A (fair?) comparison of many max-tree computation algorithms. Appendix

Edwin Carlinet[1] and Thierry Géraud[1]

EPITA Research and Development Laboratory (LRDE)
edwin.carlinet@lrde.epita.fr, thierry.geraud@lrde.epita.fr

A Immersion algorithms

A.1 Union-find without union-by-rank

The algorithm 1 is the union-find based max-tree algorithm as proposed by Berger et al. [2]. It starts with sorting pixels that can be done with a counting sort algorithm for low-quantized data or with a radix sort-based algorithm for high quantized data[1].Then it annotates all pixels as *unprocessed* with -1 (in standard implementations pixel are positive offsets in a pixel buffer). Later in the algorithm, when a pixel p is processed it becomes the root of the component i.e $parent(p) = p$ with $p \neq -1$, thus testing $parent(p) \neq -1$ stands for *is p already processed*. Since S is processed in reverse order and `merge-set` sets the root of the tree to the current pixel p ($parent(r) \leftarrow p$), it ensures that the parent p will be seen before its child r when traversing S in the direct order.

Algorithm 1 Union find without union-by-rank

```
function FIND-ROOT(par, p)
  | if par(p) ≠ p  then  par(p) ← FIND-ROOT(par, par(p))
  | return par(p)
function MAXTREE(ima)
  | for all p do parent(p) ← −1
  | S ← sorts pixels increasing
  | for all p ∈ S backward do
  |   | parent(p) ← p; zpar(p) ← p                          ▷ make-set
  |   | for all n ∈ N_p such that parent(n) ≠ −1 do
  |   |   | r ← FIND-ROOT(zpar, n)
  |   |   | if r ≠ p then
  |   |   |   | zpar(r) ← p; parent(r) ← p                  ▷ merge-set
  | CANONIZE(parent, S)
  | return (parent, S)
```

A.2 Union-find with union-by-rank

The algorithm 2 is similar to algorithm 1 but augmented with union-by-rank. It first introduces a new image *rank*. The `make-set` step creates a tree with a single node, thus with a rank set to 0. The *rank* image is then used when merging two connected sets in *zpar*. Let z_p the root of the connected component of p, and z_n the root of connected component of $n \in \mathcal{N}(p)$. When merging two components, we have to decide whether z_p or z_n becomes the new root w.r.t their rank. If $rank(z_p) < rank(z_n)$, z_p becomes the root, z_n otherwise. If both z_p and z_n have the same rank then we can choose either z_p or z_n as the new root, but the rank should be incremented by one. On the other hand, the relation *parent* is unaffected by the union-by-rank, p becomes the new root whatever the rank of z_p and z_n. Whereas without balancing the root of any point p in *zpar* matches the root of p in parent, this is not the case anymore. For every connected components we have to keep a connection between the root of the component in *zpar* and the root of the max-tree in *parent*. Thus, we introduce an new image *repr* that keeps this connection updated.

Algorithm 2 Union find with union-by-rank

```
procedure MAXTREE(ima)
    for all p do parent(p) ← −1
    S ← sorts pixels increasing
    for all p ∈ S backward do
        parent(p) ← p; zpar(p) ← p                              ▷ make-set
        rank(p) ← 0; repr(p) ← p
        z_p ← p
        for all n ∈ N_p such that parent(n) ≠ −1 do
            z_n ← FIND-ROOT(zpar, n)
            if z_n ≠ z_p then
                parent(repr(z_n)) ← p
                if rank(z_p) < rank(z_n) then swap(z_p, z_n)
                zpar(z_n) ← z_p                                  ▷ merge-set
                repr(z_p) ← p
                if rank(z_p) = rank(z_n) then
                    rank(z_p) ← rank(z_p) + 1
    CANONIZE(parent, S)
    return (parent, S)
```

A.3 Canonization

Both algorithms call the CANONIZE(p)rocedure to ensure that any node's parent is a canonical node. In algorithm 3, canonical property is broadcast downward. S is traversed in direct order such that when processing a pixel p, its parent q has the canonical property that is $parent(q)$ is a canonical element. Hence, if q and $parent(q)$ belongs to the same node i.e $ima(q) = ima(parent(q))$, the parent of p is set to the component's canonical element: $parent(q)$.

Algorithm 3 Canonization algorithm

```
procedure CANONIZE(ima, parent, S)
    for all p in S forward do
        q ← parent(p)
        if ima(q) = ima(parent(q)) then
            parent(p) ← parent(q)
```

A.4 Level compression

Union-by-rank provides time complexity guaranties at the price of an extra memory requirement. When dealing with huge images this results in a significant drawback (e.g. RAM overflow...). Since the last point processed always becomes the root, union-find without rank technique tends to create degenerated trees in flat zones. Level compression avoids this behavior by a special handling of flat zones. In algorithm 4, p is the point in process at level $\lambda = ima(p)$, n a neighbor of p already processed, z_p the root of P_p^λ (at first $z_p = p$), z_n the root of P_n^λ. We suppose $ima(z_p) = ima(z_n)$, thus z_p and z_n belong to the same node and we can choose any of them as a canonical element. Normally p should become the root with child z_n but level compression inverts the relation: z_n is kept as the root and z_p becomes a child. Since *parent* may be inverted, S array is not valid anymore. Hence S is reconstructed, as soon as a point p gets attached to a root node, p will be not be processed anymore so it is inserted in back of S. At the end S only misses the tree root which is $parent[S[0]]$.

B Flooding algorithms

B.1 Salembier's algorithm

Salembier et al. [5] proposed the first efficient algorithm to compute the max-tree. A propagation starts from the root that is the pixel at lowest level l_{min}. Pixels in the propagation front are stored in a hierarchical queue that allows a direct access to pixels at a given level in the queue. The `flood(λ, r)` procedure (see algorithm 5) is in charge of flooding the peak component P_r^λ and building the corresponding sub max-tree rooted in r. It proceeds as follows: first pixels at level λ are retrieved from the queue, their *parent* pointer is set to the canonical element r and their neighbors n are analyzed. If n is not in queue and has not yet been processed, then n is pushed in the queue for further process sing and n is marked as processed ($parent(n)$ is set to `INQUEUE` which is any value different

Algorithm 4 Union find with level compression

```
function MAXTREE(ima)
    for all p do parent(p) ← −1
    S ← sorts pixels increasing
    j = N − 1
    for all p ∈ S backward do
        parent(p) ← p; zpar(p) ← p                                ▷ make-set
        z_p = p
        for all n ∈ N_p such that parent(n) ≠ −1 do
            z_n ← FIND-ROOT(zpar, n)
            if z_p ≠ z_n then
                if ima(z_p) = ima(z_n) then SWAP(()z_p, z_n)
                zpar(z_n) ← z_p; parent(z_n) ← z_p                ▷ merge-set
                S[j] ← z_n; j ← j − 1
    S[0] ← parent[S[0]]
    CANONIZE(parent, S)
    return (parent, S)
```

from -1). If the level l of n is higher than λ then n is in the childhood of the current node, thus `flood` is called recursively to flood the peak component P_n^l rooted in n. During the recursive flood, some points can be pushed in queue between level λ and l. Hence, when `flood` ends, it returns the level l' of n's parent. If $l' > \lambda$, we need to flood level l' until $l' \leq \lambda$ i.e until there are no more points in the queue above λ. Once all pixels at level λ have processed, we need to retrieve the level *lpar* of parent component and attach r to its canonical element. A *levroot* array stores canonical element of each level component and -1 if the component is empty. Thus we just have to traverse *levroot* looking for $lpar = \max\{h < \lambda, levroot[h] \neq -1\}$ and set the parent of r to $levroot[lpar]$. Since the construction of *parent* is bottom-up, we can safely insert p in front of the S array each time $parent(p)$ is set. For a level component, the canonical element is the last element inserted ensuring a correct ordering of S. Note that the first that gets a the minimum level of the image is not necessary. Instead, we could have called `flood` in `Max-tree` procedure until the parent level returned by the function was -1, i.e the last flood call was processing the root.

B.2 Non-recursive versions of Salembier's algorithm

Salembier et al. [5]'s algorithm was rewritten in a non-recursive implementation in Hesselink [3] and later by Nistér and Stewénius [4] and Wilkinson [6]. These algorithms differ in only two points. First, [6] uses a pass to retrieve the root before flooding to mimics the original recursive version while Nistér and Stewénius [4] does not. Second, priority queues in [4] use an unacknowledged implementation of heap based on hierarchical queues while in [6] they are implemented using a standard heap (based on comparisons). The algorithm 6 is a code transcription of the method described in Nistér and Stewénius [4]. The

Algorithm 5 Salembier et al. [5] max-tree algorithm

```
function FLOOD(λ, r)
    while hqueue[λ] not empty do
        p ← POP(hqueue[λ])
        parent(p) ← r
        if p ≠ r then INSERT_FRONT(S, p)
        for all n ∈ N(p) such that parent(p) = −1 do
            l ← ima(n)
            if levroot[l] = −1 then levroot[l] ← n
            PUSH(hqueue[l], n)
            parent(n) ← INQUEUE
            while l > λ do
                l ← flood(l, levroot[l])
                                                    ▷ Attach to parent
            levroot[λ] ← −1
            lpar ← λ − 1
            while lpar ≥ 0 and levroot[lpar] = −1 do
                lpar ← lpar − 1
                if lpar ≠ −1 then
                    parent(r) ← levroot[lpar]
                INSERT_FRONT(S, r)
                return lpar
            function MAX-TREE(ima)
                for all h do levroot[h] ← −1
                for all p do parent(p) ← −1
                l_min ← min_p ima(p)
                p_min ← arg min_p ima(p)
                PUSH(hqueue[l_min], p_min)
                levroot[lmin] ← p_min
                FLOOD(l_min, p_min)
```

array *levroot* in the recursive version is replaced by a stack with the same purpose: storing the canonical element of level components. The hierarchical queue *hqueue* is replaced by a priority queue *pqueue* that stores the propagation front. The algorithm starts with some initialization and choose a random point p_{start} as the flooding point. p_{start} is enqueued and pushed on *levroot* as canonical element. During the flooding, the algorithm picks the point p at highest level (with the highest priority) in the queue, and the canonical element r of its component which is the top of *levroot* (p is not removed from the queue). Like in the recursive version, we look for neighbors n of p and enqueue those that have not yet been seen. If $ima(n) > ima(p)$, n is pushed on the stack and we immediately flood n (a *goto* that mimics the recursive call). On the other hand, if all neighbors are in the queue or already processed then p is *done*, it is removed from the queue, $parent(p)$ is set its the canonical element r and if $r \neq p$, p is added to

Algorithm 6 Non-recursive max-tree algorithm [4, 6]

```
function MAX-TREE(ima)
    for all p do parent(p) ← −1
    p_start ← any point in Ω
    PUSH(pqueue, p_start); PUSH(levroot, p_start)
    parent(p_start) ← INQUEUE
    loop
        p ← TOP(pqueue); r ← TOP(levroot)
        for all n ∈ N(p) such that parent(p) = −1 do
            PUSH(pqueue, n)
            parent(n) ← INQUEUE
            if ima(p) < ima(n) then
                PUSH(levroot, n)
                goto 7
        { p is done }
        POP(pqueue)
        parent(p) ← r
        if p ≠ r then INSERT_FRONT(S, p)
    while pqueue not empty do;
        { all points at current level done ? }
        q ← TOP(pqueue)
        if ima(q) ≠ ima(r) then                          ▷ Attach r to its parent
            PROCESSSTACK(r, q)
        repeat
        root ← POP(levroot)
        INSERT_FRONT(S, root)
```

S (we have to ensure that the canonical element will be inserted last). Once p removed from the queue, we have to check if the level component has been fully processed in order to attach the canonical element r to its parent. If the next pixel q has a different level than p, we call the procedure `ProcessStack` that pops the stack, sets parent relationship between canonical elements and insert them in S until the top component has a level no greater than $ima(q)$. If the stack top's level matches q's level, q extends the component so no more process is needed. On the other hand, if the stack gets empty or the top level is lesser than $ima(q)$, then q is pushed on the stack as the canonical element of a new component. The algorithm ends when all points in queue have been processed, then S only misses the root of the tree which is the single element that remains on the stack.

C Merge-based algorithms and parallelism

The procedure in charge of merging sub-trees T_i and T_j of two adjacent domains D_i and D_j is given in algorithm 7. For two neighbors p and q in the junction of

Algorithm 6 Non-recursive max-tree algorithm (continued)

```
procedure ProcessStack(r, q)
    λ ← ima(q)
    pop(levroot)
    while levroot not empty and λ < ima(top(levroot)) do
        insert_front(S, r)
        r ← parent(r) ← pop(levroot))
        if levroot empty or ima(top(levroot)) ≠ λ then
            push(levroot, q)
        parent(r) ← top(levroot)
        insert_front(S, r)
```

D_i, D_j, it connects components of p's branch in T_i to components of q's branch in T_j until a common ancestor is found. Let x and y, canonical elements of components to merge with $ima(x) \geq ima(y)$ (x is in the childhood to y) and z, canonical element of the parent component of x. If x is the root of the sub-tree then it gets attached to y and the procedure ends. Otherwise, we traverse up the branch of x to find the component that will be attached to y that is the lowest node having a level greater than $ima(y)$. Once found, x gets attached to y, and we now have to connect y to x's old parent. Function `findrepr(p)` is used to get the canonical element of p's component whenever the algorithm needs it.

Algorithm 8 Canonization and S computation algorithm

```
procedure CanonizeRec(p)
    dejavu(p) = true
    q ← parent(p)
    if not dejavu(q) then                          ▷ Process parent before p
        CanonizeRec(q)
    if ima(q) = ima(parent(q) then                 ▷ Canonize
        parent(p) ← parent(q)
    InsertBack(S, p)

for all p do dejavu(p) ← False
for all p ∈ Ω such that not dejavu(p) do
    CanonizeRec(p)
```

Once sub-trees have been computed and merged into a single tree, it does not hold canonical property (because non-canonical elements are not updated during merge). Also, reduction step does not merge S array corresponding to sub-trees (it would imply reordering S which is more costly than just recomputing it at the end). Algorithm 8 performs canonization and reconstructs S array from *parent*

Algorithm 7 Tree merge algorithm

```
function FINDREPR(par, p)
    if ima(p) ≠ ima(par(p)) then return p
    par(p) ← FINDREPR(par, par(p))
    return par(p)

procedure CONNECT(p,q)
    x ← FINDREPR(parent, p)
    y ← FINDREPR(parent, q)
    if ima(x) < ima(y) then SWAP(x, y)
    while x ≠ y do                                   ▷ common ancestor found ?
        parent(x) ← FINDREPR(parent, parent(x));
        z ← parent(x)
        if x = z then                                ▷ x is root
            parent(x) ← y; y ← x
        else if ima(z) ≥ ima(y) then
            x ← z
        else
            parent(x) ← y
            x ← y
            y ← z

    procedure MERGETREE(D_i, D_j)
        for all (p, q) ∈ D_i × D_j such that q ∈ N(p) do
            CONNECT(p,q)
```

image. It uses an auxiliary image *dejavu* to track nodes that have already been inserted in S. As opposed to other max-tree algorithms, construction of S and processing of nodes are top-down. For any points p, we traverse in a recursive way its path to the root to process its ancestors. When the recursive call returns, $parent(p)$ is already inserted in S and holds the canonical property, thus we can safely insert back p in S and canonize p as in algorithm 3.

Bibliography

[1] Andersson, A., Hagerup, T., Nilsson, S., Raman, R.: Sorting in linear time? In: Proc. of the Annual ACM symposium on Theory of computing. pp. 427–436 (1995)

[2] Berger, C., Géraud, T., Levillain, R., Widynski, N., Baillard, A., Bertin, E.: Effective component tree computation with application to pattern recognition in astronomical imaging. In: Proc. of ICIP. vol. 4, pp. IV–41 (2007)

[3] Hesselink, W.H.: Salembier's min-tree algorithm turned into breadth first search. Information processing letters 88(5), 225–229 (2003)

[4] Nistér, D., Stewénius, H.: Linear time maximally stable extremal regions. In: Proc. of ECCV. pp. 183–196 (2008)

[5] Salembier, P., Oliveras, A., Garrido, L.: Antiextensive connected operators for image and sequence processing. IEEE Trans. on Ima. Proc. 7(4), 555–570 (1998)
[6] Wilkinson, M.H.F.: A fast component-tree algorithm for high dynamic-range images and second generation connectivity. In: Proc. of ICIP. pp. 1021–1024 (2011)