

I/O-Efficiently Pruning Dense Spanners

Joachim Gudmundsson^{1*} and Jan Vahrenhold²

¹ Department of Mathematics and Computing Science, TU Eindhoven, 5600 MB, Eindhoven, The Netherlands. h.j.gudmundsson@tue.nl

² Westfälische Wilhelms-Universität Münster, Institut für Informatik, 48149 Münster, Germany. jan@math.uni-muenster.de

Abstract. Given a geometric graph $G = (S, E)$ in \mathbb{R}^d with constant dilation t , and a positive constant ε , we show how to construct a $(1 + \varepsilon)$ -spanner of G with $\mathcal{O}(|S|)$ edges using $\mathcal{O}(\text{sort}(|E|))$ I/O operations.

1 Introduction

Complete graphs represent ideal communication networks but they are expensive to build; sparse spanners represent low cost alternatives. The number of edges of the spanner network is a measure of its sparseness; other sparseness measures include the weight, the maximum degree, and the number of Steiner points. Spanners for complete Euclidean graphs as well as for arbitrary weighted graphs find applications in robotics, network topology design, distributed systems, design of parallel machines, and many other areas, and have been subject to considerable research [2, 6, 12, 16, 25]. Recently spanners found interesting practical applications in areas such as metric space searching [29, 30] and broadcasting in communication networks [3, 26].

Consider a set S of n points in the Euclidean space \mathbb{R}^d . Throughout this paper, we will assume that d is constant. A network on S can be modeled as an undirected graph G with vertex set S and with edges $e = (u, v)$ of weight $wt(e)$. We will study Euclidean networks, which are geometric networks where the weight of the edge $e = (u, v)$ is equal to the Euclidean distance $|uv|$ between its two endpoints u and v . If G is a geometric graph, then $\delta_G(p, q)$ denotes the Euclidean length of a shortest path in G between p and q . Hence, G is a t -spanner for S if $\delta_G(p, q) \leq t|pq|$ for any two points p and q of S . The minimum value t such that G is a t -spanner for S is called the *dilation* of G . A subgraph G' of G is a t' -spanner of G , if $\delta_{G'}(p, q) \leq t' \cdot \delta_G(p, q)$ for any two points p and q of S .

Many algorithms are known that compute t -spanners with $\mathcal{O}(|S|)$ edges that have additional properties such as bounded degree, small spanner diameter (i.e., any two points are connected by a t -spanner path consisting of only a small number of edges), low weight (i.e., the total length of all edges is proportional to the weight of a minimum spanning tree of S), and fault-tolerance; see, e.g., [2, 6–8, 12, 15–17, 21, 24, 25, 31, 34], and the surveys [18, 32]. All these algorithms compute t -spanners for any given constant $t > 1$. Chen *et al.* [13] showed that

* Supported by Netherlands Organisation for Scientific Research (NWO).

the lower bound for computing any t -spanner for a given set S of points in \mathbb{R}^d is $\Omega(|S| \log |S|)$ in the algebraic computation tree model.

For the analysis in this paper we use the standard two-level I/O model [1] which defines the following parameters:

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

where $N \gg M$ and $1 \leq B \leq M/2$. An *input/output operation* (or simply *I/O*) consists of reading a block of contiguous elements from disk into internal memory or writing a block from internal memory to disk. Computations can only be performed on objects in internal memory. This model of computation captures the characteristics of working with massive data sets that are too large to fit into main memory and thus are stored on disk. Examples of massive graphs include the “web graph”, telecommunication networks, or social networks [9, 19].

In the two-level I/O model, we measure the efficiency of an algorithm by the number of I/Os it performs, the amount of disk space it uses (in units of disk blocks), and the internal memory computation time. Aggarwal and Vitter [1] developed matching upper and lower I/O bounds for a variety of fundamental problems such as sorting and permuting. For example, they showed that sorting N items in external memory requires $\Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os while scanning N items in external memory obviously can be done in $\Theta(\frac{N}{B})$ I/Os. The upper bounds for sorting and for scanning N items are often abbreviated as $\mathcal{O}(\text{sort}(N)) = \mathcal{O}(\frac{N}{B} \log_{M/B} \frac{N}{B})$ and as $\mathcal{O}(\text{scan}(N)) = \mathcal{O}(\frac{N}{B})$, and we will use these notations throughout this paper.

I/O-efficient algorithms have been developed for several problem domains, including computational geometry, graph theory, and string processing. The practical merits of the developed algorithms have been explored by a number of authors. General recent surveys can be found in [4, 35], and there are also more specific surveys that consider I/O-efficient graph algorithms [23, 33]. Results related to I/O-efficiently constructing (planar) spanners for point sets, sometimes allowing Steiner points and/or respecting polygonal obstacles in the plane, have been obtained by several authors [20, 27, 28].

In this paper we consider the problem of I/O-efficiently *pruning* a given t -spanner, even if it has a super-linear number of edges. That is, given a geometric graph $G = (S, E)$ in \mathbb{R}^d with constant dilation t , and a positive constant ε , we consider the problem of constructing a $(1 + \varepsilon)$ -spanner of G with $\mathcal{O}(|S|)$ edges.¹

In the internal memory model two algorithms are known to prune a given t -spanner in time $\mathcal{O}(|E| \log |S|)$. The greedy algorithm of [16, 21] can be used to compute a $(1 + \varepsilon)$ -spanner G' of G . However, efficient implementations of the greedy algorithm are very complex. In [21], the edge set is partitioned into a logarithmic number of sets that are processed in phases. In each phase a cluster cover and a cluster graph is computed by running Dijkstra’s algorithm in parallel

¹ The constants implicit in the “Big-Oh” notation depend on $1/\varepsilon^d$.

from all the cluster centers. A simpler approach was presented in [22], using the well-separated pair decomposition, that produces a $(1 + \varepsilon)$ -spanner G' of G with $\mathcal{O}(|S|)$ edges. The I/O-efficient algorithm presented in this paper is inspired by the latter algorithm. More specifically, given a geometric graph $G = (S, E)$ in \mathbb{R}^d with constant dilation t , and a positive constant ε , we show how to I/O-efficiently construct a $(1 + \varepsilon)$ -spanner of G with only $\mathcal{O}(|S|)$ edges using $\mathcal{O}(\text{sort}(|E|))$ I/Os. This bound matches the (internal memory) complexity of the algorithm in [22].

While building a sparse spanner is asymptotically faster than pruning a dense spanner, the latter technique allows to specifically designate edges that should participate and edges that are not allowed in the sparse spanner to be constructed.

2 Preliminaries

Our algorithm is similar to the internal memory algorithm by Gudmundsson *et al.* [22] in that it uses the well-separated pair decomposition to decide which edges that can be pruned. We briefly review their algorithm in this section.

In [22] it was shown that a simple way of pruning an existing t -spanner G into a $(1 + \varepsilon)$ -spanner of G with only $\mathcal{O}(|S|)$ edges is to use the well-separated pair decomposition (WSPD). For completeness we include a description of the WSPD.

Definition 1. *Let $s > 0$ be a real number, and let A and B be two finite sets of points in \mathbb{R}^d . We say that A and B are well-separated with respect to s if there are two disjoint balls C_A and C_B , having the same radius, such that C_A contains A and, C_B contains B , and the distance between C_A and C_B is at least s times the radius of C_A . We refer to s as the separation ratio.*

Definition 2 ([11]). *Let S be a set of points in \mathbb{R}^d , and let $s > 0$ be a real number. A well-separated pair decomposition (WSPD) for S with respect to s is a sequence $\{A_i, B_i\}, 1 \leq i \leq m$, of pairs of non-empty subsets of S , such that*

1. $A_i \cap B_i = \emptyset$ for all $i = 1, \dots, m$,
2. for each unordered pair $\{p, q\}$ of distinct points of S , there is exactly one pair $\{A_i, B_i\}$ in the sequence, such that (i) $p \in A_i$ and $q \in B_i$, or (ii) $q \in A_i$ and $p \in B_i$,
3. A_i and B_i are well-separated with respect to s for all $i = 1, \dots, m$.

The integer m is called the size of the WSPD. Callahan and Kosaraju show that a WSPD of size $m = \mathcal{O}(|S|)$ can be computed in $\mathcal{O}(|S| \log |S|)$ time. Their algorithm uses a binary tree T , called the *split tree*. We briefly describe the main idea. They start by computing the bounding box of S , which is successively split by d -dimensional hyperplanes, each of which is orthogonal to one of the axes. If a box is split, they take care that each of the two resulting boxes contains at least one point of S . As soon as a box contains exactly one point, the process stops (for this box). The resulting binary tree T stores the points of S at its

leaves; one leaf per point. Also, each node u of T is associated with a subset of S . We denote this subset by S_u ; it is the set of all points of S that are stored in the subtree of u .

The split tree T can be computed in $\mathcal{O}(|S| \log |S|)$ time. Callahan and Kosaraju show that, given T , a WSPD of size $m = \mathcal{O}(|S|)$ can be computed in $\mathcal{O}(|S|)$ time. Each pair $\{A_i, B_i\}$ in this WSPD is represented by two nodes u_i and v_i of T , i.e., we have $A_i = S_{u_i}$ and $B_i = S_{v_i}$.

Even though $\sum_{i=1}^{\ell} (|A_i| + |B_i|)$ can be quadratic in $|S|$, it was shown by Callahan [10] that $\sum_{i=1}^{\ell} \min(|A_i|, |B_i|) = \mathcal{O}(|S| \log |S|)$.

Theorem 1. [11] *Let S be a set of points in \mathbb{R}^d , and let $s > 0$ be a real number. A WSPD for S with respect to s having size $\mathcal{O}(s^d |S|)$ can be computed in $\mathcal{O}(|S| \log |S| + s^d |S|)$ time.*

Now, assume that we are given a t -spanner $G = (S, E)$. Compute a WSPD $\{A_i, B_i\}$, $1 \leq i \leq m$, for S , with separation ratio $s = 4(1 + (1 + \varepsilon)t)/\varepsilon$ and $m = \mathcal{O}(|S|)$. Let $G' = (S, E')$ be the graph that contains for each i , exactly one (arbitrary) edge (x_i, y_i) of E with $x_i \in A_i$ and $y_i \in B_i$, provided such an edge exists. It holds that G' is a $(1 + \varepsilon)$ -spanner of G [22], and hence:

Fact 1 (Theorem 3.1 in [22]) *Given a real constant $\varepsilon > 0$ and a t -spanner $G = (S, E)$, for some real constant $t > 1$, one can compute a $(1 + \varepsilon)$ -spanner G' of G with $\mathcal{O}(|S|)$ edges in time $\mathcal{O}(|E| \log |S|)$.*

WSPD in external memory Govindarajan *et al.* [20] showed how to compute a split tree and the well-separated pair decomposition I/O-efficiently.

Fact 2 (Theorem 1 in [20]) *Given a set P of N points in \mathbb{R}^d and a separation constant $s > 0$, a well-separated pair decomposition for P can be computed in $\mathcal{O}(\text{sort}(N))$ I/Os using $\mathcal{O}(N/B)$ blocks of external memory.*

In the process they build a split tree T of P . The idea is to construct T recursively. They construct a partial split tree T' whose leaves have size $\mathcal{O}(N^\alpha)$ for some constant $1 - \frac{1}{6d} \leq \alpha < 1$. Then recursively build the split tree for the leaves, proceeding with an optimal internal memory algorithm for every leaf whose size is at most M . When the split tree has been computed they simulate the internal memory algorithm by Callahan and Kosaraju [11] in external memory by applying time-forward processing on the computation trees.

3 Tree-Labeling Techniques

In this section, we present three lemmas that demonstrate that a tree can be labeled I/O-efficiently in a hierarchical manner. We will use such a labeling in the pruning algorithm to efficiently compute all nodes assigned to a pair $\{A_i, B_i\}$ in the well-separated pair decomposition.

Lemma 1. *Given a tree T with N nodes, we can label all $\mathcal{O}(N)$ leaves in left-to-right order in $\mathcal{O}(\text{sort}(N))$ I/Os.*

Proof. We first compute an Euler-Tour for T using the algorithm of Chiang *et al.* [14] and, based upon this tour, we compute a labeling of the nodes according to the BFS-levels of T . The overall process takes $\mathcal{O}(\text{sort}(N))$ I/Os. Then, we again traverse T according to the Euler-Tour and, observing that leaves correspond to local minima of the BFS-level labeling, we can identify and label them during this traversal. The correctness of the left-to-right labeling follows from the fact that each node in the tree is visited in pre-order and thus, each node is visited before its right sibling. As the cost for the traversals is dominated by the cost for computing the Euler-Tour, the overall complexity of labeling the leaves is $\mathcal{O}(\text{sort}(N))$ I/Os. \square

Lemma 2. *Given a tree T with N nodes whose leaves are labeled in left-to-right order, we can label each internal node v with an interval $[l_v, r_v]$, $l_v, r_v \in \mathbb{N}$, such that the following holds:*

1. *Each leaf in the subtree rooted at v is labeled with some integer $\ell(v) \in [l_v, r_v]$.*
2. *There exists at least one leaf in the subtree rooted at v that is labeled with an integer $\ell(v) \in [l_v, r_v]$.*
3. *The interval $[l_v, r_v]$ is the minimal interval having this property.*

The I/O-cost for computing this labeling is in $\mathcal{O}(\text{sort}(N))$.

Proof. We prove Lemma 2 by giving an algorithm with an I/O-complexity $\mathcal{O}(\text{sort}(N))$ and showing that it computes a labeling with the desired properties.

The approach of this algorithm is to label the tree bottom-up and to assign to each internal node the minimal interval encompassing the intervals assigned to its children. For the “base case” of our algorithm we transform the label $\ell(v)$ assigned to a leaf v into an interval $[\ell(v), \ell(v)]$. This labeling obviously conforms with requirements of Lemma 2.

To propagate these levels upwards, we first sort the nodes of the tree according to their BFS-level in decreasing order and also label each node with its BFS-level, its BFS-number, and the BFS-number of its parent. Computing the BFS-level, the BFS-number, and the parents’ BFS-number for each node can be done using Euler-Tour techniques in $\mathcal{O}(\text{sort}(N))$ I/Os. Starting with i set to the maximum BFS-level. We then repeatedly extract all nodes on BFS-level i and $i - 1$ from the sorted array. We sort all nodes on BFS-level i according to the BFS-number of their parent and sort all nodes on BFS-level $i - 1$ according to their BFS-number. We then simultaneously scan both arrays and update each node v on BFS-level $i - 1$ with the minimum interval encompassing the intervals assigned to the nodes on BFS-level i having v as their parent (i.e. v ’s children).

Inductively, we see that the correctness of the labeling follows from the correctness of the labeling on leaf level. The overall complexity is $\mathcal{O}(\text{sort}(N))$ I/Os as the algorithm performs a constant number of Euler-Tour computations and as each node participates in a constant number of sorting steps. \square

Observation 1 *The bounds and properties derived in Lemma 1 and Lemma 2 also hold if each leaf v is labeled with an interval $[l_v, r_v]$ such that all these intervals are disjoint and the interval endpoints l_v are assigned to the leaves in increasing order from left to right.*

Note that the labeling proposed in Observation 1 can be applied to a split tree T built for the vertices of a geometric graph $G = (S, E)$. In this setting, the vertices in S are labeled according to the left-to-right order in which they appear in the leaves of T . The process described in Lemma 1 together with Observation 1 then implies a relabeling of the graph's vertices, i.e., each vertex $s \in S$ is labeled with a unique integer in $\ell(s) \in [1 \dots |S|]$. The following lemma shows that this labeling can be mapped to the edges in an I/O-efficient way.

Lemma 3. *Given a unique relabeling of the vertices of a geometric graph $G = (S, E)$, we can relabel the edges in E such that each edge $e = (v, w) \in E$ is labeled $(\ell(v), \ell(w))$ where $\ell(v), \ell(w) \in [1 \dots |S|]$ are the unique labels assigned to v and w . Given the set E of edges and a tree storing the labeled vertices in its leaves, we can relabel all edges in $\mathcal{O}(\text{sort}(|E|))$ I/Os.*

Proof. To relabel the edge, we first extract the labeled vertices from the tree using Euler-Tour techniques in $\mathcal{O}(\text{sort}(|S|))$ I/Os and sort them according to their original label. We then sort all edges according to the (original) label of their respective source vertices, and in a synchronized scan over both sorted lists, we can relabel the source vertices of all edges. Finally, we repeat this process for the list of edges sorted according to the (original) labels of their respective target vertices and obtain a relabeling of the target vertices. The above algorithm clearly runs in $\mathcal{O}(\text{sort}(|S| + |E|)) = \mathcal{O}(\text{sort}(|E|))$ time. \square

4 An Algorithm For Pruning Dense Spanners

We are now ready to describe our algorithm for I/O-efficiently pruning a dense t -spanner $G = (S, E)$ such that the resulting graph is a $(1 + \varepsilon)$ -spanner of G with $\mathcal{O}(|S|)$ edges.

Our algorithm first computes a well-separated pair decomposition $\{A_i, B_i\}$ with separation ratio $s = 4(1 + (1 + \varepsilon)t)/\varepsilon$, using the algorithm of Govindarajan *et al.* [20] and spending an overall number of $\mathcal{O}(\text{sort}(|S|))$ I/Os. The well-separated pair decomposition is represented by a split tree having $\mathcal{O}(|S|)$ leaves which is laid out on disk in $\mathcal{O}(|S|/B)$ disk blocks.

We then use the technique presented in the proof of Lemma 1 to label all *vertices* stored in the leaves from left to right and to label each leaf v with the minimal interval containing the labels of the points stored with v , that is we assign to each vertex v of the graph an unique integer $\ell(v) \in [1 \dots |S|]$. Finally, we perform a labeling of the internal nodes that fulfills the requirements of Lemma 2. By Lemma 1 and Lemma 2 the complexity computing this labeling is $\mathcal{O}(\text{sort}(|S|))$.

Lemma 4. *The above labeling of the nodes in the split tree has the property that each component C of a well-separated pair $\{A_i, B_i\}$ corresponds to an interval $[l(C), r(C)]$, $C \in \{A_i, B_i\}$, and that the points whose labels fall into $[l(C), r(C)]$ are exactly the members of the component C .*

Proof. Fix a component C of a given well-separated pair $\{A_i, B_i\}$. By definition of the split tree, there exist nodes w_{A_i} and w_{B_i} corresponding to the components of this well-separated pair. Let $v \in \{w_{A_i}, w_{B_i}\}$ be one of these nodes, and let $[l_v, r_v]$ be the interval assigned to v by the algorithm given in the proof of Lemma 2. This algorithm guarantees that there exists at least one point that is stored in the subtree rooted at v whose label falls into $[l_v, r_v]$, and that all other points stored in this subtree are also labeled with an integer $\ell \in [l_v, r_v]$. By the definition of the split tree, the points stored in the subtree rooted at v are exactly the points in the component of $\{A_i, B_i\}$ corresponding to v . For the reverse inclusion assume that there exists a point p that is not stored in the subtree rooted at v and whose label $\ell(p)$ also falls into $[l_v, r_v]$. As the points are labeled according to the left-to-right order of the leaves (see Observation 1), this means that the labels of points in the subtree rooted at v are either all less than or greater than $\ell(p)$. Assume that they are all less than $\ell(p)$. Let ℓ_{\max} be the maximum label of all elements in the subtree rooted at v . Then the labels of all elements in the subtree rooted at v are contained in $[l_v, \ell_{\max}]$. As $\ell_{\max} < \ell(p) \leq r_v$, we derive a contradiction to the minimality of $[l_v, r_v]$ (see Lemma 2). This completes the proof. \square

The algorithm of Gudmundsson *et al.* [22] prunes a dense spanner by only keeping one edge connecting the two components of each well-separated pair $\{A_i, B_i\}$ considered. Based upon Lemma 4, we can restate this pruning processes as a special case of the range-reporting problem. We first identify each edge $e = (v, w)$ in the original spanner with a point $p_e := (\ell(v), \ell(w)) \in [1 \dots |S|]^2$ (see Lemma 3). Using this terminology, we can derive the following corollary to Lemma 4:

Corollary 1. *Let T be a split tree for $G = (S, E)$ whose nodes have been labeled with intervals according to Lemma 2 and let a and b two nodes of T that correspond to a well-separated pair $\{A_i, B_i\}$. An edge $e = (v, w) \in E$ connects two vertices $v \in A_j$ and $w \in B_i$ if and only if $\ell(v) \in [l_a, r_a]$ and $\ell(w) \in [l_b, r_b]$.*

Let the set \mathcal{E} be defined as $\mathcal{E} := \{(\ell(v), \ell(w)) \in [1 \dots |S|]^2 \mid (v, w) \in E\}$. The above corollary allows us to perform the pruning algorithm for each well-separated pair $\{A_i, B_i\}$ corresponding to two nodes a and b in the split tree by performing an orthogonal range reporting query with query range $[l_a, r_a] \times [l_b, r_b]$ on the set \mathcal{E} while reporting exactly one point. Except for the edge corresponding to the point reported, all edges connecting points in A_i and B_i can be pruned, and this implies that the pruned spanner consists exactly of all edges corresponding to the results of all range queries.

What remains to show is that all range reporting queries can be performed I/O-efficiently. First of all, note that constructing the set \mathcal{E} from the set E of

edges can be done in $\mathcal{O}(\text{sort}(|E|))$ I/Os using the algorithm described in the proof of Lemma 3. In a similar way, we can construct the query ranges $[l_a, r_a] \times [l_b, r_b]$ for all pairs $\{A_i, B_i\}$ in the well-separated pair decomposition: We extract the labels of all nodes in the split tree and use two successive sort-merge steps to generate the set \mathcal{Q} of $\mathcal{O}(|S|)$ query ranges in $\mathcal{O}(\text{sort}(|S|))$ I/Os. The next lemma shows that we can I/O-efficiently process all $|\mathcal{Q}|$ range queries while reporting at most a constant number of answers per query.

Lemma 5. *Given a set \mathcal{Q} of orthogonal range queries on a set \mathcal{E} of points in the plane where $|\mathcal{Q}| \in \mathcal{O}(|\mathcal{E}|)$, we can process all queries in $\mathcal{O}(\text{sort}(|\mathcal{E}|))$ I/Os while at the same time reporting no more than two answers per query.*

Proof. We will process all queries in a batched manner using a variant of the algorithm proposed by Arge *et al.* [5]. Their algorithm can answer a set of N queries on a set of $\mathcal{O}(N)$ points in $\mathcal{O}(\text{sort}(N) + K/B)$ I/Os where K is the size of the answer set. Their algorithm utilizes a technique called *distribution sweeping* that combines multi-way divide-and-conquer with a plane-sweeping approach. Roughly speaking, the plane is subdivided into $\Theta(\sqrt{M/B})$ vertical strips each containing approximately the same number of points. Each strip is then swept top-to-bottom, and for each query range spanning a strip, all points inside the range are reported. The algorithm is then applied recursively to (parts of) query ranges falling within one of the strips. As, on each level, each query range appears at most three times (at most one “middle” part spanning one or more strips and at most two “excess” parts falling within a strip) there are $\mathcal{O}(|\mathcal{Q}|) = \mathcal{O}(|\mathcal{E}|)$ query ranges (or parts thereof) on each level, and it can be shown that processing one level can be done in $\mathcal{O}(|\mathcal{E}|/B)$ I/Os. As the recursion tree is of height $\mathcal{O}(\log_{M/B} |\mathcal{E}|/B)$ and is processed top-down, the overall algorithm takes $\mathcal{O}(\text{sort}(\mathcal{E}))$ I/Os *not counting* the I/Os needed to report the answer set.

Unfortunately, we cannot bound the size of the (complete) answer set in our problem setting by a better bound than $\mathcal{O}(|\mathcal{Q}| \cdot |\mathcal{E}|)$, and thus we need to modify the algorithm of Arge *et al.* as follows: we first use one top-down-sweep to process the query ranges on the current level of recursion with respect to their “middle” parts and as soon as we find a point p contained in the current query range q , we label q with the name of p , output the answer (q, p) , and stop processing q . In a second top-down sweep we distribute the “excess” parts of the query ranges that have not been labeled with an answer to the corresponding sub-problems, that is we stop processing ranges for which we already have found an answer. As each query range appears in at most two subproblems on each level where it can span a slab, there are at most two answers that can be reported before a query range is not processed anymore, and this in turn results in an answer set of size $\mathcal{O}(|\mathcal{E}|)$. The I/O-complexity of processing one level of recursion is not affected by this modification, hence the overall algorithm runs in $\mathcal{O}(\text{sort}(|E|))$ I/Os. \square

As mentioned above, we run the algorithm described in the proof of Lemma 5 on a point set of size $\mathcal{O}(|E|)$ and a query set of size $\mathcal{O}(|S|)$, and thus we obtain an answer set of size $\mathcal{O}(|S|)$ spending no more than $\mathcal{O}(\text{sort}(|E|))$ I/Os. A simple

duplicate-removal step taking $\mathcal{O}(\text{sort}(|S|))$ I/Os then reduces the output to at most one answer per query. The $\mathcal{O}(|S|)$ edges corresponding to the points in the answer set then form the desired pruned spanner [22]. This yields our main result:

Theorem 2. *Given a geometric graph $G = (S, E)$ in \mathbb{R}^d which is a t -spanner for S for some constant $t > 1$ and given a constant $\varepsilon > 0$, we can compute a $(1 + \varepsilon)$ -spanner $G' = (S, E')$ of G with $E' \subseteq E$ and $|E'| \in \mathcal{O}(|S|)$ spending $\mathcal{O}(\text{sort}(|E|))$ I/Os.*

The obvious open problem is whether the complexity of I/O-efficiently pruning dense spanners can be improved to $\mathcal{O}(\text{scan}(|E|) + \text{sort}(|S|))$ I/Os.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.
2. I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.
3. K. M. Alzoubi, X.-Y. Li, Y. Wang, P.-J. Wan, and O. Frieder. Geometric spanners for wireless ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):408–421, 2003.
4. L. A. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*. Kluwer, 2002. 313–357.
5. L. A. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 685–694, 1998.
6. S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. In *Proc. 27th ACM Symposium on Theory of Computing*, pages 489–498, 1995.
7. S. Arya, D. M. Mount, and M. Smid. Randomized and deterministic algorithms for geometric spanners of small diameter. In *Proc. 35th IEEE Symposium on Foundations of Computer Science*, pages 703–712, 1994.
8. P. Bose, J. Gudmundsson, and P. Morin. Ordered theta graphs. *Computational Geometry: Theory and Applications*, 28:11–18, 2004.
9. A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 566–575, 2000.
10. P. B. Callahan. *Dealing with higher dimensions: the well-separated pair decomposition and its applications*. Ph.D. thesis, Department of Computer Science, Johns Hopkins University, Baltimore, Maryland, 1995.
11. 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.
12. B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. *International Journal of Computational Geometry and Applications*, 5:124–144, 1995.
13. D. Z. Chen, G. Das, and M. Smid. Lower bounds for computing geometric spanners and approximate shortest paths. *Discrete Applied Mathematics*, 110:151–167, 2001.

14. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.
15. G. Das, P. Heffernan, and G. Narasimhan. Optimally sparse spanners in 3-dimensional Euclidean space. In *Proc. 9th Annual ACM Symposium on Computational Geometry*, pages 53–62, 1993.
16. G. Das and G. Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. *International Journal of Computational Geometry & Applications*, 7:297–315, 1997.
17. G. Das, G. Narasimhan, and J. Salowe. A new way to weigh malnourished Euclidean graphs. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 215–222, 1995.
18. D. Eppstein. Spanning trees and spanners. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 425–461. Elsevier Science Publishers, Amsterdam, 2000.
19. S. Eubank, V. A. Kumar, M. V. Marathe, A. Srinivasany, and N. Wang. Structural and algorithmic aspects of massive social networks. In J. I. Munro, editor, *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 718–727, 2004.
20. S. Govindarajan, T. Lukovszki, A. Maheswari, and N. Zeh. I/O-efficient well-separated pair decomposition and its application. In *Proc. 8th European Symposium on Algorithms*, volume 1879 of *Lecture Notes in Computer Science*, pages 220–231. Springer-Verlag, 2000.
21. J. Gudmundsson, C. Levkopoulos, and G. Narasimhan. Improved greedy algorithms for constructing sparse geometric spanners. *SIAM Journal of Computing*, 31(5):1479–1500, 2002.
22. J. Gudmundsson, C. Levkopoulos, G. Narasimhan, and M. Smid. Approximate distance oracles for geometric graph. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 828–837, 2002.
23. I. Katriel and U. Meyer. Elementary graph algorithms in external memory. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 62–84. Springer, Berlin, 2003.
24. J. M. Keil. Approximating the complete Euclidean graph. In *Proc. 1st Scandinavian Workshop on Algorithmic Theory*, pages 208–213, 1988.
25. C. Levkopoulos, G. Narasimhan, and M. Smid. Improved algorithms for constructing fault-tolerant spanners. *Algorithmica*, 32:144–156, 2002.
26. X.-Y. Li. Applications of computational geometry in wireless ad hoc networks. In X.-Z. Cheng, X. Huang, and D.-Z. Du, editors, *Ad Hoc wireless networking*. Kluwer, 2003.
27. T. Lukovszki, A. Maheswari, and N. Zeh. I/O-efficient batched range counting and its applications to proximity problems. In *Proc. 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 244–255, Berlin, 2001. Springer.
28. A. Maheswari, M. Smid, and N. Zeh. I/O-efficient shortest path queries in geometric spanners. In F. Dehne, J.-R. Sack, and R. Tamassia, editors, *Proc. 7th Workshop on Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pages 287–299, Berlin, 2001. Springer.
29. G. Navarro and R. Paredes. Practical construction of metric t -spanners. In *5th Workshop on Algorithmic Engineering and Experiments*, pages 69–81. SIMA Press, 2003.

30. G. Navarro, R. Paredes, and E. Chávez. t -spanners as a data structure for metric space searching. In *9th International Symposium on String Processing and Information Retrieval*, volume 2476 of *Lecture Notes in Computer Science*, pages 298–309. Springer, 2002.
31. J. S. Salowe. Constructing multidimensional spanner graphs. *International Journal of Computational Geometry & Applications*, 1:99–107, 1991.
32. M. Smid. Closest point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 877–935. Elsevier Science Publishers, Amsterdam, 2000.
33. L. I. Toma and N. Zeh. I/O-efficient algorithms for sparse graphs. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 85–109. Springer, Berlin, 2003.
34. 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.
35. J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.