I/O-Efficient Algorithms for Contour-line Extraction and Planar Graph Blocking

(Extended Abstract)

Pankaj K. Agarwal[*] Lars Arge[†] T. M. Murali[‡]
Kasturi R. Varadarajan[§] Jeffrey Scott Vitter[¶]

Center for Geometric Computing
Department of Computer Science
Duke University
Durham, NC 27708–0129

Abstract

For a polyhedral terrain Σ, the contour at z-coordinate h, denoted C_h, is defined to be the intersection of the plane $z = h$ with Σ. In this paper, we study the contour-line extraction problem, where we want to preprocess Σ into a data structure so that given a query z-coordinate h, we can report C_h quickly. This is a central problem that arises in geographic information systems (GIS), where terrains are often stored as Triangular Irregular Networks (TINs). We present an I/O-optimal algorithm for this problem which stores a terrain Σ with N vertices using $O(N/B)$ blocks, where B is the size of a disk block, so that for any query h, the contour C_h can be computed using $O(\log_B N + |C_h|/B)$ I/O operations, where $|C_h|$ denotes the size of C_h.

We also present an improved algorithm for a more general problem of blocking bounded-degree planar graphs such as TINs (i.e., storing them on disk so that any graph traversal algorithm can traverse the graph in an I/O-efficient manner), and apply it to two problms that arise in GIS.

[*]Supported in part by National Science Foundation research grant CCR–93–01259, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by a National Science Foundation NYI award and matching funds from Xerox Corporation, and by a grant from the U.S.-Israeli Binational Science Foundation. Email: pankaj@cs.duke.edu.

[†]Supported in part by U.S. Army Research Office grant DAAH04–96–1–0013. Email: large@cs.duke.edu.

[‡]Affiliated with Brown University. Supported in part by National Science Foundation research grant CCR–9522047 and by Army Research Office MURI grant DAAH04–96–1–0013. Email: tmax@cs.duke.edu.

[§]Supported in part by National Science Foundation Grant CCR–93–01259, by an Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, and by a National Science Foundation NYI award and matching funds from Xerox Corporation. Email: krv@cs.duke.edu.

[¶]Supported in part by the U.S. Army Research Office under grants DAAH04–93–G–0076 and DAAH04–96–1–0013 and by the National Science Foundation under grant CCR–9522047. Part of this work was done while visiting Bell Laboratories, Murray Hill, NJ. Email: jsv@cs.duke.edu.

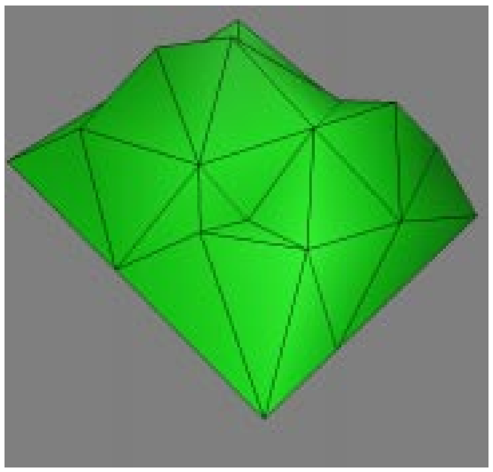


Figure 1: A terrain represented as a TIN.

1 Introduction

In Geographic Information Systems (GIS), a polyhedral terrain, which is the graph of a continuous, piecewise-linear bivariate function (see Figure 1), is often represented as a Triangulated Irregular Network (TIN). Since a terrain can be stored as a planar graph, many problems in GIS can be formulated as the traversal of a portion of a planar graph. Most known graph algorithms assume that the traversed graph can be stored in internal memory and optimize CPU time.

GIS systems, however, often store and manipulate enormous amounts of data, so we cannot assume that it fits in internal memory. Typical CPU-efficient algorithms perform poorly on such large data since they do not exploit locality of reference, and input/output communication (or simply I/O) becomes a significant bottleneck. Our focus in this paper is on developing algorithms that optimize I/O performance.

1.1 Memory model

We develop our algorithms in the standard two-level I/O model proposed by Aggarwal and Vitter [1]. This model defines the following parameters:

$$\begin{aligned} N &= \# \text{ of elements in the problem instance,} \\ M &= \# \text{ of elements fitting in internal memory,} \\ B &= \# \text{ of elements per disk block,} \end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$. An I/O is the operation of reading (or writing) a disk block from (or into) internal memory. In this model, computations can only be done on elements present in internal memory.

Our measures of performance of an algorithm are the number of I/O operations (or I/Os) that it performs and the amount of space (disk blocks) that it uses for storage. Data in GIS applications is often so large that only linear-sized data structures (which use $O(N/B)$ disk blocks of storage) are feasible. (Our algorithms are also efficient in terms of CPU execution time, but we do not discuss CPU time here due to lack of space.)

The model is motivated by the fact that the slow part of a disk access is positioning the read-write head and waiting for the disk to rotate into position; once that is done, data in subsequent locations on the disk can be accessed very quickly. To amortize (or hide) disk latency, each I/O operation transfers a large block of contiguous data.

1.2 Problem statement

We are given a terrain Σ, whose extent (or domain of definition) is the entire xy-plane. Each face of Σ is a triangle; such a terrain is called a Triangulated Irregular Network (TIN) in GIS. For the sake of simplicity, we assume that no edge or face of Σ is parallel to the xy-plane; in the full version of the paper, we describe the modifications to our algorithm that are needed to avoid this assumption.

For a terrain Σ, the *contour* C_h at z-coordinate h is the intersection of Σ with the plane $z = h$. See Figure 2 for an illustration. Since Σ is assumed not to have a horizontal face, each connected component of C_h is a closed polygonal chain consisting of edges (which we call *segments*) formed by the intersection of the plane $z = h$ with the faces of Σ. If the plane does not pass through a vertex of Σ, all components are simple polygons (which we call *cycles*). Otherwise, each component is a collection of cycles, possibly sharing vertices. We consider the following *contour-line extraction* problem: *Given a terrain Σ, preprocess Σ into a data structure so that given a query z-coordinate h, we can output the contour C_h using $O(|C_h|/B)$ blocks, where each cycle in C_h is returned as a consecutive sequence of segments in cyclic order.*

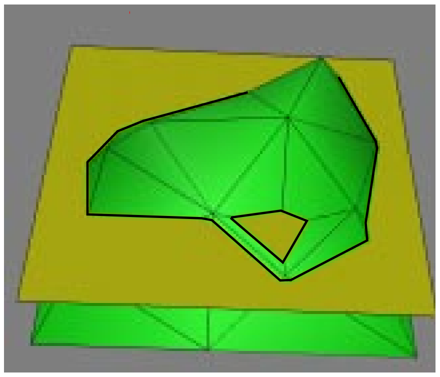


Figure 2: A contour of the terrain.

Returning the segments "in sorted order" is important in GIS applications [23], for example, when smoothing is applied to a contour before it is displayed.

1.3 Previous results

In the last few years, considerable attention has been given to the development of I/O-efficient algorithms in many problem domains, including sorting, graph algorithms, string algorithms, computational geometry, and GIS; see [1, 2, 3, 4, 5, 8, 19, 21, 27] and the references therein.

Although the contour-line extraction problem has been well-studied for terrains stored as raster images (for example, see the Marching Cubes algorithm [18]), not much work has been done when terrains are stored as TINs. van Kreveld [23] gives an internal-memory algorithm for preprocessing a terrain into a data structure of size $O(N)$ so that a contour-line query at z-coordinate h can be answered in optimal $O(\log_2 N + |C_h|)$ time. His algorithm stores the z-span of each triangle of the terrain in an interval tree [13]. Given a query z-coordinate h, the algorithm searches the interval tree to compute all the faces of the terrain whose z-spans contain h, from which it extracts the segments of C_h in $O(|C_h|)$ time. The algorithm then traverses the terrain to obtain each contour component "in sorted order." Using the external interval tree recently developed by Arge and Vitter [6], Chiang and Silva [9] extended and generalized van Kreveld's approach to external memory so that "unstructured" contour-line (or iso-surface) extraction queries (where no traversal of the components is needed) can be answered using an optimal $O(\log_B N + |C_h|/B)$ I/Os. However, their algorithm needs an additional $O(|C_h|)$ I/Os to traverse the contour in "sorted order." van Kreveld et al. [26] have recently presented some related results.

Another solution to the contour-line extraction problem follows from the results of Goodrich et al. [20]

on the graph-blocking problem (which we formally define later). For a graph G of bounded degree d and of size sufficiently larger than M, they show how to store G in a data structure using $O(N/\log_d B)$ blocks so that any path of length T in G can be traversed using $O(T/\log_d B)$ I/Os. We can use their result to store Σ in a data structure of size $O(N/\log_2 B)$ so that any contour-line extraction query h can be answered in $O(\log_B N+|C_h|/\log_2 B)$ I/Os, assuming that each cycle in the contour has $\Omega(\log_2 B)$ vertices.

1.4 Our results

In Section 2, we study the graph-blocking problem, and show how to store a planar graph G of size N and bounded degree d in optimal $O(N/B)$ blocks so that any path of length T in G can be traversed using $O(T/\log_d B)$ I/Os. The linear storage bound is a significant improvement over the results of Goodrich et al. [20]. This result implies that we can store a terrain using $O(N/B)$ blocks so that contour-line extraction queries can be answered using $O(\log_B N + |C_h|/\log_2 B)$ I/Os, assuming that each contour component has $\Omega(\log_2 B)$ vertices. We also show that our graph-blocking result can be used to perform window queries (which we define later) on a TIN in an I/O-efficient manner.

While the graph-blocking technique can be used to traverse *any* path (among a possibly exponential number of paths) in a graph, the contour-line extraction problem itself is more restricted in the sense that a TIN has only $O(N)$ combinatorially distinct contours. We exploit this fact in Section 3 to obtain optimal I/O bounds for the contour-line extraction problem. We present a data structure that can store a given terrain using $O(N/B)$ blocks so that a contour-line extraction query at z-coordinate h can be answered in optimal $O(\log_B N + |C_h|/B)$ I/Os. In order to construct the data structure, we sweep the terrain using a horizontal plane and maintain a structure representing the contour lines contained in the sweep plane. We use the persistence paradigm [12, 22] to retain all versions of this structure in order to allow contour-line queries at any z-coordinate. In order to make this approach I/O efficient, we first show that it is sufficient to maintain in a persistent fashion a collection of linked lists under $O(N)$ insert, delete, and split operations. Next, we develop an efficient external-memory implementation of the persistent collection of linked lists under the above operations. This data structure may be of independent interest and may have other applications. Once the data structure is constructed, we can answer a contour-line query by first searching in a B-tree and then traversing a list. We believe that our overall algorithm for contour-line extraction is of practical use since our algorithm is very simple.

2 Efficient Blocking of Bounded-Degree Planar Graphs

In the *graph blocking problem* [20], we are asked to store a (bounded-degree) graph G on disk, i.e., assign the nodes of G to disk blocks, so that any path in G (a sequence of nodes in which consecutive nodes are edge-adjacent) can be traversed in an I/O-efficient manner. The assignment of nodes of G to blocks on disk is a preprocessing step that is fixed before any path is requested. Let π be a query path in G that is to be traversed and let v be the node in π that is to be traversed next. We say that a node v is *served* if there is some block in internal memory that contains v; otherwise, we load a block containing v from disk. Once v is served, the next node in π, which can be any node adjacent to v, is traversed. We emphasize that the path query is *on-line* in the sense that the next vertex in π is revealed only after the current vertex is served. Our goal is to minimize the number of blocks used to store the graph G and the number of blocks that are loaded to serve any query path. We allow a node to be stored in more than one block.

We now outline an efficient scheme for blocking any planar graph $G = (V, E)$ with maximum degree d. Our method is based on a technique developed by Frederickson [15] for partitioning planar graphs. Let $(V_1, \ldots, V_k)$ be a covering of the node set of V, that is, $V_i \subseteq V$ and $\bigcup_i V_i = V$. We refer to each V_i as a *region*. A node v is *interior* to a region V_i if it is adjacent only to nodes in V_i, while a *boundary* node is one which is present in at least two regions. A B*-division* is a covering $(V_1, \ldots, V_k)$ of the node set V by $k = O(N/B)$ regions, so that

1. each region V_i has at most B nodes,
2. any node v is either a boundary node or it is interior to some region V_i, and
3. the total number of boundary nodes is $O(N/\sqrt{B})$.

Based on the separator theorem of Lipton and Tarjan [17], Frederickson gave an algorithm for constructing a B-division of any planar graph.

We use a B-division of G as follows: Let S be the set of boundary nodes of the B-division. For each $v \in S$, we grow a breadth-first tree rooted at v until we have included $\sqrt{B}$ nodes in the tree; let us call the set of nodes in the tree the $\sqrt{B}$*-neighborhood* of v. Note that the $\sqrt{B}$-neighbourhood of v contains all nodes whose distance from v is smaller than $\frac{1}{2}\log_d B$. We divide the set S of boundary nodes into $O(N/B)$ subsets so that each subset consists of at most $\sqrt{B}$ nodes. For each such subset S', we store the $\sqrt{B}$-neighborhoods of all nodes in S' in a single block on disk. We also store the nodes in each region V_i of the B-division in a single block on the disk. It is easy to see that our blocking scheme uses $O(N/B)$ blocks of storage.

Our algorithm for traversing a query path is as follows. Assume that we are currently serving the query path π using the block b corresponding to some region V_i. We continue serving π using b as long as π stays within V_i. When π leaves region V_i, that is, it requests a node v that is not in V_i, it does so at some boundary node $v' \in V_i$. At this point, we load the block b' that contains the $\sqrt{B}$-neighborhood of v' into internal memory. When π leaves block b' by requesting a node v'' that is not in the $\sqrt{B}$-neighborhood of v', we load the block corresponding to some region V_j that contains v''. It is apparent that for every two blocks that we load from disk, we traverse at least $(\log_d B)/2$ nodes in π. Thus we have obtained the following result, whose optimality follows from the results of Goodrich et al. [20]:

Theorem 2.1 *A planar graph with bounded degree d can be stored using $O(N/B)$ blocks so that any path of length T can be traversed using $O(T/\log_2 B)$ I/Os.*

Since the dual graph of a planar triangulation (and thus of a terrain stored as a TIN) is 3-regular, we can use Theorem 2.1 to block a terrain so that any path in its dual graph of length T can be traversed in $O(T/\log_2 B)$ I/Os. We now apply our blocking scheme to two problems that arise in GIS.

In the *window query* problem, we want to preprocess a terrain Σ so that, given any rectangular query window W in the xy-plane, we can report the set W_Σ of all triangles whose xy-projections intersect W. In internal memory, one space- and time-efficient way of solving the problem [11] is to construct a point-location structure on Σ [14], as well as a doubly-connected edge list (DCEL) for the dual graph of Σ, and answer a query by locating the triangle containing one of the corners of W and performing a traversal of the dual graph to report W_Σ in $O(\log_2 N + |W_\Sigma|)$ time. To solve the problem in an I/O-efficient manner, we preprocess Σ into the $O(N/B)$ space external-memory point-location data structure described by Goodrich et al. [16], which supports point-location queries in $O(\log_B N)$ I/Os. We block the dual graph of Σ as described above. We thus obtain the following result.

Theorem 2.2 *A terrain Σ can be stored in a data structure using $O(N/B)$ blocks so that the set W_Σ of all triangles in Σ that intersect a query window W can be reported using $O(\log_B N + |W_\Sigma|/(\log_2 B))$ I/Os.*

We know of no previous solution to this important problem that uses linear space and is efficient in I/O-terms. It remains open whether window queries can be answered using $O(\log_B N + |W_\Sigma|/B)$ I/Os and a linear space data structure.

Using the above blocking of a terrain we can also improve upon the algorithm described in Section 1.3 for answering a contour-line extraction query (assuming that all components have length $\Omega(\log_2 B)$). We construct an external interval tree [6] on the z-spans of the triangles in Σ, query the interval tree to get a segment in each cycle in the contour, and then traverse each cycle in order using the path traversal scheme described above. Our data structure uses $O(N/B)$ space and answers a contour-line query in $O(\log_B N + T/\log_2 B)$ I/Os, where T is the size of the reported contour.

Remark 2.3 We can use a completely different approach to solve the contour-line extraction problem by modifying the interval tree so that along with every segment in the contour returned by the query, the two segments preceding and succeeding it on the contour are also returned. Using this information, we can do a "list-ranking" of the segments output by the query to obtain the segments of each cycle in the contour in order. The "list-ranking" takes $O(T/B \log_{M/B}(T/B))$ time [8]. Thus, we get an overall blocking scheme that uses $O(N/B)$ blocks of storage and allows contour-extraction in $O(\log_B N + T/B \log_{M/B}(T/B))$ I/Os. We will provide details in the full version of the paper.

3 Contour-Line Extraction

We now describe our main result, an optimal solution to the contour-line extraction problem for a given terrain Σ. The overall approach for constructing the data structure is quite simple: We sweep Σ in the $(+z)$-direction using a horizontal plane. At any height h of the sweep, we maintain a *plane-sweep structure* from which the contour C_h can be easily obtained. The plane-sweep structure needs to be updated only when the sweep plane passes a vertex of the terrain. We use the persistence paradigm [12, 22] to create a persistent structure that retains all older versions of the plane-sweep structure. Given a query z-coordinate h, we find the relevant version of the plane-sweep structure, from which we can output the segments of C_h in "sorted order." However, there are many issues that arise in making this approach work efficiently in terms of I/O, and we address them in the remainder of this paper.

For simplicity, we assume that the z-coordinate of each vertex of Σ is distinct; this implies that no edge or face of the terrain is parallel to the xy-plane. We assume that the unbounded faces of Σ are sloping downward. In the full version of the paper, we show how these restrictions can be removed.

We say that vertices u and v of Σ are *neighbours* if there is an edge in Σ whose endpoints are u and v. Let Σ' denote the portion of $\mathbb{R}^3$ that lies on or below the terrain Σ. Let z_h denote the plane $z = h$. If z_h does not

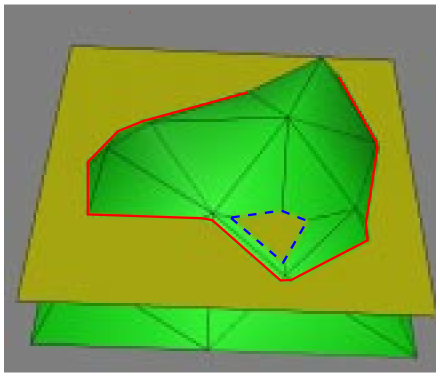


Figure 3: A red cycle (the thick solid line) and a blue cycle (the thick dashed line).

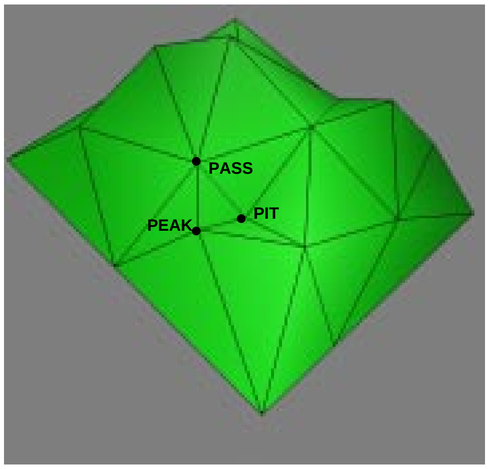


Figure 4: Types of terrain vertices.

contain a vertex of Σ, then each connected component of C_h is a simple cycle that partitions z_h into a bounded part, which we refer to as the *inside* of the cycle, and an unbounded part, which we refer to as the *outside* of the cycle. We call a component of C_h *red* if, "locally," the interior of $z_h \cap \Sigma'$ lies inside it; we call a component of C_h *blue* otherwise. See Figure 3.

We can represent the combinatorial structure of a component c of C_h (at a height h not containing any vertex of Σ) by a *cycle* of the faces that contribute a segment to c. We represent a red component by a "red" cycle of faces, and a blue component by a "blue" cycle of faces. Thus, we can represent the combinatorial structure of C_h by a collection of these red and blue cycles; with a slight abuse of notation, we use C_h to denote the combinatorial structure too. The combinatorial structure C_h is the same for all heights between two consecutive vertices of Σ. In the following, we examine how C_h changes as we vary h from $-\infty$ to $+\infty$, that is, as we sweep Σ with a horizontal plane in the $(+z)$-direction.

In order to describe the changes in C_h, it will be useful to identify three special kinds of vertices of the terrain: a *peak* is a vertex that is higher than all its neighboring vertices; a *pit* is a vertex that is lower than all its neighboring vertices, and a *pass* (or a *saddle vertex*) is a vertex having four neighboring vertices that are higher, lower, higher, and lower in cyclic order around it. See Figure 4. We will assume for simplicity that every saddle vertex is only "singly-saddle," that is, it does not have six neighboring vertices that are alternatingly higher and lower in cyclic order around it. If the plane z_h does not include any peak, pit, or pass of Σ, the components of C_h will be simple cycles with non-empty interiors. If z_h contains a pass of Σ, two of these cycles meet at the pass. If z_h includes a peak or a pit, there is a trivial component consisting of just the peak or the pit.

At the beginning of the sweep, that is, at $h = -\infty$, C_h consists of just one red cycle of the unbounded faces of Σ. (This fact follows because of the assumption that all unbounded faces of the terrain are sloping downward.) We now describe the changes in C_h when the sweep plane passes a vertex v whose z-coordinate is h. We denote a face incident to v as *old* (resp. *new*) if v is the highest (resp. lowest) vertex of f.

1. *v is not a peak, pit or pass:* The old faces incident to v belong to some cycle Q in C_h, which can be either red or blue. These old faces disappear and the new faces appear in the appropriate positions in Q.

2. *v is a pit:* A new blue cycle B' consisting of the (new) faces incident to v appears in C_h.

3. *v is a peak:* A red cycle R consisting of the (old) faces incident to v disappears from C_h.

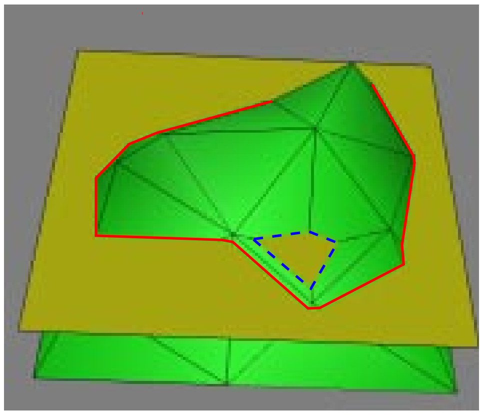
4. *v is a pass:* This is the interesting case, when either two cycles merge to from a bigger cycle, or a cycle splits into two cycles. Old faces incident to v get deleted from the cycles to which they belong, and new faces incident to v are inserted into the appropriate new cycles. It can be shown that the interaction between the cycles falls into one of the following four categories:

   (a) A red cycle R splits into two red cycles R'_1 and R'_2. See Figures 5(b) and 5(c).

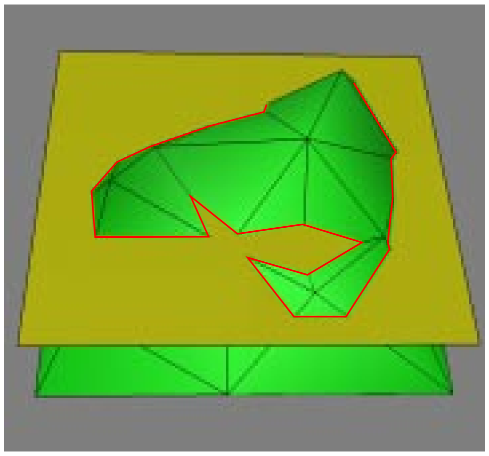
   (b) Two blue cycles B_1 and B_2 merge into a new blue cycle B'.

   (c) A blue cycle B merges with a red cycle R, resulting in a red cycle R'. See Figures 5(a) and 5(b).

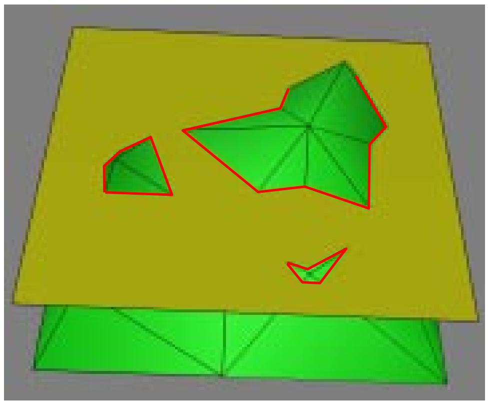
   (d) A blue cycle B splits into a red cycle R' and a blue cycle B'.



(a) A red (thick line) and a blue (dashed line) cycle.



(b) One red cycle.



(c) Three red cycles.

Figure 5: Different ways in which cycles interact during the plane sweep.

A simple but useful observation is the following lemma.

Lemma 3.1 *During the plane-sweep, if a face f appears in a red cycle in C_h, then f never appears subsequently in a blue cycle.*

During a sweep in the $(+z)$-direction, a cycle can thus change in only the following ways: it appears or disappears, a face appears in the cycle or a face disappears from the cycle, the cycle splits into two cycles, or the cycle merges with another cycle. We observe that two red cycles never merge. Furthermore, we can regard the merging of a blue and a red cycle as if the faces of the blue cycle appear in the red cycle. When a blue cycle splits into a red and a blue cycle, we can regard the event as though a new red cycle appears with the appropriate faces. Hence, the only way in which red cycles change as we sweep Σ in the $(+z)$-direction is that a new red cycle appears or disappears, a face appears in a cycle or disappears from a cycle, or a cycle splits into two cycles. From Lemma 3.1, it follows that that a face appears and disappears from the collection of red cycles in C_h at most once. Moreover, the number of times that a red cycle splits is at most the number of passes, which is $O(N)$.

These observations lie at the heart of our scheme for the contour-line extraction data structure. Our scheme uses two data structures, one for extracting the red contour components and another for extracting the blue contour components. As we observe below, building each data structure can be formulated as maintaining a collection of circular lists persistently under $O(N)$ applications of the operations of inserts, deletes, and splits; we need not handle both splits and merges. We will only describe the data structure from which the red components can be extracted; the data structure for extracting the blue components can be built by a symmetric procedure (in a sweep of the terrain in the $(-z)$-direction).

In the sweep described above, we associate a *plane-sweep structure* , (h) with the sweep plane z_h: , (h) is a collection of lists of faces, where each list stores the cycle of faces in a red cycle of C_h. We distinguish between a red cycle and the corresponding red list by regarding the former as a combinatorial representation of a cycle in a contour and the latter as a data structure. The remarks above imply that we can maintain , (h) during the sweep by a sequence of $O(N)$ applications of the following operations: create/remove an empty list, insert a face into a list, delete a face from a list, and split a list into two new lists.

In the next section, we present a data structure that maintains a set of lists persistently in external memory under the operations of insert, delete, and split. If the total number of operations is N, the persistent strucure uses $O(N/B)$ space. Any version of the structure can be extracted using $O(T/B)$ I/Os, where T is the size of the version. The preprocessing time needed to build this structure uses $O(N)$ I/Os. The preceding discussion implies the following theorem, which is the main result of our paper:

Theorem 3.2 *Given a terrain Σ of size N, we can construct a data structure that uses $O(N/B)$ blocks such that given a z-coordinate h, the contour C_h can be extracted from the data structure using $O(\log_B N + |C_h|/B)$ I/Os.*

4 Data Structure

In this section we describe how to maintain a set of ordered lists , in a data structure $\mathcal{L}$ under insert, delete, and split operations. We then show how to make $\mathcal{L}$ persistent.

An obvious choice for storing a collection of lists is to store them in a single "super-list" one after another. In this way inserts and deletes of elements are easy. Splitting a list is also easy, as it corresponds to moving a sublist of elements from somewhere in the super-list to (say) the end of the super-list. In internal memory such a structure can easily be made persistent using linear space [12, 22], and this representation is indeed almost the one we will use. However, if we use this technique to make an external version of the structure persistent, we may need one new block every time we perform a split. Thus we could end up using $\Omega(N)$ blocks of memory. In order to use only $O(N/B)$ blocks of memory, we use a clever technique to handle splits involving small lists. We divide split operations on lists into three groups: *major*, *minor* and *special.*

We say that a list L' is a *branch* of a list L, if L' results from L as a consequence of zero or more splits. That is, there exists a sequence of lists $L_0, \ldots, L_k$, where $L = L_0$ and $L' = L_k$, such that for $1 \leq i \leq k$, L_i is one of the lists formed when L_{i-1} is split.[1] The *set of descendants* of a list L, which we denote by $\text{Des}(L)$, consists of all elements f such that f occurs in some branch of L. We say that a list L is *major* if $|\text{Des}(L)| \geq B$, and *minor* if $|\text{Des}(L)| < B$. We classify the split operations performed on the lists into three kinds. A split of a list L into lists L_1 and L_2 is (1) *major* if L_1 and L_2 are major lists, (2) *minor* if L is a minor list, and (3) *special* if L is a major list and at least one of L_1 and L_2 is a minor list. Every split is major, minor, or special. Finally, we say that a minor list L is *maximal* if it is not a branch of any other minor list. Intuitively, minor splits are easier to handle than major splits because they only affect elements in one block. Furthermore, it is easy to prove the following lemma which bounds the number of major splits.

[1]Here, we think of an insert or delete operation as just modifying an already existing list in , ; that is, a list retains its "name" when an element is inserted in, or deleted from it. Thus, a "named" list is born either because of a create-empty-list operation or because of a split operation. A "named" list dies either because of a split operation on it, or because of a remove-empty-list operation

Lemma 4.1 *If a total of N insert, delete, and split operations are performed on , , there are only $O(N/B)$ major splits.*

4.1 Ephemeral structure

We are now ready to define $\mathcal{L}$ and describe how we implement the operations insert, delete, and split. We first describe $\mathcal{L}$ independently of how it is laid out on disk blocks. $\mathcal{L}$ is a list of the elements in , on which we maintain the following invariants:

I. The elements in any major list in , are stored in sequential order in a contiguous subsequence in $\mathcal{L}$.

II. The elements in , that are descendants of the same maximal list are stored as a contiguous subsequence in $\mathcal{L}$.

We do not require the elements belonging to the same minor list of , to occur as a contiguous subsequence in $\mathcal{L}$. However, as the following lemma shows, the invariants guarantee that the elements in the same minor list are not too far apart.

Lemma 4.2 *Let f and f' be two elements that belong to the same minor list in , . Then, the number of elements between f and f' in $\mathcal{L}$ is smaller than B.*

Proof: Since f and f' are in the same minor list, they are descendants of the same maximal list L. Since L is minor, the number of descendants of L is smaller than B. Invariant (II) guarantees that the descendants of L are contiguous in $\mathcal{L}$, thus implying the lemma. $\square$

We describe how we implement the operations needed on $\mathcal{L}$ so that the above invariants are maintained:

1. *Insert an element f into a new (empty) list L:* Insert f at the end of $\mathcal{L}$.
2. *Insert an element f into an existing list L:* If L is major, we scan $\mathcal{L}$ to find the "region" where the elements of L are stored, and insert f in the appropriate location. If L is minor, it is a branch of some maximal list L'. We insert f at the end of the subsequence in $\mathcal{L}$ consisting of the descendants of L'.

3. *Delete an element f from a list L:* We find and delete f from $\mathcal{L}$.

4. *Split a list L into two lists L_1 and L_2:*
   (a) *The split is major:* The elements in L form a contiguous subsequence in $\mathcal{L}$. The elements in either L_1 or L_2 occur as a contiguous subsequence α of this subsequence; assume, without loss of generality, that the elements of L_1 do. We "break off" α from $\mathcal{L}$, and append it to the end of $\mathcal{L}$. We call this operation *carving* the subsequence α from $\mathcal{L}$.

   (b) *The split is minor:* No updates are made.

   (c) *The split is special:* One of the lists, say L_1, is minor. We delete the elements of L_1 from $\mathcal{L}$, and insert them at the end of $\mathcal{L}$. The list L_1 is maximal, so this defines the "region" in $\mathcal{L}$ containing the descendents of L_1.

It can be verified that the above operations maintain the invariants on $\mathcal{L}$. The only operations performed on $\mathcal{L}$ are the operations of inserting an element, deleting an element, and carving a subsequence α from $\mathcal{L}$. We can prove the following important lemma about the number of such operations.

Lemma 4.3 *If a total of N inserts, deletes, and splits are performed on , , we perform $O(N)$ inserts and deletes, and $O(N/B)$ carve operations on $\mathcal{L}$.*

Proof: Each element is inserted and deleted once, except during a special split. As each element can be deleted and re-inserted in a special split at most once, the total number of inserts and deletes on $\mathcal{L}$ is $O(N)$. The number of carving operations is bounded by the number of major splits, which is $O(N/B)$ by Lemma 4.1. $\square$

4.2 Laying out $\mathcal{L}$ on disk

Maintaining $\mathcal{L}$ on disk now amounts to maintaining a single list of elements on disk under the operations insert, delete, and carve. The way we do this is similar to the way the leaves in a standard B-tree are maintained [7, 10]. We maintain $\mathcal{L}$ using a linked list of blocks; each block b stores a contiguous subsequence $\alpha(b)$ of $\mathcal{L}$, and the pointer from block b points to a block b' such that $\alpha(b')$ follows $\alpha(b)$ in $\mathcal{L}$. For every block b in the linked list we maintain the invariant $B/8 \leq |\alpha(b)| \leq B/2$, and implement the three operations as follows:

1. *Insert*: To insert an element e after element e' in $\mathcal{L}$, we first check if there is room for e in the block b containing e'. If this is the case we just insert it. Otherwise if b contains $B/2$ elements, we create two new blocks, distribute the elements in b evenly among them, delete b from the linked list of blocks, and insert the two new blocks into the linked list.

2. *Delete*: To delete an element e from $\mathcal{L}$, we check if the block b that contains e has more than $B/8$ elements and delete e if that is the case. Otherwise, we collect all the elements in b and one of the blocks adjacent to it in the list, and delete the two blocks. If the number of collected elements is less than $3B/8$, we create a new block with all the elements and link it into the list. If the number is greater than $3B/8$ we create two new blocks, divide the elements evenly among them, and link them into the list.

3. *Carve*: As with an insert or a delete, a carving can be performed by creating and deleting a constant number of blocks, and changing a constant number of pointers.

A careful analysis of the insert and delete operations (1 and 2 above) shows that whenever a new block is created it contains between $3B/16$ and $3B/8$ element. The carve algorithm can be designed such that the same is true for blocks created by that operation. Thus at least $B/16$ inserts/deletes, or a single carve operation, need to be performed on a newly created block if it is to be deleted as a consequence of some operation on it. From the bounds in Lemma 4.3 on the number of insert, delete, and carve operations, we obtain the following lemma; we omit its formal proof from this abstract.

Lemma 4.4 *A collection of lists , can be maintained on disk in $\mathcal{L}$, under N inserts, deletes, and splits, so that only $O(N/B)$ blocks are created and $O(N/B)$ pointers changed in total, and so that at any point in time the current version of $\mathcal{L}$ is stored in a linked list of $O(|\mathcal{L}|/B)$ blocks.*

As an aid to making our blocked data structure $\mathcal{L}$ persistent, we impose one final invariant on each block b in in the ephemeral structure $\mathcal{L}$: At most $B/2$ updates are done on b from the time it is created. In order to maintain this invariant on the ephemeral structure $\mathcal{L}$, we introduce a new operation on the blocks in $\mathcal{L}$ called the *copy* operation; to copy a block b, we simply copy the elements stored in b into a new block b'. To implement the new invariant, we store with each block a count of how many updates have been performed on it. Once the count of a block b reaches $B/2$, we copy the elements in b to a new block b', delete b, and insert b' in b's place in the linked list. It is straightforward to see that this modification does not change the result in Lemma 4.4.

4.3 Persistent structure

We now show how to make the blocked data structure $\mathcal{L}$ persistent. Due to lack of space, we omit some of the details, which we will provide in the full version of the paper. Driscoll et al. [12] describe general techniques for making an internal memory data structure persistent, and it is easy to use their so-called *node-copying* technique to obtain a (partially) persistent linked list in internal memory under insertions and deletions, so that if a total of K operations and pointer changes are performed on the list, the space used by the whole structure is $O(K)$ (see also [22]). Any "old" version of the list of length L can be traversed in $O(L)$ steps once the head of the list has been located. The head can be located in $O(\log_2 K)$ time using a search tree.

We make our external structure $\mathcal{L}$ persistent using a two-level scheme: (i) We make the linked list of blocks that store $\mathcal{L}$ persistent using the technqique mentioned above for making an internal memory linked-list persistent, except that each element of the list is a block on disk. It follows from Lemma 4.4 and the above discussion that the persistent list uses $O(N/B)$ space and that any "old" version of $\mathcal{L}$ which is L blocks long can be traversed in $O(L)$ I/Os. (ii) We make the individual blocks persistent simply by storing all updates done on a block inside the block. Given a time t and a block b existing at that time, it is then easy to reconstruct the elements actually present in b at time t. The invariant imposed on each block above guarantees that at most $B/2$ updates are performed on the elements in a block from the time it is created until it is deleted, and since each block stores at most $B/2$ elements, we are guaranteed to have room for the updates in the block.

We can now describe, given a query time t, how to report the set of lists stored in $\mathcal{L}$ at time t. First we use $O(\log_B N)$ I/Os to query a B-tree built on the "head blocks" and obtain the first block of $\mathcal{L}(t)$. Then we then traverse $\mathcal{L}$ to obtain the blocks comprising $\mathcal{L}(t)$. The invariants maintained on $\mathcal{L}$ and Lemma 4.2 imply that , (t) can be obtained efficiently from $\mathcal{L}(t)$, as follows: For each major list in , (t), Invariant I implies that while scanning $\mathcal{L}(t)$, we will process the elements of the list in contiguous blocks and in order. For a minor list, Lemma 4.2 implies that all elements in it are stored within a constant number of contiguous blocks of $\mathcal{L}(t)$. Hence, when we process a minor list, we need to read into internal memory only a constant number of blocks to load all the elements of that list, and we can then reconstruct the ordered list in internal memory. Thus, a scan of $\mathcal{L}(t)$ suffices to output , (t), and Lemma 4.4 and the discussion in this section give us our final result.

Theorem 4.5 *A collection of ordered lists , can be maintained on disk under N inserts, deletes, and splits, using $O(N/B)$ blocks, such that , (t), the version of , at time t can be retrieved using $O(\log_B N + |, |/B)$ I/Os.*

5 Conclusions

In this paper we have considered graph-traversal problems motivated by applications in GIS. We believe that the algorithms developed in this paper are of great practical interest. We are currently implementing our algorithms for contour-line extraction in order to verify this belief.

GIS applications are a fertile source of important new problems, especially in the area of I/O-efficient algorithms. One related problem that we mention here is the problem of performing a windowing query using linear space and in $O(\log_B N + |W_\Sigma|/B)$ I/Os.

References

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

[3] L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, University of Aarhus, Denmark, February/August 1996.

[4] L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS.* Springer-Verlag, 1997.

[5] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica, to appear.*

[6] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 560–569, 1996.

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[8] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[9] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, 1997.

[10] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11:121–137, 1979.

[11] M. de Berg, M. van Kreveld, R. van Oostrum, and M. Overmars. Simple traversal of a subdivision without extra storage. In *Proc. 3rd ACM Workshop on Advances in GIS*, pages 77–84, 1995.

[12] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.

[13] H. Edelsbrunner. A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13:209–219, 1983.

[14] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15:317–340, 1986.

[15] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal of Computing*, 16:1004–1022, 1987.

[16] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[17] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Math.*, 36:177–189, 1979.

[18] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3D surface construction algorithm. *Computer Graphics*, 21:163-169, 1987.

[19] J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer-Verlag, 1997.

[20] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16:181–214, August 1996.

[21] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, MA, 1989.

[22] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[23] M. van Kreveld. Efficient methods for isoline extraction from a digital elevation model based on triangulated irregular networks. In *Proc. 6th Int. Symp. on Spatial Data Handling*, pages 835–847. to appear in *Int. J. on GIS*.

[24] M. van Kreveld. Variations on sweep algorithms: Efficient computation of extended viewsheds and classifications. In *Proc. 7th Int. Symp. on Spatial Data Handling*, pages 13A.15–13A.27, 1996.

[25] M. van Kreveld. Digital elevation models: overview and selected TIN algorithms. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer-Verlag, 1997.

[26] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. ACM Annual Symposium on Computational Geometry*, pages 212–220, 1997.

[27] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12:110–147, 1994.