Contour Trees and Small Seed Sets for Isosurface Traversal*

Chandrajit Bajaj†
bajaj@ticam.utexas.edu

Marc van Kreveld‡
marc@cs.uu.nl

René van Oostrum‡
rene@cs.uu.nl

Valerio Pascucci†
pascucci@ticam.utexas.edu

Daniel R. Schikore§
schikore@llnl.gov

Abstract

For 2D or 3D meshes that represent the domain of continuous function to the reals, the contours—or isosurfaces—of a specified value are an important way to visualize the function. To find such contours, a seed set can be used for the starting points from which the traversal of the contours can begin. This paper gives the first methods to obtain seed sets that are provably small in size. They are based on a variant of the contour tree (or topographic change tree). We give a new, simple algorithm to compute such a tree in regular and irregular meshes that requires $O(n \log n)$ time in 2D for meshes with n elements, and in $O(n^2)$ time in higher dimensions. The additional storage overhead is proportial to the maximum size of any contour (linear in the worst case, but typically less). Given the contour tree, a minimum size seed set can be computed in roughly quadratic time. Since in practice this can be excessive, we develop a simple approximation algorithm giving a seed set of size at most twice the size of the minimum. It requires $O(n \log^2 n)$ time and linear storage once the contour tree is known. We also give experimental results, showing the size of the seed sets for several data sets.

1 Introduction

Scalar data defined over the plane or 3-space is quite common in fields like medical imaging, scientific visualization, and geographic information systems. Such data can be visualized after interpolation by showing one or more contours or isosurfaces: the sets of points having a specified scalar value. For example, scalar data over the plane are used to model elevation in the landscape, and a contour is just an isoline of elevation. In atmospheric pressure modelling, a contour is a surface in the atmosphere where the air pressure is constant, an isobar. In medical imaging, isosurfaces are used to show reconstructed scans of the brain or parts of the body. The scalar data can be seen as a sample of some real-valued function, which is called a terrain or elevation model in GIS, and a scalar field in imaging.

A real-valued function over 2D or 3D can be represented in a computer using a 2D or 3D mesh, which can be regular (all cells have the same size and shape) or irregular. A

*The research of the second and third authors was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL). The research of the first, fourth and fifth authors was partially supported by AFOSR grant F-49620-97-1-0278 and ONR grant N00014-97-1-0370. An extended abstract of this paper appeared at the 13th ACM Symp. on Computational Geometry, 1997.

†Department of Computer Science, University of Texas, Austin, TX 78712, U.S.A.

‡Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, The Netherlands.

§Lawrence Livermore National Lab, P.O. Box 808, L-561, Livermore, CA 94550, U.S.A.

terrain (mountain landscape) in GIS is commonly represented by regular square grid or an irregular triangulation. The elements of the grid, or vertices of the triangulation, have a scalar function value associated to them. The function value of non-vertex points in the 2D mesh can be obtained by interpolation. An easy form of interpolation for irregular triangulations is linear interpolation over each triangle. The resulting model is known as the TIN model for terrains (Triangulated Irregular Network) in GIS. In computational geometry, it is known as a polyhedral terrain. More on interpolation of spatial data and references to the literature can be found in the book by Watson [23].

One can expect that the combinatorial complexity of the contours with a single function value in a mesh with n elements is roughly proportional to $\sqrt{n}$ in the 2D case and to $n^{2/3}$ in the 3D case [14]. Therefore, it is worthwhile to have a search structure to find the mesh elements through which the contours pass. This will be more efficient than retrieving the contours of a single function value by inspecting all mesh elements.

There are basically two approaches to find the contours more efficiently. Firstly, one could store the 2D or 3D domain of the mesh in a hierarchical structure and associate the minimum and maximum occurring scalar values at the subdomains to prune the search. For example, octrees have been used this way for regular 3D meshes [24].

The second approach is to store the *scalar range*, also called *span*, of each of the mesh elements in a search structure. Kd-trees [14], segment trees [4], and interval trees [5, 22] have been suggested as the search structure, leading to a contour retrieval time of $O(\sqrt{n}+k)$ or $O(\log n + k)$, where n is the number of mesh elements and k is the size of the output. The latter bound is the optimal query time [3]. A problem with this approach is that the search structure can be a serious storage overhead, even though an interval tree needs only linear storage. One doesn't want to store a tree with a few hundred million intervals that would arise from regular 3D meshes. It is possible to reduce the storage requirements of the search structures by observing that a whole contour can be traced directly in the mesh if one mesh element through which the contour passes is known. Such a starting element of the mesh is also called a *seed*. Instead of storing the scalar range of all mesh elements, we need only store the scalar range of the seeds as intervals in the tree, and a pointer into the mesh. Of course, the seed set must be such that every possible contour of the function passes through at least one seed. Otherwise, contours could be missed. There are a few papers that take this approach [4, 12, 22]. The tracing algorithms to extract a contour from a given seed have been developed before, and they require time linear in the size of the output [2, 11, 12].

The objective of this paper is to present new methods for seed set computation. Of a seed set, we require that any possible contour in the mesh pass through at least one seed. Otherwise we could miss a contour. To construct such a small size seed set, we use a variation of the *contour tree*, a tree that captures the contour topology of the function represented by the mesh. It has been used before in image processing and GIS research [8, 9, 13, 19, 20]. Another name in use is the *topographic change tree*, and it is related to the *Reeb graph* used in Morse Theory [16, 17, 18, 20]. It can be computed in $O(n \log n)$ time for functions over 2D [7].

This paper includes the following results.

- We give a new, simple algorithm that constructs the contour tree. For 2D meshes with n elements, it runs in $O(n \log n)$ time like a previous algorithm [7], but the new method is much simpler and needs less additional storage. For meshes with n faces in d-space, it runs in $O(n^2)$ time. In typical cases, less than linear temporary storage is needed

during the construction, which is important in practice. Also, the higher-dimensional algorithm requires subquadratic time in typical cases.

- We show that $\Omega(n \log n)$ is a lower bound for the construction of the contour tree.
- We show that the contour tree is the appropriate structure to use when selecting seed sets. We give an $O(n^2 \log n)$ time time algorithm for minimum size seed sets by using minimum cost flow in a DAG [1].
- In practice one would like a close to linear time algorithm when computing seed sets. We give a simple approximation algorithm that requires $O(n \log^2 n)$ time and linear storage after construction of the contour tree. It gives at most twice as many seeds as the minimum size seed set.
- The approximation algorithm has been implemented, and we supply test results of various kind.

Previous methods to find small size seed sets didn't give any guarantee on their size [4, 12, 22]. Recently, Tarasov and Vyalyi [21] extended our contour tree construction algorithm and obtained an $O(n \log n)$ time algorithm for 3D. Their algorithm consists of a preprocessing step with two sweeps, after which our algorithm is used. They use our 3D algorithm but with the idea of the 2D algorithm to obtain the efficiency.

2 Preliminaries on scalar functions and the contour tree

In this section we provide preliminary backgroud and definitions of terms used in the following. On a continuous function $\mathcal{F}$ from d-space to the reals, the *criticalities* can be identified. These are the local maxima, the local minima, and the saddles (or passes). If we consider all contours of a specified function value, we have a collection of lower-dimensional regions in d-space (typically, $(d-1)$-dimensional surfaces of arbitrary topology). If we let the function value take on the values from $+\infty$ to $-\infty$, a number of things may happen to the contours. Contour shapes deform continuously, with changes in topology only when a criticality is met (i.e., its function value is passed). A new contour component starts to form whenever the function value is equivalent to a locally maximal value of $\mathcal{F}$. An existing contour component disappears whenever the function value is equivalent to a locally minimal value.

At saddle points, various different things can happen. It may be that two (or more) contour components adjoin, or one contour component splits into two (or more) components, or that a contour component gets a different topological structure (e.g., from a sphere to a torus in 3D). The changes that can occur have been documented well in texts on Morse theory or differential topology [10, 15]. They can be described by a structure called the contour tree, which we describe shortly.

As an example, consider a function modelled by a 2D triangular mesh with linear interpolation and consider how the contour tree relates to such meshes. For simplicity, we assume that all vertices have a different function value. If we draw the contours of all vertices of the mesh, then we get a subdivision of the 2D domain into regions. All saddle points, local minima and maxima must be vertices of the mesh in our setting. The contour through a local mimimum or maximum is simply the point itself. One can show that every region between contours is bounded by exactly two contours [7]. We let every contour in this subdivision

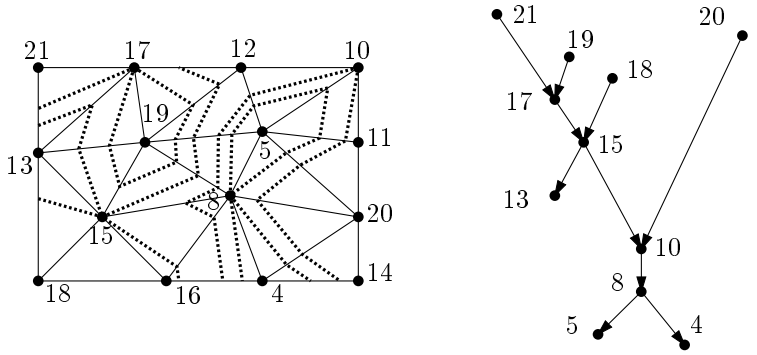

Figure 1: 2D triangular mesh with the contours of the saddles, and the contour tree.

correspond to a node in a graph, and two nodes are connected if there is a region bounded by their corresponding contours. This graph is a tree, which is easy to show [7, 22], and it is called the contour tree. All nodes in the tree have degree 1 (corresponding to local extrema), degree 2 (normal vertices), or at least 3 (saddles). In other words, every contour of a saddle vertex splits the domain into at least three regions. For each vertex in the triangulation, one can test locally whether it is a saddle. This is the case if and only if it has neighboring vertices around it that are higher, lower, higher, and lower, in cyclic order around it. If one would take the approach outlined above to construct the contour tree, $\Omega(n^2)$ time may be necessary in the worst case, because the total combinatorial complexity of all contours through saddles may be quadratic. An $O(n \log n)$ time divide-and-conquer algorithm exists, however [7].

In a general framework, we define the contour tree without assumptions on the type of mesh, interpolant, and dimension of the space over which function $\mathcal{F}$ is defined. The input data is assumed to be:

- a mesh M of size n embedded in $\mathbb{R}^d$;
- a continuous real-valued function $\mathcal{F}$ defined over all cells of M.

A *contour* is defined to be a maximal connected piece of $\mathbb{R}^d$ where the function value is the same. Usually, a contour is a $(d-1)$-dimensional hypersurface, but it can also be lower dimensional or d-dimensional. We define the contour tree $\mathcal{T}$ as follows.

- Take each contour that contains a criticality.
- These contours correspond to the *supernodes* of $\mathcal{T}$ (the tree will be extended later with additional nodes, hence we use the term supernodes here). Each supernode is labeled with the function value of its contour.
- For each region bounded by two contours, we add a superarc between the corresponding supernodes in $\mathcal{T}$.

The contour tree is well defined, because each region is bounded by two and only two contours which correspond to supernodes. In fact, it is easy to see that the contour tree is a special case of the more general Reeb graph in the $(d+1)$-dimensional space obtained from the domain (the mesh) extended with the function image space [16, 17, 18, 20]. Furthermore, one can

show that the contour tree is indeed a tree: the proof for the two-dimensional case given in [7] can easily be extended to d dimensions.

For 2D meshes, all criticalities correspond to supernodes of degree 1, or degree 3 or higher. For higher-dimensional meshes there are also criticalities that correspond to a supernode of degree 2. This occurs for instance in 3D when the genus of a surface changes, for instance when the surface of a ball changes topologically to a torus (Figure 2(b)).

Superarcs are directed from higher scalar values to lower scalar values. Thus, supernodes corresponding to the local maxima are the sources and the supernodes corresponding to the local minima are the sinks.

To be able to compute the contour tree, we make the following assumptions:

- Inside any face of any dimension of M, all criticalities and their function values can be determined.
- Inside any face of any dimension of M, the range (min, max) of the function values taken inside the face can be determined.
- Inside any face of any dimension of M, the (piece of) contour of any value in that face can be determined.

We assume that in facets and edges of 2D meshes, the items listed above can be computed in $O(1)$ time. For vertices, we assume that the first item takes time linear in its degree. Similarly, in 3D meshes we assume that both items take $O(1)$ to compute in cells and on facets, and time linear in the degree on edges and at vertices.

In d-dimensional space, a saddle point p is a point such that for any sufficiently small hypersphere around p, the contour of p's value intersects the surface of the hypersphere in at least two separate connected components. Possible criticalities in the 3-dimensional case

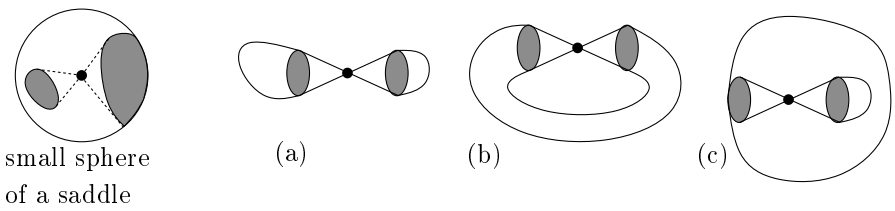

Figure 2: Citicalities in 3D.

are shown in Figure 2. When sweeping the function value from ∞ to $-\infty$, they correspond to (a) two contours merging or splitting, but not containing the other, (b) an increment or decrement of the genus of one contour surface, and (c) two contours merging or splitting, and one containing the other. More cases can occur when a criticality causes several of these changes at once, or when the contour ends at the boundary of the mesh.

3 Contour tree algorithms

In this section we assume for simplicity that the mesh M is a simplicial decomposition with n cells, and linear interpolation is used. As a consequence, all critical points are vertices of the mesh M. Instead of computing the contour tree as defined in the previous section,

we compute an extension that includes nodes for the contours of all vertices of M, also the non-critical ones. So supernodes correspond to contours of critical vertices and regular nodes correspond to contours of other vertices. Each superarc is now a sequence of arcs and nodes, starting and ending at a supernode. The algorithm we'll describe next can easily be adapted to determine the contour tree with only the supernodes. But we'll need this extended contour tree for seed selection in the next section. From now on, we call the contour tree with nodes for the contours of all vertices the contour tree $\mathcal{T}$.

The supernodes of $\mathcal{T}$ that have in-degree 1 and out-degree greater than 1 are called *bifurcations*, and the supernodes with in-degree greater than 1 and out-degree 1 are called *junctions*. All normal nodes have in-degree 1 and out-degree 1. We'll assume that all bifurcations and junctions have degree exactly 3, that is, out-degree 2 for bifurcations and in-degree 2 for junctions. This assumption can be removed; one can represent all supernodes with higher degrees as clusters of supernodes with degree 3. For example, a supernode with in-degree 2 and out-degree 2 can be treated as a junction and a bifurcation, with a directed arc from the junction to the bifurcation. The assumption that all junctions and bifurcations have degree 3 facilitates the following descriptions considerably.

3.1 The general approach

To construct the contour tree $\mathcal{T}$ for a given mesh in d-space, we let the function value take on the values from $+\infty$ to $-\infty$ and we keep track of the contours for these values. In other words, we sweep the scalar value. For 2D meshes, one can image sweeping a polyhedral terrain embedded in 3D and moving downward a horizontal plane. The sweep stops at certain event points: the vertices of the mesh. During the sweep, we keep track of the contours in the mesh at the value of the sweep function, and the set of cells of the mesh that cross these contours. The cells that contain a point with value equivalent to the present function value are called *active*. The tree $\mathcal{T}$ under construction during the sweep will be growing at the bottom at several places simultaneously, see Figure 3. Each part of $\mathcal{T}$ that is still growing corresponds

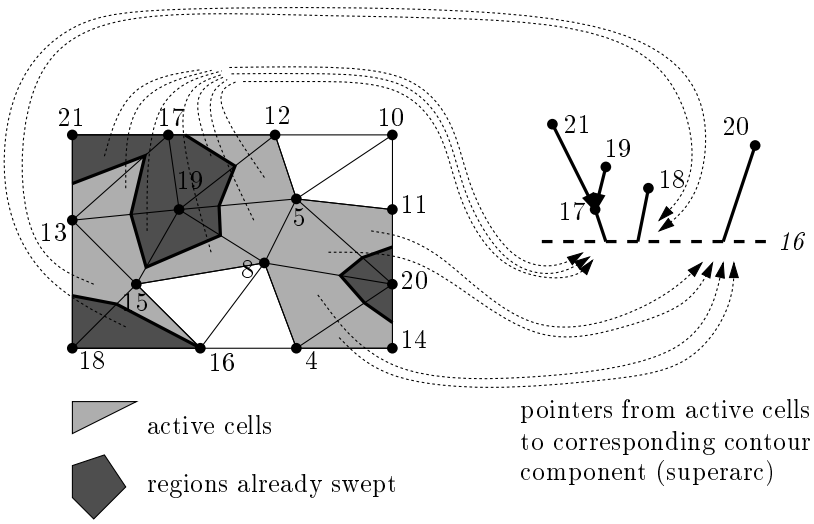

Figure 3: Situation of the sweep of a 2D mesh when the function value is 16.

to a unique contour at the current sweep value. We group the cells into contours by storing

a pointer at each active cell in the mesh to the corresponding superarc in $\mathcal{T}$. The contours can only change structurally at the event points, and the possible changes are the following:

- At a local maximum of the mesh (more correctly: of the function), a new contour appears. This is reflected in $\mathcal{T}$ by creating a new supernode and a new arc incident to it. This arc is also the start of a new superarc, which will be represented. Each cell incident to the maximum becomes active, and we set their pointer to the new superarc of $\mathcal{T}$. At this stage of the algorithm, the new superarc has no lower node attached to it yet.

- At a local minimum of the mesh, a contour disappears; a new supernode of $\mathcal{T}$ is created, and the arc corresponding to the disappearing contour at the current value of the sweep is attached to the new supernode. It is also the end of a superarc. The cells of the mesh incident to the local minimum are no longer active.

- At a non-critical vertex of the mesh, a new node of $\mathcal{T}$ is created, the arc corresponding to the contour containing the vertex is made incident to the node, and a new arc incident to the node is created. There is no new superarc. Some cells incident to the vertex stop being active, while others start being active. The pointers of the latter cells are set to the current superarc of the contour. For the cells that remain active, nothing changes: their pointer keeps pointing to the same superarc.

- At a saddle of the mesh, there is some change in topology in the collection of contours. It may be that two or more contours merge into one, one contour splits into two or more, or one contour changes its topological structure. A combination of these is also possible in general. The first thing to do is to determine what type of saddle we are dealing with. This can be decided by traversing the whole contour on which the saddle lies.

  If two contours merge, a new supernode (junction) is created in $\mathcal{T}$ for the saddle, and the superarcs corresponding to the two merging contours are made incident to this supernode. Furthermore, a new arc and superarc are created for the contour that results from the merge. The new arc is attached to the new supernode. All cells that are active in the contour after the merge set their pointer to the new superarc in $\mathcal{T}$.

  If a contour splits, then similar actions are taken. If the saddle is because of a change in topology of one single contour, a new supernode is made for one existing superarc, and a new arc and superarc are created in $\mathcal{T}$. All active cells of the contour set their pointers to the new superarc.

For the sweep algorithm, we need an event queue and a status structure. The event queue is implemented with a standard heap structure so insertions and extractions take logarithmic time per operation. The status structure is implicitly present in the mesh with the additional pointers from the cells to the superarcs in the contour tree.

Theorem 1 *Let M be a mesh in d-space with n faces in total, representing a continuous, piecewise linear function over the mesh elements. The contour tree of M can be constructed in $O(n^2)$ time and $O(n)$ storage.*

Proof: The algorithm clearly takes time $O(n \log n)$ for all heap operations. If the mesh is given in an adjacency structure, then the traversal of any contour takes time linear in the

combinatorial complexity of the contour. Any saddle of the function is a vertex, and any contour can pass through any mesh cell only once. Therefore, the total time for traversal is $O(n^2)$ in the worst case, and the same amount of time is needed for setting the pointers of the active cells. □

The quadratic running time shown above is somewhat pessimistic, since it applies only when there is a linear number of saddles for which the contour through them has linear complexity. We can also state that the running time as $O(n \log n + \sum_{i=1}^{m} |C_i|)$, where the m saddles lie on contours $C_1, \ldots, C_m$ with complexities $|C_1|, \ldots, |C_m|$.

We claimed that the additional storage of the algorithm could be made sublinear. With additional storage we mean the storage besides the mesh (input) and the contour tree (output), and we show that $O([\text{no. maxima}] + \max_{1 \leq i \leq m} |C_i|)$ extra storage suffices. We must reduce the storage rquirements of both the event queue and the status structure. additional storage required can be made linear in the maximum number of active cells and the number of local maxima. So this is $O([\text{no. maxima}] + \max_{1 \leq i \leq m} |C_i|)$ additional storage.

Regarding the event queue, we initialize it with the values of the local maxima only. During the sweep, we'll insert all vertices incident to active cells, as soon as the cell becomes active. This guarantees that the event queue uses no more additional storage than claimed above. Considering the status structure, we cannot afford using additional pointers with every cell of the mesh to superarcs any more. However, we need these pointers only when the cell is active. We'll make a copy of the active part of the mesh, and in this copy, we may use the additional pointers. When a cell becomes inactive again, we delete it from the copy. The asymptotic running time of the algorithm is not influenced by these changes.

3.2 The two-dimensional case

In the 2D case, the time bound can be improved to $O(n \log n)$ time in the worst case by a few simple adaptations. First, a crucial observation: for 2D meshes representing continuous functions, all saddles correspond to nodes of degree at least 3 in $\mathcal{T}$. Hence, at any saddle two or more contours merge, or one contour splits into at least two contours, or both. This is different from the situation in 3D, where a saddle can cause a change in genus of a contour, without causing a change in connectedness. The main idea is to implement a merge in time linear in the complexity of the *smaller* of the two contours, and similarly, to implement a split in time linear in the complexity of the *smaller resulting contour*.

In the structure, each active cell has a pointer to a *name* of a contour, and the name has a pointer to the corresponding superarc in $\mathcal{T}$. We consider the active cells and names as a union-find like structure that allows the following operations:

- *Merge*: given two contours about to merge, combine them into a single one by renaming the active cells to have a common name,
- *Split*: given one contour about to split, split it into two separate contours by renaming the active cells for one of the contours in creation to a new name.
- *Find*: given one active cell, report the name of the contour it is in.

Like in the simplest union-find structure, a *Find* takes $O(1)$ time since we have a pointer to the name explicitly. A *Merge* is best implemented by changing the name of the cells in

smaller contour to the name of the larger contour. Let's say that contours C_i and C_j are about to merge. Determining which of them is the smallest takes $O(\min(|C_i|, |C_j|))$ time if we traverse both contours simultaneously. We alternatingly take one "step" in C_i and one "step" in C_j. After a number of steps twice the combinatorial complexity of the smaller contour, we have traversed the whole smaller contour. This technique is sometimes called *tandem search*. To rename for a *Merge*, we traverse this smaller contour again and rename the cells in it, again taking $O(\min(|C_i|, |C_j|))$ time.

The *Split* operation is analogous: if a contour C_k splits into C_i and C_j, the name of C_k is preserved for the larger of C_i and C_j, and by tandem search starting at the saddle in two opposite directions we find out which of C_i and C_j will be the smaller one. This will take $O(\min(|C_i|, |C_j|))$ time. Note that we cannot keep track of the size in an integer for each contour instead of doing tandem search, because a *Split* cannot be supported efficiently.

Theorem 2 *Let M be a mesh in 2D with n faces in total, representing a continuous, piecewise linear scalar function. The contour tree of this function can be computed in $O(n \log n)$ time and linear storage.*

Proof: We can distinguish the following operations and their involved costs:

- Determining for each vertex of what type it is (min, max, saddle, normal) takes $O(n)$ in total.
- The operations on the event queue take $O(n \log n)$ in total.
- Creating the nodes and arcs of $\mathcal{T}$, and setting the incidence relationships takes $O(n)$ time in total.
- When a cell becomes active for the first time, the name of the contour it belongs to is stored with it; this can be done in $O(1)$ time, and since there are $O(n)$ such events, it takes $O(n)$ time in total.
- At the saddles of the mesh, contours merge or split. Updating the names of the contours stored with the cells takes $O(\min(|C_i|, |C_j|))$ time, where C_i and C_j are the contours merging into one, or resulting from a split, respectively. It remains to show that summing these costs over all saddles yields a total of $O(n \log n)$ time.

We prove the bound on the summed cost for renaming by transforming $\mathcal{T}$ in two steps into another tree $\mathcal{T}'$ for which the construction is at least as time-expensive as for $\mathcal{T}$, and showing that the cost at the saddles in $\mathcal{T}'$ are $O(n \log n)$ in total.

Consider the cells to be additional *segments* in $\mathcal{T}$ as follows. Any cell becomes active at a vertex and stops being active at another vertex. These vertices are nodes in $\mathcal{T}$, and the cell is represented by a segment connecting these nodes. Note that any segment connects two nodes one of which is ancestor of the other. A segment can be seen as a shortcut of a directed path in $\mathcal{T}$, where it may pass over several nodes and supernodes.

The number of cells involved in a merge or split at a saddle is equivalent to the number segments that pass over the corresponding supernode in $\mathcal{T}$; the size of the smallest set at this node determines the costs for processing the saddle (since we do tandem search).

The first transformation step is to *stretch* all segments (see Figure 4); we simply assume that a segment starts at some source node that is an ancestor of the original start node, and

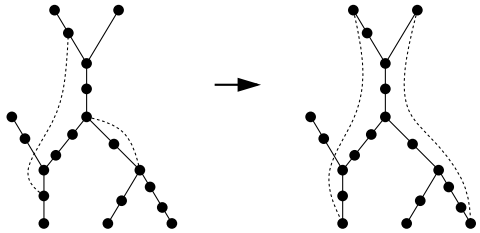

Figure 4: Stretching two segments (dotted) in $\mathcal{T}$.

ends at a sink that is a descendant of the original end node. It is easy to see that the number of segments passing any saddle can only increase by the stretch.

The second transformation step is to repeatedly *swap* superarcs, until no supernode arising from a split (bifurcation) is an ancestor of a supernode arising from a merge (junction). Swapping a superarc s from a bifurcation v to a junction u is defined as follows (see Figure 5):

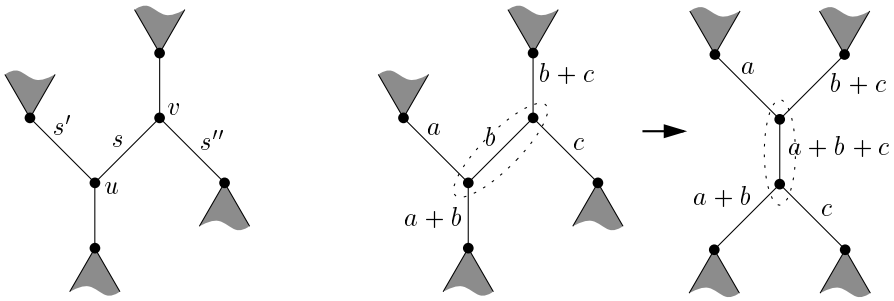

Figure 5: Swapping a superarc.

let $s' \neq s$ be the superarc that has u as its lower supernode, and let $s'' \neq s$ be the superarc that has v as its upper supernode. The number of segments passing the superarcs s', s and s'' is denoted by a, b and c, respectively, as is illustrated in Figure 5. These numbers are well-defined, since after stretching, any segment passes a superarc either completely or not at all. Now shift s' upward along s, such that v becomes its new lower supernode, and shift s'' downward along s, such that u becomes its new upper supernode. Note that all edges passing s' and all edges passing s'' before the swap now also pass s.

Before the swap, the time spent in the merge at u and the split at v, is $O(\min(a,b) + \min(b,c))$ where a, b, c denote the number of segments passing these superarcs. After the swap, this becomes $O(\min(a, b+c) + \min(a+b, c))$, which is at least as much. No segment ends, because all of them were stretched.

It can easily be verified that the $\mathcal{T}'$, with no bifurcation as an ancestor of a junction, can be derived from any tree $\mathcal{T}$ by swaps of this type only.

Now, every segment can pass $O(n)$ junctions and bifurcations, but no segment can be more than $O(\log n)$ times in the smaller set. Each time it is in the smaller set, it will be in a set of at least twice the size. Summing over the $O(n)$ segments, this results in a total of $O(n \log n)$ time for all renamings of cells. This argument is standard in the analysis of union-find structures, for instance [6]. □

As we noted before, Tarasov and Vyalyi [21] recently succeeded in extending the ideas above and obtain an $O(n \log n)$ time algorithm to construct the contour tree for 3D meshes.

The $O(n \log n)$ bounds for the contour tree construction are tight: Given a set S of n numbers $s_1, \ldots, s_n$, we can construct in $O(n)$ time a triangular mesh with n saddles at heights $s_1, \ldots, s_n$, such that in the corresponding contour tree all the saddles lie in sorted order along the path from the global minimum to the global maximum.

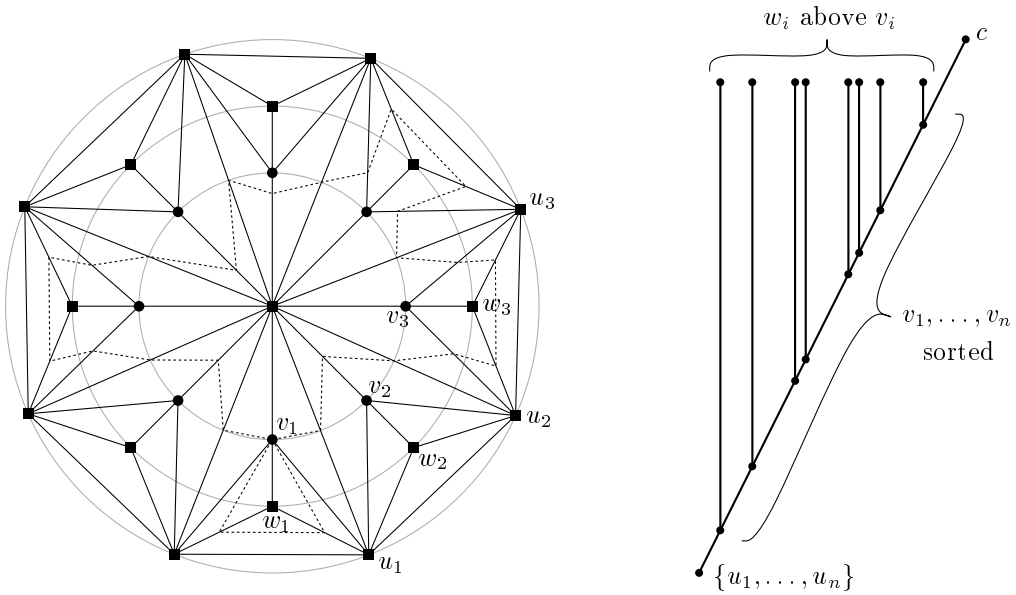

Figure 6: Unstructured mesh of which the contour tree contains the sorted sequence of the input values. The contour through v_1 is shown; we must have $v_2 < v_1 < v_3$.

The mesh is constructed as follows (see Figure 6): We place n vertices $v_1, \ldots, v_n$ equally spaced on a circle C in the (x, y)-plane with radius 2 and center at a point c. Now we elevate each v_i such that its z-coordinate is s_i. These vertices will be the saddles in the terrain. Next, we place n vertices $w_1, \ldots, w_n$ at a circle C' with radius 3 and also centered at c, such that each w_i is collinear with c and v_i. We elevate each w_i to height $\max(S)+1$; these vertices will be the local maxima. At the center c of C and C', we place one vertex at height $\max(S)+2$: the global maximum. Finally, we place a third set of vertices $u_1, \ldots, u_n$ at a circle C'' with radius 4 and centered at c, such that each vertex u_i is radially interleaved with the vertices v_i and v_{i+1}. The height of all these vertices u_i is $\min(S) - 1$; all these vertices lie on the global minimum. Edges in the terrain are as shown in Figure 6, and the corresponding contour tree is shown in the same Figure.

4 Seed set selection

A seed set is a subset of the cells of the mesh. Such a set serves as a set of starting points from which contours can be traced, for instance for visualization. A seed set is *complete* if every possible contour passes through at least one seed. Since we assume linear interpolation over the cells, the function values occurring in one cell is exactly the range between the lowest

and the highest valued vertices. Any cell is represented as a *segment* between two nodes of the contour tree $\mathcal{T}$, as in the proof of Theorem 2. Segments can only connect two nodes of which one in an ancestor of the other. Like the arcs of $\mathcal{T}$, the segments are are directed from the higher to the lower values. So each segment is in fact a shortcut of a directed path in $\mathcal{T}$. We say that the segment *passes*, or *covers*, these arcs of $\mathcal{T}$. Let $\mathcal{G}$ denote the directed acyclic graph consisting of the contour tree extended with the segments of all mesh elements. The small seed set problem now is the following graph problem: find a small subset of the segments such that every arc of $\mathcal{T}$ is passed by some segment of the subset.

In this section we give two methods to obtain complete and provably small seed sets. The first gives a seed set of minimum size possible, but it requires $O(n^2 \log n)$ time for its computations. The second method requires $O(n \log^2 n)$ time and linear storage (given the contour tree and the segments), and gives a seed set at most twice the size of the minimum possible.

4.1 Minimum size seed sets in polynomial time

We can find a minimum size seed set in polynomial time by reducing the seed set selection problem to a minimum cost flow problem. The flow network $\mathcal{G}'$ derived from $\mathcal{G}$ is defined as follows: we augment $\mathcal{G}$ with two additional nodes, a *source* σ and a *sink* σ'. The source σ is connected to all maxima and bifurcations by additional segments, and the sink is connected to all minima and junctions with additional segments. This is illustrated in Figure 7, left. In the same figure (right) a shorthand for the same flow network has been drawn: for readability, σ and σ' have been left out, and the additional segments incident to σ and σ' have been replaced by "+" and "−" signs, respectively. From now on we will use this shorthand notation in the figures.

Costs and capacities for the segments and arcs are assigned as follows: nodes in $\mathcal{G}$ are ordered on the height of the corresponding vertices in the mesh, and segments and arcs are considered to be directed: segments (dotted) go downward from higher to lower nodes, arcs (solid) go upward from lower to higher nodes. The source σ is considered to be the highest node, and σ' the lowest. Segments in $\mathcal{G}$ have capacity 1 and cost 1, and arcs have capacity ∞ and cost 0. The additional segments in $\mathcal{G}'$ incident to σ and σ' also have capacity 1, but zero cost.

From graph theory we have the following lemma:

Lemma 1 *For any tree, the number of maxima plus the number of bifurcations equals the number of minima plus the number of junctions.*

Hence, the number of pluses in $\mathcal{G}$ balances the number of minuses. Let this number be f.

Consider the following two related problems, the *flow problem* (given the flow network $\mathcal{G}'$ as defined above and a value f, find a flow of size f from σ to σ'), and the *minimum cost flow* problem (find such a flow f with minimum cost). For both problems, a solution consists of an assignment of flow for each segment and arc in $\mathcal{G}'$. For such a solution, let the *corresponding segment set* $\mathcal{S}$ be the set of segments in $\mathcal{G}$ that have a non-zero flow assigned to them (the additional segments in $\mathcal{G}'$ from σ to the maxima and bifurcations and from the minima and junctions to σ' are not in $\mathcal{S}$). Hence, the cost of an *integral solution*, where all flow values are integer, equals the number of segments in $\mathcal{S}$. We will show that for any integral solution to the minimum cost flow problem on $\mathcal{G}'$, the corresponding segment set $\mathcal{S}$ is a minimum size seed set for $\mathcal{G}$.

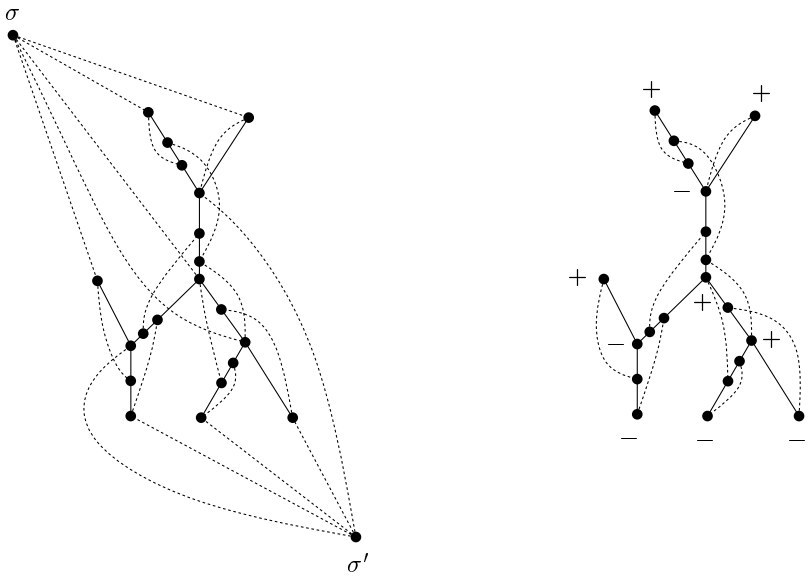

Figure 7: Flow network $\mathcal{G}'$ derived from $\mathcal{G}$, and shorthand for $\mathcal{G}'$.

Lemma 2 *For any integral solution to the flow problem on $\mathcal{G}'$, the corresponding segment set $\mathcal{S}$ is a seed set for $\mathcal{G}$.*

Proof: Suppose that there is a a flow of size f from σ to σ' in $\mathcal{G}'$ such that the corresponding segment set $\mathcal{S}$ is not a seed set for $\mathcal{G}$. In other words: there is an arc a in $\mathcal{G}'$ such that none of the segments in $\mathcal{G}'$ covering a has a non-zero flow assigned to it. We claim that the number of pluses in the subtree incident to and 'above' the highest incident node of a exceeds the number of minuses by one, and, analogously, that the number of minuses in the subtree incident to and 'below' the lowest incident node of a exceeds the number of pluses by one. This claim can easily be verified by lemma 1. Since there is no downward flow via a or any of its covering segments, the flow from σ to σ' can be at most $f - 1$. □

A seed set is *minimal* if the removal of any segment yields a set that is not a seed set. A *minimum* seed set is a seed set of smallest size.

Lemma 3 *For any minimal seed set S for $\mathcal{G}$, there is a solution to the flow problem on $\mathcal{G}'$ such that the corresponding segment set $\mathcal{S}$ for that solution equals S.*

Proof: We show this by induction on n, the number of nodes of $\mathcal{G}$. It is straightforward to verify that the lemma holds for $n \leq 3$. For $n > 3$, we observe that for any minimal seed set S there is at least one arc a in $\mathcal{G}$ that is covered by precisely one segment $s \in S$ (and possibly by some segments in $\mathcal{G}$ that are not in S); otherwise, removing an arbitrary segment from S would yield a smaller seed set, contradicting the minimality of S. If this arc a is not incident to a leaf node of $\mathcal{G}$, then we can split $\mathcal{G}$ into two subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ by cutting a and the segment s covering it. This introduces a new minimum for one of the subgraphs, and a new maximum for the other (Figure 8). Let S_1 be the set of segments from S that cover $\mathcal{G}_1$, with $s \in S$ replaced by the appropriate segment resulting from cutting s into two parts, and define S_2 in a similar way. S_1 and S_2 are minimal seeds sets for $\mathcal{G}_1$ and $\mathcal{G}_2$, respectively,

and both subgraphs have fewer than n nodes. Hence, by induction, there is a solution for the flow problem on $\mathcal{G}'_1$ such that the corresponding segment set $\mathcal{S}_1$ for that solution equals S_1, and there is a solution for the flow problem on $\mathcal{G}'_2$ such that the corresponding segment set $\mathcal{S}_2$ for that solution equals S_2. Note that the sum of the sizes of the flows for both subgraphs is $f + 1$, since we added a plus and a minus in the splitting process. Given the solutions to the flow problems for both subgraphs, it is straightforward to contruct a flow that solves the flow problem for $\mathcal{G}$: simply remove the plus and minus that were added in the split, and undo the cutting of a and s.

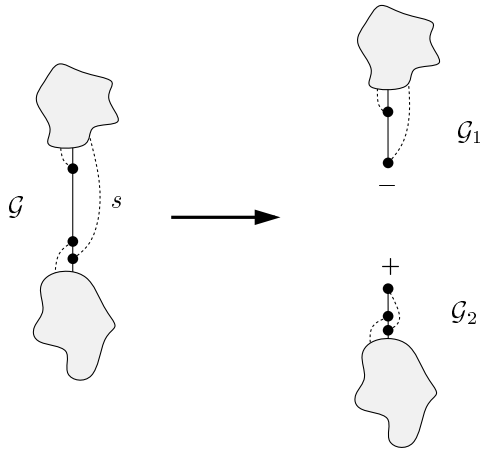

Figure 8: Splitting $\mathcal{G}$ at an internal arc.

This only works if a is not incident to a leaf node of $\mathcal{G}$, otherwise the split operation results in two subtrees, one with 2 nodes and one with n nodes, and we cannot apply induction. As noted before, there is at least one arc a that is covered by precisely one segment $s \in S$, and by zero or more segments that are in $\mathcal{G}$ but not in S. So suppose that all arcs covered by only one segment are incident to a minimum or maximum. Let a be one of those arcs, and assume that a is incident to a maximum ν of $\mathcal{G}$ (the case that a is incident to a minimum is analogous). Let the other endpoint of s be μ. As stated before, we can not apply induction directly. Instead, we tranform $\mathcal{G}$ by shortening some of its edges, such that S remains a minimal seed set and $\mathcal{G}$ can be split into two subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with with fewer than n nodes and with corresponding minimal seed sets S_1 and S_2. By induction, there is a flow of size $|S_1|$ for $\mathcal{G}'_1$ and a flow of size $|S_2|$ for $\mathcal{G}'_2$. Because the simplicity of the reduction from $\mathcal{G}$ to $\mathcal{G}_1$ and $\mathcal{G}_2$, it is straightforward to construct a flow for $\mathcal{G}'$ corresponding to S, given the flows for $\mathcal{G}'_1$ and $\mathcal{G}'_2$.

We distinguish four cases:

- ν and μ lie on the same superarc of $\mathcal{G}$ (see Figure 9). In that case, we transform $\mathcal{G}$ by retracting all segments that pass μ (i.e., μ is made their highest incident node), and by removing s and all arcs between μ and ν. Now $S \setminus \{s\}$ is a minimal seed set for the resulting graph $\mathcal{G}_1$, which has fewer than n nodes, so now we can apply induction. In this case, $\mathcal{G}_2$ is the empty graph. It is straightforward to construct a solution to the flow problem for $\mathcal{G}'$, given a solution to the flow problem for $\mathcal{G}'_1$.
- μ is not a minimum of $\mathcal{G}$ for which s is the only segment in S incident to it, and the

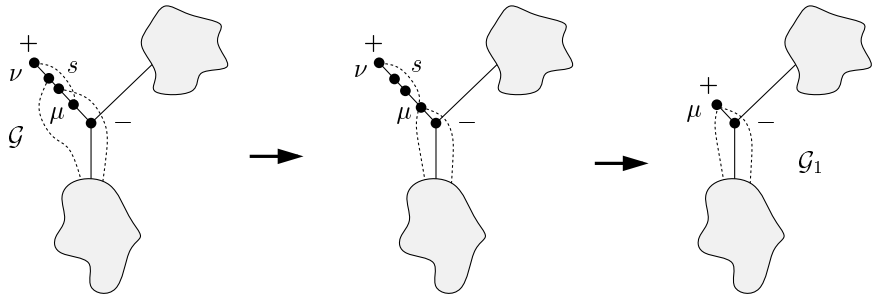

Figure 9: Retracting segments to the lower endpoint μ of s.

first supernode on the path from ν to μ is a bifurcation (see Figure 10). The segments passing that bifurcation that go into the same subtree as s remain as they are; the ones that go into the other subtree are retracted to the bifurcation. $\mathcal{G}$ is split into two subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$ by cutting off the latter subtree. S_i and S_j are minimal seed sets for the two resulting subgraphs, both of which have fewer than n nodes. Again we can apply induction, and find a solution to the flow problem for $\mathcal{G}'$, given the solutions for $\mathcal{G}'_1$ and $\mathcal{G}'_2$.

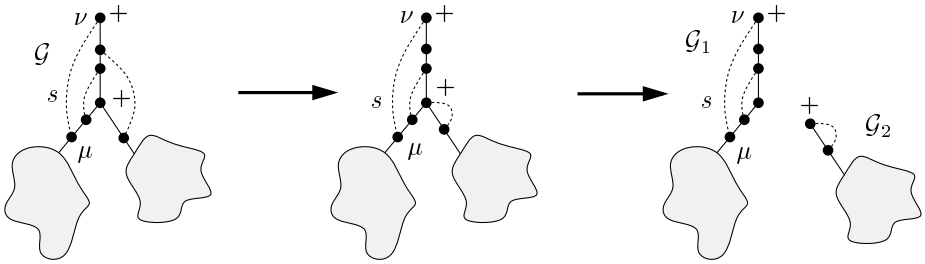

Figure 10: Retracting segments to a bifurcation.

- μ is not a minimum of $\mathcal{G}$ for which s is the only segment in S incident to it, and the first supernode on the path from ν to μ is a junction (see Figure 11). This is almost the same as the previous case; the transformations are shown in the figure. As before, induction can be applied to both subtrees, and a solutions for the flow problem for $\mathcal{G}'$ can easily be derived from the solutions for the subproblems.
- μ is a minimum of $\mathcal{G}$ for which s is the only segment in S incident to it. The first supernode on the path from ν to μ can be a junction or a bifurcation, and the same holds for the last supernode on this path, which gives us four subcases. Each subcase is solved by a combination of the previous two cases; one example is shown in Figure 12.

□

Combining Lemmas 2 and 3 gives the following result:

Theorem 3 *The minimum seed set selection problem for $\mathcal{G}$ can be solved by applying a minimum cost flow algorithm to $\mathcal{G}'$ that gives an integral solution. Such a solution is guaranteed to exist, and the corresponding segment set for that solution is an optimal seed set for $\mathcal{G}$.*

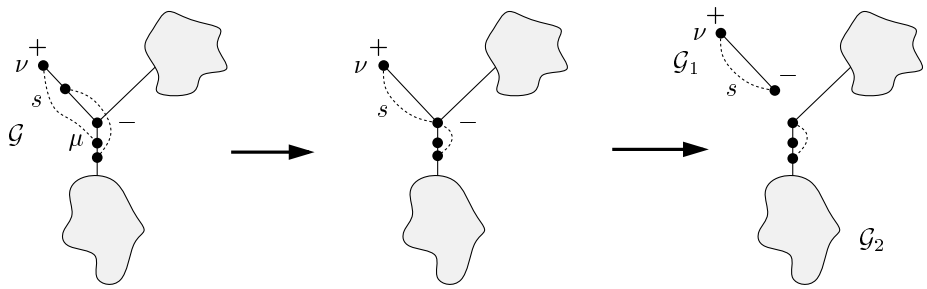

Figure 11: Retracting segments to a junction.

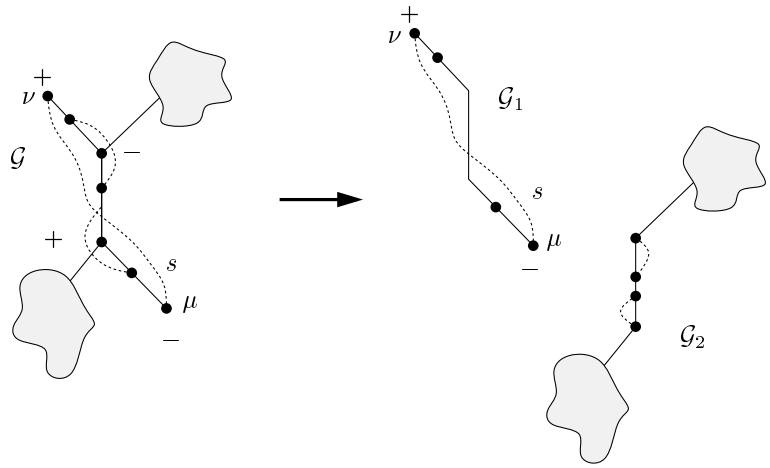

Figure 12: Combining the previous splits.

The minimum cost flow problem can be solved with a successive shortest path algorithm[1] (pp. 320–324). Starting with a zero flow, this algorithm determines at every step the shortest path π from σ to σ', where the length of an arc or segment is derived from its cost. The arc or segment with the lowest capacity c on this shortest path π determines the flow that is sent from σ to σ' along π. Then the *residual network* is calculated (costs and capacities along π are updated), and the algorithm iterates.

In our case, c is always 1 and the algorithm terminates after f iterations. If we use Dijkstra's algorithm to find the shortest path in each iteration, the algorithm runs in $O(n^2 \log n)$ time on our graph $\mathcal{G}'$, and uses $O(n)$ memory.

Corollary 1 *An optimal seed set for $\mathcal{G}$ can be found in $O(n^2 \log n)$ time, using $O(n)$ memory.*

4.2 Efficient approximation of small seed sets

The roughly quadratic time requirements for optimal seed sets makes it rather time consuming in practical applications. We therefore developed an approximation algorithm to compute a seed set that, after constructing the contour tree $\mathcal{T}$, uses linear storage and $O(n \log^2 n)$ time in any dimension. It yields a seed set of size no more than twice the size of the smallest seed set.

As before we will describe the algorithm in the simplified situation that each critical vertex of the mesh is either a minimum, maximum, junction, or bifurcation. In the case of a junction or bifurcation, we assume that the degree is exactly three. These simplifying assumptions make it easier the explain the algorithm, but they can be removed as before.

Our approximation algorithm is a simple greedy method that operates quite similarly to the contour tree construction algorithm. We first construct the contour tree $\mathcal{T}$ as before. We store with each node of $\mathcal{T}$ two integers that will help determine fast if any two nodes of $\mathcal{T}$ have an ancestor/descendant relation. The two integers are assigned as follows. Give $\mathcal{T}$ some fixed, left-to-right order of the children and parents of each supernode. Then perform a left-to-right topological sort to number all nodes. Then perform a right-to-left topological sort to give each node a second number. The numbers are such that one node u is an ancestor of another node v if and only if the first number and the second number of u is smaller than the corresponding numbers of v (see Figure 13). This preprocessing of the contour tree takes

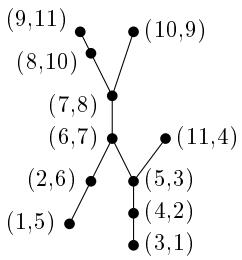

Figure 13: The numbering of $\mathcal{T}$.

$O(n)$ time, and afterwards, we can determine in $O(1)$ time for any two nodes whether one is a descendant or ancestor of the other.

Next we add the segments, one for each cell of the mesh, to the contour tree $\mathcal{T}$ to form the graph $\mathcal{G}$. Then we sweep again, now in the mesh and in the graph $\mathcal{G}$ simultaneously. At each event point of the sweep algorithm (the nodes of $\mathcal{T}$), we test whether the arc incident to and below the current node is covered by at least one of the already selected seeds. If this is not the case, then we select a new seed. The new seed will always be the greedy choice, that is, the segment (or cell) for which the function value of the lower endpoint is minimal. To determine if a new seed must be chosen, and to be able to make a greedy choice, a few data structures are needed that maintain the currently chosen seed set and the candidate seeds that may be chosen next. As before, we call the cells that contain the current sweep value *active*. The segments and seeds of currently active cells are also called active. Similarly, the superarcs for which the higher supernode has been passed, but the lower one not yet, are called active. We maintain the following sets during the sweep:

- A set S of currently chosen seeds.

  Initially, this set is empty; at the end of the algorithm, S contains a complete set of seeds.

- For an active superarc a, let $\hat{S}_a$ be the set of active seeds (already chosen) that cover a or part of it. We store a subset $S_a \subseteq \hat{S}_a$ that only contains the "deepest going" seeds of $\hat{S}_a$. More precisely, for all $s, s' \in \hat{S}_a$, if s is an ancestor of s', then s is not in S_a.

- For each active superarc a, a set $\hat{C}_a$ of active candidate seeds that cover a or part of it. We store a subset $C_a \subseteq \hat{C}_a$ that only contains the deepest going candidates of $\hat{C}_a$, and only if they go deeper than seeds of S_a. More precisely, for all $c, c' \in \hat{C}_a$ and $s \in S_a$, if c is an ancestor of c' or s, then c is not in C_a.

The algorithm to be described needs the chosen seeds to be able to determine if the next arc to be swept of superarc a is covered by some chosen seed. The subset S_a is exactly the subset of nonredundant seeds of $\hat{S}_a$. Similarly, the algorithm needs to maintain candidates that can be chosen if the next arc to be swept is not covered. The set $\hat{C}_a$ contains the active candidates, but the subset C_a contains only those candidates that could possibly be chosen. We'll show next that S_a and C_a can simply be stored in balanced binary trees.

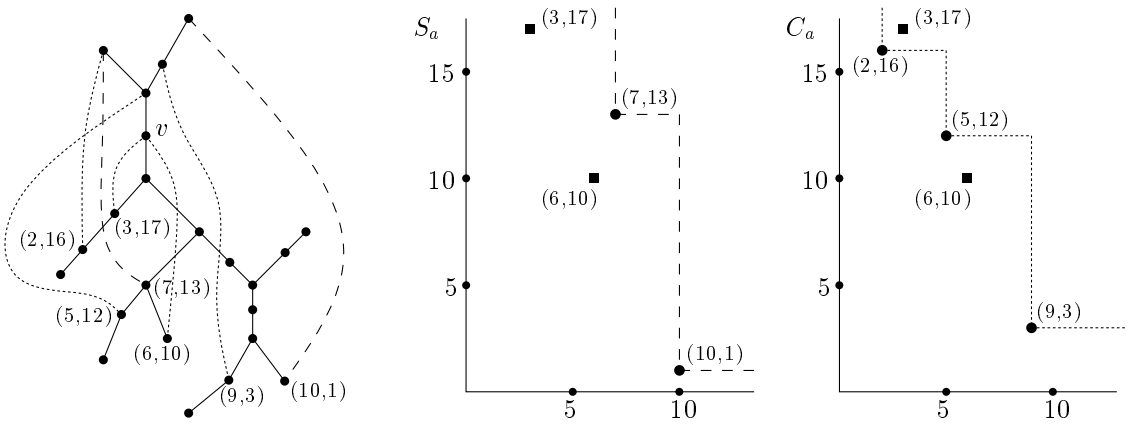

Figure 14: Just before the sweep reaches node v, the staircases of the active, chosen seeds in S_a (dashed) and of the active candidates in C_a (dotted).

The sets S_a and C_a correspond to a set of points in the plane whose coordinates are the two numbers assigned to the lower endpoints of the segments in S_a and C_a, see Figure 14. Since there are no ancestor/predecessor relationships between the endpoints of the segments in one set, none of the corresponding points lies to the right and above (or to the left and below) any other point in the same set; the points form a so-called *staircase*. This means that S_a and C_a can each be maintained as a binary search tree, ordered on the first number assigned to the lower endpoints of the segments alone. An *ancestor query* with a point (x, y) asks if the set contains a point (i, j) for which $x \geq i$ and $y \geq j$. Answering such a query is done by finding the point with maximum first number $\leq x$, and testing if this point has its second number $\leq y$. Similarly, we can determine easily whether a query point (x, y) has both numbers smaller than some point in the tree—a *descendant query*. Since the sorted order on the first number suffices, queries, insertions, and deletions in S_a and C_a can be done in time logarithmic in the size of the set.

We also maintain a heap on the set C_a, or rather, on the lower endpoints of the candidates in C_a, with cross-pointers between corresponding nodes in the heap and the binary tree for C_a. The heap allows us to extract efficiently the candidate segment with the lowest lower endpoint from C_a.

We will now describe the sweep algorithm that computes a small seed set, and analyze the total running time. We initialize the global set S of seeds to be empty. Then we sweep the nodes in $\mathcal{T}$ from high to low values. The following events can occur:

- Source: Initialize empty sets S_a and C_a for the superarc a starting at this node. This takes $O(1)$ time. Next, proceed as if the current node were a normal node.

- Normal node v: First, update the set C_a of candidate seeds for the superarc a on which the current node v lies. For each segment s that starts at v, we determine how it affects the set C_a. Let u be the lower endpoint of segment s. Perform an ancestor query on the tree storing S_a; if u is ancestor of the lower endpoint of any seed in S_a, we don't need the segment as a candidate seed. In Figure 14, the queries are performed with the segments that have lower endpoints at $(3, 17)$ and $(6, 10)$. Otherwise, perform an ancestor query with u on C_a. If u is an ancestor of the lower endpoint of any of the candidates in C_a, we also don't need the segment as a candidate. Otherwise, perform a descendant query with u. If u is the descendant of the lower endpoint of some candidate in C_a (there is at most one such candidate), then replace this candidate seed with the segment s. If u has no ancestor or descendant relation, then the query segment becomes a candidate seed; it is inserted in the binary tree and the heap for C_a. Note that we never have to worry about candidate seeds no longer being active; they will be replaced by newer candidates before this happens.

  Next, test whether the arc of $\mathcal{T}$ starting at v is covered by any of the active seeds in S_a. This is surprisingly easy: if $|S_a| > 1$, the lower endpoints of the segments in S_a lie in different subtrees rooted at one or more bifurcations below the current node, since there are no ancestor/descendant relations between the endpoints of the segments in S_a. This means that the arc incident to and and below the current node is surely covered. On the other hand, if $|S_a| = 1$, we check in constant time whether the segment in S_a ends at the current node. If that is the case, we have to remove the only segment from S_a and choose a new seed, otherwise we are done. Choosing a new seed is also easy: Extract the candidate with the lowest lower endpoint using the heap on C_a, and remove this candidate from the binary tree on C_A as well, using the cross-pointers between the nodes in the heap and the binary tree. Next, insert this candidate as a seed in S_a and in the global set of seeds S.

  The total time needed for all queries, replacements, and insertions at node v is $O(d \log n)$, where d is the degree of v in $\mathcal{G}$.

- Sink: Remove the sets S_a and C_a.

- Junction: First, for the two incoming arcs a and b at the junction, we determine which of the two values is smaller: $|S_a| + |C_a|$ or $|S_b| + |C_b|$. This takes $O(1)$ time if we keep track of the size of the sets. Suppose without loss of generality that the first of the two sums is the smallest. Then, for each seed s in S_a, we do the following. Let u be the lower endpoint of s. Perform an ancestor and descendant query on S_b with u. If u is ancestor, we do nothing; if u is descendant of the lower endpoint of some $s' \in S_b$, we replace s' by s in the tree on S_b. Otherwise, there are no ancestor/descendant relations and we insert s in the tree on S_b. If s is stored in S_b, it may be that s renders at most one of the candidates in C_b redundant: we perform a descendant query with u on C_b to discover this, and if there is a candidate whose lower endpoint is ancestor of s, we remove this candidate from C_b. The time needed for this step of the merge is $O(k \log n)$, where $k = \min(|S_a| + |C_a|, |S_b| + |C_b|)$. The merged set of active seeds is denoted $S_{a,b}$.

  Next, we do something similar for the two sets of candidate seeds. For each candidate

c in C_a, let u be the lower endpoint of c. Perform an ancestor query with u on the set $S_{a,b}$ to test if c still is a valid candidate. If u is ancestor of the lower endpoint of some seed, then we discard c. Otherwise, we query C_b to see if u is an ancestor or descendant of the lower endpoint of a candidate c' in C_b. If u is the ancestor, we discard c; if u is the descendant, we replace c' by c. Otherwise, if there are no ancestor/descendant relations and we insert c in C_b.

Finally, we proceed as if the current node were a normal node.

Note that we cannot independently insert the seeds of the smaller set of S_a and S_b in the larger, and the candidates of the smaller set of C_a and C_b in the larger; we have to compare the seeds in S_a with the candidates in C_b, and the seeds in S_b with the candidates in C_a.

- Bifurcation: We have to split the set of active seeds S_a in two sets S_b and $S_{b'}$, the sets of active seeds for the left arc b and the right arc b' below the bifurcation, respectively. Note that the lower endpoint of any segment in S_b has a smaller first number assigned in the left-to-right topological sort of $\mathcal{T}$ than the lower endpoint of any segment in $S_{b'}$. Also note that we can test in $O(1)$ time in which subtree below the bifurcation a lower endpoint of a segment in S_a lies, by comparing it with the highest nodes in the subtrees. So, if we test the segments in S_a simultaneously from low to high and from high to low values of their lower endpoints, we can determine in $O(\min(|S_b|, |S_{b'}|))$ time which of the two resulting sets will be the smaller one. Once we know this, we can split the tree for S_a into trees for S_b and $S_{b'}$ in $O(\min(|S_b|, |S_{b'}|) \log n)$ time, by extracting the seeds that will go in the smaller set from S_a and inserting them in a new set. For the set C_a of candidates, we do exactly the same to obtain the sets C_b and $C_{b'}$.

  Next, we process the current node twice as a normal node, once for the arc and the segments that go into the left subtree of the current node, and once for the right subtree.

It is easy to verify that the total time needed to process all sources, sinks and normal nodes is $O(n \log n)$; the time needed for the junctions and bifurcations can be analyzed much the same way as in Section 3.2, where we analyzed the running time for the construction of $\mathcal{T}$ for a 2D-mesh. In this case we get a bound of $O(n \log^2 n)$, which is also a bound on the overall running time.

At any stage of the sweep, the memory requirements are proportional to the size of the binary trees of all active superarcs, which is $O(n)$ worst-case.

Lemma 4 *The approximation algorithm yields a seed set with at most b more seeds than an optimum seed set, where b is the number of bifurcations.*

Proof: First, consider a contour tree that only has maxima, junctions, and one single minimum. Then a greedy approach as described finds a smallest seed set, which is easy to prove by a standard replacement argument: Let S_{opt} be an optimal seed set, let S be the chosen seed set, and assume $S \neq S_{\text{opt}}$. Let $s \in S_{\text{opt}} - S$ be the segment that ends highest. Let a be the highest arc of $\mathcal{T}$ such that s covers a but no other seed of S_{opt} does so. Such an a must exist otherwise $S_{\text{opt}} \setminus \{s\}$ also covers all of $\mathcal{T}$, contradicting the optimality of S_{opt}. Since S and S_{opt} are the same for the subtree above a, the greedy algorithm makes its choice only when it reaches the upper node of a. Suppose the greedy choice is $s' \neq s$. Then s' has its lower node on or below the lower node of s, since $\mathcal{T}$ doesn't have bifurcations. It follows that

$\{s'\} \cup S_{\text{opt}} \setminus \{s\}$ is also an optimal seed set, but one which has one more segment in common with S. Also observe that a tree with only junctions is covered by at least as many segments

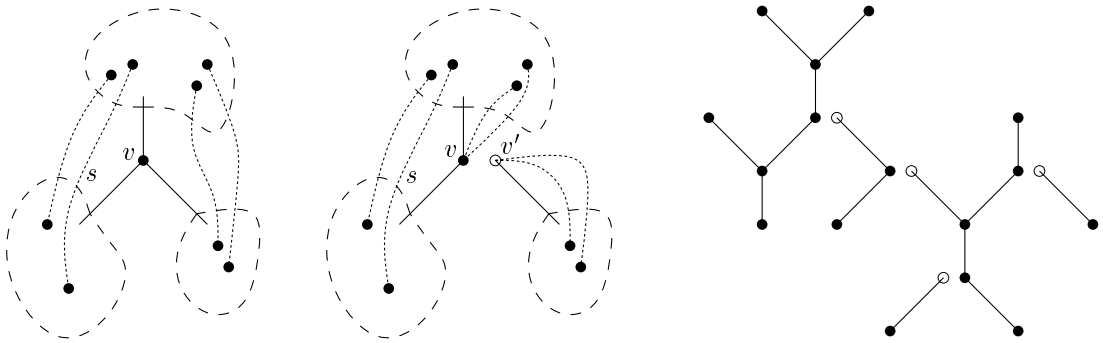

Figure 15: The way in which a branch is cut loose when segment s is the deepest one among the four segments.

as any subtree of it, using the greedy strategy. Proof is the same as above.

Consider the greedy choice strategy, and at any bifurcation, consider the chosen segment that starts at an ancestor and goes deepest in either of the subtrees. For every bifurcation v, imagine cutting loose the branch of $\mathcal{T}$ starting at v that does not contain the deepest choice. Cutting loose a branch means that one of the resulting subtrees gets a copy v' of v as a new source node of the branch, see Figure 15. This new source is called a *fake source*. Any segment that starts at an ancestor of v and leads into the branch that was cut loose is cut into two segments, one from the upper node to v and one from v' to the lower node. After imagining these actions performed for each bifurcation, a contour tree $\mathcal{T}$ with b bifurcations has fallen apart into $b + 1$ subtrees without any bifurcations.

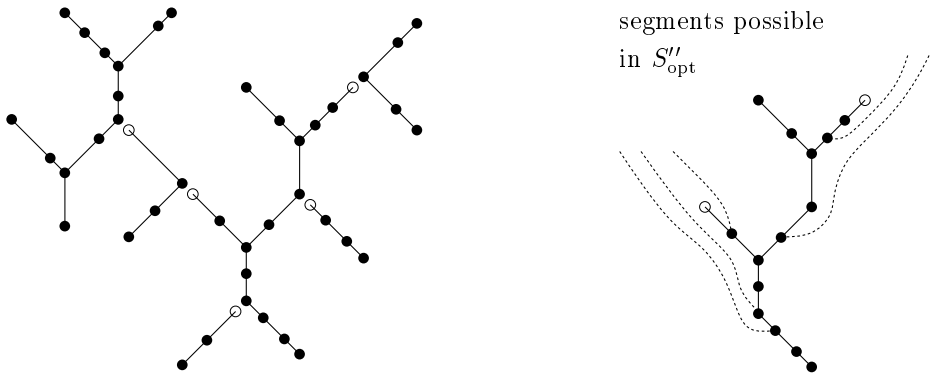

Figure 16: A contour tree with 5 bifurcations broken up into 6 subtrees with 5 fake sources in total. To the right, illustration of the proof that $|S_{\text{separ}}| \leq |S_{\text{opt}}| + b$.

If all subtrees were solved separately by the greedy strategy, we get the optimum solution for the separate subtrees. Let S_{separ} be the set of segments that would be chosen this way, from the set of all segments that appear in the subtrees. Let S_{greedy} be the segments chosen by the greedy algorithm described before, and let S_{opt} be a smallest solution. We'll show that $|S_{\text{greedy}}| \leq |S_{\text{separ}}|$ and $|S_{\text{separ}}| \leq |S_{\text{opt}}| + b$, where b is the number of bifurcations of $\mathcal{T}$ (or,

similarly, the number of fake sources). We'll count any segment at its upper node.

To prove that $|S_{\text{separ}}| \leq |S_{\text{opt}}| + b$, consider one subtree $\mathcal{T}'$ with f fake sources in isolation. Let $S'_{\text{separ}} \subseteq \S_{\text{separ}}$ be an optimal, greedy solution for $\mathcal{T}'$. Let $S'_{\text{opt}} \subseteq S_{\text{opt}}$ be segments that start at a node of $\mathcal{T}'$, and let $S''_{\text{opt}} \subseteq S_{\text{opt}}$ be the segments that cover parts of $\mathcal{T}'$ but start at an ancestor of a fake node of $\mathcal{T}'$. The optimal solution may contain many segments in S''_{opt}, but all arcs of $\mathcal{T}'$ covered by them are also covered by at most f segments in S'_{separ}, namely, for each fake source the deepest going segment from it (see Figure 16). Since segments were cut when this subtree was cut loose, all segments in S''_{opt} can also be chosen—as segments starting from fake sources—for S'_{separ} The above holds for each subtree, so we get $|S_{\text{separ}}| \leq |S_{\text{opt}}| + b$ for the whole tree $\mathcal{T}$.

To prove that $|S_{\text{greedy}}| \leq |S_{\text{separ}}|$, again consider subtree $\mathcal{T}'$, and define $S'_{\text{greedy}} \subseteq S_{\text{greedy}}$ as the segments that start start in $\mathcal{T}'$, and let $S''_{\text{greedy}} \subseteq S_{\text{greedy}}$ be the segments that don't start in $\mathcal{T}'$ but cover some arcs of $\mathcal{T}'$. We argue that $S'_{\text{greedy}} \leq S'_{\text{separ}}$. The segments from S''_{greedy} can only make S'_{greedy} smaller, so we can assume that S''_{greedy} is empty. The greedy algorithm may have chosen segments for S_{greedy} that are greedy for $\mathcal{T}$ but not for $\mathcal{T}'$. This can occur when the greedy choice in $\mathcal{T}$ goes outside $\mathcal{T}'$ at a bifurcation of $\mathcal{T}$ that has become a normal node in $\mathcal{T}'$. However, by the choice which branch was cut loose, we know that S_{greedy} must also contain a segment that is the greedy choice in $\mathcal{T}'$. □

Theorem 4 *Let M be a 2D mesh with n cells representing a real function. A seed set of size at most twice the optimum can be determined in $O(n \log^2 n)$ time and linear storage. For a mesh in d-space, the running time is $O(n^2)$.*

Proof: Note that a seed set must have at least as many seeds as there are sinks. Also note that the number of bifurcations is exactly one smaller than the number of sinks. So the approximation factor of two follows immediately from the lemma above. □

5 Test results

In this section we present empirical results for generation of seed sets within bounds of optimality. Given in Table 5 results are collected from seven data sets from various domains, both 2D and 3D. The data used for testing include:

- Heart: a 2D regular grid of MRI data from a human chest;
- Function: a smooth synthetic function sampled over a 2D domain;
- Bullet: a 3D regular grid from a structural dynamics simulation;
- HIPIP: a 3D regular grid of the wave function for the high potential iron protein;
- LAMP: a 3D regular grid of pressure from a climate simulation;
- LAMP 2d: a 2D slice of the 3D data which has been coarsened by an adaptive triangulation method;
- Terrain: a 2D triangle mesh of a height field.

Data	total cells	# seeds	storage	time (s)	# seeds by method of [4]	storage req of [4]	time (s)
Structured data sets							
Heart	256x256	5631	30651	32.68	12214	255	0.87
Function	64x64	80	664	1.23	230	63	0.15
Bullet	21x21x51	8	964	2.74	47	1000	0.30
HIPIP	64x64x64	529	8729	121.58	2212	3969	3.24
LAMP 3d	35x40x15	172	9267	6.82	576	1360	0.33
Simplicial data sets							
LAMP 2d	2720	73	473	0.69	n/a	n/a	n/a
Terrain	95911	188	2078	13.67	n/a	n/a	n/a

Table 1: Test results and comparison with previous techniques.

Presented are the total number of cells in the mesh, in addition to seed extraction statistics and comparisons to previously known efficient seed set generation methods. The methods presented here, shown to be within a factor of 2 of optimal, represent an improvement of 2 to 6 times over the method of [4], which had no claim on the seed set size. the presented storage statistics account only for the number of items, and not the size of each storage item (a constant). Note that the bounded seed set method presented here has, in general, greater storage demands, though storage remains sublinear in practice. Such tradeoffs are considered acceptable for the benefit of small seed sets.

6 Further research

This paper presented the first methods to obtain seed sets for contour retrieval that are provably small in size. We gave an $O(n^2 \log n)$ time algorithm to determine the smallest seed set, and we also gave a factor We gave a factor two approximation algorithm that takes $O(n \log^2 n)$ time for functions over 2D and $O(n^2)$ time for functions over higher-dimensional domains. In typical cases, the worst case quadratic time bound seems too pessimistic. The algorithms make use of new methods to compute the so-called contour tree.

Test results indicate that seed sets resulting from the methods described here improve on previous methods by a significant factor with respect to previous methods. Storage requirements in the seed set computation remain sublinear, as evidenced by the test results.

Our work can be extended in the following directions. Firstly, it may be possible to give worst case subquadratic time algorithms for four and higher-dimensional meshes; the 3D case was solved recently [21]. Secondly, it is important to study what properties an interpolation scheme on the mesh should have to allow for efficient contour tree construction and seed set selection.

Acknowledgements. The authors thank Hans Bodlaender, Günter Rote, and Dirk Siersma for their helpful comments.

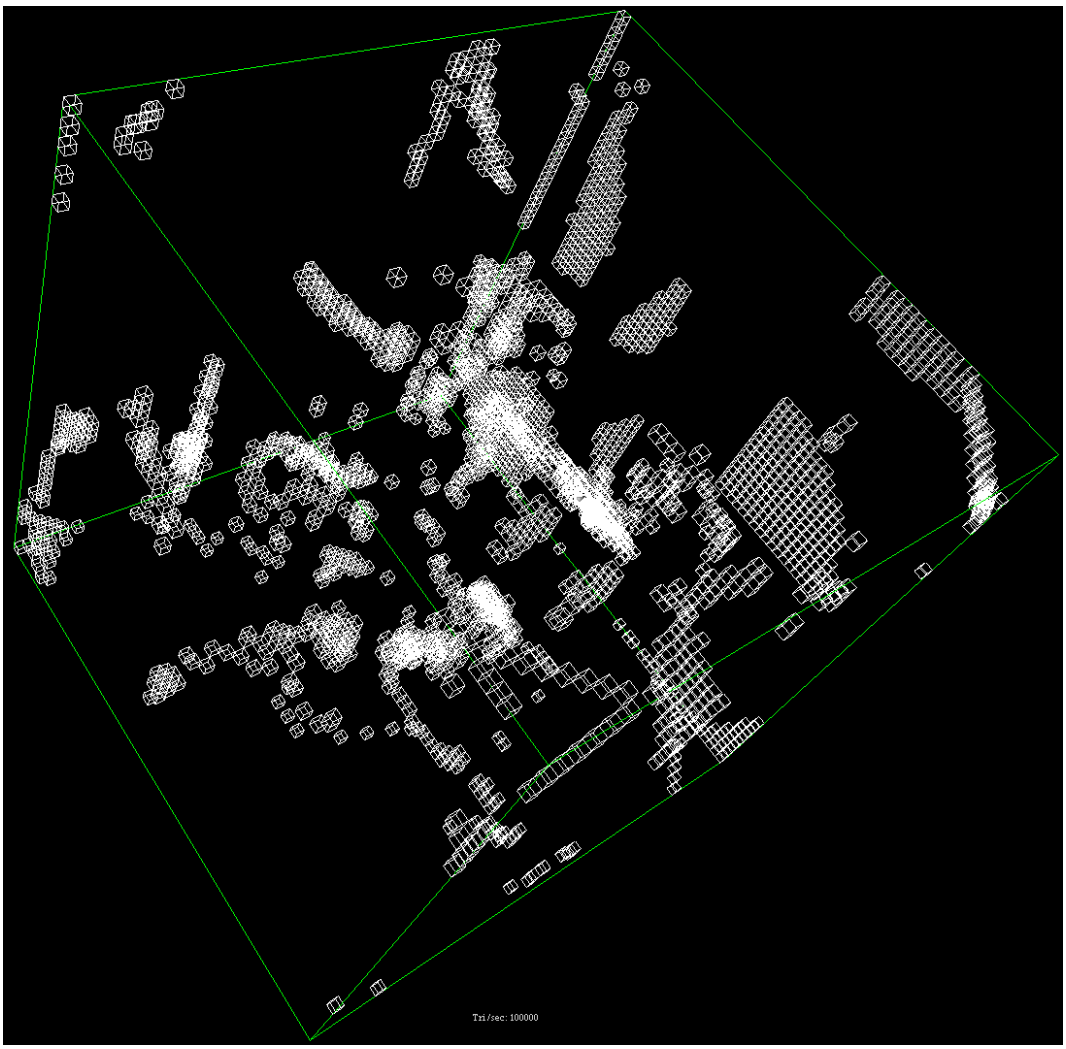

Figure 17: Example of a seed set for a 3-dimensional scalar field (HIPIP).

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] E. Artzy, G. Frieder, and G. T. Herman. The theory, design, implementation, and evaluation of 3-d surface detection algorithms. *Comput. Graph. Image Process.*, 15:1–24, 1981.

[3] C. Bajaj, V. Pascucci, and D. Schikore. Seed sets and search structures for isocontouring. *Submitted*, 1998.

[4] C.L. Bajaj, V. Pascucci, and D.R. Schikore. Fast isocontouring for improved interactivity. In *Proc. IEEE Visualization*, 1996.

[5] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Proc. IEEE Volume Visualization*, 1996.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[7] M. de Berg and M. van Kreveld. Trekking in the Alps without freezing or getting tired. *Algorithmica*, 18:306–323, 1997.

[8] H. Freeman and S.P. Morse. On searching a contour map for a given terrain profile. *Journal of the Franklin Institute*, 248:1–25, 1967.

[9] C. Gold and S. Cormack. Spatially ordered networks and topographic reconstructions. In *Proc. 2nd Internat. Sympos. Spatial Data Handling*, pages 74–85, 1986.

[10] M. W. Hirsch. *Differential Topology*. Springer-Verlag, New York, NY, 1976.

[11] C.T. Howie and E.H. Blake. The mesh propagation algorithm for isosurface construction. *Computer Graphics Forum*, 13:65–74, 1994.

[12] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Trans. on Visualization and Computer Graphics*, 1:319–327, 1995.

[13] I.S. Kweon and T. Kanade. Extracting topographic terrain features from elevation maps. *CVGIP: Image Understanding*, 59:171–182, 1994.

[14] Y. Livnat, H.-W. Shen, and C.R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2:73–84, 1996.

[15] John W. Milnor. *Morse Theory*. Princeton University Press, Princeton, NJ, 1963.

[16] G. Reeb. Sur les points singuliers d'une forme de Pfaff completement integrable ou d'une fonction numerique. *Comptes Rendus Acad. Sciences Paris*, 222:847–849, 1946.

[17] Y. Shinagawa and T.L. Kunii. Constructing a Reeb graph automatically from cross sections. *IEEE Computer Graphics and Applications*, 11:44–51, November 1991.

[18] Y. Shinagawa, T.L. Kunii, and Y.L. Kergosien. Surface coding based on morse theory. *IEEE Computer Graphics and Applications*, 11:66–78, September 1991.

[19] J.K. Sircar and J.A. Cerbrian. Application of image processing techniques to the automated labelling of raster digitized contours. In *Proc. 2nd Int. Symp. on Spatial Data Handling*, pages 171–184, 1986.

[20] S. Takahashi, T. Ikeda, Y. Shinagawa, T.L. Kunii, and M. Ueda. Algorithms for extracting correct critical points and constructing topological graphs from discrete geographical elevation data. *Eurographics '95*, 14:C–181–C–192, 1995.

[21] Sergey P. Tarasov and Michael N. Vyalyi. Contour tree construction in $o(n \log n)$ time. In *Proc. 14th Annu. ACM Symp. Comp. Geometry*, pages 68–75, 1998.

[22] M. van Kreveld. Efficient methods for isoline extraction from a TIN. *Int. J. of GIS*, 10:523–540, 1996.

[23] David F. Watson. *Contouring: A Guide to the Analysis and Display of Spatial Data.* Pergamon, 1992.

[24] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11:201–227, 1992.