Efficient Methods for Isoline Extraction from a Digital Elevation Model based on Triangulated Irregular Networks

M. van Kreveld

UU-CS-1994-21
May 1994

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Efficient Methods for Isoline Extraction from a Digital Elevation Model based on Triangulated Irregular Networks

M. van Kreveld

Technical Report UU-CS-1994-21
May 1994

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

ISSN: 0924-3275

Efficient Methods for Isoline Extraction from a Digital Elevation Model based on Triangulated Irregular Networks*

Marc van Kreveld†

Abstract

A data structure is presented to store a triangulated irregular network digital elevation model, from which isolines (contour lines) can be extracted very efficiently. If the network is based on n points, then for any elevation, the isolines can be obtained in $O(\log n + k)$ query time, where k is the number of line segments that form the isolines. This compares favorably with $O(n)$ time by straightforward computation. When a structured representation of the isolines is needed, the same query time applies. For a fully topological representation (with adjacency), the query requires additional $O(c \log c)$ or $O(c \log \log n)$ time, where c is the number of connected components of isolines. In all three cases, the required data structure has only linear size.

1 Introduction

A digital elevation model is a means of modeling any real-valued function defined over the plane. Besides modeling elevations in a mountain landscape, other applications include modeling levels of pollution, air pressure, and density of a mineral or any other feature. An important concept for elevation models is that of the isoline. For an elevation value Z, the isoline map contains all points of the plane for which the elevation is exactly Z. Isoline maps in general contain the isolines for several different elevations, see Figure 1. The derivation of isoline maps is called contouring [10, 17, 21].

Some digital elevation models are based on storing isoline maps explicitly. However, these models are unsatisfactory for several reasons. One reason is that they are not suitable for computing slopes or making shaded relief models. A second reason is

*This research is partially supported by the ESPRIT Basic Research Action 7141 (project ALCOM II: *Algorithms and Complexity*).

†Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, the Netherlands. E-mail: marc@cs.ruu.nl.

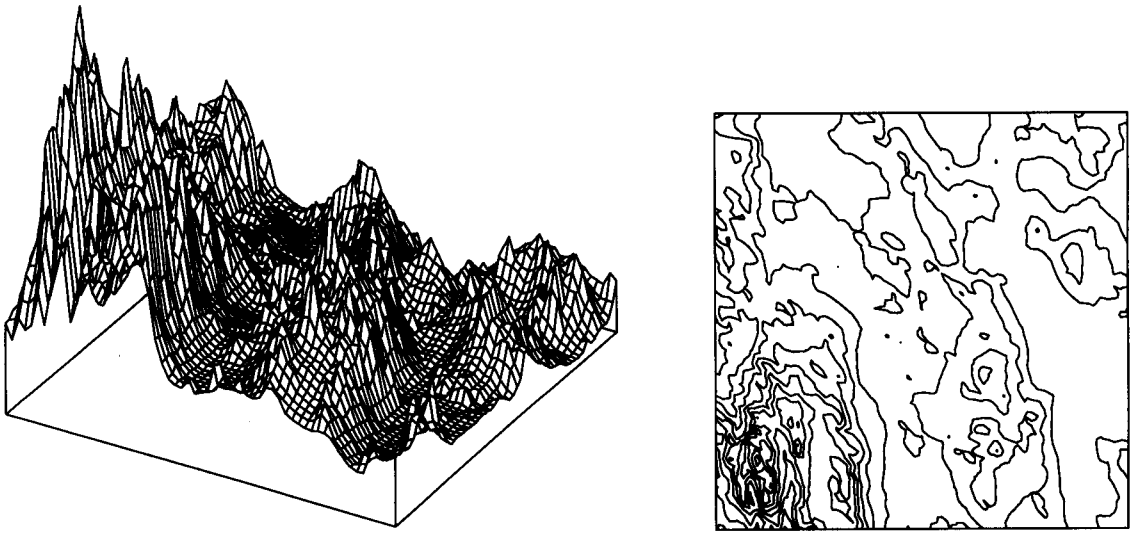

Figure 1: Perspective view of an elevation model and an isoline map of it.

that the elevation of points between isolines need still be determined. Therefore, it is a difficult and time-consuming process to obtain isolines of elevations that don't occur in the isoline map. Isolines can also be created using another digital elevation model, the triangulated irregular network (TIN), as an intermediate structure. See Figure 2 for a TIN and isolines on it. This approach has the advantage that isolines can be extracted for any elevation without any additional interpolation, and thus without extra error. Also, often a GIS already stores a TIN for other purposes, namely, to be able to show a perspective view of a digital elevation model. Therefore, using only a TIN and efficient algorithms for the extraction of isolines for any elevation provides a flexible and storage efficient solution.

A TIN (see Peucker et al. [22]) arises from any set of data points in the plane for which an elevation is given. The elevation of any intermediate point is given by linear interpolation on the elevations of three original points that form a triangle that contains the intermediate point. Often, the triangulation of choice is the Delaunay triangulation because of its natural properties [15, 23, 26]. For any set of n points, the Delaunay triangulation can be computed in $O(n \log n)$ time [23]. Other triangulations are constrained Delaunay triangulations that include ridges and valley lines as edges in the TIN [5, 25] or regular triangulations. Methods to obtain a TIN from data points have been considered extensively [9, 13, 17], as well as methods to obtain a TIN from digitized isolines [2, 4, 14]. These methods have been developed to obtain an internal representation in a GIS. In this paper we will be dealing with the efficient extraction of isolines from the internal representation. This is necessary when the user of a GIS requests for isolines. Since the user has to wait for the computation, it is important

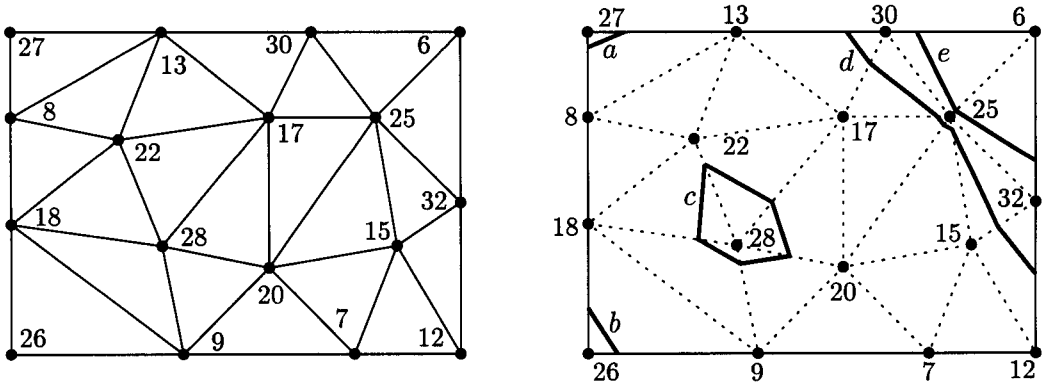

Figure 2: A TIN and the isolines of elevation 24.

that isoline extraction from the TIN is very efficient. The methods presented in this paper apply to any TIN representation of a digital elevation model regardless of how it was obtained. Therefore it is simply assumed that a TIN is given.

We will not consider raster based elevation models such as altitude matrices. An algorithm exists that retrieves contour lines from a grid [6]. The algorithm scans the whole grid, and is therefore too inefficient for large data sets and a waiting GIS user.

Given a TIN on n points, the isolines for any elevation can be extracted with a simple algorithm in $O(n)$ time, namely, by considering every triangle and testing if the given elevation occurs on that triangle. This is unsatisfactory especially when there are only few isolines for the given elevation. In Section 2 we show that by preprocessing, this can be improved considerably. By using an *interval tree* [7, 16], the isolines for any query elevation Z can be extracted in $O(\log n + k)$ time, where k is the number of line segments in the *isoline map* (the set of line segments of elevation Z).

When the connectedness of these line segments is relevant a slightly more complicated method is needed, which will be presented in Section 3. The connected components of line segments in the isoline map, the *isoline components*, can also be obtained with a query time $O(\log n + k)$. The sequence of line segments are output in order along the isoline component, which is necessary if smoothing is done. The map that is obtained is the *structured isoline map*. The method makes use of a network structure for the TIN, which can be a doubly connected edge list [20, 23], a quad edge structure [12], or a topological polygon network structure [3]. We will describe an easier variant that makes use of the fact that a TIN is already quite structured, see also Peucker [21] for similar data structures. For our purposes, an extension is needed which will also be described.

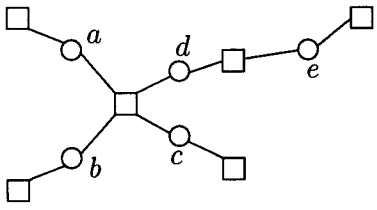

Figure 3: The adjacency structure of the isoline components of Figure 2. Circles represent isoline components and square nodes represent regions.

In some applications not only the connectedness, but also the adjacency among isoline components is relevant, for example for coloring of the regions and for geographical analysis. The topological information can be represented in a *contour tree* [11, 18, 19, 24], see Figure 3. We call the map with this additional information the *(fully) topological isoline map.* Using an additional data structure based on the contour tree, we can obtain it with $O(\log n + k + c \log c)$ or $O(\log n + k + c \log \log n)$ query time, whichever is smaller. Here, c is the number of isoline components and k is the total number of line segments in them. Section 4 presents this result.

The algorithms and data structures can be extended to the case where the isoline map of more than one elevation is required. This extension is straightforward. When the isoline map (unstructured, structured or topological) is needed for e elevation values, the query time becomes $O(e \log n + k)$ for the unstructured and structured cases, and $O(e \log n + k + c \log c)$ or $O(e \log n + k + c \log \log n)$ for the topological case.

In Section 5 we conclude the paper by presenting some quantitative results which indicate that the methods of this paper are more efficient that straightforward methods. Throughout the paper, we give some ideas for more efficient implementations, leading to data structures that use less storage in practice (some evidence is given in the conclusions section).

From now on, a data point of a TIN is called a *vertex* and a line segment between two vertices of the TIN is called an *edge.* To distinguish the pieces of isolines from edges, we refer to them as the *line segments* of the isolines. When considering isolines on a TIN, one can observe that any line segment of an isoline lies completely on a triangle or it is a horizontal edge of the TIN. An isoline component could also be a single point—a vertex of the TIN—when it is a local extremum. Let a TIN with n vertices be given, where every vertex is assigned an elevation.

2 Isoline map extraction

One obvious way of extracting all line segments of the isoline map from a TIN without any additional preprocessing is the following. Given the query elevation Z, consider all triangles in turn, and if the triangle contains a line segment of elevation Z, report it. This approach has a query time of $O(n)$, since all $O(n)$ triangles of the TIN are treated in constant time each. This brute-force approach is unsatisfactory especially when the number of triangles that cross the elevation Z is much smaller than the total number of triangles in the TIN. To obtain a more efficient solution, we describe the *interval tree*, a geometric data structure that stores a set of intervals of the real line. It was developed by Edelsbrunner [7] and by McCreight [16]. Here we give a brief

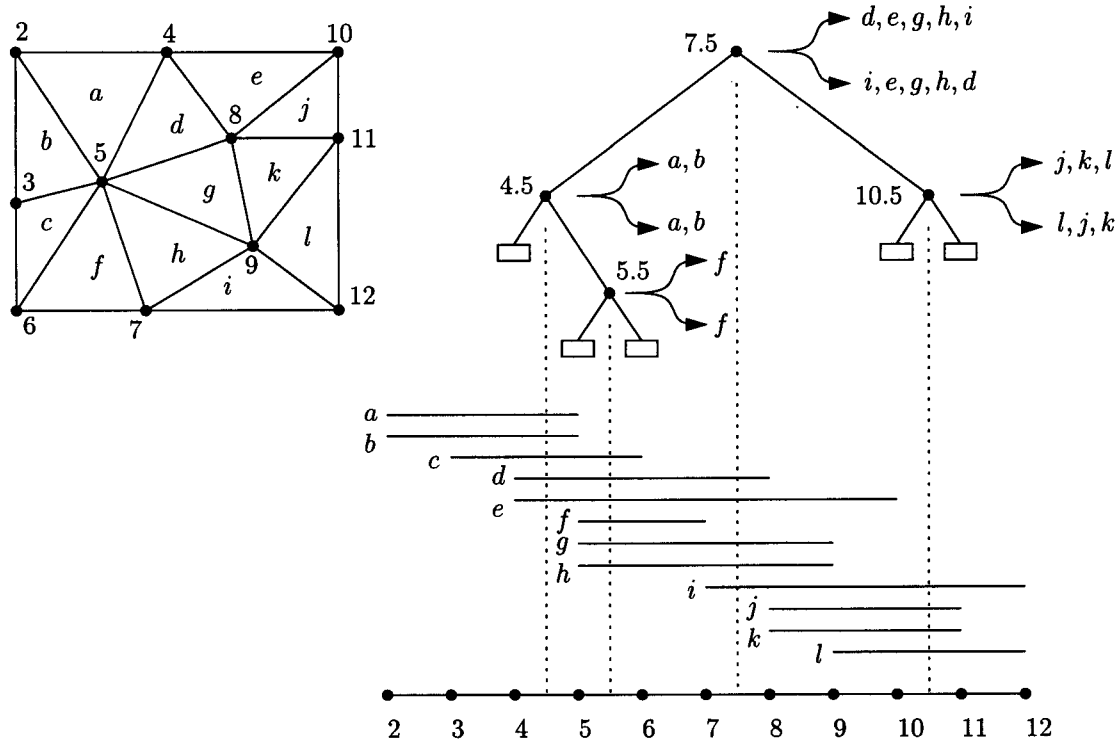

Figure 4: An example of a TIN and the corresponding interval tree. The split value and the two lists are shown with each node, where L is the upper list and R the lower list.

description (see Figure 4).

Let I be a set of open intervals of the form (a, b), where $a, b \in \mathbb{R}$ and $a < b$. The interval tree for I has a root node δ that stores a split value s. Let I_{left} be the subset of intervals (a, b) for which $b \leq s$, let I_{right} be the subset of intervals (a, b) for which $a \geq s$ and let I_δ be the subset of intervals for which $a < s < b$. The subsets $I_{left}, I_{right}, I_\delta$ form a partition of I. The subset I_δ is stored in two linear lists that are associated with node δ. One list L_δ stores I_δ on increasing value of the left endpoint, and the other list R_δ stores I_δ on decreasing value of the right endpoint. If I_{left} is not empty, then the left subtree of δ is defined recursively as an interval tree on the subset I_{left}. The right subtree of δ is defined in a similar way for I_{right}. It follows that any interval of I is stored exactly twice (namely, at one node in two lists). An interval tree for n intervals uses $O(n)$ storage, it can be constructed in $O(n \log n)$ time and if the split values s split roughly balanced, the interval tree has depth $O(\log n)$.

The query algorithm follows one path from the root to a leaf of the tree. Let q be the query value, thus, we want to report all intervals that contain q. At each node δ that is visited, it is determined by comparing q to the split value s stored at δ whether L_δ or R_δ is searched, and in which subtree the query continues. If $q < s$,

then we search in the list L_δ and report all intervals that contain the query value. These intervals appear at the start of the list by the sortedness. After searching in L_δ, the query proceeds in the left subtree. If $q > s$, then the list R_δ is searched and the query proceeds in the right subtree. All intervals that contain a query value are reported in $O(\log n + k)$ time, where k is the number of intervals that is reported. The preprocessing and query algorithms are summarized using pseudo-code in Appendix A.

To use an interval tree for our purposes of retrieving the isoline map, note that every triangle of the TIN has a z-span, given by the open interval bounded by the elevation of the vertices of the triangle with lowest and highest elevation. For any query elevation Z between this lowest and highest elevation, the triangle contributes to the isoline map with a line segment on that triangle. The set of z-spans defined by the triangles of the TIN are stored in an interval tree, and with each z-span the corresponding triangle. Not only triangles, but also horizontal edges of the TIN can contribute to the isoline map with a line segment. The z-span of a horizontal edge is the closed interval containing a single elevation, the elevation of that edge. The interval tree can easily be adapted to store these closed intervals as well. Given the query elevation Z, the search in the interval tree retrieves all triangles that lie partially below and partially above Z, and all edges with elevation Z. The line segments of elevation Z on these triangles and edges together form the isoline map for elevation Z. The query time is $O(\log n + k)$, where k is the number of segments in the isoline map.

Theorem 1 *A TIN with n vertices can be stored in $O(n)$ space such that for any query elevation Z, all line segments of the isoline map of elevation Z can be extracted in $O(\log n + k)$ time, where k is the total number of line segments retrieved. The data structure can be built in $O(n \log n)$ time.*

It can be observed that the method just described does not find local maxima and local minima of the query elevation, which are vertices of the TIN. If necessary, this can easily be corrected by finding all local maxima and minima of the TIN, and storing them in the interval tree as well. From the storage standpoint, it is better to store these vertices separately in a balanced binary tree. In this tree the horizontal edges of the TIN can be stored as well. A binary tree stores a value only once, whereas an interval tree stores it in two lists. Hence the savings in storage (a query requires more time, though, because two trees are queried instead of one).

3 Structured isoline map extraction

The solution of the previous section does not reveal the connectedness of the line segments in the isoline map. This is fine if the isoline map was only retrieved to be shown directly on a computer screen. But if further processing of the information is required, one generally needs the cycles and paths of line segments that form the connected components of the isoline map. For instance, if smoothing is applied to

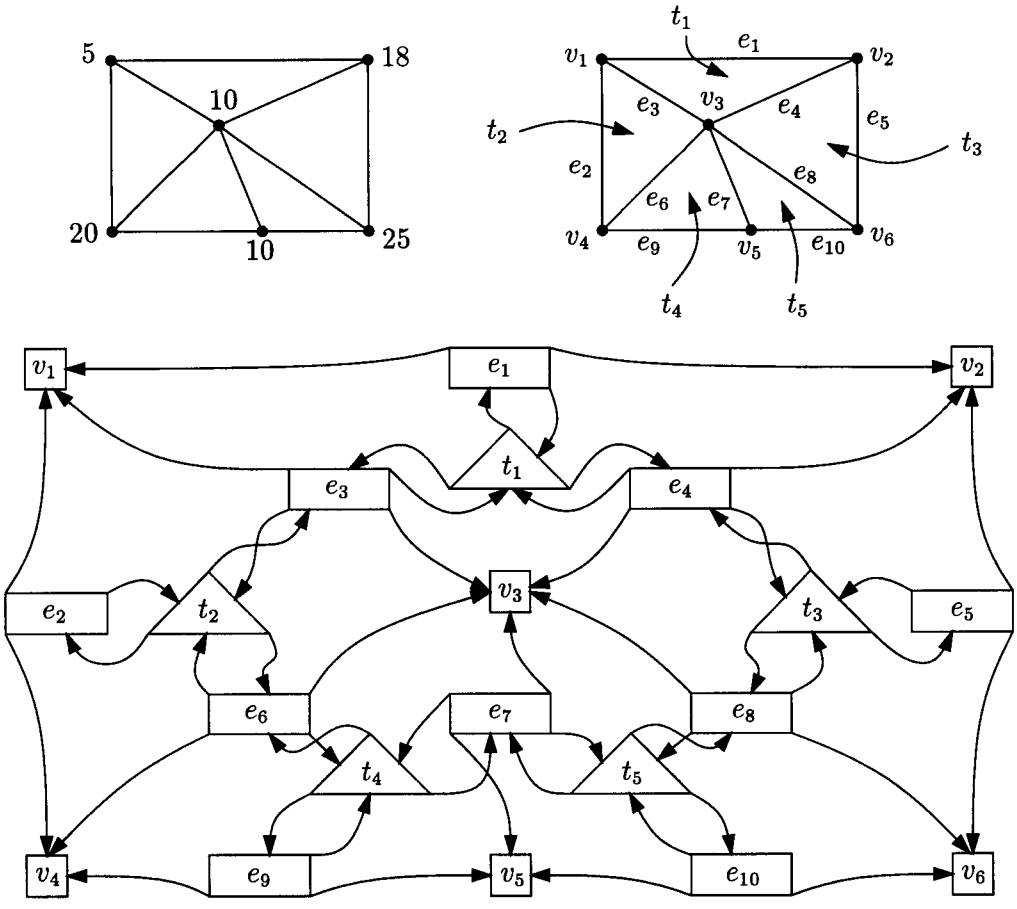

Figure 5: A TIN and the network structure for it. The three values and the list of each vertex are not shown.

isolines before they are displayed. We continue with an adaptation of the described method to obtain the cycles and paths. The adaptation is based on locating the isolines directly on a network structure of the TIN. The interval tree is still needed to have fast access to the network structure.

The network structure that we will use to store a TIN should allow of traversal operations. That is, it should be possible to get from one triangle to each of the adjacent triangles fast.

A simple data structure to store a TIN which allows of the necessary traversal operations is the following, see also Figure 5. For every triangle t, edge e, and vertex v, there is a record for that feature. The record of a triangle t has three fields with pointers. These pointers are directed to the records of each of the three edges incident to t. The record of an edge e has four fields with pointers. Two of the pointers are directed to the records of the two incident triangles, and the other two pointers are

directed to the records of the incident vertices. The record of a vertex v has four fields of which three are values and one is a pointer to a list. The list contains one element for each edge incident to v, and the element stores a pointer that is directed to the record of that edge. The three values are the x- and y-coordinates and the elevation of the vertex.

The network structure just described allows of finding—for every triangle—the elevations of its vertices in constant time, finding the adjacent triangles for a given triangle in constant time, and more. Suppose that we have discovered that triangle t_1 in Figure 5 contains a line segment of the isoline with elevation 17. By checking the adjacent triangles, we can find out on which triangles this isoline continues. In particular, we must examine the elevations of the incident vertices of these triangles. We can determine the exact location of the line segment on the isoline easily once we have the triangle. We use a mark bit in each triangle record which indicates whether this triangle has already been traversed. This allows us to determine when we have completed a cycle of an isoline component. Mark bits are also needed in the edge records of all horizontal edges of the TIN.

With this network structure, we can find the whole isoline component once we have a single triangle that contains a line segment of that component. Since the next line segment can be discovered in constant time unless that isoline component contains vertices, we can find the whole isoline component in $O(k)$ time when it contains k line segments. Appendix B contains pseudo-code that describes in more detail how to find the isolone component.

When an isoline component contains a vertex of the TIN, then the isoline component may contain several line segments incident to that vertex. In Figure 5, the vertex v_3 is incident to two triangles (t_1 and t_2) and one edge (e_7) that contribute to the isoline component of elevation 10 with a line segment. If we know in advance that every vertex in the TIN has constant degree, then we can still find the whole isoline component in $O(k)$ time if it contains k line segments. But otherwise, a vertex in a TIN can be incident to any number of edges and triangles, and we cannot find the ones that contribute with a line segment efficiently. The data structure described would require time linear in the degree of the vertex to find them.

To overcome this deficiency, we extend the record of each vertex v with a pointer to a second list. If v has elevation Z_v, then for every triangle incident to v that contains a line segment of elevation Z_v, the list has an element with a pointer to the record of that triangle. Also, for every edge incident to v that is horizontal, the list has an element with a pointer to the record of that edge. Figure 6 shows the record and lists of the vertex v_3 corresponding to the previous figure.

We have obtained the following result, using the extended network structure:

Theorem 2 *A TIN with n vertices can be stored in a data structure of size $O(n)$ such that for any elevation Z, the isoline component of that elevation can be computed in $O(k)$ time, where k is the number of line segments in the isoline component, once any triangle record of a triangle that contributes to that isoline component is given.*

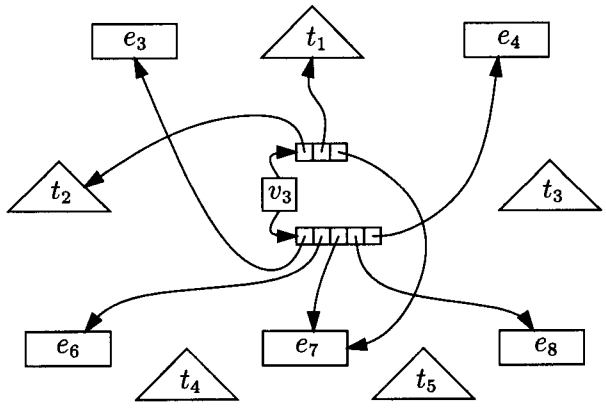

Figure 6: The pointers leaving the record of v_3 and the lists of it in the extended structure. The added list is the top one.

We will use the interval tree, defined on the z-spans of the triangles and horizontal edges as before, to obtain fast access to the network structure. With every occurrence of a z-span in the interval tree, we store a pointer to the corresponding triangle or horizontal edge in the network structure. Before each query, all records (of triangles and horizontal edges) in the network structure are unmarked. The mark bits have the following meaning. Setting the mark of a triangle or edge means that of the requested isoline, the line segment on that triangle or edge need still be reported. An unmarked triangle or edge either does not cross the query elevation, or the line segment on it has already been reported.

A query with elevation Z is performed as follows. We first search with elevation Z in the interval tree and push on a stack the triangles and edges of which the z-span contains Z. We also mark them in the network structure. Then, the triangles and edges are popped one by one. When a triangle or edge is popped, we first test whether it is marked or unmarked in the network structure. If it is unmarked, then we discard it and pop the next triangle or edge. Otherwise, we report the line segment of elevation Z on the triangle or edge and start a traversal in the network structure. The traversal traces one isoline component of elevation Z as described above. Every triangle or horizontal edge that is traversed is unmarked; it also appears somewhere on the stack and resetting the mark makes sure that it will not be traversed a second time. The traversal causes the fact that with this algorithm, the connectedness of the line segments in an isoline map is retrieved. Note that after the query, all triangles are again unmarked so that a next query can be done immediately. Appendix B summarizes the algorithm using a pseudo-code description (including a storage improvement based on left-extreme edges described below).

Every triangle and horizontal edge that gives a line segment in the structured isoline map is pushed, popped, traversed, marked and unmarked only once, and all of these operations take constant time. Therefore, the query time is $O(\log n + k)$.

It is true that the storage used by this method is somewhat larger than by the

previous one, but it is still linear in n. Also, the stack is only needed temporarily during the query, and the storage used by it comes free immediately after. The extra bits stored in the network structure form a small permanent overhead in storage.

Theorem 3 *A TIN with n vertices can be stored in $O(n)$ space such that for any query elevation Z, the structured isoline map for elevation Z can be extracted in $O(\log n + k)$ time, where k is the total number of line segments retrieved. The structure can be built in $O(n \log n)$ time.*

To reduce the storage of the interval tree, observe that only one starting triangle of every isoline component need be found in it. The other line segments will be found automatically during the traversal. Define an edge of the TIN to be *left-extreme* if an isoline component that crosses it causes a local minimum in x-coordinate in the isoline map. This can be tested for any edge in constant time by considering the elevations and coordinates of the four vertices incident to the two triangles that are incident to the edge. In Figure 2, the left-extreme edges are $(8, 18)$, $(8, 27)$, $(18, 26)$, $(18, 9)$, $(13, 30)$, $(30, 6)$, $(20, 25)$, $(9, 7)$, and $(18, 28)$. This can best be verified in Figure 7. Every isoline component must contain a left-extreme edge. Hence, for the triangles of which the z-span is stored in the interval tree, one need only use one of the triangles incident to any of the left-extreme edges. In practice, the number of left-extreme edges of a TIN is considerably smaller than the total number of edges. For random terrains, roughly one third of the edges is left-extreme, which means that roughly one half of the triangles are stored in the interval tree (by Euler's formula). For TINs with little variation the savings will be considerably better, see Table 1 in the conclusions section.

To obtain the isolines for several query elevations, we simply repeat the query for every elevation. This is straightforward, so we omit further description.

4 Topological isoline map extraction

The solution of the previous section solves the problem of finding the connected cycles and paths in an isoline map efficiently, but one aspect has not been solved: adjacency of isoline components. This is necessary when the region between two isolines should be colored, or when geographical analysis is performed on the isoline map. To discover which isoline components are adjacent to which other isoline components, a post-processing method based on *plane sweep* can be used. The idea is to sweep an imaginary line ℓ over the TIN from left to right, and retrieving the adjacency during this process. A dynamic balanced search tree is used during the sweep to maintain the intersection of the sweep line ℓ and the TIN. It can be shown that the adjacency relation of the k line segments can be determined in $O(k \log k)$ time with this method. Hence, the topological isoline map can be retrieved with $O(\log n + k \log k)$ query time. More details on plane sweep methods can be found in the book of Preparata and Shamos [23]. We will not go into details, since an improved

$O(\log n + k + \min\{c \log c,\ c \log\log n\})$ query time method will be presented next, where c denotes the number of isoline components in the isoline map for the query elevation. It uses one more data structure of linear size, namely, a rooted tree in which ancestor relations can be determined efficiently. The nodes of the tree represent the vertices of the TIN. Before going into details of the data structure and query algorithm, we define a new conceptual map.

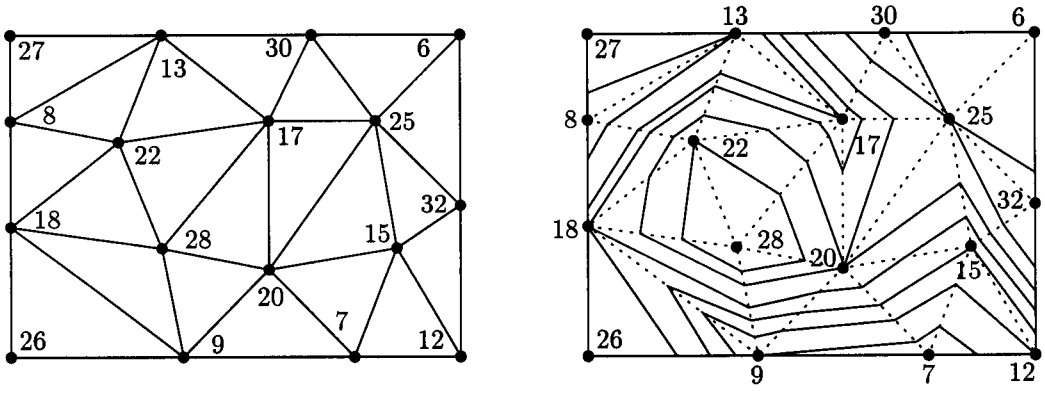

Figure 7: The TIN of Figure 2 and the corresponding vertex isoline map.

Let the *vertex isoline map* be the subdivision of a map obtained by taking all isoline components that contain vertices of the TIN, see Figure 7. Each vertex gives rise to an isoline component, although one isoline component may contain more than one vertex. Isoline components are simple polygons interior to the map, simple chains between two points on the boundary of the map, or more complicated connected sets of line segments. Two isoline components are *adjacent* in a map if they bound the same region of the map.

For any TIN, define $\mathcal{G}$ as the undirected graph with one node for every isoline component in the vertex isoline map, and an arc between two nodes representing components C_i and C_j if they are adjacent in the vertex isoline map. The graph $\mathcal{G}$ is closely related to the contour tree examined extensively by Morse [18, 19] and others [11, 24]. But it is defined on a special isoline map arising from the TIN, which includes the isolines through saddle vertices and local extrema. Therefore, it has additional properties. It can be shown that any region of the vertex isoline map is bounded by exactly two isoline components [1]. Therefore, the regions of the vertex isoline map correspond in a unique way to the arcs in $\mathcal{G}$ (this can be verified in Figures 7 and 8).

Lemma 1 *The graph $\mathcal{G}$ for any TIN is a tree (a connected acyclic graph).*

The graph $\mathcal{G}$ will be used to determine adjacency for any set of isoline components, not only those in the vertex isoline map.

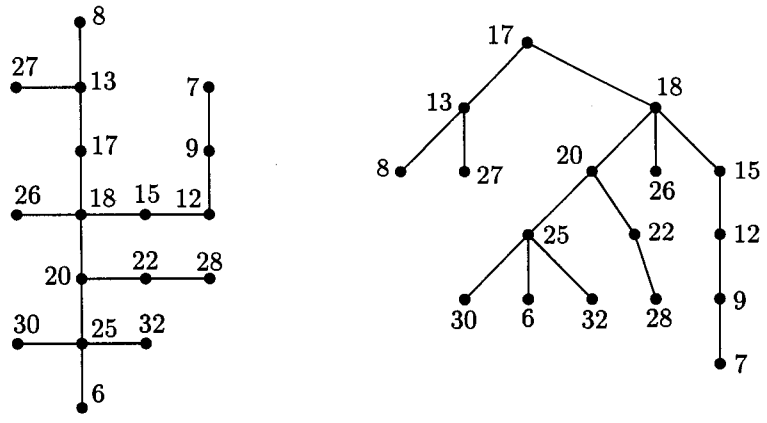

Figure 8: The graph $\mathcal{G}$ of the vertex isoline map of Figure 7 and a rooted tree version of it.

Lemma 2 *Any isoline component on a TIN either coincides with an isoline component of the vertex isoline map, or it is a simple path or simple polygon that lies in one region of the vertex isoline map. If it lies in a region, then it separates the two isoline components that bound that region of the vertex isoline map.*

For any isoline component of elevation Z that does not contain any vertex of the TIN, let $e_1, \ldots, e_j$ be the set of edges intersecting the isoline component. Then the region of the vertex isoline map in which it lies is bounded by isoline component of the lowest endpoint above Z of the edges $e_1, \ldots, e_j$ and by the isoline component of the highest endpoint below Z of the edges $e_1, \ldots, e_j$.

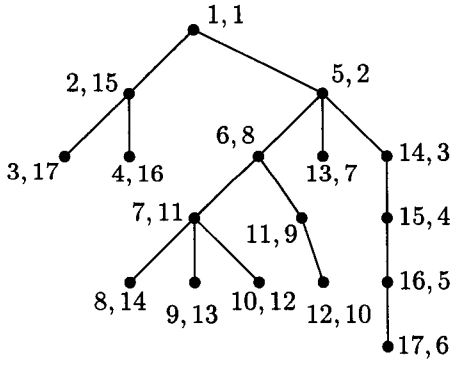

Figure 9: The two integers of the nodes of $\mathcal{T}$.

Using the observations about the geometry of isoline components given above, we define the following data structure. The graph $\mathcal{G}$ is stored as a rooted tree $\mathcal{T}$ with one node chosen arbitrarily as the root. Every node has pointers to its children. Let any parent order its children in an arbitrary manner from left to right. The nodes of $\mathcal{T}$ store two integers defined as follows, see Figure 9. The first integer is obtained using a depth-first-search tree traversal from left to right and numbering the nodes pre-order with 1 up to n. The second integer is obtained similarly, but the nodes are visited from right to left. The property of the integers is that any node δ is an ancestor of any node γ if and only if each integer of δ is smaller than the corresponding integer of γ. Summarizing, given two nodes in $\mathcal{T}$ one can determine in constant time whether one is an ancestor of the other.

The data structure that we use to retrieve the topological isoline map consists of the network structure and interval tree as described in the previous section, and the tree $\mathcal{T}$. Every record for a vertex v in the network structure is augmented with an extra pointer to the node in $\mathcal{T}$ which represents the isoline component that contains vertex v.

A query with elevation Z is performed as follows. First, using the interval tree and the network structure, we retrieve the isoline components in $O(\log n + k)$ time as before. Let $C'_1, \ldots, C'_c$ be the isoline components of elevation Z. For every isoline component C'_i, we determine where it lies on the vertex isoline map. This is done by checking for each isoline component C'_i all endpoints of the edges of the TIN that intersect it. Let v_{down} be the highest vertex below Z among these endpoints, and let v_{up} be the lowest vertex above Z among these endpoints. The vertices v_{down} and v_{up} both lie on an isoline component of the vertex isoline map, and these components C_{down} and C_{up} bound the same region of the vertex isoline map.

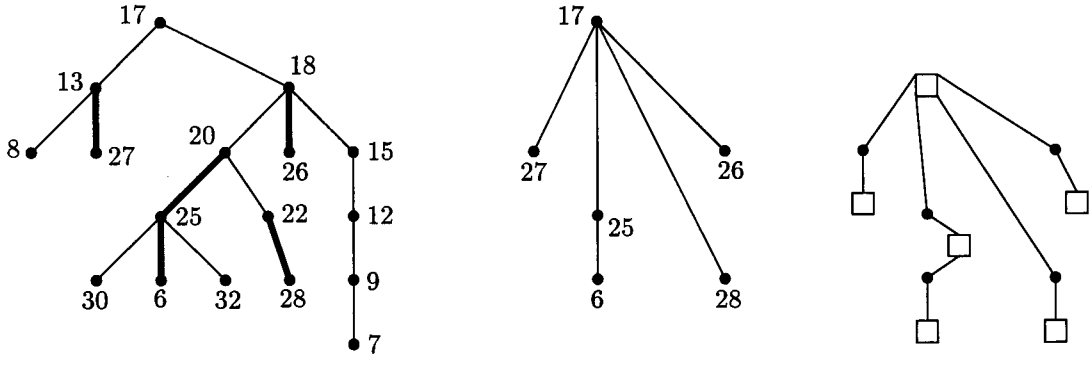

Figure 10: A tree $\mathcal{T}$ of Figure 8, the subtree that is obtained for the query with 24, and the adjacency structure of the isoline components. Compare it with Figure 3.

By an observation made earlier, the isoline component C'_i lies in the region that corresponds to the edge in the tree $\mathcal{T}$ between the nodes corresponding to C_{up} and C_{down}. Let u_i be the node corresponding to either C_{up} or C_{down}, whichever is the child of the other. We obtain a node u_i for every isoline component C'_i that was returned by the query. It is easy to see that the nodes $u_1, \ldots, u_c$ are distinct. The problem that remains is to determine the relation among the nodes $u_1, \ldots, u_c$ in $\mathcal{T}$, or the 'subtree' they define (see Figure 10). To obtain one subtree and not a forest of subtrees, we add the root node of $\mathcal{T}$ to the set of selected nodes. Let it be u_0. We show that the subtree can be computed in $O(c \log c)$ time or $O(c \log\log n)$ time by sorting algorithms.

Recall that every node in $\mathcal{T}$ stores two integers. First we sort the nodes $u_0, \ldots, u_c$ from left to right on increasing first integer. Next we scan this ordered sequence of nodes from right to left as follows. Assume without loss of generality that $u_0, \ldots, u_c$ is the ordered sequence obtained from the sorting step. This sequence is given in a list

initially with u_c at the tail (it has the largest first integer). Suppose we have scanned and treated from u_c back to u_{i+1}. To treat u_i, take the list element right of u_i, thus the first element closer to the tail. Test whether u_i is an ancestor of this element by checking the second integer: u_i is an ancestor if and only if the second integer of u_i is smaller (since the sequence is sorted on first integer, we already know that the first integer is smaller). If u_i is ancestor, we establish the adjacency in the subtree under construction, and remove the list element right of u_i from the list. Then we continue to compare u_i with the new neighbor to its right in the list in the same way. If the list element to the right of u_i was not a descendant of u_i, or if u_i is the rightmost list element, we have treated u_i and continue immediately with u_{i-1}. This completes the description of the computation of the subtree that contains the necessary topological information.

The topological isoline map for elevation Z can be extracted from this subtree with a simple transformation. If a node u_i is parent of nodes $u_{j_1}, \ldots, u_{j_m}$, then the topological isoline map has a region that is bounded by the components corresponding to these nodes. Figure 10 shows a tree $\mathcal{T}$ with a subset of nodes that could arise from a query elevation, the corresponding subtree and the adjacency information of the topological isoline map for that query elevation.

The correctness of the above method is based on the following fact for the sequence $u_0, \ldots, u_c$ sorted on the first integer: all descendants of any node u_i form a (possibly empty) sublist $u_{i+1}, \ldots, u_j$ of the list $u_1, \ldots, u_c$. The efficiency of the algorithm is determined by the time to sort and the time to scan. Sorting can be done by heapsort or quicksort in $O(c \log c)$ time or in $O(c \log\log n)$ time by a method of van Emde Boas et al. [8] (the latter method uses the fact that the integers to be sorted are in a 'bounded universe' consisting of the integers from 1 up to n). Scanning the list requires $O(c)$ time by the described algorithm.

To construct the tree $\mathcal{T}$, we rely on an algorithm described by de Berg and myself [1]. We showed that for any TIN, an $O(n \log n)$ time algorithm exists that computes for every vertex in the TIN the lowest vertex above it which can be reached with a monotonously increasing elevation path. If we link every vertex with this next higher vertex, the graph $\mathcal{G}$ is obtained. Choose a root node in $\mathcal{G}$ arbitrarily and establish parent-child pointers to obtain the tree $\mathcal{T}$. The assignment of the two integers to each node is done in linear time using a simple depth-first-search algorithm. We have established:

Theorem 4 *A TIN with n vertices can be stored in $O(n)$ space such that for any query elevation Z, the topological isoline map for elevation Z can be extracted in $O(\log n + k + \min\{c \log n, c \log\log n\})$ time, where c is the number of isoline components with elevation Z and k is the total number of line segments retrieved. The data structure can be built in $O(n \log n)$ time.*

To reduce storage requirements in practice, observe that the tree $\mathcal{T}$ can be discarded after the determination of the two integers stored at the nodes. This is because

the tree is not traversed at query-time, and none of the pointers are used after preprocessing. Thus we can simply store the two integers at the corresponding vertices in the network structure. Clearly, the extra pointer of a vertex in the network structure to the corresponding node in $\mathcal{T}$ can also be discarded. Savings add up to roughly $2n$ pointers.

To obtain the topological isoline map for several query elevations, we do the following. First, we perform the queries in the interval tree and the network structure to obtain the isoline components for all query elevations, which together form the structured isoline map as in the previous section. But this time, we also get a collection of c nodes in the tree $\mathcal{T}$, where c is the number of isoline components for all query elevations. These can then be sorted in $O(c \log c)$ time or $O(c \log\log n)$ time as we described. The sorting did not make use of the fact that before, the isoline components had the same elevation. So the topological information of the isoline map is computed after all isoline components have been obtained.

5 Conclusions

This paper proposed an approach to obtain isolines efficiently from a TIN-based digital elevation model. Instead of a straightforward computation of the isolines which requires $O(n)$ time, one can use one or two additional data structures that allow retrieval of the isolines in $O(\log n + k)$ time, where k is the number of line segments in the isolines. Thus the savings are considerable, especially when the isolines cross only a small number of triangles of the TIN, that is, when the output is small. It is also possible to obtain the isolines as a fully topological isoline map, in which case an extra post-processing phase requires additional $O(c \log c)$ or $O(c \log\log n)$ time, where c is the number of isoline components in the isoline map. The data structures use only linear storage, they can be constructed in $O(n \log n)$ time, and they are simple to implement. The isolines can be shown in a more attractive way by smoothing if we have the structured or topological isoline map (some care must be taken, however, to avoid intersections of isoline components by smoothing).

Table 1 shows the values of n, k and c for some terrains. (A simplification of the San Bernardino terrain is also shown in Figure 1 with the isolines of the elevations in the table.) The straightforward computation of the isoline map requires testing all 2,880,000 triangles. Using the data structures proposed in this paper, we find up to 35,000 triangles and horizontal edges (except in some cases and for random data). The overhead in query time is low: if the interval tree is well-balanced, its depth is at most $1 + \log_2 n$, and at each node two comparisons and following one pointer are necessary as the overhead. This amounts to only 44 comparisons and 22 times following a pointer as the overhead. The large values of k in the Lake Charles terrain are caused by plateaus of constant elevation. On the TIN based on a grid, such plateaus may contain many triangles, but for TINs that are not based on grids the plateaus will be covered by few large triangles instead, leading to much smaller

Source: Random data
Elevation range: 0–63
Left-extreme edges: 1,351,578

elevation	edges ($= k$)	comp. ($= c$)
8	968,046	124,042
16	1,620,899	113,005
24	1,823,455	68,045
32	2,093,573	1,069
40	1,958,665	101,880
48	1,710,995	180,523
56	833,033	112,781

Source: Denver (West)
Elevation range: 1557–4350
Left-extreme edges: 1,152,346

elevation	edges ($= k$)	comp. ($= c$)
1800	3127	1
2200	16,815	22
2600	22,246	71
3000	29,003	66
3400	26,403	54
3800	7615	48
4200	287	8

Source: Montreal (East)
Elevation range: 6–944
Left-extreme edges: 827,071

elevation	edges ($= k$)	comp. ($= c$)
50	5891	3
150	13,874	27
250	24,820	68
350	12,615	61
450	5589	31
550	3041	20
650	1371	10

Source: Austin (West)
Elevation range: 80–383
Left-extreme edges: 862,755

elevation	edges ($= k$)	comp. ($= c$)
100	6014	56
140	35,457	124
180	22,087	57
220	17,572	28
260	22,454	52
300	16,523	64
340	4622	28

Source: San Bernardino (East)
Elevation range: 274–3474
Left-extreme edges: 1,206,434

elevation	edges ($= k$)	comp. ($= c$)
400	1444	4
800	20,503	41
1200	18,764	78
1600	8455	20
2000	6913	15
2400	5655	13
2800	1979	8

Source: Grand Canyon (West)
Elevation range: 278–2513
Left-extreme edges: 1,255,938

elevation	edges ($= k$)	comp. ($= c$)
300	519	26
650	9883	64
1000	18,896	69
1350	34,920	228
1700	34,835	353
2050	7138	82
2400	165	3

Source: Manhattan (West)
Elevation range: 340–549
Left-extreme edges: 774,661

elevation	edges ($= k$)	comp. ($= c$)
350	1436	3
380	18,308	25
410	38,751	52
440	44,852	82
470	31,532	70
500	7027	64
530	214	4

Source: Lake Charles (West)
Elevation range: 0–152
Left-extreme edges: 363,961

elevation	edges ($= k$)	comp. ($= c$)
0	129,640	8
20	23,545	7
40	22,820	22
60	123,818	78
80	9165	17
100	2292	38
120	797	4

Table 1: Some quantitative results for elevation models. All eight TINs are based on a regular grid of 1201×1201 with 1,442,401 vertices, 4,322,400 edges and 2,880,000 triangles.

values of k. Further experiments are needed to examine these situations.

The table also shows the number of left-extreme edges, which are the ones for which an incident triangle must be stored in the interval tree. The savings in storage space for the interval tree are 50% up to 87%. Other experiments show that the ratio of the maximum value of k and the total number of triangles of the TIN decreases if the total number of triangles increases. For a square grid with 151, 301, 601 and 1201 vertices on each side, the ratios are 0.022, 0.020, 0.010 and 0.007, respectively, on the San Bernardino terrain. This indicates that the maximum value of k may be sublinear in the total number of triangles for this terrain. Again, further experiments are necessary to check this observation for other terrains and TINs.

Finally, the table indicates that—since c is small compared to k—a query time of $O(\log n + k + c \log c)$ (the presented algorithm to obtain the *topological* isoline map) is considerably better than a query time of $O(\log n + k \log k)$ (the presented algorithm to obtain the *structured* isoline map, combined with plane sweep to obtain the topology).

An interesting extension of the presented results would be to obtain the isoline map in a rectangular submap of the whole map. An approach solve this problem is by using a data structure that combines the ones presented in this paper with a k-d tree or an orthogonal range tree [23, 27]. This method requires more storage and it is difficult to analyze the worst-case behavior. However, the method should be fast in practice.

Acknowledgements: The author thanks Han La Poutré for helpful discussions.

References

[1] de Berg, M., and M. van Kreveld, Trekking in the Alps Without Freezing or Getting Tired. *Proc. 1st European Symposium on Algorithms* (1993), Lect. Notes in Comp. Science 726, Springer-Verlag, pp. 121–132.

[2] Brändli, M., A triangulation-based method for geomorphical surface interpolation from contour lines. *Proc. EGIS'92*, pp. 691–700.

[3] Burrough, P., *Principles of Geographic Information Systems for Land Resources Assessment.* Oxford Science Publications, 1986.

[4] Christensen, A.H.J., Fitting a triangulation to contours. *Proc. AUTO-CARTO 7* (1985), pp. 57–67.

[5] De Floriani, L., and E. Puppo, Constrained Delaunay triangulation for multiresolution surface description. *Proc. 9th IEEE Conf. on Pattern Recognition* (1988), pp. 566–569.

[6] Downing, II, J.A, and S. Zoraster, An adaptive grid contouring algorithm. *Proc. AUTO-CARTO 5* (1982), pp. 249–256.

[7] Edelsbrunner, H., *Dynamic data structures for orthogonal intersection queries.* Tech. Rep. F59, Tech. Univ. Graz, 1980.

[8] van Emde Boas, P., R. Kaas, and E. Zijlstra, Design and implementation of an efficient priority queue. *Math. Systems Theory* **10** (1977), pp. 99–127.

[9] Fowler, R.J., and J.J. Little, Automatic extraction of irregular network digital terrain models. *Computer Graphics* **13** (1979), pp. 199–207.

[10] Gold, C., Common sense automated contouring—some generalizations. *Cartographica* **21** (1984), pp. 121–129.

[11] Gold, C., and S. Cormack, Spatially ordered networks and topographic reconstructions. *Proc. 3th Int. Symp. on Spatial Data Handling* (1988), pp. 74–85.

[12] Guibas, L.J., and J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.* **4** (1985), pp. 74–123.

[13] Heller, M., Triangulation algorithms for adaptive terrain modelling. *Proc. 4th Int. Symp. on Spatial Data Handling* (1990), pp. 163–174.

[14] Keppel, E., Approximating complex surfaces by triangulation of contour lines. *IBM J. of Research and Development* **19** (1975), pp. 2–11.

[15] Laurini, R., and D. Thompson, *Fundamentals of Spatial Information Systems.* Academic Press, London, 1992.

[16] McCreight, E.M., Priority search trees. *SIAM J. Comput.* **14** (1985), pp. 257–276.

[17] Mark, D.M., Topological properties of geographic surfaces: applications in computer cartography, *Harvard Papers on Geographic Information Systems* **5** (1978).

[18] Morse, S.P., A mathematical model for the analysis of contour-line data. *J. ACM* **15** (1968), pp. 205–220.

[19] Morse, S.P., Concepts of use in computer map processing. *Comm. ACM* **12** (1969), pp. 145–152.

[20] Muller, D.E., and F.P. Preparata, Finding the intersection of two convex polyhedra. *Theoretical Computer Science* **7** (1978), pp. 217–236.

[21] Peucker, T.K., Data structures for digital terrain modules: discussion and comparison. *Harvard Papers on Geographic Information Systems* **5** (1978).

[22] Peucker, T.K., R.J. Fowler, J.J. Little, and D.M. Mark, The triangulated irregular network. *Proc. DTM Symp. Am. Soc. of Photogrammetry-Am. Congress on Survey and Mapping* (1978), pp. 24–31.

[23] Preparata, F.P., and M.I. Shamos, *Computational Geometry—an introduction.* Springer-Verlag, New York, 1985.

[24] Roubal, J., and T.K. Peucker, Automated contour labelling and the contour tree. *Proc. AUTO-CARTO 7* (1985), pp. 472–481.

[25] Scarlatos, L., A compact terrain model based on critical topographical features. *Proc. AUTO-CARTO 9* (1989), pp. 146–155.

[26] Watson, D.F., and G.M. Philip, Survey: systematic triangulations. *Computer Vision, Graphics, and Image Processing* **26** (1984), pp. 217–223.

[27] Willard, D.E., and G.S. Lueker, Adding range restriction capability to dynamic data structures. *J. ACM* **32** (1985), pp. 597–617.

Appendix A: Interval tree algorithms

Two recursive pseudo-code procedures are listed for the basic algorithms on an interval tree, namely, the construction and searching in it. The pseudo-code can be seen as a summary of the sequence of steps one can distinguish in an implementation. The algorithms are not meant to be complete; further refinement of some steps is necessary. For example, the algorithms assume that all intervals are open (it is straightforward to extend them to the case of open and closed intervals). Interval trees also support insertions and deletions [7, 16, 23]. Since those operations are not directly relevant to efficient isoline extraction, their description is omitted.

Algorithm *Construct-Interval-Tree*(I)
Input: A set I of n open intervals on the real line, presorted on the left endpoint and on the right endpoint.
Output: The root node δ of an interval tree that stores I.

1. **if** $|I| = 0$
2. **then return** an empty leaf node δ
3. **else** make a root node δ
4. Determine a value s such that $\leq n/2$ intervals of I have both endpoints $\leq s$, and $\leq n/2$ intervals of I have both endpoints $\geq s$. Store s as the split value in δ.
5. $I_{left} \leftarrow$ the intervals (a, b) with $b \leq s$
6. $I_{right} \leftarrow$ the intervals (a, b) with $a \geq s$
7. $I_{\delta} \leftarrow I - I_{left} - I_{right}$

8. Store I_δ sorted on increasing value of the left endpoint in a list L_δ. Store a pointer to L_δ with δ.
9. Store I_δ sorted on decreasing value of the right endpoint in a list R_δ. Store a pointer to R_δ with δ.
10. Left-Child(δ) $\leftarrow$ *Construct-Interval-Tree*(I_{left})
11. Right-Child(δ) $\leftarrow$ *Construct-Interval-Tree*(I_{right})
12. **return** δ

One can store I sorted on the left and the right endpoint by using two arrays, one for each order. In each array, the occurrence of an interval in the other array should also be stored so that determination of I_{left} and I_{right} can be done in a way that they are again presorted on left endpoint and right endpoint. The determination of the split value s in the above algorithm can be done easily when I is presorted on the left and the right endpoint. The algorithm takes $O(n \log n)$ time.

The easiest implementation is probably not keeping I sorted in any way, choosing the value s by taking a random endpoint of all endpoints in I, and sorting I_{left} and I_{right}, once they are determined, by quicksort. Some extra measures must be taken to assure that s splits well, which is a problem when there are many equivalent intervals. This randomized version requires expected $O(n \log n)$ time.

Algorithm *Query-Interval-Tree*(q, δ)
Input: A query value q and the root node δ of an interval tree.
Output: All intervals (a, b) in the interval tree for which $a < q < b$.
1. **if** δ is an empty leaf node
2. **then return**
3. **else if** $q < s$, the split value stored with δ
4. **then** scan the list L_δ from the front, testing for each interval (a, b) whether $q > a$. If so, report it and continue. Otherwise, stop scanning L_δ.
5. *Query-Interval-Tree*(q, Left-Child(δ))
6. **else** scan the list R_δ from the front, testing for each interval (a, b) whether $q < b$. If so, report it and continue. Otherwise, stop scanning R_δ.
7. *Query-Interval-Tree*(q, Right-Child(δ))

Appendix B: Structured isoline extraction algorithm

The following algorithm summarizes the steps needed to find the isolines of a TIN for a given query elevation Z. It is assumed that the triangles and horizontal edges of the TIN incident to left-extreme edges have been preprocessed into an interval tree, and that there are pointers from the intervals (z-spans) in the interval tree to the corresponding triangles and horizontal edges in the network structure representing the TIN. Contrary to the description in Section 3, the following algorithm uses a list instead of a stack during the query. Also, the mark bits are used in a slightly different way. This is more convenient when the idea of left-extreme edges is used.

It is assumed that initially, all triangles and edges in the network structure are unmarked. The algorithm itself resets all mark bits so that the structure is ready for the next query immediately. We ignore the issue of dealing with the efficiency loss when vertices have high degree. Some exception cases are also ignored.

Algorithm *Find-Isolines*(Z, TIN, δ)
Input: A query elevation Z, a network structure and the root node δ of an interval tree on the z-spans of the triangles and horizontal edges of the TIN that are incident to left-extreme edges.
Output: The isolines of elevation Z of the TIN as a linked collection of records for the vertices and edges of the isolines.

1. *Query-Interval-Tree*(Z, δ) and put all answers in a list L.
2. **for** every triangle t in L
3. **do if** t is marked in the network structure
4. **then** remove t from L
5. **else** mark t
6. Create an edge record and store the edge $(z = Z) \cap t$ in it.
7. **if** a vertex v and an edge e incident to t span elevation Z
8. **then** create a vertex record for v and link it to the edge record for t just created.
9. *Find-Component-From-Vertex*(t, v)
10. Create a vertex record for $(z = Z) \cap e$ and link it to the edge record for $(z = Z) \cap t$ just created.
11. *Find-Component-From-Edge*(t, e)
12. **else** let e and e' be the edges incident to t and which span Z.
13. Create vertex records for $(z = Z) \cap e$ and $(z = Z) \cap e'$ and link them to the edge record for $(z = Z) \cap t$ just created.
14. *Find-Component-From-Edge*(t, e)
15. *Find-Component-From-Edge*(t, e')
16. **for** every horizontal edge e in L
17. **do if** e is marked in the network structure
18. **then** remove e from L
19. **else** mark e
20. Create an edge record for the edge e.
21. Let v and v' be the vertices incident to e.
22. Create vertex records for v and v' and link them to the edge record of e.
23. *Find-Component-From-Vertex*(e, v)
24. *Find-Component-From-Vertex*(e, v')
25. **for** every triangle and horizontal edge in L which has not yet been deleted ($*$ one for each isoline component $*$)
26. **do** delete it from L
27. Traverse the whole isoline component again in a similar way to unmark all triangles and horizontal edges.

Algorithm *Find-Component-From-Edge*(t, e)

1. **if** e is incident to a triangle $t' \neq t$
2. **then** $(*$ t' doesn't exist at the boundary of the TIN $*)$
3. Create an edge record for the edge $(z = Z) \cap t'$ and link it to the vertex record created for $(z = Z) \cap e$.
4. **if** t' has a vertex v with elevation Z
5. **then** create a vertex record for v and link it to the edge record for $(z = Z) \cap t'$ just created.
6. *Find-Component-From-Vertex*(t', v)
7. **else** let $e' \neq e$ be the edge incident to t' and which spans Z.
8. Create a vertex record for $(z = Z) \cap e'$ and link it to the edge record for $(z = Z) \cap t'$ just created.
9. *Find-Component-From-Edge*(t', e')

Algorithm *Find-Component-From-Vertex*(f, v)

1. **for** all horizontal edges and triangles $f' \neq f$ incident to v which span Z
2. **do** create an edge record for the edge $(z = Z) \cap f'$ and link it to the vertex record just created for v.
3. **if** f' is a triangle
4. **then** let e be the edge incident to f' which spans Z.
5. Create a vertex record for $(z = Z) \cap e$ and link it to the edge record for $(z = Z) \cap f'$ just created.
6. *Find-Component-From-Edge*(f', e)
7. **else** let v' be the other endpoint of the horizontal edge f'.
8. Create a vertex record for v' and link it to the edge record for $(z = Z) \cap f'$ just created.
9. *Find-Component-From-Vertex*(f', v')