I/O-Efficient Algorithms for Computing Planar Geometric Spanners*

Anil Maheshwari† Michiel Smid† Norbert Zeh‡

January 19, 2007

Abstract

We present I/O-efficient algorithms for computing planar Steiner spanners for point sets and sets of polygonal obstacles in the plane.

Keywords: External-memory algorithms, computational geometry, geometric spanners, shortest paths.

1 Introduction

Geometric spanners are sparse subgraphs of the complete Euclidean graph over a set of points or vertices of obstacles. They have played a key role in efficient algorithmic solutions for several fundamental geometric problems including shortest path computations; the design of fault-tolerant networks; computing nearest neighbours, closest pairs, and approximate Euclidean minimum spanning trees; and n-body simulation. A number of efficient algorithms for constructing a variety of spanners in Euclidean space have been proposed (see [17, 25, 28] for an overview). These algorithms are designed in the RAM model, which assumes that the cost of a memory access is independent of the accessed memory location. However, large data sets are unlikely to fit into the internal memory of a computer; in such a situation, the cost of a memory access varies dramatically depending on whether the accessed data item is stored in memory or on disk. Hence, the number of disk accesses is the dominating factor that determines the running time of an algorithm on a massive data set. In [19, 23], I/O-efficient algorithms for computing Euclidean spanners have been proposed. However, once the spanner has been computed, one is usually interested in answering short(est)-path queries on the computed spanner. It is not known how to do this I/O-efficiently for the spanners constructed in [19, 23]. The best bound for reporting a path in the spanner of a

*Research supported by NSERC, GEOIDE NCE and CFI. A preliminary version of this paper appeared in the Workshop on Algorithms and Data Structures, Providence, RI, 2001.

†School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, ON, Canada K1S 5B6. {maheshwa,michiel}@scs.carleton.ca.

‡Faculty of Computer Science, 6050 University Ave, Halifax, NS, Canada B3H 1W5. nzeh@cs.dal.ca.

point set constructed in [23] is $\mathcal{O}(\mathrm{sort}(N))$[1] I/Os. For sets of obstacles in the plane and the spanners of [19], one has to revert to using a shortest-path algorithm for general graphs; but none of the existing shortest-path algorithms are I/O-efficient for sparse graphs. In this paper, we propose I/O-efficient algorithms for computing *planar* Steiner spanners for point sets and sets of polygonal obstacles in the plane. Planarity is desirable because I/O-efficient shortest-path algorithms [3] and data structures for answering shortest-path queries on planar graphs [5, 21] exist. This is the main motivation for studying planar geometric spanners in the I/O setting.

1.1 Model of Computation and Previous Work

The most widely accepted model for the design and analysis of I/O-efficient algorithms is the I/O-model of Aggarwal and Vitter [1]. In this model, the computer is assumed to be equipped with two levels of memory. The *internal memory* (or *memory* for short) is of bounded size, capable of holding M data items. The disk-based *external memory* is of conceptually unlimited size and is divided into blocks of B consecutive data items. All computation has to happen on data in internal memory. Data are moved between internal and external memory using *I/O-operations* (or *I/Os* for short), each of which transfers one block of data between the two memory levels. The complexity of an algorithm is the number of I/O-operations it performs. Our algorithms are designed and analyzed in the I/O-model.

For surveys of results obtained in the I/O-model and its extensions, refer to [24, 30]. Next we discuss the results that are relevant to our work.

It is shown in [1] that sorting an array of size N takes $\mathrm{sort}(N) = \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os in the I/O-model; scanning an array of size N takes $\mathrm{scan}(N) = \Theta(N/B)$ I/Os. A wide variety of problems whose solution takes $\Omega(N \log N)$ time in the RAM model have an $\Omega(\mathrm{sort}(N))$ lower bound in the I/O-model [4], including a wide range of geometric problems.

Some of our constructions will use the buffer tree [2], which is an extension of the well-known B-tree data structure [8]. The buffer tree outperforms the B-tree in applications where a large number of updates (insert/delete operations) and queries need to be performed and immediate query responses are not required. In particular, processing a sequence of N updates and queries takes $\mathcal{O}\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right) = \mathcal{O}(\mathrm{sort}(N))$ I/Os, while the same would take $\mathcal{O}(N \log_B N)$ I/Os using a B-tree.

Our paper deals with external-memory algorithms for constructing geometric spanners. Next we outline some of the key results in internal and external-memory in this area. A *g*eometric t-spanner is a graph that approximates the complete Euclidean graph of a point set or the visibility graph of a set of polygonal obstacles so that the distances between points or obstacle vertices are preserved up to a constant factor t. The value t is called the *s*panning ratio of the spanner. The concept of geometric spanner graphs has been introduced by Chew [14]. Keil and Gutwin [22] prove that the spanning ratio of the Delaunay triangulation is no more than $\frac{2\pi}{3\cos(\pi/6)} \approx 2.42$. Unfortunately, by a result of [14], the Delaunay triangulation cannot be used when a spanning ratio arbitrarily close to 1 is desired. In fact, it is easy to show that there are point sets such that no planar graph over such a point set has a

[1]$\mathrm{sort}(N) = \Theta\left(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$, where the parameters N, M and B are explained in Section 1.1.

spanning ratio less than $\sqrt{2}$. The first to show how to construct a t-spanner in the plane, for t arbitrarily close to one, were Keil and Gutwin [22]. Independently, Clarkson [15] discovered the same construction for two and three dimensions. Ruppert and Seidel [26] generalize the result to higher dimensions, using the construction of a θ-frame due to Yao [31]; their algorithm takes $\mathcal{O}\left(N \log^{d-1} N\right)$ time. Vaidya [29] and Salowe [27] were the first to give optimal $\mathcal{O}(N \log N)$-time algorithms for constructing t-spanners in higher dimensions. Their algorithms use hierarchical subdivisions similar to that induced by a fair split tree [11]. Consequently, the well-separated pair decomposition of [9, 12] can also be used to obtain an $\mathcal{O}(N \log N)$-time algorithm for constructing t-spanners [10]. Clarkson [15] shows that a modification of the θ-graph for sets of polygonal obstacles is a t-spanner for the visibility graph of these obstacles. Arikati et al. [7] show how to construct a planar t-spanner among obstacles that contains only $\mathcal{O}(N)$ additional vertices, called *Steiner points*. Both constructions [7, 15] take $\mathcal{O}(N \log N)$ time.

Eppstein [17], Narasimhan and Smid [25] and Smid [28] present surveys of results on spanners and proximity problems. For a comprehensive discussion of the WSPD and its applications, we refer the reader to [13].

In external memory, efficient solutions to a wide range of geometric problems have been obtained in the last decade. Related to the computation of geometric spanners, we observe that the algorithms of [16, 18] for computing the convex hull of a point set in three dimensions can be used to obtain the Voronoi diagram of a point set in two dimensions in $\mathcal{O}(\text{sort}(N))$ I/Os; the Delaunay triangulation of the point set can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os from the Voronoi diagram. In [19], it is shown how to compute a fair-split tree and a well-separated pair decomposition (WSPD) of a point set in d dimensions; the algorithm takes $\mathcal{O}(\text{sort}(N))$ I/Os and uses $\mathcal{O}(N/B)$ blocks of external-memory. A linear-size t-spanner, for any $t > 1$, can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os from the WSPD. By choosing spanner edges carefully, the spanner can be guaranteed to have spanner diameter $2 \log N$. It is also shown in [19] that $\Omega(\min\{N, \text{sort}(N)\})$ I/Os are required to compute any linear-size t-spanner of a given point set, for any $t > 1$. In [23], it is shown how to compute t-spanners for sets of polygonal obstacles in $\mathcal{O}(\text{sort}(N))$ I/Os.

1.2 New Results

The main result of this paper is an $\mathcal{O}(\text{sort}(N))$-I/O algorithm to construct a planar L_1-Steiner spanner of size $\mathcal{O}(N)$ and spanning ratio $1+\epsilon$ for any set of polygonal obstacles in the plane with N vertices and any $\epsilon > 0$. Using this result we obtain $\mathcal{O}(\text{sort}(N))$-I/O algorithms for constructing planar L_2-Steiner spanners of size $\mathcal{O}(N)$ for point sets and sets of polygonal obstacles in the plane.

The solution presented here follows the framework of the algorithm proposed in [7]. However, the details differ substantially. In particular, the data structure used to maintain the sweep-line status in the algorithm for constructing a planar L_1-Steiner spanner differs considerably from the one used in [7] and also leads to a simplified internal memory algorithm[2].

[2]Most of the proofs in [7] were omitted due to the limited space in the conference proceedings. Interested readers may to refer to Zeh [32] for proofs of some of the critical lemmas stated in [7].

The computed spanners have linear size and can be computed in linear space, except for the solution of the endpoint dominance problem, which we invoke as a subroutine in a number of places. The currently best algorithm for this problem uses $\mathcal{O}\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$ space.

1.3 Organization of the Paper

In Section 2 we introduce necessary definitions and terminology, as well as the relevant results and concepts from [7] and [19] used in this paper. In Section 3 we present an I/O-efficient algorithm for constructing a planar L_1-Steiner spanner for any set of polygonal obstacles in the plane. Algorithms for constructing planar L_2-Steiner spanners for point sets and sets of polygonal obstacles in the plane are presented in Section 4.

2 Preliminaries

This section introduces the necessary notation and terminology and provides some algorithmic background needed by our algorithms. In Section 2.1 we provide necessary definitions. In Section 2.2 we define the fair split tree for a point set. In Sections 2.3 and 2.4 we discuss internal-memory algorithms by Arikati et al. [7] for computing planar L_1-Steiner spanners for point sets and sets of obstacles, respectively, on which our algorithms are based.

2.1 Definitions

A *geometric graph* $G = (S, E)$ over a point set S in the plane is a graph with vertex set S and edge set E; in particular, every vertex in G has a location in the plane, and every edge is a straight line segment between its endpoints. A geometric graph $G = (V, E)$ is a *t-Steiner spanner* for the complete graph $\mathcal{E}(S)$ on a set S of points in the plane if $S \subseteq V$ and, for every pair of vertices $p, q \in S$, $\mathrm{dist}_G(p, q) \leq t \cdot \mathrm{dist}(p, q)$, where $\mathrm{dist}(p, q)$ denotes the distance between p and q in an appropriate metric. The vertices in $V \setminus S$ are called *Steiner points*. The *visibility graph* $\mathcal{V}(P)$ of a set P of polygonal obstacles with vertex set S is the subgraph of $\mathcal{E}(S)$ that contains only those edges that do not cross the interior of any obstacle. A geometric graph $G = (V, E)$ is a t-Steiner spanner for $\mathcal{V}(P)$ if $S \subseteq V$, no edge in G crosses the interior of any obstacle in P, and, for every pair of vertices $p, q \in S$, $\mathrm{dist}_G(p, q) \leq t \cdot \mathrm{dist}_{\mathcal{V}(P)}(p, q)$.

For a given point set $S \subset \mathbb{R}^2$, the *bounding rectangle* $R(S)$ is the smallest rectangle containing all points in S, where a *rectangle* R is the Cartesian product $[x_1, x_1'] \times [x_2, x_2']$ of two closed intervals. The *length* of R in dimension i is $\ell_i(R) = x_i' - x_i$. The minimum and maximum lengths of R are $\ell_{\min}(R) = \min\{\ell_1(R), \ell_2(R)\}$ and $\ell_{\max}(R) = \max\{\ell_1(R), \ell_2(R)\}$. We call R a *box* if $\ell_{\max}(R) \leq 3\ell_{\min}(R)$. If all lengths of R are equal, R is a *square* and we denote its side length by $\ell(R) = \ell_{\min}(R) = \ell_{\max}(R)$. For a point set S, let $\ell_i(S) = \ell_i(R(S))$, $\ell_{\min}(S) = \ell_{\min}(R(S))$, and $\ell_{\max}(S) = \ell_{\max}(R(S))$.

2.2 Fair Split Tree

A *split* of a point set S is a partition of S into two non-empty point sets lying on either side of a line perpendicular to one of the coordinate axes and not intersecting any points in S. A *split tree* T of S is a binary tree over S defined as follows: If $S = \{x\}$, T contains a single node $\{x\}$. Otherwise, perform a split to partition S into two subsets S_1 and S_2. Tree T is now constructed from two split trees for point sets S_1 and S_2 whose roots are the children of the root node S of T. Note that we use the same notation to refer to subsets of S and the nodes in T that represent them. For a node A in T, the *outer rectangle* $\hat{R}(A)$ is defined as follows: For the root S, let $\hat{R}(S)$ be the square with side length $\ell(\hat{R}(S)) = \ell_{\max}(S)$, centered at the center of $R(S)$. For all other nodes A, the hyperplane used for the split of the parent of A, $p(A)$, divides $\hat{R}(p(A))$ into two open boxes. Let $\hat{R}(A)$ be the one that contains (the set) A. A *fair split* of A is a split of A where the line splitting A is at distance at least $\ell_{\max}(A)/3$ from each of the two sides of $\hat{R}(A)$ parallel to it. A split tree formed using only fair splits is called a *fair split tree.*

The following are alternative, more restrictive, definitions of outer rectangles and fair splits which in particular ensure that all outer rectangles are boxes. The algorithm of [19] for constructing a fair split tree makes sure that the splits satisfy these conditions. For the root S of T, the outer rectangle $\hat{R}(S)$ is defined as above. Given the outer rectangle $\hat{R}(A)$ of a node, a split is fair if it splits $\hat{R}(A)$ perpendicular to its longest side, and the splitting line has distance at least $\frac{1}{3}\ell_{\max}(\hat{R}(A))$ from the two sides of $\hat{R}(A)$ parallel to it. Let $\tilde{R}(A_1)$ and $\tilde{R}(A_2)$ be the two rectangles produced by this split. We call $\tilde{R}(A_1)$ and $\tilde{R}(A_2)$ the *split rectangles* of A_1 and A_2, respectively. For $i \in \{1, 2\}$, let $R'_i = \tilde{R}(A_i)$. The following procedure defines $\hat{R}(A_i)$: If R'_i can be split fairly, let $\hat{R}(A_i) = R'_i$. Otherwise, split R'_i perpendicular to its longest side so that the splitting line has distance at least $\frac{1}{3}\ell_{\max}(R'_i)$ from the two sides of R'_i parallel to it. Only one of the two resulting rectangles is non-empty. Repeat the process after replacing R'_i with this non-empty rectangle. The following theorem states that a fair split tree can be computed I/O-efficiently.

Theorem 2.1 (Govindarajan et al. [19]) *Given a set S of N points in the plane, a fair split tree T of S can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space.*

2.3 Planar Steiner Spanner for a Point Set

Let S be the given point set and let C be a minimal square that contains all points of S. The construction of a Steiner spanner for S uses a subdivision of C, denoted by D', into $\mathcal{O}(N)$ regions of two types: *box cells* and *donut cells.* A box cell is a box and contains exactly one point in S. A donut cell is the set-theoretic difference $Z = R \setminus R'$ of two boxes R and R' with the following properties: (1) Z does not contain any point in S. (2) Box R' is contained in box R. (3) The distance of each side e' of R' from the corresponding side of R is either zero or at least $\|e'\|/3$.

Such a subdivision $D' = D'(T)$ can be obtained quite naturally from a fair split tree T of S: For every leaf p of T, we add the split rectangle $\tilde{R}(p)$ as a (box) cell to $D'(T)$. For every internal node A of T with $\hat{R}(A) \neq \tilde{R}(A)$, we add the region $\tilde{R}(A) \setminus \hat{R}(A)$ as a (donut) cell to $D'(T)$.

Lemma 2.1 (Arikati et al. [7]) *The collection $D'(T)$ defines a subdivision of C whose regions are either box or donut cells.*

Given the subdivision D' as defined above, a planar L_1-spanner D'' for point set S can be constructed in internal-memory as follows: Let $\textsc{Interval}(e, r)$ be a procedure that partitions segment e into a minimal number of subsegments of length at most r by adding equally spaced Steiner points on edge e. We perform $\textsc{Interval}(e, \gamma\|e\|)$, for every boundary edge e of a cell in the subdivision, where $\gamma = \epsilon/3$ and $t = 1 + \epsilon$ is the spanning ratio we want D'' to have. For every cell R and every boundary edge e of R, we shoot rays orthogonal to e from the endpoints of e and the Steiner points on e toward the interior of R until they meet another boundary edge of R. For every box cell R containing a point $p \in S$, we also shoot rays from p in all four axis-parallel directions until they meet the boundary of R. To preserve the planarity of D'', we introduce all intersection points between perpendicular rays as Steiner points. Then we add all ray segments between consecutive intersection points to the edge set of D''. Arikati et al. [7] show that D'' is the desired spanner.

Lemma 2.2 (Arikati et al. [7]) *Graph D'' has size $\mathcal{O}(N/\gamma^2)$ and L_1-spanning ratio at most $1 + 3\gamma$.*

2.4 Planar Steiner Spanner for a Set of Obstacles

Let P be a set of polygonal obstacles in the plane. First we define a planar subdivision D_1 so that we can obtain an L_1-spanner of P by partitioning the cells of this subdivision as in Section 2.3 and removing all edges that are inside obstacles. It is not hard to see that such a subdivision D_1 can be obtained by superimposing the subdivision D defined by the obstacles in P and the subdivision D' obtained from the set of obstacle vertices using the construction in Section 2.3. Unfortunately, D_1 may have size $\Omega(N^2)$, thereby leading to a spanner of super-linear size. In [7], it is shown that another subdivision D_2 with the desired properties can be derived from D_1 by removing edges between cells. This subdivision has linear size and can be constructed without constructing D_1 explicitly. In this section, we recall the definitions of D_1 and D_2.

Subdivision $\boldsymbol{D_1}$: Let S be the vertex set of the obstacles in P, and let $D' = (S', E')$ be the subdivision D' defined for point set S in Section 2.3, viewed as a graph. Let $D = (S, E)$ be the graph defined by the obstacles in P. Then let D_1 be the superimposition of graphs D' and D; that is, the edges of D' and D are split at their intersection points, and these intersection points are introduced as vertices of D_1, in addition to the vertices of D' and D.

In order to derive subdivision D_2 from D_1, we need to distinguish between a *region of the subdivision* D_1 and a *face of the graph* D_1. Every face f of graph D_1 is contained in a region R of subdivision D'. While the boundary of face f may have many vertices on it, we ensure that the corresponding region $R(f)$ has constant boundary size. The boundary of $R(f)$ contains only those vertices on the boundary of f that are vertices of the region R, obstacle vertices, or intersection points between the boundary of region R and obstacle edges. All other vertices are removed by combining their incident edges (which are collinear) into a single edge. See Figure 2.1 for an illustration.

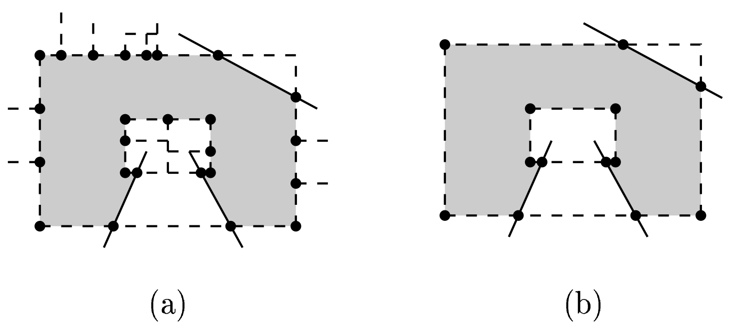

Figure 2.1: A face with many vertices on its boundary (a) and its corresponding region (b).

The regions of subdivision D_1 can be partitioned into two classes: A *red* region is a quadrilateral $R(f)$ so that no vertex of face f is in $S \cup S'$ (see Figure 2.2a), that is, all the vertices of face f are intersection points between edges in E and E'. All remaining regions are *blue* (see Figure 2.2b). A blue region $R(f)$ can be of two types, depending on whether or not face f has a vertex in $S \cup S'$. If face f has no vertex in $S \cup S'$ on its boundary, region $R(f)$ is bounded by six or eight edges. If face f has a vertex in $S \cup S'$, the shape of $R(f)$ can vary. However, region $R(f)$ can be bounded by at most 16 edges: 6 obstacle edges and 10 edges from E'. (See the right region in Figure 2.2b as an example.) Hence, every region of D_1 has constant complexity. This constant complexity of regions is maintained in D_2, which is important for the construction of the Steiner spanner from D_2.

Ladders, rungs, and the red graph: The construction of D_2 merges adjacent red regions in D_1, where adjacency is defined by means of a *red graph* of subdivision D_1. This graph is a subgraph of the dual of D_1. It contains a vertex for every red region of D_1 and an edge between two vertices if the two corresponding regions share an edge that is part of an edge in E'. Since every red region has two edges from E' on its boundary, every vertex in the red graph has degree at most two. Hence, every connected component of the red graph is a path (see Figure 2.3).

Consider the set of red regions corresponding to a connected component of the red graph. These regions are bounded by the same two obstacle edges and a set of edges in E'. We call such a set of red regions a *ladder*. The two obstacle edges on their boundaries are the *sides* of the ladder; the edges from E' are its *rungs*. In other words, the set of all rungs of graph D_1 contains all edges in D_1 that are contained in edges of D' and both of whose endpoints are intersection points of edges in E and E'. We call the topmost rung of a ladder its *top rung*. *Left*, *right*, and *bottom rungs* are defined in a similar manner. All of these four types of rungs are called *extremal rungs*. To simplify the description of the algorithm for constructing D_2, we also consider a single rung between two blue regions to be a ladder. We call such a ladder *trivial*, while a ladder formed as the union of red regions is *non-trivial*.

Subdivision D_2: Subdivision D_2 is now obtained by removing the non-extremal rungs from all ladders of D_1, that is, the red regions of each non-trivial ladder are merged into a single red region. Arikati et al. [7] show that subdivision D_2 has linear size by showing that

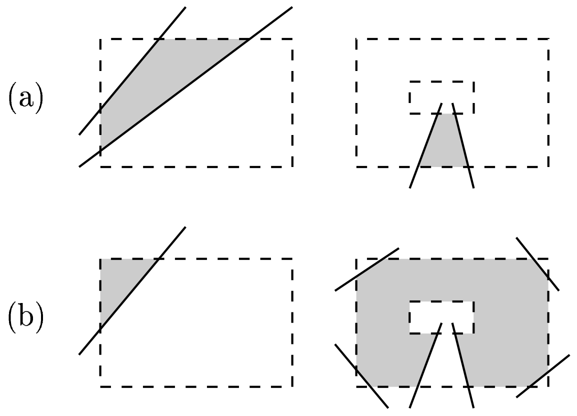

Figure 2.2: (a) Two red regions. (b) The simplest and the most complicated blue regions.

there are $\mathcal{O}(N)$ extremal rungs of ladders remaining in D_2.

Lemma 2.3 (Arikati et al. [7]) *Subdivision D_2 has size $\mathcal{O}(N)$.*

The procedure for deriving a spanner for obstacle set P from subdivision D_2 is similar to the one presented in Section 2.3: We remove all regions inside obstacles from D_2. For each remaining region R, we consider all edges on its boundary that are contained in edges of E'. For each such edge e, we apply procedure $\textsc{Interval}(e, \gamma\|e'\|)$, where e' is the edge of subdivision D' that contains edge e, and shoot rays from the endpoints of e and from the resulting Steiner points toward the interior of R until they meet another boundary edge of R. Again, we add all intersection points between these rays as Steiner points to D_2. Let D'' be the resulting graph. The construction explicitly ensures that D'' is planar. Arikati et al. [7] show that the size of D'' is small and that its spanning ratio is at most $1 + 3\gamma$.

Lemma 2.4 (Arikati et al. [7]) *Graph D'' has size $\mathcal{O}(N/\gamma^2)$ and L_1-spanning ratio at most $1 + 3\gamma$.*

3 I/O-Efficient Algorithms for Constructing Planar L_1-Steiner Spanners

In this section, we describe an I/O-efficient algorithm for constructing a planar L_1-Steiner spanner for a point set S. This is a simple extension of the results presented in Section 2.3. Then we construct a planar L_1-Steiner spanner for a set P of polygonal obstacles in the plane. Since the subdivision is constrained by the obstacle edges, it is harder to compute in this case. In order to compute the subdivision, we again follow the framework of the algorithm of [7], which employs a plane-sweep to carry out its task. However, our representation of the sweep-line status uses only a single buffer tree instead of two balanced search trees. This simplification is the key to obtaining an I/O-efficient algorithm because the approach of [7]

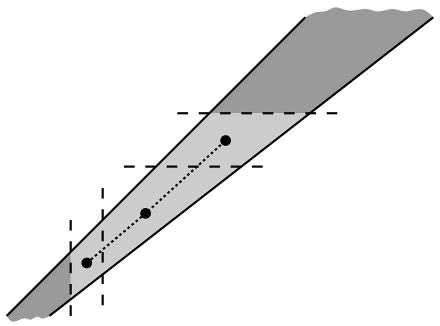
Figure 2.3: A non-trivial ladder and its corresponding connected component of the red graph.

---

requires immediate query responses on the two trees and no I/O-optimal batched search structure is known that answers queries immediately.

In Section 3.1 we present an I/O-efficient algorithm for constructing a planar L_1-Steiner spanner for a point set. In Section 3.2 we show that the subdivision D_2 defined in Section 2.4 can be used to derive a planar L_1-Steiner spanner for obstacles in an I/O-efficient manner. In Sections 3.3 and 3.4, we present an I/O-efficient algorithm for computing the subdivision D_2.

3.1 Planar L_1-Steiner Spanner for a Point Set

In this section, we show how to construct a planar L_1-Steiner spanner for a point set S in an I/O-efficient manner. First we compute a planar subdivision defined by S and introduce additional vertices and edges inside every region of this subdivision as defined in Section 2.3. Then we use this subdivision to construct the spanner.

The first step in obtaining the spanner is to compute the subdivision $D'(T)$ as defined in Section 2.3. The construction of $D'(T)$ requires computing a fair split tree T of S and scanning the vertex set of T to extract the cells of $D'(T)$. By Theorem 2.1, a fair split tree of size $\mathcal{O}(N)$ for S can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. The extraction of the cells takes $\mathcal{O}(\text{scan}(N))$ I/Os. Every node of T adds at most one cell to $D'(T)$, so that we obtain the following lemma.

Lemma 3.1 *The subdivision $D'(T)$ can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os and linear space. Its size is $\mathcal{O}(N)$.*

Given $D'(T)$, it now remains to construct the spanner graph D'' from it, as defined in Section 2.3.

Lemma 3.2 *The graph D'' can be constructed in $\mathcal{O}(\text{sort}(N/\gamma^2))$ I/Os.*

Proof. By Lemma 3.1, subdivision D' can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. Given the cells of D', we scan this set of cells to partition each of them as described in Section 2.3 and add the segments in the resulting partition to the edge set of D'' and their endpoints to the vertex set. Since each cell has a constant-size description, we can load it into internal memory and generate the set of vertices and edges in this cell in a linear number of I/Os.

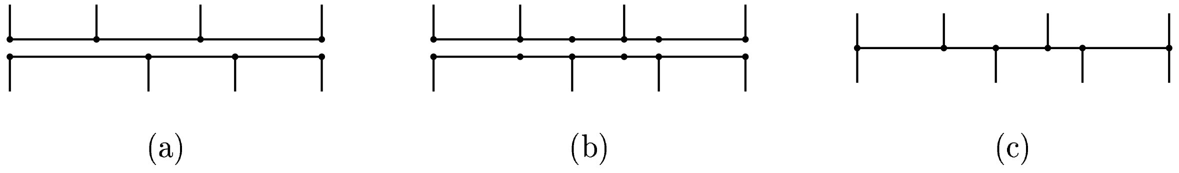

Figure 3.1: Superimposing overlapping edges.

By Lemma 2.2, the total number of vertices and edges is $\mathcal{O}(N/\gamma^2)$, so generating them takes $\mathcal{O}(\text{scan}(N/\gamma^2))$ I/Os. Since each cell is partitioned separately, the resulting vertex set may contain duplicates, and the edge set may contain edges that overlap, that is, are part of the same line and have a non-zero intersection (see Figure 3.1a). The duplicates in the vertex set of D'' can be removed in $\mathcal{O}(\text{sort}(N/\gamma^2))$ I/Os by sorting and scanning this set.

To construct the final edge set of D'', we have to replace overlapping edges, which can be done in the following manner: First we split every edge that contains vertices other than its endpoints into shorter edges at these vertices. This results in an edge set that contains every edge twice (see Figure 3.1b). We remove these duplicates (see Figure 3.1c). To perform this computation I/O-efficiently, while avoiding the generation of duplicate edges altogether, we proceed as follows: We deal with horizontal and vertical edges separately. We describe the procedure for horizontal edges. The procedure for vertical edges is similar. First we sort the horizontal edges by their y-coordinates and the x-coordinates of their left endpoints. Then we scan this sorted edge list, in order to generate the set of edges along every horizontal line in left-to-right order.

Sorting the initial edge set of D'' in this manner takes $\mathcal{O}(\text{sort}(N/\gamma^2))$ I/Os. The scan to partition the edges takes $\mathcal{O}(\text{scan}(N/\gamma^2))$ I/Os. $\square$

If we choose $\gamma = \epsilon/3$, Lemmas 2.2 and 3.2 lead to the following result.

Theorem 3.1 *Given a point set S in the plane and a real number $\epsilon > 0$, a planar Steiner spanner of size $\mathcal{O}(N/\epsilon^2)$ and with L_1-spanning ratio $1+\epsilon$ can be computed in $\mathcal{O}(\text{sort}(N/\epsilon^2))$ I/Os.*

3.2 Planar L_1-Steiner Spanner for Obstacles

In this and the next two subsections, our goal is to construct the planar L_1-Steiner spanner for obstacles I/O-efficiently. Assume for now that we are given the subdivision D_2 as defined in Section 2.4. Then the next lemma shows that the desired spanner D'' can be obtained from it in an I/O-efficient manner. Sections 3.3 and 3.4 are concerned with the I/O-efficient construction of D_2.

Lemma 3.3 *Given a set P of obstacles and the corresponding subdivision D_2, a planar L_1-Steiner spanner of size $\mathcal{O}(N/\gamma^2)$ and spanning ratio at most $1+3\gamma$ for P can be computed in $\mathcal{O}(\text{sort}(N/\gamma^2))$ I/Os.*

Proof. Similar to the construction of spanner D'' for a point set S, the set of Steiner points and incident edges contained in each region of D_2 can be generated in internal memory, because each such region has constant complexity. Hence, a scan of the set of regions suffices to generate all Steiner points and edges in $\mathcal{O}(\text{scan}(N/\gamma^2))$ I/Os. Again, we have to remove duplicates from the resulting vertex set and split overlapping edges at internal vertices. We do this using the same procedure as in Section 3.1. The only complication is that now edges are not only horizontal and vertical, as in the proof of Lemma 3.2. However, all generated edges are either part of an edge in D' or an obstacle edge, or they are new edges generated by partitioning the regions of D_2. The latter edges can be added to the edge set of D'' without modification. The former edges are the ones that may have to be split. Instead of sorting these edges by their y-coordinates, we now sort them by the identities of the obstacle edges or edges of D' containing them. All edges that are contained in the same edge e are sorted by the distances of their closer endpoints to one of the endpoints of e. Once the edges have been sorted in this manner, a single scan of the sorted edge set suffices again to perform the edge splitting. The overall complexity of this algorithm is $\mathcal{O}(\text{sort}(N/\gamma^2))$ I/Os. □

If we choose $\gamma = \epsilon/3$, the following result can be obtained by using Lemmas 2.4, 3.3 and 3.5. The latter shows that the subdivision D_2 can be computed I/O-efficiently.

Theorem 3.2 *Given a set P of polygonal obstacles with a total of N vertices and a real number $\epsilon > 0$, a planar L_1-Steiner spanner of size $\mathcal{O}(N/\epsilon^2)$ and with spanning ratio at most $1 + \epsilon$ for P can be computed in $\mathcal{O}(\text{sort}(N/\epsilon^2))$ I/Os.*

3.3 Computing the Subdivision

Having shown that subdivision D_2 can be used to derive an L_1-Steiner spanner with spanning ratio $1 + \epsilon$ for obstacle set P, we have to provide an I/O-efficient algorithm for constructing this subdivision. As mentioned before, subdivision D_1 may have size $\Omega(N^2)$, so that constructing D_1 and removing all non-extremal rungs from the ladders of D_1 does not lead to an efficient algorithm.

The solution proposed in [7], whose framework we follow here, first constructs a supergraph D_3 of D_2 whose size is $\mathcal{O}(N)$. Due to the linear size of D_3, we can afford to compute the red graph of D_3 explicitly and remove non-extremal rungs from D_3. Graph D_3 is constructed by identifying potential top, bottom, left, and right rungs of each ladder in D_1 and computing the edge set of D_3 as the union of the set of these rungs, the set of all edges in D_1 incident to vertices of D', and the set of obstacle edges. In this section, we describe the I/O-efficient construction of subdivisions D_3 and D_2, once the set of rungs of subdivision D_3 has been computed. An I/O-efficient algorithm for computing these rungs is described in Section 3.4.

3.3.1 Computing Subdivision D_3

Let E'_R be the set of potential top, bottom, left, and right rungs, and let E'_S be the set of edges in D_1 such that each such edge is completely contained in an edge of D' and has at least one endpoint in S' (the vertex set of D'). We call the edges in E'_S *short*. Subdivision D_3 is defined by all obstacle edges, that is, the set of edges in E; the set of rungs in E'_R; and the set E'_S of short edges of D_1. Since we assume that sets E and E'_R are already given, we have to compute only the set E'_S of short edges and extract a representation of subdivision D_3 as an embedded planar graph from the edge set $E \cup E'_R \cup E'_S$ of D_3. We describe these two steps next.

Computing the short edges. The set E'_S of short edges can be computed by answering ray-shooting queries on P and shortening the edges in E' appropriately. In particular, let $p \in S'$ be a vertex of D'. Then there are at most four edges in E' incident to p. For each such edge $e' \in E'$, we shoot a ray ρ from p in the direction of edge e' until it meets an obstacle edge e. Let q be the intersection point of e and ρ, and let $e'' = (p, q)$. If $e'' \subseteq e'$, we add e'' to the set E'_S of short edges. Otherwise, we add edge e' to this set. The ray-shooting queries used to compute edges e'' are axes-parallel, and there are $\mathcal{O}(N)$ of them to be answered. Hence, we can use the endpoint dominance algorithm of [6] to answer these queries in $\mathcal{O}(\text{sort}(N))$ I/Os. Given the set of answers to these queries, a single scan of this set is sufficient to decide, for every edge $e' \in E'$, whether to add edge e' or e'' to E'_S.

Computing graph D_3. A representation of subdivision D_3 as an unordered edge set $E \cup E'_R \cup E'_S$ is not very useful for computing D_2 from D_3; a representation of D_3 as a graph is required. The geometric information pertaining to the vertices and edges of D_3 then provides us with a planar embedding $\hat{D}_3$ of graph D_3.

The vertex set of D_3 is easily constructed as the set of endpoints of all edges in $E \cup E'_R \cup E'_S$. Duplicates can be removed from the generated vertex set by sorting and scanning this set. To compute the edge set of D_3, every obstacle edge has to be split at the endpoints of rungs and short edges lying on this obstacle edge. Once all obstacle edges have been split, duplicate rungs and short edges have to be removed from the edge set, since these edges may have been generated more than once.

The computation of rungs and short edges can easily be augmented so that every edge in $E'_R \cup E'_S$ stores the identities of the obstacle edges containing its endpoints. Hence, we can split the obstacle edges as follows: We sort the set of endpoints of edges in $E'_R \cup E'_S$ by the obstacle edges containing them and so that the endpoints lying on the same obstacle edge are sorted by their distances from one endpoint of that edge. We scan this sorted list of endpoints and the list E of obstacle edges to split obstacle edges. Duplicate rungs and short edges can be removed from the resulting edge set of D_3 by sorting and scanning this set.

In the above construction, we sort and scan sets of size $\mathcal{O}(N)$ a constant number of times. Hence, the construction of the graph D_3 from the edge set $E \cup E'_R \cup E'_S$ takes $\mathcal{O}(\text{sort}(N))$ I/Os. Together with the fact that the set of short edges can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, we obtain the following lemma.

Lemma 3.4 *Given the set E of obstacle edges, the set E'_R of rungs of subdivision D_3, and subdivision D', a representation of D_3 as an embedded planar graph can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

3.3.2 Computing Subdivision D_2

To construct subdivision D_2 from graph D_3, we compute the dual D_3^* of graph D_3, extract the red graph of D_3 from D_3^*, and remove all those edges from D_3 that are dual to edges in the red graph of D_3. The result is a graph D_3' whose faces define the regions of D_2. We scan the list of edges on the boundary of each face of D_3' and merge consecutive collinear edges to produce a constant-size description of each region of D_2.

Given a planar embedding $\hat{D}_3$ of D_3, it is shown in [21] how to compute the dual D_3^* of D_3 in $\mathcal{O}(\text{sort}(N))$ I/Os. The computation of this algorithm can easily be augmented to colour every vertex of D_3^* either red or blue, depending on the type of its corresponding region, and to mark every edge of D_3^* as being dual to an obstacle edge or an edge of D'. We identify all edges of D_3^* that have at least one blue endpoint. Then we scan the edge set of D_3^* to remove all edges with at least one blue endpoint and edges that are dual to obstacle edges. Let E_3' be the resulting set of edges. The edges in E_3' are the edges of the red graph of D_3. Each edge in E_3' is dual to a non-extremal rung of a ladder in D_3. We scan the set E_3' to construct the set E_3'' of rungs dual to the edges in E_3'. Finally, we sort and scan the edge set E_3 of D_3 and the set E_3'' to remove the rungs in E_3'' from D_3. This produces the graph D_3'.

The faces of D_3' correspond to the regions of subdivision D_2. To construct a representation of D_2 as a collection of regions, we compute a representation of the faces of D_3' as a collection of edge lists, each storing the edges of one face clockwise around that face. Such a representation can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os [21]. Once this representation is given, a single scan of these edge lists suffices to compute constant-size representations of the regions $R(f)$, for all faces f of D_3'. In particular, we scan each edge list and merge consecutive collinear edges of the same type (subsegment of an edge in E or E'). By the discussion in Section 2.4, only $\mathcal{O}(1)$ edges remain per face.

In the above construction of subdivision D_2 from graph D_3, the computation of the dual of D_3 and the extraction of the list of edges on the boundaries of the faces of D_3' takes $\mathcal{O}(\text{sort}(N))$ I/Os [21]. Besides that, we sort and scan lists of length $\mathcal{O}(N)$ a constant number of times. By Lemma 3.6 below, the rungs of subdivision D_3 can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. Lemma 3.4 shows that D_3 can then be obtained in $\mathcal{O}(\text{sort}(N))$ I/Os. Thus, we have the following result.

Lemma 3.5 *Given a set P of polygonal obstacles in the plane, subdivision D_2 can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, where N is the total number of obstacle vertices.*

3.4 Computing the Rungs

The construction of subdivision D_2 in Section 3.3 assumes that the set E_R' of extremal rungs of subdivision D_1 is given. Computing this set of rungs is the difficult part of the algorithm. The remainder of this section is dedicated to describing a procedure for computing set E_R' in $\mathcal{O}(\text{sort}(N))$ I/Os. In [7], a plane-sweep algorithm is used to compute each type of extremal rungs separately. We follow this approach, but simplify the sweep-line data structure so that the approach leads to an I/O-efficient algorithm.

Since top, bottom, left, and right rungs are computed using four similar plane sweeps, we only describe the computation of top rungs. Let E_h' be the set of horizontal edges of

Procedure FindTopRungs

Input: The set E'_h of horizontal edges of subdivision D' and the set E of obstacle edges.
Output: A set E'_R of size $\mathcal{O}(N)$ that contains the top rungs of all ladders in D_1.

```
1: $E'_R \leftarrow \emptyset$
2: $L \leftarrow \langle (a,b) \rangle$, where $a$ and $b$ are the left and right sides of square $C$.
   {$L$ is the list of intervals intersected by the sweep line, sorted from left to right.}
   {$C$ is the square containing all polygon vertices.}
3: Mark interval $(a,b)$ as "non-ladder".
4: $Y = \{y(p) : p \in S\} \cup \{y(e) : e \in E'_h\}$
   {$Y$ is the set of $y$-coordinates where the set of intervals in $L$ changes.}
5: for all $y \in Y$, sorted from $-\infty$ to $+\infty$ do
6:    if $y = y(p)$, for some point $p \in S$ then
7:       Let $e_1$ and $e_2$ be the two edges incident to $p$, where $e_1$ is to the left of or below $e_2$.
8:       if $p$ is the bottom corner of an obstacle in $P$ then
9:          ProcessBottomCorner($p, e_1, e_2, L, E'_R$)
10:      else if $p$ is the top corner of an obstacle in $P$ then
11:         ProcessTopCorner($p, e_1, e_2, L, E'_R$)
12:      else
13:         ProcessNonExtremalCorner($p, e_1, e_2, L, E'_R$)
14:      end if
15:   else
16:      Let $e \in E'_h$ such that $y = y(e)$.
17:      ProcessHorizontalEdge($e, L, E'_R$)
18:   end if
19: end for
20: for all remaining intervals $(l,r) \in L$ do
21:    Add the top rung of interval $(l,r)$ to $E'_R$ if $(l,r)$ is a ladder interval.
22: end for
```

Algorithm 3.1: Finding the top rungs of all ladders in D_1.

Procedure ProcessBottomCorner(p, e_1, e_2, L, E'_R)

```
1: Find the interval $(l, r) \in L$ containing point $p$.
2: if $(l, r)$ is a ladder interval then
3:     Add the top rung of $(l, r)$ to set $E'_R$.
4: end if
5: Replace $(l, r)$ with three non-ladder intervals $(l, e_1)$, $(e_1, e_2)$, and $(e_2, r)$ in $L$.
```

Procedure ProcessTopCorner(p, e_1, e_2, L, E'_R)

```
1: Let $(l, e_1)$, $(e_1, e_2)$, and $(e_2, r)$ be the three intervals incident to edges $e_1$ and $e_2$.
2: Add the top rungs of all ladder intervals in $\{(l, e_1), (e_1, e_2), (e_2, r)\}$ to $E'_R$.
3: Replace $(l, e_1)$, $(e_1, e_2)$, and $(e_2, r)$ with a single non-ladder interval $(l, r)$ in $L$.
```

Procedure ProcessNonExtremalCorner(p, e_1, e_2, L, E'_R)

```
1: Let $(l, e_1)$ and $(e_1, r)$ be the two intervals incident to edge $e_1$.
2: Add the top rungs of all ladder intervals in $\{(l, e_1), (e_1, r)\}$ to $E'_R$.
3: Replace intervals $(l, e_1)$ and $(e_1, r)$ with two non-ladder intervals $(l, e_2)$ and $(e_2, r)$, re-
   spectively.
```

Procedure ProcessHorizontalEdge(e, L, E'_R)

```
1: Locate the intervals $(l_1, r_1)$ and $(l_k, r_k)$ in $L$ that contain the endpoints of edge $e$.
2: Let $(l_2, r_2), \ldots, (l_{k-1}, r_{k-1})$ be the intervals between $(l_1, r_1)$ and $(l_k, r_k)$ in $L$.
3: Add the top rungs of all ladder intervals in $\{(l_1, r_1), (l_k, r_k)\}$ to $E'_R$.
4: Mark intervals $(l_1, r_1)$ and $(l_k, r_k)$ as "non-ladder".
5: Mark intervals $(l_2, r_2), \ldots, (l_{k-1}, r_{k-1})$ as "ladder" and make edge $e$ their top rung.
```

Algorithm 3.2: The four procedures used for processing event points in Algorithm 3.1.

subdivision D'. Then we use Algorithm 3.1 to compute all potential top rungs. The algorithm employs a plane sweep in the $(+y)$-direction to carry out its task. During the sweep, it maintains a set of intervals defined by intersections between the sweep line ℓ and obstacle edges. In particular, let $e_1, \dots, e_k$ be the edges in E intersected by the sweep line, sorted from left to right. Then the set of intervals currently stored for ℓ is $(e_1, e_2), (e_2, e_3), \dots, (e_{k-1}, e_k)$. Each such interval (e_i, e_{i+1}) is classified as *ladder* or *non-ladder*, depending on whether the algorithm has already found a rung whose endpoints lie on e_i and e_{i+1}. In particular, for an interval $I = (e_i, e_{i+1})$, let h be the highest edge in E'_h below ℓ and intersecting both e_i and e_{i+1}. Interval I is a non-ladder interval if h does not exist or the quadrilateral defined by ℓ, e_i, e_{i+1}, and h contains a point from $S \cup S'$. Otherwise, I is a ladder interval.

During the sweep, an interval can be created or destroyed only when the sweep passes the y-coordinate of an endpoint of an edge in E. The type of an interval can change only when the sweep passes the y-coordinate of an edge in E'_h. The algorithm deals with these different types of event points and maintains the list of intervals, their classification, and, for ladder intervals, their top rungs. It is easy to verify that the algorithm maintains the list of intervals and their classification correctly, so that the rule for reporting top rungs is also correct. We have to present a data structure to represent the sweep-line status so that the list of intervals can be maintained I/O-efficiently.

The data structure has to support updates of the status of a sequence of consecutive ladder intervals at the cost of $o(1)$ changes to the data structure per interval in the sequence. To see why this is necessary, observe that an edge $e \in E'_h$ may intersect $\Omega(N)$ obstacle edges, so that the status of $\Omega(N)$ intervals has to be updated when the sweep passes edge e. Since there are $\Theta(N)$ edges in E'_h, the number of required updates of the sweep-line data structure would be $\Omega(N^2)$ if each updated interval caused $\Omega(1)$ updates of the data structure. In order to achieve $o(1)$ data structure updates per modified interval, our data structure, just as the one presented in [7], exploits the following observation.

Observation 3.1 (Arikati et al. [7]) *Two consecutive ladder intervals have the same top rung.*

The data structure proposed in [7] exploits this fact by representing intervals and their status in two separate binary search trees. The first tree, T, stores the current set of intervals sorted from left to right. The second tree, T', represents the status of all intervals in a compressed form by storing only one entry for each consecutive sequence of intervals of the same type. Since every event point generates or destroys only $\mathcal{O}(1)$ intervals, procedure FINDTOPRUNGS performs only $\mathcal{O}(N)$ updates of tree T. Counting the number of updates of tree T' is not quite that easy. However, it can be shown that every generation or deletion of an entry in tree T' can be charged to a blue region in subdivision D_1 so that every region is charged only $\mathcal{O}(1)$ times. Every change of the status of an existing entry in T' can be charged to an edge in E'_h. Hence, only $\mathcal{O}(N)$ updates of tree T' are performed, and procedure FINDTOPRUNGS takes $\mathcal{O}(N \log N)$ time.

In order to obtain a data structure that allows the sweep to be performed in $\mathcal{O}(\mathrm{sort}(N))$ I/Os, we could try to replace trees T and T' with buffer trees. Unfortunately, this simple idea cannot be applied, as the updates of tree T are driven by the answers to queries on tree T' and vice versa, so that immediate query responses are required. The efficiency of the buffer tree, however, is achieved by *delaying* query responses and answering queries whenever a large enough number of queries have accumulated.

Next we show how to represent the sweep-line status using only a single buffer tree. The fact that the buffer tree delays the processing of updates and queries still creates problems that have to be dealt with. In order to support our claim that our data structure is simpler than the one proposed in [7], we present the algorithm as if it used an (a, b)-tree, which does not buffer updates or queries. In Section 3.4.2, we discuss the problems created by delayed updates and show how to deal with them.

3.4.1 A Simplified Sweep-Line Data Structure

Our sweep-line data structure consists of a single (a, b)-tree T [20]. The leaves of T store the current set of intervals sorted from left to right. Every internal node stores the obstacle edges separating the intervals stored at descendants of its children.

In order to obtain a classification of the intervals as ladder and non-ladder intervals, every interval I stores a label $\lambda(I) = (y, \tau, \rho)$, and every internal node v of T stores a label $\lambda(v) = (y, \tau, \rho)$. For an interval I, label $\lambda(I) = (y, \tau, \rho)$ signifies that interval I was of type τ just after sweep-line ℓ crossed y-coordinate y. If $\tau =$ "ladder", ρ is the topmost rung of the ladder in interval I below or on the horizontal line at y-coordinate y. For an internal node, label $\lambda(v) = (y, \tau, \rho)$ signifies that every interval I stored at a descendant of v was of type τ just after sweep-line ℓ crossed y-coordinate y. If $\tau =$ "ladder", ρ is the topmost rung of the ladder in interval I below or on the horizontal line at y-coordinate y.

We maintain the invariant that, at any time, the correct type and top rung of an interval I can be determined as follows: Let v_0 be the leaf of T storing interval I, and let $v_1, \dots, v_k$ be the proper ancestors of v_0 in T. Let $\lambda(I) = (y_0, \tau_0, \rho_0)$, and let $\lambda(v_i) = (y_i, \tau_i, \rho_i)$, for $1 \leq i \leq k$. Then interval I is of type τ_i and its top rung is ρ_i if $\tau_i =$ "ladder", where $y_i = \max\{y_j : 0 \leq j \leq k\}$; that is, in terms of the plane sweep, the most recent information stored on the path $(v_0, \dots, v_k)$ in T is the correct characterization of interval I.

The basic query operation on tree T is operation REPORT(I), which searches for interval I and reports its top rung if interval I is a ladder interval. To do this, it traverses the path in T from the root to the leaf storing interval I and finds the triple with maximal y-coordinate stored on this path. By the above invariant, this triple contains the correct type and top rung of interval I. Given at least one point in interval I, the obstacle edges stored at the internal nodes of T are sufficient to locate the leaf storing interval I.

Next we discuss the required updates of tree T when the sweep-line passes an event point. We discuss the update procedures in Algorithm 3.2 for the four different types of event points separately and argue that each of them maintains the above invariant.

PROCESSBOTTOMCORNER: A bottom corner $p \in S$ of an obstacle is a vertex whose incident edges e_1 and e_2 are both above p. Let e_1 be to the left of e_2. Then the top rung of the interval $I = (l, r)$ containing point p has to be reported if interval I is a ladder interval, and interval I has to be replaced with three non-ladder intervals (l, e_1), (e_1, e_2), and (e_2, r). To achieve this, we apply an operation REPORTANDSPLIT$(I, e_1, e_2, y(p))$ to tree T. This operation first applies operation REPORT(I) to locate interval I and report its top rung if necessary. Then it replaces interval I with three new intervals

$I_1 = (l, e_1)$, $I_2 = (e_1, e_2)$, and $I_3 = (e_2, r)$. Every interval I_j, $j \in \{1, 2, 3\}$, is assigned a label $\lambda(I_j) = (y(p), \text{"non-ladder"}, \textbf{null})$. Since $y < y(p)$, for every label $\lambda(v) = (y, \tau, \rho)$ stored at an ancestor of intervals I_1, I_2, and I_3 in T, this correctly marks all three intervals as non-ladder intervals.

The replacement of interval I with intervals I_1, I_2, and I_3 may cause tree T to become unbalanced. Tree T can be rebalanced using the standard procedure for rebalancing an (a, b)-tree after an INSERT operation. A node created during a node split inherits the labels of the node that has been split.

PROCESSTOPCORNER: A top corner $p \in S$ of an obstacle is a vertex whose incident edges e_1 and e_2 are both below p. Let e_1 be to the left of e_2. Then the top rungs of the ladder intervals among intervals $I_1 = (l, e_1)$, $I_2 = (e_1, e_2)$, and $I_3 = (e_2, r)$ incident to edges e_1 and e_2 need to be reported, and intervals I_1, I_2, and I_3 need to be replaced with a single non-ladder interval $I = (l, r)$. To achieve this, we apply three operations $o_1 = \text{REPORTANDREPLACE}(I_1, I, y(p))$, $o_2 = \text{REPORTANDDELETE}(I_2)$, and $o_3 = \text{REPORTANDDELETE}(I_3)$ to tree T. Operation o_1 searches for interval I_1, reports its top rung if it is a ladder interval, and replaces it with interval I. The label of interval I is set to $\lambda(I) = (y(p), \text{"non-ladder"}, \textbf{null})$, thereby correctly marking interval I as a non-ladder interval. Operations o_2 and o_3 search for intervals I_2 and I_3, report their top rungs if necessary, and delete these intervals. After deleting intervals I_2 and I_3, tree T can be rebalanced using the standard rebalancing procedure for DELETE operations on (a, b)-trees. There are, however, a few complications:

The first one is that we may end up fusing two nodes v_1 and v_2 with different labels into a single node v. When this happens, we first update the labels of all children of v_1 and v_2 and then store no label at all at node v. For a child w of node v_i, its new label is the one among $\lambda(v_i)$ and $\lambda(w)$ with higher y-value. It is easily verified that, given these label updates, a node fusion maintains the correct classification and top rung for every interval stored in T.

The second complication is that procedure PROCESSTOPCORNER is supplied only with the two edges e_1 and e_2 incident to point p. This allows intervals I_1, I_2, and I_3 to be located in T, but is not sufficient to provide operation o_1 with a complete description of interval I. In order to work around this problem, we apply operations o_2 and o_3 first and augment operation o_3 so that it reports the right boundary edge of interval I_3, which is also the right boundary of interval I. Together with the left boundary of interval I_1, operation o_1 can now compute a complete description of interval I.

The final problem is that the search information in T may be invalidated by operation o_1. This happens when the separating obstacle edges to the right of the path traversed by operation o_1 intersect interval I. This is possible because interval I spans three of the intervals previously stored in T. Operation o_1 can change these edges to the right boundary of interval I while it traverses the path in T to the leaf storing interval I_1.

PROCESSNONEXTREMALCORNER: A non-extremal corner $p \in S$ of an obstacle is a vertex such that an obstacle edge e_1 is incident to p from below, and another obstacle edge e_2 is incident to p from above. At such an event point, the following updates are necessary: Let $I_1 = (l, e_1)$ and $I_2 = (e_1, r)$ be the two intervals on both sides of edge e_1. Then the top rungs of both intervals need to be reported, depending on whether they are ladder intervals; and intervals I_1 and I_2 need to be replaced with two non-ladder intervals $I'_1 = (l, e_2)$ and $I'_2 = (e_2, r)$, respectively. This can be achieved by applying two operations $\textsc{ReportAndReplace}(I_1, I'_1, y(p))$ and $\textsc{ReportAndReplace}(I_2, I'_2, y(p))$ to tree T.

Observe that, after this event point, the sweep line no longer intersects edge e_1 and instead intersects edge e_2. Therefore, in order to maintain the correct search information in tree T, every occurrence of edge e_1 as a dividing segment between the children of some node v has to be replaced with e_2. It is easy to see that the affected nodes have to be stored on the root-leaf path traversed by the second ReportAndReplace operation, and this operation can replace all occurrences of e_1 with e_2 as it traverses this path.

ProcessHorizontalEdge: The last type of event point is the y-coordinate of a segment $s \in E'_h$. Let p_l and p_r be the left and right endpoints of segment s, and let I_l and I_r be the intervals containing these two endpoints, respectively. Then the top rungs of intervals I_l and I_r need to be reported, depending on whether these intervals are ladder intervals; intervals I_l and I_r have to be marked as non-ladder intervals; and all intervals between I_l and I_r have to be marked as ladder intervals with top rung s. We achieve this by applying an operation $o = \textsc{MakeLadder}(I_l, I_r, y(s))$ to tree T. Operation o searches down the tree until it finds the first node v such that intervals I_l and I_r are stored at descendants of different children of v. Let $w_1, \ldots, w_k$ be the children of v, and let w_i and w_j be the two children such that intervals I_l and I_r are stored at descendants of w_i and w_j, respectively. Then operation o changes the label of every node w_h, $i < h < j$, to $\lambda(w_h) = (y(s), \text{“ladder”}, s)$. Now we split operation o into two operations $o_1 = \textsc{MakeRightLadder}(I_l, y(s))$ and $o_2 = \textsc{MakeLeftLadder}(I_r, y(s))$. Operation o_1 continues down the path from w_i to the leaf storing interval I_l. Operation o_2 follows the path from w_j to the leaf storing interval I_r. At every visited internal node v', operation o_1 performs the following operation: Let $w'_1, \ldots, w'_k$ be the children of node v', and let interval I_l be stored at a descendant of w'_i. Then the label of every node w'_h, $i < h \leq k$, is changed to $\lambda(w'_h) = (y(s), \text{“ladder”}, s)$. When the leaf v_0 storing interval I_l is reached, the top rung of interval I_l is reported if necessary, based on the information collected by operations o and o_1 along the path from the root of T to leaf v_0. Then the label of interval I_l is set to $\lambda(I_l) = (y(s), \text{“non-ladder”}, \mathbf{null})$. Operation o_2 performs the same updates w.r.t. interval I_r, but labels all intervals to the left of the search path as ladder intervals. It is easily verified that an interval appears between I_l and I_r if and only if it is stored at a descendant of a node w_h or w'_h in T whose label is changed to “ladder”. Hence, this procedure correctly updates the type information of every interval.

Each of the above update procedures for the four different types of event points traverses a constant number of root-to-leaf paths in T. Each procedure spends $\mathcal{O}(b)$ time per visited node. In internal memory, we would choose $b = \mathcal{O}(1)$, so that every event point can be processed in $\mathcal{O}(\log N)$ time. As there are $\mathcal{O}(N)$ event points, procedure FINDTOPRUNGS finds all top rungs in $\mathcal{O}(N \log N)$ time.

3.4.2 Buffering Updates

In order to make the sweep-line data structure I/O-efficient, the first step is to turn the (a, b)-tree into a buffer tree. Query and update procedures remain the same. But they are processed in a batched fashion, so that processing $\mathcal{O}(N)$ queries and updates now takes $\mathcal{O}(\text{sort}(N))$ I/Os. However, the delayed processing of updates creates problems:

(i) The x-order of obstacle edges and, hence, the order of intervals defined by these edges is not a total order. While all segments intersected by a horizontal line have a well-defined order, edges whose y-spans are disjoint are incomparable. This does not create any problems for a standard (a, b)-tree because queries and updates are processed immediately and the search information in tree T is updated so that all separating edges stored at internal nodes of T intersect the current sweep line. In a buffer tree, on the other hand, delayed updates can lead to the situation that some nodes in T store splitter edges that are incomparable to points on the current sweep line because they are completely below the sweep line. Thus, it is not clear at this point at which child of such a node to continue the search for a particular interval.

(ii) We argued in Section 3.4.1 that, when operation REPORTANDREPLACE is applied in procedure PROCESSTOPCORNER, the information about the right boundary of interval I can be collected by first deleting interval I_3 and then replacing interval I_1 with interval I. In a buffer tree, this strategy cannot be applied because the deletion of interval I_3 would have to be processed immediately. This makes it impossible to guarantee that a large enough number of queries and updates has accumulated at a node of T before emptying the buffer of this node. But this is crucial for the I/O-efficiency of the buffer tree.

Our solution for Problem (i) is to define a total order of the intervals by precomputing the set of all intervals defined by obstacle edges and numbering them in a manner consistent with the partial x-order of these intervals. Intervals are then stored in the buffer tree sorted according to their numbers. This solves the first problem, but creates a new problem:

(iii) An update operation cannot search for an interval in T using the location of a point p in the interval because tree T no longer stores any geometric search information. Thus, every update operation has to be provided with the numbers of the intervals involved in the update.

The fourth and final problem is the following:

(iv) The procedure for updating splitter values that are invalidated by an application of operation REPORTANDREPLACE requires the immediate processing of this operation, which is not feasible using a buffer tree. Hence, the numbering of the intervals has to be defined so that, if the splitter values in the buffer tree are chosen carefully, an application of operation REPORTANDREPLACE does not invalidate any splitters.

As mentioned above, our solution for Problem (i) is to precompute the set of intervals and assign a unique number $\nu(I)$ to each interval I. We obtain a total order of the set of intervals as the total order defined by numbering ν. Below we discuss the computation of this numbering and argue that it satisfies the condition in Problem (iv). To solve Problems (ii) and (iii), we compute three lists L_R, L_D, and L_H. List L_R stores pairs (I_1, I_2), where interval I_1 is replaced by interval I_2 during the sweep. List L_D stores single intervals that are removed from tree T using operation REPORTANDDELETE during the sweep. List L_H stores pairs (I_l, I_r) of intervals, where intervals I_l and I_r contain the left and right endpoints of a segment in E'_h. The elements of lists L_R, L_D, and L_H are sorted by the y-coordinates of the corresponding event points. Every interval I in lists L_R, L_D, and L_H stores both its bounding segments as well as its number $\nu(I)$. The I/O-efficient version of Algorithm 3.1 now proceeds as follows:

We use a buffer tree instead of an (a,b)-tree to store the set of intervals intersected by the sweep line. These intervals are sorted by their numbers, which is consistent with the x-order of these intervals, by the definition of the numbering ν. Operation REPORTANDREPLACE retrieves the next pair (I_1, I_2) from list L_R. It searches for interval I_1 in T, using number $\nu(I_1)$, and replaces I_1 with I_2. Operation REPORTANDSPLIT retrieves the next three pairs (I, I_1), (I, I_2), and (I, I_3) from L_R. It searches for interval I in T and replaces this interval with intervals I_1, I_2, and I_3. Operation REPORTANDDELETE retrieves the next interval I from list L_D, searches for interval I in T, and removes this interval. Operation MAKELADDER retrieves the next pair (I_l, I_r) from list L_H and makes a ladder between intervals I_l and I_r.

The correctness of this modified version of Algorithm 3.1 is easily verified. Its I/O-complexity is $\mathcal{O}(\text{sort}(N))$ because it performs $\mathcal{O}(N)$ updates and queries on a buffer tree T of size $\mathcal{O}(N)$ and scans lists L_R, L_D, and L_H. Below we show how to compute the numbering ν as well as lists L_R, L_D, and L_H in $\mathcal{O}(\text{sort}(N))$ I/Os, which proves the following lemma.

Lemma 3.6 *Given a set P of polygonal obstacles and the subdivision D' defined by the set S of obstacle vertices, the set of top rungs of subdivision D_3 can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os, where $N = |S|$.*

The replacement tree. To compute the numbering ν and lists L_R, L_D, and L_H, we use a rooted tree defined on the set of all intervals. We call this tree the *replacement tree* T_R. We show that this tree can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os. The extraction of numbering ν and lists L_R, L_D, and L_H from tree T_R takes $\mathcal{O}(\text{sort}(N))$ I/Os, as we show below.

To construct the vertex set of T_R, we have to compute the set of all intervals defined by edges of the obstacles in P. This set of intervals is easily obtained from a trapezoidation of the obstacle set (see Figure 3.2). In particular, the y-span of each trapezoid t defines the life span of the interval defined by the two obstacle edges on the boundary of t during the sweep; the interval is created when the sweep passes the bottom boundary of the trapezoid, and it is replaced by a new interval when the sweep passes the top boundary of the trapezoid. Since a trapezoid is an interval augmented with the range of y-coordinates where this interval is valid, we henceforth consider intervals and trapezoids to be the same, choosing between these two names for the same object depending on the context.

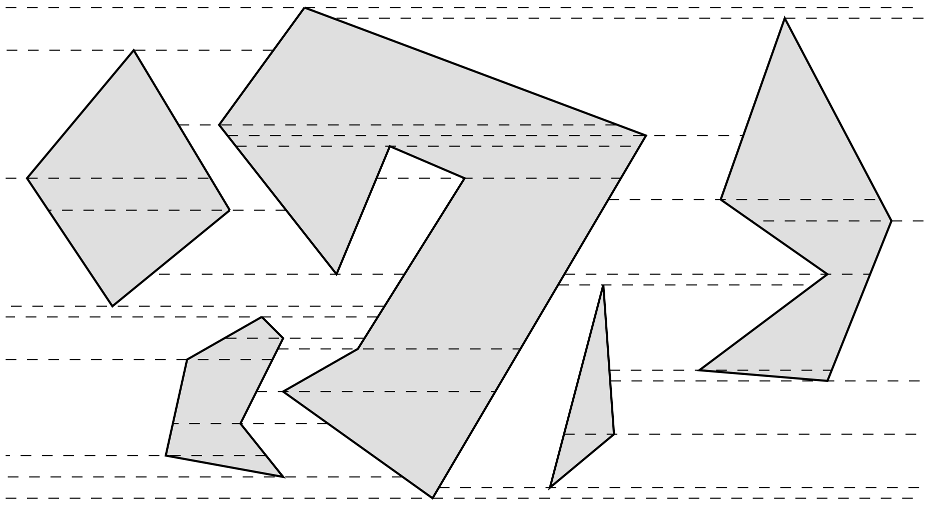

Figure 3.2: A trapezoidation of a set of polygons.

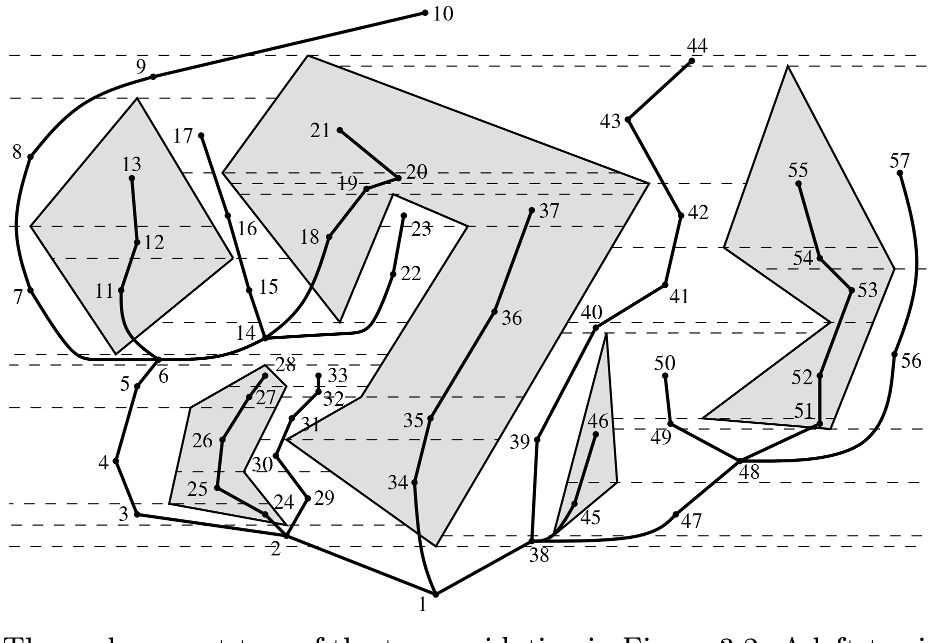

Figure 3.3: The replacement tree of the trapezoidation in Figure 3.2. A left-to-right preorder numbering of this tree defines a total order of the trapezoids.

As already mentioned, tree T_R contains one node per trapezoid. The root of T_R is the unbounded trapezoid extending to infinity in $(-y)$-direction. Every other trapezoid has a parent defined as follows: Consider an obstacle vertex $p \in S$. If p is a bottom corner, the three trapezoids I_1, I_2, and I_3 incident to p from above are the children of the trapezoid I below p. If p is a top corner, let I_1, I_2, and I_3 be the three trapezoids incident to p from below, sorted from left to right. Then the trapezoid I above p is the child of trapezoid I_1. Finally, for a non-extremal corner, there are two trapezoids I_1 and I_2 incident to p from below and two trapezoids I_1' and I_2' incident to p from above. Let I_1 be to the left of I_2, and let I_1' be to the left of I_2'. Then I_1' is the child of I_1, and I_2' is the child of I_2. This construction is illustrated in Figure 3.3.

Lemma 3.7 *The replacement tree T_R of the set of intervals defined by the obstacles in P can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Proof. A trapezoidation of the obstacle set P can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using the endpoint dominance algorithm of [6]. In particular, this algorithm computes the set of horizontal edges incident to every obstacle vertex and the obstacle edges containing the other endpoints of these edges; that is, after applying this algorithm, we obtain a representation of the trapezoidation as the set of obstacle edges and horizontal edges defining the trapezoidation. Using the same algorithm as for deriving a representation of subdivision D_3 as an embedded planar graph (see Section 3.3), we can derive a representation of the trapezoidation as an embedded planar graph in $\mathcal{O}(\text{sort}(N))$ I/Os.

Using an algorithm from [21], the faces of the trapezoidation can be identified in $\mathcal{O}(\text{sort}(N))$ I/Os. The output of the algorithm is a representation of the faces as a collection of lists, each storing the vertices of one face clockwise around the face. From this representation, we generate pairs (v, f), where v is a vertex on the boundary of face f. Then we sort the list of these pairs lexicographically, so that, for each vertex v, pairs $(v, f_1), \dots, (v, f_4)$ are stored consecutively. The edge set of tree T_R can now be produced in a single scan of this list of vertex-face pairs. □

Computing a total order of the intervals. In order to obtain a total order of all trapezoids, we compute a preorder numbering ν of T_R that is consistent with a depth-first traversal of T_R that visits the children of every node in left-to-right order (see Figure 3.3). Then we define a total order "$<$" on the intervals as $I_1 < I_2$ if $\nu(I_1) < \nu(I_2)$. The following lemma is easy to show by induction on the set of obstacle vertices, which is where the set of intervals intersected by the sweep-line changes.

Lemma 3.8 *Let $I_1, \dots, I_k$ be the set of trapezoids intersected by a horizontal line ℓ, sorted from left to right. Then $\nu(I_1) < \dots < \nu(I_k)$.*

Lemma 3.8 and the above discussion establish that the total order defined by the preorder numbering ν can be used to sort the intervals in T from left to right. Next we describe a rule for choosing the splitters in tree T so that REPORTANDREPLACE operations do not invalidate these splitters. Let w_i and w_{i+1} be two consecutive children of a node v in T. Then we choose the splitter between w_i and w_{i+1} to be $\sigma_i = \min\{\nu(I) : I \in \mathcal{I}(w_{i+1})\}$, where

$\mathcal{I}(x)$ is the set of intervals stored in the subtree rooted at node x. Note that this is a rule we apply whenever we choose a new splitter; it is not an invariant we maintain at all times.

Now consider an application of operation ReportAndReplace or ReportAndSplit that replaces an interval I_1 with another interval I_2. Let v_0 be the leaf of the buffer tree T that stores interval I_1, and let $v_1, \ldots, v_k$ be the ancestors of leaf v_0, sorted by increasing distance from v_0; that is, v_k is the root of tree T. For $1 \leq i < k$, let σ_i and σ'_i be the two splitters stored at node v_{i+1} so that, for all intervals $I' \in \mathcal{I}(v_i)$, $\sigma_i \leq \nu(I') < \sigma'_i$. Then we prove the following lemma.

Lemma 3.9 *If operation* ReportAndReplace *or* ReportAndSplit *replaces an interval I_1 with an interval I_2, and $v_1, \ldots, v_k$ are the ancestors of the leaf v_0 of the buffer tree T that stores interval I_1, then $\sigma_i \leq \nu(I_2) < \sigma'_i$, for all $1 \leq i < k$.*

Proof. Since we assume that the search information stored in tree T is correct before the replacement of interval I_1 with interval I_2, we have $\sigma_i \leq \nu(I_1) < \sigma'_i$, for all $1 \leq i < k$. This immediately implies that $\sigma_i < \nu(I_2)$, for all $1 \leq i < k$, because interval I_2 is a descendant of interval I_1 in tree T_r and, hence, $\nu(I_1) < \nu(I_2)$. We have to show that $\nu(I_2) < \sigma'_i$, for all $1 \leq i < k$.

For the sake of this proof, we introduce the notion of a *slot* in tree T. Every slot holds an interval. Operation ReportAndReplace leaves the set of slots unchanged, but changes the content of the slot holding interval I_1 by storing interval I_2 in this slot. When splitting an interval I into intervals I_1, I_2, and I_3, operation ReportandSplit places I_1 into the the slot currently holding interval I and creates two new slots holding I_2 and I_3. The two created slots are immediately to the right of the slot holding interval I_1.

Now consider a splitter σ'_i. Then $\sigma'_i = \nu(I')$, for some interval I'. Let ℓ be a horizontal line intersecting interval I', and let I_0 be the ancestor of interval I_1 in T_R that is intersected by line ℓ. If we can show that interval I_0 is to the left of interval I', we obtain that $\nu(I_0) < \nu(I')$, by Lemma 3.8. Since I_0 and I' are intersected by the same horizontal line, I' is not a descendant of I_0; interval I_2 is a descendant of interval I_0; hence, $\nu(I_2) < \nu(I') = \sigma'_i$ because any preorder numbering ν of T_R assigns a set of consecutive numbers to the descendants of any node in T_R. In order to show that interval I_0 is to the left of interval I', we show that the slot holding interval I_0 is to the left of the slot holding interval I'.

So assume the contrary, that is, that $I_0 = I'$ or interval I_0 is stored in a slot to the right of the slot storing interval I'. In both cases, the slot holding interval I_0 is stored at a descendant of a node to the right of splitter σ'_i. Hence, if I_0 and I_1 are stored in the same slot, we obtain the desired contradiction. Otherwise, the slot holding interval I_1 has been created from the slot holding interval I_0 by a sequence of ReportAndSplit operations. This operation changes the content of an existing slot and creates two new slots to the right of the existing slot. Hence, the slot holding interval I_1 is to the right of splitter σ'_i in this case as well. □

By Lemma 3.9, no updates of the splitters in the tree are required after an application of operation ReportAndReplace or ReportAndSplit. Hence, the plane sweep can be carried out I/O-efficiently using a buffer tree instead of an (a, b)-tree. We have to show that the preorder numbering ν used to order the intervals in T can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.

We use the Euler tour technique and list-ranking to compute ν. The Euler tour describes a depth-first traversal of tree T_R. We have to ensure that it visits the children of every node in left-to-right order. For every vertex $I \in T_R$ with incident edges $(I, I_1), \dots, (I, I_k)$, let I_1 be the parent of I in T_R, and let $I_2, \dots, I_k$ be the children of I_1, sorted from left to right. Then we define $\text{succ}((I_j, I)) = (I, I_{j+1})$, for $1 \le j < k$, and $\text{succ}((I_k, I)) = (I, I_1)$. This produces the desired Euler tour.

Computing the replacement and deletion lists. Lists L_R and L_D can be computed in a natural manner from tree T_R. In particular, every edge (I_1, I_2) in T_R defines an entry in list L_R because there is an edge (I_1, I_2) in T_R if and only if interval I_1 is replaced by interval I_2 during the sweep. Similarly, every leaf of T_R is removed from T during the sweep, using operation ReportAndDelete. Hence, the contents of lists L_R and L_D can be computed in a single scan of the vertex and edge sets of tree T_R. To arrange the elements $(I_1, I'_1), \dots, (I_s, I'_s)$ in L_R in the correct order, we sort them by the y-coordinates of the edges shared by trapezoids I_j and I'_j, $1 \le j \le s$. We sort the elements $I_1, \dots, I_t$ of list L_D by the y-coordinates of their top edges. This computation of lists L_R and L_D takes $\mathcal{O}(\text{sort}(N))$ I/Os.

Computing list L_H. To construct list L_H, we have to compute the intervals I_l and I_r containing the endpoints of each edge $e \in E'_h$, that is, we have to answer $\mathcal{O}(N)$ point location queries on the trapezoidation of the obstacle set P. To do this, we scan the vertex set of tree T_R and add, for every node $I \in T_R$, the left boundary of trapezoid I to a list X. Then a point p is contained in trapezoid I if and only if a ray shot from p in $(-x)$-direction meets this left boundary of interval I. Hence, we can answer the point location queries for all segment endpoints in $\mathcal{O}(\text{sort}(N))$ I/Os by applying the endpoint dominance algorithm of [6] to the set X of left boundaries and the set of endpoints of all segments in E'_h. The answers to all queries can be combined to pairs (I_l, I_r) in $\mathcal{O}(\text{sort}(N))$ I/Os. Given the set of these pairs, we obtain list L_H by sorting all pairs (I_l, I_r) by the y-coordinates of their corresponding edges in E'_h.

We summarize the above discussion in the following lemma, which completes the proof of Lemma 3.6.

Lemma 3.10 *A total order of all intervals defined by the edges of all obstacles in P as well as lists L_R, L_D, and L_H can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os.*

4 A Planar L_2-Steiner spanner

Given that planar L_1-Steiner spanners for sets of points and obstacles in the plane can be computed I/O-efficiently, we can use them to compute planar L_2-Steiner spanners. In our description, we assume that we want to find a spanner for a set of obstacles; the case of a point set is similar. The construction is based on the following observation: If an edge e is close to vertical, that is, its angle to the y-axis is at most θ, then the L_1-length of e is a $(\cos\theta + \sin\theta)$-approximation of the Euclidean length of edge e. For simplicity, we use only that $\|e\|_1 \leq (1 + \sin\theta)\|e\|_2$. We construct an L_2-spanner with spanning ratio $1 + \epsilon$ for a set P of polygonal obstacles as follows: We choose a constant ϵ' such that $(1 + \epsilon')^2 \leq 1 + \epsilon$. For example, $\epsilon' = \epsilon/3$ satisfies this condition for any $0 < \epsilon \leq 3$. Let $0 < \theta < \pi/2$ so that $\sin\theta = \epsilon'$. Then we choose $\pi/\theta = \mathcal{O}(1/\epsilon)$ coordinate systems at angles $0, \theta, 2\theta, \ldots$ to a fixed reference coordinate system and construct L_1-Steiner spanners $G_0, \ldots, G_q$ for P with stretch factor $1 + \epsilon'$ in these coordinate systems. Let G be the graph obtained as the superimposition of graphs $G_0, \ldots, G_q$; that is, the vertex set of graph G contains all vertices of graphs $G_0, \ldots, G_q$ as well as all intersection points between edges in these graphs. The edge set contains all edges obtained by splitting the edges of graphs $G_0, \ldots, G_q$ at their intersection points. The following lemma of Arikati et al. [7] shows that graph G is a planar L_2-Steiner spanner with spanning ratio $1 + \epsilon$ for the obstacle set P.

Lemma 4.1 (Arikati et al. [7]) *Graph G has size $\mathcal{O}(N/\epsilon^4)$ and is an L_2-Steiner spanner with spanning ratio $1 + \epsilon$.*

We have to show how to construct graph G I/O-efficiently. In order to compute graph G, we apply the following procedure, starting with graphs $G_0, \ldots, G_q$, and repeating it until a single graph remains, which is graph G: We form pairs of graphs in the current set of graphs. For each pair (G', G''), we apply the red-blue line segment intersection algorithm of [6] to compute the set of intersection points between the edges in G' and G''. We sort and scan the union of the vertex sets of graphs G' and G'' with the set of intersection points to remove duplicates and thereby obtain the vertex set of the superimposition G° of graphs G' and G''. The red-blue line segment intersection algorithm can be augmented so that it labels every intersection point with the two edges of graphs G' and G'' intersecting at this point. Hence, we can apply the procedure from Section 3.3 to obtain the edge set of G°. This procedure computes graph G° in $\mathcal{O}(\text{sort}(|G^\circ|))$ I/Os.

Using the above procedure, we compute a hierarchy of graphs where graphs $G_0, \ldots, G_q$ are at level 0, and every graph at level i is produced by superimposing at most two graphs at level $i - 1$. Every graph at level i is the superimposition of at most 2^i level-0 graphs. Hence, by the same arguments as in the proof of Lemma 4.1 in [7], the size of a level-i graph is at most $\mathcal{O}(2^{2i}N/\epsilon^2)$. On the other hand, the number of these graphs can be bounded by $\lceil 3\pi/(2^i\epsilon)\rceil$. Hence, one level of the hierarchy of graphs can be computed in $\mathcal{O}(\text{sort}(2^iN/\epsilon^3))$ I/Os, so that the computation of graph G takes

$$\sum_{i=0}^{h} \mathcal{O}(\text{sort}(2^iN/\epsilon^3)) = \mathcal{O}(\text{sort}(N/\epsilon^4))$$

I/Os, where $h = \lceil \log(3\pi/\epsilon)\rceil$ is the number of levels in the hierarchy. This proves the following result.

Theorem 4.1 *Given a set P of polygonal obstacles with N vertices, a planar L_2-Steiner spanner of size $\mathcal{O}(N/\epsilon^4)$ and with spanning ratio $1 + \epsilon$ can be computed in $\mathcal{O}(\text{sort}(N/\epsilon^4))$ I/Os.*

References

[1] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, September 1988.

[2] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[3] Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory mst, sssp and multi-way planar graph separation. *J. Algorithms*, 53(2):186–206, 2004.

[4] Lars Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. AMS, 1999.

[5] Lars Arge and Laura Toma. External data structures for shortest path queries on planar digraphs. In Xiaotie Deng and Ding-Zhu Du, editors, *ISAAC*, volume 3827 of *Lecture Notes in Computer Science*, pages 328–338. Springer, 2005.

[6] Lars Arge, Darren E. Vengroff, and Jeffrey S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proceedings of the 3rd Annual European Symposium on Algorithms*, volume 979 of *Lecture Notes in Computer Science*, pages 295–310. Springer-Verlag, 1995.

[7] S. Arikati, D. Z. Chen, L. P. Chew, G. Das, M. Smid, and C. D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *Proceedings of the 4th Annual European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 514–528. Springer-Verlag, 1996.

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

[9] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point sets with applications to k-nearest-neighbors and n-body potential fields. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 546–556, 1992.

[10] P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1993.

[11] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 263–272, 1995.

[12] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest neighbors and n-body potential fields. *Journal of the ACM*, 42:67–90, 1995.

[13] Paul B. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1995.

[14] P. Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of the 2nd Annual ACM Symposium on Computational Geometry*, pages 169–177, 1986.

[15] K. L. Clarkson. Approximation algorithms for shortest path motion planning. In *Proc. ACM Symp. on Theory of Computation*, volume 19, pages 56–65, 1987.

[16] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proc. ACM Symp. on Computational Geometry*, volume 14, pages 259–268. ACM Press, 1998.

[17] D. Eppstein. Spanning trees and spanners. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry.* North-Holland, 2000.

[18] Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, November 1993.

[19] Sathish Govindarajan, Tamás Lukovszki, Anil Maheshwari, and Norbert Zeh. I/O-efficient well-separated pair decomposition and its applications. *Algorithmica*, 45:585–614, 2006.

[20] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[21] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126:55–82, 2003.

[22] J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete and Computational Geometry*, 7:13–28, 1992.

[23] T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient batched range counting and its applications to proximity problems. In *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 244–255. Springer-Verlag, 2001.

[24] Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science.* Springer, 2003.

[25] G. Narasimhan and M. Smid. *Geometric Spanner Networks.* Cambridge University Press, Cambridge, UK, 2007.

[26] J. Ruppert and R. Seidel. Approximating the d-dimensional complete Euclidean graph. In *Canadian Conference in Computational Geometry*, volume 3, pages 207–210, 1991.

[27] J. S. Salowe. Constructing multidimensional spanner graphs. *International Journal on Computational Geometry & Applications*, 1:99–107, 1991.

[28] M. Smid. Closest-point problems in computational geometry. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*. North-Holland, 2000.

[29] P. M. Vaidya. A sparse graph almost as good as the complete graph on points in K dimensions. *Discrete & Computational Geometry*, 6:369–381, 1991.

[30] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33, 2001.

[31] A. C. Yao. On constructing minimum spanning trees in k-dimensional space and related problems. *SIAM Journal on Computing*, 11:721–736, 1982.

[32] N. Zeh. *I/O Efficient Algorithms for Shortest Path Related Problems.* PhD thesis, School of Computer Science, Carleton University, Ottawa, Canada, 2002.