Avoiding the Global Sort: A Faster Contour Tree Algorithm*

Benjamin Raichel[†] C. Seshadhri[‡]

Abstract

We revisit the classical problem of computing the *contour tree* of a scalar field $f : \mathbb{M} \to \mathbb{R}$, where $\mathbb{M}$ is a triangulations of a ball in $\mathbb{R}^d$. The contour tree is a fundamental topological structure that tracks the evolution of level sets of f and has numerous applications in data analysis and visualization.

All existing algorithms begin with a global sort of at least all critical values of f, which can require (roughly) $\Omega(n \log n)$ time, where n is the number of vertices of the mesh. Existing lower bounds show that there are pathological instances where this sort is required. We present the first algorithm whose time complexity depends on the contour tree structure, and avoids the global sort for non-pathological inputs. (We assume that the Morse degree of the function f at any point is at most 3.) If C denotes the set of critical points in $\mathbb{M}$, the running time is roughly $O(N + \sum_{v \in C} \log \ell_v)$, where ℓ_v is the depth of v in the contour tree and N is the total complexity of $\mathbb{M}$. This matches all existing upper bounds, but is a significant asymptotic improvement when the contour tree is short and fat. Specifically, our approach ensures that any comparison made is between nodes that are either adjacent in $\mathbb{M}$ or in the same descending path in the contour tree, allowing us to argue strong optimality properties of our algorithm.

Our algorithm requires several novel ideas: partitioning $\mathbb{M}$ in well-behaved portions, a local growing procedure to iteratively build contour trees, and the use of heavy path decompositions for the time complexity analysis.

*Erratum in Appendix §B.

[†]Department of Computer Science; University of Texas at Dallas; Richardson, TX, 75080, USA; benjamin.raichel@utdallas.edu.

[‡]Department of Computer Science; University of California, Santa Cruz; Santa Cruz, CA, 95064, USA; scomandu@ucsc.edu

1 Introduction

Geometric data is often represented as a function $f : \mathbb{R}^d \to \mathbb{R}$. Typically, a finite representation is given by considering f to be piecewise linear over some triangulated mesh (i.e. simplicial complex) $\mathbb{M}$ in $\mathbb{R}^d$. *Contour trees* are a topological structure used to represent and visualize the function f. It is convenient to think of f as a simplicial complex sitting in $\mathbb{R}^{d+1}$, with the last coordinate (i.e. height) given by f. Imagine sweeping the hyperplane $x_{d+1} = h$ with h going from $+\infty$ to $-\infty$. At every instance, the intersection of this plane with f gives a set of connected components, the *contours* at height h. As the sweeping proceeds various events occur: new contours are created or destroyed, contours merge into each other or split into new components. The contour tree is a concise representation of these events [vKvOB+97, CSA03].

If f is smooth, all points where the gradient of f is zero are *critical points*. These points are the "events" where the contour topology changes. The contour tree tracks a specific subset of events, called "joins" and "splits". We provide formal definitions later. An edge of the contour tree connects two such critical points if one event immediately "follows" the other as the sweep plane makes its pass. Figure 1 and Figure 2 show examples of simplicial complexes, with heights and their contour trees. Think of the contour tree edges as pointing downwards. Leaves are either maxima or minima, and internal nodes are either "joins" or "splits".

Consider $f : \mathbb{M} \to \mathbb{R}$, where $\mathbb{M}$ is the triangulation of a d-dimensional ball with n vertices, N faces in total, and $t \leq n$ critical points. (We assume that $f : \mathbb{M} \to \mathbb{R}$ is a linear interpolant over distinct valued vertices, with Morse degree at most 3, see Definition 2.3. The degree assumption can be achieved via vertex unfolding, as discussed later, and is standard in this literature [vKvOB+97].) A fundamental result in this area is the algorithm of Carr, Snoeyink, and Axen to compute contour trees, which runs in $O(n \log n + N\alpha(N))$ time [CSA03] (where $\alpha(\cdot)$ denotes the inverse Ackermann function). In practical applications, N is typically $\Theta(n)$ (certainly true for $d = 2$). The most expensive operation is an initial sort of all the vertex heights. Chiang *et al.* build on this approach to get a faster algorithm that only sorts the critical vertices, yielding a running time of $O(t \log t + N)$ [CLLR05]. Common applications for contour trees involve turbulent combustion or noisy data, where the number of critical points is likely to be $\Omega(n)$. There is a worst-case lower bound of $\Omega(t \log t)$ by Chiang *et al.* [CLLR05], based on a construction of Bajaj *et al.* [BKO+98].

All previous algorithms begin by sorting (at least) the critical points. Can we beat this sorting bound for certain instances, and can we characterize which inputs are hard? Intuitively, points that are incomparable in the contour tree do not need to be compared. Look at Figure 1 to see such an example. All previous algorithms waste time sorting all the maxima. Also consider the surface of Figure 2. The final contour tree is basically two binary trees joined at their roots, and we do not need the entire sorted order of critical points to construct the contour tree.

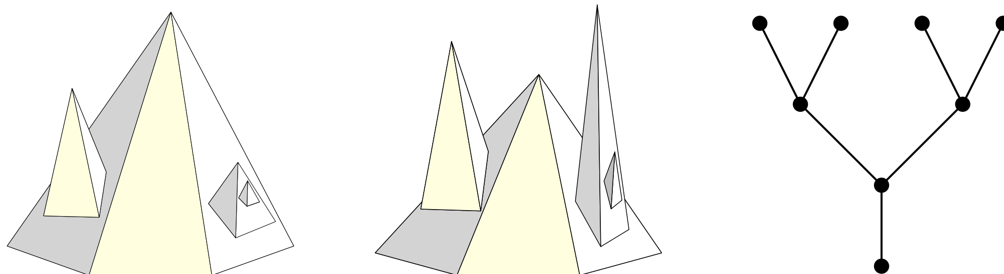

Figure 1: Two surfaces with different orderings of the maxima, but the same contour tree.

Our main result gives an affirmative answer. Remember that we can consider the contour tree

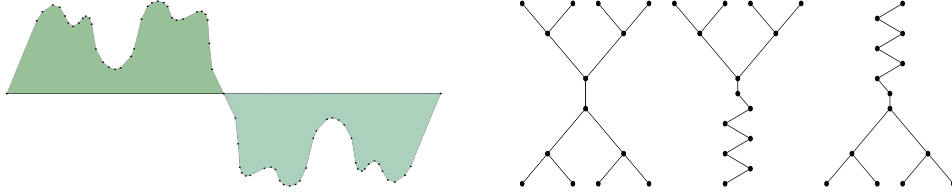

Figure 2: On left, a surface with a balanced contour tree, but whose join and split trees have long tails. On right (from left to right), the contour, join and split trees.

as directed from top to bottom. For any node v in the tree, let ℓ_v denote the length of the longest directed path passing through v.

Theorem 1.1. *Consider a simplicial complex $f : \mathbb{M} \to \mathbb{R}$, where $\mathbb{M}$ is a triangulation of a d-dimensional ball. Let N be the number of facets in $\mathbb{M}$. Let the Morse degree of f at any point be at most 3. Denote the contour tree by T with vertex set (the critical points) $C(T)$. There exists an algorithm to compute T in $O(\sum_{v \in C(T)} \log \ell_v + t\alpha(t) + N)$ time. Moreover, this algorithm only compares function values at pairs of vertices that are ancestor-descendant in T or adjacent in $\mathbb{M}$.*

Essentially, the "run time per critical point" is determined by the height/depth of the point in the contour tree. This bound immediately yields a run time of $O(t \log D + t\alpha(t) + N)$, where D is the directed diameter of the contour tree. This is a significant asymptotic improvement for short and fat contour trees. For example, if the tree is balanced, then we get a bound of $O(t \log \log t)$. Even if T contains a long path of length $O(t/\log t)$, but is otherwise short, we get the improved bound of $O(t \log \log t)$.

1.1 A refined bound with optimality properties

Theorem 1.1 is a direct corollary of a stronger but more cumbersome theorem.

Definition 1.2. *For a contour tree T, a* leaf path *is any path in T containing a leaf, which is also monotonic in the height values of its vertices. Then a* path decomposition*, $P(T)$, is a partition of the vertices of T into a set of vertex disjoint leaf paths.*

Theorem 1.3. *(Consider the input setting of Theorem 1.1.) There is a deterministic algorithm to compute the contour tree, T, whose running time is $O(\sum_{p \in P(T)} |p| \log |p| + t\alpha(t) + N)$, where $P(T)$ is a specific path decomposition (constructed implicitly by the algorithm). The number of comparisons made is $O(\sum_{p \in P(T)} |p| \log |p| + N)$. In particular, comparisons are only made between pairs of vertices that are ancestor-descendant in T or adjacent in $\mathbb{M}$.*

Note that Theorem 1.1 is a direct corollary of this statement. For any v, ℓ_v is at least the length of the path in $P(T)$ that contains v. This bound is strictly stronger, since for any balanced contour tree, the run time bound of Theorem 1.3 is $O(t\alpha(t) + N)$, and $O(N)$ comparisons are made. (Since for a balanced tree, one can show $\sum_{p \in P(T)} |p| \log |p| = O(t)$.)

The bound of Theorem 1.3 may seem artificial, since it actually depends on the $P(T)$ that is implicitly constructed by the algorithm. Nonetheless, we prove that the algorithm of Theorem 1.3 has strong optimality properties. For convenience, fix some value $t = \Omega(N)$, and consider the set of terrains ($d = 2$) with t critical points. The bound of Theorem 1.3 takes values ranging from t to $ct \log t$, for some constant c. Consider some $\gamma \in [t, t \log t]$, and consider the set of terrains where the algorithm makes $\alpha\gamma$ comparisons, for any constant α. Then *any algorithm* must make roughly γ comparisons in the worst-case over this set. (Further details are in §9.)

Theorem 1.4. *There exists some absolute constant α such that the following holds. For sufficiently large t and any $\gamma \in [t, t \log t]$, consider the set $\mathbf{F}_\gamma$ of terrains with t critical points such that the number of comparisons made by the algorithm of Theorem 1.3 on these terrains is in $[\gamma, \alpha\gamma]$. Any algebraic decision tree that correctly computes the contour tree on all of $\mathbf{F}_\gamma$ has a worst case running time of $\Omega(\gamma)$.*

1.2 Previous work

Contour trees were first used to study terrain maps by Boyell and Ruston, and Freeman and Morse [BR63, FM67]. Contour trees have been applied in analysis of fluid mixing, combustion simulations, and studying chemical systems [LBM+06, BWP+10, BWH+11, BWT+11, MGB+11]. Carr's thesis [Car04] gives various applications of contour trees for data visualization and is an excellent reference for contour tree definitions and algorithms.

The first formal result was an $O(N \log N)$ time algorithm for functions over 2D meshes and an $O(N^2)$ algorithm for higher dimensions, by van Kreveld *et al.* [vKvOB+97]. Tarasov and Vyalyi [TV98] improved the running time to $O(N \log N)$ for the 3D case. The influential paper of Carr *et al.* [CSA03] improved the running time for all dimensions to $O(n \log n + N\alpha(N))$. Pascucci and Cole-McLaughlin [PCM02] provided an $O(n + t \log n)$ time algorithm for 3-dimensional structured meshes. Chiang *et al.* [CLLR05] provide an unconditional $O(N + t \log t)$ algorithm.

Contour trees are a special case of Reeb graphs, a general topological representation for real-valued functions on any manifold. Algorithms for computing Reeb graphs is an active topic of research [SK91, CMEH+03, PSBM07, DN09, HWW10, Par12], where two results explicitly reduce to computing contour trees [TGSP09, DN13].

2 Contour tree basics

We detail the basic definitions about contour trees, following the terminology of Chapter 6 of Carr's thesis [Car04]. All our assumptions and definitions are standard for results in this area, though there is some variability in notation. The input is a continuous piecewise-linear function $f : \mathbb{M} \to \mathbb{R}$, where $\mathbb{M}$ is a fully triangulated simplicial complex of a topological ball in $\mathbb{R}^d$, except for specially designated *boundary facets*. We assume that all pairs of vertices from $\mathbb{M}$ have distinct function values, except for pairs from the same boundary facet, and that f is only explicitly defined on the vertices, and all other values are obtained by linear interpolation.

We assume that the boundary values satisfy a special property. This is mainly for convenience in presentation.

Definition 2.1. *The function f is* boundary critical *if the following holds. Consider a boundary facet F. All vertices of F have the same function value. Furthermore, all neighbors of vertices in F, which are not also in F itself, either have all function values strictly greater than or all function values strictly less than the function value at F.*

Boundary facets allow us to capture the resulting surface pieces after our algorithm makes a horizontal cut. The above definition is convenient, as any point within a given facet has a well-defined height, including the boundary facets.

We think of the dimension d, as constant, and assume that $\mathbb{M}$ is represented in a data structure that allows constant-time access to neighboring simplices in $\mathbb{M}$ (e.g. incidence graphs [Ede87]). (This is analogous to a doubly connected edge list, but for higher dimensions.) Observe that $f : \mathbb{M} \to \mathbb{R}$ can be thought of as a d-dimensional simplicial complex living in $\mathbb{R}^{d+1}$, where $f(x)$ is

the "height" of a point $x \in \mathbb{M}$, which is encoded in the representation of $\mathbb{M}$. Specifically, rather than writing our input as $(\mathbb{M}, f)$, we abuse notation and typically just write $\mathbb{M}$ to denote the lifted complex.

Definition 2.2. *The* level set *at value h is the set $\{x \in \mathbb{M} | f(x) = h\}$. A* contour *is a connected component of a level set. An h-*contour *is a contour where f-values are h.*

Note that a contour that does not contain a boundary is itself a simplicial complex of one dimension lower, and is represented (in our algorithms) as such. We let δ and ε denote infinitesimals. Let $B_\varepsilon(x)$ denote a ball of radius ε around x, and let $f|B_\varepsilon(x)$ be the restriction of f to $B_\varepsilon(x)$. The following definition of a critical point is not standard, but is convenient for our presentation.

Definition 2.3. *The* Morse up-degree *of x is the number of $(f(x) + \delta)$-contours of $f|B_\varepsilon(x)$ as $\delta, \varepsilon \to 0^+$. The* Morse down-degree *is the number of $(f(x) - \delta)$-contours of $f|B_\varepsilon(x)$ as $\delta, \varepsilon \to 0^+$.*

A regular *point has both Morse up-degree and down-degree* 1*. A* maximum *has Morse up-degree* 0*, while a* minimum *has Morse down-degree* 0*. A* Morse Join *has Morse up-degree strictly greater than* 1*, while a* Morse Split *has Morse down-degree strictly greater than* 1*. Non-regular points are called* critical*.*

The set of critical points is denoted by $\mathcal{V}(f)$. Because f is piecewise-linear, all critical points are vertices in $\mathbb{M}$. A value h is called *critical*, if $f(v) = h$, for some $v \in \mathcal{V}(f)$. A contour is called *critical*, if it contains a critical point, and it is called *regular* otherwise.

The critical points are where certain topological properties of level sets change. By assuming that our simplicial complex is boundary critical, the vertices on a given boundary are either collectively all maxima or all minima. We abuse notation and refer to this entire set of vertices as a maximum or minimum.

Definition 2.4. *Two regular contours ψ and ψ' are* equivalent *if there exists an f-monotone path p connecting a point in ψ to ψ', such that no $x \in p$ belongs to a critical contour.*

This equivalence relation gives a set of *contour classes*. Every such class maps to intervals of the form $(f(x_i), f(x_j))$, where x_i, x_j are critical points. Such a class is said to be created at x_i and destroyed at x_j.

Definition 2.5. *The* contour tree *is the graph on vertex set $\mathcal{V} = \mathcal{V}(f)$, where edges are formed as follows. For every contour class that is created at v_i and destroyed v_j, there is an edge (v_i, v_j). (Conventionally, edges are directed from higher to lower function value.)*

We denote the contour tree of $\mathbb{M}$ by $\mathcal{C}(\mathbb{M})$. The corresponding node and edge sets are denoted as $\mathcal{V}(\cdot)$ and $\mathcal{E}(\cdot)$. It is not immediately obvious that this graph is a tree, but alternative definitions of the contour tree in [CSA03] imply this is a tree. Since this tree has height values associated with the vertices, we can talk about up-degrees and down-degrees in $\mathcal{C}(\mathbb{M})$.

We assume that the Morse up (or down) degree of any point in $\mathbb{M}$ is at most 2, and that the total Morse degree is at most 3. This immediately implies analogous bounds on the up and down-degrees in $\mathcal{C}(\mathbb{M})$. This is standard in the literature among others, though it can be challenging to enforce in practice. Theoretically, a manifold with points of higher Morse degree can be converted to one with the degree condition through a process of *vertex unfolding* (refer to Section 6.3 in [EH10]). This effectively converts a multi-saddle to a set of ordinary saddles. For 2D manifolds, this conversion does not increase the surface complexity. In higher dimensions, the unfolding can increase complexity if $\mathbb{M}$ has vertices of non-constant degree.

2.1 Some remarks

1) Note that if one intersects $\mathbb{M}$ with a given ball B, then a single contour in $\mathbb{M}$ might be split into more than one contour in the intersection. In particular, two $(f(x)+\delta)$-contours of $f|_{B_\varepsilon(x)}$, given by Definition 2.3, might actually be the same contour in $\mathbb{M}$. Alternatively, one can define the up-degree (as opposed to *Morse* up-degree) as the number of $(f(x)+\delta)$-contours (in the full $\mathbb{M}$) that intersect $B_\varepsilon(x)$, a potentially smaller number. This up-degree is exactly the up-degree of x in $\mathcal{C}(\mathbb{M})$. (Analogously, for down-degree.) When the Morse up-degree is 2 but the up-degree is 1, the topology of the level set changes but not by the number of connected components changing. When $d = 2$, this distinction is not necessary, since any point with Morse degree strictly greater than 1 will have degree strictly greater than 1 in $\mathcal{C}(\mathbb{M})$.

2) As Carr points out in Chapter 6 of his thesis, the term contour tree can be used for a family of related structures. Every regular vertex in $\mathbb{M}$ is associated with an edge in $\mathcal{C}(\mathbb{M})$, and sometimes the vertex is explicitly placed in $\mathcal{C}(\mathbb{M})$ (by subdividing the respective edge). This is referred to as augmenting the contour tree, and it is common to augment $\mathcal{C}(\mathbb{M})$ with all vertices. Alternatively, one can smooth out all vertices of up-degree and down-degree 1 to get the unaugmented contour tree. (For $d = 2$, there are no such vertices in $\mathcal{C}(\mathbb{M})$.) The contour tree of Definition 2.5 is the typical definition in all results on output-sensitive contour trees. Theorem 1.3 is applicable for any augmentation of $\mathcal{C}(\mathbb{M})$ with a predefined set of vertices, though we will not delve into these aspects in this paper.

3 A tour of the new contour tree algorithm

We provide a high-level description of the main ideas of subsequent sections.

Do not globally sort: The starting point for this work is Figure 1. We have two terrains with exactly the same contour tree, but different orderings of (heights of) the critical points. Turning it around, we cannot deduce the full height ordering of critical points from the contour tree, and so sorting all critical points is computationally unnecessary. In Figure 2, the contour tree consists of two balanced binary trees, one of the joins, another of the splits. Again, it is not necessary to know the relative ordering between the mounds on the left (or among the depressions on the right) to compute the contour tree. Yet some ordering information is necessary: on the left, the little valleys are higher than the big central valley, and this is reflected in the contour tree. Leaf paths in the contour tree have points in sorted order, but incomparable points in the tree are unconstrained. How do we sort exactly what is required, without knowing the contour tree in advance?

Join and split trees: The standard contour tree algorithm of Carr, Snoeyink, and Axen [CSA03], builds two different trees, called the join and split trees, and then merges them together to get the contour tree. Consider sweeping down the hyperplane $x_{d+1} = h$ and taking the *superlevel* sets. These are the connected components of the portion of $\mathbb{M}$ above height h. For a terrain, the superlevel sets are a collection of "mounds". As we sweep downwards, these mounds keep joining each other, until finally, we end up with all of $\mathbb{M}$. The join tree, $\mathcal{J}(\mathbb{M})$, tracks exactly these events. Formally, let $\mathbb{M}_v^+$ denote the simplicial complex induced on the subset of vertices higher than v. Refer to Figure 2 for the join tree of a terrain. Note that nothing happens at splits, but these are still put as vertices in the join tree. The split tree is obtained by simply inverting this procedure, sweeping upwards and tracking sublevel sets.

A major insight of [CSA03] is an ingeniously simple linear time procedure to construct the contour tree from the join and split trees. So the bottleneck is computing these trees. Observe in Figure 2 that the split vertices form a long path in the join tree (and vice versa). Therefore, constructing these trees forces a global sort of the splits, an unnecessary computation for the contour tree. Unfortunately, in general (i.e. unlike Figure 2) the heights of joins and splits may be interleaved in a complex manner, and hence the final merging of [CSA03] to get the contour tree requires having the split vertices in the join tree. Without this, it is not clear how to get a consistent view of both joins and splits, required for the contour tree.

Our aim is to break $\mathbb{M}$ into smaller pieces, where this unnecessary computation can be avoided.

Cutting $\mathbb{M}$ into extremum dominant pieces: We define a simplicial complex endowed with a height to be *extremum dominant* if there exists only a single minimum, or only a single maximum. (When formally defined in §6, extremum dominant complexes will allow additional trivial minima or maxima which are a small complication resulting from the following cutting procedure.) We first cut $\mathbb{M}$ into disjoint extremum dominant pieces in linear time. Take an arbitrary maximum x and imagine torrential rain at the maximum. The water flows down, wetting any point that has a non-ascending path from x. We end up with two portions, the wet part of $\mathbb{M}$ and the dry part. This is similar to *watershed* algorithms used for image segmentation [RM00]. The wet part is obviously connected and can be shown to be extremum dominant. We can cut along the interface of the wet and dry, remove the wet part, and recurse on the dry parts. This is carefully done to ensure that the total complexity of the pieces is linear (see §6 for details).

We now have carved $\mathbb{M}$ up into a collection of extremum dominant pieces. In §5 we prove a simple contour surgery theorem allowing us to build the contour tree of $\mathbb{M}$ from the contour trees of these various pieces. Specifically, we argue our cutting in $\mathbb{M}$ was done in such a way that each cut corresponds to cutting only a single edge in the contour tree. Informally this means gluing $\mathbb{M}$ back together along these cuts corresponds to gluing the contour tree together.

Ultimately, the reason for cutting $\mathbb{M}$ into extremum dominant pieces is that such pieces have simpler contour tree structure. Specifically, using ideas from [CSA03], in §7 we prove that the contour tree of an extremum dominant complex is (basically) just the join tree. Thus there is no longer a disconnect between the amount of sorting required for the contour tree versus the join tree, and so the problem of computing contour trees efficiently reduces to that for join trees.

Join trees from painted mountaintops: Our main result is a faster algorithm for join trees. The key idea is *paint spilling*. Start with each maximum having a large can of paint, with distinct colors for each maximum. In arbitrary order, spill paint from each maximum, wait till it flows down, then spill from the next, etc. Paint is viscous, and only flows down edges. *It does not paint the interior of faces with dimension strictly larger than 1.* Furthermore, paints do not mix, so each edge receives a unique color, decided by the first paint to reach it. Note that the raining procedure, used above to carve the input into extremum dominant pieces, cannot be used here as it wets interiors of higher dimensional faces and so raining from each maximum may significantly increase the input complexity (see Figure 4). On the other hand, the interface between two different colors is *not* a contour, and so the join trees of each color class cannot just be simply glued together.

Our algorithm incrementally builds $\mathcal{J}(\mathbb{M})$ from the leaves (maxima) to the root (dominant minimum), by repeatedly merging colors and components together. We say that vertex v is *touched* by color c, if there is a c-colored edge with lower endpoint v. Let us focus on the initial painting, where the colors have 1-1 correspondence with the maxima. Refer to the left part of Figure 3. Consider two sibling leaves ℓ_1, ℓ_2 and their common parent v. The leaves are maxima, and v is a

join that "merges" ℓ_1, ℓ_2. In that case, there are "mounds" corresponding to ℓ_1 and ℓ_2 that merge at a valley v. Suppose this was the entire input, and ℓ_1 was colored blue and ℓ_2 was colored red. Both mounds are colored completely blue or red, while v is touched by both colors. So this indicates that v joins the blue maximum and red maximum in $\mathcal{J}(\mathbb{M})$.

This is precisely how we hope to exploit the information in the painting. We prove later that when some join v has exactly two colors among all incident edges with v as the lower endpoint, the corresponding maxima (of those colors) are exactly the children of v in $\mathcal{J}(\mathbb{M})$. To proceed further, we "merge" the colors red and blue into a new color, purple. In other words, we replace all red and blue edges by purple edges. This indicates that the red and blue maxima have been handled. Imagine flattening the red and blue mounds until reaching v, so that the former join v is now a new maximum, from which purple paint is poured. In terms of $\mathcal{J}(\mathbb{M})$, this is equivalent to removing leaves ℓ_1 and ℓ_2, and making v a new leaf. Alternatively, $\mathcal{J}(\mathbb{M})$ has been constructed up to v, and it remains to determine v's parent. The merging of the colors is not explicitly performed as that would be too expensive; instead we maintain a union-find data structure for that. So the red and blue mounds are not actually recolored purple, and instead say red now points to blue in the union find data structure.

Of course, things are more complicated when there are other mounds. There may be a yellow mound, corresponding to ℓ_3 that joins with the blue mound higher up at some vertex u (see the right part of Figure 3). In $\mathcal{J}(\mathbb{M})$, ℓ_1 and ℓ_3 are sibling leaves, and ℓ_2 is a sibling of some ancestor of these leaves. So we cannot merge red and blue, until yellow and blue merge. Naturally, we use priority queues to handle this. We know u must also be touched by blue, so all critical vertices touched by blue are put in a priority queue keyed by height, and vertices are handled in that order.

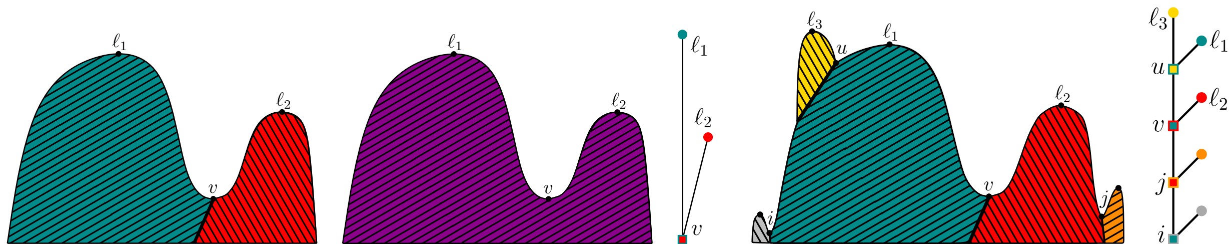

Figure 3: On the left, red and blue merge to make purple, followed by the contour tree with initial colors. On the right, additional maxima and the resulting contour tree.

What happens when finally blue and red join at v? We merge the two colors, but now have blue and red queues of critical vertices, which also need to be merged to get a consistent painting. This necessitates using a priority queue with efficient merges. Specifically, we use *binomial heaps* [Vui78], as they provide logarithmic time merges and deletes (though other similar heaps work). We stress that the feasibility of the entire approach hinges on the use of such an efficient heap structure.

In this discussion, we ignored an annoying problem. Vertices may actually be touched by numerous colors, not just one or two as assumed above. A simple solution would be to insert vertices into heaps corresponding to all colors touching it. But there could be super-constant numbers of copies of a vertex, and handling all these copies would lead to extra overhead. We show that it suffices to simply put each vertex v into at most two heaps, one for each "side" of a possible join. We are guaranteed that when v needs to be processed, all edges incident from above have at most 2 colors, because of all the color merges that previously occurred.

Running time intuition: All non-heap operations can be easily bounded by $O(t\alpha(t) + N)$. One can argue any heap always contains a subset of a single leaf to root path. Bounding each heap by

the size of this path suffices to prove the bound of Theorem 1.1, stated for join trees. However, this may grossly overestimate heap sizes, as each vertex on a path can have at most two colors (the ones getting merged). Thus as we get closer to the root the competition for which two colors a vertex gets assigned to grows, since the number of descendant leaf colors grows. This means for some vertices the size of the heap will be significantly smaller during an associated heap operation.

The intuition is that the paint spilling from the maxima in the simplicial complex, corresponds to paint spilling from the leaves in the join tree, which decomposes the join tree into a set of colored paths. Unfortunately, the situation is more complex since while a given color class is confined to a single leaf to root path, it may *not* appear contiguously on this path, as the right part of Figure 3 shows. Specifically, in this figure the far left saddle (labeled i) is hit by blue paint. However, there is another saddle on the far right (labeled j) which is not hit by blue paint. Since this far right saddle is slightly higher than the far left one, the merge event involving the component containing the blue mound (and also the yellow and red mounds) and corresponding to the far right saddle will occur before the one corresponding to the far left saddle. Hence, the vertices initially touched by blue are not contiguous in the join tree. This non-contiguous complication along with the fact that heap sizes keep changing as color classes merge, causes the analysis to be technically challenging. We employ a variant of *heavy path decompositions*, first used by Sleator and Tarjan for analyzing link/cut trees [ST83]. The final analysis charges expensive heap operations to long paths in the decomposition, resulting in the bound stated in Theorem 1.3.

4 Painting to compute join trees

We now present the main algorithmic contribution, which is a new algorithm for computing join trees of any triangulated simplicial complex $\mathbb{M}$. In subsequent sections we then show how to reduce computing the contour tree to computing the join tree by using contour surgery.

We now formally define the *join* and *split* trees as given by [CSA03]. Conventionally, all edges are directed from higher to lower function value. In the following, for vertex v, we use $\mathbb{M}_v^+$ to denote the simplicial complex obtained by only keeping vertices u such that $f(u) > f(v)$. Analogously, define $\mathbb{M}_v^-$. Note that $\mathbb{M}_v^+$ may contain numerous connected components.

Definition 4.1. *The* join tree $\mathcal{J}(\mathbb{M})$ *of* $\mathbb{M}$ *is built on vertex set* $\mathcal{V}(\mathbb{M})$*. The directed edge* (u, v) *is present when* u *is the smallest valued vertex in a connected component of* $\mathbb{M}_v^+$ and v *is adjacent to a vertex in this component (in* $\mathbb{M}$*). The* split tree $\mathcal{S}(\mathbb{M})$ *is obtained by looking at* $\mathbb{M}_v^-$ *(or alternatively, by taking the join tree of the inversion of* $\mathbb{M}$*).*

Some basic facts about these trees. All outdegrees in $\mathcal{J}(\mathbb{M})$ are at most 1, all indegree 2 vertices are joins, all leaves are maxima, and the global minimum is the root. All indegrees in $\mathcal{S}(\mathbb{M})$ are at most 1, all outdegree 2 vertices are splits, all leaves are minima, and the global maximum is the root. As these trees are rooted, we can use ancestor-descendant terminology. Specifically, for two adjacent vertices u and v, u is the parent of v if u is closer to the root (i.e. each node can have at most one parent, but can have two children).

4.1 Painting

The central tool is a notion of *painting* $\mathbb{M}$. Initially associate a color with each maximum. Imagine there being a large can of paint of a distinct color at each maximum x. We will spill different paint from each maximum and watch it flow down. *So paint only flows down edges, and it does not color the interior of higher dimensional faces.* Furthermore, paints do not mix, so every edge of $\mathbb{M}$ gets

a unique color. This process (and indeed the entire algorithm) works purely on the 1-skeleton of $\mathbb{M}$, which is just a graph.

Definition 4.2. *Let the 1-skeleton of* $\mathbb{M}$ *have edge set* E *and maxima* X*. A* painting *of* $\mathbb{M}$ *is a map* $\chi : X \cup E \to [|X|]$*, where* $\chi(z)$ *is referred to as the* color *of* z*, with the following property. Consider an edge* e*. There exists a descending path from some maximum* x *to* e *consisting of edges in* E*, such that all edges along this path have the same color as* x*.*

An initial *painting also requires that the restriction* $\chi : X \to [|X|]$ *is a bijection.*

Definition 4.3. *Fix a painting* χ *and vertex* v*.*

- *For each maximal connected component* Y *of* $\mathbb{M}_v^+$*, the set of all edges connecting* v *to* Y *is called an* up-star*. (Note* v *has at most two up-stars, as Morse up-degree is at most 2 by assumption.)*
- *A vertex* v *is* touched by color c *if* v *is incident to a* c*-colored edge with* v *at the lower endpoint. For* v*,* $col(v)$ *is the set of colors that touch* v*.*
- *A color* $c \in col(v)$ fully touches v *if all edges in an up-star are colored* c*.*
- *For any maximum* $x \in X$*, we say that* x *is both touched and fully touched by* $\chi(x)$*.*

4.2 The algorithm

We first give an informal, idealized description of the algorithm. That is, in the actual algorithm the details differ slightly and additional data structures are used to improve the running time.

1. Construct a painting of $\mathbb{M}$ by a descending BFS from each maximum. This partitions the edges into colored sets, and assigns each vertex to the color classes of the edges that touch it.
2. For each color, construct a (binomial) max-heap of the critical points touched by that color, indexed by the heights of the critical points.
3. Initialize the join tree to consist of a set of isolated vertices, one for each color class. (We now iteratively fill in the rest of the join tree from the leaves towards the root.)
4. Repeat until all heaps are empty:
   (a) Find a critical point w that is at the top of all heaps containing w.
   (b) Add w to the join tree by connecting it to the lowest vertex of each component of the join tree that contains a maximum whose color matches one of the heaps containing w.
   (c) Delete w from all these heaps, and merge them.

This motivates the definition of *ripe* vertices.

Definition 4.4. *A vertex* v *is* ripe *if: for all* $c \in col(v)$*, if* v *is in the heap of* c *then it is the highest vertex in that heap (using notation from §4.2.1, if* $v \in T(rep(c))$*, then it is highest in* $T(rep(c))$*).*

To efficiently implement the above, we need a number of data structures. Namely, binomial heaps for efficient merges, union-find to avoid explicit recoloring, and a stack to hold unripe (though ripening) vertices, which is used to effectively amortize the cost of the otherwise expensive operation of finding ripe vertices. Note we also later show it suffices to assign each vertex to only a single color from each adjacent up-star, rather than the colors of all the edges which touch it.

4.2.1 The data structures

The binomial heaps $T(c)$: For each color c, $T(c)$ is a subset of vertices touched by c, This is stored as a *binomial max-heap* keyed by vertex heights. Abusing notation, $T(c)$ refers both to the set and the data structure used to store it.

The union-find data structure on colors: We will repeatedly perform unions of classes of colors, and this will be maintained as a standard union-find data structure. For any color c, $rep(c)$ denotes the representative of its class.

Color list $col(v)$: For each vertex v, we maintain $col(v)$ as a simple list. In addition, we will maintain another (sub)list of colors L such that $\forall c \in L$, v is *not* the highest vertex in $T(rep(c))$. Note that given $c \in col(v)$ and $rep(c)$, this property can be checked in constant time. If c is ever removed from this sublist, it can never enter it again.

For notational convenience, we will not explicitly maintain this sublist. We simply assume that, in constant time, one can determine (if it exists) an arbitrary color $c \in col(v)$ such that v is not the highest vertex in $T(rep(c))$.

The stack K: This consists of non-extremal critical points, with monotonically increasing heights as we go from the base to the head.

Attachment vertex $att(c)$: For each color c, we maintain a critical point $att(c)$ of this color. We will maintain the guarantee that the portion of the join tree above (and including) $att(c)$ has already been constructed.

4.2.2 The formal algorithm

We now give a detailed description of the main algorithm. The primary challenge is to use the data structures to rapidly find ripe vertices. A simple example of the algorithm execution is given in §A.

`init`($\mathbb{M}$)

1. Construct an initial painting of $\mathbb{M}$ using a descending BFS from maxima that does not explore previously colored edges.
2. Determine all critical points in $\mathbb{M}$. For each v, look at $(f(v) \pm \delta)$-contours in $f|_{B_\varepsilon(v)}$ to determine the up and down degrees.
3. Mark each critical v as unprocessed.
4. For each critical v and each up-star, pick an arbitrary color c touching v. Insert v into $T(c)$.
5. Initialize $rep(c) = c$ and set $att(c)$ to be the unique maximum colored c.
6. Initialize K to be an empty stack.

`build`($\mathbb{M}$)

1. Run `init`($\mathbb{M}$).
2. While there are unprocessed critical points:
   (a) Run `update`(K). Pop K to get h.
   (b) Let $cur(h) = \{rep(c) | c \in col(h)\}$.
   (c) For all $c' \in cur(h)$:
      i. Add edge $(att(c'), h)$ to $\mathcal{J}(\mathbb{M})$.
      ii. Delete h from $T(c')$.
   (d) Merge heaps $\{T(c') | c' \in cur(h)\}$.
   (e) Take union of $cur(h)$ and denote resulting representative color by $\widehat{c}$.
   (f) Set $att(\widehat{c}) = h$ and mark h as processed.

`update`(K)

1. If K is empty, push arbitrary unprocessed critical point v.
2. Let h be the head of K.
3. While h is not ripe:
   (a) Find $c \in col(h)$ such that h is not the highest in $T(rep(c))$.
   (b) Push the highest of $T(rep(c))$ onto K, and update head h.

A few simple facts:

- At all times, the colors form a valid painting.
- Each vertex is present in at most 2 heaps. This is because the Morse up-degree of every point is at most 2. After processing, it is removed from all heaps.
- After v is processed, all edges incident to v from above have the same (representative) color.
- Vertices on the stack are in increasing height order.

Observation 4.5. Each unprocessed vertex is always in exactly one queue of the colors in each of its up-stars. Specifically, for a given up-star of a vertex v, `init`$(\mathbb{M})$ puts v into the queue of exactly one of the colors of the up-star, say c. As time goes on this queue may merge with other queues, but while v remains unprocessed, it is only ever (and always) in the queue of $rep(c)$, since v is never added to a new queue and is not removed until it is processed. In particular, finding the queues of a vertex in `update`(K) requires at most two union find operations (assuming each vertex records its two colors from `init`$(\mathbb{M})$).

4.3 Proving correctness

Our main workhorse is the following lemma. The *current color* of an edge, e, refers to the value of $rep(\chi(e))$, where $\chi(e)$ is the color of e from the initial painting.

Lemma 4.6. *Suppose vertex v is connected to a component $\mathbb{P}$ of $\mathbb{M}_v^+$ by an edge e which is currently colored c. At all times: either all edges in $\mathbb{P}$ are currently colored c, or there exists a critical vertex $w \in \mathbb{P}$ currently fully touched by c and currently touched by another color.*

Proof. Since e has current color c, there must exist vertices in $\mathbb{P}$ currently touched by c. Consider the highest vertex w in $\mathbb{P}$ that is currently touched by c and some other color. If no such vertex exists, this means all edges incident to a vertex currently touched by c are currently colored c. By walking through $\mathbb{P}$, we deduce that all edges are currently colored c.

So assume w exists. Take the $(f(w)+\delta)$-contour ϕ that intersects $B_\varepsilon(w)$ and intersects some currently c-colored edge incident to w. Note that all edges intersecting ϕ are also currently colored c, since w is the highest vertex to be currently touched by c and some other color. (Take the path of currently c-colored edges from the maximum to w. For any point on this path, the contour passing through this point must be currently colored c.) Hence, c currently fully touches w. But w is currently touched by another color, and the corresponding edge cannot intersect ϕ. So w must have up-degree 2 and is critical. ☐

Corollary 4.7. *Each time* `update`(K) *is called, it terminates with a ripe vertex on top of the stack.*

Proof. `update`(K) is only called if there are unprocessed vertices remaining, and so by the time we reach step 3 in `update`(K), the stack has some unprocessed vertex h on it. If h is ripe, then we are done, so suppose otherwise.

Let $\mathbb{P}$ be one of the components of $\mathbb{M}_h^+$. By construction, h was put in the heap of some initially adjacent color c. Therefore, h must be in the current heap of $rep(c)$ (see Observation 4.5). Now by

Lemma 4.6, either all edges in $\mathbb{P}$ are colored $rep(c)$ or there is some vertex w fully touched by $rep(c)$ and touched by some other color. The former case implies that if there are any unprocessed vertices in $\mathbb{P}$ then they are all in $T(rep(c))$, implying that h is not the highest vertex and a new higher up unprocessed vertex will be put on the stack for the next iteration of the while loop. Otherwise, all the vertices in $\mathbb{P}$ have been processed. However, it cannot be the case that all vertices in all components of $\mathbb{M}_h^+$ have already been processed, since this would imply that h was ripe, and so one can apply the same argument to the other non-fully processed component.

Now consider the latter case, where we have a non-monochromatic vertex w. In this case w cannot have been processed (since after being processed it is touched only by one color), and so it must be in $T(rep(c))$ since it must be in some heap of a color in each up-star (and one up-star is entirely colored $rep(c)$). As w lies above h in $\mathbb{M}$, this implies h is not on the top of this heap. □

Claim 4.8. *Consider a ripe vertex v and take the up-star connecting to some component of $\mathbb{M}_v^+$. All edges in this component and the up-star have the same current color.*

Proof. Let c be the color of some edge in this up-star. By ripeness, v is the highest in $T(rep(c))$. Denote the component of $\mathbb{M}_v^+$ by $\mathbb{P}$. By Lemma 4.6, either all edges in $\mathbb{P}$ are colored $rep(c)$ or there exists critical vertex $w \in \mathbb{P}$ fully touched by $rep(c)$ and touched by another color. In the latter case, w has not been processed, so $w \in T(rep(c))$ (contradiction to ripeness). Therefore, all edges in $\mathbb{P}$ are colored $rep(c)$. □

Claim 4.9. *The partial output on the processed vertices is exactly the restriction of $\mathcal{J}(\mathbb{M})$ to these vertices.*

Proof. More generally, we prove the following: all outputs on processed vertices are edges of $\mathcal{J}(\mathbb{M})$ and for any representative color c, $att(c)$ is the lowest processed vertex of that representative color. We prove this by induction on the processing order. The base case is trivially true, as initially the processed vertices and attachments of the color classes are the set of maxima. For the induction step, consider the situation when v is being processed.

Since v is being processed, we know by Corollary 4.7 that it is ripe. Take any up-star of v, and the corresponding component $\mathbb{P}$ of $\mathbb{M}_v^+$ that it connects to. By Claim 4.8, all edges in $\mathbb{P}$ and the up-star have the same current color, say c. If some critical vertex in $\mathbb{P}$ is not processed, it must be in $T(c)$, which violates the ripeness of v. Thus, all critical vertices in $\mathbb{P}$ have been processed, and so by the induction hypothesis, the restriction of $\mathcal{J}(\mathbb{M})$ to $\mathbb{P}$ has been correctly computed. Additionally, since all critical vertices in $\mathbb{P}$ have been processed, they all have the same representative color c of the lowest critical vertex in $\mathbb{P}$. Thus by the strengthened induction hypothesis, this lowest critical vertex is $att(c)$.

If there is another component of $\mathbb{M}_v^+$, the same argument implies the lowest critical vertex in this component is $att(c')$ (where c' is the current color of edges in the respective component). Now by the definition of $\mathcal{J}(\mathbb{M})$, the critical vertex v connects to the lowest critical vertex in each component of $\mathbb{M}_v^+$, and so by the above v should connect to $att(c)$ and $att(c')$, which is precisely what v is connected to by $\texttt{build}(\mathbb{M})$. Moreover, $\texttt{build}$ merges the representative colors c and c' and correctly sets v to be the attachment, as v is the lowest processed vertex of this merged color (as by induction $att(c)$ and $att(c')$ were the lowest vertices before merging colors). □

Theorem 4.10. *Given an input complex $\mathbb{M}$, $\texttt{build}(\mathbb{M})$ terminates and outputs $\mathcal{J}(\mathbb{M})$.*

Proof. First observe that each vertex can be processed at most once by $\texttt{build}(\mathbb{M})$. By Corollary 4.7, we know that as long as there is an unprocessed vertex, $\texttt{update}(K)$ will be called and will terminate with a ripe vertex which is ready to be processed. Therefore, eventually all vertices will be processed, and so by Claim 4.9 the algorithm will terminate having computed $\mathcal{J}(\mathbb{M})$. □

4.4 Running time

We now bound the running time of the algorithm of §4.2. In subsequent sections, through a sophisticated charging argument, this bound is then related to matching upper and lower bounds in terms of path decompositions. Therefore, it will be useful to set up some terminology that can be used consistently in both places. Specifically, the path decomposition bounds will be purely combinatorial statements on colored rooted trees, and so the terminology is of this form.

Any tree T considered in the following will be a rooted binary tree[1] where the height of a vertex is its distance from the root r (i.e. conceptually T will be a join tree with r at the bottom). As such, the children of a vertex $v \in T$ are the adjacent vertices of larger height (and v is the parent of such vertices). Then the subtree rooted at v, denoted T_v consists of the graph induced on all vertices which are descendants of v (including v itself). For two vertices v and w in T let $d(v, w)$ denote the length of the path between v and w. We use $A(v)$ to denote the set of ancestors of v. For a set of nodes U, $A(U) = \bigcup_{u \in U} A(u)$.

Definition 4.11. *A* leaf assignment χ *of a tree* T *assigns* two *distinct leaves to each internal vertex* v*, one from the left child and one from the right child subtree of* v *(naturally if* v *has only one child it is assigned only one leaf).*

For a vertex $v \in T$, we use H_v to denote the *heap* at v. Formally, $H_v = \{u | u \in A(v), \chi(u) \cap L(T_v) \neq \emptyset\}$, where $L(T_v)$ is the set of leaves of T_v. In words, H_v is the set of ancestors of v which are colored by some leaf in T_v.

Definition 4.12. *The subroutine* `init`$(\mathbb{M})$ *from §4.2 naturally defines a leaf assignment to* $\mathcal{J}(\mathbb{M})$ *according to the priority queue for each up-star we put a given vertex in. Call this the* initial coloring *of the vertices in* $\mathcal{J}(\mathbb{M})$*. This initial coloring also defines the* H_v *values for all* $v \in \mathcal{J}(\mathbb{M})$*.*

The following lemma should justify these definitions.

Lemma 4.13. *Let* $\mathbb{M}$ *be a simplicial complex with* t *critical points. For every vertex in* $\mathcal{J}(\mathbb{M})$*, let* H_v *be defined by the initial coloring of* $\mathbb{M}$*. The total running time of* `build`$(\mathbb{M})$*, including the subroutine call to* `init`$(\mathbb{M})$ *and all calls to* `update`*, is* $O(N + t\alpha(t) + \sum_{v \in \mathcal{J}(\mathbb{M})} \log |H_v|)$*.*

Proof. First we look at the initialization procedure `init`$(\mathbb{M})$. This procedure runs in $O(N)$ time. Indeed, the painting procedure consists of several BFS's but as each vertex is only explored by one of the BFS's, it is linear time overall. Determining the critical points is a local computation on the neighborhood of each vertex and so is linear (i.e. each edge is viewed at most twice). Finally, each vertex is inserted into at most two heaps and so initializing the heaps takes linear time in the number of vertices.

Now consider the union-find operations performed by `build` and `update`. Initially the union find data structure has a singleton component for each leaf (and no new components are ever created), and so each union-find operation takes $O(\alpha(t))$ amortized time. For `update`, by Observation 4.5, each iteration of the while loop requires a constant number of finds (and no unions). Specifically, if a vertex is found to be ripe (and hence processed next) then these can be charged to that vertex. If a vertex is not ripe, then these can be charged to the vertex put on the stack. As each vertex is put on the stack or processed at most once, `update` performs $O(t)$ finds overall. Finally, `build`$(\mathbb{M})$ performs one union and at most two finds for each vertex. Therefore the total number of union find operations is $O(t)$.

[1]Note that technically the trees considered should have a leaf vertex hanging below the root in order to represent the global minimum of the complex. This vertex is (safely) ignored to simplify presentation.

For the remaining operations, observe that for every iteration of the loop in `update`, a vertex is pushed onto the stack and each vertex can only be pushed onto the stack once (since the only way it leaves the stack is by being processed). Therefore the total running time due to `update` is linear (ignoring the find operations).

What remains is the time it takes to process a vertex v in $\texttt{build}(\mathbb{M})$. In order to process a vertex there are a few constant time operations, union-find operations, and queue operations. Therefore the only thing left to bound are the queue operations. Let v be a vertex in $\mathcal{J}(\mathbb{M})$, and let c_1 and c_2 be its children (the same argument holds if v has only one child). At the time v is processed, the colors and queues of all vertices in a given component of $\mathbb{M}_v^+$ have merged together. In particular, when v is processed we know it is ripe and so all vertices above v in each component of $\mathbb{M}_v^+$ have been processed, implying these merged queues are the queues of the current colors of c_1 and c_2. Again since v is ripe, it must be on the top of these queues and so the only vertices left in these queues are those in H_{c_1} and H_{c_2}.

Now when v is handled, three queue operations are performed. Specifically, v is removed from the queues of c_1 and c_2, and then the queues are merged together. By the above arguments the sizes of the queues for each of these operations are $|H_{c_1}|$, $|H_{c_2}|$, and $|H_v|$, respectively. As merging and deleting takes logarithmic time in the heap size for binomial heaps, the claim now follows. □

For any critical point v let ℓ_v denote the length of the longest directed path passing through v in the join tree. As $|H_v|$ only counts vertices in a v to root path, trivially $|H_v| \leq \ell_v$, immediately implying the analog of Theorem 1.1 for join trees. Note however that there is fair amount of slack in this argument as $|H_v|$ may be significantly smaller than ℓ_v. This slack allows for the more refined upper and lower bounds mentioned in §1.1. Quantifying this slack however is quite challenging, and requires a significantly more sophisticated analysis involving path decompositions, which is the subject of §8.

5 Divide and conquer through contour surgery

Now that we have an efficient join tree algorithm, we now describe how to reduce computing the contour tree to computing the join tree, as discussed in §3. Specifically, in this section we first describe our basic cutting tool, which we call contour surgery. In §6 we then describe how we use this tool to cut the surface into so called extremum dominant pieces where computing the contour tree reduces to computing the join tree, and this equivalence is proven in §7. The pieces we create are technically not simplicial complexes, in that each piece may have facets that are not simplices, that is facets of larger size than the dimension. These can easily be completed into simplicial complexes, though as we do not need this, for ease of exposition, we do not explicitly perform this step in our algorithm.

The cutting operation: We define a "cut" operation on $f : \mathbb{M} \to \mathbb{R}$ that cuts along a regular contour to create a new simplicial complex with an added boundary. Given a contour ϕ, roughly speaking, this constructs the simplicial complex $\mathbb{M} \setminus \phi$. We will always enforce the condition that ϕ never passes through a vertex of $\mathbb{M}$. Again, we use ε for an infinitesimally small value. We denote ϕ^+ (resp. ϕ^-) to be the contour at value $f(\phi) + \varepsilon$ (resp. $f(\phi) - \varepsilon$), which is at distance ε from ϕ.

An h-contour is achieved by intersecting $\mathbb{M}$ with the hyperplane $x_{d+1} = h$ and taking a connected component. (Think of the $d+1$-dimension as height.) Given some point x on an h-contour ϕ, we can walk along $\mathbb{M}$ from x to determine ϕ. We can "cut" along ϕ to get a new (possibly) disconnected simplicial complex $\mathbb{M}'$. This is achieved by splitting every face F that ϕ intersects into an "upper" face and "lower" face. Algorithmically, we cut F with ϕ^+ and take everything above ϕ^+ in F to make the upper face. Analogously, we cut with ϕ^- to get the lower face. The faces are then

triangulated to ensure that they are all simplices. This creates the two new boundaries ϕ^+ and ϕ^-, and we maintain the property of constant f-value at a boundary.

Note that by assumption ϕ cannot cut a boundary face, and moreover all non-boundary faces have constant size. Therefore, this process takes time linear in $|\phi|$, the number of faces ϕ intersects. This new simplicial complex is denoted by $\texttt{cut}(\phi, \mathbb{M})$. We now describe a high-level approach to construct $\mathcal{C}(\mathbb{M})$ using this cutting procedure.

> $\texttt{surgery}(\mathbb{M}, \phi)$
> 1. Let $\mathbb{M}' = \texttt{cut}(\mathbb{M}, \phi)$.
> 2. Construct $\mathcal{C}(\mathbb{M}')$ and let A, B be the nodes corresponding to the new boundaries created in $\mathbb{M}'$. (One is a minimum and the other is maximum.)
> 3. Since A, B are leaves, they each have unique neighbors A' and B', respectively. Insert edge (A', B') and delete A, B to obtain $\mathcal{C}(\mathbb{M})$.

Theorem 5.1. *For any regular contour ϕ, the output of $\texttt{surgery}(\mathbb{M}, \phi)$ is $\mathcal{C}(\mathbb{M})$.*

To prove Theorem 5.1, we require a theorem from [Car04] (Theorems 6.6) that map paths in $\mathcal{C}(\mathbb{M})$ to $\mathbb{M}$.

Theorem 5.2. *For every path P in $\mathbb{M}$, there exists a path Q in the contour tree corresponding to the contours passing through points in P. For every path Q in the contour tree, there exists at least one path P in $\mathbb{M}$ through points present in contours involving Q.*

In particular, for every monotone path P in $\mathbb{M}$, there exists a monotone path Q in the contour tree to which P maps, and vice versa.

Theorem 5.1 is a direct consequence of the following lemma.

Lemma 5.3. *Consider a regular contour ϕ contained in a contour class (of an edge of $\mathcal{C}(\mathbb{M})$) (u, v) and let $\mathbb{M}' = \texttt{cut}(\mathbb{M}, \phi)$. Then $\mathcal{V}(\mathcal{C}(\mathbb{M}')) = \{\phi^+, \phi^-\} \cup \mathcal{V}(\mathbb{M})$ and $\mathcal{E}(\mathcal{C}(\mathbb{M}')) = \{(u, \phi^+), (\phi^-, v)\} \cup (\mathcal{E}(\mathbb{M}) \setminus (u, v))$.*

Proof. First observe that since ϕ is a regular contour, the vertex set in the complex $\mathbb{M}'$ is the same as the vertex set in $\mathbb{M}$, except with the addition of the newly created vertices on ϕ^+ and ϕ^-. Moreover, $\texttt{cut}(\mathbb{M}, \phi)$ does not affect the local neighborhood of any vertex in $\mathbb{M}$. Therefore since a vertex being critical is a local condition, with the exception of new boundary vertices, the critical vertices in $\mathbb{M}$ and $\mathbb{M}'$ are the same. Finally, the new vertices on ϕ^+ and ϕ^- collectively behave as a minimum and maximum, respectively, and so $\mathcal{V}(\mathcal{C}(\mathbb{M}')) = \{\phi^+, \phi^-\} \cup \mathcal{V}(\mathbb{M})$.

Now consider the edge sets of the contour trees. Any contour class in $\mathbb{M}'$ (i.e. edge in $\mathcal{C}(\mathbb{M}')$) that does not involve ϕ^+ or ϕ^- is also a contour class in $\mathbb{M}$. Furthermore, a maximal contour class satisfying these properties is also maximal in $\mathbb{M}$. So all edges of $\mathcal{C}(\mathbb{M}')$ that do not involve ϕ^+ or ϕ^- are edges of $\mathcal{C}(\mathbb{M})$. Analogously, every edge of $\mathcal{C}(\mathbb{M})$ not involving ϕ is an edge of $\mathcal{C}(\mathbb{M}')$.

Consider the contour class corresponding to edge (u, v) of $\mathcal{C}(\mathbb{M})$. There is a natural ordering of the contours by function value, ranging from $f(u)$ to $f(v)$. All contours in this class "above" ϕ form a maximal contour class in $\mathbb{M}'$, represented by edge (u, ϕ^+). Analogously, there is another contour class represented by edge (ϕ^-, v). We have now accounted for all contours in $\mathcal{C}(\mathbb{M}')$, completing the proof. ∎

A useful corollary of this lemma shows that a contour actually splits the simplicial complex into two disconnected complexes.

Theorem 5.4. $\texttt{cut}(\mathbb{M}, \phi)$ *consists of two disconnected simplicial complexes, each of which is the triangulation of a d-dimensional ball.*

Proof. Denote (as in Lemma 5.3) the edge containing ϕ to be (u, v). Suppose for contradiction that there is a path between vertices u and v in $\mathbb{M}' = \texttt{cut}(\mathbb{M}, \phi)$. By Theorem 5.2, there is a path in $\mathcal{C}(\mathbb{M}')$ between u and v. Since ϕ^+ and ϕ^- are leaves in $\mathcal{C}(\mathbb{M}')$, this path cannot use their incident edges. Therefore by Lemma 5.3, all the edges of this path are in $\mathcal{E}(\mathcal{C}(\mathbb{M})) \setminus (u, v)$. So we get a cycle in $\mathcal{C}(\mathbb{M})$, a contradiction. To show that there are exactly two connected components in $\texttt{cut}(\mathbb{M}, \phi)$, it suffices to see that $\mathcal{C}(\mathbb{M}')$ has two connected components (by Lemma 5.3) and then applying Theorem 5.2. □

6 Raining to partition $\mathbb{M}$

In this section, we describe a linear time procedure that partitions $\mathbb{M}$ into special *extremum dominant* simplicial complexes.

Definition 6.1. *A simplicial complex with a height function is* minimum dominant *if there exists a minimum x such that every non-minimal* vertex *in the complex has a non-ascending path to x. Analogously define* maximum dominant.

The first aspect of the partitioning is "raining". Start at some point $x \in \mathbb{M}$ and imagine rain at x. The water will flow downwards along non-ascending paths and "wet" all the points encountered. Note that this procedure considers all points of the complex, not just vertices.

Definition 6.2. *Fix $x \in \mathbb{M}$. The set of points $y \in \mathbb{M}$ such that there is a non-ascending path from x to y is denoted by* $\texttt{wet}(x, \mathbb{M})$ *(which in turn is represented as a simplicial complex). A point z is at the* interface *of* $\texttt{wet}(x, \mathbb{M})$ *if every neighborhood of z has non-trivial intersection with* $\texttt{wet}(x, \mathbb{M})$ *(i.e. the intersection is neither empty nor the entire neighborhood).*

The following claim gives a description of the interface.

Claim 6.3. *For any x, each non-empty component of the interface of* $\texttt{wet}(x, \mathbb{M})$ *contains a join vertex.*

Proof. If $p \in \texttt{wet}(x, \mathbb{M})$, all the points in any contour containing p are also in $\texttt{wet}(x, \mathbb{M})$. (Follow the non-ascending path from x to p and then walk along the contour.) The converse is also true, so $\texttt{wet}(x, \mathbb{M})$ contains entire contours.

Let ε, δ be sufficiently small as usual. Fix some y at the interface. Note that $y \in \texttt{wet}(x, \mathbb{M})$. (Otherwise, $B_\varepsilon(y) \cap \mathbb{M}$ is dry.) The points in $B_\varepsilon(y) \cap \mathbb{M}$ that lie below y have a descending path from y and hence must be wet. There must also be a dry point in $B_\varepsilon(y) \cap \mathbb{M}$ that is above y, and hence, there exists a dry, regular $(f(y) + \delta)$-contour ϕ intersecting $B_\varepsilon(y)$.

Let Γ_y be the contour containing y. Suppose for contradiction that $\forall p \in \Gamma_y$, p has up-degree 1 (see Definition 2.3). Consider the non-ascending path from x to y and let z be the first point of Γ_y encountered. There exists a wet, regular $(f(y)+\delta)$-contour ψ intersecting $B_\varepsilon(z)$. Now, walk from z to y along Γ_y. If all points w in this walk have up-degree 1, then ψ is the unique $(f(y)+\delta)$-contour intersecting $B_\varepsilon(w)$. This would imply that $\phi = \psi$, contradicting the fact that ψ is wet and ϕ is dry. □

Note that $\texttt{wet}(x, \mathbb{M})$ (and its interface) can be computed in time linear in the size of the wet simplicial complex. We perform a non-ascending search from x. Any face F of $\mathbb{M}$ encountered

is partially (if not entirely) in $\texttt{wet}(x, \mathbb{M})$. The wet portion is determined by cutting F along the interface. Since each component of the interface is a contour, this is equivalent to locally cutting F by a hyperplane. All these operations can be performed to output $\texttt{wet}(x, \mathbb{M})$ in time linear in $|\texttt{wet}(x, \mathbb{M})|$.

We define a simple `lift` operation on the interface components. Consider such a component ϕ containing a join vertex y. Take any dry increasing edge incident to y, and pick the point z on this edge at height $f(y)+\delta$ (where δ is an infinitesimal, but larger than the value ε used in the definition of `cut`). Let $\texttt{lift}(\phi)$ be the unique contour through the regular point z. Note that $\texttt{lift}(\phi)$ is dry. The following claim follows directly from Theorem 5.4.

Claim 6.4. *Let ϕ be a connected component of the interface. Then* $\texttt{cut}(\mathbb{M}, \texttt{lift}(\phi))$ *results in two disjoint simplicial complexes, one consisting entirely of dry points.*

Proof. By Theorem 5.4, $\texttt{cut}(\mathbb{M}, \texttt{lift}(\phi))$ results in two disjoint simplicial complexes. Let $\mathbb{N}$ be the complex containing the point x (the argument in $\texttt{wet}(x, \mathbb{M})$), and let $\mathbb{N}'$ be the other complex. Any path from x to $\mathbb{N}'$ must intersect $\texttt{lift}(\phi)$, which is dry. Hence $\mathbb{N}'$ is dry. □

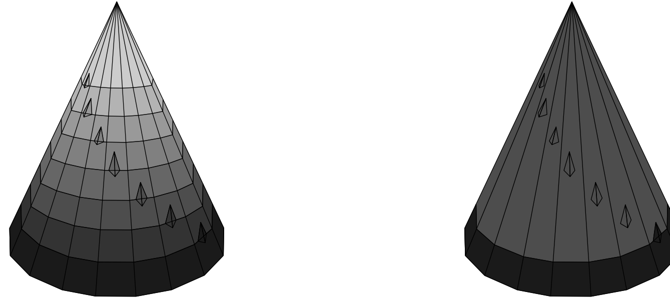

Figure 4: On left, downward rain spilling only (each shade of gray represents a piece created by each different spilling), producing a grid. Note we are assuming raining was done in sorted order of the maxima (i.e. lowest to highest). On right, flipping the direction of rain spilling.

We describe the main partitioning procedure that cuts a simplicial complex $\mathbb{N}$ into extremum dominant complexes, shown below as $\texttt{rain}(x, \mathbb{N})$. It takes an additional input of a maximum x. To initialize, we begin with $\mathbb{N}$ set to $\mathbb{M}$ and x as an arbitrary maximum. When we start, rain flows downwards. In each recursive call, the direction of rain is *switched* to the opposite direction. This is crucial to ensure linear running time, and Figure 4 shows how failing to switch directions can lead to a blow up in complexity. The switching is easily implemented by inverting a complex $\mathbb{N}'$, achieved by negating the height values. We can now let rain flow downwards, as it usually does in our world.

$\texttt{rain}(x, \mathbb{N})$

1. Determine interface of $\texttt{wet}(x, \mathbb{N})$.
2. If the interface is empty, simply output $\mathbb{N}$. Otherwise, denote the connected components by $\phi_1, \phi_2, \ldots, \phi_k$ and set $\phi'_i = \texttt{lift}(\phi_i)$.
3. Initialize $\mathbb{N}_1 = \mathbb{N}$.
4. For i from 1 to k:
   (a) Construct $\texttt{cut}(\mathbb{N}_i, \phi'_i)$, consisting of dry complex $\mathbb{L}_i$ and remainder $\mathbb{N}_{i+1}$.
   (b) Let the newly created boundary of $\mathbb{L}_i$ be B_i. Invert $\mathbb{L}_i$ so that B_i is a maximum. Recursively call $\texttt{rain}(B_i, \mathbb{L}_i)$.
5. Output $\mathbb{N}_{k+1}$ together with any complexes output by recursive calls.

For convenience, denote the total output of $\texttt{rain}(x, \mathbb{M})$ by $\mathbb{M}_1, \mathbb{M}_2, \ldots, \mathbb{M}_r$.

Lemma 6.5. *Each output $\mathbb{M}_i$ is extremum dominant.*

Proof. Consider a call to $\texttt{rain}(x, \mathbb{N})$. If the interface is empty, then all of $\mathbb{N}$ is in $\texttt{wet}(x, \mathbb{N})$, so $\mathbb{N}$ is trivially extremum dominant. So suppose the interface is non-empty and consists of $\phi_1, \ldots, \phi_k$ (as denoted in the procedure). By repeated applications of Claim 6.4, $\mathbb{N}_{k+1}$ contains $\texttt{wet}(x, \mathbb{M})$. Consider $\texttt{wet}(x, \mathbb{N}_{k+1})$. The interface is exactly $\phi_1, \ldots, \phi_k$. So the only dry vertices of $\mathbb{N}_{k+1}$ are those on the boundaries created by calling `cut` for each ϕ_i, but such boundaries are all maxima. □

As $\texttt{rain}(x, \mathbb{M})$ proceeds, new faces/simplices are created because of repeated cutting. The key to the running time of $\texttt{rain}(x, \mathbb{M})$ is bounding the number of newly created faces, for which we have the following lemma.

Lemma 6.6. *A face $F \in \mathbb{M}$ is cut*[2] *at most once during* $\texttt{rain}(x, \mathbb{M})$.

Proof. Notation here follows the pseudocode of `rain`. First, by Theorem 5.4, all the pieces on which `rain` is invoked are disjoint. Second, all recursive calls are made on dry complexes.

Consider the first time that F is cut, say, during the call to $\texttt{rain}(x, \mathbb{N})$. Specifically, say this happens when $\texttt{cut}(\mathbb{N}_i, \phi'_i)$ is constructed. $\texttt{cut}(\mathbb{N}_i, \phi'_i)$ will cut F with two horizontal cutting planes, one ε above ϕ'_i and one ε below ϕ'_i. This breaks F into lower and upper portions which are then triangulated (there is also a discarded middle portion). The lower portion, which is adjacent to ϕ_i, gets included in $\mathbb{N}_{k+1}$, the complex containing the wet points, and hence does not participate in any later recursive calls. The upper portion (call it U) is in $\mathbb{L}_i$. Note that the lower boundary of U is in the boundary B_i. Since a recursive call is made to $\texttt{rain}(B_i, \mathbb{L}_i)$ (and $\mathbb{L}_i$ is inverted), U becomes wet. Hence U, and correspondingly F, will not be subsequently cut. □

The following are direct consequences of Lemma 6.6 and the `surgery` procedure.

Theorem 6.7. *The total running time of* $\texttt{rain}(x, \mathbb{M})$ *is $O(|\mathbb{M}|)$.*

Proof. The only non-trivial operations performed are `wet` and `cut`. Since `cut` is a linear time procedure, Lemma 6.6 implies the total time for all calls to `cut` is $O(|\mathbb{M}|)$. As for the `wet` procedure, observe that Lemma 6.6 additionally implies there are only $O(|\mathbb{M}|)$ new faces created by `rain`. Therefore, since `wet` is also a linear time procedure, and no face is ever wet twice, the total time for all calls to `wet` is $O(|\mathbb{M}|)$. □

Claim 6.8. *Given $\mathcal{C}(\mathbb{M}_1), \mathcal{C}(\mathbb{M}_2), \ldots, \mathcal{C}(\mathbb{M}_r)$, $\mathcal{C}(\mathbb{M})$ can be constructed in $O(|\mathbb{M}|)$ time.*

Proof. Consider the tree of recursive calls in $\texttt{rain}(x, \mathbb{M})$, with each node labeled with some $\mathbb{M}_i$. Walk through this tree in a leaf first ordering. Each time we visit a node we connect its contour tree to the contour tree of its children in the tree using the `surgery` procedure. Each `surgery` call takes constant time, and the total time is the size of the recursion tree. □

7 Contour trees of extremum dominant complexes

The previous section allows us to restrict attention to extremum dominant complexes. We will orient so that the extremum in question is always a *minimum*. We will fix such a simplicial complex $\mathbb{M}$, with the dominant minimum m^*.

[2] What we are calling a single cut is actually done with two hyperplanes.

Recall the definitions of join and split trees from §4. The main theorem of this section asserts that contour trees of minimum dominant complexes have a simple description in terms of $\mathcal{J}(\mathbb{M})$. First we make the key observation that $\mathcal{S}(\mathbb{M})$ is trivial for a minimum dominant $\mathbb{M}$.

Lemma 7.1. *For minimum dominant $\mathbb{M}$, $\mathcal{S}(\mathbb{M})$ consists of:*

- *A single path (in sorted order) with all vertices except non-dominant minima.*
- *Each non-dominant minimum is attached to a unique split (which is adjacent to it).*

Proof. It suffices to prove that each split v has one child that is just a leaf, which is a non-dominant minimum. Specifically, any minimum is a leaf in $\mathcal{S}(\mathbb{M})$ and thereby attached to a split, which implies that if we removed all non-dominant minima, we must end up with a path, as asserted above.

Consider a split v. For sufficiently small ε, δ, there are exactly two $(f(v) - \delta)$-contours ϕ and ψ intersecting $B_\varepsilon(v)$. Both of these are regular contours. There must be a non-ascending path from v to the dominant minimum m^*. Consider the first edge (necessarily decreasing from v) on this path. It must intersect one of the $(f(v) - \delta)$-contours, say ϕ. By Theorem 5.4, $\texttt{cut}(\mathbb{M}, \phi)$ has two connected components, with one (call it $\mathbb{L}$) having ϕ^- as a boundary maximum. This complex contains m^* as the non-ascending path intersects ϕ only once. Let the other component be called $\mathbb{M}'$.

Consider $\texttt{cut}(\mathbb{M}', \psi)$ with connected component $\mathbb{N}$ having ψ^- as a boundary. $\mathbb{N}$ does not contain m^*, so any path from the interior of $\mathbb{N}$ to m^* must intersect the boundary ψ^-. But the latter is a maximum in $\mathbb{N}$, so there can be no non-ascending path from the interior to m^*. Since $\mathbb{M}$ is overall minimum dominant, the interior of $\mathbb{N}$ can only contain a single vertex w, a non-dominant minimum.

The split v has two children in $\mathcal{S}(\mathbb{M})$, one in $\mathbb{N}$ and one in $\mathbb{L}$. The child in $\mathbb{N}$ can only be the non-dominant minimum w, which is a leaf. □

It is convenient to denote the non-dominant minima as $m_1, m_2, \ldots, m_k$ and the corresponding splits (as given by the lemma above) as $s_1, s_2, \ldots, s_k$.

Using the above lemma we can now prove that computing the contour tree for a minimum dominant complex amounts to computing its join tree. Specifically, to prove our main theorem, we rely on the correctness of the merging procedure from [CSA03] that constructs the contour tree from the join and split trees. It actually first constructs the *augmented contour tree* $\mathcal{A}(\mathbb{M})$, which is obtained by replacing each edge in the contour tree with a path of all regular vertices (sorted by height) whose corresponding contour belongs to the equivalence class of that edge.

Consider a tree T with a vertex v of in and out degree at most 1. *Erasing* v from T is the following operation: if v is a leaf, just delete v. Otherwise, delete v and connect its neighbors by an edge (i.e. smooth v out). This tree is denoted by $T \ominus v$. The merging procedure of [CSA03] is shown below as $\texttt{merge}(\mathcal{J}(\mathbb{M}), \mathcal{S}(\mathbb{M}))$. This procedure builds $\mathcal{A}(\mathbb{M})$ by iteratively adding edges that are adjacent to leaves of either $\mathcal{J}(\mathbb{M})$ or $\mathcal{S}(\mathbb{M})$, erasing the corresponding vertex from both $\mathcal{J}(\mathbb{M})$ and $\mathcal{S}(\mathbb{M})$, and repeating. $\mathcal{C}(\mathbb{M})$ is then produced by erasing all regular vertices in $\mathcal{A}(\mathbb{M})$. We use the fact that this procedure correctly produces the contour tree (regardless of the order leaves are processed), and refer the interested reader to [CSA03] for the proof of this fact, as it is not needed for our purposes.

```
merge(𝒥(𝕄), 𝒮(𝕄))
  1. Set 𝒥 = 𝒥(𝕄) and 𝒮 = 𝒮(𝕄).
  2. Denote v as a candidate if the sum of its indegree in 𝒥 and outdegree in 𝒮 is 1.
  3. Add all candidates to queue.
  4. While candidate queue is non-empty:
     (a) Let v be head of queue. If v is leaf in 𝒥, consider its edge in 𝒥. Otherwise
         consider its edge in 𝒮. In either case, denote the edge by (v, w).
     (b) Insert (v, w) in 𝒜(𝕄).
     (c) Set 𝒥 = 𝒥 ⊖ v and 𝒮 = 𝒮 ⊖ v. Enqueue any new candidates.
  5. Erase all regular vertices in 𝒜(𝕄) to get 𝒞(𝕄).
```

Definition 7.2. *The* critical join tree $\mathcal{J}_C(\mathbb{M})$ *is built on the set V' of all critical points other than the non-dominant minima. The directed edge (u, v) is present when u is the smallest valued vertex in V' in a connected component of $\mathbb{M}_v^+$ and v is adjacent (in $\mathbb{M}$) to a vertex in this component.*

Theorem 7.3. *Let $\mathbb{M}$ have a dominant minimum. The contour tree $\mathcal{C}(\mathbb{M})$ consists of all edges $\{(s_i, m_i)\}$ and $\mathcal{J}_C(\mathbb{M})$.*

Proof. We have flexibility in choosing the order of processing in `merge`. We first put the non-dominant minima $m_1, \ldots, m_k$ into the queue. As these are processed, the edges $\{(s_i, m_i)\}$ are inserted into $\mathcal{A}(\mathbb{M})$. Once all the m_i's are erased, $\mathcal{S}$ becomes a path, so all outdegrees are at most 1. Moreover the join tree is now $\mathcal{J}(\mathbb{M}) \ominus \{m_i\}$, and so we can now process $\mathcal{J}$ leaf by leaf, and all remaining edges of $\mathcal{J}$ are inserted into $\mathcal{A}(\mathbb{M})$. Note that smoothing out regular points from $\mathcal{J}(\mathbb{M}) \ominus \{m_i\}$ yields the edges of $\mathcal{J}_C(\mathbb{M})$. Thus after smoothing out all regular points from $\mathcal{A}(\mathbb{M})$ we recover the edges of $\mathcal{J}_C(\mathbb{M})$. Hence the resulting contour tree contains these edges and the $\{(s_i, m_i)\}$ as claimed. □

Remark 7.4. The above theorem, combined with the previous sections, implies that in order to get an efficient contour tree algorithm, it suffices to have an efficient algorithm for computing $\mathcal{J}_C(\mathbb{M})$. Note however that for minimum dominant complexes, converting between $\mathcal{J}_C$ and $\mathcal{J}$ is trivial, as $\mathcal{J}$ is just $\mathcal{J}_C$ with each non-dominant minimum m_i augmented along the edge leaving s_i. (Indeed $\mathcal{J}_C(\mathbb{M})$ was only defined to make the above theorem statement clearer.) Thus to get an efficient contour tree algorithm it suffices to have an efficient algorithm for computing $\mathcal{J}(\mathbb{M})$.

7.1 Putting it all together

We now have all the pieces required to prove Theorem 1.1.

Proof of Theorem 1.1. Consider a critical point v of the initial input complex. By Theorem 5.4 this vertex appears in exactly one of the pieces output by `rain`. As in the Theorem 1.1 statement, let ℓ_v denote the length of the longest directed path passing through v in the contour tree of the input complex, and let ℓ'_v denote the longest directed path passing through v in the join tree of the piece containing v. By Theorem 5.1, ignoring non-dominant extrema introduced from cutting (whose cost can be charged to a corresponding saddle), the join tree on each piece output by `rain` is isomorphic to some connected subgraph of the contour tree of the input complex, and hence $\ell'_v \leq \ell_v$. Moreover, $|H_v|$ only counts vertices in a v to root path and so trivially $|H_v| \leq \ell'_v$, implying Theorem 1.1. □

Ultimately our goal is to prove the more refined upper and lower bounds mentioned in §1.1. To achieve this, in the next section we give a careful analysis of how heap sizes evolve over the course of our join tree algorithm, rather than using the blunt bound $|H_v| \leq \ell'_v$, as done in the above proof.

8 Leaf assignments and path decompositions

In this section, we set up a framework to analyze the time taken to compute a join tree $\mathcal{J}(\mathbb{M})$ (see Definition 4.1), and consequently for contour trees by the above discussion. We adopt all notation already defined in §4.4. From here forward we will often assume binary trees are full binary trees (this assumption simplifies the presentation but is not necessary).

Let χ be some fixed leaf assignment to a rooted binary tree T, which in turn fixes all the heaps H_v. We partition the edges of T into a *path decomposition*, where each set of the partition (a path) is a subset of edges in T with each internal vertex of degree at most 2. This naturally gives a path decomposition. For each internal vertex $v \in T$, add the edge from v to $\arg\max_{v_l, v_r}\{|H_{v_l}|, |H_{v_r}|\}$ where v_l and v_r are the children of v (if $|H_{v_l}| = |H_{v_r}|$ then pick one arbitrarily). This is called the *maximum* path decomposition, denoted by $P_{\max}(T)$.

Our main goal in this section is to prove the following theorem. We use $|p|$ to denote the number of vertices in a path p.

Theorem 8.1. $\sum_{v \in T} \log |H_v| = O(\sum_{p \in P_{\max}(T)} |p| \log |p|)$.

We conclude this section in §8.6 by showing that proving this theorem implies our main result Theorem 1.3.

8.1 Tall and short paths

The paths in $P_{\max}(T)$ naturally define a tree[3] of their own. Specifically, in the original tree T contract each path down to its root. Call the resulting tree the *shrub* of T corresponding to the path decomposition $P_{\max}(T)$. Abusing notation, we simply use $P_{\max}(T)$ to denote the shrub. As a result, we use terms like 'parent', 'child', 'sibling', etc. for paths as well. The shrub gives a handle on the heaps of a path. We use $b(p)$ to denote the *base* of the path, which is the vertex in p closest to root of T. We use $\ell(p)$ to denote the leaf in p. We use H_p to denote $H_{b(p)}$.

We wish to prove $\sum_{v \in T} \log |H_v| = O(\sum_{p \in P_{\max}} |p| \log |p|)$. The simplest approach is to prove $\forall p \in P_{\max}$, $\sum_{v \in p} \log |H_v| = O(|p| \log |p|)$. This is unfortunately not true, which is why we divide paths into two categories.

Definition 8.2. *For $p \in P_{\max}(T)$, p is* short *if $|p| < \sqrt{|H_p|}/100$, and* tall *otherwise.*

The following lemma demonstrates that tall paths can "pay" for themselves.

Lemma 8.3. *If p is tall, $\sum_{v \in p} \log |H_v| = O(|p| \log |p|)$. If p is short, $\sum_{v \in p} \log |H_v| = O(|H_p| \log |H_p|)$.*

Proof. For $v \in p$, $|H_v| \leq |H_p| + |p|$ (as v is a descendant of $b(p)$ along p). Hence, $\sum_{v \in p} \log |H_v| \leq \sum_{v \in p} \log(|H_p| + |p|) = |p| \log(|H_p| + |p|)$. If p is a tall path, then $|p| \log(|H_p| + |p|) = O(|p| \log |p|)$. If p is short, then $|p| \log(|H_p| + |p|) = O(|p| \log |H_p|)$. For short paths, $|p| = O(|H_p|)$. □

There are some short paths that we can also "pay" for. Consider any short path p in the shrub. We will refer to the *tall support chain* of p as the tall ancestors of p in the shrub which have a path to p which does not use any short path (i.e. it is a chain of tall paths adjacent to p).

Definition 8.4. *A short path p is* supported *if at least $|H_p|/100$ vertices v in H_p lie in paths in the tall support chain of p.*

[3] The term 'tree' is the most natural term to use here, but the reader should be careful not to confuse this tree on paths from T with the tree T itself.

Let $\mathcal{L}$ be the set of short paths, $\mathcal{L}'$ be the set of supported short paths, and $\mathcal{H}$ be the set of tall paths given by $P_{\max}(T)$ (notation which will be used for the remainder of this section).

Claim 8.5. $\sum_{p\in\mathcal{L}'} |H_p| \log |H_p| = O(\sum_{q\in\mathcal{H}} \sum_{v\in q} \log |H_v|)$.

Proof. Pick $p \in \mathcal{L}'$. As we traverse the tall support chain of p, there are at least $|H_p|/100$ vertices of H_p that lie in these paths. These are encountered in a fixed order. Let H'_p be the first $|H_p|/200$ of these vertices. When $v \in H'_p$ is encountered, there are $|H_p|/200$ vertices of H_p not yet encountered. Hence, $|H_v| \geq |H_p|/200$. Hence, $|H_p| \log |H_p| = O(\sum_{v\in H'_p} \log |H_v|)$. Since all the vertices lie in tall paths, we can write this as $O(\sum_{q\in\mathcal{H}} \sum_{v\in H'_p\cap q} \log |H_v|)$. Summing over all p, the expression is $\sum_{q\in\mathcal{H}} \sum_{p\in\mathcal{L}'} \sum_{v\in H'_p\cap q} \log |H_v|$.

Consider any $v \in H'_p$. Let S be the set of paths $\widetilde{p} \in \mathcal{L}'$ such that $v \in H'_{\widetilde{p}}$. We now show $|S| \leq 2$ (i.e. it contains at most one path other than p). First observe that any two paths in S must be unrelated (i.e. S is an anti-chain), since paths which have an ancestor-descendant relationship have disjoint tall support chains. However, any vertex v receives exactly one color from each of its two subtrees (in T), and therefore $|S| \leq 2$ since any two paths which share descendant leaves in T (i.e. their heaps are tracking the same color) must have an ancestor-descendant relationship.

In other words, any $\log |H_v|$ appears at most twice in the above triple summation. Hence, we can bound it by $O(\sum_{q\in\mathcal{H}} \sum_{v\in q} \log |H_v|)$. ☐

Corollary 8.6. $\sum_{p\in\mathcal{L}'} \sum_{v\in p} \log |H_v| = O(\sum_{p\in P_{\max}} |p| \log |p|)$

Proof. By applying Lemma 8.3, Claim 8.5, and then Lemma 8.3 again, we have the following.

$$\sum_{p\in\mathcal{L}'} \sum_{v\in p} \log |H_v| = O\left(\sum_{p\in\mathcal{L}'} |H_p| \log |H_p|\right) = O\left(\sum_{p\in\mathcal{H}} \sum_{v\in p} \log |H_v|\right) = O\left(\sum_{p\in P_{\max}} |p| \log |p|\right)$$

☐

8.2 Unsupported shrubs

So far we know that tall paths and supported short paths can be "paid for." We will also be able to pay for unsupported paths, though doing so is much trickier.

Recall that $\mathcal{L}$ denotes the set of short paths, $\mathcal{L}'$ the set of supported short paths, and $\mathcal{H}$ the set of tall paths given by $P_{\max}(T)$. We now describe a partitioning of the unsupported paths into a forest of shrubs. Consider $p \in \mathcal{L} \setminus \mathcal{L}'$, and traverse the chain of ancestors from p. Eventually, we must reach another short path q. (If not, we have reached the root r of $P_{\max}(T)$. Hence, p is supported.) Insert an edge from p to q, so q is the parent of p. This construction leads to the *shrub forest* of $\mathcal{L} \setminus \mathcal{L}'$, denoted $\mathcal{F}$, where all the roots are supported short paths, and the remaining nodes are the unsupported short paths.

Let $\mathcal{U}$ be a shrub in $\mathcal{F}$. For any node q in the shrub $\mathcal{U}$, let $TSC(q)$ denote the set of all vertices of T from all paths in the tall support chain of q. For q in $\mathcal{U}$ we then define the *loss* to be $\lambda(q) = TSC(q) \cap H_q$. Intuitively, if c is a child node of a parent node p in $\mathcal{U}$, then $\lambda(c)$ are the vertices "lost" from the heap of c to its tall support chain, by the time one reaches p.

The following is a key property of unsupported paths which we will use multiple times. It is worth noting that the proof of this lemma is where the construction of $P_{\max}(T)$ enters the picture.

Lemma 8.7. *Let $\mathcal{U}$ denote a connected component of $\mathcal{F}$, and let q be the child of some node p in $\mathcal{U}$. Then $|H_p| \geq \frac{3}{2}|H_q|$.*

Proof. Let $h(q)$ denote the tall path that is a child of p in $P_{\max}(T)$, and an ancestor of q. If no such tall path exists, then by construction p is the parent of q in $P_{\max}(T)$, and the following argument will go through by setting $h(q) = q$.

The chain of ancestors from q to $h(q)$ consists only of tall paths. Since q is unsupported, these paths contain at most $|H_q|/100$ vertices of H_q. Thus, $|H_{h(q)}| \geq 99|H_q|/100$.

Consider the base of $h(q)$, which is a node w in T. Let v denote the sibling of w in T. Their parent is called u. Note that both u and v are nodes in the path p. Now, the decomposition $P_{\max}(T)$ put u and v in the same path p. This implies $|H_v| \geq |H_w|$. Since $|H_u| \geq |H_v| + |H_w| - 2$, $|H_u| \geq 2|H_w| - 2$. Let b be the base of p. We have $|H_p| = |H_b| \geq |H_u| - |p| \geq 2|H_w| - |p| - 2$. Since p is a short path, $|p| < \sqrt{|H_p|}/100$. Applying this bound, we get $|H_p| \geq (2 - \delta)|H_w|$ (for a small constant $\delta > 0$). Since w is the base of $h(q)$, $H_w = H_{h(q)}$. We apply the bound $|H_{h(q)}| \geq 99|H_q|/100$ to get $|H_p| \geq 197|H_q|/100$, implying the lemma. □

Corollary 8.8. *Let $\mathcal{U}$ denote a connected component of $\mathcal{F}$. Let p and d be any two nodes in $\mathcal{U}$ such that d is a descendant of p, and p is not the root of $\mathcal{U}$. Then for any $w \in d$ and $v \in \lambda(p)$, $|H_w| = O(|H_v|)$.*

Proof. By the previous lemma, $|H_w| = O(|H_p|)$. Note that there are at most $|H_p|/100$ nodes in $\lambda(p)$, since these are all in the tall support chain of p (which is unsupported). Thus, for any $v \in \lambda(p)$, $|H_v| \geq 99|H_p|/100$, completing the proof. □

To handle the unsupported paths, for every shrub $\mathcal{U}$ in $\mathcal{F}$ we first prune away what we call buffered sub-shrubs. In the following subsection, we show how to charge away the cost of these sub-shrubs to their tall support chains. In §8.4 we then show how to pay for the remaining pruned shrubs, which is the most difficult part of the argument. Afterwards in §8.5, it is then a straightforward task to put everything together to prove Theorem 8.1.

8.3 Pruning away buffered sub-shrubs

Ultimately our goal in this subsection is to prune away subtrees from $\mathcal{U}$ which can be paid for by charging to their tall support chains (similarly to supported short paths). The remaining portion of the shrub will then be handled in a later subsection. We now formally define this pruning process, which is quite intuitive if one thinks of $\mathcal{U}$ as an actual physical shrub.

Definition 8.9. *Let $\mathcal{U}$ denote a shrub in $\mathcal{F}$. A* pruning *$\mathcal{U}'$ of $\mathcal{U}$ is any subgraph defined by some number of rounds of the following iterative process. Initially set $\mathcal{U}' = \mathcal{U}$. In each iteration select some subtree $\mathcal{U}'_r$ rooted at a node $r \in \mathcal{U}'$ and set $\mathcal{U}' = \mathcal{U}' \setminus \{\mathcal{U}'_r\}$.*

The above definition does not specify which subtrees are pruned, which we now formally define.

Definition 8.10. *Let $\mathcal{U}'$ be a pruning of a shrub of $\mathcal{U}$ of $\mathcal{F}$. Then a rooted subtree $\mathcal{B}$ of $\mathcal{U}'$ is called a* buffered sub-shrub*, if it does not contain the root of $\mathcal{U}'$ (i.e. it is a proper subtree), and the following conditions hold:*

1. $\sum_{p \in \mathcal{B}} |\lambda(p)| > \frac{1}{100} \sum_{p \in \mathcal{B}} |p|$
2. *No rooted subtree of $\mathcal{B}$ satisfies the above inequality (other than $\mathcal{B}$ itself).*

The following is quite straightforward. By pruning from the leaves towards the root, we can remove all buffered sub-shrubs from a given $\mathcal{U}$.

Lemma 8.11. *Any shrub $\mathcal{U}$ can be partitioned into $\{\mathcal{U}', \mathcal{B}_1, \mathcal{B}_2, \dots \mathcal{B}_r\}$ where each $\mathcal{B}_i$ is buffered. Furthermore, no subtree of $\mathcal{U}'$ is buffered.*

Proof. Take any reverse topological order of the nodes in $\mathcal{U}$ (so descendants occur before ancestors). Initialize $\mathcal{U}'$ to $\mathcal{U}$. We process the nodes in order. For node p, let $\mathcal{U}'_p$ denote the rooted subtree of p in $\mathcal{U}'$. We simply check if $\sum_{q \in \mathcal{U}'_p} |\lambda(q)| \geq \sum_{q \in \mathcal{U}'_p} |q|/100$. If so, we prune off $\mathcal{U}'_p$, unless p is the root. (Note that $\mathcal{U}'$ is redefined to be the remainder.)

At the end of this process, we have the desired decomposition. Since we process in reverse topological order, the minimality condition of Definition 8.10 automatically holds. Thus, each pruned sub-shrub is buffered and the remaining $\mathcal{U}'$ has the desired property. □

We prove an important lemma. The cost of heap operations corresponding to a buffered sub-shrub can be charged to tall support chains.

Lemma 8.12. *Let $\mathcal{B}$ be a buffered sub-shrub. Then $\sum_{p \in \mathcal{B}} \sum_{w \in p} \log |H_w| = O(\sum_{p \in \mathcal{B}} \sum_{v \in \lambda(p)} \log |H_v|)$*

Proof. The proof is done by a charging argument. For every $p \in \mathcal{B}$ and $v \in \lambda(p)$, we create 100 tokens. Each of these tokens is labeled $\log |H_v|$, and this is called the "value" of the token. Observe that the sum of token values is, up to a constant factor, the bound we wish to show. The set of tokens created for all $v \in \lambda(p)$ is denoted T_p.

There is a grand total of $100 \sum_{p \in \mathcal{B}} |\lambda(p)|$ tokens, which (by the buffered property of $\mathcal{B}$) is strictly more than $\sum_{p \in \mathcal{B}} |p|$. We will place all these tokens on the nodes in $\mathcal{B}$, using the following process. We reverse topologically sort (descendants before ancestors) the nodes in $\mathcal{B}$. For each p in order, we will place tokens in T_p on the nodes of the subtree $\mathcal{B}_p$. Take an arbitrary token in T_p, and find the smallest (in the ordering) descendant q of p that currently has strictly less than $|q|$ tokens. Place the token on q, and repeat. This is repeated until all descendants q have $|q|$ tokens, or all tokens in T_p are used up. When a node q has $|q|$ tokens, we use "q is full".

Now for a useful claim. *For non-root p, all tokens of T_p are used up.* Suppose not. Consider the first such p in the ordering, and look at the situation when we finish processing T_p. Every descendant q of p must be full. Furthermore, all the tokens on $q \in \mathcal{B}_p$ must have come from some $T_{p'}$, where $p' \in \mathcal{B}_p$. Thus, $\sum_{q \in \mathcal{B}_p} |q| < |\bigcup_{p' \in \mathcal{B}_p} T_{p'}| = 100 \sum_{p' \in \mathcal{B}_p} |\lambda(p')|$. This violates the second condition of buffered shrubs.

Let r be the root of $\mathcal{B}$. We argue that T_r is *not used up*. Suppose the contrary, so T_r is used up. By the previous paragraph, all tokens are used up. But this implies that $\sum_{p \in \mathcal{B}} |p| \geq |\bigcup_{p \in \mathcal{B}} T_p|$ $= 100 \sum_{p \in \mathcal{B}} |\lambda(p)|$, violating the buffered condition.

Since T_r is not used up, it must mean that all p's are full, as r is the root and so T_r tokens can be placed on any path which is not full. Consider any p, and let K_p be the set of tokens placed at p. Each token is labeled $\log |H_v|$, for some $v \in \lambda(\hat{p})$, where $\hat{p}$ is an ancestor of p or p itself (as descendants do not place tokens on ancestors). By Corollary 8.8, for every $w \in p$, $\log |H_w| = O(\log |H_v|)$. Because p is full, there are exactly $|p|$ tokens on p. Thus, $\sum_{w \in p} \log |H_w|$ is $O(V_p)$, where V_p is the sum of token values in K_p. Thus summing over all p in $\mathcal{B}$ completes the proof. Specifically, $\sum_{p \in \mathcal{B}} \sum_{w \in p} \log |H_w| = O(\sum_{p \in \mathcal{B}} V_p) = O(\sum_{p \in \mathcal{B}} \sum_{v \in \lambda(p)} \log |H_v|)$ □

For any fixed vertex v on a tall path, there are at most two unsupported short paths such that v is contained in $\lambda(\cdot)$. Specifically, v is assigned exactly two colors, one from each of its child subtrees in T. Pick one of these two colors, which corresponds to some leaf in T. Any two unsupported short paths which are tracking the color of this leaf, must be ancestor-descendant in $P_{\max}$, however such paths have disjoint tall support chains, and hence only one such path can count v in $\lambda(\cdot)$. (Note this is the same argument as in Claim 8.5.) Thus by summing over all tall support chains we have the following.

Corollary 8.13. *Let $B = \{\mathcal{B}_1, \ldots, \mathcal{B}_k\}$ be any collection of disjoint buffered sub-shrubs from $\mathcal{F}$. Then, $\sum_{i=1}^{k} \sum_{p \in \mathcal{B}_i} \sum_{v \in p} \log |H_v| = O(\sum_{p \in \mathcal{H}} \sum_{v \in p} \log |H_v|)$.*

8.4 Paying for pruned shrubs: the root is everything

We focus attention on the remaining shrub left by Lemma 8.11: this is a shrub $\mathcal{U}$ such that no subtree of $\mathcal{U}$ is buffered. Refer to such a shrub as *buffer-pruned*.

The goal of this section is to prove the following lemma.

Lemma 8.14. *Let $\mathcal{U}'$ denote a buffer-pruned sub-shrub of a shrub in $\mathcal{F}$ and let r be the root of $\mathcal{U}'$. (i) For any $v \in p$ such that $p \in \mathcal{U}'$, $|H_v| = O(|H_r|)$. (ii) $\sum_{p \in \mathcal{U}'} |p| = O(|H_r|)$.*

Lemma 8.14 asserts the root r in $\mathcal{U}'$ pretty much encompasses all sizes and heaps in $\mathcal{U}'$. Observe that part (i) of Lemma 8.14 is immediate, as it follows directly from Lemma 8.7. Proving part (ii) requires much more work.

We need the following technical claim. Let $N(\mathcal{U}')$ denote the set of nodes in $\mathcal{U}'$.

Claim 8.15. *There exists a function $\alpha : N(\mathcal{U}') \mapsto \mathbb{R}^+$ with the following properties. (We use α_p to denote the function value.) Let $A_p = \sum_{q \in \mathcal{U}'_p} \alpha_q$. Then, $A_p - 3 \sum_{q \in \mathcal{U}'_p} |q| \leq |H_p| \leq A_p$.*

Proof. We define α inductively, from leaves upwards. For a leaf ℓ in $\mathcal{U}'$, set $\alpha_\ell = |H_\ell|$. Consider an internal node p with children $q_1, q_2, \ldots, q_k$. The heap H_p is formed by the union of the H_{q_i} heaps, after deleting certain vertices from them, and potentially adding some new vertices. The vertices in p and those in the $\lambda(q_i)$ are deleted. Define α_p to be the number of new vertices added to H_p.

Thus, we have $|H_p| = \sum_{i \leq k} |H_{q_i}| + \alpha_p - \#$ vertices deleted from H_{q_i}'s. The number of vertices deleted is at most $2(|p| + \sum_{i \leq k} |\lambda(q_i)|)$. There is a factor 2 because each vertex appears in at most 2 heaps.

It is evident that $|H_p| \leq \sum_{q \in \mathcal{U}'_p} \alpha_q = A_p$. Note that $|H_p| \geq A_p - 2 \sum_{q \in \mathcal{U}'_p, q \neq p} |\lambda(q)| - 2 \sum_{q \in \mathcal{U}'_p} |q|$. Crucially, since $\mathcal{U}'$ is buffer-pruned, $\sum_{q \in \mathcal{U}'_p, q \neq p} |\lambda(q)| \leq \sum_{q \in \mathcal{U}'_p, q \neq p} |q|/100 \leq \sum_{q \in \mathcal{U}'_p} |q|/100$. This completes the proof. □

We have the main charging claim. This assumes that A_p is increasing exponentially as we go up the tree. That property is proven later by induction, using the following claim for sub-shrubs.

Claim 8.16. *Fix any path $p \in \mathcal{U}' \setminus r$. Suppose for any $q, q' \in \mathcal{U}'_p$ where q is a parent of q' in $\mathcal{U}'_p$, $A_q \geq (5/4) A_{q'}$. Then $\sum_{q \in \mathcal{U}'_p} |q| \leq A_p/20$.*

Proof. Since q is an unsupported short path, $|q| < \sqrt{|H_q|}/100 \leq \sqrt{|A_q|}/99$. We prove that $\sum_{q \in \mathcal{U}'_p} \sqrt{|A_q|}/99 \leq A_p/20$ by a charge redistribution scheme. Assume that each $q \in \mathcal{U}'_p$ starts with $\sqrt{A_q}/99$ units of charge. We redistribute this charge over all nodes in $\mathcal{U}'_q$, and then calculate the total charge. For $q \in \mathcal{U}'_p$, spread its charge to all nodes in $\mathcal{U}'_q$ proportional to α values. In other words, give $(\sqrt{A_q}/99) \cdot (\alpha_{q'}/A_q)$ units of charge to each $q' \in \mathcal{U}'_q$.

After the redistribution, let us compute the charge deposited at q. Every ancestor in $\mathcal{U}'_p$, denoted $a_0, a_1, a_2, \ldots, a_k$, contributes to the charge at q. The charge is expressed in the following equation. We use the assumption that $A_{a_i} \geq (5/4) A_{a_{i-1}}$ and hence $A_{a_i} \geq (5/4)^i A_{a_0} \geq (5/4)^i$, as a_0 is an unsupported short path and hence $A_{a_0} \geq 1$.

$$(\alpha_q/99) \sum_{a_i} 1/\sqrt{A_{a_i}} \leq (\alpha_q/99) \sum_{a_i} (4/5)^{i/2} \leq \alpha_q/20$$

The total charge is $\sum_{q \in \mathcal{U}'_p} \alpha_q t/20 = A_p/20$. □

Claim 8.17. *Fix any path $p \in \mathcal{U}' \setminus r$. Let q be any path in $\mathcal{U}'_p$, then for any child path q' of q it holds that $A_q \geq (5/4)A_{q'}$*

Proof. We prove this by induction over the depth of q. When q is a leaf, this is vacuously true. Consider some internal node q, with children $q'_1, q'_2, \ldots, q'_k$. Applying the induction hypothesis to the subshrubs rooted at q'_i, we can invoke Claim 8.16 on each subshrub $\mathcal{U}'_{q'_i}$. Thus, for all i, $\sum_{\hat{q} \in \mathcal{U}'_{q'_i}} |\hat{q}| \leq A_{q'_i}/20$. By Claim 8.15, $|H_{q'_i}| \geq 17A_{q'_i}/20$ and $A_q \geq |H_q|$. By Lemma 8.7, $|H_q| \geq (3/2)|H_{q'_i}|$. We combine these bounds to complete the proof. □

The proof of Lemma 8.14 just combines all these claims, and essentially repeats the argument in the previous claim. We apply Claim 8.17 and Claim 8.16 to every child of the root r. Denoting the children as $q_1, q_2, \ldots$, we get that $\sum_{p \in \mathcal{U}'_{q_i}} |p| \leq A_{q_i}/20$. Summing this over the children, $\sum_{p \in \mathcal{U}', p \neq r} |p| \leq A_r/20$. By Claim 8.15, $|H_r| \geq 17A_r/20 - 3|r|$. Since r represented a short path, we have the bound $|r| \leq \sqrt{|H_r|}/100$. Thus, $|H_r| = \Omega(A_r)$. We combine these bounds to show that $\sum_{p \in \mathcal{U}'} |p| = O(|H_r|)$.

8.5 Proving Theorem 8.1

We split the summation into tall, supported short, and unsupported short paths.

$$\sum_{p \in P_{\max}(T)} \sum_{v \in p} \log |H_v| \quad = \quad \sum_{p \in \mathcal{L} \setminus \mathcal{L}'} \sum_{v \in p} \log |H_v| + \sum_{p \in \mathcal{L}'} \sum_{v \in p} \log |H_v| + \sum_{p \in \mathcal{H}} \sum_{v \in p} \log |H_v|$$

By Corollary 8.6 and Lemma 8.3, the second and last term can be bounded by $O(\sum_{p \in P_{\max}(T)} |p| \log |p|)$. The following claim applies the results of the previous two subsections to give the same bound on the first term, thus completing the proof of Theorem 8.1.

Claim 8.18. $\sum_{p \in \mathcal{L} \setminus \mathcal{L}'} \sum_{v \in p} \log |H_v| = O(\sum_{p \in P_{\max}} |p| \log |p|)$.

Proof. Apply Lemma 8.11 to $\mathcal{U}$ to get the family of buffered sub-shrubs $B(\mathcal{U})$ and buffer-pruned $\mathcal{U}'$.

$$\begin{aligned}
\sum_{p \in \mathcal{L} \setminus \mathcal{L}'} \sum_{v \in p} \log |H_v| \leq \sum_{\mathcal{U} \in \mathcal{F}} \sum_{p \in \mathcal{U}} \sum_{v \in p} \log |H_v| &= \sum_{\mathcal{U} \in \mathcal{F}} \sum_{p \in \mathcal{U}'} \sum_{v \in p} \log |H_v| + \sum_{\mathcal{U} \in \mathcal{F}} \sum_{\mathcal{B} \in B(\mathcal{U})} \sum_{p \in \mathcal{B}} \sum_{v \in p} \log |H_v| \\
&= \sum_{\mathcal{U} \in \mathcal{F}} \sum_{p \in \mathcal{U}'} \sum_{v \in p} \log |H_v| + O\left(\sum_{p \in \mathcal{H}} \sum_{v \in p} \log |H_v|\right) \\
&= \sum_{\mathcal{U} \in \mathcal{F}} \sum_{p \in \mathcal{U}'} \sum_{v \in p} \log |H_v| + O\left(\sum_{p \in P_{\max}} |p| \log |p|\right),
\end{aligned}$$

where the last two lines follow from Corollary 8.13 and Lemma 8.3, respectively.

So what remains is to bound the first term. By Lemma 8.14, $|H_v| = O(|H_r|)$, where r is the root of $\mathcal{U}'$, and furthermore, $\sum_{p \in \mathcal{U}'} |p| = O(|H_r|)$. We have $\sum_{p \in \mathcal{U}'} \sum_{v \in p} \log |H_v| = O((\log |H_r|) \sum_{p \in \mathcal{U}'} |p|) = O(|H_r| \log |H_r|)$. Note that the roots in the shrub forest are supported short paths, and hence when we sum this over all $\mathcal{U}$ in the shrub forest we get,

$$\sum_{\mathcal{U} \in \mathcal{F}} O\left(|H_{r(\mathcal{U}')}| \log |H_{r(\mathcal{U}')}|\right) = \sum_{p \in \mathcal{L}'} O(|H_p| \log |H_p|) = O\left(\sum_{p \in P_{\max}} |p| \log |p|\right),$$

where the last equality follows by Claim 8.5 (via the same argument as in Corollary 8.6). □

8.6 Our main result

We now show that Theorem 8.1 allows us to upper bound the running time for our join tree and contour tree algorithms in terms of path decompositions.

Theorem 8.19. *Let $f : \mathbb{M} \to \mathbb{R}$ be the linear interpolant over distinct valued vertices, where the Morse degree of any point is at most 3. There is an algorithm to compute the join tree whose running time is $O(\sum_{p \in P_{\max}(\mathcal{J})} |p| \log |p| + t\alpha(t) + N)$.*

Proof. By Theorem 4.10 we know that `build`($\mathbb{M}$) correctly outputs $\mathcal{J}(\mathbb{M})$, and by Lemma 4.13 we know this takes $O(\sum_{v \in \mathcal{J}(\mathbb{M})} \log |H_v| + t\alpha(t) + N)$ time, where the H_v values are determined as in Definition 4.12. Therefore by Theorem 8.1, `build`($\mathbb{M}$) takes $O(\sum_{p \in P_{\max}(\mathcal{J})} |p| \log |p| + t\alpha(t) + N)$ time to correctly compute $\mathcal{J}(\mathbb{M})$. □

This result for join trees easily implies our main result, Theorem 1.3, which we now restate and prove.

Theorem 8.20. *Let $f : \mathbb{M} \to \mathbb{R}$ be the linear interpolant over distinct valued vertices, where the Morse degree of any point is at most 3. There is an algorithm to compute $\mathcal{C}$ whose running time is $O(\sum_{p \in P(\mathcal{C})} |p| \log |p| + t\alpha(t) + N)$, where $P(\mathcal{C})$ is a specific path decomposition (constructed implicitly by the algorithm).*

Proof. First, lets review the various pieces of our algorithm. On a given input simplicial complex, we first break it into extremum dominant pieces using `rain`($\mathbb{M}$) (and in $O(|\mathbb{M}|)$ time by Theorem 6.7). Specifically, Lemma 6.5 proves that the output of `rain`($\mathbb{M}$) is a set of extremum dominant pieces, $\mathbb{M}_1, \dots, \mathbb{M}_k$, and Claim 6.8 shows that given the contour trees, $\mathcal{C}(\mathbb{M}_1), \dots, \mathcal{C}(\mathbb{M}_k)$, the full contour tree, $\mathcal{C}(\mathbb{M})$, can be constructed (in $O(|\mathbb{M}|)$ time).

Now one of the key observations was that for extremum dominant complexes, computing the contour tree is roughly the same as computing the join tree. Specifically, Theorem 7.3 implies that given $\mathcal{J}_C(\mathbb{M}_i)$, we can obtain $\mathcal{C}(\mathbb{M}_i)$ by simply sticking on the non-dominant minima at their respective splits (which can easily be done in linear time). Remark 7.4 implies that $\mathcal{J}_C(\mathbb{M}_i)$ is trivially obtained from the $\mathcal{J}(\mathbb{M}_i)$, and by the above theorem we know $\mathcal{J}(\mathbb{M}_i)$ can be computed in $O(\sum_{p \in P_{\max}(\mathcal{J}(\mathbb{M}_i))} |p| \log |p| + t_i\alpha(t_i) + N_i)$ (where t_i and N_i are the number of critical points and faces when restricted to $\mathbb{M}_i$).

At this point we can now see what the path decomposition referenced in theorem statement should be. It is just the union of all the maximum path decompositions across the extremum dominant pieces, $P_{\max}(\mathcal{C}(\mathbb{M})) = \cup_{i=1}^{k} P_{\max}(\mathcal{J}(\mathbb{M}_i))$. Since all procedures besides computing the join trees take linear time in the size of the input complex, we can therefore compute the contour tree in time

$$O\left(N + \sum_{i=1}^{k} \left(\sum_{p \in P_{\max}(\mathcal{J}(\mathbb{M}_i))} |p| \log |p| \right) + t_i\alpha(t_i) + N_i \right) = O\left(\left(\sum_{p \in P_{\max}(\mathcal{C}(\mathbb{M}))} |p| \log |p| \right) + t\alpha(t) + N \right)$$

□

9 Lower bound by path decomposition

We first prove a lower bound for join trees, and then generalize to contour trees. Note that the form of the theorem statements in this section differ from Theorem 1.4, as they are stated directly in terms of path decompositions. Theorem 1.4 is an immediate corollary of the final theorem of this section, Theorem 9.7.

9.1 Join trees

We focus on terrains, so $d = 2$. Consider any path decomposition P of a valid join tree (i.e. any rooted binary tree). When we say "compute the join tree", we require the join tree to be labeled with the corresponding vertices of the terrain.

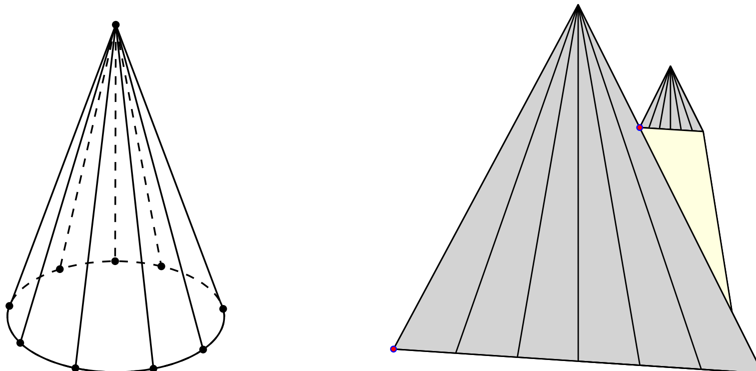

Figure 5: Left: angled view of a tent / Right: a parent and child tent put together

Lemma 9.1. *Fix any path decomposition P. There is a family of terrains, $\mathbf{F}_P$, all with the same triangulation, such that $|\mathbf{F}_P| = \Pi_{p_i \in P}(|p_i| - 1)!$, and no two terrains in $\mathbf{F}_P$ define the same labeled join tree.*

Proof. We describe the basic building block of these terrains, which corresponds to a fixed path $p \in P$. Informally, a *tent* is an upside down cone with m triangular faces (see Figure 5). Construct a slightly tilted cycle of length m with the two antipodal points at heights 1 and 0. These are called the anchor and trap of the tent, respectively. The remaining $m-2$ vertices are evenly spread around the cycle and heights decrease monotonically when going from the anchor to the trap. Next, create an apex vertex at some appropriately large height, and add an edge to each vertex in the cycle. Note that in order to obtain a terrain, a cycle here means the vertices lie on a circle but are connected with straight line segments rather than by the curved edges of the circle.

Now we describe how to attach two different tents. In this process, we glue the base of a scaled down "child" tent on to a triangular cone face of the larger "parent" tent (see Figure 5). Specifically, the anchor of the child tent is attached directly to a face of the parent tent at some height h. The remainder of the base of the child cone is then extended down (at a slight angle) until it hits the face of the parent. Note that this gluing process introduces some non-triangular faces, and so these must be triangulated in order to maintain a proper triangulated terrain.

The full terrain is obtained by repeatedly gluing tents. For each path $p_i \in P$, we create a tent of size $|p_i| + 1$. The two faces adjacent to the anchor are always empty, and the remaining faces are for gluing on other tents. (Note that tents have size $|p_i| + 1$ since $|p_i| - 1$ faces represent the joins of p_i, the apex represents the leaf, and we need two empty faces next to the anchor.) Now we glue together tents of different paths in the same way the paths are connected in the shrub $P_{\mathcal{S}}$ (see §8.1). Specially, for two paths $p, q \in P$ where p is the parent of q in $P_{\mathcal{S}}$, we glue q onto a face of the tent for p as described above. (Naturally for this construction to work, tents for a given path will be scaled down relative to the size of the tent of their parent). By varying the heights of the gluing, we get the family of terrains.

Observe now that the only saddle points in this construction are the anchor points. Moreover, the only maxima are the apexes of the tents. We create a global boundary minimum by setting the

vertices at the base of the tent representing the root of $P_{\mathcal{S}}$ all to the same height (and there are no other minima). Therefore, the saddles on a given tent will appear contiguously on a root to leaf path in the join tree of the terrain, where the leaf corresponds to the maximum of the tent (since all these saddles have a direct line of sight to this apex). In particular, this implies that, regardless of the heights assigned to the anchors, the join tree has a path decomposition whose corresponding shrub is equivalent to $P_{\mathcal{S}}$.

There is a valid instance of this described construction for any relative ordering of the heights of the saddles on a given tent. In particular, there are $(|p_i|-1)!$ possible orderings of the heights of the saddles on the tent for p_i, and hence $\Pi_{p_i \in P}(|p_i|-1)!$ possible terrains we can build. Each one of these functions will result in a different (labeled) join tree. All saddles on a given tent will appear in sorted order in the join tree. So, any permutation of the heights on a given tent corresponds to a permutation of the vertices along a path in P. □

Lemma 9.2. *For all $\mathbb{M} \in \mathbf{F}_P$, the total number of heap operations performed by* `build`($\mathbb{M}$) *is* $O(\sum_{p \in P} |p| \log |p|)$.

Proof. The primary "non-determinism" of the algorithm is the initial painting constructed by `init`($\mathbb{M}$). We show that regardless of how paint spilling is done, the number of heap operations is bounded as above.

Consider an arbitrary order of the initial paint spilling over the surface. Consider any join on a face of some tent, which is the anchor point of some connecting child tent. The join has two up-stars, each of which has exactly one edge. Each edge connects to a maximum and must be colored by that maximum. Hence, the two colors touching this join (according to Definition 4.12) are the colors of the apexes of the child and parent tent.

Take any join v, with two children w_1 and w_2. Suppose w_1 and v belong to the same path in the decomposition. The key is that any color from a maximum in the subtree at w_2 cannot touch any ancestor of v. This subtree is exactly the join tree of the child tent attached at v. The base of this tent is completely contained in a face of the parent tent. So all colors from the child "drain off" to the base of the parent, and do not touch any joins on the parent tent.

Hence, $|H_v|$ is at most the size of the path in P containing v. By Lemma 4.13, the total number of heap operations is at most $\sum_v \log |H_v|$, completing the proof.

□

The following is the equivalent of Theorem 1.4 for join trees, and immediately follows from the previous lemmas. Note that the phrasing of the two theorems differ, but can be connected by observing that for any value of γ from Theorem 1.4, there is a path decomposition P of some tree T such that $\gamma = \Theta(\sum_{p \in P} |p| \log |p|)$.

Theorem 9.3. *Consider a rooted tree T and an arbitrary path decomposition P of T. There is a family $\mathbf{F}_P$ of terrains such that any algebraic decision tree computing the join tree[4] (on $\mathbf{F}_P$) requires $\Omega(\sum_{p \in P} |p| \log |p|)$ time. Furthermore, our algorithm makes $O(\sum_{p \in P} |p| \log |p|)$ comparisons on all these instances.*

Proof. The proof is a basic entropy argument. Any algebraic decision tree that is correct on all of $\mathbf{F}_P$ must distinguish all inputs in this family. By Stirling's approximation, the depth of this tree is $\Omega(\sum_{p_i \in P} |p_i| \log |p_i|)$. Lemma 9.2 completes the proof. □

[4]Note that for the referenced family of terrains, the join tree and contour tree are equivalent

9.2 Contour trees

We first generalize previous terms to the case of contour trees. In this section T will denote an arbitrary contour tree with every internal vertex of degree 3.

For simplicity we now restrict our attention to path decompositions consistent with the raining procedure described in §6 (more general decompositions can work, but it is not needed for our purposes).

Definition 9.4. *A path decomposition, $P(T)$, is called* rain consistent *if its paths can be obtained as follows. Perform an downward BFS from an arbitrary maximum v in T, and mark all vertices encountered. Now recursively run a directional BFS from all vertices adjacent to the current marked set. Specifically, for each BFS run, make it an downward BFS if it is at an odd height in the recursion tree and upward otherwise.*

This procedure partitions the vertex set into disjoint rooted subtrees of T, based on which BFS marked a vertex. For each such subtree, now take any partition of the vertices into leaf paths.[5]

The following is analogous to Lemma 9.1, and in particular uses it as a subroutine.

Lemma 9.5. *Let P be any rain consistent path decomposition of some contour tree. There is a family of terrains, $\mathbf{F}_P$, all with the same triangulation, such that the size of $\mathbf{F}_P$ is $\Pi_{p_i \in P}(|p_i|-1)!$, and no two terrains in $\mathbf{F}_P$ define the same contour tree.*

Proof. As P is rain consistent, the paths can be partitioned into sets $P_1, \ldots, P_k$, where P_i is the set of all paths with vertices from a given BFS, as described in Definition 9.4. Specifically, let T_i be the subtree of T corresponding to P_i and let r_i be the root vertex of this subtree. Note that the P_i sets naturally define a tree where P_i is the parent of P_j if r_i (i.e. the root of T_i) is adjacent to a vertex in P_j.

As the set P_i is a path decomposition of a rooted binary tree T_i, the terrain construction of Lemma 9.1 for P_i is well defined. Actually the only difference is that here the rooted tree is not a full binary tree, and so some of the (non-achor adjacent) faces of the constructed tents will be blank. Specifically, these blank faces correspond to the adjacent children of P_i, and they tell us how to connect the terrains of the different P_i's.

So for each P_i construct a terrain as described in Lemma 9.1. Now each T_i is (roughly speaking) a join or a split tree, depending on whether the BFS which produced it was an upward or downward BFS, respectively. As the construction in Lemma 9.1 was for join trees, each terrain we constructed for a P_i which came from a split tree, must be flipped upside down. Now we must described how to glue the terrains together.

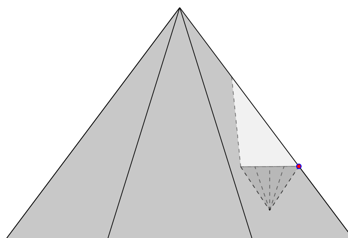

Figure 6: A child tent attached to a parent tent with opposite orientation.

[5] Note that the subtree of the initial vertex is rooted at a maximum. For simplicity we require that the path this vertex belongs to also contains a minimum.

By construction, the tents corresponding to the paths in P_i are connected into a tree structure (i.e. corresponding to the shrub of P_i). Therefore the bottoms of all these tents are covered except for the one corresponding to the path containing the root r_i. If r_i corresponds to the initial maximum that the rain consistent path decomposition was defined from, then this will be flat and corresponds to the global outer face. Otherwise, P_i has some parent P_j in which case we connect the bottom of the tent for r_i to a free face of a tent in the construction for P_j, specifically, the face corresponding to the vertex in T which r_i is adjacent to. This gluing is done in the same manner as in Lemma 9.1, attaching the anchor for the root of P_i directly the corresponding face of P_j, except that now P_i and P_j have opposite orientations. See Figure 6.

Just as in Lemma 9.1 we now have one fixed terrain structure, such that each different relative ordering of the heights of the join and split vertices on each tent produces a surface with a distinct contour tree. The specific bound on the size of $\mathbf{F}_P$, defining these distinct contour trees, follows by applying the bound from Lemma 9.1 to each P_i. □

Lemma 9.6. *For all $\mathbb{M} \in \mathbf{F}_P$, the number of heap operations is $\Theta(\sum_{p \in P} |p| \log |p|)$*

Proof. This lemma follows immediately from Lemma 9.2. The heap operations can be partitioned into the operations performed in each P_i. Apply Lemma 9.2 to each of the P_i separately and take the sum. □

We now restate Theorem 1.4, which follows immediately from an entropy argument, analogous to Theorem 9.3. Again, as discussed above for Theorem 9.3, the phrasing differs from Theorem 1.4.

Theorem 9.7. *Consider any rain consistent path decomposition P. There exists a family $\mathbf{F}_P$ of terrains ($d = 2$) with the following properties. Any contour tree algorithm makes $\Omega(\sum_{p \in P} |p| \log |p|)$ comparisons in the worst case over $\mathbf{F}_P$. Furthermore, for any terrain in $\mathbf{F}_P$, our algorithm makes $O(\sum_{p \in P} |p| \log |p|)$ comparisons.*

Remark 9.8. Note that for the terrains described in this section, the number of critical points is within a constant factor of the total number of vertices. In particular, for this family of terrains, all previous algorithms required $\Omega(n \log n)$ time.

Acknowledgements. We thank Hsien-Chih Chang, Jeff Erickson, and Yusu Wang for numerous useful discussions. This work is supported by the Laboratory Directed Research and Development (LDRD) program of Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. Work on this paper by Benjamin Raichel was also partially supported by NSF CRII Award 1566137.

References

[BKO+98] C. Bajaj, M. van Kreveld, R. W. van Oostrum, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. Technical Report UU-CS-1998-25, Department of Information and Computing Sciences, Utrecht University, 1998.

[BR63] R. Boyell and H. Ruston. Hybrid techniques for real-time radar simulation. In *Proceedings of Fall Joint Computer Conference*, pages 445–458, 1963.

[BWH+11] K. Beketayev, G. Weber, M. Haranczyk, P.-T. Bremer, M. Hlawitschka, and B. Hamann. Visualization of topology of transformation pathways in complex chemical systems. In *Computer Graphics Forum (EuroVis 2011)*, pages 663–672, 2011.

[BWP+10] P.-T. Bremer, G. Weber, V. Pascucci, M. Day, and J. Bell. Analyzing and tracking burning structures in lean premixed hydrogen flames. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):248–260, 2010.

[BWT+11] P.-T. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. Bell. Analyzing and tracking burning structures in lean premixed hydrogen flames. *IEEE Transactions on Visualization and Computer Graphics*, 17(9):1307–1325, 2011.

[Car04] H. Carr. *Topological Manipulation of Isosurfaces.* PhD thesis, University of British Columbia, 2004.

[CLLR05] Y. Chiang, T. Lenz, X. Lu, and G. Rote. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Computational Geometry: Theory and Applications*, 30(2):165–195, 2005.

[CMEH+03] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Loops in reeb graphs of 2-manifolds. In *Proceedings of the Symposium on Computational Geometry (SoCG)*, pages 344–350, 2003.

[CSA03] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry: Theory and Applications*, 24(2):75–94, 2003.

[DN09] H. Doraiswamy and V. Natarajan. Efficient algorithms for computing reeb graphs. *Computational Geometry: Theory and Applications*, 42:606–616, 2009.

[DN13] H. Doraiswamy and V. Natarajan. Computing reeb graphs as a union of contour trees. *IEEE Transactions on Visualization and Computer Graphics*, 19(2):249–262, 2013.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer, 1987.

[EH10] H. Edelsbrunner and J. Harer. *Computational Topology.* AMS, 2010.

[FM67] H. Freeman and S. Morse. On searching a contour map for a given terrain elevation profile. *Journal of the Franklin Institute*, 284(1):1–25, 1967.

[HWW10] W. Harvey, Y. Wang, and R. Wenger. A randomized $O(m \log m)$ time algorithm for computing reeb graph of arbitrary simplicial complexes. In *Proceedings of the Symposium on Computational Geometry (SoCG)*, pages 267–276, 2010.

[LBM+06] D. Laney, P.-T. Bremer, A. Macarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1053–1060, 2006.

[MGB+11] A. Mascarenhas, R. Grout, P.-T. Bremer, V. Pascucci, E. Hawkes, and J. Chen. Topological feature extraction for comparison of length scales in terascale combustion simulation data. In *Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications*, pages 229–240, 2011.

[Par12] S. Parsa. A deterministic $O(m \log m)$ time algorithm for the reeb graph. In *Proceedings of the Symposium on Computational Geometry (SoCG)*, pages 269–276, 2012.

[PCM02] V. Pascucci and K. Cole-McLaughlin. Efficient computation of the topology of level sets. In *IEEE Visualization*, pages 187–194, 2002.

[PSBM07] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas. Robust on-line computation of reeb graphs: simplicity and speed. *ACM Transactions on Graphics*, 26(58), 2007.

[RM00] J. B. T. M. Roerdink and A. Meijster. The watershed transform: definitions, algorithms, and parallelization strategies. *Fundamenta Informaticae*, 41:187–228, 2000.

[SK91] Y. Shinagawa and T. Kunii. Constructing a reeb graph automatically from cross sections. *IEEE Comput. Graphics Appl.*, 11(6):44–51, 1991.

[ST83] D. Sleator and R. Tarjan. A data structure for dynamic trees. *Journal of Computing and System Sciences*, 26(3):362–391, 1983.

[TGSP09] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Trans. on Visualization and Computer Graphics*, 15(6):1177–1184, 2009.

[TV98] S. Tarasov and M. Vyalyi. Construction of contour trees in 3d in $O(n \log n)$ steps. In *Proceedings of the Symposium on Computational Geometry (SoCG)*, pages 68–75, 1998.

[vKvOB+97] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the Symposium on Computational Geometry (SoCG)*, pages 212–220, 1997.

[Vui78] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978.

A §4.2 Algorithm Example

Here we provide a simple example of the execution of the main algorithm from §4.2. An example input mesh is shown in Figure 7, followed by the different steps of the join construction in Figure 8, and finally a table showing the states of the data structures and how they are modified in Figure 9.

Note that in Figure 9 a "stage" refers to any time at which one of the data structures is modified. For simplicity, the trivial stages where the maxima are removed from their corresponding heaps are not shown. (Alternatively the algorithm in §4.2 could be modified to exclude the maxima from the heaps, but we wanted to keep the algorithm description simple.)

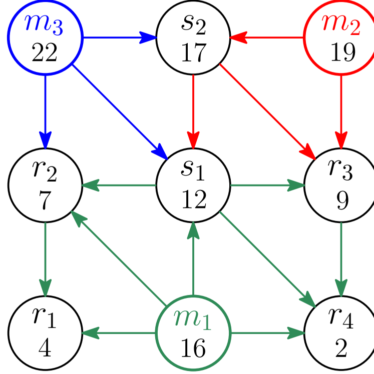

Figure 7: A piece of a potentially larger mesh, with maxima m_1, m_2, m_3, saddles s_1, s_2, and regular vertices r_1, r_2, r_3, r_4. An initial painting is shown where green 'g' is spilled from m_1, then red 'r' from m_2, and finally blue 'b' from m_3. Arrows are directed from higher to lower function value.

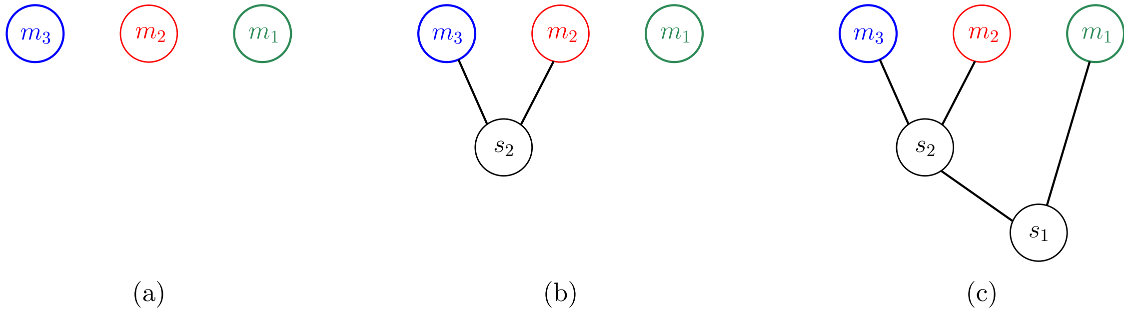

Figure 8: Snapshots of the join tree.

Stage	$att(r)$	$att(g)$	$att(b)$	$T(r)$	$T(g)$	$T(b)$	K	Current tree
Initial	m_2	m_1	m_3	$\{s_2\}$	$\{s_1\}$	$\{s_1, s_2\}$	$()$	Figure 8a
1	m_2	m_1	m_3	$\{s_2\}$	$\{s_1\}$	$\{s_1, s_2\}$	(s_1)	Figure 8a
2	m_2	m_1	m_3	$\{s_2\}$	$\{s_1\}$	$\{s_1, s_2\}$	(s_1, s_2)	Figure 8a
3	NA	m_1	s_2	NA	$\{s_1\}$	$\{s_1\}$	(s_1)	Figure 8b
4	NA	NA	s_1	NA	NA	$\{\}$	$()$	Figure 8c

Figure 9: Stages of the algorithm run on the initial painting shown in Figure 7. The above table assumes that in the stack update, s_1 is the first arbitrary unprocessed critical point selected.

B Erratum

The paper presents a new algorithm to compute contour trees, whose running time depends on the shape of the tree, and in particular avoids the initial global sort of all the critical points. The algorithm has two parts. The first part is a new algorithm to compute join trees, again whose running time depends on the shape of the tree. The second part effectively reduces computing the contour tree to join tree computations, by cutting the input simplicial complex into disjoint pieces, where for each piece the contour tree and join tree are essentially the same. The paper claims this later step can be performed in linear time. While the actual cutting can be performed in linear time, it is not clear how long it takes to determine where to make the cuts. Specifically, it is not clear how long it takes to determine the interface of the wet complex, namely the first step in the `rain` procedure of section §6, and thus it is unclear whether Theorem 6.7 holds. At the most basic level, the issue boils down to how efficiently we can determine which critical points are joins.

Before describing the potential issue, we make a few remarks. First, we stress that the new algorithm to compute the join tree and its analysis (sections §4 and §8), which are the main contributions of the paper, are unaffected by this issue. Moreover, the lower bound in section §9 for join trees is also unaffected. The issue is only with the attempt to extend the result to contour trees, and here there is no issue with the correctness of the algorithm, it is only the running time that is unclear. Let us also recall why the cutting procedure is even needed. Carr *et al.* [CSA03] showed that given the join and the split trees, in linear time one can compute the contour tree. However, as shown in Figure 2, there are inputs where roughly speaking the contour tree is more balanced than the join or split trees, and thus requires less sorting to compute. In particular, for inputs where the contour tree has a similar shape to the join or split tree, our running time bounds still apply. Note that even if the shape differs, our bounds would still apply if we could efficiently determine (or are given in advance) which critical vertices are joins.

For a maximum $x \in \mathbb{M}$, as defined in §6, $\texttt{wet}(x, \mathbb{M})$ is the set of points $y \in \mathbb{M}$ such that there is a non-ascending path from x to y. A point z is at the *interface* of $\texttt{wet}(x, \mathbb{M})$ if every neighborhood of z has non-trivial intersection with $\texttt{wet}(x, \mathbb{M})$ (i.e. the intersection is neither empty nor the entire neighborhood). The potential running time issue is that the paper does not describe precisely how to compute $\texttt{wet}(x, \mathbb{M})$. The natural approach is to perform a descending BFS or DFS from x. Suppose we reach a critical vertex v which is a join. We wet an edge in one of its up-stars when we reached it, and its other up-star is entirely dry since it is a join. As our procedure wets the interior of faces, at this point we naturally wish to walk around the interface contour adjacent to this join, and then rain down (i.e. continue the DFS) from the contour. For two distinct join vertices, with a wet edge in one up-star, the edge sets crossed by their corresponding dry contours will be disjoint. Thus the cost of walking along these contours for all joins can be charged to the size of the interface which is linear in the size of $\texttt{wet}(x, \mathbb{M})$. The issue, however, is that while we know locally whether a given vertex v is a critical vertex, we do not know whether it is a join. So suppose v is critical with one dry up-star, but that it is not a join. Then one can still walk around the contour from the dry up-star and rain down. However, if we do this for all such v it may no longer be true that the edge sets crossed by the contours are disjoint. Thus depending on the order in which vertices are handled, this could lead to edges being traversed a super-constant number of times.

It remains an open question whether this issue can be resolved without affecting the asymptotic running time. If for example, one knew which critical vertices were joins, then there is no issue. Also, observe that the highest critical vertex with a dry up-star that is hit by the initial BFS/DFS on the edges will be a join. However, repeatedly extracting and raining from the highest critical vertex could potentially lead to sorting all the critical points. On the other hand, this sorting could be avoided if for example we knew the locations of the wet critical vertices within the split

tree of the current wet portion of the complex. Note that ultimately the algorithm in the paper computes the split tree on $\texttt{wet}(x, \mathbb{M})$. Thus conceivably one could combine the wetting procedure with the split tree computation, alternately raining and growing the tree, without affecting the overall asymptotic running time.

We are currently working on trying to fix the issue described above. Regardless of whether we are able to resolve the matter or not, we plan to submit an erratum to DCG in the future.