

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

Mark de Berg¹, Herman Haverkort¹, Shripad Thite², and Laura Toma³

¹ Dept. of Computer Science, Eindhoven University of Technology, the Netherlands
mberg@win.tue.nl, cs.herman@haverkort.net

² Center for the Mathematics of Information, California Institute of Technology, USA
shripad@caltech.edu

³ Dept. of Computer Science, Bowdoin College, USA
ltoma@bowdoin.edu

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 $O(\text{sort}(n))$ I/O's into a compressed linear quadtree such that one can:

- (i) compute the overlay of two preprocessed subdivisions in $O(\text{scan}(n))$ 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 $O(\text{scan}(n) + \text{sort}(k))$ 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 to minimize the CPU computation time.

This has led to the study of so-called I/O-efficient algorithms, also known as external-memory or out-of-core algorithms. The by now standard way of analyzing I/O-efficient 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

* MdB and ST were supported by the Netherlands' Organisation for Scientific Research (NWO).

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 the internal memory and the disk. In this model, scanning—reading a set of n consecutive items from disk—can be done in $scan(n) = \lceil n/B \rceil$ 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 has always been 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 the 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 the 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. Even though the map-overlay problem is one of the most basic algorithmic problems in spatial databases and GIS, which are main application areas for I/O-efficient algorithms, the map-overlay problem has still not been solved satisfactorily in the I/O-model.

Background. In this paper we study the following map overlay problem, also known as the 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. [3] showed how to solve this problem in $O(sort(n) + k/B)$ I/O's, where k is the number of intersections. Even though this is optimal in the worst case, it is not satisfactory for several reasons. First, as pointed out by the authors, their solution is complicated. Hence, the practical value of the solution is unclear. Second, the algorithm presented by Arge et al. [3] uses $\Theta(n \log_{M/B} n/B)$ storage (that is, $\Theta(n/B \cdot \log_{M/B} n/B)$ disk blocks). A randomized solution for computing the intersections in a set of line segments is described by Crauser et al. [10]. They give the same expected I/O-bound of $O(sort(n) + k/B)$ I/O's and linear space under some (realistic) assumptions for M, B, n . We do not know whether this algorithm is practical.

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, the overlay of two triangulations that are suitably stored—in a doubly-connected edge list [14], say—can trivially be computed in $O(n+k)$ time in internal memory, which raises the question whether the overlay of two suitably stored triangulations can be computed in $O(scan(n+k))$ I/O's. In fact, in internal memory one can overlay two subdivisions in $O(n+k)$ time when these subdivisions are connected [15]. 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? And can it be done *cache-obliviously* [16], that is, can it be done without specifying the memory size M and the block size B in our algorithms, so that no parameter tuning is necessary and the I/O-behavior is good over all levels in a multilevel memory hierarchy?

Most research into I/O-efficient algorithms has focused on algorithms that are efficient for any worst-case input. However, worst-case inputs are often artificial constructions that do not usually occur in real life. In computational geometry this has led to the study of input models where inputs are assumed to have properties that make them resemble realistic inputs better [13]. The most common assumption in such models is that the objects are *fat*, that is, they are not arbitrarily long and narrow. Fatness has been studied extensively in computational geometry in the recent years, and solutions to many fundamental problems have been improved by exploiting fatness and related notions, see De Berg et al. [13,12] and references therein.

In this paper, we consider two types of subdivisions: fat triangulations and low-density subdivisions. 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 include most subdivisions encountered in practice, for reasonable values of δ and λ .

The data structures on which we will base our solutions are modifications of the so-called *linear quadtree*, which was introduced by Gargantini [17]. 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 [19]. They present algorithms for constructing (or: bulk-loading, as it is often called in GIS) the quadtree, for insertions, and for bulk-insertions. Although their experiments indicate their method performs well in practice, it has several disadvantages. First, the I/O-complexity of their algorithms is analysed in terms of various parameters that depend on the data and the algorithm in a way that is not well-understood. In particular, the performance of their algorithms does not seem to be worst-case optimal. Second, the stopping rule for splitting quadtree cells is based on two user-defined parameters (the maximum depth and a so-called splitting threshold), and so the method is not fully automatic.

Our results. In this paper we show how to overcome these disadvantages for fat triangulations and low-density subdivisions and present improved and simplified algorithms for map overlay and point location in external memory. Our results are based on a quadtree 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.

For fat triangulations our quadtree is defined by recursively splitting the unit square into quadrants until all edges that intersect a cell are incident to a common vertex. We prove that this stopping rule yields a quadtree of linear size. Nevertheless, due to the potentially large depth of the quadtree, it is still difficult or impossible to build the quadtree I/O-efficiently by distributing the edges from the root down into the quadtree while splitting nodes as needed. Fortunately our stopping

rule makes a completely different approach possible: we give an algorithm that is simple and elegant—simpler than the algorithm of Hjaltason and Samet [19]—and uses only $O(\text{sort}(n))$ I/O's.

For low-density subdivisions we continue splitting until each cell contains only a single bounding-box vertex of any edge. This stopping rule leads to cells with a constant number of edges, but the number of cells cannot be bounded. Therefore we combine the ideas of compressed quadtrees and linear quadtrees to get a *linear compressed quadtree*, rather than a regular quadtree. We show that with the stopping rule just defined, the compressed quadtree has linear size. We also give a construction algorithm that uses only $O(\text{sort}(n))$ I/O's.

Once we have proved that these quadtrees have linear size and each leaf region intersects a constant number of edges, our other results come almost for free: overlaying two subdivisions boils down to a simple merge of the ordered lists of quadtree leaves taking $O(\text{scan}(n))$ 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 performing k batched point location queries can be done in $O(\text{scan}(n) + \text{sort}(k))$ I/O's by sorting the points along the space-filling curve and merging the sorted list with the list of quadtree leaves. The results for map overlay apply to pairs of fat triangulations, low-density subdivisions, or low-density sets of line segments, as well as to mixed pairs of maps of these types. The structure for fat triangulations can be made fully dynamic at a cost of $O(\log_B n)$ I/O's per update.

An optimal static structure for point location in general planar subdivisions was already given by Goodrich et al. [18] for the standard I/O-model and by Bender et al. [5] for the cache-oblivious model. Batched point location can be done with $O(\text{sort}(n+k))$ I/O's in the I/O-model using the algorithm by Arge et al. [3]. The result on dynamization, however, is new as far as we know: the best known dynamic I/O-efficient point location structures use $O(\log_B^2 n)$ I/O's per query [2] in the I/O-model. All our data structures and query algorithms are cache-oblivious. Our construction and update algorithms for triangulations can be made cache-oblivious, except that updates will then take $O(\log_B n + \frac{1}{B} \log^2 n)$ I/O's. These results constitute the first results for cache-oblivious map overlay, batched point location and dynamic point location.

We omit all proofs from this extended abstract; please refer to the full paper for the proofs.

2 Fat Triangulations

In this section we describe our solution for fat triangulations. A δ -fat triangulation is a triangulation consisting of δ -fat triangles, that is, triangles all of whose angles are at least δ for some fixed constant $\delta > 0$. We assume that $B = \Omega(1/\delta)$ and $M = \Omega(1/\delta^3)$. We assume that all triangulations are triangulations of the unit square $[0, 1]^2$. (Our algorithms and proofs extend to triangulations of convex regions—we leave the details for the full paper.) We can show the following:

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 $O(\text{sort}(n/\delta^2))$ I/O's, a linear*

quadtrees for \mathcal{F} that stores $O(n/\delta^2)$ cell-edge intersections. With this structure we can perform the following operations:

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

In the cache-oblivious model the same bounds hold, except that updates then take $O((\log_B n + \frac{1}{B} \log^2 n)/\delta^4)$ I/O's.

2.1 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 $\text{mom}(\sigma)$ 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:

Stopping rule for fat triangulations: 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 where σ is not intersected by any edges. We can prove the following:

Lemma 1. *Let \mathcal{F} be a δ -fat triangulation of the unit square with n edges. Then the stopping rule defined above leads to a quadtree subdivision with $O(n/\delta)$ cells, each intersected by at most $2\pi/\delta$ triangles.*

2.2 Storing the Quadtree Subdivision and the Triangulation

We will store the quadtree subdivision defined above as a so-called *linear quadtree* [17]. 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 will 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 sub-quadrants 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 the resulting 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 [19]. 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 (but novel) way of storing a triangulation in a linear quadtree is now obtained by storing all cell-triangle intersections in a B-tree [9]: each cell-triangle intersection (σ, Δ) of a cell σ corresponding to the z-index interval $[z_1, z_2)$ is represented by storing triangle Δ with key z_1 . With this way of storing the linear quadtree, the leaf cells of the quadtree are stored implicitly: each pair of consecutive different keys z_1 and z_2 constitutes the z-index interval of a quadtree leaf cell. For a cache-oblivious solution we can use a cache-oblivious B-tree [5,6,7].

In the remainder we will sometimes need to compute or compare z-indices. Whether this takes constant time depends on the operations allowed by the model of computation. In any case, since we care mainly about I/O-efficiency, such computations do not effect the analysis of our algorithms.

2.3 Building the Quadtree I/O-Efficiently

The natural algorithm to build a quadtree would take a set of triangles and a canonical square (initially all triangles and the unit square) as input, check if the condition of the stopping rule is satisfied, and if not, distribute the triangles among the four children and subdivide the children recursively. Unfortunately, this algorithm takes $O(n^2)$ time and $O(n^2/B)$ I/O's, as the quadtree may have height $O(n)$. 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 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} . We can prove the following:

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

On the basis of this lemma we can 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 (cache-oblivious) B-tree on the list of triangles with their keys.

Lemma 3. *The quadtree described above for a δ -fat triangulation with n edges can be constructed with $O(\text{sort}(n/\delta^2))$ I/O's.*

Eliminating superfluous cells. The I/O-complexity of the construction and the storage requirements in practice can be reduced with an easy optimization: we merge all active cells that lie properly inside triangles with their successors or predecessors in the z-order. In fact, in step 2 of the algorithm, we will not even output such cells. Instead we only output triangle-key pairs for triangle-cell intersections such that an edge of the triangle intersects the cell. We sort these triangle-key pairs, and then scan them. Whenever two consecutive triangles have different keys z_1 and z_2 , we identify the most significant bit that differs between them. Let z be the lowest z-index in $[z_1, z_2]$ for which this bit has value 1. For each triangle stored with key z_2 , we now replace its key by z . Thus all cells with z-indices in $[z_1, z)$ are merged with each other, and all cells with z-values in $[z, z_2]$ are merged with each other.

Since the interval $[z_1, z)$ of the z-order curve covers a connected area in the plane, all cells in the range $[z_1, z)$ that do not intersect any edge must be completely contained in a triangle that already intersects the cell that starts with z-index z_1 . Similarly, $[z, z_2]$ and the cell that starts with z_2 together cover a connected area in the plane, so the triangles that intersect it must have been stored with key z_2 already. Hence no more triangles need to be stored as a result of merging cells.

Updates. We support the following operations: inserting a vertex, moving a vertex, deleting a vertex, and flipping an edge. By Lemma 2, all leaf cells that intersect a triangle $\Delta = (u, v, w)$ can be computed from the local quadtrees of $star(u)$, $star(v)$ and $star(w)$. Since the size of $star(u)$, $star(v)$ and $star(w)$ is $O(1/\delta)$, by Lemma 1 the total number of cells that intersect Δ is $O(1/\delta^3)$. Since each of the supported operations changes only $O(1/\delta)$ triangles, we can compute the structure of the quadtree locally in the area of the update, and determine what the

changes entail for the data stored on the disk. All changes can thus be made in $O((\log_B n)/\delta^4)$ I/O's when a normal B-tree is used, and in $O((\log_B n + \frac{1}{B} \log^2 n)/\delta^4)$ I/O's when a cache-oblivious B-tree is used [5,6,7].

2.4 Overlaying Maps and Point Location

Lemma 4. *The linear quadtree for δ -fat triangulations as described above supports map overlay in $O(\text{scan}(n/\delta^2))$ I/O's, and point location in $O(\log_B(n/\delta))$ I/O's, where n is the number of points in the triangulation. Batched point location for k points takes $O(\text{scan}(n/\delta^2) + \text{sort}(k))$ I/O's.*

Proof. 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 will scan the two quadtrees simultaneously in Z-order, at any point keeping in memory the triangles stored with the last key read from the first list and those stored with the last key read from the second list. Starting from the beginning of the lists, we repeat the following until both lists have been read completely: we read the next key from the list with the smallest unread key, we load all triangles stored with that key into memory, and we compute the intersections with the triangles in memory that were read from the other list. The input has size $O(n/\delta^2)$. The output consists of $O(n/\delta)$ intersections since a δ -fat triangulation has density $O(1/\delta)$ [13], which implies the claim.

To perform point location with a point p , we compute the Z-index $Z(p)$ of p and search the B-tree for the triangles with the highest keys less than or equal to $Z(p)$. To do 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 (similar to the overlay operation as described above). \square

3 Low-Density Subdivisions

In this section we describe our solution for storing planar low-density subdivisions. For a planar object o , let $\text{diam}(o)$ denote the diameter of the smallest enclosing disk of o . The *density* of a set S of objects in the plane is the smallest number λ such that the following holds: any disk D is intersected by at most λ objects $o \in S$ such that $\text{diam}(o) \geq \text{diam}(D)$ [13]. We say that a planar subdivision \mathcal{F} has density λ if its edge set has density λ . In other words, any disk D is intersected by at most λ edges whose length is at least the diameter of D . We assume that $B = \Omega(\lambda)$, and that the input lies in the unit square $[0, 1]^2$. In this section we describe the following result.

Theorem 2. *Let \mathcal{F} be a subdivision or a set of non-intersecting line segments of density λ with n edges. Knowing the memory size M and the block size B , we can construct, in $O(\text{sort}(\lambda n))$ 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 perform the following operations:*

- (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 $O(\text{scan}(\lambda n))$ 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 $O(\text{scan}(\lambda n) + \text{sort}(k))$ I/O's.*

The data structure, the overlay algorithm and the query algorithms are cache-oblivious. (The algorithm that constructs the data structure is not.)

Any set of disjoint δ -fat triangles in the plane has density $O(1/\delta)$ [13]. Thus the results of this section can be used for δ -fat triangulations. However, the solution from the previous section is simpler and dynamic.

Below we explain our data structure, and how to construct it. The query algorithms are the same as described in the previous section.

3.1 The Compressed Quadtree Subdivision for Low-Density Maps

Let \mathcal{F} be a subdivision of the unit square with n edges and of density λ . In general it is impossible to construct a standard quadtree on \mathcal{F} consisting of a linear number of cells that are each intersected by a constant number of edges. Indeed, in a general subdivision of the unit square there can be many vertices arbitrarily close together, even if the subdivision has constant density. To overcome this problem we shall use a so-called *compressed quadtree*.

Let \mathcal{G} be the set of vertices of the axis-parallel bounding boxes of the edges of \mathcal{F} . This set has a nice property:

Lemma 5 ([11]). *Any square σ that does not contain any bounding-box vertex of an object in a set S with density λ , intersects $O(\lambda)$ objects from S .*

We now construct a quadtree for \mathcal{F} with the following stopping rule.

Stopping rule for low-density subdivisions: Stop splitting when the cell σ under consideration contains at most one point from \mathcal{G} .

Consider the quadtree that we get from this stopping rule. Its cells intersect $O(\lambda)$ edges, but the number of cells cannot be bounded. Hence, we compress the quadtree [20], by building it as follows. We recursively subdivide each canonical square σ that contains more than one point from \mathcal{G} into *five* regions. Let σ' be the smallest canonical square that contains all points of $\sigma \cap \mathcal{G}$. 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. When $\sigma' = \sigma$, the first region is skipped; when σ' is smaller than σ , we call $\sigma \setminus \sigma'$ a *proper donut*.

Lemma 6. *Let \mathcal{F} be a subdivision of the unit square with n edges and of density λ . Then a compressed quadtree subdivision based on the stopping rule defined above has $O(n)$ cells, and each cell is intersected by at most $O(\lambda)$ edges.*

3.2 Storing the Compressed Quadtree Subdivision and the Low-Density Map

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 (cache-oblivious) 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 $[z_1, z_2)$ and $[z_3, z_4)$; 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 . As described in Section 2.3, 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*. Map overlay and point location are done in exactly the same way as with the linear quadtree described earlier.

3.3 Building the Compressed Quadtree I/O-Efficiently

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 \mathcal{L} by z-order, removing duplicates.

Lemma 7. *The above algorithm generates a subdivision of the z-order curve that corresponds to a compressed quadtree on \mathcal{G} in $O(\text{sort}(n))$ I/O's.*

Having constructed the compressed quadtree subdivision, we now distribute the edges in \mathcal{F} to the faces of the quadtree subdivision, or rather, to the corresponding sections of the z-order curve.

To do so, we first build a B-tree on the subdivision of the z-order curve as computed above. We then load the (roughly) M/B nodes of the B-tree, that reside $\log_B(M/B)$ levels below the root, in memory. Note that each of these nodes covers a certain section of the z-order curve, and together they form a subdivision of the z-index interval covered by the root. We assign an output stream to each of these nodes, and reserve a buffer of one block in memory for each of them. We now read the edges from the input one by one, and distribute each edge to the output streams of the nodes whose section of the z-order curve is intersected by the edge. Note that each edge may be copied to several streams. Once all edges have been

read, we distribute the edges in each node's stream recursively into the subtree rooted at that node.

After distributing all edges recursively to the leaves, we collect all edge-cell intersections, ordered by the z -indices of the 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, up to a cell that stores an edge of the face of \mathcal{F} that contains σ (see Section 2.3 for an explanation).

Lemma 8. *The compressed quadtree as defined above for a subdivision of density λ with n edges can be constructed with $O(\text{sort}(\lambda n))$ I/O's.*

4 Conclusions

We described how one can efficiently store and overlay planar maps in the I/O-model of computation. Our algorithms work for planar maps that are fat triangulations or have low density. The solution for triangulations is based on quadtrees, is considerably simpler than previous solutions, and supports even dynamic maintenance of the planar maps under updates. The second construction, for low density planar maps, is based on compressed quadtrees and is somewhat more complicated; however our analysis gives a better dependency on the parameter that describes the input. Unfortunately it is not clear if the construction algorithm can be made cache-oblivious and if the structure can be made to support updates.

Both constructions use linear space, improving on the previous space bound of $O(n \log_{M/B} n)$ of Arge et al. [3]. Which of our two structures would give the most compact data structure for triangulations in practice remains to be seen. The first approach's dependency on the fatness may be better than it seems (perhaps an analysis in terms of *average* fatness is possible), while the second approach may introduce many guards (a triangulation of n vertices has roughly $3n$ edges and thus roughly $6n$ extra bounding box vertices as guards).

Our data structures can be used for range searching queries. In general this would not be very efficient, but we believe it is possible to achieve good bounds for *approximate range searching* [4]. However, the data structure for low-density subdivisions as presented in this paper does not give good bounds immediately, and needs to be subjected to some post-processing for this purpose. The post-processing also reduces the size of the data structure to $O(n)$, independent of the density λ . We are currently working out the details.

Acknowledgement. We thank Sariel Har-Peled for his extensive contribution.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. *Commun. ACM* 31, 1116–1127 (1988)
2. Arge, L., Vahrenhold, J.: I/O-efficient dynamic planar point location. In: Proc. 16th Annu. ACM Symp. Comput. Geom., pp. 191–200 (2000)

3. Arge, L., Vengroff, D.E., Vitter, J.S.: External-memory algorithms for processing line segments in geographic information systems. In: Proc. 3th Annu. European Sympos. Algorithms, pp. 295–310 (1995)
4. Arya, S., Mount, D.M.: Approximate range searching. *Comput. Geom. Theory Appl.* 17, 135–152 (2000)
5. Bender, M., Cole, R., Raman, R.: Exponential structures for efficient cache-oblivious algorithms. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 195–207. Springer, Heidelberg (2002)
6. Bender, M.A., Demaine, E.D., Farach-Colton, M.: Cache-oblivious B-trees. In: Proc. 41th Annu. IEEE Symp. Found. Comput. Sci., pp. 339–409 (2000)
7. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache oblivious search trees via binary trees of small height. In: Proc. 13th ACM-SIAM Symp. Discrete Algorithms, pp. 39–48 (2002)
8. Chiang, Y.-J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proc. 6th ACM-SIAM Symp. Discrete Algorithms, pp. 139–149 (1995)
9. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*. MIT Press / McGraw-Hill, Cambridge, Mass (2001)
10. Crauser, A., Ferragina, P., Mehlhorn, K., Meyer, U., Ramos, E.: Randomized external-memory algorithms for some geometric problems. *Comput. Geom. Theory Appl.* 11(3), 305–337 (2001)
11. de Berg, M.: Linear size binary space partitions for uncluttered scenes. *Algorithmica* 28, 353–366 (2000)
12. de Berg, M.: Improved bounds on the union complexity of fat objects. In: Proc. 25th Conf. Found. Soft. Tech. Theoret. Comput. Sci., pp. 116–127 (2005)
13. de Berg, M., Katz, M.J., Stappen, A.v., Vleugels, J.: Realistic input models for geometric algorithms. *Algorithmica* 34, 81–97 (2002)
14. de Berg, M., van Kreveld, M., Overmars, M.H., Schwarzkopf, O.: *Computational Geometry: Algorithms and Applications*, 2nd edn. Springer, Heidelberg (2000)
15. Finke, U., Hinrichs, K.: Overlaying simply connected planar subdivisions in linear time. In: Proc. 11th Annu. ACM Symp. Comput. Geom., pp. 119–126 (1995)
16. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Annu. IEEE Symp. Found. Comput. Sci., pp. 285–298 (1999)
17. Gargantini, I.: An effective way to represent quadtrees. *Commun. ACM* 25(12), 905–910 (1982)
18. Goodrich, M.T., Tsay, J.-J., Vengroff, D.E., Vitter, J.S.: External-memory computational geometry. In: Proc. 34th Annu. IEEE Symp. Found. Comput. Sci., pp. 714–723 (1993)
19. Hjaltason, G.R., Samet, H.: Speeding up construction of pmr quadtree-based spatial indexes. *VLDB Journal* 11, 137–190 (2002)
20. Samet, H.: *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*. Addison-Wesley, Reading, MA (1989)