Avoiding the Global Sort: A Faster Contour Tree Algorithm*

Benjamin Raichel[1] and C. Seshadhri[2]

1 **Department of Computer Science, University of Texas at Dallas, Richardson, USA**
benjamin.raichel@utdallas.edu

2 **Department of Computer Science, University of California, Santa Cruz, USA**
scomandu@ucsc.edu

Abstract

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

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

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

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, I.1.2 Algorithms, I.3.5 Computational Geometry and Object Modeling

Keywords and phrases contour trees, computational topology, computational geometry

Digital Object Identifier 10.4230/LIPIcs.SoCG.2016.57

1 Introduction

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

* The full updated version of this paper is available on the arXiv [20].

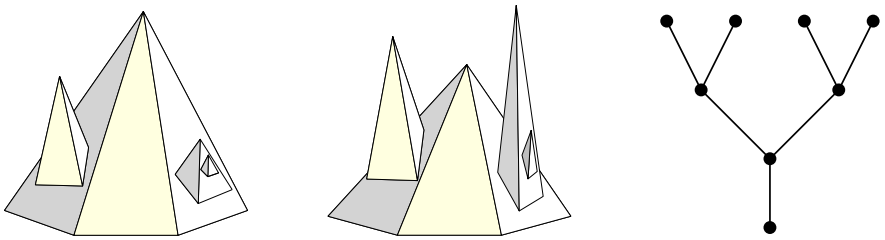

Figure 1 Two surfaces with different orderings of the maxima, but the same contour tree.

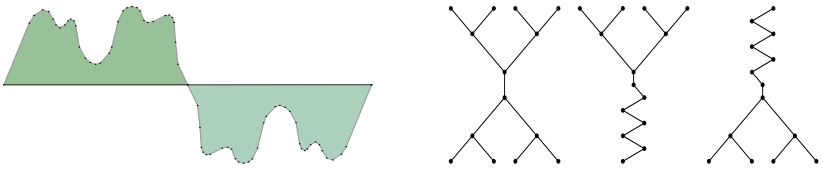

Figure 2 On left, a surface with a balanced contour tree, but whose join and split trees have long tails. On right (from left to right), the contour, join and split trees.

each other or split into new components, contours acquire or lose handles. The contour tree is a concise representation of all these events. Throughout we follow the definition of contour trees from [24] which includes all changes in topology. For $d > 2$, some subsequent works, such as [7], only include changes in the number of components.

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 of the contour tree 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. Leaves are either maxima or minima, and internal nodes are either "joins" or "splits".

Consider $f : \mathbb{M} \to \mathbb{R}$, where $\mathbb{M}$ is a triangulated mesh with n vertices, N faces in total, and $t \leq n$ critical points. (We assume that $f : \mathbb{M} \to \mathbb{R}$ a linear interpolant over distinct valued vertices, where the contour tree T has maximum degree 3. The degree assumption simplifies the presentation, and is commonly made [24].) 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 [7] (where $\alpha(\cdot)$ denotes the inverse Ackermann function). In practical applications, N is typically $\Theta(n)$ (certainly true for $d = 2$). The most expensive operation is an initial sort of all the vertex heights. Chiang *et al.* build on this approach to get a faster algorithm that only sorts the critical vertices, yielding a running time of $O(t \log t + N)$ [8]. Common applications for contour trees involve turbulent combustion or noisy data, where the number of critical points is likely to be $\Omega(n)$. There is a worst-case lower bound of $\Omega(t \log t)$ by Chiang *et al.* [8], based on a construction of Bajaj *et al.* [1].

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

Our main result gives an affirmative answer. Remember that we can consider the contour tree as directed from top to bottom. For any node v in the tree, let ℓ_v denote the length of the longest directed path passing through v.

▶ **Theorem 1.** *Consider a simplicial complex $f : \mathbb{M} \to \mathbb{R}$, described as above, and denote the contour tree by T with vertex set (the critical points) $C(T)$. There exists an algorithm to compute T in $O(\sum_{v \in C(T)} \log \ell_v + t\alpha(t) + N)$ time. Moreover, this algorithm only compares function values at pairs of vertices that are ancestor-descendant in T or adjacent in $\mathbb{M}$.*

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

1.1 A refined bound with optimality properties

Theorem 1 is a direct corollary of a stronger but more cumbersome theorem.

▶ **Definition 2.** For a contour tree T, a *leaf path* is any path in T containing a leaf, which is also monotonic in the height values of its vertices. Then a *path decomposition*, $P(T)$, is a partition of the vertices of T into a set of vertex disjoint leaf paths.

▶ **Theorem 3.** *There is a deterministic algorithm to compute the contour tree, T, whose running time is $O(\sum_{p \in P(T)} |p| \log |p| + t\alpha(t) + N)$, where $P(T)$ is a specific path decomposition (constructed implicitly by the algorithm). The number of comparisons made is $O(\sum_{p \in P(T)} |p| \log |p| + N)$. In particular, comparisons are only made between pairs of vertices that are ancestor-descendant in T or adjacent in $\mathbb{M}$.*

Note that Theorem 1 is a direct corollary of this statement. For any v, ℓ_v is at least the length of the path in $P(T)$ that contains v. This bound is strictly stronger, since for any balanced contour tree, the run time bound of Theorem 3 is $O(t\alpha(t) + N)$, and $O(N)$ comparisons are made. (Since for a balanced tree, one can show $\sum_{p \in P(T)} |p| \log |p| = O(t)$.)

The bound of Theorem 3 may seem artificial, since it actually depends on the $P(T)$ that is implicitly constructed by the algorithm. Nonetheless, we prove that the algorithm of Theorem 3 has strong optimality properties. For convenience, fix some value $t = \Omega(N)$, and consider the set of terrains ($d = 2$) with t critical points. The bound of Theorem 3 takes values ranging from t to $t \log t$. Consider some $\gamma \in [t, t \log t]$, and consider the set of terrains where the algorithm makes γ comparisons. Then *any algorithm* must make roughly γ comparisons in the worst-case over this set. (Further details are in the full version [20].)

▶ **Theorem 4.** *There exists some absolute constant α such that the following holds. For sufficiently large t and any $\gamma \in [t, t \log t]$, consider the set $\boldsymbol{F}_\gamma$ of terrains with t critical points such that the number of comparisons made by the algorithm of Theorem 3 on these terrains is in $[\gamma, \alpha\gamma]$. Any algebraic decision tree that correctly computes the contour tree on all of $\boldsymbol{F}_\gamma$ has a worst case running time of $\Omega(\gamma)$.*

1.2 Previous Work

Contour trees were first used to study terrain maps by Boyell and Ruston, and Freeman and Morse [3, 13]. Contour trees have been applied in analysis of fluid mixing, combustion

simulations, and studying chemical systems [15, 4, 2, 5, 16]. Carr's thesis [6] gives various applications of contour trees for data visualization and is an excellent reference for contour tree definitions and algorithms.

The first formal result was an $O(N \log N)$ time algorithm for functions over 2D meshes and an $O(N^2)$ algorithm for higher dimensions, by van Kreveld *et al.* [24]. Tarasov and Vyalya [22] improved the running time to $O(N \log N)$ for the 3D case. The influential paper of Carr *et al.* [7] improved the running time for all dimensions to $O(n \log n + N\alpha(N))$. Pascucci and Cole-McLaughlin [18] provided an $O(n + t \log n)$ time algorithm for 3-dimensional structured meshes. Chiang *et al.* [8] provide an unconditional $O(N + t \log t)$ algorithm.

Contour trees are a special case of Reeb graphs, a general topological representation for real-valued functions on any manifold. Algorithms for computing Reeb graphs is an active topic of research [21, 9, 19, 10, 14, 17], where two results explicitly reduce to computing contour trees [23, 11].

2 Contour tree basics

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

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

▶ **Definition 5.** The function f is *boundary critical* if the following holds. Consider a boundary facet F. All vertices of F have the same function value. Furthermore, all neighbors of vertices in F, which are not also in F itself, either have all function values strictly greater than or all function values strictly less than the function value at F.

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, including the infinite facet, which is required to be a boundary facet. However, we allow for other boundary facets, to capture the resulting surface pieces after our algorithm makes a horizontal cut.

We think of the dimension d, as constant, and assume that $\mathbb{M}$ is represented in a data structure that allows constant-time access to neighboring simplices in $\mathbb{M}$ (e.g. incidence graphs [12]). (This is analogous to a doubly connected edge list, but for higher dimensions.) Observe that $f : \mathbb{M} \to \mathbb{R}$ can be thought of as a d-dimensional simplicial complex living in $\mathbb{R}^{d+1}$, where $f(x)$ is the "height" of a point $x \in \mathbb{M}$, which is encoded in the representation of $\mathbb{M}$. Specifically, rather than writing our input as $(\mathbb{M}, f)$, we abuse notation and typically just write $\mathbb{M}$ to denote the lifted complex.

▶ **Definition 6.** The *level set* at value h is the set $\{x | f(x) = h\}$. A *contour* is a connected component of a level set. An *h-contour* is a contour where f-values are h.

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

▶ **Definition 7.** The *Morse up-degree* of x is the number of $(f(x)+\delta)$-contours of $f|B_\varepsilon(x)$ as $\delta, \varepsilon \to 0^+$. The *Morse down-degree* is the number of $(f(x)-\delta)$-contours of $f|B_\varepsilon(x)$ as $\delta, \varepsilon \to 0^+$.

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

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

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

▶ **Definition 8.** Two regular contours ψ and ψ' are *equivalent* if there exists an f-monotone path p connecting a point in ψ to ψ', such that no $x \in p$ belongs to a critical contour.

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

▶ **Definition 9.** The *contour tree* is the graph on vertex set $\mathcal{V} = \mathcal{V}(f)$, where edges are formed as follows. For every contour class that is created at v_i and destroyed v_j, there is an edge (v_i, v_j). (Conventionally, edges are directed from higher to lower function value.)

We denote the contour tree of $\mathbb{M}$ by $\mathcal{C}(\mathbb{M})$. The corresponding node and edge sets are denoted as $\mathcal{V}(\cdot)$ and $\mathcal{E}(\cdot)$. It is not immediately obvious that this graph is a tree, but alternate definitions of the contour tree in [7] 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 $\mathcal{C}(\mathbb{M})$. Similar to [24] (among others), multi-saddles are treated as a set of ordinary saddles, which can be realized via vertex unfolding (which can increase surface complexity if multi-saddle degrees are allowed to be super-constant). Therefore, to simplify the presentation, for the remainder of the paper up and down-degrees are at most 2, and total degree is at most 3.

See the full version [20] for further technical remarks on the above definitions.

3 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 a contour ϕ, roughly speaking, this constructs the simplicial complex $\mathbb{M} \setminus \phi$. We will always enforce the condition that ϕ never passes through a vertex of $\mathbb{M}$. Again, we use ε for an infinitesimally small value. We denote ϕ^+ (resp. ϕ^-) to be the contour at value $f(\phi) + \varepsilon$ (resp. $f(\phi) - \varepsilon$), which is at distance ε from ϕ.

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

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

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

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

The following theorems are intuitively obvious, and are proven in the full version [20].

▶ **Theorem 10.** *For any regular contour ϕ, the output of* $\texttt{surgery}(\mathbb{M}, \phi)$ *is* $\mathcal{C}(\mathbb{M})$.

▶ **Theorem 11.** $\texttt{cut}(\mathbb{M}, \phi)$ *consists of two disconnected simplicial complexes.*

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

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

▶ **Definition 13.** Fix $x \in \mathbb{M}$. The set of points $y \in \mathbb{M}$ such that there is a non-ascending path from x to y is denoted by $\texttt{wet}(x, \mathbb{M})$ (which in turn is represented as a simplicial complex). A point z is at the *interface* of $\texttt{wet}(x, \mathbb{M})$ if every neighborhood of z has non-trivial intersection with $\texttt{wet}(x, \mathbb{M})$ (i.e. the intersection is neither empty nor the entire neighborhood).

The following claim gives a description of the interface.

▶ **Claim 14.** *For any x, each component of the interface of* $\texttt{wet}(x, \mathbb{M})$ *contains a join vertex.*

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

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

Let Γ_y be the contour containing y. Suppose for contradiction that $\forall p \in \Gamma_y$, p has up-degree 1 (see Definition 7). Consider the non-ascending path from x to y and let z be the first point of Γ_y encountered. There exists a wet, regular $(f(y) + \delta)$-contour ψ intersecting

$B_\varepsilon(z)$. Now, walk from z to y along Γ_y. If all points w in this walk have up-degree 1, then ψ is the unique $(f(y)+\delta)$-contour intersecting $B_\varepsilon(w)$. This would imply that $\phi = \psi$, contradicting the fact that ψ is wet and ϕ is dry. ◀

Note that $\texttt{wet}(x, \mathbb{M})$ (and its interface) can be computed in time linear in the size of the wet simplicial complex. We perform a non-ascending search from x. Any face F of $\mathbb{M}$ encountered is partially (if not entirely) in $\texttt{wet}(x, \mathbb{M})$. The wet portion is determined by cutting F along the interface. Since each component of the interface is a contour, this is equivalent to locally cutting F by a hyperplane. All these operations can be performed to output $\texttt{wet}(x, \mathbb{M})$ in time linear in $|\texttt{wet}(x, \mathbb{M})|$.

We define a simple `lift` operation on the interface components. Consider such a component ϕ containing a join vertex y. Take any dry increasing edge incident to y, and pick the point z on this edge at height $f(y)+\delta$ (where δ is an infinitesimal, but larger than the value ε used in the definition of `cut`). Let $\texttt{lift}(\phi)$ be the unique contour through the regular point z. Note that $\texttt{lift}(\phi)$ is dry. The following claim follows directly from Theorem 11.

▶ **Claim 15.** *Let ϕ be a connected component of the interface. Then $\texttt{cut}(\mathbb{M}, \texttt{lift}(\phi))$ results in two disjoint simplicial complexes, one consisting entirely of dry points.*

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. When we start, rain flows downwards. In each recursive call, the direction of rain is *switched* to the opposite direction. This is crucial to ensure a linear running time. The switching is easily implemented by inverting a complex $\mathbb{N}'$, achieved by negating the height values. We can now let rain flow downwards, as it usually does in our world.

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

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

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

▶ **Lemma 16.** *Each output $\mathbb{M}_i$ is extremum dominant.*

Proof. Consider a call to $\texttt{rain}(x, \mathbb{N})$. If the interface is empty, then all of $\mathbb{N}$ is in $\texttt{wet}(x, \mathbb{N})$, so $\mathbb{N}$ is trivially extremum dominant. So suppose the interface is non-empty and consists of $\phi_1, \ldots, \phi_k$ (as denoted in the procedure). By repeated applications of Claim 15, $\mathbb{N}_{k+1}$ contains $\texttt{wet}(x, \mathbb{M})$. Consider $\texttt{wet}(x, \mathbb{N}_{k+1})$. The interface is exactly $\phi_1, \ldots, \phi_k$. So the only dry vertices are those in the boundaries $B_1, \ldots, B_k$. But these boundaries are maxima. ◀

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

▶ **Lemma 17.** *A face $F \in \mathbb{M}$ is cut*[1] *at most once during* `rain`$(x, \mathbb{M})$.

Proof. Notation here follows the pseudocode of `rain`. First, by Theorem 11, 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. `cut`$(\mathbb{N}_i, \phi_i')$ will cut F with two horizontal cutting planes, one ε above ϕ_i' and one ε below ϕ_i'. This breaks F into lower and upper portions which are then triangulated (there is also a discarded middle portion). The lower portion, which is adjacent to ϕ_i, gets included in $\mathbb{N}_{k+1}$, the complex containing the wet points, and hence does not participate in any later recursive calls. The upper portion (call it U) is in $\mathbb{L}_i$. Note that the lower boundary of U is in the boundary B_i. Since a recursive call is made to `rain`$(B_i, \mathbb{L}_i)$ (and $\mathbb{L}_i$ is inverted), U becomes wet. Hence U, and correspondingly F, will not be subsequently cut. ◀

The following are direct consequences of Lemma 17 and the `surgery` procedure.

▶ **Theorem 18.** *The total running time of* `rain`$(x, \mathbb{M})$ *is* $O(|\mathbb{M}|)$.

▶ **Claim 19.** *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.*

5 Contour trees of extremum dominant complexes

The previous section allows us to restrict attention to extremum dominant complexes. We will orient so that the extremum in question is always a *minimum*. We will fix such a simplicial complex $\mathbb{M}$, with the dominant minimum m^*. 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 contour trees of minimum dominant complexes have a simple description. Intuitively, the cutting procedure of Section 4 introduced "stumps" where we made cuts, which is why we allowed for non-dominant minima and maxima in the definition of extremum dominant complexes. The punchline is that, ignoring these stumps, the contour tree of an extremum dominant complex is equivalent to its *join* (or *split*) tree, defined in [7]. Hence the critical join tree below is just the join tree without these stumps. Additional definitions and proofs are given in the full version [20].

▶ **Definition 20.** The *critical join tree* $\mathcal{J}_C(\mathbb{M})$ is built on the set V' of all critical points other than the non-dominant minima. The directed edge (u, v) is present when u is the smallest valued vertex in V' in a connected component of $\mathbb{M}_v^+$ and v is adjacent (in $\mathbb{M}$) to a vertex in this component. The *join tree*, $\mathcal{J}(\mathbb{M})$, is defined analogously, but instead on $\mathcal{V}(\mathbb{M})$.

Each non-dominant minimum, m_i, connects to the contour tree at a corresponding split s_i. We have the following (see the full version [20] for more details).

▶ **Theorem 21.** *Let $\mathbb{M}$ have a dominant minimum. The contour tree $\mathcal{C}(\mathbb{M})$ consists of all edges $\{(s_i, m_i)\}$ and $\mathcal{J}_C(\mathbb{M})$.*

▶ **Remark 22.** The above theorem, combined with the previous sections, implies that in order to get an efficient contour tree algorithm, it suffices to have an efficient algorithm for computing $\mathcal{J}_C(\mathbb{M})$. Due to minor technicalities, it is easier to phrase the following section

[1] Technically what we are calling a single cut is done with two hyperplanes.

instead in terms of computing $\mathcal{J}(\mathbb{M})$ efficiently. Note however that for minimum dominant complexes output by `rain`, converting between $\mathcal{J}_C$ and $\mathcal{J}$ is trivial, as $\mathcal{J}$ is just $\mathcal{J}_C$ with each non-dominant minimum m_i augmented along the edge leaving s_i.

6 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 maximum x. We will spill different paint from each maximum and watch it flow down. This is analogous to the raining of Section 4, but paint is a much more viscous liquid. *So paint only flows down edges, and it does not color the interior of higher dimensional faces.* Furthermore, paints do not mix, so every edge of $\mathbb{M}$ gets a unique color. This process (and indeed the entire algorithm) works purely on the 1-skeleton of $\mathbb{M}$, which is just a graph.

▶ **Definition 23.** Let the 1-skeleton of $\mathbb{M}$ have edge set E and maxima X. A *painting* of $\mathbb{M}$ is a map $\chi : X \cup E \to [|X|]$, where $\chi(z)$ is referred to as the *color* of z, with the following property. Consider an edge e. There exists a descending path from some maximum x to e consisting of edges in E, such that all edges along this path have the same color as x.

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

▶ **Definition 24.** 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 v is incident to a c-colored edge with v at the lower endpoint. For v, $col(v)$ is the set of colors that touch v.
- A color $c \in col(v)$ *fully touches* v if all edges in an up-star are colored c.
- For any maximum $x \in X$, we say that x is both touched and fully touched by $\chi(x)$.

6.1 The data structures

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

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

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

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

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

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

6.2 The algorithm

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

▶ **Definition 25.** A vertex v is *ripe* if: for all $c \in col(v)$, v is present in $T(rep(c))$ and is also the highest vertex in this 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 an arbitrary color c touching v. Insert v into $T(c)$.
5. Initialize $rep(c) = c$ and set $att(c)$ to be the unique maximum colored c.
6. Initialize K to be an empty stack.

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

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

`update`(K)

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

A few simple facts:

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

▶ **Observation 26.** Each unprocessed vertex is always in exactly one queue of the colors in each of its up-stars. Specifically, for a given up-star of a vertex v, `init`($\mathbb{M}$) puts v into the

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

6.3 Proving correctness

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

▶ **Lemma 27.** *Suppose vertex v is connected to a component $\mathbb{P}$ of $\mathbb{M}_v^+$ by an edge e which is currently colored c. At all times: either all edges in $\mathbb{P}$ are currently colored c, or there exists a critical vertex $w \in \mathbb{P}$ 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 $B_\varepsilon(w)$ and intersects 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 28.** *Each time $\texttt{update}(K)$ is called, it terminates with a ripe vertex on top of the stack.*

Proof. $\texttt{update}(K)$ is only called if there are unprocessed vertices remaining, and so by the time we reach step 3 in $\texttt{update}(K)$, 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 26). Now by Lemma 27, 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. ◀

▶ **Claim 29.** *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 27, 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 30.** *The partial output on the processed vertices is exactly the restriction of $\mathcal{J}(\mathbb{M})$ to these vertices.*

Proof. More generally, we prove the following: all outputs on processed vertices are edges of $\mathcal{J}(\mathbb{M})$ and for any 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 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 28 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 29, 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 $\mathcal{J}(\mathbb{M})$ 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 $\mathcal{J}(\mathbb{M})$, 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 31.** *Given an input complex $\mathbb{M}$,* `build`$(\mathbb{M})$ *terminates and outputs $\mathcal{J}(\mathbb{M})$.*

Proof. First observe that each vertex can be processed at most once by `build`$(\mathbb{M})$. By Corollary 28, 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 30 the algorithm will terminate having computed $\mathcal{J}(\mathbb{M})$. ◀

6.4 Upper Bounds for Running Time

The algorithm `build`$(\mathbb{M})$ processes vertices in $\mathcal{J}(\mathbb{M})$ one at a time. The main processing cost comes from priority queue operations. The cost of these operations is a function of the size of the queue which is in turn a function of the choices made by the subroutine `init`$(\mathbb{M})$.

▶ **Definition 32.** A *leaf assignment* χ of a binary tree T assigns *two* distinct leaves to each internal vertex v, one from the left and one from the right subtree of v (or only one leaf if v has only one child). The subroutine `init`$(\mathbb{M})$ naturally defines a leaf assignment to $\mathcal{J}(\mathbb{M})$ (which is a rooted binary tree) according to the priority queue for each up-star we put a given vertex in. Call this the *initial coloring* of the vertices in $\mathcal{J}(\mathbb{M})$, and denote it by χ.

For a vertex v in $\mathcal{J}(\mathbb{M})$, let $L(v)$ denote the set of leaves of the subtree rooted at v, and let $A(v)$ denote the set of ancestors of v, i.e. the vertices on the v to root path. For a

vertex $v \in \mathcal{J}(\mathbb{M})$, and an initial coloring χ, we use H_v to denote the *heap* at v. Formally, $H_v = \{u | u \in A(v), \chi(u) \cap L(v) \neq \emptyset\}$, i.e. the set of ancestors colored by some leaf in $L(v)$.

Given the above technical definition, the proof of the following lemma is straightforward, though long and so has been moved to the full version [20].

▶ **Lemma 33.** *Let $\mathbb{M}$ be a simplicial complex with t critical points. The running time of* `build`$(\mathbb{M})$ *is $O(N + t\alpha(t) + \sum_{v \in \mathcal{J}(\mathbb{M})} \log |H_v|)$, where H_v is defined by an initial coloring.*

Our main result, Theorem 1, is an easy corollary of the above lemma.

Proof of Theorem 1. Consider a critical point v of the initial input complex. By Theorem 11 this vertex appears in exactly one of the pieces output by `rain`. As in the Theorem 1 statement, let ℓ_v denote the length of the longest directed path passing through v in the contour tree of the input complex, and let ℓ'_v denote the longest directed path passing through v in the join tree of the piece containing v. By Theorem 10, ignoring non-dominant extrema introduced from cutting (whose cost can be charged to a corresponding saddle), the join tree on each piece output by `rain` is isomorphic to some connected subgraph of the contour tree of the input complex, and hence $\ell'_v \leq \ell_v$. Moreover, $|H_v|$ only counts vertices in a v to root path and so trivially $|H_v| \leq \ell'_v$, implying Theorem 1. ◀

Note that there is fair amount of slack in this argument as $|H_v|$ may be significantly smaller than ℓ'_v. This slack allows for the more refined upper and lower bounds mentioned in Section 1.1. Quantifying this slack however is quite challenging, and requires a significantly more sophisticated analysis involving path decompositions, detailed in the full version [20].

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

References

1 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.

2 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.

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

4 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.

5 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.

6 H. Carr. *Topological Manipulation of Isosurfaces.* PhD thesis, University of British Columbia, 2004.

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

8 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. doi:10.1016/j.comgeo.2004.05.002.

9 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. doi:10.1145/777792.777844.

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

11 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.

12 H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer, 1987.

13 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.

14 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.

15 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.

16 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.

17 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.

18 V. Pascucci and K. Cole-McLaughlin. Efficient computation of the topology of level set. In *IEEE Visualization*, pages 187–194, 2002. doi:10.1109/VISUAL.2002.1183774.

19 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.

20 B. Raichel and C. Seshadhri. Avoiding the global sort: A faster contour tree algorithm. *CoRR*, abs/1411.2689, 2014.

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

22 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. doi:10.1145/276884.276892.

23 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.

24 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. doi:10.1145/262839.269238.