I/O-Efficient Shortest Path Queries in Geometric Spanners

Anil Maheshwari[1,⋆], Michiel Smid[2], and Norbert Zeh[1,⋆]

[1] School of Computer Science, Carleton University, Ottawa, Canada
{maheshwa,nzeh}@scs.carleton.ca
[2] Fakultät für Informatik, Otto-von-Guericke-Universität, Magdeburg, Germany
michiel@isg.cs.uni-magdeburg.de

Abstract. We present I/O-efficient algorithms to construct planar Steiner spanners for point sets and sets of polygonal obstacles in the plane, and for constructing the "dumbbell" spanner of [6] for point sets in higher dimensions. As important ingredients to our algorithms, we present I/O-efficient algorithms to color the vertices of a graph of bounded degree, answer binary search queries on topology buffer trees, and preprocess a rooted tree for answering prioritized ancestor queries.

1 Introduction

Motivation: Geometric spanners are sparse subgraphs of the complete Euclidean graph over a set of points in $\mathbb{R}^d$. They play a key role in efficient algorithmic solutions for several fundamental geometric problems. Several efficient algorithms for constructing spanners in Euclidean space are known, including I/O-efficient algorithms [12], thereby enabling the processing of much bigger data sets that do not fit into internal memory. With respect to geometric shortest path problems, in internal memory, spanners are useful because they are sparse, so that approximate shortest path queries on the complete Euclidean graph, whose size is $\Theta(N^2)$, can be answered by solving the single-source shortest path (SSSP) problem on a graph of size $O(N)$. In external memory, sparseness is not sufficient to obtain I/O-efficient algorithms, as the best known single source shortest path algorithm takes $O(|V| + (|E|/B)\log_2|E|)$ I/Os [14]. The focus of this paper is to construct spanners in such a way that spanner paths can be reported I/O-efficiently.

Computational Model and Previous Results: In the Parallel Disk Model (PDM) (see [17]), an external memory (EM) consisting of D disks is attached to a machine with an internal memory of size M. Each of the D disks is divided into blocks of B consecutive data items. Up to D blocks, at most one per disk, can be transferred between internal and external memory in a single I/O-operation. The complexity of an algorithm in this model is the number of I/O operations it performs. It has been shown that sorting an array of size N takes $\text{sort}(N) = \Theta((N/DB)\log_{(M/B)}(N/B))$ I/Os in the PDM (see [17]). Scanning an array of size N takes $\text{scan}(N) = \Theta(N/DB)$ I/Os. Due to the lack of space, we are forced to omit a discussion of previous related work, except for the most relevant results. However, we refer the reader to [15] for geometric spanners, [9] for the

⋆ Research supported by NSERC and NCE GEOIDE.

well-separated pair decomposition (WSPD), and [17] for external memory models and algorithms.

In [12] an algorithm to compute the WSPD and the corresponding spanner of a point set in $\mathbb{R}^d$ in $O(\text{sort}(N))$ I/Os using $O(N/B)$ blocks of external memory has been proposed. By carefully choosing spanner edges, the diameter of the spanner graph is shown to be at most $2\log_2 N$. Reporting a spanner path in this spanner takes $O(1)$ I/Os per edge in the path. Moreover, a lower bound of $\Omega(\min\{N, \text{sort}(N)\})$ I/Os for computing a t-spanner of a given point set is presented.

New Results: In Sec. 3, we present an algorithm to construct the dumbbell spanner of [6] for a set of N points in $\mathbb{R}^d$ in $O(\text{sort}(N))$ I/Os, show how to compute an augmented spanner of size $O(N)$ and spanner diameter $O(\log_2 N)$ (resp. $O(\alpha(N))$) and how to report spanner paths in these two spanners in $O(\log_2 N/(DB))$ (resp. $O(\alpha(N))$) I/Os. These spanners are induced by a constant number of rooted trees, called dumbbell trees, so that reporting a spanner path reduces to reporting a path in one of these trees. The latter can be done I/O-efficiently [13]. To construct dumbbell spanners, we have to solve several interesting subproblems, including vertex coloring of graphs of bounded degree, answering prioritized ancestor queries, and answering queries on topology buffer trees (see Sec. 2). In Sec. 4, we present an external version of the algorithm of [5] to construct in $O(\text{sort}(N))$ I/Os a planar Steiner spanner of size $O(N)$ and spanning ratio $1+\varepsilon$ for a given set of N points, or polygonal obstacles with N vertices in the plane. Planarity is desirable, as planar graphs can be blocked [1], and an I/O-efficient single source shortest path algorithm for embedded planar graphs is known [3]. Also, planar graphs can be preprocessed for fast shortest path queries [13].

Preliminaries: A Euclidean graph $G = (V, E)$ is a *t-Steiner spanner* for the complete Euclidean graph $\mathcal{E}(S)$ defined on a set S of points in $\mathbb{R}^d$, if $S \subseteq V$ and for every pair of vertices $p, q \in S$, $\text{dist}_G(p,q) \leq t \cdot \text{dist}_2(p,q)$. The vertices in $V \setminus S$ are called *Steiner points*. G is a t-Steiner spanner for the visibility graph $\mathcal{V}(P)$ defined on a set P of polygonal obstacles with vertex set S, if $S \subseteq V$ and no edge in G crosses the interior of any obstacle in P, and for every pair of vertices $p, q \in S$, $\text{dist}_G(p,q) \leq t \cdot \text{dist}_{\mathcal{V}(P)}(p,q)$. We call graph G a *t-spanner* if $V = S$.

Given an axes-parallel box R and a point set S contained in R, a *fair split* of R is a partition of R into two boxes R_1 and R_2, each containing at least one point in S, using an axes-parallel hyperplane H; the distance of H from the two sides of R parallel to H has to be at least $\ell/3$, where ℓ is the shortest side of the bounding box of S. Given a point set S, let $R(S)$ be its bounding box, and $\hat{R}(S)$ be the smallest axes-parallel hypercube containing S and centered at the center of $R(S)$. The following recursive procedure defines a *fair split tree* $T(S)$ for S: If $|S| = 1$, then $T(S)$ consists of the single node S. Otherwise, we split $\hat{R}(S)$ into two non-empty boxes $R'(S_1)$ and $R'(S_2)$, using a fair split. $\hat{R}(S_i)$ is defined as an axes-parallel box of aspect-ratio at most 3 which is contained in $R'(S_i)$, contains S_i, can be partitioned using a fair split, and such that every side e of $\hat{R}(S_i)$ either coincides with the corresponding side e' of $R'(S_i)$ or is at distance at least $l'/3$ from e', where l' is the length of the side of $R'(S_i)$ perpendicular to e'. Now recursively compute trees $T(S_1)$ and $T(S_2)$ and make their roots children of S. Note that every node in $T(S)$ corresponds to a unique subset of S. We identify this subset with the node.

A *well-separated pair decomposition* (WSPD) of S consists of a fair split tree $T(S)$ and a set of pairs $\{\{A_1,B_1\},\{A_2,B_2\},\ldots,\{A_m,B_m\}\}$ such that A_i,B_i are nodes in $T(S)$, for $1 \le i \le m$, for every pair of points $p,q \in S$, there is a unique pair $\{A_i,B_i\}$ such that $p \in A_i$ and $q \in B_i$, and for all $1 \le i \le m$, $\text{dist}_2(p,q) \ge s \cdot \text{dist}_2(x,y)$, for all $p \in A_i$, $q \in B_i$, and either $x,y \in A_i$ or $x,y \in B_i$. The real number $s > 0$ is called the *separation constant*.

2 Techniques

Coloring Graphs of Bounded Degree: Given a graph G with N vertices and maximal degree bounded by some constant Δ, the following algorithm colors the vertices of G with $\Delta+1$ colors so that any two adjacent vertices in G have different colors: Number the vertices of G in their order of appearance in the given vertex list, and direct the edges of G from the vertices with smaller numbers to those with larger numbers. We now process the vertices by increasing numbers, one at a time. When processing a vertex we color it with the smallest color different from the colors of its in-neighbors. This technique can easily be realized in $O(\text{sort}(N))$ I/Os using the time-forward processing technique [10, 2].

Prioritized Ancestor Queries: Given a rooted tree T and an assignment of priorities $\text{priority}(v) \in \{0,1,\ldots,k\}$ to the vertices of T, we want to build a data structure $\mathcal{D}$ that allows answering queries of the following type in $O(1)$ I/Os: Given a vertex v and an ancestor u of v in T, find the highest vertex priority h on the path π from v to u, and report the first vertex $\text{first}(v,u)$ and the last vertex $\text{last}(v,u)$ on π with priority h. We call these queries *prioritized ancestor queries*. We show how to find $\text{first}(v,u)$. A slight modification of this procedure finds $\text{last}(v,u)$. We augment T with an artificial root r with $\text{priority}(r) = k+1$, and make r the parent of the original root of T. Let $W_i = \{v \in T : \text{priority}(v) = i\}$. Then every node $v \in W_i$, $i \le k$, has an ancestor of higher priority. For every node v, let $p'(v)$ be the lowest ancestor u of v in T such that $\text{priority}(u) > \text{priority}(v)$. Then we define a tree T' by making v the child of $p'(v)$.

Lemma 1. *Given a node $v \in T$ and an ancestor u of v in T, $\text{first}(v,u) = u$ or $\text{first}(v,u)$ is a child of the lowest common ancestor of u and v in T', denoted by $\text{LCA}_{T'}(v,u)$.*

We compute a binary tree $\mathcal{T}'$ with $|\mathcal{T}'| = O(|T|)$ from T' as follows: Replace every node v with children $w_1,\ldots,w_t$ by a path $v_1,\ldots,v_t$ of new nodes, such that v_{i+1} is the right child of v_i, for $1 \le i < t$. We call v_1 the *representative* $\text{rep}(v)$ of v and v_t its *anchor* $\text{anchor}(v)$. Let the children $w_1,\ldots,w_t$ be sorted by decreasing depth in T. Then we make $\text{rep}(w_i)$ the left child of v_i, for $1 \le i \le t$ and give node v_i a label $\text{left}(v_i) = w_i$. The following result follows from Lemma 1 and gives an $O(1)$ I/O procedure to report $\text{first}(v,u)$.

Lemma 2. *Given a node v and an ancestor u of v in T, $\text{first}(v,u) \in \{u,z\}$, where $z = \text{left}(\text{LCA}_{\mathcal{T}'}(\text{anchor}(v),\text{anchor}(u)))$.*

We compute the parents of the nodes in T' for every set W_i separately. Let $T_0 = T$. Then we mark every vertex $w \in T_0$ with $\text{priority}(w) > 0$. For every node, we compute its lowest marked ancestor in T_0. This produces all parents $p'(v)$, $v \in W_0$. We remove all

vertices in W_0 from T_0 and make every vertex $w \notin W_0$ the child of its lowest marked ancestor in T_0. Let T_1 denote the resulting tree. We now recursively apply this procedure to T_1 to obtain all parents $p'(v)$, for $v \notin W_0$. Using time-forward processing, each recursive step takes $O(\mathrm{sort}(|T_i|))$ I/Os. Once T' has been computed, it takes $O(\mathrm{sort}(|T'|))$ I/Os to compute $\mathcal{T}'$ from T' and to preprocess $\mathcal{T}'$ for answering LCA-queries in $O(1)$ I/Os. If $|W_i| \leq c|W_{i-1}|$, for some constant $0 < c < 1$ and $1 \leq i \leq k$, we obtain the following result.

Theorem 1. *Given a rooted tree T with vertex priorities* $\mathrm{priority}(v) \in \{0,1,\ldots,k\}$, *let* $W_i = \{v \in T : \mathrm{priority}(v) = i\}$, $0 \leq i \leq k$. *If* $|W_i| \leq c|W_{i-1}|$, *for some constant* $0 < c < 1$ *and* $1 \leq i \leq k$, *it takes* $O(\mathrm{sort}(N))$ *I/Os and* $O(N/B)$ *blocks of external memory to construct a data structure $\mathcal{D}$ that allows answering prioritized ancestor queries on T in $O(1)$ I/Os.*

Querying Topology Buffer Trees: Given a binary tree T whose nodes store $O(1)$ information each, we call a binary search query q *strongly local* on T if the information stored at a node w and an ancestor v of w is sufficient to decide whether all, none, or some of the answers to q in $T(v)$ are stored in $T(w)$, and we are required to report *all* answers to q stored in T. We call q *weakly local* on T if the information stored at a node v is sufficient to decide whether $T(v)$ contains an answer to q, and we have to report *one* answer to q.

A *topology tree* $\mathcal{T}$ [11] is a balanced representation of a possibly unbalanced binary tree T. $\mathcal{T}$ has height $O(\log_2 N)$, where N is the number of nodes in T, and allows answering weakly local binary search queries on T in $O(\log_2 N)$ time. To construct $\mathcal{T}$, one starts with a tree $T_0 = T$, and recursively constructs a sequence $T_0, T_1, \ldots, T_k$ of binary trees, where T_{i+1} is obtained from T_i by contracting a carefully chosen set of edges in T_i. The vertex set of $\mathcal{T}$ is the disjoint union of the vertex sets of trees $T_0, T_1, \ldots, T_k$. A vertex v in T_{i+1} is the parent of a vertex w in T_i if v is the result of contracting an edge $\{u,w\}$, or v is a copy of w in T_{i+1} and no edge $\{u,w\}$ in T_i has been contracted.

If T is static, we can extend the idea of [7] to obtain a topology buffer tree. We construct a topology tree $\mathcal{T}$ for T and cut it into layers of height $\log_2(M/B)$. Each layer is a collection of rooted trees. We contract each such tree into a single node. The resulting tree $\mathcal{B}$ is the topology buffer tree corresponding to T. $\mathcal{B}$ has height $O(\log_{(M/B)} N)$; each node of $\mathcal{B}$ represents a subtree of $\mathcal{T}$ of size $O(M/B)$ and has at most M/B children. Thus, every node of $\mathcal{B}$ fits into internal memory. Combining the ideas of topology B-trees [7] and buffer trees [2], we obtain the following result.

Theorem 2. *Given a topology buffer tree $\mathcal{B}$ representing an N node binary tree T and $O(N)$ (weakly or strongly) local binary search queries, it takes* $O(\mathrm{sort}(N+K))$ *I/Os and* $O((N+K)/B)$ *blocks of external memory to answer all queries, where K is the size of the output, provided that* $M \geq B^2$.

3 Spanners of Low Diameter

Given a point set S in $\mathbb{R}^d$, we want to construct linear size spanner graphs of spanner diameters $O(\log_2 N)$ and $O(\alpha(N))$ that can be represented by data structures to report spanner paths in $O(\log_2 N/(DB))$ and $O(\alpha(N))$ I/Os, respectively.

3.1 Dumbbell Trees

Given a WSPD $\mathcal{D}$ for S, let $T(S)$ be the fair split tree of $\mathcal{D}$, and $C = \hat{R}(S)$. We refer to the well-separated pairs of $\mathcal{D}$ as *dumbbells* (as they look like dumbbells if we connect the two centers of the bounding boxes of each well-separated pair by a straight line segment). We define the *length* of a dumbbell to be the length of this line segment. Refer to the two bounding boxes as the *heads* of the dumbbell. Also, refer to C as a head (which does not belong to any dumbbell). We want to partition the set of dumbbells into a constant number of groups such that the lengths of two dumbbells in the same group differ by a factor of at most 2 or by a factor of at least $1/\delta$, for some $0 < \delta < \frac{1}{2}$ to be defined later; the heads of two dumbbells in the same group whose lengths differ by a factor of at most two are required to have distance at least $c\ell$ from each other, where c is a constant to be specified later, and ℓ is the length of the shorter dumbbell. We call the former the *length grouping* property; the latter the *separation* property.

For every such group $\mathcal{G}$ of dumbbells we define a *dumbbell tree* $T_{\mathcal{G}}$ as follows: $T_{\mathcal{G}}$ contains one *dumbbell node* per dumbbell in $\mathcal{G}$, one *head node* per dumbbell head of the dumbbells in $\mathcal{G}$, and one node per point in S. The points in S are the leaves of $T_{\mathcal{G}}$. The head node corresponding to the special head C is the root of $T_{\mathcal{G}}$. For every dumbbell $\{A, B\}$, heads A and B are the children of $\{A, B\}$. The parent of dumbbell $\{A, B\}$ is the smallest head node containing one of its heads. Thus, every node, except C, has a well-defined parent. Such a tree can be computed in $O(\text{sort}(N))$ I/Os per group by marking all nodes in the fair split tree corresponding to dumbbell heads in the group, making every leaf of the fair split tree a child of its lowest marked ancestor, and making every dumbbell node the child of the lowest marked ancestor of one of its heads. Leaves and dumbbell nodes without marked ancestors are children of the head node C.

In order to compute groups $\mathcal{G}$ with the above properties, we first compute $O(1)$ groups having the length-grouping property and then refine these groups to ensure the separation property. The length grouping property can be guaranteed by simulating the algorithm of [6] in external memory, which takes $O(\text{sort}(N))$ I/Os. In particular, we compute a number of groups $\mathcal{G}_{i,j}$, where $0 \leq j \leq b$, for some constant b, such that each group $\mathcal{G}'_j = \bigcup_i \mathcal{G}_{i,j}$ has the length grouping property and the dumbbells in each group $\mathcal{G}_{i,j}$ differ by a factor of at most two in length. To guarantee the separation property, we partition each group $\mathcal{G}_{i,j}$ into $O(1)$ subgroups $\mathcal{G}_{i,j,k}$ such that the dumbbells in each subgroup satisfy the separation property. We merge groups $\mathcal{G}_{i,j,k}$ into $O(1)$ groups $\mathcal{G}'_{j,k} = \bigcup_i \mathcal{G}_{i,j,k}$, each having the length grouping and separation properties. Consider one particular group $\mathcal{G}_{i,j}$, and let ℓ be the length of the shortest dumbbell in $\mathcal{G}_{i,j}$.

In order to compute groups $\mathcal{G}_{i,j,k}$, we need to modify the dumbbells in $\mathcal{D}$ slightly. Consider a dumbbell $D = \{A, B\}$, and let $D' = \{A', B\}$ be its parent in the computation tree.[1] Then A' has been split into two boxes A_1 and A_2, where A is contained in A_1. In the following, we will consider $\{A, B\}$ to be dumbbell $\{A_1, B\}$. It follows from the properties of a fair split tree and its WSPD that the shortest side of head A_1 has length at least $\ell' = \frac{\ell}{(3s+12)\sqrt{d}}$.

[1] See [9] for the definition of computation trees. Intuitively, A' is the parent of A in the fair split tree, $\{A', B\}$ is not well-separated, and A' is larger than B.

The core of our algorithm is the construction of a *proximity graph* $\mathcal{P}$ containing one vertex per dumbbell in $\mathcal{G}_{i,j}$ and an edge between two vertices if the two corresponding dumbbells are too close. We do this as follows: For every dumbbell $\{A,B\} \in \mathcal{G}_{i,j}$, put a box $\mathcal{B}$ of side length $(c+8/s+4)\ell$ around the center of head A. Then every dumbbell $\{E,F\} \in \mathcal{G}_{i,j}$ that is too close to $\{A,B\}$ must have both its heads within this box. Partition $\mathcal{B}$ into $O(1)$ grid cells of side length $\ell'/2$. Then head E_1 must contain at least one of the grid vertices because it has side length at least ℓ'. Thus, if p is a grid point generated by dumbbell $\{A,B\}$, and $\{E,F\}$ is a dumbbell whose enlarged head E_1 contains p, we add an edge between the vertices corresponding to $\{A,B\}$ and $\{E,F\}$ to $\mathcal{P}$. Next we show how to find all dumbbell heads E_1 containing a grid point p.

The set of dumbbell heads containing a point p are stored along a path in the fair split tree T. Only a constant number of them can be heads of dumbbells in $\mathcal{G}_{i,j}$, as the minimal side length of a dumbbell head in $\mathcal{G}_{i,j}$ is at most $2\ell/s$, the minimal side length of the parent of a dumbbell head in $\mathcal{G}_{i,j}$ is at least ℓ', and the side lengths of the boxes along a root-to-leaf path in the fair split tree decrease by a factor of $2/3$ every d steps. For every grid point p, we report all these heads using strongly local binary search on T. The total number of heads reported for all grid points and all dumbbells is $O(N)$. It takes sorting and scanning to find the dumbbells in $\mathcal{G}_{i,j}$ having the reported heads and to add the corresponding edges to $\mathcal{P}$. It follows from standard packing arguments that $\mathcal{P}$ has bounded degree, so that we can compute a vertex coloring of $\mathcal{P}$ with a constant number of colors. The resulting color classes are the desired subgroups $\mathcal{G}_{i,j,k}$ of $\mathcal{G}_{j,k}$.

Lemma 3. *Given a point set S in $\mathbb{R}^d$ and a WSPD $\mathcal{D}$ for S, it takes $O(\mathrm{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory to partition the dumbbells of $\mathcal{D}$ into $O(1)$ groups, each having the length grouping and separation properties. Each group can be represented by a dumbbell tree. The construction of all dumbbell trees takes $O(\mathrm{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory.*

3.2 Spanners of Logarithmic Diameter

Let $T_1,\ldots,T_r$ be the dumbbell trees constructed in the previous section. Then we construct graphs $G_1,\ldots,G_r$, each having vertex set S, from those trees. We merge all these graphs $G_1,\ldots,G_r$ into a spanner G.

We construct G_i as follows: For every node $v \in T_i$, let $\omega(v) = |T_i(v)|$. We choose a representative point $r(v)$, for every node $v \in T_i$. If v is a leaf, then $r(v)$ is the point represented by v. Otherwise, let $w_1,\ldots,w_k$ be the children of v in T_i. Then $r(v) = r(w_j)$, where $\omega(w_j) = \max\{\omega(w_h) : 1 \le h \le k\}$. We add an edge $\{r(v),r(w)\}$ to G_i, for every edge $\{v,w\}$ in T_i with $r(v) \neq r(w)$.

For two points $p,q \in S$, let $\{A,B\}$ be the unique dumbbell such that $p \in A$ and $q \in B$. Let T_i be the dumbbell tree containing the dumbbell node corresponding to $\{A,B\}$, and let v and w be the two leaves of T_i such that $p = r(v)$ and $q = r(w)$. There is a unique path $\pi = (v = v_0, v_1, \ldots, v_k = w)$ from v to w in T_i. This path corresponds to a path $\tilde{\pi} = (p = r(v_0), r(v_1), \ldots, r(v_k) = q)$ in G_i. It is shown in [6] that $\tilde{\pi}$ has length at most $t \cdot \mathrm{dist}_2(p,q)$ if we choose $s = O(d/(t-1))$, $\delta = 1/s$, and $c = 2/\delta$ in the construction of the dumbbell trees. Thus, graph G is a t-spanner. Moreover, once we know tree T_i such that G_i contains the spanner path $\tilde{\pi}$ as constructed above, we can easily report $\tilde{\pi}$

by traversing the paths from v and w to their LCA in T_i; but π may be much longer than $\tilde{\pi}$ because many nodes along π may have the same representative. Observe, however, that all nodes in T_i with the same representative $r(v)$ form a path from the leaf ℓ with $r(\ell) = r(v)$ to some ancestor of ℓ in T_i. We construct a tree T_i' by compressing all non-leaf nodes on such a path into a single node. It follows from the choice of representatives in T_i that tree T_i' has height at most $\log_2 N + 1$, so that we can report $\tilde{\pi}$ in $O(\log_2 N/(DB))$ I/Os [13]. Unfortunately, we do not know which of the dumbbell trees contains the dumbbell node corresponding to $\{A,B\}$. However, as there are only $O(1)$ dumbbell trees, we can afford to query all dumbbell trees and report the shortest path found.

Theorem 3. *It takes $O(\mathrm{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory to construct a t-spanner of spanner diameter $O(\log_2 N)$ and size $O(N)$ for a given set S of N points in $\mathbb{R}^d$, along with a data structure using $O(N/B)$ blocks of external memory that allows reporting a t-spanner path with $O(\log_2 N)$ edges between any two query points in $O(\log_2 N/(DB))$ I/Os.*

3.3 Spanners of Nearly Constant Diameter

Next we present an I/O-efficient algorithm to reduce the spanner diameter of all graphs G_i to $O(\alpha(N))$ and to construct a data structure that allows spanner paths with $O(\alpha(N))$ edges to be reported at a cost of $O(1)$ I/Os per edge. The construction is based on [16]. The idea is to augment every dumbbell tree T_i' with additional edges between nodes and subsets of their ancestors, so that the shortest monotone path from a node v to one of its ancestors contains $O(\alpha(N))$ edges, where a path is *monotone* if its nodes appear in the same order along a leaf-to-root path in T_i. Let T_i° be the resulting graph and G_i° be the supergraph of G_i containing edges $\{r(v), r(w)\}$, $\{v,w\} \in T_i^\circ$. For a node v and an ancestor u of v, let π be the path from v to u in T_i', and π° be the shortest monotone path from v to u in T_i°. Let $\tilde{\pi}$ and $\tilde{\pi}^\circ$ be the corresponding spanner paths in G_i and G_i°. Then $\tilde{\pi}^\circ$ is no longer than $\tilde{\pi}$, by the triangle inequality.

We need some definitions: Given a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ such that $f(0) = 0$ and $f(x) < x$, for $x > 0$, we define $f^{(0)}(x) = x$ and $f^{(i)}(x) = f(f^{(i-1)}(x))$, for $i > 0$. Then $f^*(x) = \min\{k \geq 0 : f^{(k)}(x) \leq 1\}$. We define a series of functions ϕ_i, where $\phi_0(x) = \lfloor\sqrt{x}\rfloor$ and $\phi_i(x) = \phi_{i-1}^*(x)$, for $i > 0$. For a forest F and a set W of vertices in F, let $F \cap W$ be the forest obtained by contracting every maximal subtree whose non-root nodes are not in W to a single node. Let $F \star W$ be the set of edges containing edges between every vertex v in W and all its ancestors and descendants u in $F \setminus W$ so that the path from u to v does not contain a vertex in $W \setminus \{v\}$. Denote the irreflexive transitive closure of a DAG G by G^+. Given two parameters $0 \leq k < x$, let $V(x,k)$ be the set of vertices in F at levels $k, k+x, k+2x, \ldots$. We call $V(x,0), \ldots, V(x,x-1)$ the *x-strided levels* of F.

The algorithm of [16] consists of two parts. The top-level procedure SHORTCUT computes the 13-strided level W of forest F which has minimum size, outputs edges $F \star W$ to be added to F, and then recursively shortcuts $F \cap W$, calling procedure RECSHORTCUT with parameter $\beta = \min\{k \geq 0 : \phi_k(h(F)) \leq 4\}$, where $h(F)$ is the maximum height of any tree in F. Procedure RECSHORTCUT computes two parameters $x_1 = \phi_{\beta-1}(h(F))$ and $x_2 = 3$, and the minimum x_1-strided level V_1 of F. If $\beta = 1$,

it outputs the edges in $(F \cap V_1)^+ \cup (F \star V_1)$ to be added to F and recursively calls RECSHORTCUT$(F \setminus V_1, 1)$. If $\beta > 1$, let $F_1 = F \cap V_1$, V_2 be the minimum x_2-strided level of F_1, and $F_2 = F_1 \cap V_2$; then RECSHORTCUT returns the edges in $(F \star V_1) \cup F_1 \cup F_2$ to be added to F and recursively shortcuts $F \setminus V_1$ and F_2 by invoking RECSHORTCUT$(F \setminus V_1, \beta)$ and RECSHORTCUT$(F_2, \beta - 1)$.

It is shown in [16] that the graph T° produced by augmenting a rooted tree T with the output of SHORTCUT(T) has size $O(|T|)$. The shortest monotone path in T° from a node v to any of its ancestors u in T contains $O(\alpha(|T|))$ edges. A data structure to find the shortest monotone path in T° between two query vertices v and u can be derived quite naturally from the computation of algorithm SHORTCUT. In particular, denote the input forest to SHORTCUT by $F = F_{2\beta+1}$, and consider the tree $\mathcal{R}$ of recursive calls to RECSHORTCUT triggered by SHORTCUT. Then for every β, the recursive invocations with parameter β form a set of paths in $\mathcal{R}$. For each such path p_i, let I_i be the topmost node in $\mathcal{R}$. Then we define a forest $F_{2\beta}$ as the union of the input forests to all such top-level invocations I_i. We define another forest $F_{2\beta-1}$ as the union of forests F_1 in the description of RECSHORTCUT for invocations with parameter β. If $\beta = 1$, then forest $F_{2\beta-1}$ does not exist. Thus, we obtain a sequence $F_{2\beta+1}, \ldots, F_2$ of forests.

Given a forest F which is the input to invocation I, let $\mathcal{J}_1, \ldots, \mathcal{J}_k$ be the descendants of I in $\mathcal{R}$ that represent invocations of RECSHORTCUT with parameter β. That is, $I = \mathcal{J}_1$ represents RECSHORTCUT(F, β), $\mathcal{J}_2$ represents RECSHORTCUT$(F \setminus V_1, \beta)$, and so on. Then every node v of F appears in the set V_1 for at most one invocation $\mathcal{J}_i$. We give v priority i. For forests $F_{2\beta-1}$, we give nodes in $F_1 \cap V_2$ priority 1. For forest $F_{2\beta+1}$, all nodes in W get priority 1. All nodes with no priority label are assumed to have infinite priority.

Given this labeling, a shortest monotone path from a node $v = v_{2\beta+1}$ to its ancestor $u = u_{2\beta+1}$ can be found as follows: Since $F = F_{2\beta+1}$, we start by examining $F_{2\beta+1}$. Given two nodes v_γ and u_γ whose shortest monotone path in $F_2 \cup F_3 \cup \cdots \cup F_\gamma$ we want to find, we find the minimum priority i such that there is a vertex on the path from v_γ to u_γ in F_γ with priority i, and report the lowest and highest such vertices $v_{\gamma-1}$ and $u_{\gamma-1}$ on this path. There have to be edges $\{v_\gamma, v_{\gamma-1}\}$ and $\{u_{\gamma-1}, u_\gamma\}$ in T°. If $\gamma = 2$, there is also an edge $\{v_{\gamma-1}, u_{\gamma-1}\}$ in T°. Otherwise, we recursively find the shortest monotone path from $v_{\gamma-1}$ to $u_{\gamma-1}$ in $F_2 \cup F_3 \cup \cdots \cup F_{\gamma-1}$. If there is no vertex with finite priority on the path from v_γ to u_γ in F_γ, this path must be short. We report this path by traversing F_γ and do not recurse.

Finding vertices $v_{\gamma-1}$ and $u_{\gamma-1}$ in forest F_γ for two query vertices v_γ and u_γ is a prioritized ancestor query; we just report minimum priority ancestors instead of maximum priority ancestors. Thus, given data structures $\mathcal{F}_{2\beta+1}, \ldots, \mathcal{F}_2$ to answer prioritized ancestor queries on $F_{2\beta+1}, \ldots, F_2$, the above shortest path procedure takes $O(1)$ I/Os per step along the shortest path from v to u in T°, $O(\alpha(N))$ I/Os in total. It remains to show that these data structures can be built I/O-efficiently. The following lemma is crucial for this.

Lemma 4. *Given a forest F_γ, $2 \le \gamma \le 2\beta + 1$, and a partitioning of the vertices of F_γ into subsets $W_1, \ldots, W_k$ such that W_i contains all vertices in F_γ having priority i, $1 \le i \le k$, $|W_i| \le \frac{1}{2}|W_{i+1}|$, for $1 \le i < k$.*

Forests $F_{2\beta+1}, \ldots, F_2$ are easily obtained by simulating the computation of procedures SHORTCUT and RECSHORTCUT. By Theorem 1 and Lemma 4, it takes $O(\mathrm{sort}(N))$ I/Os to compute data structures $\mathcal{F}_{2\beta+1}, \ldots, \mathcal{F}_2$ from forests $F_{2\beta+1}, \ldots, F_2$, as the total size of the forests $F_{2\beta+1}, \ldots, F_2$ is $O(N)$ [16].

Theorem 4. *It takes $O(\mathrm{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory to construct a t-spanner of spanner diameter $O(\alpha(N))$ and size $O(N)$ for a given set S of N points in $\mathbb{R}^d$, along with a data structure using $O(N/B)$ blocks of external memory that allows reporting a t-spanner path with $O(\alpha(N))$ edges between any two query points in $O(\alpha(N))$ I/Os.*

4 Planar Steiner Spanners

Given a set P of simple polygonal obstacles with vertex set S in the plane, we want to construct a planar Steiner spanner G of size $O(|S|)$ and spanning ratio $1+\varepsilon$ for the visibility graph $\mathcal{V}(P)$ of P. Our algorithm follows the framework of [5]. It constructs a planar subdivision based on the position of the vertices in S and then combines this subdivision with the subdivision defined by the obstacle edges to obtain an L_1-Steiner spanner. A planar Euclidean Steiner spanner is computed by superimposing a constant number of planar L_1-Steiner spanners.

4.1 Planar L_1-Steiner Spanners for Point Sets

We make frequent use of a procedure interval(s,r) that partitions the segment s into subsegments of length r each by adding Steiner vertices on s. The following planar subdivision D' of a minimal axes-parallel square C containing all points of S is the basis for our spanner construction. The cells of D' are of two types. Let a *box* be an axes-parallel rectangle of aspect ratio at most 3. A *box cell* is a box and contains exactly one point of S. A *donut cell* is the set-theoretic difference of two boxes B and B', does not contain any point of S, and for every side e of B, the distance to the corresponding side e' of B' is either zero or at least $\|e'\|/6$.

Given such a subdivision D', construct a planar L_1-Steiner spanner D'' for S as follows: Perform interval$(e,\gamma\ell)$, for every edge e of D', where ℓ is the length of the shortest edge of the box to which e belongs, and $0<\gamma<1$ is an appropriately chosen constant to be defined later. For every cell R and every boundary edge e of R shoot rays orthogonal to e from the endpoints of e and from the Steiner vertices on e toward the interior of R until they meet another edge. For every box cell R containing a point $p \in S$, we also shoot rays from p in all four axes-parallel directions until they meet the boundary of R. To preserve the planarity of the resulting graph, we introduce all intersection points between such rays as Steiner vertices. The following lemma now follows from [5] and [13].

Lemma 5. *Given a set S of N points in the plane and a linear size subdivision D' as above, it takes $O(\mathrm{sort}(N)+\mathrm{scan}(N/\gamma^2))$ I/Os to construct a planar L_1-Steiner spanner of size $O(N/\gamma^2)$ and with spanning ratio $1+6\gamma$ for S.*

Subdivision D' is quite naturally derived from a fair split tree T for S. The rectangles $\hat{R}(v)$ associated with the leaves of T are the box cells of D'. These box cells cover almost all the square C containing all points in S. The uncovered parts of C can be covered by regions $R'(v) \setminus \hat{R}(v)$, where $R'(v)$ was shrunk to $\hat{R}(v)$ before splitting $\hat{R}(v)$. We include these regions as the donut cells of D'. Using the fair split tree construction of [12], and choosing $\gamma = \varepsilon/6$, we obtain the following result.

Theorem 5. *Given a set S of N points in the plane and a constant $\varepsilon > 0$, it takes $O(\mathrm{sort}(N) + \mathrm{scan}(N/\varepsilon^2))$ I/Os to construct a planar L_1-Steiner spanner of size $O(N/\varepsilon^2)$ and with spanning ratio $1+\varepsilon$.*

4.2 Planar Steiner Spanners among Polygonal Obstacles

First we construct a planar L_1-Steiner spanner for a given set P of polygonal obstacles with vertex set S in the plane. We construct the subdivision D' w.r.t. set S and combine it in an appropriate manner with the graph defined by the obstacles in P to obtain a subdivision D_2. The spanner is then constructed from D_2 in a manner similar to the construction of D''. Our algorithm to construct D_2 is based on [5]. However, we use only one (a,b)-tree to represent the sweep line status instead of using two balanced binary trees. This simplification is crucial to allow an I/O-efficient implementation of this procedure.

Let D_1 be the superimposition of subdivision $D' = (S', E')$, viewed as a graph, and the graph $D = (S, E)$ defined by the set of obstacles in P. That is, the vertex set of D_1 contains all vertices in $S \cup S'$ and all intersection points between edges in E' and E. The edge set of D_1 is obtained by splitting the edges in $E' \cup E$ at their intersection points with other edges. D_1 may have size $\Omega(N^2)$. That is why we base the construction of an L_1-spanner for P on a linear size subgraph D_2 of D_1, which we construct without constructing D_1 first.

We divide the regions of D_1 into two classes: A *red* region is a quadrilateral none of whose vertices is in $S \cup S'$. The remaining regions are *blue* regions. Let the *red graph* of D_1 be the subgraph of the dual of D_1 containing 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 the boundary of a box or donut cell. The connected components of the red graph are paths. The red regions along such a path 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*. Call the topmost horizontal rung of a ladder its *top rung*; we define *left*, *right*, and *bottom rungs* in a similar manner. All of these four types of rungs are called *extremal rungs*. We call a ladder *trivial* if it consists only of a single red region. Otherwise, it is *non-trivial*. Subdivision D_2 is obtained from D_1 by replacing every ladder in D_1 by a single region. It is shown in [5] that D_2 has size $O(N)$. Using arguments from [5] and a construction similar to that of Sec. 4.1, we obtain the following result.

Lemma 6. *Given a subdivision D_2 as defined above, it takes $O(\mathrm{sort}(N) + \mathrm{scan}(N/\gamma^2))$ I/Os to construct a planar L_1-Steiner spanner of size $O(N/\gamma^2)$ and spanning ratio $1+6\gamma$ for a given set P of polygonal obstacles with N vertices.*

In order to construct D_2, we use four plane sweeps to compute potential top, bottom, left, and right rungs. The resulting subdivision D_3 may still contain non-trivial ladders. But its size is $O(N)$, so that we can afford to construct D_3 explicitly and remove all non-extremal rungs to obtain D_2. We describe the construction of potential top rungs.

Let E'_h be the set of horizontal edges in E'. We use a bottom-up sweep to compute all potential top rungs. During the sweep, we maintain a set of intervals defined by intersections between the sweep line ℓ and obstacle edges. In particular, let $e_1, \ldots, e_k$ be the edges in E intersected by the sweep line from left to right. Then the intervals currently stored for ℓ are $(e_1, e_2), (e_2, e_3), \ldots, (e_{k-1}, e_k)$. An interval $I = (l, r)$ in this list is a *ladder interval* if there is a ladder between l and r, and we have already found at least one horizontal rung of that ladder. Otherwise, it is a *non-ladder interval*.

We start the sweep with a single non-ladder interval defined by the left and right boundaries of the square C containing the whole vertex set S. Event points of the sweep are the y-coordinates of edges in E'_h and endpoints of obstacle edges. We perform the following updates, depending on the type of event point. Let $I_1, \ldots, I_k$ be the current set of intervals defined by the sweep line. When we encounter a horizontal edge $e \in E'_h$, we find intervals I_l and I_r containing the left and right endpoints of e. Intervals $I_{l+1}, \ldots, I_{r-1}$ now become ladder intervals with their current top rungs set to e. Intervals I_l and I_r become non-ladder intervals. If I_l or I_r was classified as a ladder interval before ℓ passed e, we output the top rung for I_l or I_r, respectively. Similar procedures are applied to update the interval list when the sweep line passes an obstacle vertex.

We use a buffer tree [2] to represent the sweep line status. In particular, we store the current set of intervals sorted from left to right in this tree.[2] Every node in the buffer tree stores a time-stamped tag classifying all intervals stored in this subtree as ladder or non-ladder intervals and describing the top rung in the case of a ladder interval. These tags are chosen so that for every interval, at any time, the most recent tag along the path from the root of the tree to the leaf storing the interval represents the type and top rung of the interval correctly.

When the sweep passes a horizontal edge e, we search for the leaves l_l and l_r of T storing I_l and I_r. Denote the paths from the LCA of l_l and l_r in T to l_l and l_r by p_l and p_r. For all right siblings of nodes on p_l, we store that their descendants store ladder intervals with top rung e. We do the same for the left siblings of nodes on p_r. As this is the most recent information added to T, all intervals between I_l and I_r are now tagged as ladder intervals with top rung e. Intervals I_l and I_r are being tagged as non-ladder intervals. It is easy to find the most recent tags for I_l and I_r on the way down p_l and p_r, so that the top rungs for I_l and I_r are output if necessary. The procedures to update the tree when the sweep line passes an endpoint of an obstacle edge are similar.

Theorem 6. *Given a set of polygonal obstacles in the plane with N vertices in total, a planar L_1-Steiner spanner of spanning ratio $1+\varepsilon$ and size $O(N/\varepsilon^2)$ can be computed in $O(\mathrm{sort}(N) + \mathrm{scan}(N/\varepsilon^2))$ I/Os using $O(N/(\varepsilon^2 B))$ blocks of external memory.*

Combining the final step of the algorithm of [5] with the red-blue line intersection algorithm of [4], we obtain the following corollary.

[2] As buffer trees can only be used to represent sets drawn from a total order, we have to augment the partial left-to-right order of the obstacle segments to a total order. We show in the full paper how to do this.

Corollary 1. *Given a set of polygonal obstacles in the plane with N vertices in total, a planar Euclidean Steiner spanner of spanning ratio $1+\varepsilon$ and size $O(N/\varepsilon^4)$ can be computed in $O(\mathrm{sort}(N/\varepsilon^3)+\mathrm{scan}(N/\varepsilon^4))$ I/Os using $O(N/(\varepsilon^4 B))$ blocks of external memory.*

References

1. P.K. Agarwal, L. Arge, M. Murali, K.R. Varadarajan, J.S. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. *SODA'98*, 1998.
2. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. *WADS'95*, pp. 334–345, 1995.
3. L. Arge, G.S. Brodal, L. Toma. On external memory MST, SSSP, and multi-way planar separators. *SWAT'2000*, 2000.
4. L. Arge, D.E. Vengroff, J.S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *ESA'95*, pp. 295–310, 1995.
5. S. Arikati, D.Z. Chen, L.P. Chew, G. Das, M. Smid, C.D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. *ESA'96*, pp. 514–528, 1996.
6. S. Arya, G. Das, D.M. Mount, J.S. Salowe, M. Smid. Euclidean spanners: Short, thin, and lanky. *STOC'95*, pp. 489–498, 1995.
7. P.B. Callahan, M. Goodrich, K. Ramaiyer. Topology B-trees and their applications. *WADS'95*, pp. 381–392, 1995.
8. P.B. Callahan, S.R. Kosaraju. A decomposition of multidimensional point sets with applications to *k*-nearest neighbors and *n*-body potential fields. *J. ACM*, 42:67–90, 1995.
9. P.B. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications*. PhD thesis, Johns Hopkins, Baltimore, 1995.
10. Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter. External-memory graph algorithms. *SODA'95*, 1995.
11. Greg N. Frederickson. A data structure for dynamically maintaining rooted trees. *J. of Algorithms*, 24:37–65, 1997.
12. S. Govindarajan, T. Lukovszki, A. Maheshwari, N. Zeh. I/O-efficient well-separated pair decomposition and its applications. *ESA'2000*, pp. 220–231, 2000.
13. D. Hutchinson, A. Maheshwari, N. Zeh. An external memory data structure for shortest path queries. *COCOON'99*, pp. 51–60, 1999 (to appear in Disc. App. Maths).
14. V. Kumar, E.J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. *SPDC'96*, 1996.
15. M. Smid. Closest-point problems in computational geometry. J.-R. Sack, J. Urrutia (eds.), *Handbook of Computational Geometry*, pp. 877–936. North-Holland, 2000.
16. M. Thorup. Parallel shortcutting of rooted trees. *J. of Algorithms.*, 23:123–159, 1997.
17. J.S. Vitter. *External memory algorithms and data structures.* J. Abello, J.S. Vitter (eds.), *External Memory Algorithms and Visualization*, AMS, 1999.