I/O-efficient map overlay and point location in low-density subdivisions

Document status and date:
Published: 01/01/2007

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

Download date: 06. May. 2021

I/O-Efficient Map Overlay and Point Location in Low-Density Subdivisions

Mark de Berg[*]   Herman Haverkort[*]   Shripad Thite[*]   Laura Toma[†]

Abstract

We present improved and simplified I/O-efficient algorithms for two problems on planar low-density subdivisions, namely map overlay and point location. More precisely, we show how to preprocess a low-density subdivision with n edges in $O(sort(n))$ I/O's into a compressed linear quadtree such that one can:

(i) compute the overlay of two such preprocessed subdivisions in $O(scan(n))$ I/O's, where n is the total number of edges in the two subdivisions,

(ii) answer a single point location query in $O(\log_B n)$ I/O's and k batched point location queries in $O(scan(n) + sort(k))$ I/O's.

For the special case where the subdivision is a fat triangulation, we show how to obtain the same bounds with an ordinary (uncompressed) quadtree, and we show how to make the structure fully dynamic using $O(\log_B n)$ I/O's per update. Our algorithms and data structures improve on the previous best known bounds for general subdivisions both in the number of I/O's and storage usage, they are significantly simpler, and several of our algorithms are cache-oblivious.

1 Introduction

The traditional approach to algorithms design considers each atomic operation to take roughly the same amount of time. Unfortunately this simplifying assumption is invalid when the algorithm operates on data stored on disk: reading data from or writing data to disk can be a factor 100,000 or more slower than doing an operation on data that is already present in main memory. Thus, when the data is stored on disk it is usually much more important to minimize the number of disk accesses rather than CPU operations.

This has led to the study of so-called I/O-efficient algorithms, also known as external-memory algorithms. The standard way of analyzing such algorithms is with the model introduced by Aggarwal and Vitter [1]. In this model, a computer has an internal memory of size M and an arbitrarily large disk. The data on disk is stored in blocks of size B, and whenever an algorithm wants to work on data not present in internal memory, the block(s) containing the data are read from disk. The I/O-complexity of an algorithm is the number of I/O's it performs—that is, the number of block transfers between internal memory and disk. Scanning—reading a set of n consecutive items from disk—takes $scan(n) = \lceil n/B \rceil$ I/O's, and sorting takes $sort(n) = \Theta((n/B) \log_{M/B}(n/B))$ I/O's.

One of the main application areas for I/O-efficient algorithms is the area of geographic information systems (GIS), because GIS typically work with massive amounts of data and loading all of it into memory is often infeasible. In GIS data for a particular geographic region is stored as a number of separate thematic layers. There can be a layer storing the road network, a layer storing the river network, a layer storing a subdivision of the region according to land usage or soil type, and so on. To combine information from two such layers—for example to find the crossings between the river network and the road network—one has to compute the overlay of the layers.

Background. The problem of map overlay can be formulated as a red-blue intersection problem: given a set of non-intersecting blue segments and a set of non-intersecting red segments in the plane, compute all intersections between the red and blue segments. Arge et al. show how to do this in $O(sort(n) + k/B)$ I/O's, where k is the number of intersections [2]. This is optimal in the worst case, but it is not satisfactory: the algorithm is complicated and uses $\Theta(n \log_{M/B}(n/B))$ storage. Crauser et al. [4] describe a randomized solution with the same (expected) bound of $O(sort(n) + k/B)$ I/O's and linear space under some realistic assumptions on M, B and n. Whether this algorithm is practical is not clear.

Although the I/O-complexity of the above algorithms is optimal for general sets of line segments, there are important special cases for which this is not clear. For example, in internal memory one can overlay two subdivisions in $O(n+k)$ time when these subdivisions are connected [6]. This brings us to the topic of our paper: is it possible to do the overlay of two planar maps in $O(scan(n+k))$ I/O's?

We will describe solutions based on modifications

of the so-called *linear quadtree*, introduced by Gargantini [8]. The linear quadtree is a quadtree variant where only the leaf regions are stored, and not the internal nodes. To facilitate a search in the quadtree, a linear order is defined on the leaves based on some space-filling curve; then a B-tree is constructed on the leaves using this ordering—see Section 2 for details. The idea of using linear quadtrees to store planar subdivisions has been used by Hjaltason and Samet [9]. They present algorithms for constructing the quadtree and for insertions. Although their experiments indicate their method performs well in practice, the I/O-complexity of their algorithms is not fully understood and does not seem to be worst-case-optimal. Furthermore, the stopping rule for splitting quadtree cells is based on user-defined parameters, so their method is not fully automatic.

Our results. We show how to overcome these disadvantages for two types of subdivisions: fat triangulations and low-density subdivisions [5]. A δ-*fat triangulation* is a triangulation in which every angle is bounded from below by a fixed positive constant δ. A λ-*low-density subdivision* is a subdivision such that any disk D is intersected by at most λ edges whose length is at least the diameter of D, for some fixed constant λ. We believe these two types of subdivisions are representative for most subdivisions encountered in practice, for reasonable values of δ and λ.

We present improved external-memory algorithms for map overlay and point location for such subdivisions. Our results are based on quadtrees which we define to ensure that (i) each leaf intersects only a constant number of edges of the subdivision, (ii) that we create only $O(n)$ leaves, and (iii) that we can construct the leaves efficiently. To store the quadtrees, we combine the ideas of compressed quadtrees and linear quadtrees to get a *linear compressed quadtree*. Our algorithms to construct the quadtrees are simple and elegant—simpler than the algorithm of Hjaltason and Samet [9]—and use only $O(sort(n))$ I/O's.

Our other results then come almost for free: overlaying two subdivisions boils down to a simple merge of the ordered lists of quadtree leaves taking $O(scan(n))$ I/O's, point location can be done in $O(\log_B n)$ I/O's by searching in the B-tree built on top of the list of quadtree leaves, and k batched point location queries can be done in $O(scan(n)+sort(k))$ I/O's by sorting the points along the space-filling curve and merging the sorted list with the list of quadtree leaves. The structure for fat triangulations can be made fully dynamic at the cost of $O(\log_B n)$ I/O's per update.

All our data structures and query algorithms can be made cache-oblivious [7] by plugging in the cache-oblivious variants of the various building blocks used, so that no tuning for B and M is necessary. For triangulations, also the construction and update algorithms can be made cache-oblivious, except that updates then take $O(\log_B n + \frac{1}{B}\log^2 n)$ I/O's.

In this abstract we assume that our inputs are subdivisions of the unit square $[0,1]^2$.

2 Our solution for fat triangulations

Theorem 1 *Let $\mathcal{F}$ be a δ-fat triangulation with n edges. Knowing the memory size M and the block size B, we can construct, in $O(sort(n/\delta^2))$ I/O's, a linear quadtree for $\mathcal{F}$ with $O(n/\delta^2)$ cells such that each cell intersects $O(1/\delta)$ triangles and the total number of intersections between cells and triangles is $O(n/\delta^2)$. With this structure we can do the following:*

(i) **Map overlay:** *Given two δ-fat triangulations with n triangles in total, each stored in such a linear quadtree, we can find all pairs of intersecting triangles in $O(scan(n/\delta^2))$ I/O's.*

(ii) **(Batched) point location:** *for any query point p we can find the triangle of $\mathcal{F}$ that contains p in $O(\log_B n)$ I/O's, and for any set P of k query points we can find for each point $p \in P$ the triangle of $\mathcal{F}$ that contains p in $O(scan(n/\delta^2) + sort(k))$ I/O's.*

(iii) **Updates:** *Inserting a vertex, moving a vertex, deleting a vertex, and flipping an edge can all be done in $O((\log_B n)/\delta^4)$ I/O's.*

The quadtree subdivision for fat triangulations A quadtree is a hierarchical subdivision of the unit square into quadrants, where the subdivision is defined by a criterion to decide what quadrants are subdivided further, and what quadrants are leaves of the hierarchy. A *canonical square* is any square that can be obtained by recursively splitting the unit square into quadrants. For a canonical square σ, let $mom(\sigma)$ denote its parent, that is, the canonical square that contains σ and has twice its width. The leaves of the quadtree form the quadtree subdivision; that is, a *quadtree subdivision* for a set of objects in the unit square is a subdivision into disjoint canonical squares (*quadtree cells*), such that each cell obeys the stopping rule while its parent does not. The stopping rule we use is as follows:

> Stop splitting when all edges intersecting the cell σ under consideration are incident to a common vertex.

Note that the stopping rule includes the case were σ is not intersected by any edges. We can prove that this stopping rule leads to a quadtree subdivision with $O(n/\delta^2)$ cells, such that each cell is intersected by at most $2\pi/\delta$ triangles, and the total number of triangle-cell intersections is $O(n/\delta^2)$.

We store the quadtree subdivision defined above as a linear quadtree [8]. To this end, we define an ordering on the leaf cells of the quadtree subdivision. The

ordering is based on a space-filling curve defined recursively by the order in which it visits the quadrants of a canonical square. We use the z-*order space-filling curve* for this, which visits the quadrants in the order bottom left, top left, bottom right, top right, and within each quadrant, the z-order curve visits its subquadrants recursively in the same order. Since the intersection of every canonical square with this curve is a contiguous section of the curve, this yields a well-defined ordering of the leaf cells of the quadtree subdivision. We call this order the z-order.

The z-order curve not only orders the leaf cells of the quadtree subdivision, but it also provides an ordering for any two points in the unit square—namely the z-order of any two disjoint canonical squares containing the points. (We assume that canonical squares are closed at the bottom and the left side, and open at the top and the right side.) The z-order of two points can be determined as follows. For a point $p = (p_x, p_y)$ in $[0, 1)^2$, define its z-index $z(p)$ to be the value in the range $[0, 1)$ obtained by interleaving the bits of the fractional parts of p_x and p_y, starting with the first bit of p_x. The value $z(p)$ is sometimes called the *Morton block index* of p. The z-order of two points is now the same as the order of their z-indices [9]. The z-indices of all points in a canonical square σ form a subinterval $[z_1, z_2)$ of $[0, 1)$, where z_1 is the z-index of the bottom left corner of σ. Note that any subdivision of the unit square $[0, 1)^2$ into k leaf cells of a quadtree corresponds directly to a subdivision of the unit interval $[0, 1)$ of z-indices into k subintervals.

A simple way of storing a triangulation in a linear quadtree is now obtained by storing all cell-triangle intersections in a B-tree [3]: each cell-triangle intersection $(\sigma, \triangle)$ of a cell σ corresponding to the z-index interval $[z_1, z_2)$ is represented by storing triangle $\triangle$ with key z_1. Thus the leaf cells are stored implicitly: each pair of consecutive different keys z_1 and z_2 constitutes the z-index interval of a quadtree leaf cell.

Building the quadtree   Since the quadtree may have height $\Theta(n)$, a natural top-down construction algorithm could take $\Theta(n^2/B)$ I/O's. Below we describe a faster algorithm that computes the leaf cells that result with our stopping rule directly, using local computations instead of a top-down approach.

For any vertex v of the given triangulation $\mathcal{F}$, let $star(v)$ be the *star* of v in $\mathcal{F}$; namely, it is the set of triangles of $\mathcal{F}$ that have v as a vertex. Recall that a canonical square is any square that can be obtained by recursively subdividing the unit square into quadrants. For a set S of triangles inside the unit square, we say that a canonical square of σ is *active* in S if it lies completely inside S and all edges from S that intersect σ are incident to a common vertex, while $mom(\sigma)$ intersects multiple edges of S that are not all incident to a common vertex. Thus the cells of the

quadtree subdivision we wish to compute for $\mathcal{F}$ are exactly the active canonical squares in $\mathcal{F}$.

Lemma 2 *Let $\triangle uvw$ be a triangle of $\mathcal{F}$ and σ a canonical square that intersects $\triangle uvw$. Then σ is active in $\mathcal{F}$ if and only if σ is active in $star(u)$, $star(v)$ or $star(w)$.*

We now construct the linear quadtree as follows:

1. Compute an adjacency list for each vertex.
2. Scan the adjacency lists for all vertices: for each vertex u load its adjacency list in memory and compute the active cells of $star(u)$, with for each cell σ the triangles that intersect σ. Output each triangle with the key z_1 of the z-index interval $[z_1, z_2)$ that corresponds to σ.
3. Sort the triangles by key, removing duplicates.
4. Build a B-tree on the list of triangles with keys.

This algorithm runs in $O(sort(n/\delta^2))$ I/O's. Note that by Lemma 2, local update operations such as inserting a vertex can be done by computing the structure of the quadtree locally in the area of the update, and determining what the changes entail for the data stored on the disk.

Overlaying maps and point location   Recall that each triangulation's quadtree, or rather, subdivision of the z-order curve, is stored on disk as a sorted list of z-indices with triangles. To overlay the two triangulations, we scan the two quadtrees simultaneously in z-order, reporting, for any pair of intersecting leaf cells, the intersections between the triangles stored with the cells. The input has size $O(n/\delta^2)$. The output consists of $O(n/\delta)$ intersections since a δ-fat triangulation has density $O(1/\delta)$, as shown by De Berg et al. [5]. Thus map overlay takes only $O(scan(n/\delta^2))$ I/O's.

To locate a point p we compute its z-index $z(p)$ and search the B-tree for the triangles with the highest keys less than or equal to $z(p)$. For batched point location, we sort the set P of query points by z-index, and scan the leaves of the B-tree and P in parallel.

3   Our solution for low-density subdivisions

The *density* of a set of line segments in the plane is the smallest number λ such that the following holds: any disk D is intersected by at most λ line segments with length at least the diameter of D. We say that a subdivision $\mathcal{F}$ has density λ if its edge set has density λ.

Theorem 3 *Let $\mathcal{F}$ be a subdivision or a set of non-intersecting line segments of density λ with n edges. Knowing M and B, we can construct, in $O(sort(\lambda n))$ I/O's, a linear compressed quadtree for $\mathcal{F}$ with $O(n)$ cells that each intersect $O(\lambda)$ edges. With this structure we can do the following:*

(i) **Map overlay:** *If we have two subdivisions (or sets of non-intersecting line segments) of density λ with n edges in total, both stored in such a linear compressed quadtree, then we can find all pairs of intersecting edges in $O(scan(\lambda n))$ I/O's.*

(ii) **(Batched) point location:** *for any query point p we can find the face of $\mathcal{F}$ that contains p in $O(\log_B n)$ I/O's, and for any set P of k query points we can find for each point $p \in P$ the face of $\mathcal{F}$ that contains p in $O(scan(\lambda n) + sort(k))$ I/O's.*

Below we explain our data structure. The query algorithms are the same as in the previous section.

The compressed quadtree subdivision for low-density maps Let $\mathcal{G}$ be the set of vertices of the axis-parallel bounding boxes of the edges of $\mathcal{F}$. We construct a quadtree for $\mathcal{F}$ with the following rule:

> Stop splitting when the cell σ under consideration contains at most one point from $\mathcal{G}$.

To be able to bound the number of cells to $O(n)$, we use five-way splits as in a compressed quadtree [10], as follows. Let σ be a canonical square that contains more than one point from $\mathcal{G}$, and let σ' be the smallest canonical square that contains all points of $\sigma \cap \mathcal{G}$. Then σ is split into five regions. The first region is the *donut* $\sigma \setminus \sigma'$. The remaining four regions are the four quadrants of σ'. Note that the first region does not contain any points of $\mathcal{G}$, so it is never subdivided further. We can prove that a quadtree subdivision based on the above stopping rule and five-way splits has $O(n)$ cells, each intersected by at most $O(\lambda)$ edges.

We store the cell-edge intersections of the compressed quadtree subdivision in a list sorted by the z-order of the cells, indexed by a B-tree. The only difference with the previous section is that we now have to deal with donuts as well as square cells. Recall that a canonical square (a square that can be obtained from the unit square by a recursive partitioning into quadrants) corresponds to an interval on the z-order curve. For a donut this is not true. However, a donut corresponds to at most two such intervals, because a donut is the set-theoretic difference of two canonical squares. Thus the solution of the previous section can be applied if we represent each donut by two intervals $[z_1, z_2\rangle$ and $[z_3, z_4\rangle$; edges intersecting the first part of the donut are stored with key z_1 and edges intersecting the second part are stored with key z_3. We merge cells that do not intersect any edge with their immediate successors or predecessors in the z-order. We call the resulting structure—the B-tree on the cell-edge intersections whose keys imply a compressed quadtree subdivision—a *linear compressed quadtree*.

Building the quadtree We construct the leaves of the compressed quadtree, or rather, the corresponding

subdivision of the z-order curve, as follows. We sort $\mathcal{G}$ into z-order, and scan the sorted points. For each pair of consecutive points, say u and v, we construct their lowest common ancestor $lca(u, v)$ by examining the longest common prefix of the bit strings representing $z(u)$ and $z(v)$. We output the five z-indices that bound and separate the z-order intervals of the four children of $lca(u, v)$. To complete the subdivision of the z-order curve, we sort the output into a list by z-order, removing duplicates.

We now build a B-tree on the subdivision of the z-order curve, and distribute the edges of $\mathcal{F}$ to the leaves that intersect them. This is done in $O(sort(\lambda n))$ I/O's in a straightforward top-down manner.

Finally we collect all edge-leaf intersections, ordered by the z-indices of the leaf cells, and put a new B-tree on top of them. Each cell σ without any intersecting edges is merged with the cells that precede or follow it in the z-order.

The complete algorithm runs in $O(sort(\lambda n))$ I/O's.

Acknowledgment

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[2] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. European Sympos. Algorithms*, pages 295–310, 1995.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press / McGraw-Hill, Cambridge, Mass., 2001.

[4] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *Comput. Geom. Theory Appl.*, 11(3):305–337, June 2001.

[5] M. de Berg, M. J. Katz, A. v. Stappen, and J. Vleugels. Realistic input models for geometric algorithms. *Algorithmica*, 34:81–97, 2002.

[6] U. Finke and K. Hinrichs. Overlaying simply connected planar subdivisions in linear time. In *Proc. ACM Symp. Comp. Geom.*, pages 119–126, 1995.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symp. Found. Comp. Sc.*, pages 285–298, 1999.

[8] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[9] G. R. Hjaltason and H. Samet. Speeding up construction of pmr quadtree-based spatial indexes. *VLDB Journal*, 11:190–137, 2002.

[10] H. Samet. *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*. Addison-Wesley, Reading, MA, 1989.