A Mountaintop View Requires Minimal Sorting: A Faster Contour Tree Algorithm*

Benjamin Raichel† C. Seshadhri‡

Abstract

Consider a scalar field $f : \mathbb{M} \mapsto \mathbb{R}$, where $\mathbb{M}$ is a triangulated simplicial mesh in $\mathbb{R}^d$. A level set, or contour, at value v is a connected component of $f^{-1}(v)$. As v is changed, these contours change topology, merge into each other, or split. *Contour trees* are concise representations of f that track this contour behavior. The vertices of these trees are the critical points of f, where the gradient is zero. The edges represent changes in the topology of contours. It is a fundamental data structure in data analysis and visualization, and there is significant previous work (both theoretical and practical) on algorithms for constructing contour trees.

Suppose $\mathbb{M}$ has n vertices, N facets, and t critical points. A classic result of Carr, Snoeyink, and Axen (2000) gives an algorithm that takes $O(n \log n + N\alpha(N))$ time (where $\alpha(\cdot)$ is the inverse Ackermann function). A further improvement to $O(t \log t + N)$ time was given by Chiang et al. All these algorithms involve a global sort of the critical points, a significant computational bottleneck. Unfortunately, lower bounds of $\Omega(t \log t)$ also exist.

We present the first algorithm that can avoid the global sort and has a refined time complexity that depends on the contour tree structure. Intuitively, if the tree is short and fat, we get significant improvements in running time. For a partition of the contour tree into a set of descending paths, P, our algorithm runs in $O(\sum_{p \in P} |p| \log |p| + t\alpha(t) + N)$. This is at most $O(t \log D + N)$, where D is the diameter of the contour tree. Moreover, it is $O(t\alpha(t) + N)$ for balanced trees, a significant improvement over the previous complexity.

Our algorithm requires numerous ideas: partitioning the contour tree into join and split trees, a local growing procedure to iteratively build contour trees, and the use of heavy path decompositions for the time complexity analysis. There is a crucial use of a family of binomial heaps to maintain priorities, ensuring that any comparison made is between comparable nodes of the contour tree. We also prove lower bounds showing that the $\sum_{p \in P} |p| \log |p|$ complexity is inherent to computing contour trees.

*The full updated version of this paper is available online http://web.engr.illinois.edu/~raichel2/paint.pdf

†Department of Computer Science; University of Illinois; Urbana, IL, USA; raichel2@uiuc.edu; http://illinois.edu/~raichel2.

‡Sandia National Laboratories; Livermore, CA, USA; scomand@sandia.gov

1 Introduction

Massive geometric data is often represented as a function $f : \mathbb{R}^d \mapsto \mathbb{R}$. 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 specific topological data structure used to represent and visualize the function f.

It is convenient to think of f as a $(d+1)$-dimensional manifold, or alternately consider $\mathbb{M}$ 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, contours acquire or lose handles. More generally, these are the events where the topology of the level set changes. The contour tree is a concise representation of all these events.

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 and form the vertices of the contour tree. An edge connects two critical points if one event immediately "follows" the other as the sweep plane makes its pass. (We provide formal definitions later.)

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, so we can talk of the "up-degree" and "down-degree" of vertices. Leaves of down-degree 1 are maxima, because a new contour is created at such points. Leaves of up-degree 1 are minima, because a contour is destroyed there. Up-degree 2 vertices are "joins", where contours merge, and down-degree 2 vertices are "splits". When $d = 2$ (as in Figure 1), contours are closed loops. In dimensions $d \geq 3$, we may have internal vertices of both up and down-degree 1. For example, these could be events where contours gain or lose handles.

Suppose we have $f : \mathbb{M} \mapsto \mathbb{R}$, where $\mathbb{M}$ is a triangulated mesh with n vertices, N facets in total, and $t \leq n$ critical points. It is standard to assume that f is PL-Morse (a well-behaved assumption) and that the maximum degree in the contour tree is 3. 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 [CSA00]. (Henceforth, $\alpha(\cdot)$ denotes the inverse Ackermann function.) It is natural to think of N as $\Theta(n)$ in practical applications (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 $\Omega(n)$. So we are still left with the bottleneck of sorting a large set of numbers. Unfortunately, there is a worst-case lower bound of $\Omega(t \log t)$ from Chiang et al [CLLR05], based on a construction of Bajaj et al [BKO+98].

But this is only a worst-case bound. All previous algorithm begin by sorting (at least) the critical points, and hence must pay $\Omega(t \log t)$ for all instances. Can we beat this sorting bound for some instances? Is there a more refined version of the complexity of computing contour trees, and can we characterize what inputs are hard? Our main result gives an affirmative answer. First, we require some technical definitions.

In a contour tree T, a *leaf path* is simply a contiguous path in T containing a leaf, which is also monotonic in the height values of its vertices.

Definition 1.1. *For a contour tree T, a* path decomposition*, $P(T)$, is a partition of T into a set of vertex disjoint leaf paths.*

Theorem 1.2. *Consider a PL-Morse*[1] $f : \mathbb{M} \mapsto \mathbb{R}$*, where the contour tree* T *has maximum degree* 3*. There is a deterministic algorithm to compute* 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 $O(\sum_{p \in P(T)} |p| \log |p| + N)$*.*

Let us interpret the running time. For any path decomposition $P(T)$, obviously $\sum_{p \in P(T)} |p| = |T| = t$, and so the running time is $O(t \log t + N)$. But this is tight only when T is tall and thin. A better but still obvious upper bound is $O(t \log D + t\alpha(t) + N)$, where D is the diameter of T. This is clearly an improvement when the tree is short and fat. But it can be even better. A calculation yields that for any decomposition of a rooted balanced binary tree, our bound is $O(t\alpha(t) + n)$. Consider the example of Figure 2 extended to a larger binary tree both above and below. (In the example, $t = \Omega(n)$.) The running time promised by Theorem 1.2 is $O(n\alpha(n))$, whereas all previous algorithms require $\Omega(n \log n)$.

We complement Theorem 1.2 with a lower bound, suggesting that the comparison complexity of $\sum_{p \in P(T)} |p| \log |p|$ is inherent for any contour tree algorithm. The exact phrasing of the lower bound requires some care. Consider the $d = 2$ (terrain) case. Let a path decomposition P be called *valid* if it is actually output by the algorithm on some input.

Theorem 1.3. *Consider a valid 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* $\mathbf{F}_P$*, the algorithm of Theorem 1.2 makes* $O(\sum_{p \in P} |p| \log |p|)$ *comparisons.*

In other words, for any choice of $\sum_{p \in P} |p| \log |p|$, we can design a hard family where any algorithm requires this cost in the worst case, and our algorithm matches this bound.

Beyond worst-case analysis: Our result provides understanding of the complexity of contour tree construction beyond the standard worst-case analysis. Our theorem and lower bound explain *when* computing contour trees is expensive, and provide insight into the hardness of particular instances. Trees that are short and fast are easy to compute, whereas trees with long paths are hard. This is tangentially related to the concept of instance optimal algorithms, as defined by Afshani *et al.* [ABC09]. Their notion of optimality is far more refined that what we prove, but it shares the flavor of understanding the spectrum of easy to hard instances. From a mathematical perspective, this sort of analysis forces one to look closely at the relationship between sorting and the contour tree problem.

1.1 Previous Work

Contour trees were (unsurprisingly) used to study terrain maps by Boyell and Ruston, and Freeman and Morse [BR63, FM67]. They were used for isoline extraction in geometric data by van Krevald et al [vKvOB+97], who provided the first formal algorithm. Contour trees and related topological data structures 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 trees definitions and algorithms.

There has been a long series of improved algorithms for computing contour trees. van Kreveld *et al.* [vKvOB+97] presented an $O(N \log N)$ time algorithm for functions over 2D meshes and

[1] This is a standard "well-behaved" assumption that implies that all critical points have distinct values, and the contour tree has max-degree 3. Our algorithm does not need this, but it simplifies the proof and presentation.

an $O(N^2)$ algorithm for higher dimensions. Tarasov and Vyalya [TV98] improved the running time to $O(N \log N)$ for the 3D case. The influential paper of Carr *et al.* [CSA00] improved the running time for all dimensions to $O(n \log n + N\alpha(N))$. One cannot overstate the importance of this result, since it forms the basis of subsequent practical and theoretical work in computational topology applications. Cole-McLaughlin and Pascucci [PCM02] provided an $O(n + t \log n)$ time algorithm for 3-dimensional structured meshes. Chiang *et al.* [CLLR05] improve upon this result, and provide an unconditional $O(N + t \log t)$ algorithm. The latter result crucially uses monotone paths to improve the running time, and we repeatedly exploit numerous properties of monotone paths. Carr's thesis shows relationships between monotone paths in $\mathbb{M}$ and in its contour tree $\mathcal{C}(\mathbb{M})$.

Contour trees are a special case of Reeb graphs, a general topological representation for real-valued functions on any manifold. We refer the reader to Chapter 6 in [HE10] for more details. 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].

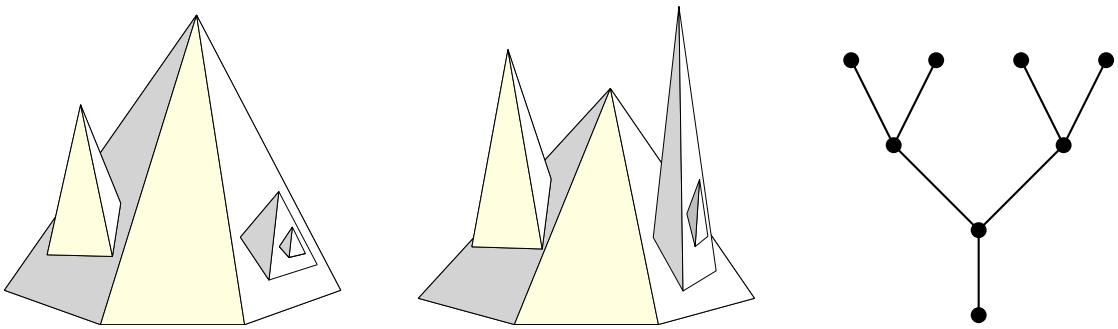

Figure 1: Two surfaces with different orderings of the maxima, but the same contour tree.

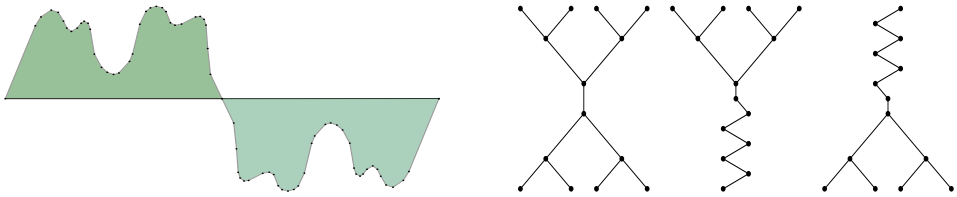

Figure 2: On left, a surface will a balanced contour tree, but whose join and split trees have long tails. On right (from left to right), the contour, join and split trees.

2 Contour tree basics

We detail the basic definitions about contour trees. We follow the terminology of Chapter 6 of Carr's thesis [Car04], which provides one of the first rigorous, Morse-theoretic defintions of contour trees. 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 (PL) function $f : \mathbb{M} \mapsto \mathbb{R}$, where $\mathbb{M}$ is a fully triangulated simplicial complex in $\mathbb{R}^d$, except for specially designated *boundary facets*. So f is defined only on the vertices of $\mathbb{M}$, and all other valued are obtained by interpolation.

We assume that f has distinct values on all vertices, except for boundaries, as expressed in the following constraint.

Definition 2.1. *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 F not in F either have all function values strictly greater or strictly less than this value.*

This is convenient, as we can now assume that f is defined on $\mathbb{R}^d$. Any point inside a boundary facet has a well-defined height, and this includes the infinite facet. We think of d as constant, and assume that $\mathbb{M}$ is represented in a data structure that allow constant-time traversals of neighboring simplices in $\mathbb{M}$ (e.g. [BM12]). (This is analogous to a DCEL for higher dimensions.) Observe that $f : \mathbb{M} \to \mathbb{R}$ can be thought of as a d-dimensional simplicial complex living in $\mathbb{R}^{d+1}$. We can think of $f(x)$ as the "height" of a point $x \in \mathbb{M}$, and encoded in the representation of $\mathbb{M}$.

Definition 2.2. *The level set at value h is the set $\{x | f(x) = h\}$. A contour is a maximal 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, 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 $B_\varepsilon(x)$.

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

We categorize points depending on the local changes in topology.

Definition 2.4. *A point x is categorized as follows.*

- *Regular: Both Morse up-degree and down-degrees are* 1.
- *Maximum: Morse up-degree is* 0.
- *Minimum: Morse down-degree is* 0.
- *Morse Join: Morse up-degree is strictly greater than* 1.
- *Morse Split: Morse down-degree is strictly greater than* 1.

Non-regular points are critical. *Morse joins and splits are* saddles. *Maxima and minima are* extrema.

We use $\mathcal{V}(f)$ to denote the set of critical points. 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 $v \in \mathcal{V}(f)$. The critical points are exactly where the topology of level sets change. By assuming that our manifold 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 maxima or minima.

Definition 2.5. *Two contours ψ and ψ' are equivalent if the following holds. 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.*

These give the *contour classes*. By continuity, a regular point is present on a single contour, and we can partition the regular contours into these 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 2.6. *The contour tree is the graph on vertex set $\mathcal{V}$, 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 alternate definitions of the contour tree in [CSA00] 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 there are no "multi-saddles", so up and down-degrees are at most 2, and the total degree is at most 3. This is again a standard assumption in topological algorithms and can be achieved by vertex unfolding (Section 6.3 in [HE10]).

2.1 Some technical remarks

The $(f(x) + \delta)$-contours in $f|_{B_\varepsilon(x)}$ given by Definition 2.3 might actually be the same contour in f. The up-degree (as opposed to *Morse* up-degree) is defined as the number of $(f(x) + \delta)$-contours 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. For example, when $d = 3$ this is equivalent to the contour gaining a handle. When $d = 2$, this distinction is not necessary, since any point with > 1 Morse degree will have > 1 degree in $\mathcal{C}(\mathbb{M})$.

As Carr points out in Chapter 6 of his thesis, the term contour tree can be used for a family of related structures. Every vertex in $\mathbb{M}$ is associated with an edge in $\mathcal{C}(\mathbb{M})$, and sometimes the vertex is explictly 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 $\mathcal{C}(\mathbb{M})$ will 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 2.6 is the typical definition in all results on output-sensitive contour trees, and is the smallest tree that contains all the topological changes of level sets. Theorem 1.2 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 contour tree algorithm

Our final algorithm is quite technical and has numerous moving parts. This section provides a high level view of the entire result, and highlights the main ideas and intuition. It is helpful to keep the $d = 2$ case in mind, so the input is just a triangulated terrain. In the interest of presentation, the definitions and theorem statements in this section will slightly differ from those in the main body. They may also differ from the original definitions proposed in earlier work.

Do not globally sort: Arguably, 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. Therefore, clearly sorting all critical points is computationally unnecessary for constructing the contour tree. In Figure 2, the contour tree basically 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. More generally, leafs 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?

3.1 Breaking $\mathbb{M}$ into simpler pieces

Let us begin with the algorithm of Carr, Snoeyink, and Axen [CSA00]. The key insight is to build two different trees, called the join and split trees, and then merge them together into the contour tree. Consider sweeping down via 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 tracks exactly these events. Formally, let $\mathbb{M}_v^+$ denote the simplicial complex induced on the subset of vertices which are higher than v.

Definition 3.1. *The critical join tree $\mathcal{J}_C(\mathbb{M})$ is built on the set $\mathcal{V}$ of all critical points. The directed edge (u, v) is present when u is the smallest valued vertex in $\mathcal{V}$ in a connected component of $\mathbb{M}_v^+$ and v is connected to this component (in $\mathbb{M}$).*

(Abusing notation, we just call this the join tree.) 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. They simply form a long path. The split tree is obtained by simply inverting this procedure, sweeping upwards and tracking sublevel sets.

A major insight of [CSA00] 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 [CSA00] 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.

Contour surgery: We first need a divide-and-conquer lemma. Any contour ϕ can be associated with an edge e of the contour tree. Suppose we "cut" $\mathbb{M}$ along this contour. We prove that $\mathbb{M}$ is split into two disconnected pieces, such the contour trees of these pieces is obtained by simply cutting e in $\mathcal{C}(\mathbb{M})$. Alternatively, the contour trees of these pieces can be glued together to get $\mathcal{C}(\mathbb{M})$. This is not particularly surprising, and is fairly easy to prove with the right definitions. The idea of loop surgery has been used to reduce Reeb graphs to contour trees [TGSP09, DN13]. Nonetheless, our theorem appears to be new and works for all dimensions.

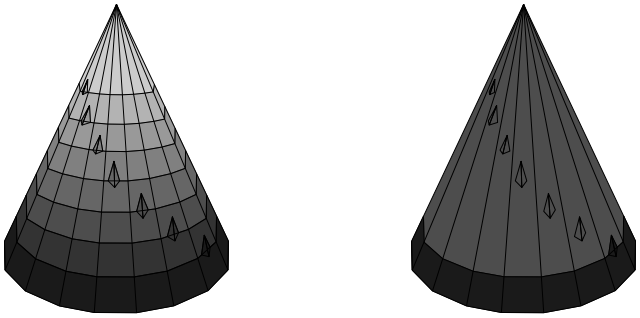

Figure 3: On left, downward rain spilling only (each shade of gray represents a piece created by each different spilling), producing a grid. Note we are assuming raining was done in reverse sorted order of maxima. On right, flipping the direction of rain spilling.

Cutting $\mathbb{M}$ into extremum dominant pieces: We define a simplicial complex endowed with a height to be *minimum dominant* if there only exists a single minima. (Our real definition is more complicated, and involves simplicial complexes that allow additional "trivial" minima.) In such a complex, there exists a non-ascending path from any point to this unique minima. Analogously, we can define maximum dominant simplicial complexes, and both are collectively called extremum dominant.

We will cut $\mathbb{M}$ into disjoint extremum dominant pieces, in linear time. The best way to think of our procedure is a meteorology analogy. Take an arbitrary maxima x, and imagine torrential rain at the maxima. 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, while there may be numerous disconnected dry parts. The interface between the dry and wet parts is a set of contours, given by the "water line". The wet part is clearly maximum dominant, since all wet points have a non-descending path to x. So we can simply cut along the interface contours to get the wet maximum dominant piece $\mathbb{M}'$. By our contour surgery theorem, we are left with a set of disconnected dry parts, and we can recur this procedure on them.

But here's the catch. Every time we cut $\mathbb{M}$ along a contour, we potentially increase the complexity of $\mathbb{M}$. Water flows in the interior of facets, and the interface will naturally cut some facets. Each cut introduces new vertices, and a poor choice of repeated raining leads to a large increase in complexity. Consider the left of Figure 3. Each raining produces a single wet and dry piece, and each cut introduces many new vertices. If we wanted to partition this terrain into maximum dominant simplicial complexes, the final complexity would be forbiddingly large.

A simple trick saves the day. Unlike reality, we can choose rain to flow solely downwards or solely upwards. Apply the procedure above to get a single wet maximum dominant $\mathbb{M}'$ and a set of dry pieces. Observe that a single dry piece $\mathbb{N}$ is boundary critical with the newly introduced boundary ϕ (the wet-dry interface) behaving as a minima. So we can rain upwards from this minima, and get a *minimum dominant* portion $\mathbb{N}'$. This ensures that the new interface (after applying the procedure on $\mathbb{N}$) does not cut any facet previously cut by ϕ. For each of the new dry pieces, the newly introduced boundary is now a maximum. So we rain downwards from there. More formally, we alternate between raining upwards and downwards as we go down the recursion tree. We can prove that an original facet of $\mathbb{M}$ is cut at most once, so the final complexity can be bounded. In Figure 3, this procedure would yield two pieces, one maximum dominant, and one minimum dominant.

Using the contour surgery theorem previously discussed, we can build the contour tree of $\mathbb{M}$ from the contour trees of the various pieces created. All in all, we prove the following theorem.

Theorem 3.2. *There is an $O(N)$ time procedure that cuts $\mathbb{M}$ into extremum dominant simplicial complexes $\mathbb{M}_1, \mathbb{M}_2, \ldots$. Furthermore, given the set of contour trees $\{\mathcal{C}(\mathbb{M}_i)\}$, $\mathcal{C}(\mathbb{M})$ can be constructed in $O(N)$ time.*

Extremum dominance simplifies contour trees: We will focus on minimum dominant simplicial complexes $\mathbb{M}$. By Theorem 3.2, it suffices to design an algorithm for contour trees on such inputs. For the $d = 2$ case, it helps to visualize such an input as a terrain with no minima, except at a unique boundary face (think of a large boundary triangle that is the boundary). All the valleys in such a terrain are necessarily joins, and there can be no splits. Look at Figure 2. The portion on the left is minimum dominant in exactly this fashion. More formally, $\mathbb{M}_v^-$ is connected for all v, so there are no splits.

We can prove that the split tree is just a path, and the contour tree is exactly the join tree.

The formal argument is a little involved, and we employ the merging procedure of [CSA00] to get a proof. We demonstrate that the merging procedure will actually just output the join tree, so we do not need to actually compute the split tree. (The real definition of minimum dominant is a little more complicated, so the contour tree is more than just the join tree. But computationally, it suffices to construct the join tree.)

We stress the importance of this step for our approach. Given the algorithm of [CSA00], one may think that it suffices to design faster algorithms for join trees. But this cannot give the sort of optimality we hope for. Again, consider Figure 2. Any algorithm to construct the true join tree must construct the path of splits, which implies sorting all of them. It is absolutely necessary to cut $\mathbb{M}$ into pieces where the cost of building the join tree can be related to that of building $\mathcal{C}(\mathbb{M})$.

3.2 Join trees from painted mountaintops

Arguably, everything up to this point is a preamble for the main result: a faster algorithm for join trees. Our algorithm does not require the input to be extremum dominant. This is only required to relate the join trees to the contour tree of initial input $\mathbb{M}$. For convenience, we use $\mathbb{N}$ to denote the input here. Note that as defined in Definition 3.1, the join tree is defined purely combinatorially in terms of the 1-skeleton (the underlying graph) of $\mathbb{N}$.

The join tree $\mathcal{J}_C(\mathbb{N})$ is a rooted tree with the dominant minimum at the root, and we direct edges downwards (towards the root). So it makes sense to talk of comparable vs incomparable vertices. We arrive at the primary challenge: how to sort only those critical points that are comparable, without constructing the join tree? The join tree algorithm of [CSA00] is a typical event-based computational geometry algorithm. We have to step away from this viewpoint to avoid the global sort.

The key idea is *paint spilling*. Start with each maxima having a large can of paint, with distinct colors for each maxima. In arbitrary order, we spill paint from each maxima, 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 facets.* That is, this process is restricted to the 1-skeleton of $\mathbb{N}$. Furthermore, our paints do not mix, so each edge receives a unique color, which is decided by the first paint to reach it.

Definition 3.3. *Let the 1-skeleton of $\mathbb{N}$ have edge set E and maxima X. A* painting *of $\mathbb{N}$ is a map $\chi : X \cup E \mapsto [|X|]$ 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 has the additional property that the restriction $\chi : X \mapsto [|X|]$ is a bijection.*

Note that a painting only colors edges, not vertices. Our definition also does not require the timing aspect of iterating over colors, though that is one way of painting $\mathbb{N}$. We begin with an initial painting, since all maxima colors are distinct. A few comments on paint vs water. The interface between two regions of different color is *not* a contour, and is actually difficult to characterize. So we cannot apply the divide-and-conquer approach of contour surgery. On the other hand, painting does not cut $\mathbb{N}$, so there is no increase in complexity. Clearly, an initial painting can be constructed in $O(N)$ time. This is the tradeoff between water and paint. Water allows for an easy divide-and-conquer, at the cost of more complexity in the input. For an extremum dominant input, using water to divide the input $\mathbb{N}$ raises the complexity too much.

Our algorithm incrementally builds $\mathcal{J}_C(\mathbb{M})$ from the leaves (maxima) to the root (dominant minimum). 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 an initial painting, where the colors have 1-1 correspondence with the maxima. Refer to the left part of Figure 4. 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 the mounds are colored completely blue or red, while v is touched by both colors. So this indicates that v joins the blue maxima and red maxima in $\mathcal{J}_C(\mathbb{M})$.

This is precisely how we hope to exploit the information in the painting. We prove later that when some join v has all incident edges with exactly two colors, the corresponding maxima (of those colors) are exactly the children of v in $\mathcal{J}_C(\mathbb{M})$. 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 maxima. Suppose purple paint was poured from v. In terms of $\mathcal{J}_C(\mathbb{M})$, this is equivalent to removing leaves ℓ_1 and ℓ_2, making v a new leaf. Alternately, $\mathcal{J}_C(\mathbb{M})$ 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; we maintain a union-find data structure for that.

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 4). In $\mathcal{J}_C(\mathbb{M})$, 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 issue. We know that u must also be touched by blue. So all critical vertices touched by blue are put into a priority queue keyed by height, and vertices are handled in that order.

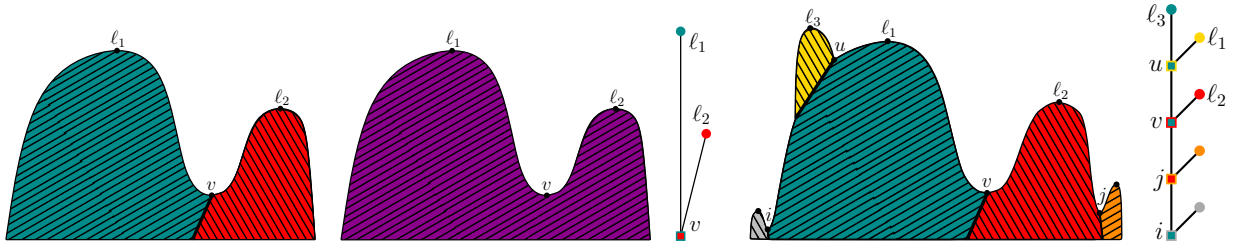

Figure 4: On the left, red and blue merge to make purple, followed by the contour tree with initial colors. On the right, additional maxima and the resulting contour tree.

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. We also need to merge the priority queues to get a consistent painting. And that decides the use of *binomial heaps* [Vui78] to store the critical points of one color. Merges can be done in logarithmic time, the same time required for deletes. We stress that the feasibility of the entire approach hinges on the use of binomial heaps.

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 have at most 2 colors, because of all the color merges that previously occured.

The running time analysis: Relating the running time to a path decomposition is the most technical part of the paper. All the non-heap operations can be easily bounded by $O(t\alpha(t) + N)$ (the $t\alpha(t)$ is from the union-find data structure for colors). It is not hard to argue that at all times, any heap always contains a subset of a leaf to root path. Unfortunately, this subset is *not* contiguous, as the right part of Figure 4 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, it will merge into the component containing the blue mound (and also the yellow and red mounds) before the far left one. Hence, the vertices initially touched by blue are not contiguous in the contour tree.

Nonetheless, we can get a non-trivial (but non-optimal) bound. Let d_v denote the distance to the root for vertex v in the join tree. The total cost (of the heap operations) is at most $\sum_v \log d_v$. This immediately proves a bound of $O(t \log D)$, where D is the maximum distance to the root, an improvement over previous work. But this bound is non-optimal. For a balanced binary tree, this bound is $O(t \log \log t)$, whereas the cost of any path decomposition is $O(t)$.

The cost of heap operations depends on the sizes of the heaps, which keeps changing because of the repeated merging. This is a major headache for the analysis. There are situations where the initial heap sizes are small, but they eventually merge to create larger heaps. We employ a variant of *heavy path decompositions*, first used by Sleator and Tarjan for analyzing link/cut trees [ST83]. The final analysis basically charges expensive heap operations to long paths in the decomposition.

3.3 The lower bound

Consider a contour tree T and the path decomposition $P(T)$ used to bound the running time. Denoting $cost(P(T)) = \sum_{p \in P(T)} |p| \log |p|$, we construct a set of $\prod_{p \in P(T)} |p|!$ functions on a fixed domain such that each function has a distinct (labeled) contour tree. By a simple entropy argument, any algebraic decision tree that correctly computes the contour tree on all instances requires worst case $\Omega(cost(P(T)))$ time. We prove that our algorithm makes $\Theta(cost(P(T)))$ comparisons on all these instances. In general, any algorithm requires $\Omega(C)$ time to solve the set of instances that our algorithm solves with C comparisons.

We have a fairly simple construction that works for terrains. In $P(T)$, consider the path p that involves the root. The base of the construction is a conical "tent", and there will be $|p|$ triangular faces that will each have a saddle. The heights of these saddles can be varied arbitrarily, and that will give $|p|!$ different choices. Each of these saddles will be connected to a recursive construction involving other paths in $P(T)$. Effectively, one can think of tiny tents that are sticking out of each face of the main tent. The contour trees of these tiny tents attach to a main branch of length $|p|$. Working out the details, we get $\prod_{p \in P(T)} |p|!$ terrains each with a distinct contour tree. It is possible to argue that our algorithm requires $\Theta(cost(P(T)))$ comparisons on all these instances.

4 Divide and conquer through contour surgery

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 contour ϕ, this is constructing 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 (with some simplex that x is present in), we can walk along $\mathbb{M}$ from x and determine ϕ. We can "cut"

along ϕ to get a new (possibly) disconnected simplicial complex $\mathbb{M}'$. This is achieved by splitting every facet F that ϕ intersects into an "upper" facet and "lower" facet. Algorithmically, we cut facet F with ϕ^+ and take everything above ϕ^+ in F to make the upper facet. Analogously, we cut with ϕ^- to get the lower facet. The facets are then triangulated to ensure that they are all simplices. Given that ϕ cannot cut a boundary and all non-boundary facets have constant size, we omit the algorithmic description for this process. We only note that this creates the two new boundaries ϕ^+ and ϕ^-, and we maintain the property of constant f-value at a boundary. This new simplicial complex is denoted by $\texttt{cut}(\phi, \mathbb{M})$. This can be constructed in time linear in $|\phi|$.

We now describe a high-level approach to construct $\mathcal{C}(\mathbb{M})$.

$\texttt{surgery}(\mathbb{M}, \phi)$

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 minima and the other is maxima.)
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 4.1. *For any regular contour ϕ, the output of* $\texttt{surgery}(\mathbb{M}, \phi)$ *is* $\mathcal{C}(\mathbb{M})$.

We require some theorems from [Car04] (Theorems 6.6 and 6.7) that map paths in $\mathcal{C}(\mathbb{M})$ to $\mathbb{M}$.

Theorem 4.2. *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.*

Theorem 4.3. *For every monotone path P in $\mathbb{M}$, there exists a monotone path Q in the contour tree to which P maps (as in the previous theorem), and vice versa.*

The main theorem is a direct consequence of the following lemma.

Lemma 4.4. *Consider a regular contour ϕ contained in a contour class (of an edge of $\mathcal{C}(\mathbb{M})$) (u, v) and denote $\mathbb{M}' = \texttt{cut}(\mathbb{M}, \phi)$. 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. The contours of $\mathbb{M}'$ are exactly the contours of $\mathbb{M}$ with ϕ^+ and ϕ^-, without ϕ. All new vertices created in $\mathbb{M}'$ are present on ϕ^+ and ϕ^-. That proves the first part.

Any contour class in $\mathbb{M}'$ (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 4.5. $\texttt{cut}(\mathbb{M}, \phi)$ *consists of two disconnected simplicial complexes.*

Proof. Denote (as in Lemma 4.4) 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 4.2, 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 obviously cannot use edges incident to them. By Lemma 4.4, 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 4.4) and apply Theorem 4.2. □

5 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 5.1. *A simplicial complex is* minimum dominant *if there exists a minimum x such that every non-minimal* vertex *in the manifold has a decreasing 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 descending paths and "wet" all the points encountered. Note that this procedure considers all points of the manifold, not just vertices.

Definition 5.2. *Fix $\mathbb{M}$ and $x \in \mathbb{M}$. 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 is 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 $\texttt{wet}(x, \mathbb{M})$.*

The following claim gives a description of the interface. While its meaning is quite intuitive, the proof is tedious.

Claim 5.3. *For any x, the interface of $\texttt{wet}(x, \mathbb{M})$ is a set of contours, each containing 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 to 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, $B_\varepsilon(y)$ is dry.) The points in $B_\varepsilon(y)$ that lie below y have a descending path from y and hence must be wet. There must also be a dry point in $B_\varepsilon(y)$ 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.

Therefore, there must exist a join vertex w in Γ_y, such that $B_\varepsilon(w)$ intersects both ϕ and ψ. As $\delta, \varepsilon \to 0^+$, the limit of ϕ (which is dry) lies in Γ_y (which is wet). Hence this limit gives a contour of the interface containing the join w. □

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 facet F of $\mathbb{M}$ encountered is

partially (if not entirely) in $\texttt{wet}(x, \mathbb{M})$. This portion is determined by cutting F along the interface. Since the interface is a contour, this is equivalent to 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 contours. Consider such a contour ϕ containing a join vertex y. Take any dry increasing edge incident to y, and pick the point z on this edge at height $f(y) + \delta$. Let $\texttt{lift}(\phi)$ be the unique contour through the regular point z. Note that $\texttt{lift}(\phi)$ is dry.

Claim 5.4. *Let ϕ be an interface contour. Then $\texttt{cut}(\mathbb{M}, \texttt{lift}(\phi))$ results in two disjoint simplicial complexes, one consisting entirely of dry points.*

Proof. By Theorem 4.5, $\texttt{cut}(\mathbb{M}, \texttt{lift}(\phi))$ 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. □

We describe the main partitioning procedure that cuts a simplicial complex $\mathbb{N}$ into extremum dominant complexes. 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. One of the critical aspects of the procedure is that the rain alternately flows downwards and upwards, since otherwise faces may be cut a super constant number of times (see Figure 3). In other words, when we start, rain flows downwards. In each recursive call, the direction of rain is switched to the opposite direction. While one can implement `rain` this way, it is conceptually easier to think of *inverting* a complex $\mathbb{N}'$ when a recursive call is made. Inversion is easily achieved by just negating the height values. (From a running time standpoint, it suffices to maintain a single bit associated with $\mathbb{N}'$ that determines whether heights are inverted or not.) 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 contours 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 $\mathbb{M}_1, \mathbb{M}_2, \ldots, \mathbb{M}_r$. We first argue the correctness of `rain`.

Lemma 5.5. *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 $\phi_1, \phi_2, \ldots, \phi_k$ (as denoted in the procedure). By repeated applications of Claim 5.4, $\mathbb{N}_{k+1}$ contains $\texttt{wet}(x, \mathbb{M})$. Consider $\texttt{wet}(x, \mathbb{N}_{k+1})$. The interface must exactly be $\phi_1, \phi_2, \ldots, \phi_k$. So the only dry vertices are those in the boundaries $B_1, B_2, \ldots, B_k$. But these boundaries are maxima. □

Focus on $\texttt{rain}(x, \mathbb{M})$. As `rain` proceeds, new facets/simplices are created because of repeated cutting. Indeed, any facet F in any simplicial complex $\mathbb{N}$ that is recursively invoked, is achieved by cutting facet F' in $\mathbb{M}$. The key to the running time of $\texttt{rain}(x, \mathbb{M})$ is bounding the number of newly created facets, for which we have the following lemma.

Lemma 5.6. *A facet $F \in \mathbb{M}$ is cut at most once during* `rain`$(x, \mathbb{M})$.

Proof. All notation here follows that in the pseudocode of `rain`. First, by Theorem 4.5, 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 `rain`$(x, \mathbb{N})$. Specifically, say this happens when `cut`$(\mathbb{N}_i, \phi_i')$ is constructed. So F is cut by some hyperplane into lower and upper portions (which are then triangulated). Since the cut is infinitesimally above an interface, the lower portion is basically wet. It can never be part of $\mathbb{L}_1, \mathbb{L}_2, \ldots$, which go into recursive calls. This portion is in $\mathbb{N}_{k+1}$. The upper portion/facet (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 `rain`$(B_i, \mathbb{L}_i)$ (and $\mathbb{L}_i$ is inverted), U becomes wet. Hence, U will not be cut subsequently. All in all, F is cut at most once. □

Theorem 5.7. *The total running time of* `rain`$(x, \mathbb{M})$ *is* $O(|\mathbb{M}|)$.

Proof. The operations performed are the raining and cutting. We can bound this by the total complexities of the wet complexes and the new boundaries created. No facet is ever wet twice, and by Lemma 5.6, the total number of facets that every appear is $O(|\mathbb{M}|)$. This also bounds the complexities of the boundaries created. □

Claim 5.8. *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 `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 is constant time, and the total time is the size of the recursion tree. □

6 Contour trees of extremum dominant manifolds

The previous section allows us to restrict attention to extremum dominant manifolds. 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^*. The set of non-dominant minima is denoted by M. 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.

The main theorem of this section asserts that the contours trees of minimum dominant manifolds have a simple description. The exact statement will require some definitions and notation. We required the notions of *join* and *split* trees, as given by [CSA00]. Conventially, all edges are directed from higher to lower function value.

Definition 6.1. *The join tree $\mathcal{J}(\mathbb{M})$ of $\mathbb{M}$ is built on vertex set $V(\mathbb{M})$. 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 connected to this component (in $\mathbb{M}$). The split tree $\mathcal{S}(\mathbb{M})$ is obtained by looking at $\mathbb{M}_v^-$ (or alternately, 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, leaves are minima, and the global maximum is the root. As these trees 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).

The key observation is that $\mathcal{S}(\mathbb{M})$ is trivial for a minimum dominant $\mathbb{M}$.

Lemma 6.2. *$\mathcal{S}(\mathbb{M})$ 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 4.5, $\texttt{cut}(\mathbb{M}, \phi)$ 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 manifold amounts to computing its join tree. Specifically, to prove our main theorem, we rely on the correctness of the merging procedure from [CSA00] that constructs the contour tree from the join and split trees. It actually construsts the *augmented contour tree* $\mathcal{A}(\mathbb{M})$, which is the contour tree with regular points inserted into the edges. Consider tree T with vertex v of in and outdegree at most 1. *Erasing* v from T is the following operation: if v is a leaf, just delete v. Otherwise, smooth v out (delete v and connect its neighbors by an edge). This tree is denoted by $T \ominus v$.

$\texttt{merge}(\mathcal{J}(\mathbb{M}), \mathcal{S}(\mathbb{M}))$

1. Set $\mathcal{J} = \mathcal{J}(\mathbb{M})$ and $\mathcal{S} = \mathcal{S}(\mathbb{M})$.
2. Denote v as *candidate* if sum of indegree of v in $\mathcal{J}$ and outdegree of v in $\mathcal{S}$ 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 $\mathcal{J}$, consider its edge in $\mathcal{J}$. Otherwise consider its edge in $\mathcal{S}$. In either case, denote the edge by (v, w).
   (b) Insert (v, w) in $\mathcal{A}(\mathbb{M})$.
   (c) Set $\mathcal{J} = \mathcal{J} \ominus v$ and $\mathcal{S} = \mathcal{S} \ominus v$. Enqueue any new candidates.
5. Smooth out all regular vertices in $\mathcal{A}(\mathbb{M})$ to get $\mathcal{C}(\mathbb{M})$.

Definition 6.3. *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 connected to this component (in $\mathbb{M}$).*

Theorem 6.4. *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 first show that $\mathcal{A}(\mathbb{M})$ is $\mathcal{J}(\mathbb{M}) \ominus \{m_i\}$ with edges $\{(s_i, m_i)\}$. We have flexibility in choosing the order of processing in `merge`. We first put the non-dominant maxima $m_1, \ldots, m_k$ 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. The join tree is now $\mathcal{J}(\mathbb{M}) \ominus \{m_i\}$. We can now process $\mathcal{J}$ leaf by leaf, and all edges of $\mathcal{J}$ are inserted into $\mathcal{A}(\mathbb{M})$.

Note that $\mathcal{C}(\mathbb{M})$ is obtained by smoothing out all regular points from $\mathcal{A}(\mathbb{M})$. Smoothing out regular points from $\mathcal{J}(\mathbb{M}) \ominus \{m_i\}$ exactly yields the edges described in the theorem. □

7 Painting to compute contour trees

The main algorithmic contribution is a new algorithm for computing join trees of any triangulated simplicial complex $\mathbb{M}$.

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 maxima x. We will spill different paint from each maximum and watch it flow down. This is analogous to the raining in the previous section, but paint is a much more viscous liquid. *So paint only flows down edges, and it does not color the interior of facets.* 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.

We now restate Definition 3.3.

Definition 7.1. *Let the 1-skeleton of $\mathbb{M}$ have edge set E and maxima X. A* painting *of $\mathbb{M}$ is a map $\chi : X \cup E \mapsto [|X|]$ 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 has the additional property that the restriction $\chi : X \mapsto [|X|]$ is a bijection.*

A painting only colors edges, not vertices. We associate certain sets of colors with vertices.

Definition 7.2. *Fix a painting χ and vertex v.*

- *An* up-star *of v is the set of edges that all connected to a fixed component of $\mathbb{M}_v^+$.*
- *A vertex v is* touched by color c *if x 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 v.*

Each non-maxima v participates in at least 1 and at most 2 up-stars (the latter iff v is a join).

7.1 The data structures

We discuss all the initializations in the next section.

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.

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 contour tree above (and including) $att(c)$ has already been constructed.

7.2 The algorithm

We formally describe the algorithm below. We require a technical definition of *ripe* vertices.

Definition 7.3. *A vertex v is* ripe *if: for all $c \in col(v)$, v is present in $T(rep(c))$ and is also highest vertex in the heap.*

`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 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.

`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 $\mathcal{J}_C(\mathbb{M})$.
      ii. Delete h from $T(c')$.
   (d) Merge heaps $\{T(c') | c' \in cur(h)\}$.
   (e) Take union of $cur(h)$ and denote resulting 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 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. After processing, it is removed from all heaps.
- After v is processed, all edges incident to v have the same color (technically, same representative).
- Vertices on the stack are in increasing height order.

Observation 7.4. Each unprocessed vertex is always in exactly one queue of the colors in each up-star. Specifically, for a given up-star of a vertex v, `init`($\mathbb{M}$) puts v into the queue of exactly

one color, say c. As time goes on this queue may merge with other queues, but while unprocessed v 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`$(\mathbb{M})$)

7.3 Proving correctness

Our main workhorse is the following technical lemma.

Lemma 7.5. *Suppose vertex v is connected to a component $\mathbb{P}$ of $\mathbb{M}_v^+$ by edge e which is currently colored c. Either all edges in $\mathbb{P}$ are currently colored c, or there exists critical vertex $w \in \mathbb{P}$ fully touched by c and touched by another color.*

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

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

Corollary 7.6. *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 initial adjacent color c. Therefore, h must be in the current heap of $rep(c)$ (see Observation 7.4). Now by Lemma 7.5, either all edges in $\mathbb{P}$ are colored $rep(c)$ or there is some vertex w fully touched by $rep(c)$ and 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. □

We prove a series of claims that will lead to the correctness proof.

Claim 7.7. *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 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 7.5, 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 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 7.8. *The partial output on the processed vertices is exactly the restriction of $\mathcal{J}_C(\mathbb{M})$ to these vertices.*

Proof. More generally, we prove the following: all outputs on processed vertices are edges of $\mathcal{J}_C(\mathbb{M})$ and for any current color c, $att(c)$ is the lowest processed vertex of that color. We prove this by induction on the processing order. The base case is 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 7.6 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 7.7, all edges in $\mathbb{P}$ and the up-star have the same 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 $\mathcal{J}_C(\mathbb{M})$ to $\mathbb{P}$ has been correctly computed. Additionally, since all critical vertices in $\mathbb{P}$ have processed, they all have the same 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 color of edges in the respective component). Now by the definition of $\mathcal{J}_C(\mathbb{M})$, 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 `build`$(\mathbb{M})$. Moreover, `build` merges the 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 7.9. *Given an input complex $\mathbb{M}$,* `build`$(\mathbb{M})$ *terminates and outputs $\mathcal{J}_C(\mathbb{M})$.*

Proof. First observe that each vertex can be processed at most once by `build`$(\mathbb{M})$. By Corollary 7.6, we know that as long as there is an unprocessed vertex, `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 7.8 the algorithm will terminate having computed $\mathcal{J}_C(\mathbb{M})$. □

7.4 Running Time

We now bound the running time of the above algorithm. In the subsequent sections we will then provide a matching lower bound for the running time. Therefore, it will be useful to set up some terminology that can be used consistently in both places. Specifically, the lower bound proof will be a purely combinatorial statement on colored rooted trees, and so the terminology is of this form.

Any tree T considered in following will be a rooted binary tree[2] 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

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

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 7.10. *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 color).*

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

Definition 7.11. *Note that the subroutine* `init`$(\mathbb{M})$ *from §7.2 naturally defines a leaf assignment to $\mathcal{J}_C(\mathbb{M})$ according to the priority queue for each up-star we put a given vertex in. Call this the* initial coloring *of the vertices in $\mathcal{J}_C(\mathbb{M})$. Note also that this initial coloring defines the H_v values for all $v \in \mathcal{J}_C(\mathbb{M})$.*

The following lemma should justify these technical definitions.

Lemma 7.12. *Let $\mathbb{M}$ be a simplicial complex with t critical points. For every vertex in $\mathcal{J}_C(\mathbb{M})$, let H_v be defined by the initial coloring of $\mathbb{M}$. The running time of* `build`$(\mathbb{M})$ *is $O(N + t\alpha(t) + \sum_{v \in \mathcal{J}_C(\mathbb{M})} \log |H_v|)$.*

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 as 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 $O(\alpha(t))$ time. For `update`, by Observation 7.4, 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 $O(t)$.

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 `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 $\mathcal{J}_C(\mathbb{M})$, 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 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. □

8 Leaf assignments and path decompositions

In this section, we set up a framework to analyze the time taken to compute a critical join tree $\mathcal{J}_C(\mathbb{M})$ (see Definition 6.3). We adopt all notation already defined in §7.4. 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 choose a special path decomposition that is best defined as a subset of edges in T such that each internal vertex has 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 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 §8.4 by showing that proving this theorem implies our main result Theorem 1.2.

8.1 Shrubs, tall paths, and short paths

The paths in $P(T)$ 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 $P(T)$. Abusing notation, we simply use $P(T)$ 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 vertex in p closest to root of T. We use $\ell(p)$ to denote the leaf in p. We use H_p to denote the $H_{b(p)}$.

Lemma 8.2. *Let p be any path in $P(T)$ and let $\{q_1, \dots q_k\}$ be the children on p. Then $H_{\ell(p)} + \sum_{i=1}^{k} |H_{q_i}| \leq |H_p| + 2|p|$.*

Proof. For convenience, denote $H_i = H_{q_i}$ and $H_0 = H_{\ell(p)}$. Consider $v \in \bigcup_i H_i$ that lies below $b(p)$ in T. Note that such a vertex has only one of its two colors in $L(b(p))$. Since the colors tracked by H_i and H_j for $i \neq j$ are disjoint, such a vertex can appear in only one of the H_i's. On the other hand, a vertex $u \in p$ can appear in more than one H_i, but since any vertex has exactly two colors it can appear in at most two such heaps. Hence, $\sum_i |H_i| \leq |H_p| + 2|p|$. □

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

Definition 8.3. *For $p \in P(T)$, p is* short *if $|p| < \sqrt{|H_p|}/100$, and* tall *otherwise.*

[3]Please excuse the overloading of the term 'tree', it is the most natural term to use here.

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

Lemma 8.4. *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 paths adjacent to p).

Definition 8.5. *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.*

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 $P_{\max}(T)$. We now construct the shrub of unsupported short paths. 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 edge from p to q, so q is the parent of p in $\mathcal{U}$. This construction leads to the shrub forest of $\mathcal{L} \setminus \mathcal{L}'$, where all the roots are supported short paths, and the remaining nodes are the unsupported short paths.

Most of the work goes into proving the following technical lemma.

Lemma 8.6. *Let $\mathcal{U}$ denote a connected component (shrub) in the shrub forest of $\mathcal{L} \setminus \mathcal{L}'$ 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) $\sum_{p\in\mathcal{U}} |p| = O(|H_r|)$.*

We split the remaining argument into two subsections. We first prove Theorem 8.1 from Lemma 8.6, which involves routine calculations. Then we prove Lemma 8.6, where the interesting work happens.

8.2 Proving Theorem 8.1

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

$$\sum_{p\in\mathcal{L}} \sum_{v\in p} \log |H_v| \quad = \quad \sum_{p\in\mathcal{L}\setminus\mathcal{L}'} \sum_{v\in p} \log |H_v| + \sum_{p\in\mathcal{L}'} \sum_{v\in p} \log |H_v| + \sum_{p\in\mathcal{H}} \sum_{v\in p} \log |H_v|$$

The last term can be bounded by $O(\sum_{p\in P_{\max}(T)} |p| \log |p|)$, by Lemma 8.4. The second term can be bounded by $O(\sum_{p\in\mathcal{L}'} |H_p| \log |H_p|)$, by Lemma 8.4 again. The following claim shows that this in turn is at most the last term.

Claim 8.7. $\sum_{p\in\mathcal{L}'} |H_p| \log |H_p| = O(\sum_{q\in\mathcal{H}} \sum_{v\in q} \log |H_v|)$.

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 $O(\sum_{q \in \mathcal{H}} \sum_{v \in q} \log |H_v|)$. □

The first term (unsupported short paths) can be charged to the second term (supported short paths). This is where the critical Lemma 8.6 plays a role.

Claim 8.8. $\sum_{p \in \mathcal{L} \setminus \mathcal{L}'} \sum_{v \in p} \log |H_v| = O(\sum_{p \in \mathcal{L}'} |H_p| \log |H_p|)$.

Proof. Let $\mathcal{U}$ denote a connected component of the shrub forest. We have $\sum_{p \in \mathcal{L} \setminus \mathcal{L}'} \sum_{v \in p} \log |H_v| \leq \sum_{\mathcal{U}} \sum_{p \in \mathcal{U}} \sum_{v \in p} \log |H_v|$. By Lemma 8.6, $|H_v| = O(|H_r|)$, where r is the root of $\mathcal{U}$. Furthermore, $\sum_{p \in \mathcal{U}} |p| = O(|H_r|)$. We have $\sum_{p \in \mathcal{U}} \sum_{v \in p} \log |H_v| = O((\log |H_r|) \sum_{p \in \mathcal{U}} |p|) = O(|H_r| \log |H_r|)$. We sum this over all $\mathcal{U}$ in the shrub forest, and note that roots in the shrub forest are supported short paths. □

8.3 Proving Lemma 8.6: the root is everything in $\mathcal{U}$

Lemma 8.6 asserts the root r in $\mathcal{U}$ pretty much encompasses all sizes and heaps in $\mathcal{U}$. We will work with the *reduced* heap $\widetilde{H}_p$. This is the subset of vertices of H_p that do not appear on the tall support chain of p. By definition, for any unsupported short path (hence, any non-root $p \in \mathcal{U}$), $|\widetilde{H}_p| \geq 99|H_p|/100$. We begin with a key property, which is where the construction of $P_{\max}(T)$ enters the picture.

Lemma 8.9. *Let q be the child of some path p in $\mathcal{U}$, then $|H_p| \geq \frac{3}{2}|H_q|$. Moreover, if $p \neq r(\mathcal{U})$, then $|\widetilde{H}_p| \geq \frac{3}{2}|\widetilde{H}_q|$.*

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 first part of the lemma. For the second part, observe that if $p \neq r(\mathcal{U})$, then p is unsupported and so $|\widetilde{H}_p| \geq 99|H_p|/100$, and therefore the second part follows since $|H_q| \geq |\widetilde{H}_q|$. □

This immediately proves part (i) of Lemma 8.6. Part (ii) requires much more work.

We define a *residue* R_p for each $p \in \mathcal{U}$. Suppose p has children $q_1, q_2, \ldots, q_k$ in $\mathcal{U}$. Then $R_p = |\widetilde{H}_p| - \sum_i |\widetilde{H}_{q_i}|$. By definition, $|\widetilde{H}_p| = \sum_{q \in \mathcal{U}_p} R_p$. Note that R_p can be negative. Now, define $R_p^+ = \max(R_p, 0)$, and set $W_p = \sum_{q \in \mathcal{U}_p} R_p^+$. Observe that $W_p \geq |\widetilde{H}_p|$. We also get an approximate converse.

Claim 8.10. *For any path $p \in \mathcal{U}$, $|\widetilde{H}_p| \geq W_p - 2\sum_{q\in\mathcal{U}_p} |q|$.*

Proof. We write $W_p - |\widetilde{H}_p| = \sum_{q\in\mathcal{U}_p} R_q^+ - R_q = -\sum_{q\in\mathcal{U}_p : R_q<0} R_q$. Consider $q \in \mathcal{U}_p$ and denote the children in $\mathcal{U}_p$ by $q'_1, q'_2, \ldots$. Note that R_q is negative exactly when $|\widetilde{H}_q| < \sum_i |\widetilde{H}_{q'_i}|$. Traverse $P_{\max}(T)$ from q'_i to q. Other than q, all other nodes encountered are in the tall support chain of q'_i and hence do not affect its reduced heap. The vertices of $\widetilde{H}_{q'_i}$ that are deleted are exactly those present in the path q. Any vertex in q can be deleted from at most two of the reduced heaps (of the children of q in $\mathcal{U}_p$), since theses reduced heaps do not have an ancestor-descendant relationship. Therefore when R_q is negative, it is at most by $2|q|$. We sum over all q to complete the proof. □

The main challenge of the entire proof is bounding the sum of path lengths, which is done next. We stress that the all the previous work is mostly the setup for this claim.

Claim 8.11. *Fix any path $p \in \mathcal{U} \setminus \{r(\mathcal{U})\}$. Suppose for any $q, q' \in \mathcal{U}_p$ where q is a parent of q' in $\mathcal{U}_p$, $W_q \geq (4/3)W_{q'}$. Then $\sum_{q\in\mathcal{U}_p} |q| \leq W_p/20$.*

Proof. Since q is an unsupported short path, $|q| < \sqrt{|H_q|}/100 \leq \sqrt{|\widetilde{H}_q|}/99 \leq \sqrt{|W_q|}/99$. We prove that $\sum_{q\in\mathcal{U}_p} \sqrt{|W_q|}/99 \leq W_p/20$ by a charge redistribution scheme. Assume that each $q \in \mathcal{U}_p^-$ starts with $\sqrt{W_q}/99$ 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 R^+ values. In other words, give $(\sqrt{W_q}/99) \cdot (R^+_{q'}/W_q)$ units of charge to each $q' \in \mathcal{U}_q$.

After the redistribution, let us compute the charge deposited at q. Every ancestor in $\mathcal{U}_p^-$ $q = a_0, a_1, a_2, \ldots, a_k$ contributes to the charge at q. The charge is expressed in the following equation. We use the assumption that $W_{a_i} \geq (4/3)W_{a_{i-1}}$ and hence $W_{a_i} \geq (4/3)^i W_{a_0} \geq (4/3)^i$, as a_0 is an unsupported short path and hence $W_{a_0} \geq 1$.

$$(R_q^+/99)\sum_{a_i} 1/\sqrt{W_{a_i}} \leq (R_q^+/99)\sum_{a_i}(3/4)^{i/2} \leq R_q^+/20$$

The total charge is $\sum_{q\in\mathcal{U}_p} R_p^+/20 = W_p/20$. □

Corollary 8.12. *Let r be the root of $\mathcal{U}$, and suppose that for any paths $q, q' \in \mathcal{U} \setminus \{r\}$, where q is a parent of q' in $\mathcal{U}$, $W_q \geq (4/3)W_{q'}$. Then $\sum_{p\in\mathcal{U}} |p| \leq W_r/20 + |r|$.*

Proof. Let $c_1, \ldots, c_m$ be the children of r in $\mathcal{U}$. By definition, $W_r = \sum_i W_{c_i} + R_r^+ \geq \sum_i W_{c_i}$. By Claim 8.11, for each c_i we have $W_{c_i}/20 \geq \sum_{p\in\mathcal{U}_{c_i}} |p|$. Combining these to facts yields the claim. □

We wrap it all up by proving part (ii) of Lemma 8.6.

Claim 8.13. $\sum_{p\in\mathcal{U}} |p| \leq |H_{r(\mathcal{U})}|/10$.

Proof. We use r for $r(\mathcal{U})$. Suppose $W_q \geq (4/3)W_{q'}$ (for any choice in $\mathcal{U} \setminus \{r\}$ of q parent of q'), then by Corollary 8.12, $\sum_{p\in\mathcal{U}} |p| \leq W_r/20 + |r|$. By Claim 8.10, $|\widetilde{H}_r| \geq W_r - 2\sum_{p\in\mathcal{U}} |p|$, and so combining these inequalities gives,

$$\sum_{p\in\mathcal{U}} |p| \leq \frac{10}{9}\Big(|\widetilde{H}_r|/20 + |r|\Big) \leq \frac{10}{9}\Big(|H_r|/20 + \sqrt{|H_r|}/100\Big) \leq |H_r|/10.$$

We now prove that for any q parent of q' (other than r), $W_q \geq (4/3)W_{q'}$. Suppose not. Let p, p' be the counterexample furthest from the root, where p is the parent of p'. Note that for q and child

q' in $\mathcal{U}_{p'}$, $W_q \geq (4/3)W_{q'}$. We will apply Claim 8.11 for $\mathcal{U}_{p'}$ to deduce that $\sum_{q \in \mathcal{U}_{p'}} |q| \leq W_{p'}/20$. Combining this with Claim 8.10 gives, $|\widetilde{H}_{p'}| \geq 19W_{p'}/20$. By Lemma 8.9, $|\widetilde{H}_p| \geq (3/2)|\widetilde{H}_{p'}|$. Noting that $W_p \geq |\widetilde{H}_p|$, we deduce that $W_p \geq (4/3)W_{p'}$. Hence, p, p' is not a counterexample, and more generally, there is no counterexample. That completes the whole proof. □

8.4 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 8.14. *Consider a PL-Morse $f : \mathbb{M} \mapsto \mathbb{R}$, where the join tree $\mathcal{J}_C(\mathbb{M})$ has maximum degree 3. There is an algorithm to compute the join tree whose running time is $O(\sum_{p \in P_{\max}(\mathcal{J}_C)} |p| \log |p| + t\alpha(t) + N)$.*

Proof. By Theorem 7.9 we know that `build`$(\mathbb{M})$ correctly outputs $\mathcal{J}_C(\mathbb{M})$, and by Lemma 7.12 we know this takes $O(\sum_{v \in \mathcal{J}_C(\mathbb{M})} \log |H_v| + t\alpha(t) + N)$ time, where the H_v values are determined as in Definition 7.11. Therefore by Theorem 8.1, `build`$(\mathbb{M})$ takes $O(\sum_{p \in P_{\max}(\mathcal{J}_C)} |p| \log |p| + t\alpha(t) + N)$ time to correctly compute $\mathcal{J}_C(\mathbb{M})$. □

This result for join trees easily implies our main result, Theorem 1.2, which we now restate and prove.

Theorem 8.15. *Consider a PL-Morse $f : \mathbb{M} \mapsto \mathbb{R}$, where the contour tree $\mathcal{C} = \mathcal{C}(\mathbb{M})$ has maximum degree 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 $P(T)$ 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 we break it into extremum dominant pieces using `rain`$(\mathbb{M})$ (and in $O(|\mathbb{M}|)$ time by Theorem 5.7). Specifically, Lemma 5.5 proves that the output of `rain`$(\mathbb{M})$ is a set of extremum dominant pieces, $\mathbb{M}_1, \ldots, \mathbb{M}_k$, and Claim 5.8 shows that given the contour trees, $\mathcal{C}(\mathbb{M}_1), \ldots, \mathcal{C}(\mathbb{M}_k)$, 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 manifolds, computing the contour tree is roughly the same as computing the join tree. Specifically, Theorem 6.4 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). However, by the above theorem we know $\mathcal{J}_C(\mathbb{M}_i)$ can be computed in $O(\sum_{p \in P_{\max}(\mathcal{J}_C(\mathbb{M}_i))} |p| \log |p| + t_i\alpha(t_i) + N_i)$ (where t_i and N_i are the number of critical points and facets 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 decomposition across the extremum dominant pieces, $P_{\max}(\mathcal{C}(\mathbb{M})) = \cup_{i=1}^{k} P_{\max}(\mathcal{J}_C(\mathbb{M}_i))$. 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

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

□

9 Lower Bound by Path Decomposition

We first prove a lower bound for join trees, and then generalize to contour trees.

9.1 Join Trees

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

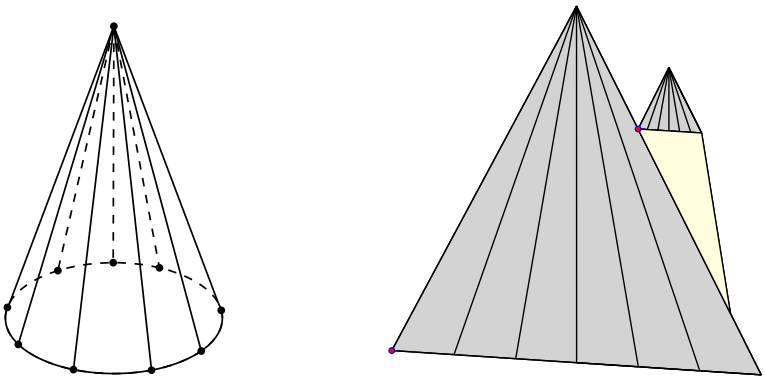

Figure 5: Left: angled view of a diamond / Right: a parent and child diamond put together

Lemma 9.1. *Fix a valid 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 join tree.*

Proof. We describe the basic building block of these terrains, which corresponds to a fixed path $p \in P$. Informally, a *diamond* 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 diamond, 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 each vertex in the cycle.

Now we describe how to attach two different diamonds. In this process, we glue the base of a scaled down "child" diamond on to a triangular cone face of the larger "parent" diamond (see Figure 5). Specifically, the anchor of the child diamond is attached directly to a face of the parent diamond at some height h. The remaineder of the base of the child cone is then extended down (at a slight angle) until it hits the face of the parent.

The full terrain is obtained by repeatedly gluing diamonds. For each path $p_i \in P$, we create a diamond of size $|p_i| + 1$. The two faces adjacent to the anchor are always empty, and the remaining faces are for gluing on other diamonds. (Note that diamonds 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 diamonds 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 diamond for p as described above. (Naturally for this construction to work, diamonds for a given path will be scaled down relative to the size of the diamond 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 diamonds. We create a global boundary minimum by setting the vertices at the base of the diamond representing the root of $P_{\mathcal{S}}$ all to the same height (and

there are no other minima). Therefore, the saddles on a given diamond will appear contiguously on a root to leaf path in the join tree of the terrain, where the leaf corresponds to the maxima of the diamond (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 diamond. In particular, there are $(|p|-1)!$ possible orderings of the heights of the saddles on the diamond for p, 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 diamond will appear in sorted order in the join tree. So, any permutation of the heights on a given diamond corresponds to a permutation of the vertices along a path in P. □

Two path decompositions P_1 and P_2 (of potentially different complexes and/or height functions) are equivalent if: there is a 1-1 correspondence between the sizes of the constituent paths, and the shrubs are isomorphic.

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

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 defined by f. Consider any join on a face of some diamond, which is the anchor point of some connecting child diamond. The join has two up-stars, each of which has exactly one edge. Each edge connects to a maxima and must be colored by that maxima. Hence, two colors touching this join (according to Definition 7.11) are the colors of the apexes of the child and parent diamond.

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 maxima in the subtree at w_2 cannot touch any ancestor of v. This subtree is exactly the contour tree of the child diamond attached at v. The base of this diamond is completely contained in a face of the parent diamond. So all colors from the child "drain off" to the base of the parent, and do not touch any joins on the parent diamond.

Hence, $|H_v|$ is at most the size of the path in P containing v. By Lemma 7.12, the total number of heap operations is at most $\sum_v \log |H_v|$, completing the proof.

□

The following is the equivalent of Theorem 1.3 for join trees, and immediately follows from the previous lemmas.

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 contour tree (on $\mathbf{F}_P$) requires $\Omega(\sum_{p \in P} |p| \log |p|)$ time. Furthermore, our algorithm makes $O(\sum_{p \in P} |p| \log |p|)$ 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 average depth in this tree is $\Omega(\sum_{p_i \in P} |p_i| \log |p_i|)$. Lemma 9.2 completes the proof. □

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 §5 (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 maxima 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.[4]

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

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

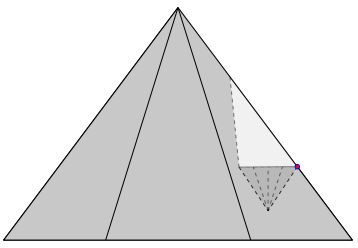

Figure 6: A child diamond attached to a parent diamond with opposite orientation.

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

By construction, the diamonds 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 diamonds are covered except for the one corresponding to the path containing the root r_i. If r_i corresponds to the initial maxima 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 diamond for r_i to a free face of a diamond 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 diamond produces a surface with a distinct join tree. The specific bound on the size of $\mathbf{F}$, defining these distinct join 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. □

Theorem 1.3 follows immediately from an entropy argument, analogous to Theorem 9.3.

Remark 9.7. 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 Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[ABC09] P. Afshani, J. Barbay, and T. Chan. Instance-optimal geometric algorithms. In *Proceedings of the Foundations of Computer Science (FOCS)*, pages 129–138, 2009. 2

[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. 1

[BM12] J.-D. Boissonnat and C. Maria. The simplex tree: An efficient data structure for general simplicial complexes. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 731–742, 2012. 4

[BR63] R. Boyell and H. Ruston. Hybrid techniques for real-time radar simulation. In *Proceedings of Fall Joint Computer Conference*, pages 445–458, 1963. 2

[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. 2

[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. 2

[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. 2

[Car04] H. Carr. *Topological Manipulation of Isosurfaces.* PhD thesis, University of British Columbia, 2004. 2, 3, 11

[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. 1, 3

[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. 3

[CSA00] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Proceedings of the Symposium on Discrete Algorithms*, pages 918–926, 2000. 1, 3, 5, 6, 8, 14, 15

[DN09] H. Doraiswamy and V. Natarajan. Efficient algorithms for computing reeb graphs. *Computational Geometry: Theory and Applications*, 42:606–616, 2009. 3

[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. 3, 6

[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. 2

[HE10] J. Harer and H. Edelsbrunner. *Computational Topology.* AMS, 2010. 3, 5

[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. 3

[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. 2

[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. 2

[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. 3

[PCM02] V. Pascucci and K. Cole-McLaughlin. Efficient computation of the topology of level set. In *IEEE Visualization*, pages 187–194, 2002. 3

[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. 3

[RM00] J. B. T. M. Roerdink and A. Meijster. The watershed transform: definitions, algorithms, and parallelization strategies. *Fundamenta Informaticae*, 41:187–228, 2000. 7

[SK91] Y. Shinagawa and T. Kunii. Constructing a reeb graph automatically from cross sections. *IEEE Comput. Graphics Appl.*, 11(6):44–51, 1991. 3

[ST83] D. Sleator and R. Tarjan. A data structure for dynamic trees. *Journal of Computing and System Sciences*, 26(3):362–391, 1983. 10

[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. 3, 6

[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. 3

[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. 2

[Vui78] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978. 9