Efficient and Simple Construction of Contour Trees Using Monotone Paths

Tobias Lenz* Günter Rote*

Submitted to the *19th Annual Symposium on Computational Geometry, 2003*

Abstract

Contour trees are used when high-dimensional data are preprocessed for efficient extraction of isocontours for the purpose of visualization. So far, efficient algorithms for contour trees are based on processing the data in sorted order. We present a new algorithm that avoids sorting of the whole data set, but sorts only the *critical points*, which form only a small fraction of the points, for typical data that arise in practice. The algorithm works in any dimension.

1 Introduction

Visualization by Isosurfaces. In many scientific fields huge amounts of spatial data are produced, either by measurements or by computer simulations. Examples are electromagnetic waves in nuclear spin tomography, heat distribution in fuel cells, or molecular structures in X-ray crystallography. The field of Scientific Visualization is concerned with methods that help the user to extract meaningful information from the data in a visual and intuitive way.

One way to visualize for example planar volumetric data is by displaying iso-lines or iso-surfaces, see [4]. Figure 1a shows iso-lines of a bivariate function $f(x, y)$. For three-dimensional data, an iso-surface is a surface in space, and it is impossible to view many iso-surfaces at different heights simultaneously because they hide each other. Therefore it is important for the user to select the heights interactively in order to get an intuitive understanding of the data. An efficient interactive system must be able to display contours quickly.

Tracing of Contours. When the data are interpolated on a triangulated mesh, it is quite easy to extract a component of an iso-surface, once an initial *seed* point on that component is given. One can simply walk along the surface through the cells using the adjacency of the cells.

In order to find a seed point on each component of an iso-surface at a given height without searching the whole data, the data are preprocessed into a *contour tree.*

Contour Trees. We illustrate this concept with the smooth bivariate function $f(x, y)$ of Figure 1a, which shows a contour map of f consisting of iso-lines at equally spaced height values. The *iso-line* $f_=(h)$ at height h is the set of points (x, y) with $f(x, y) = h$. A *contour* is a connected component of an iso-line. As the height h decreases continuously, contours may join or split, and they may appear (at a local maximum) or disappear (at a local minimum).

*Institut für Informatik, Freie Universität Berlin, Takustraße 9, D-12247 Berlin, Germany,
`tlenz,rote@inf.fu-berlin.de`

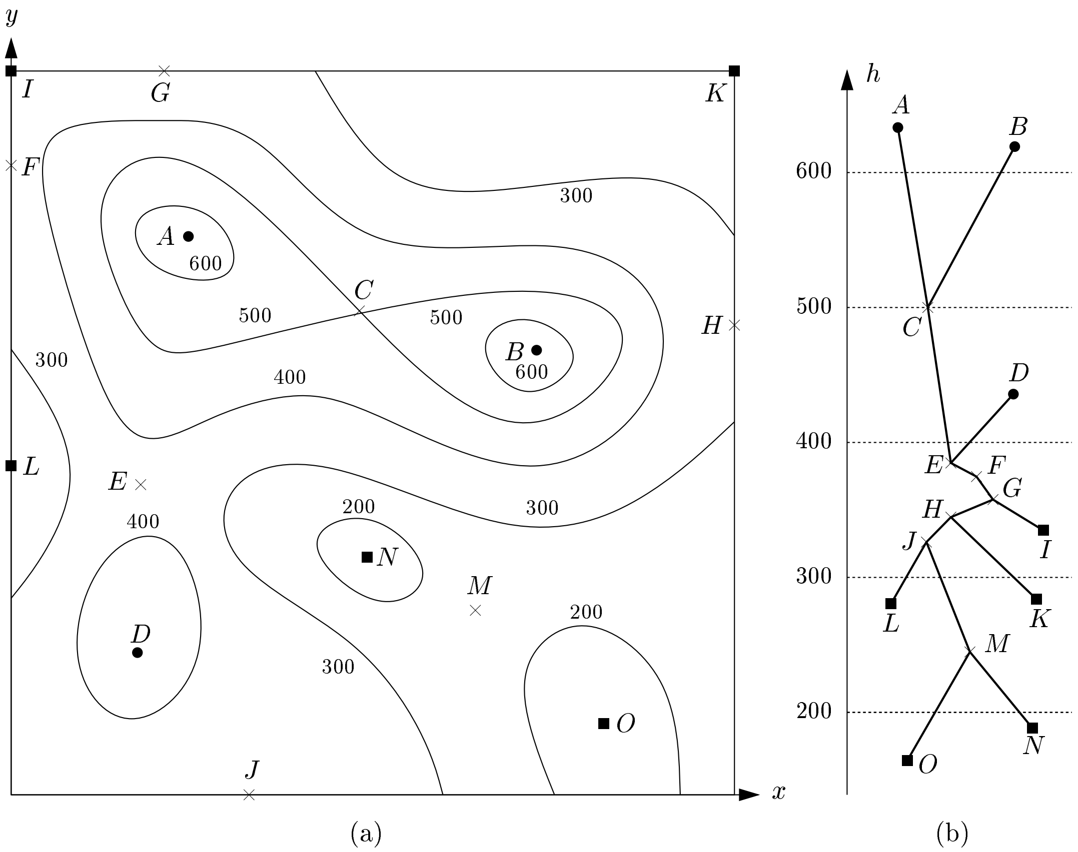


Figure 1: A contour map and the corresponding contour tree. Minima and maxima are indicated by squares and circles, and crosses denote saddle points.

The contour tree is shown in Figure 1b. Each contour is represented by a point. The contours at a given height h are represented by all points at height h in the contour tree. The points where contours join or split are vertices of the contour tree. The vertices of degree one in the tree are the points where contours appear or disappear.

The *critical points* are the local maxima, minima, and the saddle points of the function f. These are the points where the topology of the iso-surface changes. The other points are the *regular points*, where nothing interesting happens. For example, in Figure 1, as h varies between $f(B)$ and $f(C)$, the two circular contours around A and B grow or shrink, but their topology (and in particular, their number) is constant.

To use the contour tree for efficient iso-surface extraction, we must store its edges in an interval tree, which allows to report, for a given value h, all edges whose h-interval contains this value. In addition, for each edge of the tree that is found in this way, we must find one seed edge in the triangulation at which we can start to trace the contour. Our algorithm implicitly constructs such seed edges, although they are not the smallest possible seed sets.

Previous Work. Contour trees [5, 6], were introduced in computational geometry by van Kreveld, van Oostrum, Bajaj, Pascucci, and Schikore [10]. In the topological literature, contour trees are known as Reeb graphs [9], after the American poet John Reeb (1887–1920). When the domain of f is simply connected, the Reeb graph is a tree.

The algorithm of van Kreveld et al. makes a single pass over the data. It sweeps from highest to lowest through the sorted data values and keeps track of the components which are cut by the sweep plane in its current position. It creates a contour tree in $O\left(n^2\right)$ time for n data values in three dimension and in $O\left(n \log n\right)$ for the two-dimensional case.

This algorithm was extended by Pascucci in [7, 8] to obtain information about all topological changes inside the data set by calculating the Betti numbers for each contour.

Carr et al. [1, 2] improved the running time to $O\left(N + n \log n\right)$ for any dimension, where N is the number of simplices. The crucial idea is to construct two trees, the *join tree* and the *split tree*, in two separate sweeps. These two trees are finally merged into the final contour tree, in linear time.

Our Algorithm. We will follow Carr's idea, constructing the join tree and the split tree separately. Our algorithm differs in the way how these trees are constructed. We concentrate on the critical points v, and process them in sorted order, according to the height $f(v)$. We do not waste too much effort by sweeping over regular points. In order to determine the connectivity of the components which meet (locally) at v, we start a monotone path from v into every direction. It is very easy to follow a monotone path, by just searching through the neighbors. We follow each path until we either hit a previously visited vertex or we get stuck at a local extremum.

The main advantage is that only the critical points have to be sorted. The critical points must be found in an initial (unordered) scan of the data. A critical point can be identified by looking at its neighbors. The running time of our algorithm is $O(N + t \log t)$ if there are t critical points. The algorithm is easy to implement and works in any dimension.

Comparison to Previous Algorithms. In Section 7 we will discuss some additional advantages over previous approaches when it comes to huge data sets: Only a small fraction of the data must be copied and processed in the algorithm's private memory, and the greater part of the data is only scanned once in the initialization phase.

Overview. In Section 2 we state the precise problem definition and we establish some basic facts. Section 3 discusses the definition and determination of critical points. Section 4 presents the algorithm and Section 5 its analysis. In Section 6 we report some preliminary computational results. In Section 7 we discuss the results and point out some open questions.

2 Definitions.

We assume that the function f is specified as a piecewise linear interpolated function on some n data points V. We are given a triangulation T of a convex domain $D \subset \mathbb{R}^d$ with vertex set V, and the values $f(v) \in \mathbb{R}$ for $v \in V$. The triangulation T can be given explicitly as a list of N d-simplices, or, when the data points V form a grid ("a structured mesh"), the triangulation can be implicitly given by some fixed triangulation of the grid following some regular pattern. In the first case ("an unstructured mesh"), we assume that each simplex is linked to its $d + 1$ neighboring simplices. For a structured mesh, we will have $N = O(n)$. In the worst case,

$N = O(n^{\lceil d/2 \rceil})$, although we do not think that triangulations with superlinear N will be very useful in this context.

[2] have also done some work in the direction of extending the construction of contour trees to interpolants of higher degree.

The graph of the triangulation (consisting only of the vertices and edges) is denoted by $G = (V, E)$.

Technical Assumption. We assume that all n values $f(v_i)$ are distinct. If we would allow values at adjacent vertices to be equal, the notion of critical points would have to be adapted, and this would cause considerable technical difficulties. Given the intended purpose of contour trees (visualization of measured data), it is appropriate to resolve such cases by perturbation.

In the other hand, we could allow non-adjacent vertices to have equal values without difficulty. In this case, our assumption is merely a convenience.

Level sets and contours. The upper level set, the lower level set, and the (equality) level set of f at height h are defined as

$$\begin{aligned} f_>(h) &:= \{\, x \in D \mid f(x) > h \,\}, \\ f_<(h) &:= \{\, x \in D \mid f(x) < h \,\}, \\ f_=(h) &:= \{\, x \in D \mid f(x) = h \,\}. \end{aligned}$$

A *contour* is a connected component of a level set $f_=(h)$. As we sweep through the data by decreasing h, the set $f_>$ grows while $f_<$ shrinks continuously, and $f_=$ is their common boundary. The *join tree* represents the evolution of the components of the set $f_>(h)$ as h varies. The split tree is defined similarly. (The names join tree and split tree refer to the arbitrary convention of sweeping from higher to lower values.)

Let us look at the components of $f_>(h)$ as h decreases from higher to lower values. Since $f_>$ grows during this process, the following events can happen.

(a) A new component may appear, starting out at a local maximum.

(b) Several components may merge into one.

(c) In addition, there may be topological changes which don't change the topology, like a component touching itself and forming a ring.

Also, events of types (b) and (c) may occur together.

We will construct the join tree by performing this sweep from top to bottom. The split tree is constructed analogously for the lower level sets $f_<$, by sweeping from bottom to top. As $f_=$ is the boundary of $f_>$, it is not hard to see that every join of two components of $f_=$ corresponds to a join for $f_>$ and is therefore reflected in the join tree, and vice versa. A similar statement holds for splits of $f_=$ and the split tree, and thus the join tree and the split tree together hold all information that is necessary to construct the contour tree.

Theorem 1 ([1, 2]) *The contour tree can be constructed in linear time from the join and the split tree.* □

We need a little lemma about the connection between connected components of level sets (in the topological sense) and connected components in the graph. Let $G_>(h)$ be the subgraph of G induced by the vertices v with $f(v) > h$.

Lemma 1 *The number of connected components of $f_>(h)$ is equal to the number of connected components of $G_>(h)$ (in the graph-theoretic sense).*

Proof. A simplex may only (partially) belong to one connected component of $f_>(h)$, and if it does, then a vertex must belong to $f_>(h)$. On the other hand, all vertices v of a simplex with $f(v) > h$ lie in the same connected component of $G_>(h)$. The lemma follows from these observations. □

3 Regular and Critical points.

We define a *regular point* to be a point v where we can conclude, from looking only at its neighborhood, that the number of components of the level sets don't change as h passes through v. All other points are *critical points.*

We will now turn this into a workable criterion. The *neighborhood* $N(v)$ of a vertex v is the graph obtained by taking all vertices and edges of the simplices containing v, and removing v.[1] $N_+(v)$ and $N_-(v)$ are the subgraphs of $N(v)$ induced by the vertices w with $f(w) > f(v)$ and $f(w) < f(v)$ respectively, and $C_+(v)$ and $C_-(v)$ denotes the number of connected components in these graphs.

We note that the maxima and minima can be recognized easily from $C_+(v)$ and $C_-(v)$: A local maximum is characterized by $C_+ = 0$ (and hence $C_- = 1$ since $N(v)$ is connected), and similarly for a minimum.

A *regular point* is now defined as a point v with $C_+(v) = C_-(v) = 1$.[2] A *join candidate* is a point with $C_+(v) > 1$, and a *split candidate* is a point with $C_-(v) > 1$.

Lemma 2 *If h passes a regular point v, the number of connected components of $f_>(h)$ and $f_<(h)$ is unchanged.*

Proof. We show that a point at which the number of components of $f_>(h)$ changes must be a maximum or a join candidate. The lemma follows from this and the analogous statement for split candidates.

If $f_>(h)$ changes at v and v is not a maximum, then it must be a join. Thus, by Lemma 1, two components of $G_>(h)$ must become connected when the vertex v is added to the graph. This is only possible when $C_+(v) > 1$. □

By this lemma, we can easily identify a critical point by searching its neighborhood graph in linear time. In total, every simplex contributes $\binom{d}{2}$ edges each to the $d+1$ neighborhood graphs of its vertices. Thus we have:

Lemma 3 *The critical points can be identified in $O(N)$ time by scanning through the triangulation.* □

4 The New Algorithm

We describe the construction of the join tree. We scan all join candidates v in decreasing order of $f(v)$. For each join candidate v, we select a neighbor w in each of the $C_+(v)$ components of $N_+(v)$. These neighbors are processed sequentially. We process w by starting a monotone increasing path at w, continuing until we get stuck in a local maximum or we hit a previously visited vertex.

Let H denote the graph of all traversed edges in all monotone paths up to now. H includes

[1] Contrary to the usual graph-theoretic definition, the neighborhood $N(v)$ is *not* the subgraph induced by the neighbors of v. There might be an edge between two neighbors of v which does not belong to a common simplex with v, in the triangulation. However, Lemma 2 would remain true even if we used the alternative definition.

[2] In topology, where it is customary to investigate smooth functions f, a point v is defined to be critical if the topology of the level sets changes as h passes through v, and it is a regular point otherwise. However, this definition would be too strong for us: We are not interested in the complete topology of the level sets, we are interested only in the number of components. Moreover, we consider piecewise linear functions, and the situation at a vertex may be much too complicated to determine the topology of the level sets in the neighborhood of v, when the dimension is high.

the edges from all already processed candidates v' to their selected neighbors w', but it does not include the current vertex v. We call a component of $f_>(h)$ *trivial* if it contains only a single critical point (which must be a local maximum).

Lemma 4 *The graph H has precisely one connected component in every component of $f_>(f(v))$, except for trivial components of $f_>(f(v))$ which do not have v on their boundary.*

Proof. This is easy by induction. Assume that the lemma holds for the current vertex v. If we change h from $f(v)$ to $f(v) - \varepsilon$, we have to connect all components in the neighborhood of v. This is achieved by adding the edges from v to its neighbors w to H. If we further decrease h, the lemma continues to hold: Any local maximum that we pass will only generate a trivial component. So, the lemma still holds for the next candidate $\hat{v}$ *before* the paths from its neighbors $\hat{w}$ are added to H. However, each path $\hat{w}$ starts in one component of $f_>(f(\hat{v}))$, and since we only walk upwards, the path cannot leave that component. So it will either connect to the corresponding component of H, or it will end in a local maximum. In the latter case, we have discovered a new trivial component, which is just about to stop being trivial since it touches $\hat{v}$. On the other hand, it is clear that every trivial component which touches $\hat{v}$ is discovered in this way. $\square$

Note that we don't process maxima and minima explicitly. Maxima are found automatically when walking from the nearest saddle point. Minima are identified in the split tree.

We have to describe how we manage the connected components of H and how we build up the join tree. The natural representation for maintaining connected components under insertions is of course a UNION-FIND data structure. Our ground set is the set of join candidates. We choose as a representative of a component of H the lowest join candidate (that was added last) in that component. This critical point is also the lowest point in the component of the join tree that has been constructed so far, in which all descending paths in that component come together.

As we walk along a monotone path from a neighbor w of v, we mark the visited vertices and give them a pointer to v. (The vertex v will become the lowest vertex in their component.) The vertices on the path do not become part of the UNION-FIND structure; their pointers are never changed.

Now, if a monotone path hits a vertex z which was already visited, we first follow the pointer from z to its candidate vertex (unless z happens to be a candidate vertex itself). Then we query this candidate vertex in the UNION-FIND structure and FIND its representative r. If it turns out that z is already in the same component as v ($r = v$) we don't have to do anything else. Otherwise, we add an edge from v to r into the join tree, we perform the UNION of the components containing v and r, and we select v to be the representative of the resulting set.

The case when a monotone path ends in a local maximum r is easy: We simply add an edge from v to a new vertex representing r into the join tree.

When we add a new edge to the join tree, we can also remember the initial portion of the path, from v up to the height of r, as a possible seed set for this edge. This is not necessarily a very good, i.e., small seed set, but it has the nice property that the h-intervals are disjoint. The selection of optimal or approximately optimal seed sets was discussed in [10]. However, it is more in line with our minimalistic approach to be satisfied with reasonable seed sets that are given by simple heuristics, instead of striving for the optimum. (Later, when the join tree is combined with the split tree, we have the option of choosing the better of two possible seed sets.)

It may happen that a join candidate receives only one outgoing edge in the join tree. For example, in two dimensions, a saddle point (a point with $C_+ > 1$ and $C_- > 1$) will be a join candidate and a split candidate, but it will in general be either a join vertex or a split vertex but not both. In three and higher dimensions, a saddle point might just reflect a change in topology and not in connectivity (like a change from a torus to a sphere), and become neither a join nor a split vertex.

We leave these vertices of degree two in the tree during the construction, and eliminate them in a final purification step before combining the join and split tree to the contour tree.[3] (The point F in Figure 1 is a critical point which is neither in the join tree nor in the split tree, and hence it should not belong to the contour tree proper. It is included in the figure only to clarify this point.)

5 Analysis

If there are t critical points, and a total of $s = \sum^t C_+(v)$ of their neighbors are processed, the algorithm performs at most t UNION and at most s FIND operations to construct the join tree. The dominating step is usually the walking of the paths, because this is repeated for each of the s neighbors, and the paths may be very long. But in the worst case, this is bounded by the size of the graph $|E|$, which is $O(N)$ and thus dominated by the initialization time. All other operations are linear in t and s. Including the time for sorting the t critical points, the time is

$$O(N + t\log t + s\alpha(s,t)) = O(N + t\log t),$$

[3]The join tree that we construct deviates slightly from our definition because it stops at the lowest critical point and it misses the edge from there to the global minimum. Since the global minimum is found in the split tree, this is easily taken care of.

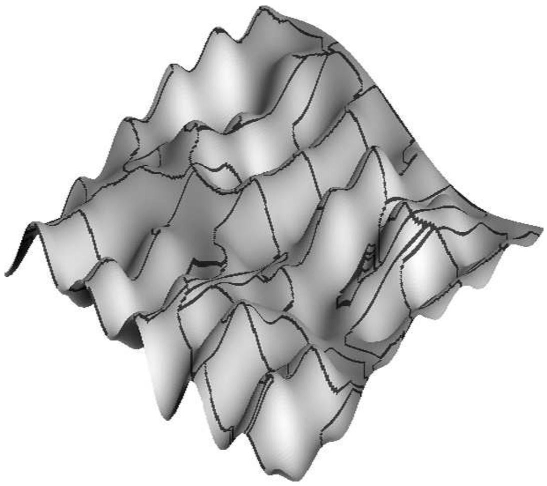


Figure 2: Only few points are visited.

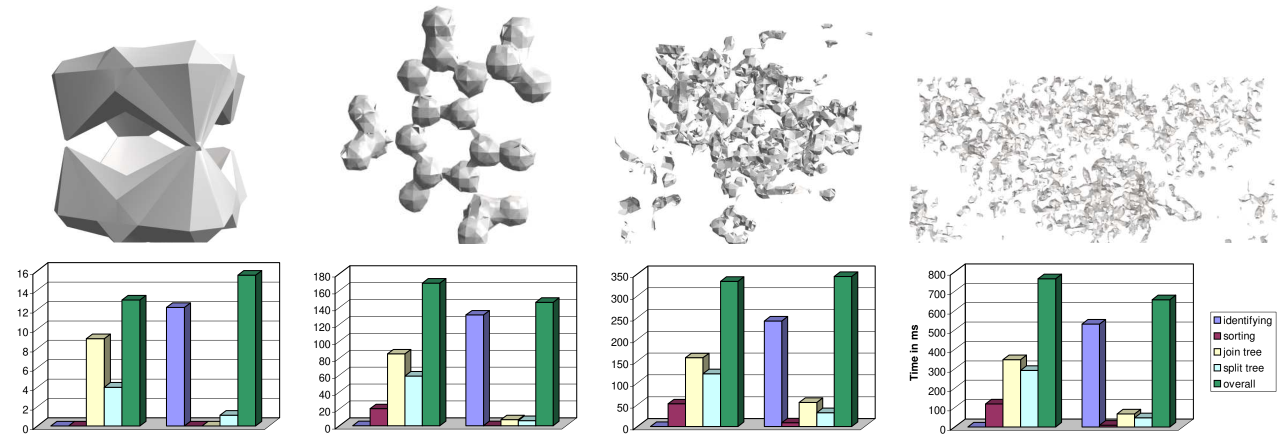
where $\alpha(s,t)$ is the extremely slowly growing inverse of the Ackermann function. The term $s\alpha(s,t))$ is dominated by the first two terms: For $s > t\log\log t$, we have $\alpha(s,t) \leq 1$, and $s\alpha(s,t)) = O(s) = O(N)$. For $s \leq t\log\log t$, we have $s\alpha(s,t) \leq t\log\log t\ \alpha(s,t) \leq t\log\log t\ \alpha(t,t) = O(t\log t)$.

Theorem 2 *A contour tree in a triangulated d-dimensional mesh with N cells and t critical points can be computed in $O(N + t\log t)$ time and $O(N)$ space.* □

Figure 2 shows a two-dimensional example with $n = 65536$ points. There are $t = 75$ join or split candidates, and only 5622 points are visited on monotone paths. The visited points are highlighted in dark gray.

6 Experimental Results

To prove the concept, we ran some experiments with some simplified versions of the algorithm A simple version of the sweep algorithm by Carr and the algorithm described in this paper were implemented. Both algorithms were applied to



1 2 3 4

Figure 3: The pictures show a sample iso-surface from each data set. The diagrams below illustrate the time needed for certain steps. In each diagram the left set of five bars represents the sweep algorithm and the right set represents the algorithm introduced in this paper. The bars in the diagrams represent the time in milliseconds taken by the following steps: identification of critical points, sorting points, building the join tree, building the split tree, and the last one is the overall time. The sweep algorithm does not have an identification step, and the time for this step is zero.

the examples shown in Figure 3 which were taken with friendly permission from Hamish Carr's website.[4] The points in these examples all lie on a regular grid. The mesh was given impicitly by a dissection of each cube into five tetrahedra, and $N = O(n)$ holds. This also allows fast access to the neighbors of a vertex without storing the triangulation explicitly.

To simplify matters, the implementation did not distinguish between join and split candidates and started a monotone path with *every* higher neighbor of a candidate vertex.

Figure 3 shows some statistics about running times. These data may depend on implementation details, whereas Table 1 shows some more invariant figures of the two algorithms.

Table 1 illustrates the described advantages of using monotone paths opposed to a sweep over all points. If the number of critical points is only a small fraction of the total number of points, then the number of vertices visited by paths during the algorithm is also only a small fraction of the total number of vertices. The sweep algorithm always visits every vertex twice.

The diagrams in figure 3 show that sorting is very fast and reducing it to sorting only candidates is just a small advantage while the main benefit in runtime comes from the faster creation of the trees due to processing fewer vertices. This benefit of offset by the inital sweep over the data.

7 Conclusion

7.1 Comparison to Previous Algorithms

Theoretical Comparison. The best previous algorithm takes time $O(N + n \log n)$. The new complexity $O(N + t \log t)$ is superior if $t \ll n$ and if $n \log n$ dominates N. For "nice" meshes,

[4] `http://www.cs.ubc.ca/~hcarr/research/contourtrees.html`

Example	n	t	t/n	T_{join}	T_{split}	V_{sweep}	V_{path}	$V_{\text{path}}/V_{\text{sweep}}$
1	125	15	13%	9	3	250	95	39%
2	13851	40	1%	49	1	27702	2227	9%
3	27968	2833	11%	1089	1424	55936	29200	53%
4	66120	3922	6%	2074	1659	132240	52834	40%

Table 1: A summary of the examples. Among n points, there were t join and split candidates, and the join and split tree had T_{join} resp. T_{split} nodes. V_{sweep} and V_{path} denotes the number of visited vertices for the sweep algorithm and for our new approach.

$N = O(n)$ and the last condition is always fulfilled. The size of t depends of course very much on the data. It is easy to construct a planar mesh with about $n/2$ saddle points and $n/2$ maxima/minima. However, if one thinks about reasonably smooth functions that are successively represented on a finer and finer finer grid, the number of critical points stays essentially constant and does not grow with n. On the other hand, in cases of simulations of difficult physical phenomena like turbulence, a large fraction of vertices may be critical.

Practical Issues for Large Data Sets. From a theoretical point of view, sorting of data is not an issue in practice. Numeric data that comes from real-world applications is usually reasonably well-behaved and can be sorted in linear expected time by bucketing techniques. *Radix sort* is a practical way to sort even arbitrary sets of `floats` or `doubles` in linear time. Thus, the theoretical $\Theta(n \log n)$ lower bound on sorting does not play a role.

However, for huge data sets, our algorithm can be expected to have additional advantages. Assume that the input data (the data points together with connectivity information) do not fit into main memory, but are kept in virtual memory or even in external memory (on a disk). We will discuss the case that the critical points are only a small fraction of the data.

In the initialization phase, the algorithm scans the data once and selects the critical points. This is a purely local operation. The critical points can be kept in main memory and sorted quickly. Walking monotone paths is an operation that accesses the data in a spatially coherent fashion. If the layout of the data in its storage medium reflects spatial locality (adjacent memory cells store points which are contiguous in space), We may speculate that our algorithm will perform well with today's hierarchical memory systems.

Contrast this with an algorithm that processes the points in sorted order. This will lead to a rather random access pattern, which limits the size of problems that can be treated to what can fit into main memory. The data for our experiments in Section 6 are too small to see any effect of this sort.

In our algorithm, we must mark the visited points. This is the only place where the original data set is modified. However, even this can be avoided if the visited nodes on all monotone paths are still a small enough subset. Then we can keep a copy of the visited nodes in memory, storing them in a hash table, for example. Thus, all access to the original data is *read-only*.

7.2 Future Work

It might be worth while to explore different strategies for selecting the outgoing edge in a monotone walk. There is a trade-off between several criteria. A steepest-ascent-type selection might lead to paths that achieve a big height difference and reach the top quickly, and to small seed sets. On the other hand, it may lead to a path that runs for a long time straight "in parallel" to an existing older path, where a movement to the side would lead to quick termination. Other heuristics, like the first edge in cyclic order (starting at some chosen starting edge), some combination of left-most and right-most (in the plane), or completely random selection might perform better in practice. Of course, all of this may depend on the characteristics of the data.

We are currently exploring the possibility to extend our approach to calculate Betti numbers, as in Pascucci [7, 8]. It seems that we would have to trace out the contours of the critical points at least, to obtain enough information about topological changes at critical points.

For "flat" areas where the function is almost constant, small perturbations may cause many spurious critical points to appear. Moreover, in the vicinity of saddle points (which are also "flat" regions), such jitter may also be caused by the discretization alone, without any noise in the data. One way to address this problem is the notion of topological *persistence* [3], which allows to define a threshold of importance and to cancel out critical points locally. However, in the current setup, this theory requires global topological information, and therefore it cannot be approached by our simple monotone path idea. It would be interesting whether some easy sort of elimination can be done locally, which would at least approximate the idea of canceling critical points according to topological persistence.

References

[1] Carr, H. Efficient generation of contour trees in three dimensions. Master's thesis, University of British Columbia, 2000.

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

[3] Edelsbrunner, H., Letscher, D., and Zomorodian, A. Topological persistence and simplification. In *Proc. 41st Ann. Symp. Found. Computer Sci., Redondo Beach, California* (2000), IEEE Computer Society Press, pp. 454–463.

[4] Kettner, L., and Snoeyink, J. A prototype system for visualizing time-dependent volume data. In *Proc. 17th Ann. Symp. Comput. Geom.* (2001), pp. 327–328.

[5] Morse, S. P. A mathematical model for the analysis of contour-line data. *J. Assoc. Comput. Mach.* **15**, 2 (1968), 205–220.

[6] Morse, S. P. Concepts of use in contour map processing. *Commun. ACM* **12**, 3 (1969), 147–152.

[7] Pascucci, V. On the topology of the level sets of a scalar field. Tech. Rep. technical report UCRL-JC-142262, Lawrence Livermore National Laboratory, Feb 2001.

[8] Pascucci, V. On the topology of the level sets of a scalar field. In *Proc. 13th Canad. Conf. Comput. Geom. (CCCG 2001)* (University of Waterloo, Ontario, Canada, Aug 2001), T. Biedl, Ed., pp. 141–144. `http://compgeo.math.uwaterloo.ca/~cccg01/proceedings/`.

[9] Reeb, G. Sur les points singuliers d'une forme de Pfaff complement integrable ou d'une fonction numérique. *Comptes Rendus Acad. Sci. Paris* **222** (1946), 847–849.

[10] van Kreveld, M., van Oostrum, R., Bajaj, C., Pascucci, V., and Schikore, D. Contour trees and small seed sets for isosurface traversal. In *Proc. 13th Ann. Sympos. Comput. Geom.* (1997), pp. 212–220.