Contour Trees and Small Seed Sets for Isosurface Traversal*

Marc van Kreveld† marc@cs.ruu.nl
René van Oostrum† rene@cs.ruu.nl
Chandrajit Bajaj‡ bajaj@cs.purdue.edu
Valerio Pascucci‡ pascucci@cs.purdue.edu
Dan Schikore‡ drs@cs.purdue.edu

Abstract

For 2D or 3D meshes that represent a continuous function to the reals, the contours—or isosurfaces—of a specified value are an important way to visualize it. To find such contours, a seed set can be used for the starting points from which the traversal of the contours can start. 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 polynomial time and storage. Since in practice at most linear storage is allowed, 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 in 2D and $O(n^2)$ time otherwise, and requires linear storage. We also give experimental results, showing the size of the seed sets and supporting the claim that sublinear storage is used.

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 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. In this paper we will consider the interpolation issues only as long as may they affect the isocontour computation. In particular in Section 2 we will state briefly the properties we assume for the interpolating function (satisfied by the most commonly used linear interpolation over simplicial complexes or multi-linear interpolation over regular grids). More on interpolation of spatial data and references to the literature can be found in the book by Watson [24].

One can expect that the 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 [15]. 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 [25].

The second approach is to store the *scalar range*, also called *span*, of all the mesh elements in a search structure. Kd-trees [15], segment trees [3], and interval trees [5, 23] 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. 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. Still, one doesn't want to store a tree with a few hundred million intervals

*The research of the first and second authors was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL).The research of the third, fourth and fifth authors was partially supported by AFOSR grant F-49620-94-1-0080, NSF grant CCR 92-22467 and ONR grant N00014-94-1-0370

Computational Geometry 9^th^ Nice France

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. There are a few papers that take this approach [3, 13, 23]. The tracing algorithms to extract a contour have been developed before, and they require time linear in the size of the output [2, 12, 13].

The objective of this paper is to present new methods for seed set computation. Of a seed set, we require that any possible connected component of any contour in the mesh pass through at least one seed. Otherwise we could miss a (portion of 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 [9, 10, 14, 21, 22]. Another name in use is the *topographic change tree*, and it is related to the *Reeb graph* used in Morse Theory [18, 19, 20, 22]. It can be computed in $O(n \log n)$ time for piecewise linear functions over 2D [6].

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 [6], 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 the contour tree is the appropriate structure to use when selecting seed sets. We give a polynomial time and storage algorithm for minimum size seed sets by using min-cost flow in a DAG [1].
- In practice one can use at most linear storage when computing seed sets. We give a simple approximation algorithm that requires $O(n \log^2 n)$ time and linear storage, and gives at most twice as many seeds as the minimum size seed set. In d-space, the algorithm takes $O(n^2)$ time.
- The approximation algorithm has been implementated, and we supply test results of various kind.

Previous methods to find small size seed sets didn't give any guarantee on their size [3, 13, 23].

2 Preliminaries on scalar functions and the contour tree

On a 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 specific 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 [11, 16]. They can be described by a structure called the contour tree, which we describe shortly.

For example, consider 2D triangular meshes with linear interpolation and note how the contour tree relates to such meshes. For simplicity, we assume that all vertices have a distinct function value. If we draw the contours of all critical vertices of the mesh, then we get a subdivision of the 2D domain into regions (see Figure 1). Since all saddle points must be vertices in our setting, one can show that every region between contours is bounded by exactly two contours. We let every contour in this subdivision correspond to a

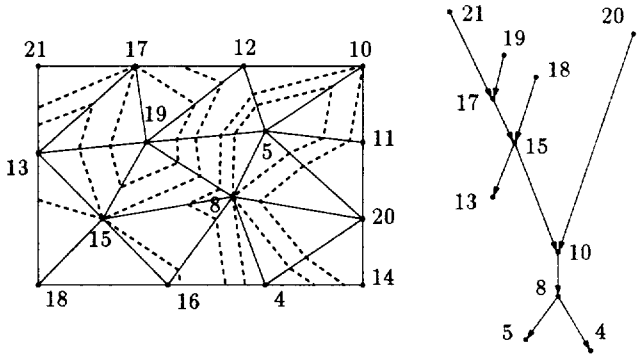


Figure 1: 2D triangular mesh with the contours of the saddles, and the contour tree.

node in a graph, and two nodes are connected (from max to min) if there is a region bounded by their corresponding contours. This graph is a tree, which is easy to show [6, 23], 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 complexity of all contours through saddles may be quadratic [6]. An $O(n \log n)$ time divide-and-conquer algorithm exists, however [6].

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.

We define the contour tree $\mathcal{T}$ as follows.

- Take each maximal connected contour component which contains a criticality.
- These components correspond to the *supernodes* of $\mathcal{T}$ (the tree will be augmented 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 contour components, we add a superarc between the corresponding supernodes in $\mathcal{T}$, oriented from the the higher to the lower function value.

The contour tree is well defined, because each region is bounded by two and only two contour components 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 [18, 19, 20, 22]. Furthermore, one can show that the contour tree is indeed a tree.

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.

Since the type of the mesh and the function used to interpolate the associated discrete data may have some impact on the the seed cells selection and contour tree computation we define the weakest conditions required to apply the present approach. Note that such conditions are satisfied by the most common simplicial decompositions with linear interpolant and regular grids with multi-linear interpolant.

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.

We assume that in facets and edges of 2D meshes, the items 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 3D, a saddle point p is a point such that for any sufficiently small ϵ-sphere around p, the contour of p's value intersects the ϵ-sphere in at least two components. Possible criticalities 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.

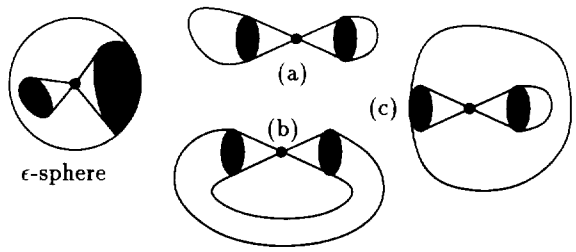


Figure 2: Citicalities in 3D.

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 a variation that includes nodes for all the vertices of M, also the non-critical ones. So supernodes correspond to critical vertices and normal nodes correspond to other vertices. Superarcs are now sequences of arcs, and connect two supernodes. It is easy to determine the contour tree with only the supernodes using the same technique. From now on, we call the contour tree with nodes for all vertices the contour tree $\mathcal{T}$. We'll need this augmented tree for seed selection in the next section.

The critical nodes of $\mathcal{T}$ that have in-degree 1 and out-degree greater than 1 are called *bifurcations*, and the nodes with in-degree greater than 1 and out-degree 1 are called *junctions*. 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, but it facilitates the following descriptions considerably. Basically, other critical nodes can be seen as clusters of critical nodes of degree 3. For example, a node 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.

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 down a horizontal plane. The sweep stops at certain event points: the vertices of the mesh. During the sweep, we keep track of the contour components in the mesh at the value of the sweep function, and the set of cells of the mesh that cross these components. 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. Each such part of $\mathcal{T}$ that is still growing corresponds to a unique contour component at the current sweep value. We group the cells into contour components 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: 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 supernodenode of $\mathcal{T}$ is created, and the arc corresponding to the disappearing component 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 stop being active, while others incident to the vertex start being active. Their pointers 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 can be implemented with a standard heap structure, such that 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 are a linear number of saddles for which the contour through them has linear complexity. We can also state that the running time is $O(n \log n + \sum_{i=1}^{m} |C_i|)$, where the m saddles lie on contours $C_1, \ldots, C_m$ with complexities $|C_1|, \ldots, |C_m|$. Also, besides the mesh (input) and the contour tree (output), the additional storage required can be made linear in the maximum number of active cells and the number of local maxima. So this is $O([\text{no. maxima}] + \max_{1 \le i \le m} |C_i|)$ additional storage. We can avoid the pointers from the mesh cells to the superarcs by copying the active part of the mesh into a separate, temporary structure. Then these pointers don't form a permanent storage overhead.

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. The main idea is to implement a merge in time linear in the size of the *smaller* of the two contours, and similarly, to implement a split in time linear in the size of the *smaller resulting contour*. We also show how to maintain the pointers with the active cells efficiently.

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 $\mathcal{T}$. Maintaining pointers now comes down to changing names. 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 them to a single name,

- *Split*: given one contour about to split, split it into two separate contours by renaming one subset of the contour cells 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. Lets' say that contour 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 size of the smaller contour, we have traversed the whole smaller contour. This technique is sometimes called a *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 largest 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.*

Sketch of the proof for the claimed time bound:

- 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 on $O(1)$ 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 $O(\min(|C_i|, |C_j|))$, 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 saddle node 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; we simply assume that a segment starts at some source node that is an ancestor of the original start node, and ends at a sink that is a descendant of the original end node. It is easy to see that the number passing segments at all saddles cannot decrease 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 from a bifurcation to a junction is illustrated in Figure 4; the integers a, b, c represent the number of segments passing over the superarcs shown. 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 $O(\min(a, b + c) + \min(a + b, c))$,

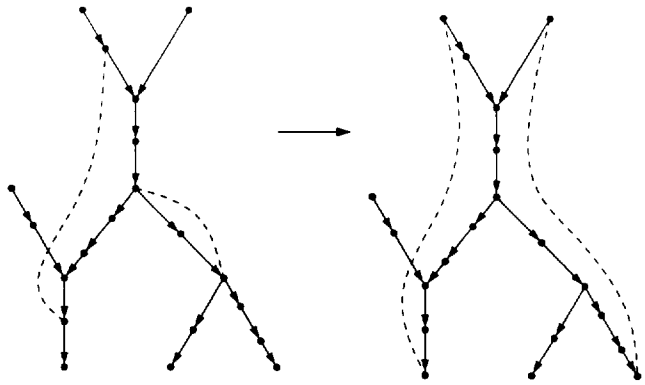


Figure 3: Stretching two segments (dashed) in $\mathcal{T}$.

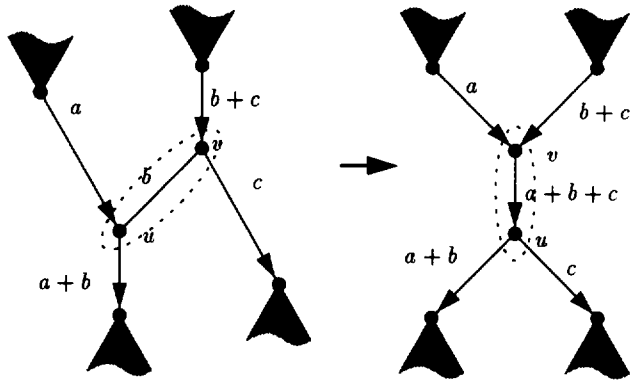


Figure 4: Swapping a superarc.

which is at least as much. No segment ends, because all of them were stretched.

It can easily be verified that by repeatedly swapping superarcs, $\mathcal{T}'$ can be derived such that no junction in $\mathcal{T}'$ has a bifurcation as an ancestor.

Now, every segment can pass $O(n)$ junctions and bifurcations, but no segment can be more than $O(\log n)$ times in the smallest set. Summing this over the $O(n)$ segments, this results in a total of $O(n \log n)$ time for all renaming of the cells.

4 Seed set selection

A seed set is a subset of the cells (triangles or tetrahedra) the mesh. 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. This range is simply a one-dimensional interval, and is also called the *span*. For each cell, its span can be located in the contour tree, where it is represented by two nodes. We add the span to the contour tree as a directed arc, which we call a *segment* to distinguish it from the arcs of the contour tree. Let $\mathcal{T}$ denote the contour tree, and let $\mathcal{G}$ denote the DAG that is the contour tree extended with the segments of all mesh elements. Observe that each segment is a shortcut in $\mathcal{T}$ for one or more arcs on a directed

path. We say that the segment *passes*, or *covers*, these arcs of $\mathcal{T}$ (see Figure 4.1(a)). The small seed set problem now is the following graph problem: find a small subset of the segments such that each arc of $\mathcal{T}$ is passed by some segment of the subset.

In this section we give two methods to obtain complete seed sets. The first gives a seed set of minimum size, but it requires polynomial time and storage. The second method requires $O(n \log^2 n)$ time and linear storage in 2D, and gives a seed set at most twice the size of the minimum. In d-space, this approximation algorithm takes $O(n^2)$ time and linear storage.

4.1 Minimum seed sets in polynomial time and storage

We define a bipartite graph $\mathcal{B} = (U \cup V, A)$ as follows. The set U of nodes corresponds to the set of segments of the mesh cells, and the set V corresponds to the set of arcs of $\mathcal{T}$. An arc $(u, v) \in A$ if the segment corresponding to $u \in U$ passes the arc corresponding to $v \in V$ ((see Figure 4.1(b))). A complete seed set corresponds to a subset of U that dominates all nodes in V, that is, each node in V should have a neighbor in the chosen subset. The smallest cardinality subset $U' \subseteq U$ corresponds to a minimum seed set. The graph $\mathcal{B}$ can have a number of arcs quadratic in the size of $\mathcal{G}$.

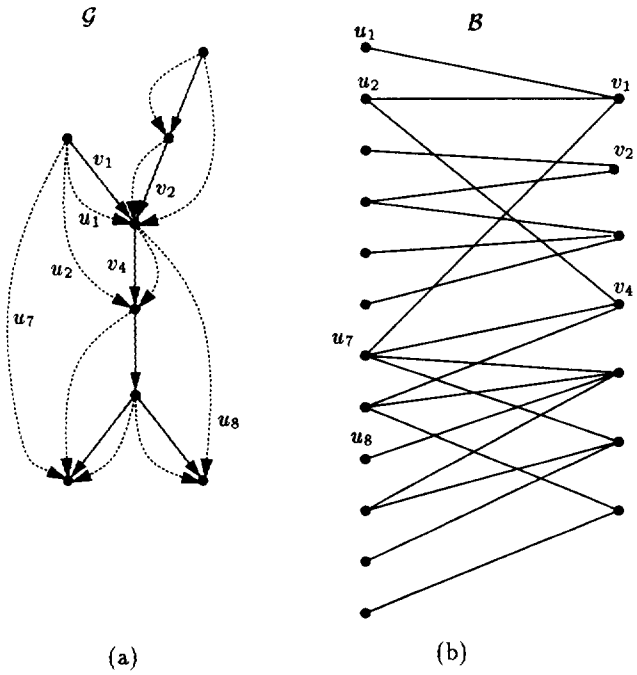


Figure 5: (a) A DAG $\mathcal{G}$ with the segments shown dashed. (b) The bipartite graph $\mathcal{B}$ assiciated with $\mathcal{G}$.

Observe that $\mathcal{B}$ is strongly chordal: every cycle of even length exceeding 5 has an odd chord. This is for the following reason. Since the nodes in U represent segments in $\mathcal{G}$, and three segments in $\mathcal{G}$ cannot have pairwise overlap of an arc in $\mathcal{T}$ (or $\mathcal{G}$) without having a triple overlap as well, every cycle of even length exceeding 5 has a node in U and in V that are connected but aren't in the cycle (a chord). We can augment $\mathcal{B}$ by turning U into a clique; $\mathcal{B}$ remains strongly chordal. A minimum dominating clique on strongly chordal graphs can be computed in linear time, because the same result holds for the superclass of dually chordal graphs [4, 8]. This gives a minimum seed set.

Theorem 3 *Given a contour tree $\mathcal{T}$ for a continuous function defined by a mesh with n cells, an optimum size seed set can be computed in polynomial time and storage.*

4.2 Approximation of small seed sets in linear storage

The excessive time and storage requirements for optimal seed sets makes it nearly useless in practical applications. We therefore developed an approximation algorithm to compute a seed set using linear storage and $O(n \log^2 n)$ time in the 2D case. It yields a seed set of size no more than twice the size of the smallest seed set. In higher dimensions, the running time is $O(n^2)$.

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 shown before. We add the segments to form the graph $\mathcal{G}$. Then we sweep again, now in the mesh and in the graph $\mathcal{G}$ simultaneously. During the sweep, greedy choices are made in the set of segments in $\mathcal{G}$; these segments correspond to cells that will be seeds. The greedily chosen segments are stored in a data structure $\mathcal{D}$ that allows for insertions of newly chosen segments and for queries. A query specifies a node v of $\mathcal{T}$, and asks whether some chosen segment in $\mathcal{D}$ ends at v, and whether v is passed by some chosen segment in $\mathcal{D}$. The segments are grouped by superarc they pass as usual, corresponding to cells intersected by the same contour. To make the greedy choice at any superarc, we store all active segments at a superarc sorted by function value of the lower node of $\mathcal{G}$ in a binary search tree. The following events can occur:

- Source: Choose the segment leaving it with the lowest value at the other end node, the greedy choice. Initialize a set of active segments for this superarc of $\mathcal{T}$.
- Normal node: Update the currently active segments.

  Query with the node in $\mathcal{D}$ to decide if any chosen segment passes it. If not, it must be the end node of a chosen segment. Choose a new segment greedily and add it to $\mathcal{D}$.
- Sink: Remove the group of active segments.
- Junction: Update the active segments, merge the two groups of active segments into one, and merge the corresponding two binary search trees.

  Query with the node as for a normal node.
- Bifurcation: Split the group of active segments into two, and update them.

  For each highest node v below the bifurcation, test by querying in $\mathcal{D}$ if any chosen segment ends at it or passes it. If neither is true, choose a segment active at the bifurcation and into this superarc greedily. Add the chosen segments (zero or more) to $\mathcal{D}$.

Lemma 1 *The greedy strategy has an approximation factor of 2.*

We give a sketch of the proof. Note that if $\mathcal{T}$ only contains junctions, then a greedy strategy is optimal. At a bifurcation, some segment leading into one superarc may be the chosen one, whereas the greedy choice for the other superarc would have been optimal. We can bound the number of segments chosen by the greedy strategy from above by assuming that for *both* superarcs the greedy choice was taken. Then we continue on both parts of the tree below these superarcs in the same way. We can bound the minimum number from below by assuming that only *one* segment was taken, and we continue the argument on the same two subtrees as in the greedy algorithm. So the subtrees are the same, and the greedy method chooses at most twice minimum.

We treat junctions and bifurcations efficiently as before, by tandem search in the contours on the mesh. We can traverse in time linear in the size of the smaller group to decide how to merge or split the segments efficiently. A merge or split of components C_i and C_j takes $O(\min(|C_i|, |C_j|) \log n)$ time, because it involves merging two binary search trees, or splitting one. So we need $O(n \log^2 n)$ time for manipulating the binary search trees.

To define the data structure $\mathcal{D}$, we first need a transformation of $\mathcal{T}$ and its segments. 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

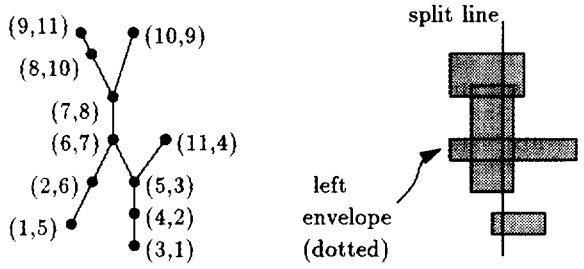


Figure 6: Left, the numbering of $\mathcal{T}$. Right, the left envelope of the rectangles intersecting a split line.

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 6). These numbers can be seen as coordinates in the plane. Any segment from a start node u to an end node v in $\mathcal{T}$ transforms to a rectangle by using the numbers as coordinates. The lower left corner of the rectangle has the coordinates of v, and the upper right corner has the coordinates of u. The greedily chosen segments are stored as such rectangles in the data structure $\mathcal{D}$. A query with a node of $\mathcal{T}$ asks if the point is contained in some rectangle; this corresponds to some chosen segment passing the node. $\mathcal{D}$ is an interval tree with associated structures [17]. The main tree is defined on the intervals of the first coordinate. Each node stores a vertical split line and two segment trees, one for query points to the left of the split line and one for query points to the right. We store the rectangles that intersect the vertical split line in the left segment tree by storing the left envelope of the rectangles only (see Figure 6). This segment tree requires only linear storage, and queries and insertions take logarithmic time. For the whole interval tree with associated structures, the storage is still linear, the query time is $O(\log^2 n)$ and insertions take $O(\log n)$ time. The number of queries and insertions is linear in the number of nodes in the contour tree. Therefore, all operations on $\mathcal{D}$ also take $O(n \log^2 n)$ time together. More details are in the full paper.

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 $O(n^2)$.*

5 Test results

In this section we present empirical results for generation of seed sets within bounds of optimality. Given in Table 1 are results collected from six datasets, both 2d and 3d. Presented are the total number of cells in the mesh, in addition to seed extraction statistics and comparisons to previously known efficient approximation methods. The methods presented here, shown to be within a factor of 2 of optimal, represent an improvement of 2 to 20 times over the method of [3], which had no claim on the seed set size. The presented storage statistics account only for the number of stored 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 seed sets within guaranteed bounds of optimality. Sample images from the test function and LAMP datasets are given in the Appendix.

6 Further research

This paper presented the first methods to obtain seed sets for contour retrieval that are provably small in size. We gave a polynomial time and storage algorithm to determine the smallest seed set, and we also 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 3D. 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 improve on previous methods by greater than an order of magnitude in seed set size for some cases. 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 higher-dimensional meshes. 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. We are currently studying these extensions.

Acknowledgements. The authors thank Dieter Kratsch, Hans Bodlaender and Dirk Siersma for their helpful comments.

Sample images from the some of the test data is given in the Appendix. The heart data appears courtesy Tsuyoshi Yamamoto and Hiroyuki Fukuda of Hokkaido University. The LAMP data is courtesy the Space Science and Engineering Center at the University of Wisconsin. The bullet data is courtesy Lawrence Livermore National Laboratory. The terrain data is a triangulated irregular network from USGS DEM data.

Data	total cells	#seeds	storage	time (s)	#seeds by method of [3]	storage req of [3]	time (s)
Heart	65025	5631	30651	32.68	12214	255	0.87
Function	3969	80	664	1.23	230	63	0.15
Bullet	20000	8	964	2.74	47	1000	0.30
LAMP 3d	19040	172	9267	6.82	576	1360	0.33
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: Seed cell statistics for regular (top) and irregular (bottom) data

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.L. Bajaj, V. Pascucci, and D.R. Schikore. Fast isocontouring for improved interactivity. In *Proc. 1996 IEEE Symposium on Volume Visualization*, pages 39-46, San Francisco, Oct 7-8, 1996.

[4] A. Brandstädt, V. D. Chepoi, and F. F. Dragan. The algorithmic use of hypertree structure and maximum neighbourhood orderings. In *20th International Workshop "Graph-Theoretic Concepts in Computer Science"(WG'94)*, volume 903 of *Lecture Notes in Computer Science*, pages 65–80. Springer-Verlag, Berlin, 1995.

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

[6] Mark de Berg and Marc van Kreveld. Trekking in the alps without freezing or getting tired. In *1st Annual European Symposium on Algorithms (ESA '93)*, volume 726 of *Lecture Notes in Computer Science*, pages 121–132. Springer-Verlag, 1993.

[7] F. F. Dragan and A. Brandstädt. Dominating cliques in graphs with hypertree structure. In *International Symposium on Theoretical Aspects of Computer Science (STACS'94)*, Lecture Notes in Computer Science number 775 , pages 735–746. Springer-Verlag, 1994.

[8] F. F. Dragan and A. Brandstädt. Dominating cliques in graphs with hypertree structure. In *International Symposium on Theoretical Aspects of Computer Science (STACS'94)*, volume 775 of *Lecture Notes in Computer Science*, pages 735–746, Berlin, 1994. Springer-Verlag.

[9] 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.

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

[11] M.W. Hirsch. *Differential Topology*, volume 33 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 1976.

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

[13] 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.

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

[15] 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.

[16] J. Milnor. *Morse Theory*, volume 51 of *Annals of Mathematics Studies*. Princeton University Press, 1963.

[17] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[18] 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.

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

[20] 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.

[21] 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.

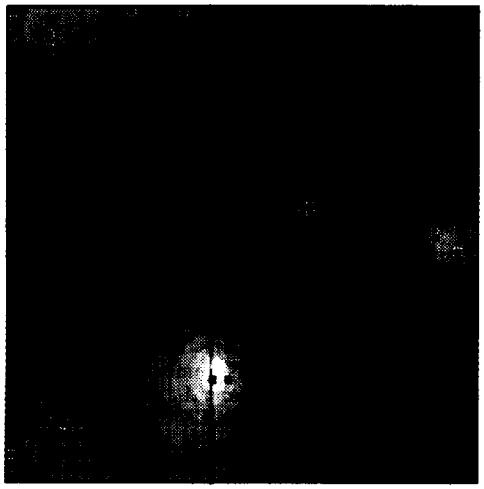
[22] S. Takahashi, T. Ikeda, Y. Shinagawa, T.L. Kunii, and M. Ueda. Algorithms for extracting correct critical points and constructing topological graphs from discrete geographical elevation data. *Eurographics '95*, 14:C–181–C–192, 1995.

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

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

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

A Appendix



(a)

(b)

Figure 7: Seed sets from 2d scalar data
(a) seed set from a synthetic smooth function
(b) seed set for a slice of wind speed data (LAMP)