ELSEVIER

Computational Geometry 24 (2003) 75–94

Computational Geometry
Theory and Applications

www.elsevier.com/locate/comgeo

Computing contour trees in all dimensions

Hamish Carr [a], Jack Snoeyink [b,*], Ulrike Axen [c]

[a] *Department of Computer Science, University of British Columbia, Vancouver, BC, Canada*
[b] *Department of Computer Science, University of North Carolina at Chapel Hill, NC, USA*
[c] *School of EECS, Washington State Univ., Pullman, WA, USA*

Received 26 January 2001; received in revised form 8 August 2001; accepted 15 August 2001

Communicated by P. K. Agarwal and L. Arge

Abstract

We show that contour trees can be computed in all dimensions by a simple algorithm that merges two trees. Our algorithm extends, simplifies, and improves work of Tarasov and Vyalyi and of van Kreveld et al.

Keywords: Iso-surfaces; Simplicial meshes; Morse theory; Resolving singularities

1. Introduction

Many imaging technologies and scientific simulations produce data in the form of sample points with intensity values. One way to visualize this data is to convert it into geometric models by thresholding or by taking level sets. In this paper, we focus on one tool that can help in choosing threshold values or in interactive exploration of such data: the *contour tree*.

Contour trees were used by van Kreveld et al. [28] to compute isolines on terrain maps in geographic information systems. With terrain maps, a surface model is computed from elevation values at sample points in the plane. Isolines, often called contours, are the curves that can be seen on a topographic map, and consist of points at a given height. Contours can be traced from a surface model relatively easily, given a starting point, or "seed" on each. Van Kreveld et al. use the contour tree to generate "seed sets" for any query height value, guaranteeing that each contour has at least one seed.

We use the contour tree to compute seed sets, to trace whole or partial isosurfaces in $\mathbb{R}^3$, and to determine important values of the height function where topological changes occur in the level sets;

* Corresponding author.
E-mail addresses: hcarr@cs.ubc.ca (H. Carr), snoeyink@cs.unc.edu (J. Snoeyink), axen@eecs.wsu.edu (U. Axen).

0925-7721/02/$ – see front matter
PII: S0925-7721(02)00093-7

these changes may correspond to important phenomena in the data studied. While van Kreveld et al. do discuss the extension of their approach to $\mathbb{R}^3$, their algorithm runs in quadratic time, which is prohibitive.

Tarasov and Vyalyi [26] gave an $O(N \log N)$ algorithm for computing contour trees in $\mathbb{R}^3$, where N is the number of simplices in the decomposition of the data. We describe their algorithm and the handling of singularities in more detail later, but their approach can multiply the number of simplices by a factor of 360, and is difficult to implement.

Our algorithm for contour trees begins with Tarasov and Vyalyi's idea of three passes through the data, but makes the following simplifications and improvements. The first two sweeps do not maintain level sets, but construct "join" and "split" trees, which store partial topological information about the data. We then apply a simple merge procedure to obtain the contour tree. The resulting algorithm handles multiple singularities and extends to all dimensions. Because there are some applications in which multiple singularities must be replaced by simple singularities, we also observe that Tarasov and Vyalyi's approach to resolving singularities can be extended to all dimensions.

After reviewing isosurfaces in Section 2, we define contour trees and look at their properties in Section 3. We then give our algorithm to construct contour trees in Section 4. Finally, in Section 5, we extend Tarasov and Vyalyi's resolution of singularities to arbitrary dimensions. We state our conlusions in Section 6, and give some future directions in Section 7.

2. Isosurfaces

Suppose that we are given a set of n points $\{p_1, p_2, \ldots, p_n\}$ in a fixed-dimensional space $\mathbb{R}^d$, with corresponding scalar measurements $\{h_1, h_2, \ldots, h_n\}$. We assume that the h_i are unique, perhaps by perturbation of our data using simulation of simplicity [10].

To extend the data to the entire space, we choose a mesh M with vertex set $\{p_1, p_2, \ldots, p_n\}$. Meshes used for isosurfaces include regular rectilinear meshes (also known as *voxels* or *cuberilles*) [1,8,9,13,15, 16,18,19,29,30], regular simplicial meshes [26,28,29,31], and irregular meshes [14,18]. We then choose a continuous function f to interpolate at points not in $\{p_1, p_2, \ldots, p_n\}$, and require that $f(p_i) = h_i$: this is typically a piecewise-linear function. For convenience, we use *height* to refer to the function value.

A *level set of f at some height h* is the set $\{x \in \mathbb{R}^d \mid f(x) = h\}$, and may consist of $0, 1$, or more connected components. In 2-D, these connected components are called *isolines*, and in 3-D, *isosurfaces*. We use *contour* as a general term for a connected component of a level set in a space of arbitrary dimension.

If we think of the height $f(x)$ as time and watch the evolution of the level sets of f over time, then we see contours appear, split, change genus, join, and disappear. The *contour tree*, which we define in Section 3, is a graph that tracks contours of the level set as they split and appear or join and disappear.

2.1. Previous work on isosurfaces

Isosurfaces have been widely used for segmentation and rendering, in fields such as medical imaging [1,18,19], fluid dynamics [18], and X-ray crystallography [9,13]. The principal algorithm used to generate isosurfaces is the "Marching Cubes" algorithm [19], which computes the desired level set by finding the intersection of the level set with each cell of the mesh. This algorithm has several disadvantages: the isosurface generated may have visible cracks, the time required to render an isosurface is $O(N)$ in

the number of cells, and the algorithm fails to distinguish between the contours of the level set. The first disadvantage, that visible cracks appear in the generated model, can be dealt with by Nielsen and Hamann's Asymptotic Decider [21], or by subdividing the cells into simplices (in $\mathbb{R}^3$, tetrahedra). As we see in Section 3.1, a simplicial mesh is required for the contour tree algorithm, so we choose to subdivide the cells into simplices.

As regards the run-time, various techniques have been proposed to reduce the cost of generating an individual isosurface to as little as $O(\log N + k)$, where k is an output-sensitive term. These techniques include octrees [30], span space [18], interval trees [7–9], extrema graphs [15,16], segment trees [2,14], and contour trees [3,28]. Of these, octrees, span space, and interval trees require large run-time data structures ($\Theta(N)$ in the size of the mesh), and retain the inability to distinguish contours of the level set. One reason for this is that these techniques fail to take advantage of the fact that each contour must be connected: each intersection of a cell and a contour is treated as a separate object.

In contrast, the extrema graph, segment tree, and contour tree approaches take advantage of the connectivity of individual contours. If we start at a cell known to intersect the isosurface (a *seed cell*), it is possible to "follow" the contour out the faces of the cell to adjacent cells, and repeat until a complete contour has been traced. The task remaining is then to specify sufficient seed cells to guarantee that any contour at any isovalue intersects at least one of the seed cells. This can be done interactively [14], heuristically (extrema graphs [15,16]), by a mark-and-sweep algorithm [2], or using contour trees [3,28].

3. Contour trees

The contour tree was introduced by Boyell and Ruston [5], as a summary of the evolution of contours on a map (i.e. in 2-D), and used by Freeman and Morse to find terrain profiles in a contour map [11]. It has been used for image processing and geographic information systems [12,17,24,25], but principally in 2-D applications. It appears that van Kreveld et al. were the first to identify its applicability to isosurfaces as well as isolines [28]. Since the contour tree is related to the field of Morse theory, we will make some remarks about Morse theory in Section 3.1, then give a definition of the contour tree in Section 3.2, a definition of a related structure called the augmented contour tree in Section 3.3, and a summary of previous work in Section 3.4.

3.1. Morse theory

The field of Morse theory [4,20,23] studies the changes in topology of level sets as the height h is varied. Points at which the topology of the level sets change are called *critical points.* Morse theory requires that the critical points are isolated—i.e., that they occur at distinct points and values. A function that satisfies this condition is called a *Morse function.* All points other than critical points are called *regular points* and do not affect the number or genus of the contours.

In order to take advantage of this, we choose our mesh M to be a simplicial mesh, and our function f to be a piecewise-linear function such that:

(1) f is a linear function within each simplex, and
(2) $f(p_i) = h_i$ for all $i = 1, \ldots, n$.

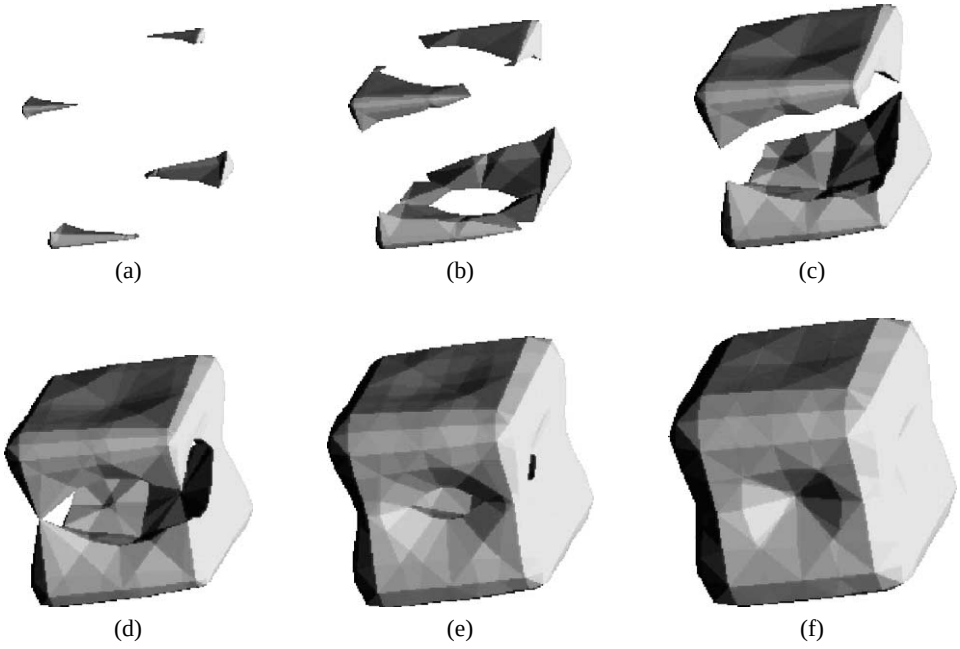

Fig. 1. Level sets of f as $f(x)$ decreases.

This definition of f, as a linear interpolant over a simplicial mesh with unique data values at vertices, ensures that f is a Morse function, and that the critical points occur at vertices of the mesh [4]. This makes it possible to deal with the continuously-defined function f using a combinatorial approach. Note that for generating isosurfaces, we are interested in a subset of the Morse critical points: we do not care about changes of topological genus (e.g., from a disk to a torus), since these changes do not affect the number of contours, or the number of seed cells required.

In Fig. 1, we show a set of isosurfaces from a small dataset, with large values at 4 corners of a cube, medium values at the other 4 corners and on the faces of the cube, and small values inside and outside the cube. As the height (i.e., the value of the function) decreases, we see contours appear, split, change genus, join, and disappear. In particular, the level set evolves from four sticks (a), to two rings (between (b) and (c)), to two cushions (c), to one surface (d), which gradually turns into two nested surfaces as the "inside" and "outside" separate (between (e) and (f)). Finally (although we cannot see this), the inner surface collapses to a point, leaving us with a single surface once more.

3.2. The contour tree

The *contour tree* is a graph that tracks contours of the level set as they split, join, appear, and disappear. Fig. 2 shows the contour tree for the small example illustrated in Fig. 1. Starting at the global maximum, four small contours appear in sequence $(10, 9, 8, 7)$: these correspond to the four leaves at the top of the contour tree. The surfaces join $(6, 5)$ in pairs, forming larger contours, which quickly become rings. These rings then flatten out into cushions, which join (4) to form a single contour. This contour gradually wraps around a hollow core, and pinches off at (3), splitting into two contours: one faces inwards, the

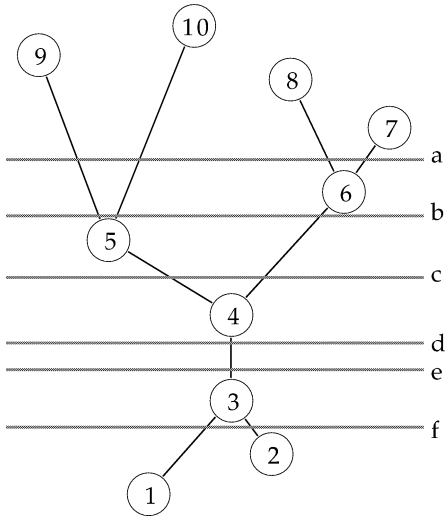

Fig. 2. Contour tree for Fig. 1.

other outwards. The inward contour contracts until it disappears at (2): the outward contour expands until it reaches the global minimum (1).

In the foregoing description, we refer to the evolution of level sets as we vary the height. We make this "evolution" more precise by defining an equivalence relation between two contours. We define a *join* to be a critical point x with an ε-neighbourhood that intersects at least 2 contours at $f(x) + \delta$, where δ, ε are suitably small values. A *split* is then a critical point x with an ε-neighbourhood that intersects at least 2 contours at $f(x) - \delta$. Note that a *local maximum* x must have an ε-neighbourhood that does not intersect any contours at $f(x) + \delta$. Similarly, a *local minimum* x must have an ε-neighbourhood that does not intersect any contours at $f(x) - \delta$. Collectively, we refer to joins, splits, local maxima and local minima as *critical points*: these critical points are a subset of the critical points in Morse theory Section 3.1. We then define the equivalence relation as follows:

Definition 3.1. Let γ and γ' be contours at heights h and h', respectively, with $h < h'$. Then γ and γ' are equivalent ($\gamma \equiv \gamma'$) if all of the following are true:

(1) neither γ nor γ' passes through a join, split, local maximum or local minimum,
(2) γ and γ' are in the same connected component Γ of $\{x\colon\ f(x) \geqslant h\}$, and there is no join $x_i \in \Gamma$ such that $h < h_i < h'$, and
(3) γ and γ' are in the same connected component Δ of $\{x\colon\ f(x) \leqslant h'\}$, and there is no split $x_i \in \Delta$ such that $h < h_i < h'$.

We refer to the equivalence classes of this relation as *contour classes.* Contours that do not pass through critical points belong to contour classes that map 1–1 with open intervals (h_i, h_j), where x_i and x_j are critical points and $x_i < x_j$. We describe a contour class as being *created* at j, at h_j, or at x_j, and being *destroyed* at i, at h_i, or at x_i, thus preserving the intuitive description of a sweep from high to low values. Contours that do pass through critical points must be the sole members of the contour classes

to which they belong (i.e., finite contour classes). This correspondence between critical points and finite contour classes, and between open intervals and infinite contour classes, leads to the definition of the contour tree for a simplicial mesh:

We define the *contour tree*, illustrated in Fig. 2, as a graph (V, E). Following van Kreveld et al. [28] we refer to V and E as *supernodes* and *superarcs*, respectively.

The set V contains a supernode for each finite contour class (i.e., for each critical point.) We distinguish two types of supernodes: An *interior supernode* corresponds to a critical point at which at least one infinite contour class is created, and at least one infinite contour class is destroyed. A *leaf supernode* corresponds to a local maximum, at which an infinite contour class is created, or a local minimum, at which an infinite contour class is destroyed.

The set E contains a superarc for each infinite contour class. Specifically, if and only if an infinite contour class is created at the critical point corresponding to the supernode u and destroyed at the critical point corresponding to v, then the superarc $(u, v) \in E$.

3.3. The augmented contour tree

For some purposes, such as the generation of isosurfaces, information about vertices that are not critical points is also required. We augment the contour tree with the remaining points to produce an *augmented contour tree.* For each vertex x_i in the mesh, take the contour γ_i to which x_i belongs, and insert x_i into the superarc representing the contour class $[\gamma_i]$. Again following van Kreveld et al., we refer to the resulting vertices and edges of the graph as *arcs* and *nodes.* Note that this replaces the superarcs between supernodes with a path consisting of arcs and nodes.

Because of difficulties illustrating the behaviour of level sets in 3-D, we have constructed a small 2-D mesh (Fig. 3), with the same contour tree (Fig. 2) as our original 3-D mesh (Fig. 1). We will continue using this mesh as an example for the balance of this paper: since the algorithm works in arbitrary dimensions, nothing is lost by this choice. Note that, in this 2-D example, the vertices with non-integer labels are not critical points: Fig. 4(a) shows the augmented contour tree for this mesh. Clearly, if we know the nodes and arcs, we can generate the supernodes and superarcs. We simplify the presentation of

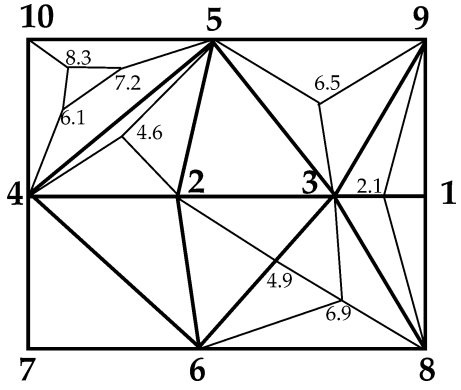

Fig. 3. A small 2-D example, with the same contour tree as Fig. 1.

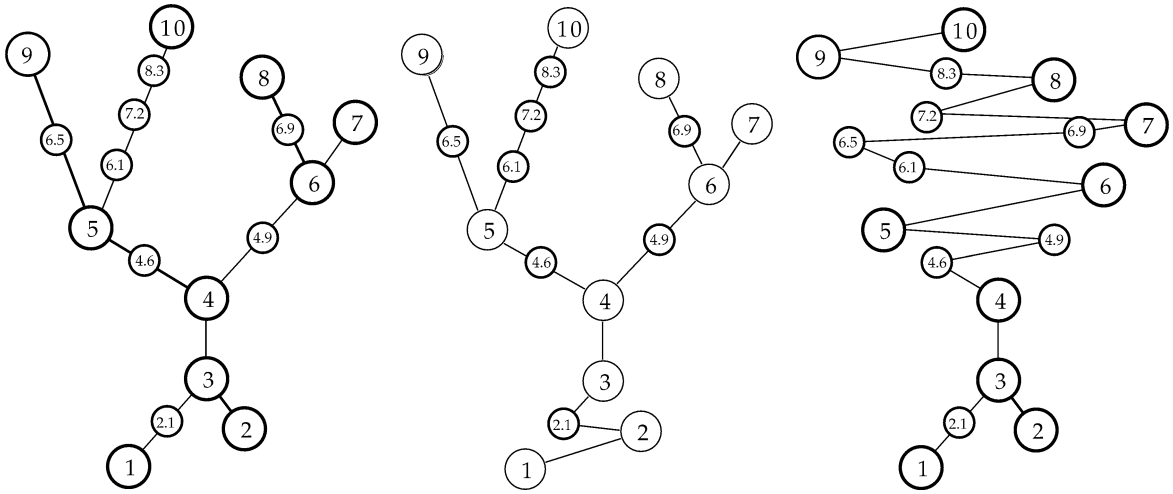

Fig. 4. A small 2-D example, continued. (a) Augmented contour tree. (b) Join tree. (c) Split tree.

the algorithm by working only with the nodes and arcs, and use "contour tree" to refer to the augmented contour tree for the balance of this paper (see Fig. 4(a) for an example).

3.4. Previous work

Van Kreveld et al. [28] reported the first efficient algorithm for constructing contour trees. This algorithm performs the extraction in $O(N \log N)$ time in 2-D data fields, and $O(N^2)$ time in higher dimensions, where N is the number of simplices (triangles) in the mesh of the n data points. The algorithm performs a sweep from low to high value, maintaining each contour, and examines the data set locally to determine when saddle points are encountered and how to deal with them. Multi-saddle points are treated as a set of ordinary saddle points. The most time-consuming step is merging contours. In the plane, the running time is reduced to $O(N \log N)$ by always merging a smaller contour into a larger; a coordinated search in both contours is used to determine which is the smaller.

Tarasov and Vyalyi [26] presented an $O(N \log N)$ algorithm for 3-D data fields. Their algorithm performs three sweeps: one sweep to identify joins, a second to identify splits, and a third to combine the results of the two sweeps. Again, the level set is maintained at all times during the sweep. Multi-saddle points are dealt with by a complicated preprocessing step (see Section 5). Running time is kept to $O(N \log N)$ by a variation of the method used by van Kreveld et al. in the plane. Finally, boundary effects at the edge of the dataset are handled by special cases inside the algorithm.

In both algorithms, two factors contribute to the runtime: the initial sort takes $O(n \log n)$ time, and maintaining the level sets takes $O(N \log N)$ time. Bounds on the number of simplices, N, are $N = \Omega(n)$ and $N = O(n^{\lceil d/2 \rceil})$, for a mesh with n vertices in d dimensions. In dimensions greater than 2, the difference between N and n can become significant for irregular meshes. It is, however, always possible to construct a mesh in any fixed dimension such that $N = \Theta(n)$ (for example, a regular grid). As a result, the difference between n and N is, in most instances, a small constant factor.

4. The contour tree algorithm

We propose an improved algorithm for constructing the contour tree for a real-valued field F interpolated over a simplicial mesh of n vertices and N simplices, with the following characteristics:

(1) time requirements of $O(n \log n + N\alpha(N))$ for constructing augmented contour trees, in any number of dimensions,
(2) space requirements of $O(N)$ for the mesh and $O(n)$ additional working storage,
(3) simple treatment of boundary effects, and
(4) simple treatment of multi-saddle points.

The algorithm has two stages: in the first stage, we build a *join tree* and a *split tree* to identify contour joins and splits (Section 4.1). In the second stage, we merge these two trees to obtain the contour tree (Section 4.2).

Although the algorithm applies to any arbitrary dimension, the illustrations are in two dimensions for clarity (see Fig. 3 for our example).

4.1. Join and split trees

In this subsection, we introduce the *join tree* and *split tree* for a height graph G (a graph with associated heights). We demonstrate that the join tree J_M of the simplicial mesh M used to define our height field F is identical to the join tree J_C of the contour tree C of F. We then present an algorithm for constructing $J_C (= J_M)$ in $O(n \log n + N\alpha(N))$ time and $O(n)$ space.

The *join tree* is a graph that encapsulates all joins in the contour tree; the corresponding *split tree* encapsulates all splits. These trees are dual if we negate all heights, so we will examine only the join tree in detail.

We define a *height graph* to be any graph G with heights $\{h_i\}$ associated with the vertices $\{x_i\}$. For example, the mesh M underlying our function f is a height graph. Throughout the rest of Section 4, we use the notation G_i^+ to refer to the subgraph of G induced by the vertices with height $> h_i$. Also, although the join tree is notionally on the same set of vertices as G, we adopt the convention that if x_i is a vertex in G, then y_i is the corresponding vertex in the join tree.

Definition 4.1. The join tree J_G of a height graph G is the graph on the vertices $y_1, \ldots, y_{\|G\|}$ in which two vertices y_i and y_j, with $h_i < h_j$, are connected when:

(1) x_j is the smallest-valued vertex of some connected component Γ of G_i^+, and
(2) x_i is adjacent in G to a vertex of Γ.

In Fig. 4(b) and (c), we give the join and split trees corresponding to the 2-D example mesh in Fig. 3.

In order for the join tree to be useful, we must relate it to the height field that we are studying: we do so by showing that $J_C = J_M$, i.e., that the contour tree C and the mesh M have the same join tree. We need a couple of preliminary lemmas to show that the connected components of C_i^+ and M_i^+ are identical.

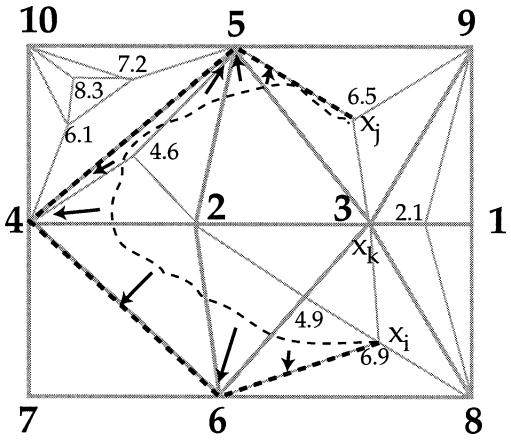

Fig. 5. Constructing a graph path from a path in space.

Lemma 4.2. *x_i and x_j belong to the same component of M_k^+ precisely when they belong to the same component of $\{x\colon\ f(x) > h_k\}$.*

Proof. Suppose that x_i and x_j belong to the same component of M_k^+ for any k. Therefore, there must be a path in M connecting x_i and x_j such that each vertex on the path has height $> h_k$. But since M is embedded in the volume over which f is defined, this path also connects x_i and x_j in the set $\{x\colon\ f(x) > h_k\}$, so x_i and x_j belong to the same component of $\{x\colon\ f(x) > h_k\}$. Now suppose that x_i and x_j belong to the same component of M_k^+. Then x_i and x_j are connected in $\{x\colon\ f(x) > h_k\}$ by some path P. If we trace the path P through the simplices of the mesh M (as in Fig. 5), we can "push" P up to edges of the simplex that are above the value h_k (see Fig. 5). This gives us a path P' connecting x_i and x_j in the mesh M. It follows that x_i and x_j belong to the same component of M_k^+ exactly when they (x_i and x_j) belong to the same component of $\{x\colon\ f(x) > h_k\}$. □

Lemma 4.3. *For each component in C_k^+, there exists a component in M_k^+ containing exactly the same vertices* (*and vice versa*).

Proof. Proof is by finite induction, starting with the highest vertex x_n, for which the property is trivially true. For convenience, assume that the vertices are indexed in sorted order: i.e., that $h_1 < h_2 < \cdots < h_n$.

Assuming that the hypothesis is true for $k \leqslant i \leqslant n$, consider the vertex x_{k-1}: the only difference between the components of M_k^+ and M_{k-1}^+ is that the arcs from x_k to adjacent, higher vertices have been added to the latter. We break the proof into three cases, based on the type of vertex that x_k is: local maximum, join, or neither:

If x_k is a local maximum, then it has no arcs leading upwards, and there are no edges added to M_k^+ to obtain M_{k-1}^+. A local maximum only has one edge, to a lower vertex. Thus no edges are added to C_k^+ to obtain C_{k-1}^+, and the hypothesis follows.

If x_k is a join, then let $x_k x_j$ be any edge incident to x_k in C_{k-1}^+. From Section 3.3, x_k and x_j both either belong to some superarc, or are endpoints of it. Since the superarcs and supernodes correspond to contour classes, we take the union of these contour classes, and obtain a connected set in the original space of points with values between h_k and h_j. Therefore, there is a path P from x_k to x_j in this set.

But this set is contained in some component γ of $\{x\colon f(x) > h_{k-1}\}$. So, by Lemma 4.2, x_k and x_j must also be connected in M^+_{k-1}. This is true for each edge $x_k x_j$ in C (with $h_k < h_j$). Also, the components of M^+_k and $\{x\colon f(x) > h_k\}$ have the same vertex sets by the induction hypothesis. Thus, it follows that x_k is connected to the same components of M^+_k in M as in C.

As a result, the component of M^+_{k-1} to which x_k belongs will correspond directly to the component of $\{x\colon f(x) > h_{k-1}\}$ to which x_k belongs. Components to which x_k does not connect will be unaffected, so we conclude that the components of M^+_{k-1} and C^+_{k-1} contain the same vertices, as required.

If x_k is neither a local maximum nor a join, then it must be adjacent to exactly one component of M^+_k and an argument similar to that of Case II applies to show that the components of M^+_{k-1} and C^+_{k-1} contain the same vertices. □

Theorem 4.4. *The contour tree C and the mesh M have the same join tree* (*i.e.*, $J_C = J_M$).

Proof. In Definition 4.1, I defined the join tree of a height graph G in terms of the components of G^+_i. By Lemma 4.3, these components are identical in C and M, and we saw in the proof of Lemma 4.3 that x_i will be connected to the same components of C^+_i and M^+_i. It follows immediately from Definition 4.1 that $J_C = J_M$. □

Having now defined the join tree, we now present an algorithm to compute it efficiently.

Algorithm 4.1 (*Algorithm To Construct J_M*). Given the mesh M, we compute the join tree J_M of the mesh (see Fig. 6) as follows. Since Definition 4.1 requires that we know the components of M^+_i to determine edges incident to x_i, we use Tarjan's union-find algorithm [27] to determine connectivity. This information is stored in array called *Component*. In addition, Definition 4.1 requires that we know the smallest-valued vertex in each component, so we maintain a separate array, *LowestVertex*, for this information.

We sort the vertices of the mesh by the corresponding height values, then process the vertices from highest to lowest value. For each vertex x_i, we add each edge $x_i x_j$ to a union-find structure iff $h_i < h_j$. After each vertex x_i has been processed, all edges between two vertices whose values are at least h_i must be contained in the union-find structure. At each vertex x_i, we generate one edge in the join tree for each

```
Algorithm to compute J_C = J_M:
Input: the mesh M, with vertices x_1...x_n in sorted order (i.e., h_1 < h_2...h_n)
Output: the join tree J_C, with vertices y_1...y_n
 1. for i := n downto 1 do:
    (a) Component[i] := i
    (b) LowestVertex[i] := y_i
    (c) for each vertex x_j adjacent to x_i
        i.   if (j < i) or (Component[i] = Component[j]) skip x_j
        ii.  UFMerge(Component [i], Component[j])
        iii. AddEdgeToJoinTree(y_i, LowestVertex[Component[j]])
        iv.  LowestVertex[Component[j]] := y_i
```

Fig. 6. Algorithm 4.1 to construct a join tree.

component of M_i^+ to which x_i connects, and update both the union-find structure and the smallest vertex of each component. After we have processed all vertices, the join tree has been computed.

To see that we construct the join tree with this procedure, suppose that we are processing vertex x_i, and that $x_i x_k$ is an edge from x_i to a vertex x_k whose height h_k is higher than h_i. Because Tarjan's union-find algorithm computes connectivity incrementally, *Component* represents the connectivity of all edges added so far. Since each edge in M_i^+ has both ends higher than x_i, they must already have been added, and *Component* therefore represents the connectivity of M_i^+ at this stage. Now, since $h_i < h_k$, x_k has already been processed, and must belong to some component Γ of M_i^+. Since this satisfies the second condition of Definition 4.1, we use *LowestVertex* to identify the smallest-valued vertex x_j of Γ, and add edge $x_i x_j$ to the output if it has not already been added. After processing x_i, we set *LowestVertex*[*Component*[i]] to point to x_i, as it is now the lowest-valued vertex in the component to which it belongs.

This algorithm requires a sort in $O(n \log n)$ time, followed by the union-find algorithm in $O(N + M\alpha(M))$, where N is the number of edges in the mesh, and M is the number of union-find merges performed (at most equal to the number of local maxima in the mesh). Note that $M + m \leqslant t \leqslant 2(M + m) - 1$, where t is the number of supernodes in the contour tree, and M, m are the number of local maxima and minima, respectively. Thus, we can express the bound for constructing the join and split trees as $O(n \log n + N + t\alpha(t))$.

4.2. Merging to form the contour tree

In this section, we give the main contribution of this paper: a simple algorithm to merge join and split trees. First we give an overview of the concept behind the merge algorithm, define some terms, and provide a recursive proof that the merge algorithm works. We then give a non-recursive implementation of the algorithm that takes $O(n)$ time.

To reconstruct the contour tree C from the join tree J_C and split tree S_C, we identify a leaf x_i of C and its incident edge $x_i x_j$. We delete x_i from C, J_C and S_C to produce a reduced graph $C \setminus x_i$, along with the corresponding join tree $J_{C \setminus x_i}$ and $S_{C \setminus x_i}$. We repeat the process until all edges of C have been identified.

Before embarking on the reconstruction, we define some terms that we rely on. We use *up-arc* and *down-arc* to refer to arcs leading up and down from a given vertex in a given graph, and *up-degree*(δ^+) and *down-degree*(δ^-) to refer to the number of up- and down- arcs at a given vertex. Note that the up-degree of a vertex y_i in J_C is identical to the up-degree of the corresponding vertex x_i in C, and that the down-degree of a vertex y_i in J_C is always 1, except at the global minimum vertex, where it is 0. Similarly, a vertex z_i in the split tree has identical down-degree to the corresponding vertex x_i in C, and the up-degree of a vertex z_i in the split tree is 1 except at the global maximum. Since we can find the up-degree of x_i (in C) by examining y_i (in J_C) and the down-degree of x_i (in C) by examining z_i in S_C, we note that we can tell the exact degree of any vertex x_i in C, even if we do not know the edges in C. In particular, we can use this information to identify which vertices are leaves of C. For convenience, we will refer to leaves of C with up-degree of 0 as *upper leaves* and those with down-degree of 0 as *lower leaves.*

When deleting a vertex x_i from C, we preserve connectivity by contracting the incident arcs into a single arc (see Fig. 7). This operation is called *reduction* to distinguish it from the simple removal of a vertex from a graph. Theorem 4.8 will then show that applying the reduction operation on the join and split trees gives the join and split trees of the new, smaller graph.

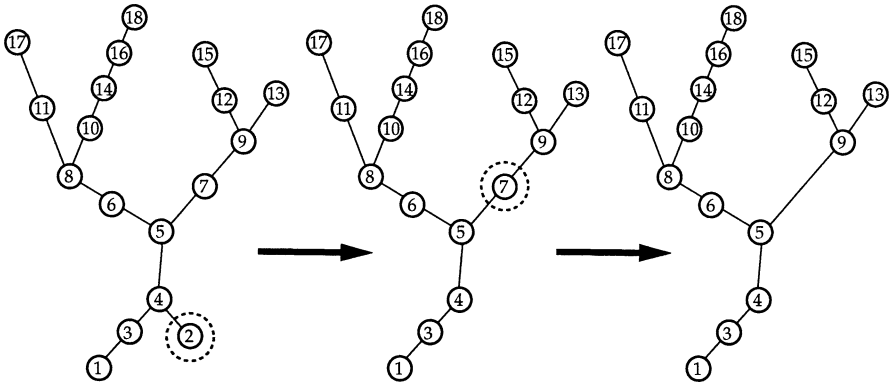

Fig. 7. Vertex reductions applied to vertices 2 and 7.

Definition 4.5. Define $C \ominus x_i$, the reduction of a graph C by a vertex x_i whose up-degree and down-degree are both $\leqslant 1$, to be:

(1) If x_i has arcs $x_i x_j$ up and $x_i x_k$ down in C, then: $C \ominus x_i = C \setminus x_i \cup x_j x_k$.
(2) Otherwise, $C \ominus x_i = C \setminus x_i$.

Lemma 4.6. *If x_i is an upper leaf in C, and $y_i y_j$ is the incident arc to y_i in J_C, then $x_i x_j$ is the incident arc to x_i in C.*

Proof. Let x_i belong to some component γ in C_j^+. Suppose that x_i is not the only vertex in γ. Then, since γ is a connected component, there is some other vertex x_k in γ to which x_i is connected. By Definition 4.1, x_i is the smallest-valued vertex in γ, so $x_i x_k$ must be an up-arc at x_i. But, since x_i is an upper leaf, it has no up-arcs. It follows that x_i is the only vertex in γ. Applying Definition 4.1, $y_i y_j$ is an arc of J_C, then x_j must be connected to some vertex in γ. But, since x_i is the only vertex in γ, it follows that x_j is connected to x_i. □

We now consider what happens when we remove an edge $x_i x_j$ from C. Recall that our convention (from Definition 4.1) is that x_i refers to a vertex in C, and y_i the same vertex in J_C.

Lemma 4.7. *If x_i is a leaf of C, and $y_j y_k$ is an arc of the corresponding join tree J_C such that $h_j < h_k$, and $i \neq j, k$, then $y_j y_k$ is also an arc of $J_{C \setminus x_i}$.*

Proof. By Definition 4.1, x_j is adjacent to some vertex x_l in the component γ of C_j^+ to which x_k belongs: i.e., there exists some path P from x_l to x_k in γ. Since x_i is a leaf, it could only be at an end of the path, but x_j, x_k are the path-ends, and $x_i \neq x_j, x_k$. Thus, P exists in $C \setminus x_i$, and therefore in $C \setminus x_{i_j}^+$, the subgraph of $C \setminus x_i$ consisting of edges whose vertices have higher values than x_i does. Since $x_j x_l$ is also in $C \setminus x_i$, x_j is adjacent to the component ρ of $C \setminus x_{i_j}^+$ to which x_k belongs.

Note that each path P connecting two vertices of γ is also in ρ, except for paths starting or ending at x_i: thus the vertices of γ are the same as those of ρ, with the possible exception of x_i. It then follows that x_k is the smallest-valued vertex of ρ, so by Definition 4.1, y_j is adjacent to y_k in $J_{C \setminus x_i}$. □

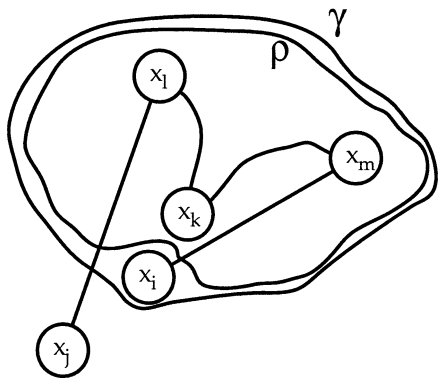

Fig. 8. Reducing a join tree at a lower leaf.

Theorem 4.8. *If x_i is a leaf of a contour tree C, then $J_{C\setminus x_i} = J_C \ominus y_i$.*

Proof. From Lemma 4.7, each edge of J_C that is not incident to y_i is also in $J_{C\setminus x_i}$. We know that both J_C and $J_{C\setminus x_i}$ are trees, with $n-1$ and $n-2$ edges, respectively.

Suppose that y_i is a leaf in J_C. Then there are $n-2$ edges of J_C that are not incident to y_i, and by Lemma 4.7, each of them must be in $J_{C\setminus x_i}$, so $J_{C\setminus x_i} = J_C \ominus y_i$.

Since y_i is not a leaf in J_C, $\delta^+(y_i) = 1$; since y_i is not the global minimum, $\delta^-(y_i) = 1$. After excluding these two edges, only $n-3$ edges of J_C remain that are not incident to y_i. Again, by Lemma 4.7, each of them must be in $J_{C\setminus x_i}$, so only one edge remains to be found.

Let the down-arc at y_i be $y_i y_j$, and the up-arc be $y_i y_k$ (see Fig. 8). From Definition 4.1, x_i belongs to some component γ of C_j^+, and x_j is adjacent to some vertex x_l in γ. Note that $x_l x_j$ must be a down-arc, and since x_i has no down-arcs, x_l cannot be x_i. Also, since x_i is the smallest-valued vertex in γ, $h_i < h_l$.

Consider the component ρ of C_i^+ to which x_k belongs. Since each vertex of ρ has value $\geqslant h_k$, and $h_k > h_j$, we know that $\rho \subseteq \gamma$ and x_k must be in γ. Since x_i is the smallest-valued vertex of γ, the only arc of γ that is not in ρ must be the arc incident to x_i, so $\rho = \gamma \setminus x_i$. But this must be a component of $(C \setminus x_i)_i^+$, the subgraph of $C \setminus x_i$ containing only vertices with heights $> h_i$. Since $x_l \neq x_i$, it follows that x_l must have been connected to x_i in γ, as was x_k. Then x_l must be connected to x_k by a path whose vertices all have values $> h_i$. Therefore, x_l and x_k belong to the same component of ρ, and since x_k is the smallest-valued vertex of $\delta \setminus x_i$, y_j must be connected to y_k in $J_{C\setminus x_i}$.

Note that $y_j y_k$ cannot be an arc in J_C, because $y_i y_j y_k$ would then be a cycle in J_C. Thus, we have added an arc to the $n-3$ arcs that we had already shown to be in $J_C \ominus y_i$, for a total of $n-2$. Since $J_{C\setminus x_i}$ is a tree on $n-1$ vertices, there are no more arcs to be found in $J_{C\setminus x_i}$. From Definition 4.5, it follows that $J_{C\setminus x_i} = J_C \ominus y_i$. □

We can implement this algorithm to run in time that is linear in the size of the tree. In fact, by eliminating the tail-recursion and using static data structures for C, J_C, and S_C, this step changes from being the slowest of the three sweeps in Tarasov and Vyalyi [26] to being the fastest.

Algorithm 4.2 (*Algorithm To Merge J_C and S_C*). In the merge algorithm (Fig. 9), we assume that the join tree J_C and split tree S_C are stored as adjacency lists using half-arcs: that is, each arc $y_i y_j$ in J_C is stored as a directed arc α in y_i's adjacency list, linked to a directed arc α' in y_j's adjacency list.

```
Algorithm to compute the contour tree:
Input: the join tree J_C and split tree S_C corresponding to C,
  stored as adjacency lists
Output: the contour tree C
  1. For each vertex x_i, if up-degree in J_C+ down-degree in S_C
     is 1, enqueue x_i
  2. Initialize C to an empty graph on ||J_C|| vertices
  3. While leaf queue size > 1
     (a) Dequeue the first vertex, x_i, on the leaf queue.
     (b) If x_i is an upper leaf, find incident arc y_i y_j in J_C.
         Else find incident arc z_i z_j in S_C.
     (c) Add x_i x_j to C.
     (d) J_C ← J_C ⊖ y_i,  S_C ← S_C ⊖ z_i.
     (e) If x_j is now a leaf, enqueue x_j.
```

Fig. 9. Algorithm 4.2 to merge the join and split trees.

Note that, since there are $n-1$ edges in the contour tree, the main loop of the algorithm iterates $n-1$ times, leaving one vertex on the queue at the end. The first and last 4 steps of this algorithm on the example in Fig. 4 are shown in Fig. 10.

As we observed in Section 3.3, this algorithm in fact computes the augmented contour tree, but we can convert this to the contour tree proper in $O(n)$ time by applying the reduction operation to each regular vertex: these can readily be identified in the contour tree, since they are the only vertices to have one arc leading upwards and one downwards. Alternately, it is not difficult to modify Algorithm 4.1 so that instead of storing the lowest vertex, we store the vertex at which the last join or maximum occurred. Edges are only added to the join tree when another join is encountered, or at the global minimum. After a separate pass to determine the split tree, all supernodes will be present in at least one of the two trees. All supernodes that are only present in the join tree are added to the split tree, along the appropriate arc. Although this reduces the cost of merging to $O(t)$ from $O(n)$, the asymptotic running time of the algorithm is not improved.

4.3. Boundary effects and multiple singularities

Although we previously reported [6] that special treatment was required for vertices on the boundary of the data set, it turns out that the algorithm given above needs no special cases for boundary vertices. In addition, no special cases are required for dealing with multiple saddle points, although we extend Tarasov and Vyalyi's result [26] to arbitrary dimensions in Section 5, below.

4.4. Computing the (non-augmented) contour tree

As noted in Section 3.3, the presentation of the algorithm is simpler if we work with the augmented contour tree instead of the contour tree, then reduce all regular points in the augmented contour tree to obtain the contour tree. In practice, a slightly more efficient implementation is possible. To compute the contour tree with Algorithm 4.2, we need to compute the join and split trees for the contour tree, rather

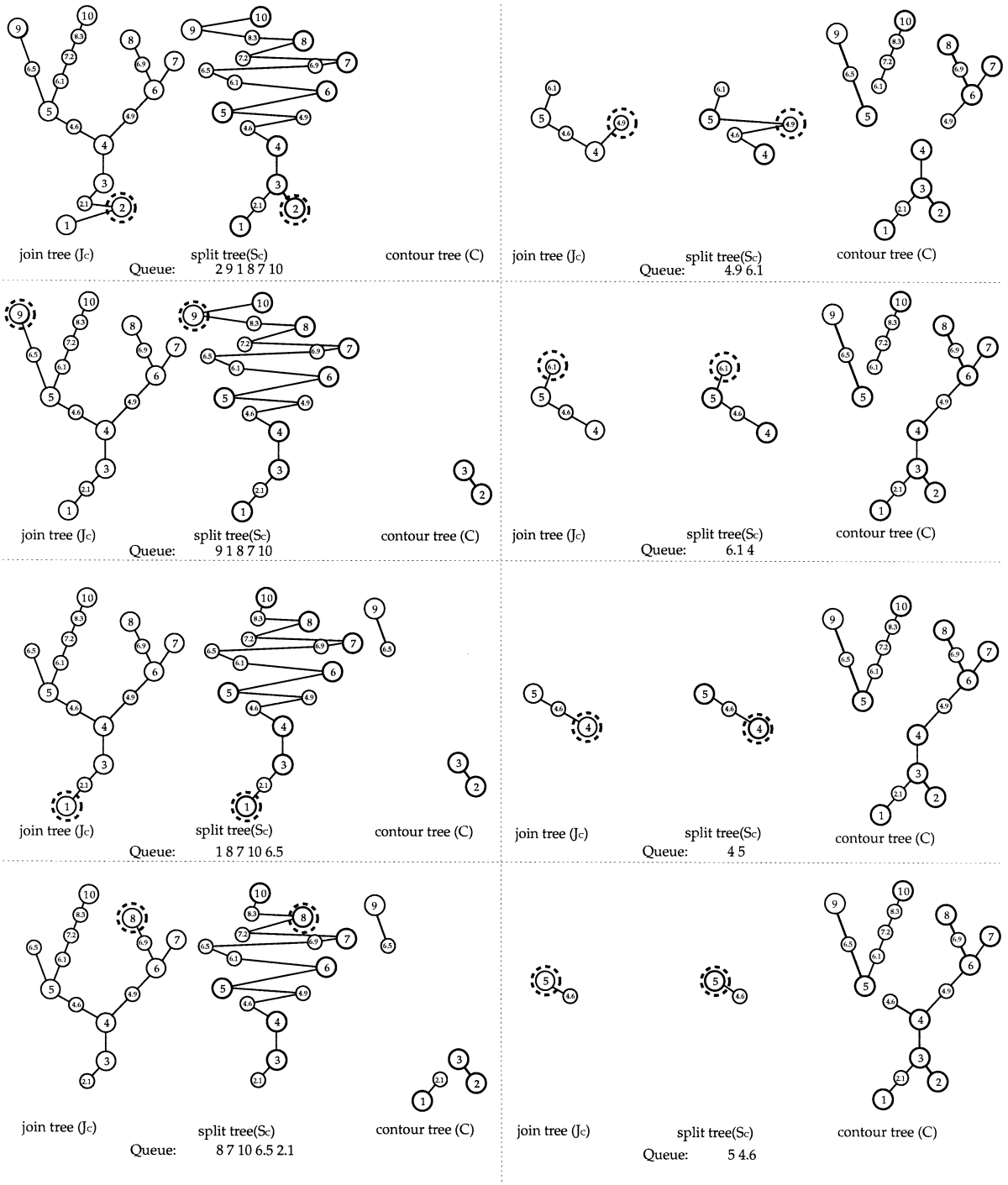

Fig. 10. First four and last four steps of merge algorithm.

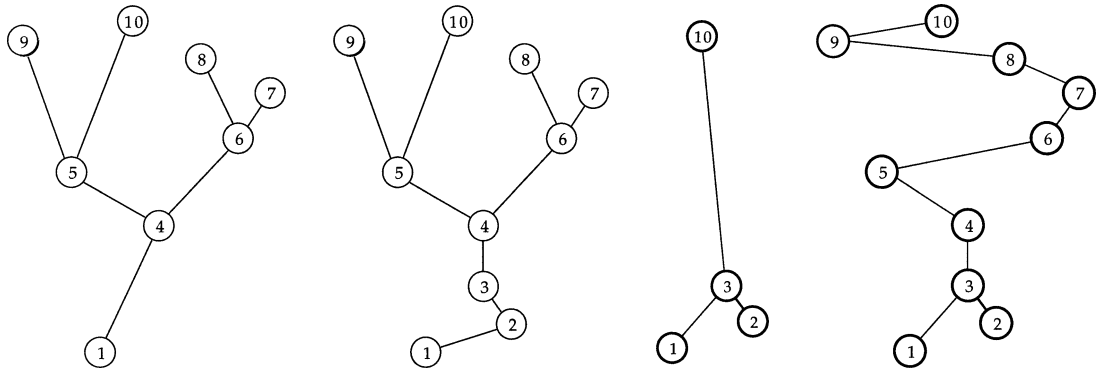

Fig. 11. Computing the (non-augmented) contour tree. (a) Join tree. (b) Full join tree. (c) Split tree. (d) Full split tree.

than for the augmented contour tree. This can be done by omitting regular points during the construction of the join and split trees in Algorithm 4.1. Instead of adding an edge to the join tree at every vertex, we do so only at joins and at the global minimum: the upper end of the edge will be the vertex at which the component in the union-find data structure was last changed (i.e., created or merged). This will give us a join tree with local maxima, joins and the global minimum only. We augment this join tree with the splits and local minima, as shown in Fig. 11, then apply Algorithm 4.2 to compute the contour tree.

5. Resolving multiple singularities

The algorithm described by Tarasov and Vyalyi [26] requires *simple singularities*, so they describe a method for breaking multi-saddle points into multiple simple singularities in time $O(N \lg N)$. Although our algorithm handles multi-saddles, their method is of independent interest for computation of Morse singularities in higher dimensions; if non-simple singularities are resolved, then a general function on a complex K is a Morse function. We therefore briefly show that their method applies in all dimensions. We assume familiarity with concepts of PL topology such as barycentric subdivisions, star, and link [22].

We first summarize the subdivision and perturbation given in [26] and extend it trivially to general dimensions. We then considerably simplify the proof that this method resolves non-simple singularities, and we extend it to all dimensions. Assume that K is a m-dimensional simplicial complex, $m \geqslant 3$, in $\mathbb{R}^d$ and f is a general function on K (i.e., $f(v) \neq f(w)$ for any pair of vertices $v, w \in K$). The first step is to construct the barycentric subdivision, $\operatorname{sd} K$, and extend f linearly over $\operatorname{sd} K$. This yields a new function f_0 with the property that no two critical points are adjacent, but which may not be a general function. A small perturbation described in [26] transforms f_0 into a general function f_1 over $K_1 = \operatorname{sd} K$.

Now the star of each non-simple singularity is further refined. Let v be a non-simple saddle point. For each k-dimensional simplex in the link of v, $\operatorname{Lk}(v)$, a new so-called k-vertex is added in the star of v, $\operatorname{St}(v)$, as follows. For each vertex w in $\operatorname{Lk}(v)$, a corresponding 0-vertex is added on the edge vw, at a point which is $\frac{1}{4}$ distance from v to w. For each k-simplex σ in $\operatorname{Lk}(v)$, $k \geqslant 1$, a k-vertex is added in the $(k+1)$-simplex formed by v and σ, at $\frac{1}{3}$ distance from v to the barycenter of σ. See Fig. 12 for an illustration in 2 dimensions.

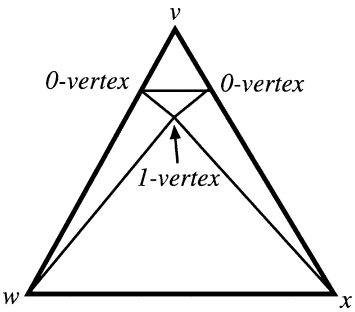

Fig. 12. The subdivision of a 2-simplex vwx at a non-simple singularity v.

Simplices of this subdivision are defined as follows. Let σ be a m-simplex in $\mathrm{St}(v)$, i.e., a simplex of highest dimension; it contains m 0-vertices. These together with v form a new m-simplex. The rest of σ is then a prism with two $(m-1)$-simplices as bases. Now each cell containing a 1-vertex is star triangulated from the 1-vertex, then each 2-vertex defines a star triangulation to form tetrahedra, and so on up to the $(m-1)$-vertex, where the star triangulation results in m-simplices.

The neighborhoods of all non-simple singularities are refined in this manner, yielding a new complex K_2. Now f_1 is extended over K_2 to yield a new function f_2. By definition, $f_1 = f_2$ at all vertices common to K_1 and K_2. We now describe the extension of f_1 to f_2, again very similar to that described in [26].

Let h be a linear function over $\mathbb{R}^d$ that has different values at all vertices of K_2, and let H be the maximum difference between any two values of h on K_2, i.e.,

$$H = \max_{v,w}\bigl\{h(v) - h(w)\bigr\}.$$

Let δ be the minimum gap between successive values of f_1 on K_1. For each vertex u added in the star of a non-simple singularity v, let

$$f_2(u) = f_1(v) + \frac{\delta}{2H}\bigl(h(u) - h(v)\bigr).$$

Function f_2 on K_2 now has the property that all singularities are simple, i.e., that the level set at $f_2(v)$ divides $\mathrm{St}(v)$ into at most three components. Indeed, it is easy to see that all former regular points and simple singularities are still regular or simple (see [26]), so we restrict ourselves here to proving that a former non-simple singularity v is regular, and that all points added in K_2 are either regular or simple. To see that v is a regular point, notice that after the local refinement around v, $\mathrm{St}(v)$ consists only of the simplices formed by 0-vertices and v. f_2 is by construction linear over $\mathrm{St}(v)$ and so v must be a regular point. Now we use an inductive proof to show that the added k-vertices are either regular or simple. We define the *restricted star* or *restricted link* to be the restriction of the star or link of an added point u to simplices formed only by vertices added in K_2.

Lemma 5.1. *All k-vertices, $k \geqslant 0$, added in the subdivision around non-simple singularities are either regular points or simple singularities of f_2.*

Proof. Let u be a 0-vertex, u is adjacent to two original vertices from K_1: the non-simple singularity v, and the vertex w which was used to construct u. Otherwise, u is only adjacent to other added vertices.

Since f_2 is linear over the simplices formed by v and the added vertices, the level set at $f_2(u)$ divides the restricted $\mathrm{St}(u)$ into at most two connected components, one with values greater than $f_2(u)$ and the other with values less than $f_2(u)$. w either belongs to one of those connected components or it forms its own connected component. Thus, u is either a regular point or a simple singularity.

Now let u be a k-vertex, $k \geqslant 1$. By construction, u is not adjacent to any vertices of K_1 other than the vertices of the k-simplex that define u. Again, the restricted $\mathrm{St}(u)$ and $\mathrm{Lk}(u)$ can be broken by the level set at $f_2(u)$ into at most 2 components. We now make the inductive assumption that a $(k-1)$-simplex $\sigma \in \mathrm{Lk}(u)$ from K_1 divides $\mathrm{Lk}(u)$ further into at most three components and show that under this assumption, a k-simplex from K_1 in $\mathrm{Lk}(u)$ cannot divide $\mathrm{Lk}(u)$ further into more than three connected components. Let $\sigma \in \mathrm{Lk}(u)$ be a $(k-1)$-simplex from K_1, and let $w \in \mathrm{Lk}(u)$ be the additional vertex from K_1 that forms a k-simplex in $\mathrm{Lk}(u)$. There are three cases to consider.

(1) Suppose first that some vertices of σ have value in f_2 greater than $f_2(u)$ and others have value less than $f_2(u)$. Then w necessarily belongs to one of the existing connected components.
(2) Suppose σ belongs to one of the connected components of the restricted $\mathrm{Lk}(u)$. Then $\mathrm{Lk}(u)$ without w consists of at most two components, and w can increase this to at most three components.
(3) Finally, assume that σ forms a separate connected component. w is adjacent to both σ and vertices of the restricted $\mathrm{Lk}(u)$, so regardless of the value at $f_2(w)$, vertex w belongs to an existing component.

These three cases complete the proof. □

Note that in the proof we do not need to distinguish between boundary simplices and interior simplices.

6. Conclusions

Tarasov and Vyalyi [26] stated an algorithm for constructing contour trees in three dimensions, based on the work of van Kreveld et al. [28]. We have taken this algorithm, simplified it, and extended it to arbitrary dimensions. We have discarded the explicit construction of contours during the third sweep in their algorithm. In addition, our algorithm needs no special cases or preprocessing to deal with boundary vertices or with multiple singularities. Our algorithm applies to any arbitrary dimensional data with the same asymptotic performance, since it is no longer dependent on explicit construction of level sets during the sweep. For cases where it is desirable to substitute simple singularities for multiple singularities, we have also extended Tarasov and Vyalyi's pre-processing step to arbitrary dimensions. We have also improved the asymptotic time bound from $\mathrm{O}(N \log N)$ to $\mathrm{O}(n \log n + t\alpha(t))$.

7. Future work

We have implemented the algorithm stated for relatively small data sets ($< 10^6$ vertices). Unlike Marching Cubes [19] and its derivatives, we require the input data to be on a simplicial mesh. We intend to modify the algorithm and the contour tree approach to work directly with voxels, and also intend to implement a parallel algorithm for working with large data sets.

Acknowledgements

Portions of this work were performed while the first two authors were affiliated with the University of British Columbia, Vancouver, Canada. Support from Canada's NSERC through a postgraduate fellowship, a research grant, and the IRIS NCE is gratefully acknowledged. Further support came from the University of North Carolina at Chapel Hill.

References

[1] E. Artzy, G. Frieder, G.T. Herman, The theory, design, implementation and evaluation of a three-dimensional surface detection algorithm, Computer Graphics and Image Processing 15 (1981) 1–24.
[2] C.L. Bajaj, V. Pascucci, D.R. Schikore, Fast isocontouring for improved interactivity, in: IEEE Proceedings on Volume Visualization 96, IEEE, 1996, pp. 39–47.
[3] C.L. Bajaj, V. Pascucci, D.R. Schikore, Seed sets and search structures for optimal isocontour extraction, Technical Report 99-35, Austin, TX, 1999.
[4] T.F. Banchoff, Critical points and curvature for embedded polyhedra, J. Differential Geom. 1 (1967) 245–256.
[5] R.L. Boyell, H. Ruston, Hybrid techniques for real-time radar simulation, in: IEEE Proceedings Fall Joint Computer Conference 63, IEEE, 1963, pp. 445–458.
[6] H. Carr, J. Snoeyink, U. Axen, Computing contour trees in all dimensions, in: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM, 2000, pp. 918–926.
[7] Y.-J. Chiang, C.T. Silva, I/O optimal isosurface extraction, in: IEEE Proceedings on Visualization 96, IEEE, 1997, pp. 303–310.
[8] Y.-J. Chiang, C.T. Silva, W.J. Schroeder, Interactive out-of-core isosurface extraction, in: IEEE Proceedings an Visualization 98, IEEE, 1998, pp. 167–174.
[9] P. Cignoni, C. Montani, E. Puppo, R. Scopigno, Multiresolution representation and visualization of volume data, IEEE Trans. on Visualization and Computer Graphics 3 (4) (1997) 352–369.
[10] H. Edelsbrunner, E. Mücke, Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms, ACM Trans. on Graphics 9 (1) (1990) 66–104.
[11] H. Freeman, S. Morse, On searching a contour map for a given terrain elevation profile, J. Franklin Institute 284 (1) (1967) 1–25.
[12] C. Gold, S. Cormack, Spatially ordered networks and topographic reconstruction, in: Proceedings of the 2nd International Symposium on Spatial Data Handling, ACM, 1986, pp. 74–85.
[13] B. Guo, Interval set: A volume rendering technique generalizing isosurface extraction, in: IEEE Proceedings on Visualization 95, IEEE, 1995, pp. 3–10.
[14] C. Howie, E.H. Blake, The mesh propagation algorithm for isosurface construction, Computer Graphics Forum 13 (1994) 65–74.
[15] T. Itoh, K. Koyamada, Automatic isosurface propagation using an extrema graph and sorted boundary cell lists, IEEE Trans. on Visualization and Computer Graphics 1 (4) (1995) 319–327.
[16] T. Itoh, K. Koyamada, Isosurface extraction by using extrema graphs, IEEE Trans. on Visualization and Computer Graphics 1 (1995) 77–83.
[17] I.S. Kweon, T. Kanade, Extracting topographic terrain features from elevation maps, CVGIP: Image Understanding 59 (1994) 171–182.
[18] Y. Livnat, H.-W. Shen, C.R. Johnson, A near optimal isosurface extraction algorithm using the span space, IEEE Trans. on Visualization and Computer Graphics 2 (1) (1996) 73–84.
[19] W.E. Lorenson, H.E. Cline, Marching cubes: A high resolution 3D surface construction algorithm, Computer Graphics 21 (4) (1987) 163–169.
[20] J.W. Milnor, Morse Theory, Princeton University Press, Princeton, NJ, 1963.
[21] G.M. Nielson, B. Hamann, The asymptotic decider: Resolving the ambiguity in marching cubes, in: IEEE Proceedings on Visualization 91, IEEE, 1991, pp. 83–91.

[22] C. Rourke, B. Sanderson, Introduction to Piecewise-Linear Topology, Springer, Berlin, 1972.
[23] Y. Shinagawa, T.L. Kunii, Y.L. Kergosien, Surface coding based on Morse theory, IEEE Computer Graphics Appl. 11 (1991) 66–78.
[24] J.K. Sircar, J.A. Cebrian, Application of image processing techniques to the automated labelling of raster digitized contour maps, in: Proceedings of the 2nd International Symposium on Spatial Data Handling, ACM, 1986, pp. 171–184.
[25] S. Takahashi, T. Ikeda, Y. Shinagawa, T.L. Kunii, M. Ueda, Algorithms for extracting correct critical points and constructing topological graphs from discrete geographical elevation data, Computer Graphics Forum 14 (3) (1995) C-181–C-192.
[26] S.P. Tarasov, M.N. Vyalyi, Construction of contour trees in 3D in $O(n \log n)$ steps, in: Proceedings of the 14th ACM Symposium on Computational Geometry, ACM, 1998, pp. 68–75.
[27] R. Tarjan, Efficiency of a good but not linear set union algorithm, J. ACM 22 (1975) 215–225.
[28] M. van Kreveld, R. van Oostrum, C.L. Bajaj, V. Pascucci, D.R. Schikore, Contour trees and small seed sets for isosurface traversal, in: Proceedings of the 13th ACM Symposium on Computational Geometry, ACM, 1997, pp. 212–220.
[29] J. Wilhelms, A. van Gelder, Topological considerations in isosurface generation, Computer Graphics 24 (5) (1990) 79–86.
[30] J. Wilhelms, A. van Gelder, Octrees for faster isosurface generation, ACM Trans. on Graphics 11 (3) (1992) 201–227.
[31] Y. Zhou, B. Chen, A. Kaufman, Multiresolution tetrahedral framework for visualizing regular volume data, in: IEEE Proceedings on Visualization 97, IEEE, 1997, pp. 135–142.