar approach suffer a significant penalty on computation cost since detection and removal of REs requires calling FINDROOT, which is $O(log|E|)$, to the best of our knowledge.

[1] Here we define bit depths lower or equal to 8-bit as LDR, higher than 8-bit and lower or equal to 16-bit as HDR and higher than 16-bit as XDR.

It is possible to significantly alleviate the slowdown caused by those edges by using a flooding algorithm using hierarchical queue, because computational costs of hierarchical queue operations are all O(1) [21]. However, since the number of queues in the hierarchical queue is equal to the dynamic range of the image, hierarchical queue becomes simply infeasible in HDR and XDR since it needs millions if not billions of queues and each deletion operation requires a linear scan of those queues. Thus, current state-of-the-art algorithms suffer significant inefficiency in XDR, and this problem becomes even worse if higher pixel connectivity is used. For example, in 8-pixel connectivity the number of edges is doubled compared with 4-pixel connectivity, and hence the number of redundant edges increases by three times.

The better design of the priority queue is the key to improve the processing speed of the α-tree algorithm in XDR, because as shown in Sec. III priority queuing is the major bottleneck of the timing of alpha-tree algorithms in XDR. The priority queue design is a well-studied subject in pending event set (PES) in discrete event simulation (DES) [22]–[26]. The ladder queue (LQ) is one of the best performing priority queue in PES which has $O(1)$ enqueue (push) and $O(1)$ amortized dequeue (pop) computational complexity by using a hierarchy of queues each of which has dynamically set range and is only sorted on demand [25], [26]. The structure of the LQ consist of the top, the middle and the bottom rung where the top and the middle rungs store items as unsorted arrays and the bottom rung stores the only sorted array in the whole queue. The ladder queue has some similarities with the hierarchical heap queue (HHQ) we propose in this paper, such as the layered structure and sorting of items only when it is necessary. However, the LQ and the HHQ have some significant differences in their structures and target applications. The LQ is not designed to store a large number of items; the maximum number of rungs set by [25] and [26] is from 2 to 5, which may store up to a few tens of thousand items. The low number of rungs is not only appropriate for handling PES where the number of enqueues and dequeues are expected to be comparable, but also necessary for low computation cost as enqueuing an item to an LQ requires comparing the timestamp of the item to each rung. This is very different from the priority queue in alpha-tree algorithms, where up to millions or even billions of edges can be stored in the priority queue. The purpose of LQ design is to adopt to dataset with varying probability distributions, which is not necessary in alpha-tree algorithms where the majority of dissimilarity metrics have exponential or similar right-skewed distribution. Even if the assumption of dissimilarity distribution were to be wrong, it is easy to pre-determine the optimum layer structure by inspecting the image before the alpha-tree construction. However, we found empirically that assuming exponential distribution always outperforms using the best fit estimate in the HHQ.

In this study, we improve the α-tree algorithm by proposing a novel HHQ algorithm that can handle REs very efficiently. The proposed HHQ uses a hierarchy of queues that are sorted only on demand. We also propose the priority queue cache that not only handles high priority edges at very low cost, but also allowing the HHQ to have a large number of levels

by reducing its level inspection cost dramatically. Empirical complexity analysis shows that both push and pop operation of the HHQ are O(1) amortized. The comparative test using the proposed HHQ, hierarchical queue [21], heap queue [27] and trie queue [28] shows the HHQ significantly improves the α-tree algorithm timing in XDR by reducing the computation cost of REs. This paper is organized as follows: Section II introduces a novel hierarchical heap priority queue and the flooding α-tree algorithm using the HHQ, and Section III shows runtimes, analysis and profiling results of the proposed and other α-tree algorithms.

II. METHOD

A. Definitions

We briefly summarize the definition of the α-tree and related terms here. We define terms pertaining to an image as follows:

- $G = (V, E, w)$: the image represented as an undirected graph.
- V: the set of pixels.
- E: the set of edges.
- C: the pixel connectivity.
- w: the edge-weight mapping ($E \rightarrow \mathbb{R}$).
- W: the output set of w.

We refer to an edge-weight $w(e)$ of an edge $e \in E$ as an "α value" of e. In many α-tree flooding algorithms, $w(e)$ can also be directly interpreted as a *level* of e, which acts as a loop control variable in flooding algorithms: here we use those terms interchangeably depending on the context. As we deal with only single channel images in this paper, the pixel connectivity can be only either 4-neighbor (4-N) or 8-neighbor (8-N).

An α-tree of an image G is a tree data structure where nodes correspond to sets of connected pixels in the image (i.e., CCs). The inner nodes of an α-tree are formed by creating unions of leaf nodes and/or other inner nodes, connected each other by edges. Because the minimum spanning binary tree has only $|V| - 1$ nodes while the number of edges is $|E| = 0.5C|V|$, there are always edges that are redundant, which are not used in the α-tree construction [6]. We categorize α-tree edges into following types:

- Redundant edge (RE) : an edge connecting nodes that are already connected by another edges at lower levels [6].
- Residual edge: an RE that has the root node of an α-tree as its descendant.
- Non-redundant edge (non-RE) : an edge that is not redundant.

Figure 2 shows an example of α-tree construction from a 2×3 image, with REs and residual edges. Fig. 2(a) and (b) show an 2×3 image with its pixel values and edge-weights (i.e., α values), and the graph $G = (V, E, w)$ representation of the image, respectively. Fig. 2(c) shows an α-tree constructed from the graph in Fig. 2(b), where there is an RE e_0 and a residual edge e_4. A root node of a subtree where all nodes have α values below or equal to α_r is called root at level α_r. Node e_6 and node e_5 in Fig. 2(c) are root at level 5 and 2, respectively.
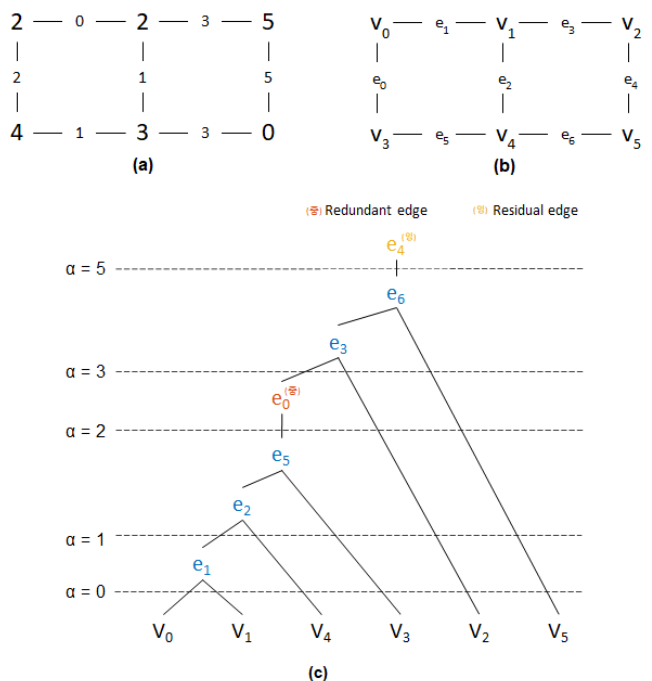

Fig. 2. An α-tree built from a 2×3 image: (a) a 2×3 image with 4-N and l^2 norm, (b) a graph $G = (V, E, w)$ representation of (a), and (c) an α-tree built from (b).

B. Priority Queues in Alpha-Tree Algorithms

Priority queues are an essential part of any α-tree flooding algorithm. They are used to queue image edges in ascending order of edge-weights, and they have a variety of data structures including hierarchical queue, heap queue and trie queue [21], [27], [28]. Hierarchical queues work very well in α-tree flooding algorithms for LDR and HDR pixel dissimilarities [20], but for XDR traditional priority queues are an inefficient solution for α-tree flooding. This is because, as we show in this section, approximately 85–90% of image edges in α-tree flooding algorithms are handled inefficiently, which could benefit from our optimized queuing strategy.

To investigate how the priority queuing works in α-tree flooding algorithms, we ran the flooding algorithm on a randomly generated image and recorded following information from each edge:

- Queue time: the number of items removed from a priority queue while an item is waiting in the queue.
- $Qpos_{max}(e)$: the maximum value of $Qpos(e)$, where $Qpos(e)$ is the position of e is the queue. For example, $Qpos(e) = 0$ when e has the highest priority in the queue.
- $rank_e$: the α value rank of an edge e in E.

Here a rank of an edge or an item $rank_e$ is its position in the hierarchy when all edges in E are sorted in ascending order. In addition to the above, we also obtained histograms of non-redundant, redundant, and residual edges.

Figure 3 shows priority queue statistics obtained from α-trees built from single-channel 64-bit randomly generated 500×500 images. High-priority edges (i.e. edges with weight ranks close to zero) have both very short queue times and very low $Qpos_{max}(e)$)s in both connectivities. We put an arbitrary
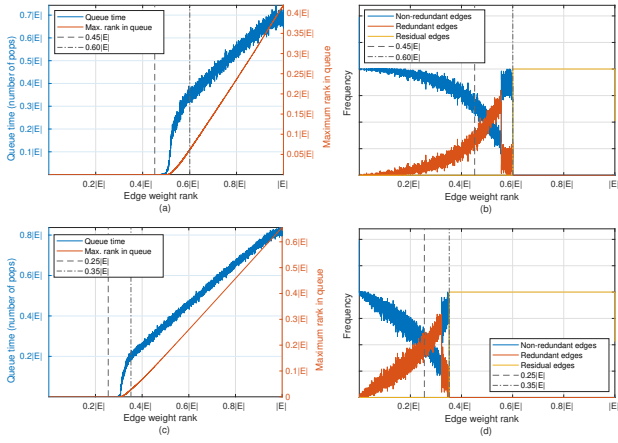
Fig. 3. Priority queue statistics obtained from α-trees of single-channel 64-bit randomly generated 500×500 images. Top panels ((a) and (b)) show the priority queue statistics obtained from an α-tree using 4-N, and bottom panels ((c) and (d)) show those obtained from an α-tree using 8-N. Left panels ((a) and (c)) show mean queue times and $Qpos_{max}(e)$)s for each edge, and right panels ((b) and (d)) show histograms of non-redundant, redundant, and residual edges. The number of edges ($|E|$) in 8-N is about twice that as in 4-N.

threshold r_0 in all panels in Fig. 3 at points where $Qpos_{max}(e)$) becomes higher than 12, which corresponds to $0.45|E|$ in 4-N and $0.25|E|$ in 8-N. The histograms in Fig. 3(b) and (d) show that most of those high-priority edges are non-redundant, meaning that they are the majority of edges that correspond to inner nodes in an α-tree. This explains why r_0 is higher in 4-N, as the number of inner nodes of a minimum spanning tree is $|V| - 1 \approx 0.5|E|$ in 4-N and $|V| - 1 \approx 0.25|E|$ in 8-N. Because most of those high-priority edges are removed from the priority queue as soon as they are inserted, it is inefficient to pay computational costs to process them in the priority queue.

Low-ranking edges, on the other hand, have high queue times and high $Qpos_{max}(e)$). The higher queue times the edges have, the more they have negative effects on computational costs of priority queue operations, by increasing the number of items in priority queues. The problem in α-tree flooding algorithms is that most of those low ranking edges are residual edges. They are inserted into a priority queue and stay there until the α-tree construction is complete. This has a critical effect on computational costs for priority queues with non-constant time operations such as heap queues. We put thresholds r_1 at points where the residual edge distributions start. As expected, $r_1 = 0.35|E|$ in 8-N is less than $r_1 = 0.60|E|$ in 4-N, because there are more edges in 8-N while the number of edges to build α-tree inner nodes remains the same ($|V| - 1$).

The only edges that are actually need priority queuing are edges within the range r_0–r_1, which is only about 10–15% of all edges. This is not a problem in LDR or HDR, because hierarchical queues can process insertions and deletions in constant time. However, this is not the case in XDR, because hierarchical queues are not available in that range. Therefore, better ways to process edges with weight ranks below r_0 and above r_1 are necessary to improve efficiencies of α-tree algorithms for XDR images. The simplest solution would be

to make separate queues for $rank_e < r_0$, $r_0 \leq rank_e \leq r_1$ and $rank_e > r_1$, but this is not a feasible because $rank_e$ is usually not available unless the edges are sorted. Moreover, r_0 and r_1 are also unknown before building an α-tree. Thus, to improve the priority queuing in α-tree, we need a novel priority queue algorithm that processes only edges in range r_0–r_1, without having to sort edges and without knowing values of r_0 and r_1 *a priori*.

C. The Priority Queue Cache

In this section we investigate how to deal with high-priority edges in priority queues. As mentioned in the previous section, high-priority edges are edges with low α-values that are removed from a priority queue very soon after they are inserted into the queue. We propose a priority queue cache that stores a small number of the highest-ranking edges in a small array, to bypass non-constant time insertion and deletion operations of traditional comparison-based priority queue operations.
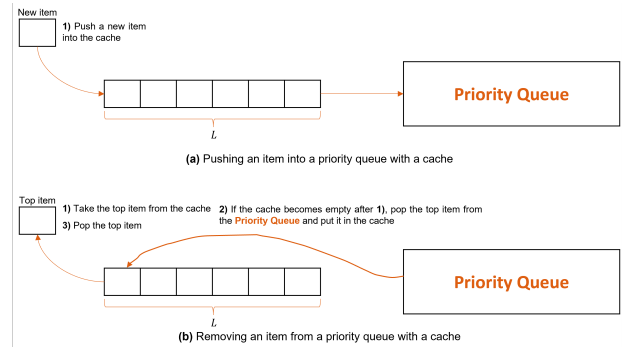

Fig. 4. Priority queue operations with a cache of size L; (a) queue insertion and (b) queue deletion.

Figure 4 shows how (a) insertion and (b) deletion work in a priority queue with a cache of size L. The cache is essentially a priority queue of a very small size which always has a priority over the main priority queue in both operations. The insertion always tries to put a new item first into a cache, and an item is only pushed into the main queue when the cache is full. Likewise, the deletion always removes an item from the cache first, and an item from the queue is only removed when the cache is empty. The size L should be a very small number to keep the cache insertion and deletion cost low, but large enough to store all high-priority edges. We empirically found $L = 12$ to be reasonable in most cases.

By using the cache in a priority queue, high-priority edges can be processed quickly because they are processed only by the cache, not the queue. This is a huge advantage especially when queue operations are costly, and/or there are many high-priority edges such as in 4-N. Although the cache can adopt any priority queue data structures, we found that a simple sorted array is the most efficient.

D. The Hierarchical Heap Priority Queue

In this section we propose a novel hierarchical heap priority queue which optimizes priority queuing in α-tree flooding algorithms in three ways: 1) by reducing the time complexity

of priority queue insertions of residual edges, by using unsorted arrays for low priority edges, 2) reducing time complexities of insertion and deletion of high-priority edges, by using a priority queue cache and 3) by keeping depths of heap queues small by using multiple quaternary heap queues [29] instead of a single binary tree. The data structure of the HHQ is similar to that of the hierarchical queue and the heap queue, and it has advantages of both queues: it has low computational costs for insertion and deletions as in hierarchical queues, and it can process XDR inputs as in heap queues.

The HHQ has a hierarchical array of queues sorted in ascending order of their *qlevel*s. Items in the HHQ are stored in one of those queues according to their α values. This is similar to hierarchical queues [21], but in hierarchical queues α values are used directly as queue indices while in HHQs we take the logarithm of α values to compute *qlevel*s:

$$qlevel := \lfloor d\log_2(\alpha+1) \rfloor \tag{1}$$

The parameter d controls the number of levels of queues. The advantage taking the logarithm of α is that we can keep the number of queues feasible even if the dynamic range of α is high, thereby removing one of the biggest disadvantage of the hierarchical queue. However, items with different α values can be stored in the same queue since different α values can be floored to the same *qlevel*. We sort items at each level using quaternary heap queues, but we sort only *qlevel*s with high priorities (*qlevel* < *qthr*), and those queues at low-priority *qlevel*s are left unsorted. This is because in an α-tree algorithm, low-priority items in the queue are most likely become residual nodes which we do not need to remove from the queue.

Figure 5 depicts how priority queue operations in an HHQ work. Fig. 5(a) shows an HHQ storing 15 items at six different *qlevel*s. Numbers on each item represent their α values which are used to compute their *qlevel*s. In this example *qthr* = 4, which means queues are sorted as quaternary heap queue only if their *qlevel* is less than 4. Fig. 5(b) shows the insertion of an item with $\alpha = 11$ into the queue in Fig. 5(a). The item is inserted into a sorted queue at *qlevel* = 3 since *qlevel* = $\lfloor d\log_2(\alpha+1) \rfloor$ = 3 from Eq. 1 ($d = 1$ in this example). Fig. 5(c) shows the queue after inserting an item with $\alpha = 41$. This item is inserted into the unsorted array at *qlevel* = 5. Fig. 5(d) shows the removal of a top item from the queue. The quaternary heap queue at *qlevel* = 0 becomes empty as a result, thus *top_qlevel* moves to the next non-empty *qlevel*, which is 1. Removing eight further items leaves only one item with $\alpha = 8$ in sorted queues as shown in Fig. 5(e). Fig. 5(f) shows what happens when the last item from the sorted queues is removed. Since the next *top_qlevel* at *qlevel* = 4 is unsorted array, it needs to be transformed into a sorted queue, increasing *qthr* by 1.

The computational cost of heap queue operations is logarithmically proportional to the maximum size of the queue $|E|$. By dividing the heap queue into an array of heap queues with smaller sizes, we reduce the computational cost by a factor of $\log_4(d\log_2(|W|+1))$ on average, where $d\log_2(|W|+1)$ is equivalent to the number of *qlevel*s. The size of each heap queue is predetermined by computing a histogram of the

qlevel values when the α values are computed. Because the histogram is usually non-uniform, heap queues in the HHQ have different sizes and different computational costs. We have tried to equalize the computational costs by equalizing the histogram, but this has not improved the overall timing of the algorithm. This is possibly because the histogram equalization does not make the queue sizes uniform due to the large number of ties in α values and/or a non-uniform distribution of queue sizes could be more beneficial than the uniform one because of the use of unsorted arrays.

Residual edges in Fig. 3 are stored in unsorted arrays in the HHQs. Most of those unsorted arrays remain unsorted until the α-tree construction is complete, because residual edges have higher α values than that of the root node of the α-tree. Therefore, the computational costs of queuing for residual edges are constant in insertion and essentially zero in deletion. Considering the fact that the number of residual edges is $0.40|E|$ in 4-N and $0.65|E|$ in 8-N from Fig. 3, reducing computational complexities of queuing for residual edges can significantly improve timings of HHQ operations.

E. Flooding Alpha-tree Algorithm Using the Hierarchical Heap Queue

---
Algorithm 1 Flooding α-tree construction algorithm

1: **procedure** FLOOD($V,E,w,|W|,C$)
2:     Compute $w(E)$ and the difference histogram *dhist*
3:     *node* := new AlphaTreeNode[$|V|+|E|$]
4:     *queue* := new HierHeapQueue(*dhist*,C,$|W|$)
5:     *curlevel* := MAX($w(E)$)
6:     Push 0 to *queue* at *curlevel*
7:     *stacktop* := 0
8:     Create a new node *node*[0] at *curlevel*
9:     **while** *node*[*stacktop*].*area* < $|V|$ **do**
10:         **while** *queue.minlevel* $\leq$ *curlevel* **do**
11:             p := *queue*.POP()
12:             **if** p is visited **then**
13:                 continue
14:             **end if**
15:             Visit p
16:             Push neighbors of p to *queue* at $w(p,q)$
17:             **if** *curlevel* > *queue.minlevel* **then**
18:                 Create a new node *node*[*newtop*] at *curlevel*
19:                 Set *node*[*stacktop*] as a parent of *node*[*newtop*]
20:                 *stacktop* := *newtop*
21:             **else**
22:                 Create a singleton node if *curlevel* = 0
23:                 Add the pixel p to *node*[*stacktop*]
24:             **end if**
25:         **end while**
26:         *newtop* := *node*[*stacktop*].*parent*
27:         **if** *queue.minlevel* < *node*[*newtop*].*alpha* **then**
28:             Create a new node *node*[*newtop*] at *queue.minlevel*
29:         **end if**
30:         *node*[*newtop*] := *node*[*newtop*] + *node*[*stacktop*]
31:         *stacktop* := *newtop*
32:         *curlevel* := *node*[*stacktop*].*alpha*
33:     **end while**
34:     **return** *node*
35: **end procedure**

We have implemented a flooding α-tree algorithm using the HHQ with a cache using HHQ. Algorithm 1 is the pseudocode
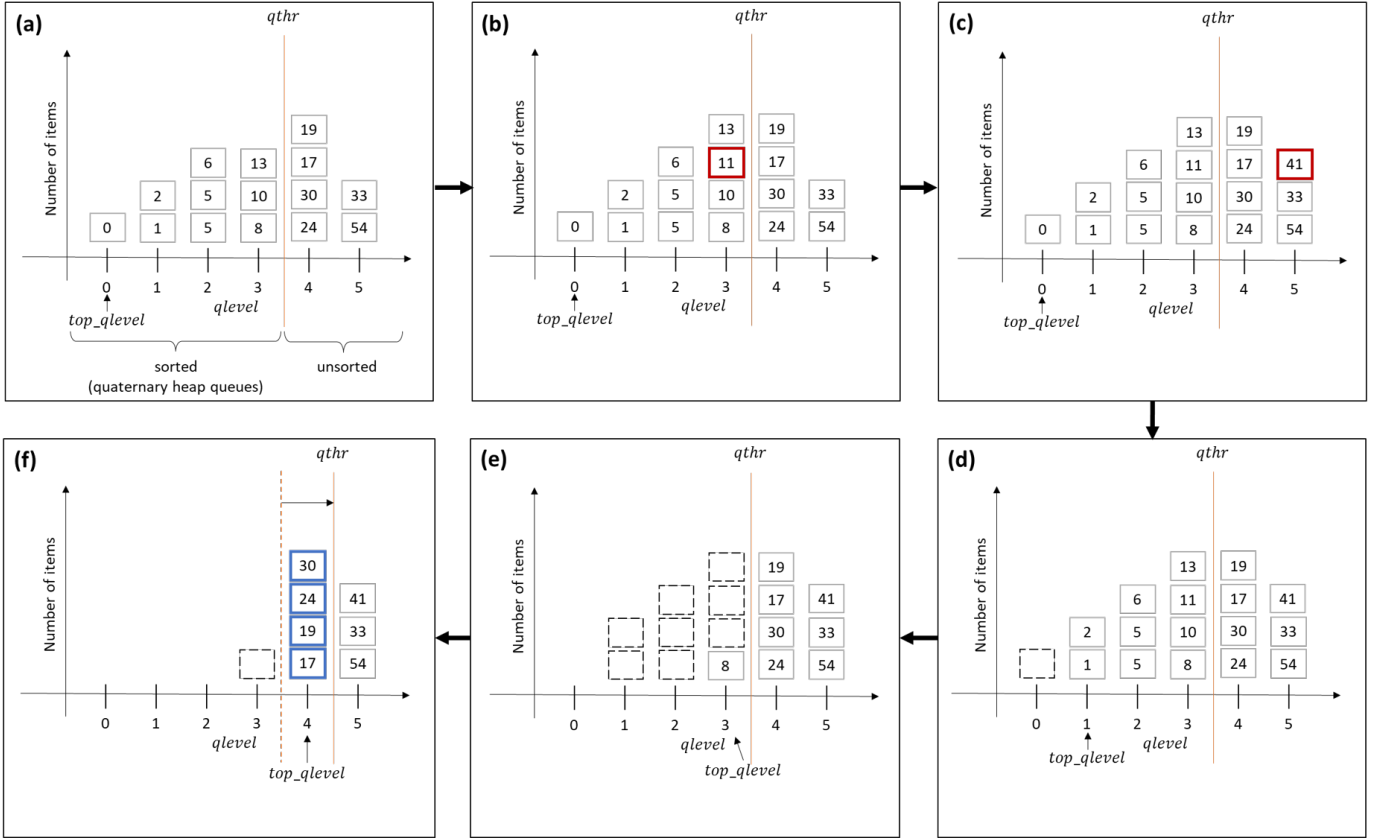
Fig. 5. Illustration of hierarchical queue operations. (a) An HHQ storing 15 items. Numbers on each item represent their α values. (b) Inserting an item with $\alpha = 11$ into the queue. The item is inserted into a sorted queue at $qlevel = 3$ since $qlevel = \lfloor d \log_2(\alpha + 1) \rfloor = 3$ from Eq. 1 ($d = 1$ in this example). (c) Inserting an item with $\alpha = 41$, which is inserted into the unsorted array at $qlevel = 5$. (d) Removing a top item from the queue. The quaternary heap queue at $qlevel = 0$ becomes empty, thus top_qlevel moves to the next non-empty $qlevel$. (e) Removing eight more items. The item with $\alpha = 8$ at $qlevel = 3$ becomes the only item in sorted queues. (f) Removing the last item in the sorted queues. A non-empty unsorted queue at the next top_queue is transformed into a sorted queue, increasing $qthr$ by 1.

of the flooding α-tree algorithm, which is similar to that in [20] but for XDR. The key difference between Alg. 1 and the one in [20] are (1) the histogram (*dhist*) computed in Alg. 1 is a histogram of quantized log of pixel dissimilarities (from Eq. 1), instead of raw pixel dissimilarities used in [20], (2) the root at level array (*levelroot*) from [20] is replaced by keeping track of node traversal of the flooding algorithm as a linked-list stack where *stacktop* in Alg. 1 represents the top of the stack, and (3) *isVisited* array is used in the *pop()* operation (*queue.pop(isVisited)*) in line 11 to discard REs. We have implemented the α-tree flooding and the HHQ in C++, thus the pseudocodes in this section are in C++ style. The C++ code is available at [30].

We define a C++ class for the hierarchical heap priority queue as follows:

```
class HierHeapQueue
{
    HQentry cache[L];
    HeapQueue **HQ;                  ▷ array of heap queues
    HQentry **UA;                    ▷ array of unsorted arrays
    Imgidx qthr; ▷ boundary between HQ and UA: a queue at
qlevel is HQ if qlevel < qthr and is UA otherwise
    Imgidx L;                        ▷ cache size
```

```
    Imgidx num_queues;               ▷ number of queues
    Imgidx *queue_sizes;             ▷ sizes of each queue
    double d;                        ▷ from Eq. 1
}
```

where *HeapQueue* is a standard quaternary heap queue [29], *Imgidx* is an abstract datatype used for pixel indexing that should have a dynamic range wider than $[0, |V| + |E|)$ and *HQentry* is a struct used to store priority queue items as follows:

```
struct HQentry
{
    Imgidx pidx;                     ▷ ending point of the edge
    Pixel alpha;                     ▷ edge-weight
}
```

where *Pixel* is an abstract datatype for pixel values.

Algorithm 2 shows the pseudocode of the constructor and methods of the class *HierHeapQueue*. HIERHEAPQUEUE is the constructor for the class *HierHeapQueue*, where class variables are set and initialized. We found optimal values to be $L_{in} = 4$, $d_{in} = 45$ $r_q = 0.15$ in 4-N and $L_{in} = 13$, and $d_{in} = 45$ $r_q = 0.15$ in 8-N from simulations (see Fig. 12 in Appendix).

Algorithm 2 Hierarchical heap priority queue algorithm

1: **procedure** HIERHEAPQUEUE($dhist, C, |W|, L_{in} = 13, d_{in} = 45, r_q = 0.15$)
2:   $num\_queues := d \times \log_2(|W| + 1)$
3:   $top\_qlevel := num\_queues$ ▷ non-empty queue with the highest priority
4:   $cache\_cur := -1$ ▷ location of the last item in *cache*
5:   $qthr := \lfloor r_q \times num\_queues \rfloor$
6:   $HQ :=$ Array of *num_queues* heap queues
7:   $UA :=$ Array of *num_queues* unsorted arrays
8: **end procedure**

9: **procedure** PUSH($pidx, alpha$)
10:   **if** $alpha$ is lower than the lowest alpha value in HQ **then**
11:     **if** *cache* is full **then**
12:       PUSHQUEUE($cache[L-1].pidx, cache[L-1].alpha$)
13:       $cache[L-1] := (pidx, alpha)$
14:     **end if**
15:     $cache[cache\_cur{+}{+}] := (pidx, alpha)$
16:   **else**
17:     PUSHQUEUE($pidx, alpha$)
18:   **end if**
19: **end procedure**

20: **procedure** PUSHQUEUE($pidx, alpha$)
21:   $qlevel := d \times log2(1 + alpha)$ ▷ from Eq. 1
22:   **if** $qlevel < top\_qlevel$ **then**
23:     $top\_qlevel := qlevel$
24:   **end if**
25:   **if** $qlevel < qthr$ **then**
26:     Add $(pidx, alpha)$ to $HQ[qlevel]$
27:   **else**
28:     Add $(pidx, alpha)$ to $UA[qlevel]$
29:   **end if**
30: **end procedure**

31: **procedure** POP($isVisited$)
32:   $ret := cache[0].pidx$
33:   **if** $cache\_cur = 0$ **then**   ▷ *cache* has only one item
34:     **while** CHECKQUEUELEVEL() $= 0$ **do**
35:       $top\_qlevel{+}{+}$
36:     **end while**
37:     $cache[0] :=$ Top item from $HQ[top\_qlevel]$
38:     POPQUEUE()
39:   **else**
40:     $cache[0 : cache\_cur - 1] := cache[1 : cache\_cur]$ ▷ Shift all items in *cache* towards the top by one slot
41:     $cache\_cur{-}{-}$
42:   **end if**
43:   **return** ret
44: **end procedure**

45: **procedure** POPQUEUE
46:   $HQ[top\_qlevel] \rightarrow$ POP()
47:   **if** $HQ[top\_qlevel] \rightarrow cursize = 0$ **then**
48:     **do**
49:       $top\_qlevel{+}{+}$
50:     **while** CHECKQUEUELEVEL() $= 0$
51:   **end if**
52: **end procedure**

53: **procedure** CHECKQUEUELEVEL
Ensure: The queue at a level *top_qlevel* is a heap queue
54:   **if** $top\_qlevel < qthr$ **then**
55:     **return** $HQ[top\_qlevel].cursize$
56:   **else**
57:     $qthr{+}{+}$
58:     $HQ[qthr] :=$ new HQ of size $UA[qthr].size()$
59:     **for** each $(pidx, alpha)$ in $UA[qthr]$ **do**
60:       **if** pixel $pidx$ is not visited **then**
61:         Add $(pidx, alpha)$ to $HQ[qlevel]$
62:       **end if**
63:     **end for**
64:     Delete $UA[qthr]$
65:     **return** $HQ[top\_qlevel].cursize$
66:   **end if**
67: **end procedure**

MINLEV is a method that returns the α value of the top item in the queue. The insertion function PUSH first checks if the new item should be inserted in *cache*, as in Fig. 4(a). If *cache* is full and the new item has higher α value than the last item of *cache*, the new item is inserted to the HHQ by calling *PushQueue* method, whose algorithm is as we described in Fig. 5(a).

The method POP removes an item from the queue. As described in Fig. 4(b), POP removes an item from *cache*, and if *cache* becomes empty as a result, it removes an item from the HHQ to keep the first slot in *cache* occupied (lines 34–38). Before removing an item from the HHQ, it makes sure that *top_qlevel* is a non-empty heap queue (i.e., *top_qlevel* < *qthr* and $HQ[top\_qlevel].get\_cursize() > 0$), by iteratively invoking CHECKQUEUELEVEL until the condition is met. CHECKQUEUELEVEL is the method that checks if *top_qlevel* is a heap queue, and if it is an unsorted array, it converts the unsorted array into a heap queue, as described in Fig. 5(c) (lines 58–64). When transferring an items from the unsorted array to the heap queue, the subroutine checks if the pixel pointed by an item has already visited by inspecting *isVisited*. If the pixel already has been visited, this means there

is another path at a lower α-value that connects the endpoints of an edge corresponding to the item being processed, thus making it redundant or residual edge. Therefore, the algorithm discards items pointing to already-visited pixels (line 60). We found discarding REs to be crucial in the algorithm performance. Experimental results in the following section show that 85–88% of REs are discarded. After the loop in lines 34–38 is finished, the top item from the queue replaces the first slot in *cache* (line 37), and POPQUEUE is executed to remove the top item from the queue. POPQUEUE removes a top item from $HQ[top\_qlevel]$ and finds a new *top_qlevel* if $HQ[top\_qlevel]$ becomes empty, similarly to the deletion in hierarchical queue [21].

III. RESULT AND DISCUSSION

We have implemented the α-tree flooding algorithm using the proposed hierarchical heap priority queue in C++ [30]. We have also implemented other α-tree algorithms for comparison: union-find (UF) [16], flooding using hierarchical queue (HierQ) [21], heap queue (HeapQ) [27] and trie queue [28]. In UF edges are pre-sorted using radix sort and an α tree is built by iterating over sorted edges in ascending order and creating unions of

two subtree roots associated to the endpoints of edges [16]. In addition, we have used path compression to speed up the search for subtree roots [31] and union by rank up to 16-bit to reduce depth of the tree [32]. Flooding using HeapQ is mostly an α-tree algorithm version of its max-tree counterpart [27], except for the use of quaternary tree instead of binary tree to reduce the depth of the tree. In TrieQ we have used 64-ary trie where each single-bit trie node representing an edge has 64 child nodes stored in 64-bit data. Since the trie data structure is designed to represent ranks of edges instead of their α values, TrieQ is the only flooding algorithm in this study that operates on ranks of edges instead of α values (see [28] and [30] for detail). We have added an extra one bit as a child of each leaf node in the trie, which is necessary to indicate the direction of flooding, even though it nearly doubles the size of the trie.

We have applied the priority queue cache not only to the proposed but also to other flooding algorithms, to investigate how the cache affects performances of other types of priority queues. Since all priority queues should produce the same output given the same input, interactions between the cache and the queue should be identical regardless of the choice of priority queue, except for the trie which operates differently as mentioned above. Optimal cache sizes can differ by queues since computation timings of queue operations can be different. We have used randomly generated images of size 100 Mpix (unless stated otherwise) in different bit depths ranging from 6 bits to 64 bits, as well as five Sentinel-2A remotely sensed images of size 120.56 Mpix [33]. The experiment has been performed on a computer with an AMD Ryzen 7 4800H CPU and 64GB memory.

A. Processing Speed and Memory Use

Figure 6(a) shows the processing speed and memory use of different α-tree algorithms using HierQ, HeapQ, TrieQ, UF, and HHQ, with and/or without the cache (C) and with and/or without unsorted array (UA) in different bit depths in 4-N, on randomly generated images of size 100Mpix. The priority queue cache improves all types of priority queues especially the HHQ in all bit depths and the HierQ in 12–24 bits. This is because the HHQ in all bit depths and the HierQ in HDR and XDR have high computational costs for deletion, thus the cache improves its performance significantly by replacing the queue deletion ($O(\log |W|)$) by the cache deletion ($O(1)$). Runtimes of HeapQ are also improved by the cache in all bit depths because computational complexities of HeapQ operations are ($O(\log |E|)$), which is independent of the bit depth. The TrieQ benefits the least from the cache. We presume that this is because the TrieQ already has quite low computational costs for both insertion and deletion. The proposed HHQ + C performs significantly better than any other α-tree algorithms above 16 bits. It has processing speeds of 5.46 Mpix/s in 32 bits and 4.79 Mpix/s in 64 bits, while the best conventional α-tree algorithm (HeapQ) has those of 1.59 Mpix/s in 32 bits and 1.52 Mpix/s in 64 bits.

Figure 6(b) shows runtimes of α-tree algorithms in 8-N. The cache still improves timings of priority queues, but performance gains are not as high as in 4-N. This is because there are

fewer high-priority edges that benefit from the cache: 25% of $|E|$ in 8-N compared to 45% in 4-N. The ratio of non-REs in 8-N is approximately half of that in 4-N. In the union-find algorithm this is not a huge problem because edges are processed in order of α values, and most of non-REs have lower α values than the rest. However, in flooding algorithms, all incident edges of pixels visited have to be inserted into a priority queue in PUSHNEIGHBORS, thus having a large number of superfluous edges has a huge negative effect on flooding algorithms. The effect is especially severe for the heap and trie queues because they have higher computation cost in queue operations: flooding algorithms using these queues are slowed down by approximately a factor of two in 8-N compared to those in 4-N. Flooding algorithms using hierarchical and HHQs are also slowed down but by a lesser degree (about 35–50%), because these queues do not suffer significantly from pushing extra edges, as they have very low computation cost in queue insertion. The proposed algorithm has processing speeds of 3.41 Mpix/s in 32 bits and 2.94 Mpix/s in 64 bits, while the best conventional α-tree algorithm (union-find) has 0.61 Mpix/s in both 32 and 64 bits.

Figures 6(c) and (d) show the memory use of α-tree algorithms in different bit depths in 4-N and 8-N, respectively. As the cache has a negligible effect on the memory use, we omitted algorithms without the cache here. HierQ and HHQ who use the *HQentry* structure in Sec. II-E have higher memory use compared to the rest, because an *HQentry* item have to store both α-value and a pixel index to visit, while in UF, HierQ and TrieQ one of those do not need to be stored explicitly. HHQ has higher memory use than HeapQ as it needs additional pointers to queues at different levels of queues in addition to HQs and UAs themselves. In our implementation of flooding algorithm using TrieQ, an extra temporary memory is used to track mappings from edges to their ranks, thus TrieQ has the highest memory footprint in 8-N and 64-bit in 4-N as the size of the temporary memory doubles as the connectivity goes from 4-N to 8-N. HeapQ and HierQ have significantly lower memory use in 6- and 8-bit because they reduce memory use by predicting the number of redundant nodes using tree size estimation (TSE) [20]. TSE can be applied to any α-tree algorithms in LDR, but we have not implemented TSE on other algorithms since HierQ significantly outperforms everything else in terms of both timing and memory use in LDR. UF uses union by rank, which requires an extra memory space to keep track of ranks of each node only when the bit depth is lower or equal to 16-bit because the frequency of nodes having the same α value decreases significantly beyond 16-bit. This is why the UF's memory decreases from 16- to 18-bit.

Figure 7 shows the timings of α-tree algorithms as a function of image sizes ranging from 2.25Mpix to 64Mpix. Fig. 7 (a)–(d) show runtimes of α-tree algorithms in 4-N, in (a) 8-bit, (b) 16-bit, (c) 32-bit and (d) 64-bit, and Fig. 7 (a)–(d) runtimes in 8-N, in (e) 8-bit, (f) 16-bit, (g) 32-bit and (h) 64-bit. Time complexities of UF, TrieQ, and HeapQ are $O(|E|log|E|)$ [16], [27], [28], and that of HHQ is also $O(|E|log|E|)$ as it also uses heap queues. HierQ has a time complexity of $O(|W||E|)$ [21], thus its runtimes increase exponentially as the bit depth increases. While having the same time complexity as most
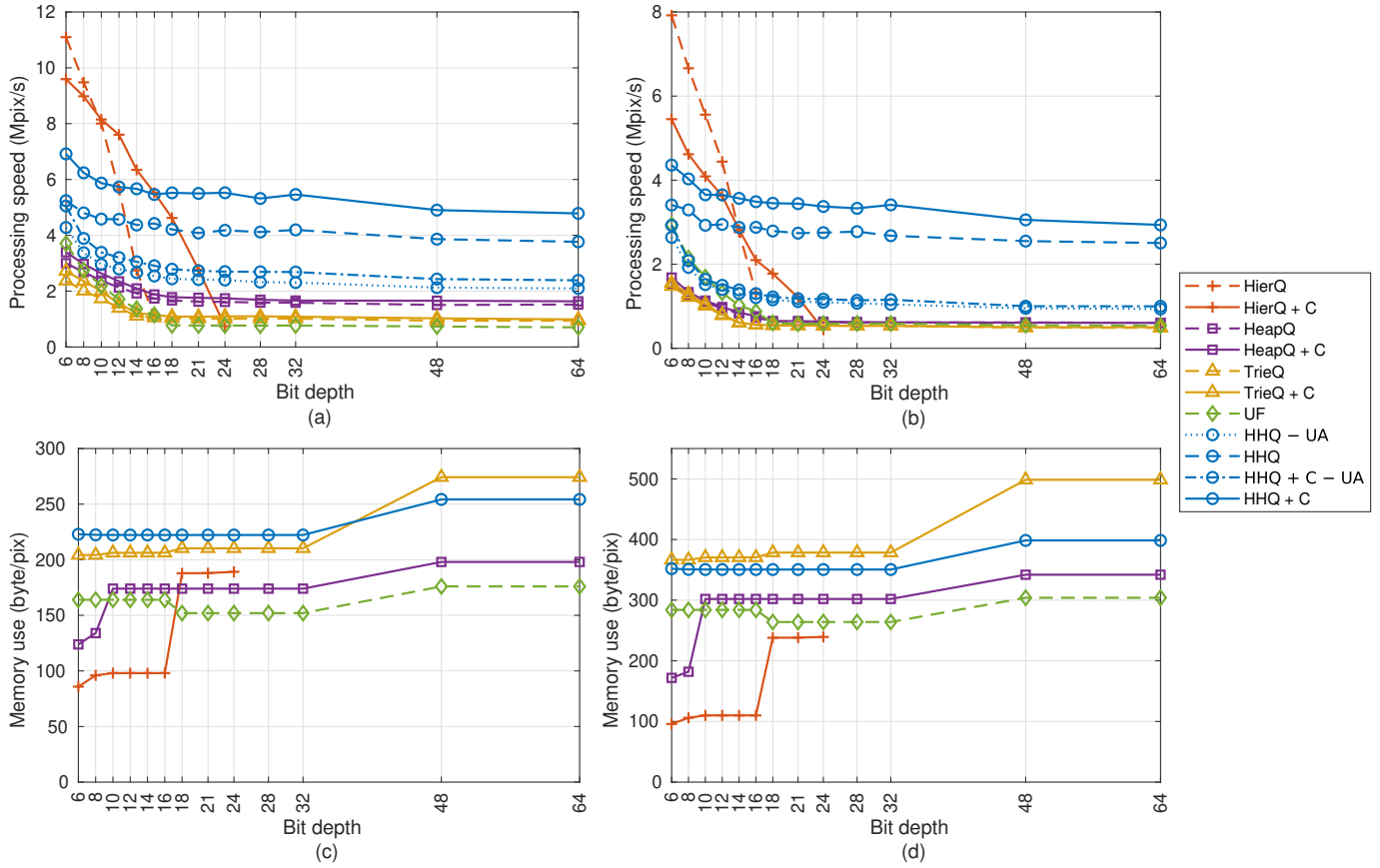
Fig. 6. Performances of different α-tree algorithms using hierarchical queue (HierQ), heap queue (HeapQ), trie queue (TrieQ), union-find (UF), hierarchical heap queue (HHQ), with and/or without cache (C) and with and/or without unsorted array (UA) on randomly generated images of size 100Mpix: (a) runtimes of α-tree algorithms in 4-N, (b) runtimes of α-tree algorithms in 8-N, (c) memory use of α-tree algorithms in 4-N and (d) memory use of α-tree algorithms in 8-N.
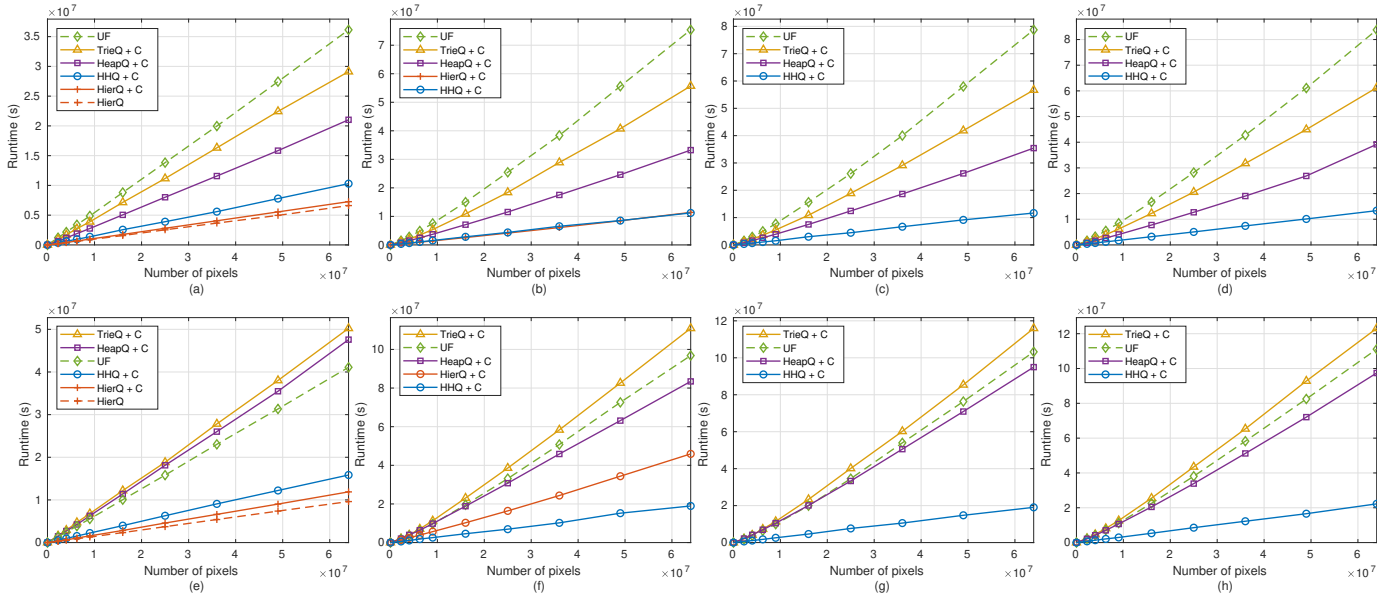

Fig. 7. Runtimes of α-tree algorithms as a function of image sizes ranging from 2.25Mpix to 64Mpix. (a)–(d) runtimes of α-tree algorithms in 4-N, in (a) 8-bit, (b) 16-bit, (c) 32-bit and (d) 64-bit. (e)–(h) runtimes in 8-N, in (e) 8-bit, (f) 16-bit, (g) 32-bit and (h) 64-bit. The proposed hierarchical queue shows constantly low runtimes even in high bit depths and higher connectivity.

of other algorithms, HHQ has much better timings in all bit depths except for 8-bit where HierQ dominates. This is because, according to our finding based on an analysis on the HHQ presented later in this section, HHQ processes 85–88% of REs

in $O(1)$. Once the REs are pushed into an HHQ, 85–88% of them are never moved or inspected, before they are discarded after a single check on *isVisited* array in line 60 in Alg. 2 if and only if the UA it belongs to turns into an HQ. The computation costs of PUSH and POP for REs in the HHQ are similar to those of PUSH and POP in the HierQ. Because the proportion of REs is approximately 50% in 4-N and 75% in 8-N, processing 85–88% of REs in $O(1)$ means processing 42.5% (in 4-N) and 66.6% (in 8-N) of all edges in a very lost computation cost, as low as in HierQ. This is why the HHQ does not scale badly with the number of pixels in XDR and/or in 8-N, compared to other algorithms.

B. Analysis on the Proposed Hierarchical Heap Queue and the Priority Queue Cache

In this section we provide empirical analysis on the proposed HHQ and the priority queue cache. We have conducted an empirical complexity analysis on the HHQ + C, by keeping track of average number of memory moves (NMMs) in the HHQ as a function of the number of items in the queue (Qsize) in push and pop operations. Fig. 8 shows NMMs of the HHQ + C and a naive binary heap queue in push and pop operations in the alpha-tree construction of a randomly-generated image of size 1Mpix using 4N connectivity. The heap queue is "naive" because it does not use the priority queue cache and it does not try to reduce the number of NMMs by delaying the pop operation until it is absolutely necessary, as in Heap + C in Fig. 6. This data can be obtained by setting "PROFILE" macro in our C++ code [30]. NMMs are smoothed over Qsizes using Gaussian kernels ($\sigma = 600$ samples for push and 300 for pop) for better visualization. Fig. 8(a) shows the push operation of HHQ is O(1) with very flat NMMs on every Qsize. This is likely due to low-cost push using the cache and UAs, and sizes of HQs being kept low thanks to a large number of levels in the HHQ and redundant edges being removed when UAs are turning into HQs (line 60 in Alg. 2). Fig. 8(b) shows O(1) amortized complexity of HHQ pop operations. NMMs of pop operation shows high peaks in high Qsizes, with the highest peak located near Qsize = 0.6M. The highest peak represents transformation of UAs with a large number of items into HQs. NMMs beyond this point are consistently high because those pop operations were performed mostly in a large HQ instead of a small one or the cache. It is interesting to note that NMMs beyond the highest peak has a periodic pattern, which is likely from HQ on each layer being emptied one by one.

In addition to the complexity analysis, we have investigated the number of operations and statistics inside the proposed HHQ and HHQ + C by altering the number of queue in the HHQ and the cache size, to analyze how the proposed HHQ and the priority cache speed up the processing speed of the α-tree algorithm. Figure 9 shows statistics of operations in the HHQ while the α-tree algorithm was running on a 64-bit randomly generated image of size 1M pixels. Fig. 9(a) presents the number of items moved inside heap queues in the HHQ. This represents items, which are already pushed into the HHQ before, being moved into or out from one of its heap queues. It does not count an item pushed into or popped out of the
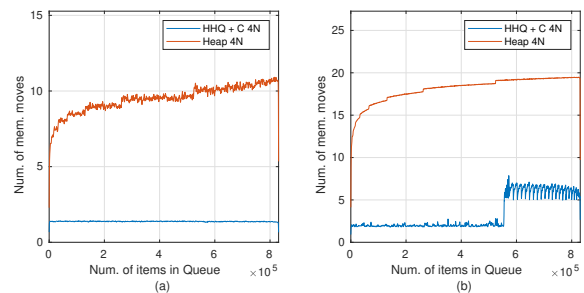

Fig. 8. Empirical complexity analysis of the HHQ and the heap queue on alpha-tree construction of an image of size 1M pixels using 4N. (a) The number of memory moves (NMMs) in terms of the number items in queue in push (b) NMMs in pop. The push and pop operation of HHQ show O(1) amortized complexity.

HHQ, because those counts are constant and are irrelevant to the performance of the queue. An item pushed into or popped from the cache or a UA are counted as moves. Since we used a 64-bit image, $|W| = 2^{64}$ and $log_2(|W|) = 64$ here. The items moves decay exponentially as the number of queues in the HHQ increases, and this is mainly due to more REs being stored in UAs instead of HQs thanks to the sufficient number of queues in the HHQ. If an RE is stored in the same UA as a non-RE that is incident to the same pixel that the RE is incident to, the RE has to be sorted during CHECKQUEUELEVEL in Alg. 2. The probability of REs being sorted decreases as the number of queues in the HHQ increases, therefore the item moves decreases exponentially as the number of queue increases. In addition, the decrease in item moves is also thanks to the lesser depths of quaternary trees in heap queues, by a small amount. This is why the HHQ without UA performs better than HeapQ as shown in Fig. 6. The item moves increases by a small amount when the cache is added to the HHQ, both in 4-N and 8-N. This is due to the items moved into the queue from the cache when a "cache miss" happens, which will be defined later in this section, or an item is popped from the HHQ to fill the empty cache.

Fig. 9(b) shows the number level inspections performed in the HHQ. We define the level inspection as checking a level in a hierarchy of queues if it is empty of items. CHECKQUEUELEVEL in Alg. 2 performs the level inspection which is called at least once whenever POPQUEUE is executed. The cache reduces the number of level inspections by orders of magnitude, both in 4-N and 8-N. This huge decrease in the number of level inspections comes from high priority edges causing *top_qlevel* of the HHQ to fluctuate. For example, when an edge with $\alpha = 0$ enters the HHQ without cache whose *top_qlevel* is 1000 at the time, the edge will be stored at an HQ at level 0 and *top_qlevel* will be updated to 0 since $qlevel = d \times log_2(1 + \alpha) = 0$ from Eq. 1. This edge will be popped immediately right at the next POP, and after the edge with $\alpha = 0$ has left the queue CHECKQUEUELEVEL has to inspect 1000 empty levels until *top_qlevel* climbs all the way back up to 1000. Having a cache in the HHQ solves this problem by queuing the high priority edges in the cache instead of HQs. The cache acts as a "breakwater" that protects
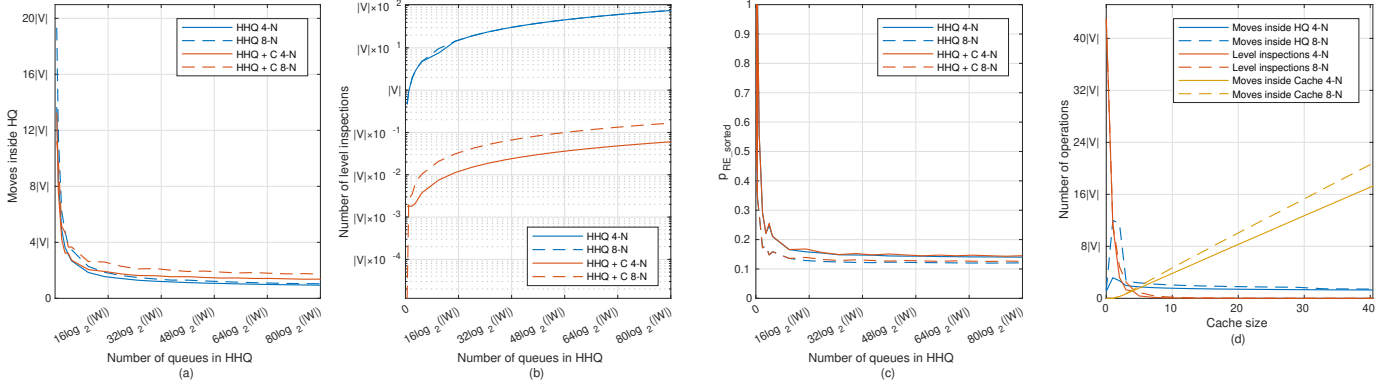
Fig. 9. Statistics in the HHQ in terms of its parameters, using a 64-bit randomly generated image of size 1Mpix. (a) The number of items moved inside heap queues, (b) the number of inspections on queue levels, and (c) the proportion of redundant and residual nodes sorted in the queue, in terms of the number of queues in the HHQ. (d) The number of items moved inside heap queue, the number of level inspections and the number of items moved in the cache as a function of the cache size.

the HHQ against waves of edges, keeping top_qlevel from fluctuating along with the high-frequency fluctuation of α values of edges entering the HHQ. The significant reduction in the number of level inspections thanks to the cache allows the HHQ to have higher number of queues that leads to even lower computation cost. Even a cache of size 1 reduces the number of level inspections significantly, as shown in Fig. 9(d). However, a cache with small sizes increases the number of item moves in the HHQ due to a high probability of the cache being overflowed or emptied causing a lot of items moves between the cache and HQs. Enlarging the size of the cache alleviates this problem, but larger cache sizes leads to higher computation costs for item moves insides the cache. The optimal cache size is a size that compromises the number of level inspections, item moves in HQs, and item moves in the cache. We found the optimal cache size usually lies between 4 and 20 depending on the type of priority queue and the connectivity. See Fig. 12 in Appendix for the detail. Having a cache in the HHQ also makes the implementation of the HHQ easier. Without the cache it is possible to turn low-priority UAs to HQs prematurely if the all of the first batch of edges have high α values, which will significantly slow down the HHQ as the benefit of using UAs is reduced. Avoiding this without using cache is not a trivial task, therefore it can be said that the cache is imperative part of the HHQ.

Fig. 9(c) shows p_{RE_sorted}, the share of REs sorted in the cache or HQs in HHQ. The definition of p_{RE_sorted} is as follows:

$$p_{RE\_sorted} := \frac{|\{e \in E_{RE} | e \text{ is sorted in an HQ}\}|}{|E_{RE}|} \quad (2)$$
$$E_{RE} := \{e \in E | e \text{ is redundant or residual}\}$$

Clearly, it is desirable to have a low p_{RE_sorted}. When the number of queues is 1, in which case the HHQ becomes a HeapQ, p_{RE_sorted} becomes 1. As the number of queues increases p_{RE_sorted} decreases exponentially, and the use of cache also decreases p_{RE_sorted} by a small margin because REs sorted in the cache does not count in p_{RE_sorted}. The values of p_{RE_sorted} using optimal parameters (see Fig. 12 in Appendix)

are 15% without cache and 12% with cache, which means the HHQ processes 85–88% of REs in $O(1)$.

Fig. 9(d) shows the HHQ statistics in terms of the cache size. Level inspections decrease exponentially, down by 75% as the cache size increase from 0 to 1. However, small cache sizes increase the item moves inside HQs, especially in 8-N. This is because items in the cache has to be transferred more frequently when the cache size is small, due to higher chance of cache overflow. We define probabilities of cache miss, cache hit and other cache-related statistics as follows:

$$p_{cache} := \frac{|\{e \in E | e \text{ is stored in cache}\}|}{|E|}$$
$$p_{cache\_miss} := \frac{|\{\text{cached items moved to HQ}\}|}{|\{\text{cached items}\}|}$$
$$p_{cache\_hit} := 1 - p_{cache\_miss}$$
$$p_{cache\_hit\_R} := p_{cache\_hit} \text{ of REs}$$
$$p_{cache\_R} := \frac{|\{e \in (E - E_{RE}) | e \text{ is stored in cache}\}|}{|E|}$$
$$p_{cache\_hit\_NR} := p_{cache\_hit} \text{ of non-REs}$$
$$p_{cache\_NR} := \frac{|\{e \in E_{RE} | e \text{ is stored in cache}\}|}{|E|}$$
$$p_{cache\_moves\_R} := \frac{\text{number of RE item moves inside cache}}{\text{number of item moves inside cache}}$$

(3)

As defined in Eq. 3, we define cache miss as an event when an item entered to the cache is moved to an HQ, because a new item with a higher priority enters to the cache that is already full. This happens more often in 8-N because the frequency of pushes in 8-N is approximately double of that in 4-N. Cache misses cause extra item moves into and from HQs, which is why item moves in HQ in 8-N peaks much higher than in 4-N. Because push and pop operations in the cache are $O(L)$, increasing the size of the cache increases its computation cost quadratically while the benefits of using cache quickly become flattened. We found small cache sizes below 20 are usually the best compromise, as the simulation on randomly generated images in Fig. 12 in Appendix shows. We used a simple sorted
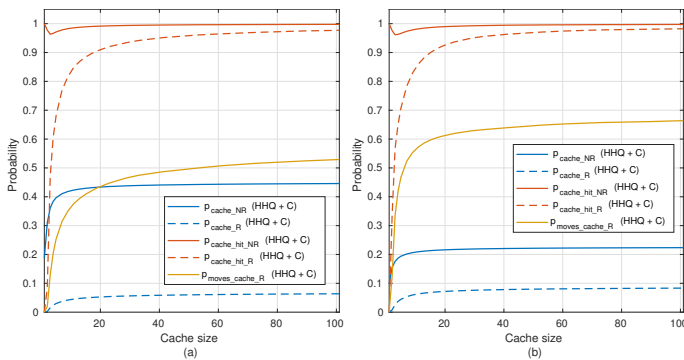
Fig. 10. Statistics of the cache in the HHQ in (a) 4-N and (b) 8-N, from HHQ + C run on a 64-bit randomly generated image of size 1Mpix.

Fig. 11. Processing speeds of α-tree algorithms using the HHQ, HHQ + C, HeapQ and HeapQ + C on five three-channel 16-bit remotely sensed images from Sentinel-2A of size 120.56Mpix, and those of randomly generated images of the same number of channels, bit depth and size, in (a) 4-N and in (b) 8-N.

array to implement the cache. There are other implementation choices with better time complexities such as circular array or linked list, but we found that a simple sorted array of a very small size has the best performance.

Figure 10 shows statistics of the cache in the HHQ in, from HHQ + C run on a 64-bit randomly generated image of size 1Mpix. The proportions of non-REs (p_{cache_NR}) and their cache hit rates ($p_{cache\_hit\_NR}$) shows over 40% in 4-N and over 20% in 8-N edges are cached and 99% of them are cache hit. This attributes the cache to process 37.1% and 20.8% of non-RE edges in $O(1)$, in 4-N and 8-N respectively. Due to low p_{cache_R} and $p_{cache\_hit\_R}$, the cache saves the computations costs of only 1.5% and 5.9% of REs in 4-N and 8-N, respectively; however, increasing the cache size improves the overall performance of the priority queue by reducing high cache miss rate of REs. We analyzed the cache statistics of other priority queues in Fig. 13 in Appendix.

C. Performance on Real-World Images

In addition to the experiment on randomly generated images, we conducted an experiment on Sentinel-2A remotely sensed images [33], to show how the proposed and other α-tree algorithms perform on real-world images that have higher correlations between neighboring pixels compared to randomly generated images. To most α-tree algorithms except for HierQ, the distribution of pixel values does not necessarily have an effect on their timings. Instead, the number of ties and zeros in α values are the major factors in how fast α-tree algorithms run. The pixel distribution might affect the number of ties in α values in a low bit depth and/or when using certain dissimilarity measures, but it is not always the case. For example, even images with large flat zones, which would create many ties and zeros in α values in a low bit depth, will not create as many of those in XDR images since XDR images with their fine-grained quantization can preserve uniqueness of pixel dissimilarities even when the pixel distribution has a very small variance. Even in low bit depth, it is possible to have a low number of ties. Zhang and Wilkinson used odd Gabor filters in pixel dissimilarity to alleviate the chaining effect, which can yield different floating-point dissimilarities even for 8-bit neighboring pixel pairs with the same pixels values, depending on their Gabor filter outputs [15].
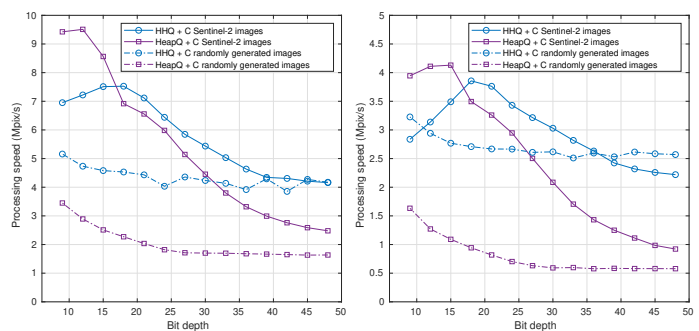
Figure 11 shows processing speeds of α-tree algorithms using the HHQ, HHQ + C, HeapQ and HeapQ + C on five three-channel 16-bit remotely sensed images from Sentinel-2A of size 120.56 Mpix, and those of randomly generated images of the same number of channels, bit depth and size; Fig. 11(a) is in 4-N and Fig. 11(b) is in 8-N. To see the effect of the dynamic range, we incrementally downshifted each pixel from 0 to 13. The dynamic range of pixel dissimilarities without downshifting is 48-bit since we used L^2 norm, and it goes down to 9-bit since downshifting on each channel by 1 will decrease the dynamic range of the L^2 norm by 3 bits. HeapQ + C performs much faster on Sentinel-2 images compared with HeapQ + C on randomly generated images. This is because having more ties in pixel dissimilarities directly affects the number of swaps performed in the heap queue, and Sentinel-2 images produces many more ties compared to randomly generated images even in 48-bit. In the proposed hierarchical heap queue, the consequence of having a lot of ties is less straightforward. As the number of ties increases, Heap queues in the HHQ perform faster but it becomes harder for the HHQ to distinguish redundant and non-REs when they have the same α value, making it difficult for the HHQ to save computation cost on REs by separating them on UAs. The difference in processing speed between HHQ + C on Sentinel-2 and HHQ + C on randomly generated images is not as high as in HeapQ + C, especially in 4-N and in lower bit depths. An interesting phenomenon is that the performance of HHQ + C drops once the bit-depth goes below 18-bit where too many REs become indistinguishable from non-REs due to ties in α value. HeapQ + C outperforms HHQ + C in LDR on Sentinel-2 images, because thanks to the high number of zeros and ties in α value, the vast majority of edges stored in a few heap queues in the HHQ, eliminating all the benefits of the HHQ and rendering it an inefficient heap queue. However the under-performance of the HHQ in LDR is irrelevant since the HierQ outperforms other algorithms by a large margin on such data.

D. Algorithm Profiling

Figure 12 shows profilings of α-tree algorithms in (a) 32 bits and (b) 64 bits. All flooding algorithms in this figure use cache in their priority queues. Compared with the heap queue,
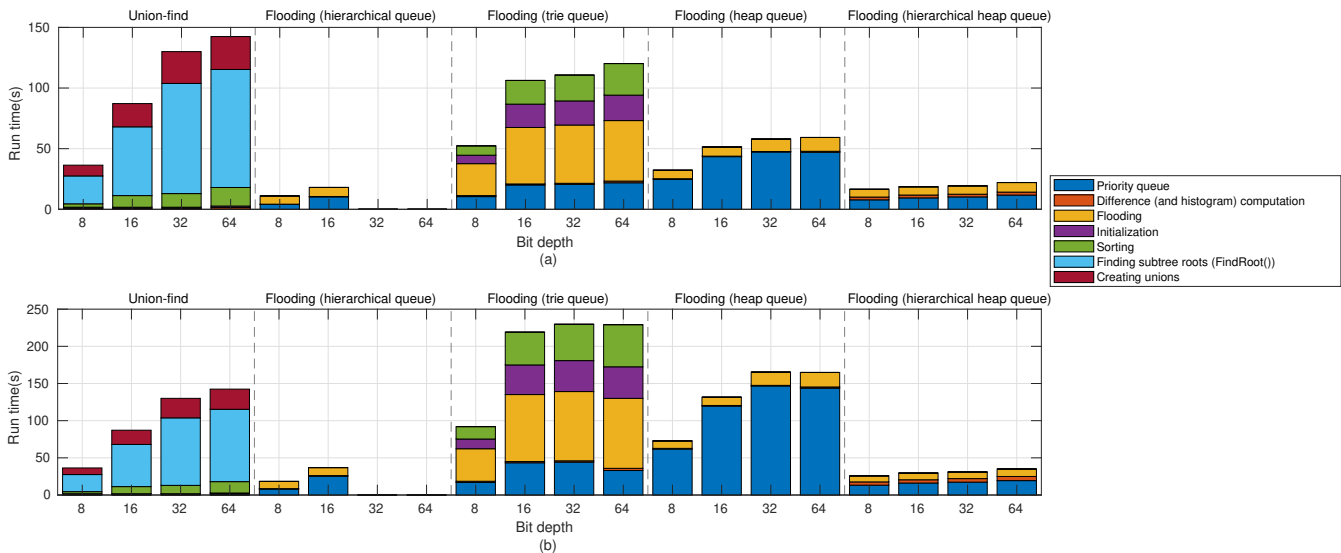
Fig. 12. Profilings of α-tree algorithms in (a) 4-N and (b) 8-N. The hierarchical heap queue significantly reduces the priority queue timing in high dynamic ranges, without having to run extra computations such as in the trie queue. In 64 bits, the hierarchical queue reduces the priority queue timing by a factor of $\frac{47.038s}{11.737s} \approx 4.01$ in 4-N and $\frac{143.7610s}{19.3700s} \approx 7.43$ in 8-N, by replacing the heap queue. These numbers are consistent with ratios of the number of edges processed by those priority queues, which are $\frac{0.55|E|}{0.15|E|} \approx 3.67$ in 4-N and $\frac{0.75|E|}{0.1|E|} = 7.5$ in 8-N (Fig. 3).

the HHQ significantly reduces the priority queue timing in both connectivity. In 64 bits and 4-N, the runtime of priority queue is reduced from 47.0s to 11.7s by replacing a HeapQ with an HHQ, which is approximately a factor of 4.0. This is consistent with the ratio of the number of edges that have to be processed by those queues from what is shown in Fig. 3(a), Sec. II-B: the HHQ processes $0.15|E|$ edges from the edge-weight rank of $r_0 = 0.45|E|$ to $r_1 = 0.60|E|$, while the heap queue processes $0.55|E|$ edges from $r_0 = 0.45|E|$ to $|E|$, which is $\frac{0.55|E|}{0.15|E|} \approx 3.67$ times that of the HHQ. This is also the case in 8-N: the HHQ runs $7.43 \approx \frac{143.8s}{19.4s}$ times faster than the heap queue, where the ratio of the number of edges processed is $\frac{0.75|E|}{0.1|E|} = 7.5$ from Fig. 3(c). This makes clear that the HHQ improves the priority queue timing mainly by handling residual edges better than traditional priority queues.

The HHQ takes more time computing the pixel differences and their histogram. This is because Eq. 1 has to be computed for each edge-weight, but the computation time increase due to this computation is insignificant compared to the time saved by the more efficient priority queue. The trie queue also runs faster than the heap queue, but flooding using trie queue requires sorting all edges and creating mappings between the edge indices and the ranks, which takes much time and causes more cache misses. Union-find does not seem to be affected by the connectivity: it has seemingly the same runtimes and the same profiles in both connectivities.

IV. CONCLUSION

We analyzed the priority queuing in flooding α-tree algorithms in XDR, and implemented a new hierarchical heap queue algorithm that significantly improves α-tree algorithm speed in XDR, largely by avoiding processing of REs. With the help of our efficient α-tree algorithm, a wider variety of

pixel dissimilarity measures in XDR can be used in future α-tree applications on both color or multichannel images and XDR images. The proposed algorithm improved timing of the flooding α-tree algorithms by 1.68 times in 4-N and 2.41 times in 8-N on Sentinel-2A images, and 2.56 times and 4.43 times on randomly generated images. Lower speedups in multichannel images (Sentinel-2) is due to higher cost on computation of pixel dissimilarities, which is constant with respect to the α-tree algorithm. The proposed HHQ with cache have shown superior performance compared to HeapQ especially in images with fewer ties in α values.

In future work, we will try to apply the proposed hierarchical heap queue to other applications that use priority queues, as the proposed queue can outperform traditional priority queues in any applications where items with low priorities are ignored, such as the minimum spanning algorithm. We are also designing an efficient parallel algorithm for α-tree construction in all dynamic ranges.

REFERENCES

[1] P. Soille, "Constrained connectivity for hierarchical image partitioning and simplification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 7, pp. 1132–1145, 2008.

[2] S. J. F. Guimarães, J. Cousty, Y. Kenmochi, and L. Najman, "A hierarchical image segmentation algorithm based on an observation scale," in *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*. Springer, 2012, pp. 116–125.

[3] Y. Xu, E. Carlinet, T. Géraud, and L. Najman, "Hierarchical segmentation using tree-based shape spaces," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 3, pp. 457–469, 2016.

[4] Z. Liu, W. Zou, L. Li, L. Shen, and O. Le Meur, "Co-saliency detection based on hierarchical segmentation," *IEEE Signal Processing Letters*, vol. 21, no. 1, pp. 88–92, 2013.

[5] Y. Fu, S. Liu, H. H. Li, and D. Yang, "Automatic and hierarchical segmentation of the human skeleton in ct images," *Physics in Medicine & Biology*, vol. 62, no. 7, p. 2812, 2017.

[6] G. K. Ouzounis and P. Soille, *The alpha-tree algorithm, theory, algorithms, and applications*, ser. JRC Technical Reports, Joint Research Centre. European Commission, 2012.

[7] W. Tabone, M. H. F. Wilkinson, A. E. Gaalen, J. Georgiadis, and G. Azzopardi, "Alpha-tree segmentation of human anatomical photographic imagery," in *Proceedings of the 2nd International Conference on Applications of Intelligent Systems*, 2019, pp. 1–6.

[8] F. Pacifici, G. K. Ouzounis, L. Gueguen, G. Marchisio, and W. J. Emery, "Very high spatial resolution optical imagery: Tree-based methods and multi-temporal models for mining and analysis," in *Mathematical Models for Remote Sensing Image Processing*. Springer, 2018, pp. 81–135.

[9] F. Merciol and S. Lefèvre, "Fast image and video segmentation based on alpha-tree multiscale representation," in *2012 Eighth International Conference on Signal Image Technology and Internet Based Systems*. IEEE, 2012, pp. 336–342.

[10] M. Silla, "Pest and disease identification in greenhouses for agriculture," *Master's thesis, University of Salerno*, 2021.

[11] C. Shi and C.-M. Pun, "Superpixel-based 3d deep neural networks for hyperspectral image classification," *Pattern Recognition*, vol. 74, pp. 600–616, 2018.

[12] F. Xie, Q. Gao, C. Jin, and F. Zhao, "Hyperspectral image classification based on superpixel pooling convolutional neural network with transfer learning," *Remote sensing*, vol. 13, no. 5, p. 930, 2021.

[13] S. Lefèvre, L. Chapel, and F. Merciol, "Hyperspectral image classification from multiscale description with constrained connectivity and metric learning," in *2014 6th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, 2014, pp. 1–4.

[14] F. Merciol, L. Chapel, and S. Lefèvre, "Hyperspectral image representation through alpha-trees," in *ESA-EUSC-JRC 9th Conference on Image Information Mining*, Bucarest, Romania, 2014, pp. 37–40. [Online]. Available: https://hal.science/hal-00998255

[15] X. Zhang and M. H. F. Wilkinson, "Preventing chaining in alpha-trees using gabor filters," in *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*. Springer, 2019, pp. 268–280.

[16] C. Berger, T. Geraud, R. Levillain, N. Widynski, A. Baillard, and E. Bertin, "Effective component tree computation with application to pattern recognition in astronomical imaging," *Proceedings of IEEE International Conference on Image Processing*, vol. 4, pp. 4–41, 2007.

[17] J. Havel, F. Merciol, and S. Lefèvre, "Efficient tree construction for multiscale image representation and processing," *Journal of Real-Time Image Processing*, vol. 16, no. 4, pp. 1129–1146, 2019.

[18] L. Najman, J. Cousty, and B. Perret, "Playing with kruskal: algorithms for morphological trees in edge-weighted graphs," in *Mathematical Morphology and Its Applications to Signal and Image Processing: 11th International Symposium, ISMM 2013, Uppsala, Sweden, May 27-29, 2013. Proceedings 11*. Springer, 2013, pp. 135–146.

[19] F. Díaz-del Río, P. Sanchez-Cuevas, H. Molina-Abril, P. Real, and M. J. Moron-Fernández, "Building hierarchical tree representations using homological-based tools," in *Computer Analysis of Images and Patterns: 19th International Conference, CAIP 2021, Virtual Event, September 28–30, 2021, Proceedings, Part II*. Springer, 2021, pp. 120–130.

[20] J. You, S. C. Trager, and M. H. F. Wilkinson, "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*, 2019, pp. 256–267.

[21] P. Salembier, A. Oliveras, and L. Garido, "Antiextensive connected operators for image and sequence processing," *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 555–570, 1998.

[22] R. Rönngren, J. Riboe, and R. Ayani, "Lazy queue: an efficient implementation of the pending-event set," *ACM SIGSIM Simulation Digest*, vol. 21, no. 3, pp. 194–204, 1991.

[23] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 7, no. 2, pp. 157–209, 1997.

[24] R. Brown, "Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, no. 10, pp. 1220–1227, 1988.

[25] W. T. Tang, R. S. M. Goh, and I. L.-J. Thng, "Ladder queue: An o (1) priority queue structure for large-scale discrete event simulation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 15, no. 3, pp. 175–204, 2005.

[26] A. Furfaro and L. Sacco, "Exploiting adaptive ladder queue into repast simulation platform," in *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 2018, pp. 1–5.

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

[28] P. Teeninga and M. H. F. Wilkinson, "Fast and memory efficient sequential max-tree construction using a trie-based priority queue," *Submitted to Pattern Recognition Letters*, 2023.

[29] D. B. Johnson, "Priority queues with update and finding minimum spanning trees," *Information Processing Letters*, vol. 4, no. 3, pp. 53–57, 1975.

[30] J. You, "Efficient Alpha-Tree Algorithms." [Online]. Available: https://github.com/jwRyu/AlphaTreeAlgorithms

[31] R. E. Tarjan, "Applications of path compression on balanced trees," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 690–715, 1979.

[32] L. Najman and M. Couprie, "Building the component tree in quasi-linear time," *IEEE Transactions on Image Processing*, vol. 15, no. 11, pp. 3531–3539, 2006.

[33] European Space Agency, "Copernicus Open Access Hub." [Online]. Available: https://scihub.copernicus.eu
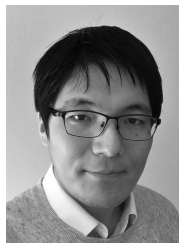
Jiwoo Ryu received his BSc (2011) and MSc (2015) in computer engineering from Kwangwoon University, Seoul, Korea. He is currently a PhD student at Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence in University of Groningen. His research interest include image processing, image morphology, data structures, algorithms and machine learning.

Scott C. Trager received his PhD (1997) in astronomy and astrophysics from the University of California, Santa Cruz. After a Carnegie Starr Fellowship and a Hubble Fellowship, both at the Observatories of the Carnegie Institution of Washington in Pasadena, California, he joined the Kapteyn Astronomical Institute at the University of Groningen as assistant professor of astronomy in 2002, where he was appointed associate professor in 2007 and has been full professor of the Formation and Evolution of Elliptical Galaxies and Astronomical Instrumentation since 2015. His research interests include the formation and evolution of the most massive galaxies in the Universe, the life-cycle of gas and stars in galaxies, astronomical instrumentation, mathematical morphology, and high-dimensional data visualization.

Michael H. F. Wilkinson obtained an MSc in astronomy from the Kapteyn Astronomical Institute, University of Groningen in 1993, after which he worked on image analysis of intestinal bacteria at the Department of Medical Microbiology, University of Groningen, obtaining a PhD at the Institute of Mathematics and Computing Science, also in Groningen, in 1995. He was appointed as researcher at the Centre for High Performance Computing in Groningen working on simulating the intestinal microbial ecosystem on parallel computers. After this he worked as a researcher at the Johann Bernoulli Institute for Mathematics and Computer Science on image analysis of diatoms. He is currently associate professor at the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, working on morphological image analysis and especially connected morphology and models of perceptual grouping. An important research focus is on handling giga- and tera-scale images in remote sensing, astronomy and other digital imaging modalities.