

# Pruning spanners and constructing well-separated pair decompositions in the presence of memory hierarchies<sup>☆</sup>

Fabian Gieseke<sup>a,\*</sup>, Joachim Gudmundsson<sup>b,1</sup>, Jan Vahrenhold<sup>a,\*</sup>

<sup>a</sup>*Faculty of Computer Science, LS XI, Technische Universität Dortmund, 44227 Dortmund, Germany*

<sup>b</sup>*NICTA, 13 Garden Street, ATP, Eveleigh NSW 2015, Australia*

---

## Abstract

Given a geometric graph  $G = (S, E)$  in  $\mathbb{R}^d$  with constant dilation  $t$ , and a positive constant  $\varepsilon$ , we show how to construct a  $(1 + \varepsilon)$ -spanner of  $G$  with  $\mathcal{O}(|S|)$  edges using  $\mathcal{O}(\text{sort}(|E|))$  memory transfers in the cache-oblivious model of computation. The main building block of our algorithm, and of independent interest in itself, is a new cache-oblivious algorithm for constructing a well-separated pair decomposition which builds such a data structure for a given point set  $S \subset \mathbb{R}^d$  using  $\mathcal{O}(\text{sort}(|S|))$  memory transfers.

*Key words:* External-memory algorithms, cache-oblivious algorithms, geometric graphs, spanners, well-separated pair decomposition

---

## 1. Introduction

A geometric network on a set  $S$  of  $n$  points in  $d$ -dimensional space is an undirected weighted graph  $G(S, E)$  with vertex set  $S$  and with edges  $e \in E$  of weight  $wt(e)$ . The edges in a geometric graph are straight-line segments connecting pairs of points in  $S$  and the weight of an edge  $e = \{p, q\}$  is equal to the distance  $|pq|$  between its two endpoints  $p$  and  $q$ . Often the space considered is the Euclidean plane, but other metrics and/or higher dimensions can be considered as well. Geometric networks naturally model many real-life networks, such as transport networks and communication networks. When designing a network for a given set  $S$  of points, several criteria can be taken into account. In many applications it is important to ensure a fast connection between every pair of points in  $S$ . For this it would be ideal to have a direct connection between every pair of points but in most applications this is unacceptable due to the high costs. This leads to the concept of spanners, as defined next.

Let  $\delta_G(p, q)$  denote the length of the shortest path in a graph  $G(S, E)$  between two points  $p$  and  $q$  in  $S$ . A graph  $G$  with vertex set  $S$  is a  $t$ -spanner for  $S$  if  $\delta_G(p, q) \leq t|pq|$  for any two points  $p$  and  $q$  of  $S$ . The

---

<sup>☆</sup>A preliminary version of this paper considering only the external-memory model of computation has been presented at the *Japan Conference on Discrete and Computational Geometry 2004* [1].

\*Corresponding author

*Email addresses:* [fabian.gieseke@cs.tu-dortmund.de](mailto:fabian.gieseke@cs.tu-dortmund.de) (Fabian Gieseke), [joachim.gudmundsson@nicta.com.au](mailto:joachim.gudmundsson@nicta.com.au) (Joachim Gudmundsson), [jan.vahrenhold@cs.tu-dortmund.de](mailto:jan.vahrenhold@cs.tu-dortmund.de) (Jan Vahrenhold)

<sup>1</sup>NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

minimum value  $t$  such that  $G$  is a  $t$ -spanner for  $S$  is called the *dilation*, or *stretch factor*, 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  $S$ .

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, 3, 4, 5, 6]. Recently, spanners found interesting practical applications in areas such as metric space searching [7, 8] and broadcasting in communication networks [9, 10].

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., the book by Narasimhan and Smid [11] and the surveys [12, 13, 14, 15]. Chen *et al.* [16] showed that 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 decision tree model of computation.

For the analysis in this paper we use the *cache-oblivious model* [17], which is a variant of the standard two-level *I/O model* introduced by Aggarwal and Vitter [18]. The standard two-level I/O model 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  $B$  contiguous elements from disk into internal memory or writing such 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 [19, 20].

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 [18] 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 can obviously 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.

Recently, a variety of results has been obtained in the *cache-oblivious* model of computation introduced by Frigo *et al.* [17]. In this model, we also have the parameters  $N$ ,  $M$ , and  $B$  for the analysis of an algorithm, but in the design-phase of an algorithm, the parameters of  $M$  and  $B$  must not be used. Another difference is that the cache-oblivious model usually assumes that  $M \geq B^2$  holds. This assumption, which we will also

make in this paper, is the so-called *tall cache assumption*. Nevertheless, the I/O model can be seen as a special instance of the cache-oblivious model, and any algorithm that is optimal for any  $N$  and unknown  $M$  and  $B$  will be optimal in the I/O model (but not vice-versa). Furthermore, an algorithm that is efficient in the cache-oblivious model will be efficient on any two successive levels of a multilevel hierarchical memory, as the algorithm does not depend on the specific values of  $M$  and  $B$  that are valid for the two levels in question. Frigo *et al.* developed a number of cache-oblivious algorithms; for example, they showed that sorting  $N$  items can be done in  $\Theta(\text{sort}(N))$  memory transfers, i.e., the complexity of sorting remains the same even if  $M$  and  $B$  are unknown.

I/O-efficient and cache-oblivious 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. In the literature, general surveys [21, 22] as well as more specific surveys considering, for example, I/O-efficient graph algorithms [23, 24] can be found. 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 [25, 26, 27].

In this paper, we consider the problem of efficiently *pruning* a given  $t$ -spanner in the presence of memory hierarchies if the spanner 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  $\varepsilon$ , we consider the problem of constructing a  $(1 + \varepsilon)$ -spanner of  $G$  with  $\mathcal{O}(|S|)$  edges.<sup>1</sup>

In the internal memory model, two algorithms are known to solve this problem in time  $\mathcal{O}(|E| \log |S|)$ . The greedy algorithm [5, 28] can be used to compute a  $(1 + \varepsilon)$ -spanner  $G'$  of  $G$ . However, efficient implementations of the greedy algorithm are very complex. Gudmundsson *et al.* [28], for example, partition the edge set 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 from all the cluster centers. A simpler approach uses the well-separated pair decomposition [29] and produces such a  $(1 + \varepsilon)$ -spanner  $G'$  of  $G$  having  $\mathcal{O}(|S|)$  edges within the mentioned time bound [30].

The 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  $\varepsilon$ , we will show how to efficiently construct a  $(1 + \varepsilon)$ -spanner of  $G$  with only  $\mathcal{O}(|S|)$  edges spending  $\mathcal{O}(\text{sort}(|E|))$  memory transfers in the cache-oblivious model. This bound matches the (internal memory) complexity of the algorithm given by Gudmundsson *et al.* [30]. Similar to this algorithm, the main building block of our cache-oblivious algorithm is an efficient method for constructing a well-separated pair decomposition which is of independent interest.

---

<sup>1</sup>The constants hidden in the “Big-Oh” notation depend on roughly  $(1/\varepsilon)^d$ .

## 2. Preliminaries

### 2.1. Well-Separated Pair Decompositions

We start by specifying some definitions associated with the well-separated pair decomposition (WSPD) [31, 29]. Let  $S$  be a point set in  $\mathbb{R}^d$ . A *hyperrectangle* is a Cartesian product of the form  $R = [x_1, x'_1] \times \cdots \times [x_d, x'_d] \subset \mathbb{R}^d$  of a set of closed intervals. The *length* of  $R$  in dimension  $i$  is defined as  $l_i(R) = x'_i - x_i$  and we use  $l_{max}(R)$  and  $l_{min}(R)$  to denote the maximum and minimum lengths of  $R$ , respectively. The *bounding hyperrectangle*  $R(S)$  of  $S$  is the smallest hyperrectangle containing all points of  $S$ .

Given two point sets  $A, B \subset \mathbb{R}^d$  and a real number  $s > 0$ , we say that  $A$  and  $B$  are *well-separated* with respect to  $s$  if there are two disjoint balls  $B_{(a,r)}$  and  $B_{(b,r)}$  with radius  $r \geq 0$  and centers  $a, b \in \mathbb{R}^d$  such that  $R(A) \subset B_{(a,r)}$ ,  $R(B) \subset B_{(b,r)}$  and the distance between  $B_{(a,r)}$  and  $B_{(b,r)}$  is at least  $sr$ . We refer to  $s$  as the *separation ratio*. Callahan and Kosaraju [29] defined a *well-separated pair decomposition* (WSPD) for the point set  $S$  with respect to a real constant  $s > 0$  as a sequence  $\{\{A_1, B_1\}, \dots, \{A_m, B_m\}\}$  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.

A tree  $T$  *associated* with a point set  $S$  is a binary tree whose leaves are in a one-to-one correspondence to the points in  $S$ . Furthermore, an internal node of  $T$  represents the subset of  $S$  corresponding to the leaves of  $T$  that are descendants of the node. For simplicity of exposition, and following the notation used by Govindarajan *et al.* [25], we refer to a node representing the subset  $A \subseteq S$  also as  $A$ , i.e., we use the denotation for nodes and subsets interchangeably whenever the meaning is clear from the context. A *split tree* of  $S$  is a binary tree associated with  $S$  and is defined recursively as follows: If  $|S| = 1$ , then  $T$  consists of a single node representing the singleton. Otherwise,  $T$  consists of the root  $S$  and two subtrees that are split trees for two sets on either side of an arbitrary *splitting hyperplane* that is perpendicular to one of the coordinate axes.

The *outer hyperrectangle*  $\hat{R}(A)$  of a node  $A$  in  $T$  is defined as follows: For the root  $S$  of  $T$ , the outer hyperrectangle  $\hat{R}(S)$  is the (hyper)cube with length  $l_{max}(R(S))$  and with the same center as  $R(S)$ . For any other node  $A$ , consider the hyperplane that is used for the split of the parent node of  $A$ , denoted  $p(A)$ , and which divides  $\hat{R}(p(A))$  into two hyperrectangles. The outer hyperrectangle  $\hat{R}(A)$  of  $A$  is defined as the hyperrectangle that contains  $A$ .

A *fair split* of  $A$  is a split of  $A$  where the splitting hyperplane  $H$  has a distance of at least  $l_{max}(A)/3$  from each of the two sides of  $\hat{R}(A)$  parallel to it, where  $l_{max}(A)$  is the length of the longest side of  $R(A)$ . A split tree  $T$  obtained by using only fair splits is called a *fair split tree*. A *partial fair split tree*  $T'$  of  $S$  is a

subtree of  $T$  containing the root of  $T$ , i.e., its leaves can correspond to subsets of  $S$  consisting of more than one element.

## 2.2. Pruning Dense Spanners in Internal Memory

As mentioned above, our cache-oblivious algorithm for pruning dense spanners stems from the internal memory algorithm presented by Gudmundsson *et al.* [30]; their algorithm uses a well-separated pair decomposition constructed for the vertex set of the dense graph to decide which edges can be pruned. We therefore briefly review this internal memory algorithm.

The primary component of the internal memory algorithm is the efficient construction of a WSPD. Callahan and Kosaraju [29] showed that a WSPD for a point set  $S \subset \mathbb{R}^d$  with separation ratio  $s > 0$  and having size  $\mathcal{O}(s^d|S|)$  can be computed in  $\mathcal{O}(|S| \log |S| + s^d|S|)$  time. Each pair  $\{A_i, B_i\}$  of the WSPD is represented by two nodes of a fair split tree  $T$  associated with  $S$ . In the first phase of their algorithm, such a tree  $T$  is computed in  $\mathcal{O}(|S| \log |S|)$  time. In the second phase, this tree is used to compute the WSPD of size  $\mathcal{O}(s^d|S|)$  in linear time.

**Theorem 1.** ([29]) *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  $G = (S, E)$  and a real constant  $\varepsilon > 0$ . Gudmundsson *et al.* compute a WSPD  $\{\{A_1, B_1\}, \dots, \{A_m, B_m\}\}$  for  $S$  with separation ratio  $s = 4(1 + (1 + \varepsilon)t)/\varepsilon$  and size  $m = \mathcal{O}(s^d|S|)$ . Let  $G' = (S, E')$  be the graph that contains, for each pair  $\{A_i, B_i\}$  of the WSPD, exactly one (arbitrary) edge  $\{x_i, y_i\} \in E$  with  $x_i \in A_i$  and  $y_i \in B_i$  or  $y_i \in A_i$  and  $x_i \in B_i$ , provided such an edge exists. It can be shown that  $G'$  is a  $(1 + \varepsilon)$ -spanner of  $G$  and that the edge selection process can be performed in  $\mathcal{O}(|E| \log |S|)$  time resulting in an overall runtime of  $\mathcal{O}(s^d|S| + |S| \log |S| + |E| \log |S|) = \mathcal{O}(|E| \log |S|)$  for the pruning algorithm [30].

**Theorem 2.** ([30]) *Given a real constant  $\varepsilon > 0$  and a  $t$ -spanner  $G = (S, E)$  in  $\mathbb{R}^d$  for some real constant  $t > 1$ , one can compute a  $(1 + \varepsilon)$ -spanner  $G'$  of  $G$  with  $\mathcal{O}(s^d|S|)$  edges in  $\mathcal{O}(|E| \log |S|)$  time.*

It should be noted that it is possible to compute such a sparse  $(1 + \varepsilon)$ -spanner *from scratch*, i.e. starting with only the vertices and adding  $\mathcal{O}(|S|)$  edges, in  $\mathcal{O}(|S| \log |S|)$  time [32]. The benefit of pruning, however, is that we are able to start from a given spanner, i.e., that the algorithm can be easily modified to respect prescribed edges that should appear in the final graph. As our cache-oblivious pruning algorithm is inspired by the approach depicted above, we will omit giving further details of the corresponding algorithm.

## 3. Pruning Dense Spanners in Hierarchical Memory

Similar to the internal memory algorithm, our cache-oblivious algorithm for pruning dense spanners consists of two phases: In the first phase, we compute a WSPD for the vertex set  $S$  of the  $t$ -spanner

$G = (S, E)$  with respect to  $s = 4(1 + (1 + \varepsilon)t)/\varepsilon$  and having size  $\mathcal{O}(s^d|S|)$ . The output is a sequence  $\{\{A_1, B_1\}, \dots, \{A_m, B_m\}\}$  of pairs of non-empty subsets of  $S$ . In the second phase, we traverse this sequence and collect, for each pair  $\{A_i, B_i\}$  of the WSPD, the (possibly empty) set  $E_i$  of all edges in  $E$  that connect a vertex in  $A_i$  with a vertex in  $B_i$ . If  $E_i$  contains more than one edge, we prune all edges but (an arbitrary) one from  $E_i$ . With  $E' := \cup_{i=1}^m E_i$ , the desired pruned spanner graph then is  $G' := (S, E')$ , and we have  $|E'| = \mathcal{O}(s^d|S|)$ .

The major algorithmic ingredient that allows for efficient pruning in the cache-oblivious model is to restate the pruning problem as a special range reporting problem on a two-dimensional integer grid. The grid and the point data are obtained by labeling each node with a unique integer in  $\{1, \dots, |S|\}$  derived from the fair split tree and by representing each edge by the pair of it's nodes' labels. The pairs of the WSPD then are transformed into “matching” query ranges obtained from the split tree as well, and we will show how to process these query ranges to obtain no more than one edge per pair of the WSPD.

For the sake of exposition, we will assume that we are able to compute a WSPD cache-efficiently and defer the formal proof to Section 4. Thus, it remains to discuss how to use the fair split tree returned by this computation to find the labels and query ranges mentioned above and how to use these ranges to solve the pruning problem. The main challenge associated with processing the fair split tree is that the construction algorithm returns the tree in an edge-list representation and that we have no influence over the blocking of these edges. Furthermore, when extracting the labels we need to synchronize the processing of the nodes corresponding to the subsets of the WSPD with the traversal of the tree to avoid having to perform one memory transfer per node, i.e.  $\Omega(|S|)$  memory transfers in total.

### 3.1. Tree-Labeling and Applications

Before divulging the details of the cache-oblivious pruning algorithm, we will present three lemmas that demonstrate how to label a tree in a hierarchical manner and thus to address the above mentioned problems. The tree-labeling techniques will be used to efficiently compute all edges (of the graph which has to be pruned) assigned to a specific pair  $\{A_i, B_i\}$  of the WSPD (of the graph's vertices).

Let  $T$  be an ordered, undirected tree. The *breadth-first-search (BFS) level* of a node  $v$  in  $T$  is the number of edges on the path from the root of the tree to  $v$ . The *BFS-numbering* of the nodes in  $T$  is defined by the order in which the nodes are visited in a BFS traversal of  $T$ . The BFS-levels and BFS-numbers are illustrated in Figure 1.

The first tree labeling technique labels the leaves of a given (ordered) tree in a “left-to-right” manner:

**Lemma 1.** *Given an ordered undirected tree  $T$  with  $N$  nodes, we can label all  $\mathcal{O}(N)$  leaves in left-to-right order spending  $\mathcal{O}(\text{sort}(N))$  memory transfers in the cache-oblivious model.*

**Proof:** We start by computing an Euler tour for  $T$  using the results of Arge *et al.* [33], who have shown that such a tour can be computed cache-obliviously in  $\mathcal{O}(\text{sort}(N))$  memory transfers. Based upon this tour, we

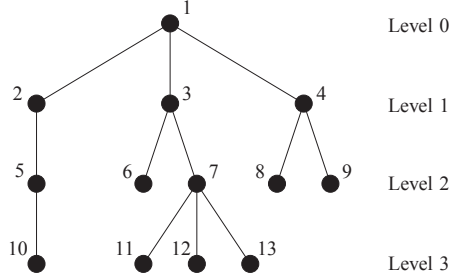


Figure 1: Illustrating the BFS level and the BFS number of a rooted tree.

compute a labeling of the nodes according to the maximum number of BFS-levels of  $T$ . By rescanning  $T$ , we can detect all leaves of  $T$  as their BFS-levels correspond to local maxima and, in the course of this traversal, we can label each leaf with respect to the designated left-to-right order (we note in passing that this labeling function is injective). The correctness of this 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. The cost for all traversals is dominated by the computation of the Euler tour. Hence, we can perform the labeling spending  $\mathcal{O}(\text{sort}(N))$  memory transfers.  $\square$

The next lemma shows that such a left-to-right labeling can be propagated upwards efficiently:

**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  $[low_v, high_v]$ ,  $low_v, high_v \in \mathbb{N}$ , such that the following holds:*

1. *Each leaf in the subtree rooted at  $v$  is labeled with some integer  $\ell(v) \in [low_v, high_v]$ .*
2. *The interval  $[low_v, high_v]$  is the minimal interval having this property.*

*The computation of this labeling can be performed spending  $\mathcal{O}(\text{sort}(N))$  memory transfers in the cache-oblivious model.*

**Proof:** We will prove this lemma by giving an algorithm which computes the labeling with the desired properties and which causes  $\mathcal{O}(\text{sort}(N))$  memory transfers. The approach of this algorithm is to label the tree bottom-up and to assign 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 the requirements of the lemma. Note that by definition, the subtree rooted at any given *internal* node  $v$  contains at least one leaf, and that it thus follows immediately from the above construction that for each internal node  $v$  there exists at least one leaf in the subtree rooted at  $v$  that is labeled with an integer  $\ell(v) \in [low_v, high_v]$

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  $\mathcal{O}(\text{sort}(N))$  memory transfers.

Starting with  $i$  set to the maximum BFS-level, we 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 amount of caused memory transfers is  $\mathcal{O}(\text{sort}(N))$  as the algorithm performs a constant number of Euler tour computations and as each node participates in a constant number of sorting steps.  $\square$

Note that the labeling in Lemma 2 can be applied to a split tree  $T$  built for the vertices of a geometric graph  $G = (S, E)$ . The process described in (the proof of) Lemma 1 then implies a relabeling of the graph's vertices, i.e., each vertex  $s \in S$  is labeled with a unique integer  $\ell(s) \in \{1, \dots, |S|\}$ . The following observation shows that this labeling can be mapped cache-obliviously to the edges in an efficient manner.

**Observation 1.** *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  $e = \{p, q\} \in E$  is labeled with  $\{\ell(p), \ell(q)\}$ , where  $\ell(p), \ell(q) \in \{1, \dots, |S|\}$  are the unique labels assigned to  $p$  and  $q$ . Given the set  $E$  of edges and a tree storing the labeled vertices in its leaves, we can relabel all edges spending  $\mathcal{O}(\text{sort}(|E|))$  memory transfers in the cache-oblivious model.*

### 3.2. The Pruning Algorithm

We are now ready to describe our algorithm for efficiently pruning a dense  $t$ -spanner  $G = (S, E)$  in a cache-oblivious manner such that the resulting graph is a  $(1 + \varepsilon)$ -spanner of  $G$  with  $\mathcal{O}(s^d|S|)$  edges. The framework of the algorithm is given by the function PRUNESPANNER (Algorithm 3.1): We start by computing a WSPD  $\{\{A_1, B_1\}, \dots, \{A_m, B_m\}\}$  of size  $m = \mathcal{O}(s^d|S|)$  with separation ratio  $s = 4(1 + (1 + \varepsilon)t)/\varepsilon$ . The WSPD is represented by a set  $\{\{A_1, B_1\}, \dots, \{A_m, B_m\}\}$  of pairs of nodes of the appropriate fair split tree  $T$  associated with  $S$ . In Steps 2 and 3, the tree-labeling techniques presented in the previous three lemmas are used to label  $T$ , to relabel the vertices in  $S$  and to map this relabeling to the edge set  $E$ . After the application of these techniques, every directed edge<sup>2</sup>  $(p, q) \in E$  can be identified with a point  $M_{(p,q)} = (\ell(p), \ell(q)) \in \{1, \dots, |S|\} \times \{1, \dots, |S|\}$ . The key idea of the pruning algorithm is based on the following lemma:

---

<sup>2</sup>We identify the undirected graph  $G$  with a directed graph having two directed edges  $(p, q)$  and  $(q, p)$  for each undirected edge  $\{p, q\}$ .

---

**Function** PRUNESPANNER( $G, \varepsilon$ )

**Require:** A  $t$ -spanner  $G = (S, E)$  with constant dilation  $t > 1$  and  $S \subset \mathbb{R}^d$  and a real constant  $\varepsilon > 0$

**Return:** A  $(1 + \varepsilon)$ -spanner  $G' = (S, E')$  of  $G$  with  $E' \subseteq E$  and  $|E'| = \mathcal{O}(s^d|S|)$

- 1:  $E' = \emptyset$
  - 2: Compute a WSPD  $\{\{A_1, B_1\}, \dots, \{A_m, B_m\}\}$  of  $S$  with separation ratio  $s = 4(1 + (1 + \varepsilon)t)/\varepsilon$ . The WSPD is represented by a set  $\{\{A_1, B_1\}, \dots, \{A_m, B_m\}\}$  of pairs of nodes of the appropriate fair split tree  $T$ . (See Section 4 below.)
  - 3: Label each node of  $T$  based on the labeling-techniques depicted in the proofs of Lemma 1 and Lemma 2. After performing the labeling, every vertex  $p \in S$  is labeled with a number  $\ell(p) \in \{1, \dots, |S|\}$ .
  - 4: Use Observation 1 to map the above labeling of  $S$  to the edge set  $E$ . After this mapping, every directed edge  $(p, q) \in E$  can be identified with a point  $M_{(p,q)} = (\ell(p), \ell(q)) \in \{1, \dots, |S|\} \times \{1, \dots, |S|\}$ .
  - 5: Construct the point set  $\mathcal{E} = \{(\ell(p), \ell(q)) \mid (p, q) \in E\}$ .
  - 6: Construct the query set  $\mathcal{Q} = \{[low_{A_i}, high_{A_i}] \times [low_{B_i}, high_{B_i}] \mid i \in \{1, \dots, m\}\}$ .
  - 7: Select for every query rectangle  $R \in \mathcal{Q}$  with  $R \cap \mathcal{E} \neq \emptyset$  exactly one point  $M_{(p,q)} \in R \cap \mathcal{E}$ .
  - 8: Add for each point  $M_{(p,q)}$  selected in Step 7 the corresponding undirected edge  $\{p, q\} \in E$  to  $E'$ .
  - 9: **return**  $G' = (S, E')$
- 

Algorithm 3.1: Pruning algorithm in the cache-oblivious model.

**Lemma 3.** *The labeling of the nodes in the split tree as described in Lemma 2 has the property that each set  $C \in \{A_i, B_i\}$  of a well-separated pair corresponds to an interval  $[low_C, high_C]$  and that the points whose labels fall into  $[low_C, high_C]$  are exactly the members of the set  $C$ .*

**Proof:** Fix a set  $C$  of a given well-separated pair  $\{A_i, B_i\}$ . By the definition of the split tree, there exist two nodes  $A_i$  and  $B_i$  in  $T$  corresponding to the two sets. Let  $v$  be the node corresponding to  $C$ , and let  $[low_v, high_v]$  be the interval assigned to  $v$  by the algorithm given in the proof of Lemma 2, and set  $[low_C, high_C] := [low_v, high_v]$ .

To prove the first inclusion (every point in  $C$  is labeled with an integer in  $[low_C, high_C]$ ), we fix a point  $p \in C$  (note that  $C \neq \emptyset$ —see the definition of a split tree and the proof of Lemma 2). By the definition of the split tree, this point is stored in (a leaf of) the subtree rooted at  $v$ , and thus Property 1 of Lemma 2 guarantees that  $\ell(p) \in [low_C, high_C]$ .

For the reverse inclusion (every point labeled with an integer from  $[low_C, high_C]$  is a member of  $C$ ) assume that there exists a point  $p$  that is not stored in the subtree rooted at  $v$  (hence, not a member of  $C$ ) whose label  $\ell(p)$  falls into  $[low_v, high_v]$  ( $= [low_C, high_C]$ ). As the points are labeled according to the left-to-right order of the leaves and since the labeling is an injective function (see Lemma 1), this means that the labels of points in the subtree rooted at  $v$  are either all strictly less than or strictly greater than  $\ell(p)$ , say strictly less than  $\ell(p)$ . Let  $\ell_{\max}$  be the maximum label of all elements in the subtree rooted at  $v$ . Then the labels

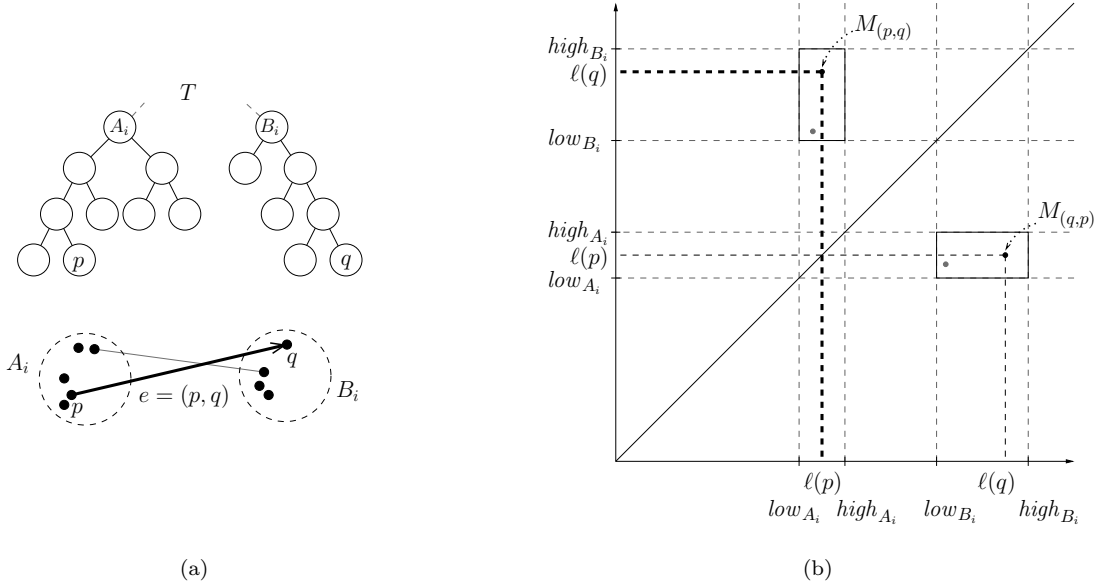


Figure 2: Mapping of edges and WSPD-pairs to points and rectangles.

of all elements in the subtree rooted at  $v$  are contained in  $[low_v, \ell_{\max}]$ . As  $\ell_{\max} < \ell(p) \leq high_v$ , we derive a contradiction to the property that  $[low_v, high_v]$  is minimal (see Lemma 2). This completes the proof.  $\square$

The internal memory pruning algorithm of Gudmundsson *et al.* [30] prunes a dense spanner by only keeping one edge connecting the two sets of each well-separated pair  $\{A_i, B_i\}$ . Based upon Lemma 3, we can restate this pruning process as a special case of the range-reporting problem: In Step 5 of function PRUNESPANNER, we construct the point set  $\mathcal{E}$  of mapped edges and in Step 6, we construct a query set  $\mathcal{Q}$ . Using this terminology, we can derive the following corollary to Lemma 3, see Figure 2:

**Corollary 1.** *Let  $T$  be a split tree of  $S$  whose nodes have been labeled with intervals according to Lemma 2 and let  $\{A_i, B_i\}$  be a pair of nodes of  $T$  that corresponds to a well-separated pair  $\{A_i, B_i\}$ . A directed edge  $e = (p, q) \in E$  connects two vertices  $p \in A_j$  and  $q \in B_i$  if and only if  $\ell(p) \in [low_{A_i}, high_{A_i}]$  and  $\ell(q) \in [low_{B_i}, high_{B_i}]$ .*

The above corollary allows us to perform the pruning algorithm for each well-separated pair  $\{A_i, B_i\}$  corresponding to two nodes  $A_i$  and  $B_i$  in the split tree by performing an orthogonal range reporting query with query range  $[low_{A_i}, high_{A_i}] \times [low_{B_i}, high_{B_i}]$  on the set  $\mathcal{E}$  while reporting exactly one point (if existent), i.e., we can conduct the whole pruning process by reporting for each  $R \in \mathcal{Q}$  with  $R \cap \mathcal{E} \neq \emptyset$  exactly one edge  $\{p, q\}$  corresponding to a point  $M_{(p,q)} \in R \cap \mathcal{E}$ .

It remains to show that all queries can be performed efficiently in the cache-oblivious model. Using the cache-oblivious algorithm for constructing a WSPD (see Section 4), Step 2 of function PRUNESPANNER takes  $\mathcal{O}(\text{sort}(|S|))$  memory transfers. Further, the application of the labeling techniques in Steps 3 and 4 can be

conducted in  $\mathcal{O}(\text{sort}(|S|))$  and  $\mathcal{O}(\text{sort}(|E|))$ , respectively. Concerning the range queries, the construction of the set  $\mathcal{E}$  in Step 5 can be done using the algorithm behind Observation 1. In a similar way, we can construct query ranges  $[low_{A_i}, high_{A_i}] \times [low_{B_i}, high_{B_i}]$  for all pairs  $\{A_i, B_i\}$  in Step 6: We extract the labels of all nodes in the split tree and use two successive sort-and-merge steps to generate the set  $\mathcal{Q}$  of  $\mathcal{O}(s^d|S|)$  query ranges in  $\mathcal{O}(\text{sort}(|S|))$  memory transfers. The next lemma shows that we can process all  $|\mathcal{Q}|$  range queries and thereby the overall pruning process in Steps 7 and 8 efficiently in a cache-oblivious manner:

**Lemma 4.** *Considering the set  $\mathcal{Q}$  of orthogonal range queries on the set  $\mathcal{E}$  of points in function PRUNESPANNER (Algorithm 3.1), we can process all queries spending  $\mathcal{O}(\text{sort}(|\mathcal{E}|))$  memory transfers in the cache-oblivious model.*

**Proof:** The range searching instance and the query ranges can each be constructed in a single scan which is cache-oblivious *per se*. To process these queries efficiently, we use the results of Brodal and Fagerberg [34] who have shown that, given a set of  $\mathcal{Q}$  of orthogonal range queries on a set  $\mathcal{E}$  of points in the plane with  $|\mathcal{Q}| = \mathcal{O}(|\mathcal{E}|)$ , one can answer these queries performing  $\mathcal{O}(\text{sort}(|\mathcal{E}|) + T/B)$  memory transfers, where  $T$  is the number of reported points. Due to the properties of a WSPD, the total number of reported points in PRUNESPANNER is bounded by  $\mathcal{O}(|\mathcal{E}|)$ . Hence, we can process all queries using an overall number of  $\mathcal{O}(\text{sort}(|\mathcal{E}|) + |\mathcal{E}|/B) = \mathcal{O}(\text{sort}(|\mathcal{E}|))$  memory transfers.  $\square$

Since the size of the WSPD is  $\mathcal{O}(s^d|S|)$ , the edge set  $E'$  of the returned graph  $G = (S, E')$  has size  $\mathcal{O}(s^d|S|)$ . This yields the following intermediate result:

**Lemma 5.** *Given a geometric graph  $G = (S, E)$  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  $|E'| \in \mathcal{O}(s^d|S|)$  spending – in the cache-oblivious model –  $\mathcal{O}(\text{sort}(|E|))$  memory transfers in addition to the memory transfers needed to compute a well-separated pair decomposition for  $S$ .*

#### 4. A Cache-Oblivious Algorithm for Constructing a Well-Separated Pair Decomposition

In this section, we will prove that it is indeed possible to efficiently construct a WSPD in the cache-oblivious model of computation spending only  $\mathcal{O}(\text{sort}(|S|))$  memory transfers.

An I/O-efficient algorithm has been developed by Govindarajan *et al.* [25] who proved the following:

**Theorem 3.** ([25]) *Given a set  $S$  of points in  $\mathbb{R}^d$  and a separation constant  $s > 0$ , a well-separated pair decomposition for  $S$  with separation ratio  $s$  having size  $\mathcal{O}(s^d|S|)$  can be computed spending  $\mathcal{O}(\text{sort}(|S|))$  I/Os and  $\mathcal{O}(|S|/B)$  blocks of external memory in the I/O-model.*

A large part of their algorithm is based on scanning and sorting or on techniques which have a cache-oblivious counterpart. However, some parts of their algorithm use the values of  $M$  and  $B$  (which cannot be

used in the design-phase of an algorithm in the cache-oblivious model) or take advantage of data structures having no (direct) cache-oblivious correspondence. In particular, Govindarajan *et al.* make use of two types of cache-aware data structures, namely *buffer trees* [35] and *topology buffer trees* [36, 37]. The replacement of components involving these data structures by efficient cache-oblivious computation steps constitutes a major part of our modifications. In the following, we will describe our cache-oblivious method by outlining the original I/O-algorithm along with the necessary modifications.<sup>3</sup>

In a nutshell, Govindarajan *et al.* [25] compute a WSPD of a given point set  $S$  as follows: They start by constructing a fair split tree  $T$  of  $S$ . The idea is to build  $T$  recursively. First, a partial fair split tree  $T'$  whose leaves have size at most  $|S|^\alpha$  for some constant  $1 - \frac{1}{2d} \leq \alpha < 1$  is constructed. Subsequently, they recursively build the split trees for the leaves, proceeding with the optimal internal memory algorithm for every leaf whose size is at most  $M$ . After the computation of the split tree they simulate the internal memory algorithm of Callahan and Kosaraju [29] for constructing the pairs  $\{A_i, B_i\}$  in external memory by applying the time-forward processing technique [35].

In the remainder of this subsection, we will describe our cache-oblivious algorithm for constructing a WSPD. As mentioned above, we can benefit from the original I/O-algorithm as most of the building blocks can be adopted directly. To keep the exposition self-contained, yet as concise as possible, we will only sketch these parts and refer the reader to the work of Govindarajan *et al.* [25] for a more detailed description of them. Whenever a part of the I/O-algorithm has to be changed, we will outline the I/O-efficient implementation of each part along with the modifications necessary to make it cache-oblivious.

#### 4.1. Construction of a Fair Split Tree

As for the internal memory algorithm, and the I/O-algorithm, the well-separated pairs of the WSPD for a given point set  $S \subset \mathbb{R}^d$  will be represented by nodes of a fair split tree  $T$  associated with  $S$ . Hence, we will start by presenting a cache-oblivious method for constructing such a tree. The algorithm is given as the function FAIRSPLITTREE (Algorithm 4.1). To construct the desired split tree, we initially provide it with the point set  $S$ , a cube  $R_0$  containing  $S$ , and a constant  $\alpha \in [1 - \frac{1}{4d}, 1)$ . The cube has side length  $l_{max}(R(S))$  and its center coincides with the one of  $R(S)$ . The fair split tree  $T$  is then computed recursively.

The structure and the analysis of function FAIRSPLITTREE are close to those of the original I/O-efficient algorithm. However, we do not stop the recursion at a leaf of size  $M$ , as this value cannot be used in the cache-oblivious model for the design of algorithms. Moreover, we chose the constant  $\alpha$  from  $[1 - \frac{1}{4d}, 1)$  instead of from  $[1 - \frac{1}{2d}, 1)$ , thus effectively but not asymptotically slowing down the recursion. This seemingly minor modification is essential for our approach and will enable the replacement of components using the buffer and topology buffer trees mentioned above. More precisely, this modification will allow for an surprisingly simple, yet asymptotically efficient, application of a nested-loop approach to solving two certain subproblems

---

<sup>3</sup>A detailed description of all modifications can be found in the thesis of Gieseke [38].

---

**Function** FAIRSPLITTREE( $S, R_0, \alpha$ )

**Require:** A non-empty point set  $S \subset \mathbb{R}^d$ , a cube  $R_0$  with  $S \subset R_0$ , and a real constant  $\alpha \in [1 - \frac{1}{4d}, 1)$

**Return:** A fair split tree  $T$  of  $S$

- 1: **if**  $|S| = 1$  **then**
  - 2:   Construct a fair split tree consisting of a single node.
  - 3: **else**
  - 4:   Apply the function PARTIALFAIRSPLITTREE (Algorithm 4.2) to  $S, R_0$ , and  $\alpha$  to get a partial fair split tree  $T'$  of  $S$ . Let  $S_1, \dots, S_k$  be the leaves of  $T'$ . Each leaf  $S_i$  has size at most  $|S|^\alpha$ .
  - 5:   **for**  $i = 1$  to  $k$  **do**
  - 6:     Apply the function FAIRSPLITTREE recursively to the point set  $S_i$ , the outer hyperrectangle  $\hat{R}(S_i)$ , and the constant  $\alpha$  to compute a fair split tree  $T_i$  of  $S_i$ .
  - 7:   **end for**
  - 8:    $T = T' \cup T_1 \cup \dots \cup T_k$
  - 9: **end if**
  - 10: **return**  $T$
- 

Algorithm 4.1: Construction of a fair split tree in the cache-oblivious model.

in Step 2(b) of our algorithm. Note that none of our modifications affects the solution of the recurrence for the number of memory transfers:

**Lemma 6.** *Given a set  $S$  of points in  $\mathbb{R}^d$ , function FAIRSPLITTREE (Algorithm 4.1) computes a fair split tree of  $S$  spending  $\mathcal{O}(\text{sort}(|S|))$  memory transfers and  $\mathcal{O}(|S|/B)$  blocks of memory in the cache-oblivious model.*

**Proof:** Assuming that the function PARTIALFAIRSPLITTREE (Algorithm 4.2) requires at most  $c \cdot \text{sort}(|S|)$  memory transfers for an appropriate chosen constant  $c > 0$ , we obtain the same recurrence as Govindarajan *et al.* for the total number of memory transfers caused by the application of FAIRSPLITTREE:

$$\mathcal{I}(|S|) \leq c \cdot \text{sort}(|S|) + \sum_{i=1}^k \mathcal{I}(|S_i|) = \mathcal{O}(\text{sort}(|S|))$$

In Lemma 4.2 it will be shown that the function PARTIALFAIRSPLITTREE computes a valid partial fair split tree  $T'$  of  $S$ . Thus, the overall algorithm correctly computes the desired split tree  $T$  of  $S$  performing  $\mathcal{O}(\text{sort}(|S|))$  memory transfers. The linear bound for the space consumption follows from the fact that the function PARTIALFAIRSPLITTREE uses  $\mathcal{O}(|S|/B)$  space [25] and from the fact that  $\sum_{i=1}^k |S_i| = |S|$ .  $\square$

---

**Function** PARTIALFAIRSPLITTREE( $S, R_0, \alpha$ )

**Require:** A non-empty point set  $S \subset \mathbb{R}^d$ , a box  $R_0$  (i.e. a hyperrectangle  $R_0$  with  $l_{\max}(R_0) \leq 3 \cdot l_{\min}(R_0)$ ) with  $S \subset R_0$ , and a real constant  $\alpha \in [1 - \frac{1}{4d}, 1)$

**Return:** A partial fair split tree  $T'$  of  $S$  whose leaves have size at most  $|S|^\alpha$

- 1: Compute a compressed pseudo split tree  $T_c$  of  $S$ , where every node represents a box.
  - 2: Expand  $T_c$  to the pseudo split tree  $T''$ .
  - 3: Remove every node of  $T''$  that does not contain any points of  $S$  and compress every path of nodes having one child into a single edge to obtain  $T'$ .
  - 4: **return**  $T'$
- 

Algorithm 4.2: Construction of a partial fair split tree in the cache-oblivious model.

#### 4.2. Construction of a Partial Fair Split Tree

It remains to show that the function PARTIALFAIRSPLITTREE (Algorithm 4.2) causes  $\mathcal{O}(\text{sort}(|S|))$  memory transfers and uses  $\mathcal{O}(|S|/B)$  blocks of memory. The framework of this function is identical to that of the original I/O-efficient algorithm for computing the partial fair split tree  $T'$ . In the first step, a binary tree  $T_c$ , called *compressed pseudo split tree*, is computed whose nodes correspond to boxes. Note that a hyperrectangle  $R$  is called a *box* if and only if  $l_{\max}(R) \leq 3 \cdot l_{\min}(R)$ . In the second step, the tree  $T_c$  is expanded to a tree  $T''$ , called a *pseudo split tree*. In the last step, leaves whose boxes do not contain any points of  $S$  are removed and paths consisting of nodes having one child are compressed to single edges. The resulting tree is the desired partial fair split tree  $T'$  of  $S$ .

We will next describe and analyze each step of function PARTIALFAIRSPLITTREE in detail.

*Step 1: Construction of  $T_c$ .* Govindarajan *et al.* [25] observe that, in principle, the tree  $T_c$  could be obtained by recursively splitting smaller and smaller hyperrectangles where the root of  $T_c$  would correspond to the starting hyperrectangle  $R_0$ . The recursion could be stopped as soon as a hyperrectangle contains at most  $|S|^\alpha$  points of  $S$ . However, this two-way recursive approach would not be efficient in the I/O model. To obtain an I/O-efficient algorithm, and following the ideas presented by Callahan [31], Govindarajan *et al.* first partition every dimension of the starting hyperrectangle  $R_0$  into slabs each containing at most  $|S|^\alpha$  points. For each dimension, these slabs are bounded by  $\lceil |S|^{1-\alpha} \rceil + 1$  axis-parallel hyperplanes, called the *slab boundaries*. The two extreme slab boundaries in each dimension are positioned in such a way that they contain the two sides of the starting hyperrectangle  $R_0$  in this dimension. The construction of the tree  $T_c$  is only based on the information provided by these slab boundaries. Once the partition is given, Govindarajan *et al.* consider three cases based on which a hyperrectangle  $R$  is split into one or two smaller hyperrectangles at each step of the recursion approach depicted above while maintaining certain invariants, see Figures 3 and 4. Let  $R'$  denote the largest hyperrectangle which is completely contained in  $R$  and whose sides lie in slab boundaries.

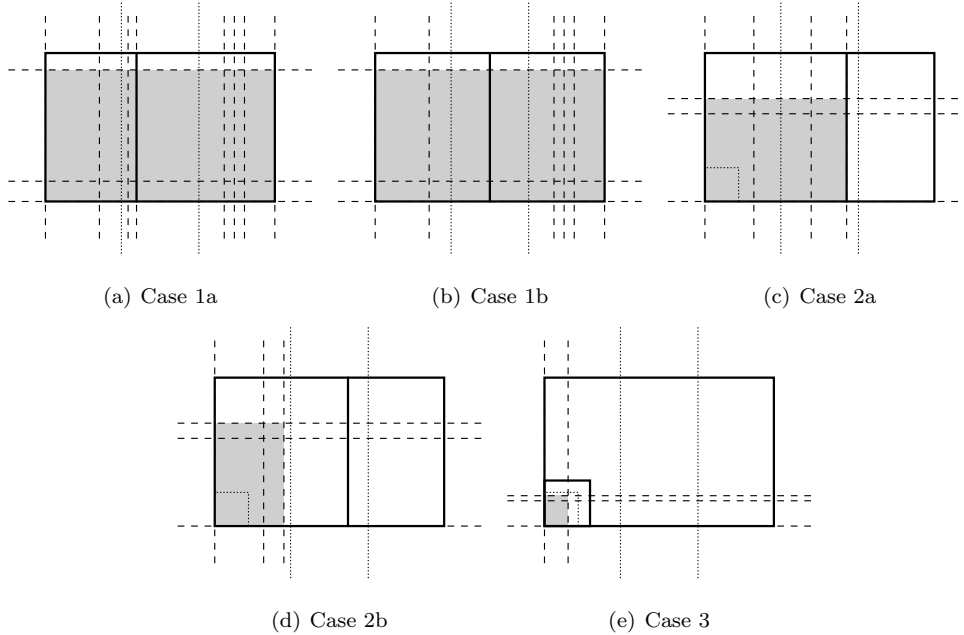


Figure 3: Cases for splitting the current hyperrectangle  $R$  (thick lines) into one or two smaller hyperrectangles. The gray solid hyperrectangle is  $R'$ . The slab boundaries are depicted as dashed lines.

Then, the invariants maintained by Govindarajan *et al.* can be stated as follows:

1. For each dimension, at least one side of  $R$  coincides with a slab boundary.
2. For all  $i = 1, \dots, d$ , either  $l_i(R') = l_i(R)$  or  $l_i(R') \leq \frac{2}{3}l_i(R)$ , where  $l_i(R)$  is the length of  $R$  in dimension  $i$ .
3.  $l_{\min}(R) \geq \frac{1}{3}l_{\max}(R)$ , where  $l_{\min}(R)$  and  $l_{\max}(R)$  are the lengths of the shortest and longest sides of  $R$ , respectively.

The key observation leading to an I/O-efficient algorithm is the following: Each hyperrectangle side obtained by a split can be described uniquely using a constant number of slab boundaries and a constant amount of additional information. As this is also true for the starting hyperrectangle  $R_0$  (each of its sides is contained in a slab boundary), any hyperrectangle that can be obtained from  $R_0$  through a sequence splits can be described using a constant number of slab boundaries and a constant amount of extra information. The key idea of their I/O-efficient algorithm is to construct the set of *all* hyperrectangles that can be described this way. Callahan [31] calls these hyperrectangles *constructible* and gives a bound for the total number of them (which is slightly improved by Govindarajan *et al.* [25]):

**Theorem 4.** ([31, 25]) *There are  $\mathcal{O}(|S|^{2d(1-\alpha)})$  constructible hyperrectangles.*

Hence, by setting the value for the constant  $\alpha \in [1 - \frac{1}{2d}, 1)$ , Govindarajan *et al.* get  $\mathcal{O}(|S|)$  constructible hyperrectangles. Note that, due to our modification of the value of  $\alpha$ , we can bound the number of all constructible hyperrectangles by  $\mathcal{O}(\sqrt{|S|})$ . Ultimately, this will allow us to replace both the use of buffer trees as well as the use of topology buffer trees by efficient cache-oblivious computation steps.

In each of the cases depicted in Figure 3, the thick hyperrectangle is the current hyperrectangle  $R$  which has to be split into one or two smaller hyperrectangles. Let  $i_{max}(R)$  denote a dimension such that  $l_{i_{max}(R)}(R) = l_{max}(R)$ . The gray solid hyperrectangle is  $R'$ . The slab boundaries are depicted as dashed lines. Further, the area between the two dotted vertical (longer) lines is the middle third of  $R$  in dimension  $i_{max}(R)$  and the small dotted hyperrectangle in the lower left corner has side length  $\frac{4}{27}l_{max}(R)$ . Govindarajan *et al.* consider three cases:

**Case 1:**  $l_{max}(R) = l_{i_{max}}(R')$ : If there exists a slab boundary in the middle third in dimension  $i_{max}(R)$ , then this slab boundary is used to split  $R$  into two new hyperrectangles (Case 1a). Otherwise, we split  $R$  into two equal halves in this dimension (Case 1b). Hence, this case “produces” two new hyperrectangles. If  $R$  does not satisfy Case 1, then  $l_{i_{max}}(R') < \frac{2}{3}l_{max}(R)$  holds due to the invariants specified by Govindarajan *et al.* and exactly one of the sides of  $R$  in dimension  $i_{max}$  lies in a slab boundary. Let  $H$  denote this slab boundary.

**Case 2:**  $l_{max}(R') \geq \frac{4}{27}l_{max}(R)$ : If  $l_{i_{max}(R)}(R') \geq \frac{1}{3}l_{max}(R)$  (Case 2a), we split  $R$  along the slab boundary containing the side of  $R'$  farthest away from  $H$ . Otherwise (Case 2b), we split  $R$  along a hyperplane having a distance of  $\frac{2}{3}(\frac{4}{3})^j l_{max}(R')$  from  $H$ , where  $j$  is the unique integer such that  $\frac{1}{2}l_{max}(R) < \frac{2}{3}(\frac{4}{3})^j l_{max}(R') \leq \frac{2}{3}l_{max}(R)$ . As  $j$  satisfies  $0 \leq j \leq -\lfloor \log_{\frac{4}{3}} \frac{4}{27} \rfloor$ , there are only  $\mathcal{O}(1)$  choices for  $j$ . For technical reasons, this case produces only one new hyperrectangle, namely the one that contains  $R'$ .

**Case 3:**  $l_{max}(R') < \frac{4}{27}l_{max}(R)$ : Due to the invariants specified by Govindarajan *et al.*,  $R'$  shares a unique corner with  $R$  and we construct a cube  $C$  that contains  $R'$ , shares the same corner with  $R'$  and  $R$  and has side length  $l(C) = \frac{3}{2}l_{max}(R')$ . Thus, this case produces only one new hyperrectangle as well.

Figure 4: Rules for splitting a hyperrectangle  $R$  fulfilling specific invariants into one or two smaller hyperrectangles fulfilling these invariants as well.

Aiming at an I/O-efficient construction of  $T_c$ , Govindarajan *et al.* construct a graph  $\Gamma$  with nodes corresponding to the set of constructible hyperrectangles and with an edge set defined as follows: There exists a directed edge  $(R_1, R_2)$  between two constructible hyperrectangles  $R_1$  and  $R_2$  if and only if  $R_1$  is intersected by at least one slab boundary in every dimension and  $R_2$  can be obtained from  $R_1$  by performing one of the three mentioned cases. Govindarajan *et al.* observe that the graph  $\Gamma$  is a directed acyclic graph and that each node has out-degree at most two, which guarantees that the total size of the graph is linear in the number of nodes, i.e., in the number of constructible hyperrectangles. Concerning the tree  $T_c$ , they observe that  $T_c$  consists of all nodes of  $\Gamma$  that are accessible from  $R_0$  along with their outgoing edges. The two main steps of their I/O-efficient construction of  $T_c$  are (a) building the graph  $\Gamma$  in a preprocessing step and (b)

extracting the tree  $T_c$  afterwards. We will now outline these I/O-efficient computation steps along with our modifications to obtain a cache-oblivious version of them.

**Step 1a: Constructing  $\Gamma$**  Govindarajan *et al.* compute the slab boundaries by iterating over all  $d$  dimensions. For each dimension they sort the points with respect to this dimension and subsequently scan the resulting sorted list to place a slab boundary between the  $(j \cdot |S|^\alpha)$ -th and the  $(j \cdot |S|^\alpha + 1)$ -th point for  $1 \leq j \leq \lfloor |S|^{1-\alpha} \rfloor$ . Moreover, they place two extreme slab boundaries that contain the two sides of  $R_0$  in this dimension. As the overall process is only based on sorting and scanning, it can be implemented in a cache-oblivious manner such that it uses  $\mathcal{O}(d \cdot \text{sort}(|S|)) = \mathcal{O}(\text{sort}(|S|))$  memory transfers.

The vertex set of  $\Gamma$ , i.e., the set of all constructible hyperrectangles, can be constructed based on the information given by the slab boundaries. Govindarajan *et al.* do this by performing  $\mathcal{O}(d)$  nested scans using  $\mathcal{O}(\text{scan}(|\Gamma|))$  I/Os each. After these scans, every vertex of  $\Gamma$  stores a complete representation of the  $d$ -dimensional hyperrectangle it represents. In the face of the construction of the edge set of  $\Gamma$ , Govindarajan *et al.* augment every vertex  $R$  with a description of the largest hyperrectangle  $R'$  which is contained in  $R$  and which is bounded by slab boundaries (see Figure 3). In addition, they augment every vertex  $R$  per dimension  $i$  with a description of the slab boundary, that comes closest to the “middle” of  $R$  in this dimension. For both tasks and for each dimension, they build a buffer tree [35] over the appropriate list of slab boundaries and answer standard search queries on the constructed tree. The overall process of augmenting the vertices takes  $\mathcal{O}(\text{sort}(|S|))$  I/Os. After this augmentation every node  $R$  of  $\Gamma$  stores a complete representation of the hyperrectangles  $R$  and  $R'$  and the slab boundary that comes closest to splitting  $R$  in half in dimension  $i_{\max}(R)$ . Based on the information stored in each vertex of  $\Gamma$ , Govindarajan *et al.* show how to construct the edge set of  $\Gamma$  by a single scan over the vertex set of  $\Gamma$ , resulting in an overall procedure for the construction of  $\Gamma$  taking  $\mathcal{O}(\text{sort}(|S|))$  I/Os.

The vertex set of  $\Gamma$  can be constructed in a cache-oblivious manner using  $\mathcal{O}(d)$  nested scans as well. Observe that, due to our choice of  $\alpha$ , and since  $d \geq 1$ , the resulting algorithm performs only  $\mathcal{O}(\text{scan}(|S|^{1-\alpha})) = \mathcal{O}(\text{scan}(\sqrt{|S|}))$  memory transfers during this step. Instead of using buffer trees, we apply a brute-force approach for augmenting the nodes of  $\Gamma$  with the appropriate information as follows. Let  $R = [x_1, x'_1] \times \dots \times [x_d, x'_d]$  be one of the vertices of  $\Gamma$ . By scanning the (sorted) list of slab boundaries in the  $i$ -th dimension, we can find the slab boundary whose projection  $s$  onto the  $i$ -th coordinate axis minimizes the distance to  $x_i$  and satisfies  $s - x_i \geq 0$ . Simultaneously, the slab boundary whose projection  $s'$  onto the  $i$ -th coordinate axis minimizes the distance to  $x'_i$  and satisfies  $x'_i - s' \geq 0$ , can be found. By processing all  $d$  dimensions, the vertex  $R$  can be augmented with a description of the resulting hyperrectangle  $R'$ . Due to our choice of  $\alpha$ , each vertex of  $\Gamma$  can be processed this way using an overall number of  $\mathcal{O}(d \cdot \text{scan}(|S|^{1-\alpha})) = \mathcal{O}(\text{scan}(\sqrt{|S|}))$  memory transfers. The augmentation of each vertex  $R$  with the information about the slab boundary that comes closest to the “middle” of  $R$  in the appropriate dimensions can be done analogously using  $\mathcal{O}(\text{scan}(\sqrt{|S|}))$  memory transfers per vertex. Thus, the overall process for all  $\mathcal{O}(\sqrt{|S|})$  vertices of  $\Gamma$  requires  $\mathcal{O}(\text{scan}(|S|))$

memory transfers.

To construct the edge set of  $\Gamma$ , Govindarajan *et al.* observe that for each hyperrectangle  $R$ , at most two outgoing edges have to be found and that the information computed for each hyperrectangle by the above augmentation procedure is sufficient to distinguish between all cases depicted in Figure 3. Hence, the edge set of  $\Gamma$  can be obtained by a single scan over the vertex set of  $\Gamma$  taking at most  $\mathcal{O}(\text{scan}(|\Gamma|)) = \mathcal{O}(\text{scan}(\sqrt{|S|}))$  memory transfers. We store the edge set of  $\Gamma$  in the same way as Govindarajan *et al.*, i.e., we represent the edges implicitly by storing all nodes of  $\Gamma$  as triples. If a node  $R$  has two children  $R_1$  and  $R_2$ , then the triple has the form  $(R, R_1, R_2)$ . Otherwise, if  $R$  has only one child  $R_1$ , the triple has the form  $(R, R_1, \text{null})$ , and, if  $R$  has no children, the triple has the form  $(R, \text{null}, \text{null})$ . This completes the construction of  $\Gamma$ .

**Step 1b: Extracting  $T_c$  from  $\Gamma$**  The I/O-efficient extraction of  $T_c$  from  $\Gamma$  can be implemented using the time-forward processing technique. To apply this technique, the graph  $\Gamma$  needs to be sorted topologically [35]. Govindarajan *et al.* do this by sorting the nodes of  $\Gamma$  by their sums of their side lengths in decreasing order. Subsequently, they process  $\Gamma$  “downwards” to extract  $T_c$  consisting of all nodes that are accessible from  $R_0$  along with their outgoing edges. The sorting step, applying the time-forward processing technique and a scan over the vertex set, can be performed in  $\mathcal{O}(\text{sort}(|\Gamma|))$  I/Os [35]. As all these techniques can be implemented efficiently in the cache-oblivious model [33, 39], the extraction of  $T_c$  can be done the same way by performing  $\mathcal{O}(\text{sort}(|\Gamma|))$  memory transfers. Hence, taking the value of  $\alpha$  into account, this step takes  $\mathcal{O}(\text{sort}(\sqrt{|S|}))$  memory transfers.

*Step 2: Constructing  $T''$ .* Given the compressed pseudo split tree  $T_c$  for  $S$ , Govindarajan *et al.* construct the pseudo split tree  $T''$  in three phases: In the first phase (see Step 2a), they attach the leaves to  $T_c$  which were discarded during the construction of  $T_c$ . Considering the obtained tree, they observe that every point in  $S$  is either contained in a leaf or in a region  $R \setminus C$ , where  $(R, C)$  is a so-called *compressed edge* (a compressed edge is an edge induced by Case 3). Hence, they distribute all points of  $S$  to the hyperrectangles and regions in the second phase (see Step 2b). Finally, in the third phase (see Step 2c), they replace every compressed edge by a sequence of splits which results in the desired tree  $T''$ . We will now give the details of the three steps along with their cache-oblivious implementations.

**Step 2a: Attaching missing leaves** A split with respect to Case 2 results in only one hyperrectangle  $R_1$  fulfilling the invariants specified by Govindarajan *et al.* and containing the hyperrectangle  $R'$ . The other hyperrectangle  $R_2 = R \setminus R_1$  is discarded during the construction of  $\Gamma$  as it violates the first invariant in Case 2b. However,  $R_2$  is contained in a single slab in at least one dimension and therefore contains at most  $|S|^\alpha$  points of  $S$ . Hence, it is possible to attach these discarded boxes directly to  $T_c$  without violating the constraint that no leaf contains more than  $|S|^\alpha$  points.

Govindarajan *et al.* observe that, as complete representations of the hyperrectangles  $R$  and  $R'$  are stored in each node of  $T_c$ , a single scan over the vertex set of  $T_c$  is sufficient to detect and to attach all discarded

boxes. During this process, every triple  $(R, R_1, null)$  representing such a split induced by Case 2 is modified to the triple  $(R, R_1, R_2)$  and for each such modification, a new vertex  $(R_2, null, null)$  is added to the vertex set of  $T_c$ . As at most one child is attached to each node of  $T_c$ , the resulting graph, called  $T_c^+$ , has size  $\mathcal{O}(|T_c|)$  as well. Thus, the overall process takes  $\mathcal{O}(\text{scan}(|T_c|))$  I/Os and can be implemented efficiently in the cache-oblivious model spending the same amount of memory transfers. Taking the new value for  $\alpha$  into account, this step takes  $\mathcal{O}(\text{scan}(\sqrt{|S|}))$  memory transfers.

**Step 2b: Distributing the points of  $S$**  Considering  $T_c^+$ , every point of  $S$  is either contained in a leaf of  $T_c^+$  or in a region  $R \setminus C$ , where  $(R, C)$  is a compressed edge induced by Case 3. To distribute the points of  $S$  to these boxes and regions, Govindarajan *et al.* answer so called *deepest containment queries* on  $T_c^+$  for all points in  $S$  using a topology buffer tree [36, 37]. The distribution of all points takes  $\mathcal{O}(\text{sort}(|S|))$  I/Os.

Instead of using a topology buffer tree, we apply the following brute-force approach for distributing the points which is inspired by Callahan’s parallel approach [31]: Subdivide the space into cells bounded by the hyperplanes which occur as boundaries for the constructible hyperrectangles. Using the same argument as for bounding the number of constructible hyperrectangles ensures that there are only  $\mathcal{O}(|S|^{2d(1-\alpha)}) = \mathcal{O}(\sqrt{|S|})$  cells of this type. Furthermore, a list of these cells can be constructed using  $\mathcal{O}(d)$  nested scanning and sorting steps using  $\mathcal{O}(\text{sort}(\sqrt{|S|}))$  memory transfers. By sorting the points and the list of hyperplanes with respect to the  $i$ th coordinate and by subsequently scanning both sorted lists, we can label each point with the pair of adjacent hyperplanes in this dimension, which determines the slab that contains it. Processing all points and all  $d$  dimensions this way takes  $\mathcal{O}(\text{sort}(|S|))$  memory transfers and labels each point with a description of the cell that contains it. After computing this correspondence between the points in  $S$  and the cells, we compute an appropriate correspondence between the cells and the hyperrectangles and regions. Note that, by construction, each cell is either contained in a hyperrectangle being a leaf of  $T_c^+$  or in a region  $R \setminus C$ , where  $(R, C)$  is a compressed edge of  $T_c^+$ . As the number of cells and the number of hyperrectangles and regions are bounded by  $\mathcal{O}(\sqrt{|S|})$ , this correspondence can be obtained by a brute-force approach performing  $\mathcal{O}(\text{scan}(|S|))$  memory transfers. To compute the desired correspondence between the points in  $S$  and the hyperrectangles and regions, we first sort both lists lexicographically according to the cell entries. By scanning both sorted lists, we can subsequently augment each point with a representation of the hyperrectangle rather or region which contains the point. Both steps can be performed in  $\mathcal{O}(\text{sort}(|S|))$  memory transfers.

**Step 2c: Expanding compressed edges** As already mentioned, the third case produces only one hyperrectangle when applied to a hyperrectangle  $R$  in  $T_c$ . More precisely, applying the third case to a hyperrectangle  $R$  results in a cube  $C$  with side length  $\frac{3}{2}l_{\max}(R')$  which shares a corner with  $R$  and  $R'$ . The edge  $(R, C)$  is a compressed edge and needs to be replaced with a sequence of splits. Govindarajan *et al.* replace all these compressed edges I/O-efficiently by simulating one phase of the internal memory

algorithm [29] for constructing a fair split tree for each compressed edge  $(R, C)$ . In the process, such an edge is replaced by a tree  $T(R, C)$  whose leaves form a partition of  $R$  into boxes and which consists of a path from  $R$  to  $C$  with an extra leaf attached to each node on the path except  $C$ .

Govindarajan *et al.* prove that the resulting tree  $T''$  has size  $\mathcal{O}(|S|)$ . Further, they provide an I/O-efficient implementation based on scanning and sorting which uses  $\mathcal{O}(\text{sort}(|S|))$  I/Os. Hence, the expansion of all edges can be performed cache-obliviously using  $\mathcal{O}(\text{sort}(|S|))$  memory transfers as well.

*Step 3: Construction of  $T'$ .* Given the pseudo split tree  $T''$  of  $S$ , Govindarajan *et al.* apply time-forward processing to remove every node  $R$  with  $R \cap S = \emptyset$  from  $T''$  and to compress all paths consisting of nodes having one child to a single edge in the resulting tree. The overall process is only based on scanning, sorting and time-forward processing and can therefore be implemented efficiently in the cache-oblivious model spending  $\mathcal{O}(\text{sort}(|S|))$  memory transfers. The obtained tree is the desired fair split tree of  $S$ . As the bound for the space consumption follows from the layout of the algorithm, this completes the proof of the following lemma and, hence, also Lemma 6:

**Lemma 7.** *Given a set  $S$  of points in  $\mathbb{R}^d$  and a constant  $\alpha \in [1 - \frac{1}{4d}, 1)$ , function PARTIALFAIRSPPLITTREE (Algorithm 4.2) computes a partial fair split tree  $T'$  of  $S$  spending  $\mathcal{O}(\text{sort}(|S|))$  memory transfers and  $\mathcal{O}(|S|/B)$  blocks of memory in the cache-oblivious model. Each leaf of  $T'$  represents a subset of  $S$  with at most  $|S|^\alpha$  points.*

#### 4.3. Construction of the Well-Separated Pairs

Once a fair split tree for the point set  $S$  is computed, Govindarajan *et al.* simulate the internal memory algorithm for constructing the pairs  $\{A_i, B_i\}$  of the designated WSPD in external memory by applying the time-forward processing technique performing an overall number of  $\mathcal{O}(\text{sort}(|S|))$  I/Os. Besides time-forward processing, the I/O-efficient algorithm is only based on scanning and sorting. Thus, the complete procedure can be implemented in a cache-oblivious manner in  $\mathcal{O}(\text{sort}(|S|))$  memory transfers as well.

As we followed the framework of Govindarajan *et al.* [25], the correctness of our algorithm for constructing a WSPD follows directly. Hence, we have shown the following theorem:

**Theorem 5.** *Given a set  $S$  of points in  $\mathbb{R}^d$  and a separation constant  $s > 0$ , a well-separated pair decomposition for  $S$  with separation ratio  $s$  having size  $\mathcal{O}(s^d|S|)$  can be computed spending  $\mathcal{O}(\text{sort}(|S|))$  memory transfers and  $\mathcal{O}(|S|/B)$  blocks of memory in the cache-oblivious model.*

## 5. Conclusions

Combining Theorem 5 with the results summarized in Lemma 5, we obtain our main result:

**Theorem 6.** *Given a geometric graph  $G = (S, E)$  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  $|E'| \in \mathcal{O}(s^d|S|)$  spending  $\mathcal{O}(\text{sort}(|E|))$  memory transfers in the cache-oblivious model.*

We conclude by noting that the cache-oblivious algorithm for constructing a WSPD is of independent interest. For example, constructing  $t$ -spanners with a linear number of edges can be performed easily in a cache-oblivious manner using the WSPD: Given a set  $S$  of points in  $\mathbb{R}^d$  and a real constant  $t > 1$ , construct a WSPD for  $S$  with separation ratio  $s = 4\frac{t+1}{t-1}$  having size  $\mathcal{O}(s^d|S|)$  and add, for each pair  $\{A_i, B_i\}$  of the WSPD, exactly one (arbitrary) pair  $\{x, y\}$  of points with  $x \in A_i$  and  $y \in B_i$  to an initially empty edge set  $E$ . It has been shown by Callahan and Kosaraju [40] that the resulting graph  $G = (S, E)$  is a  $t$ -spanner for  $S$  with a linear number of edges. Furthermore, Arya *et al.* [41] show that one can construct a  $t$ -spanner of spanner diameter at most  $2 \log |S|$  by first selecting a special representative  $r(A) \in A$  for each node  $A$  of the corresponding fair split tree and by subsequently adding the edge  $\{r(A_i), r(B_i)\}$  for every pair  $\{A_i, B_i\}$  of the WSPD to the initially empty edge set  $E$ . An I/O-efficient algorithm for choosing the representatives for each pair of the WSPD is given by Govindarajan *et al.* [25]. Their approach is only based on scanning, sorting, and time-forward processing and can therefore be implemented in an efficient cache-oblivious manner as well. Using Theorem 5, we can restate their result [25, Theorem 4] in the cache-oblivious model of computation.

**Corollary 2.** *Given a point set  $S \subset \mathbb{R}^d$  and some constant  $t > 1$ , we can compute a  $t$ -spanner  $G = (S, E)$  of linear size and diameter at most  $2 \log |S|$  spending  $\mathcal{O}(\text{sort}(|S|))$  memory transfers in the cache-oblivious model.*

*Acknowledgment.* We thank Norbert Zeh for helpful discussions regarding the I/O-efficient algorithm [25] for computing a well-separated pair decomposition.

## References

- [1] J. Gudmundsson, J. Vahrenhold, I/O-efficiently pruning dense spanners, in: J. Akiyama, M. Kano, X. Tan (Eds.), Revised Selected Papers of the Japanese Conference on Discrete and Computational Geometry (JCDCG 2004), Vol. 3742 of Lecture Notes in Computer Science, Springer, Berlin, 2005, pp. 106–116.
- [2] I. Althöfer, G. Das, D. P. Dobkin, D. Joseph, J. Soares, On sparse spanners of weighted graphs, *Discrete & Computational Geometry* 9 (1993) 81–100.
- [3] S. Arya, G. Das, D. M. Mount, J. S. Salowe, M. Smid, Euclidean spanners: short, thin, and lanky, in: Proceedings of the 27th ACM Symposium on Theory of Computing, 1995, pp. 489–498.
- [4] B. Chandra, G. Das, G. Narasimhan, J. Soares, New sparseness results on graph spanners, *International Journal of Computational Geometry and Applications* 5 (1/2) (1995) 124–144.
- [5] G. Das, G. Narasimhan, A fast algorithm for constructing sparse Euclidean spanners, *International Journal of Computational Geometry and Applications* 7 (4) (1997) 297–315.

- [6] C. Levcopoulos, G. Narasimhan, M. Smid, Improved algorithms for constructing fault-tolerant spanners, *Algorithmica* 32 (1) (2002) 144–156.
- [7] G. Navarro, R. Paredes, Practical construction of metric  $t$ -spanners, in: 5th Workshop on Algorithmic Engineering and Experiments, SIAM Press, 2003, pp. 69–81.
- [8] G. Navarro, R. Paredes, E. Chávez,  $t$ -spanners as a data structure for metric space searching, in: 9th International Symposium on String Processing and Information Retrieval, Vol. 2476 of Lecture Notes in Computer Science, Springer, Berlin, 2002, pp. 298–309.
- [9] K. M. Alzoubi, X.-Y. Li, Y. Wang, P.-J. Wan, O. Frieder, Geometric spanners for wireless ad hoc networks, *IEEE Transactions on Parallel and Distributed Systems* 14 (4) (2003) 408–421.
- [10] X.-Y. Li, Applications of computational geometry in wireless ad hoc networks, in: X.-Z. Cheng, X. Huang, D.-Z. Du (Eds.), *Ad Hoc wireless networking*, Kluwer, 2003, pp. 197–264.
- [11] G. Narasimhan, M. Smid, *Geometric Spanner Networks*, Cambridge University Press, 2007.
- [12] D. Eppstein, Spanning trees and spanners, in: J.-R. Sack, J. Urrutia (Eds.), *Handbook of Computational Geometry*, Elsevier Science Publishers, Amsterdam, 2000, pp. 425–461.
- [13] J. Gudmundsson, C. Knauer, Dilation and detours in geometric networks, in: T. Gonzalez (Ed.), *Handbook of Approximation Algorithms and Metaheuristics*, Chapman & Hall/CRC, 2007, pp. 52.1–52.16.
- [14] J. Gudmundsson, G. Narasimhan, M. Smid, Geometric spanners, in: M.-Y. Kao (Ed.), *Encyclopedia of Algorithms*, Springer, Berlin, 2008, pp. 360–364.
- [15] M. Smid, Closest point problems in computational geometry, in: J.-R. Sack, J. Urrutia (Eds.), *Handbook of Computational Geometry*, Elsevier Science Publishers, Amsterdam, 2000, pp. 877–935.
- [16] D. Z. Chen, G. Das, M. Smid, Lower bounds for computing geometric spanners and approximate shortest paths, *Discrete Applied Mathematics* 110 (2001) 151–167.
- [17] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran, Cache-oblivious algorithms, in: *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science*, IEEE Computer Press, 1999, pp. 285–299.
- [18] A. Aggarwal, J. S. Vitter, The input/output complexity of sorting and related problems, *Communications of the ACM* 31 (9) (1988) 1116–1127.
- [19] A. L. Buchsbaum, J. R. Westbrook, Maintaining hierarchical graph views, in: *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, Association for Computing Machinery, 2000, pp. 566–575.

- [20] S. Eubank, V. A. Kumar, M. V. Marathe, A. Srinivasany, N. Wang, Structural and algorithmic aspects of massive social networks, in: J. I. Munro (Ed.), Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Association for Computing Machinery, 2004, pp. 718–727.
- [21] L. Arge, External memory data structures, in: J. Abello, P. M. Pardalos, M. G. C. Resende (Eds.), Handbook of Massive Data Sets, Kluwer, 2002, pp. 313–357.
- [22] J. S. Vitter, Algorithms and data structures for external memory, Foundations and Trends in Theoretical Computer Science 2 (4) (2006) 305–474.
- [23] I. Katriel, U. Meyer, Elementary graph algorithms in external memory, in: U. Meyer, P. Sanders, J. Sibeyn (Eds.), Algorithms for Memory Hierarchies, Vol. 2625 of Lecture Notes in Computer Science, Springer, Berlin, 2003, pp. 62–84.
- [24] L. Toma, N. Zeh, I/O-efficient algorithms for sparse graphs, in: U. Meyer, P. Sanders, J. Sibeyn (Eds.), Algorithms for Memory Hierarchies, Vol. 2625 of Lecture Notes in Computer Science, Springer, Berlin, 2003, pp. 85–109.
- [25] S. Govindarajan, T. Lukovszki, A. Maheshwari, N. Zeh, I/O-efficient well-separated pair decomposition and its applications, Algorithmica 45 (4) (2006) 585–614.
- [26] T. Lukovszki, A. Maheshwari, 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, Vol. 2245 of Lecture Notes in Computer Science, Springer, Berlin, 2001, pp. 244–255.
- [27] A. Maheshwari, M. Smid, N. Zeh, I/O-efficient shortest path queries in geometric spanners, in: F. Dehne, J.-R. Sack, R. Tamassia (Eds.), Algorithms and Data Structures, 7th International Workshop, WADS 2001, Vol. 2125 of Lecture Notes in Computer Science, Springer, Berlin, 2001, pp. 287–299.
- [28] J. Gudmundsson, C. Levcopoulos, G. Narasimhan, Improved greedy algorithms for constructing sparse geometric spanners, SIAM Journal on Computing 31 (5) (2002) 1479–1500.
- [29] P. B. Callahan, 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 (1) (1995) 67–90.
- [30] J. Gudmundsson, C. Levcopoulos, G. Narasimhan, M. Smid, Approximate distance oracles for geometric graphs, in: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Association for Computing Machinery, 2002, pp. 828–837.
- [31] 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).

- [32] J. Gudmundsson, G. Narasimhan, M. Smid, Fast pruning of geometric graphs, in: Proceedings of the 22nd Annual Symposium on Theoretical Aspects of Computer Science, Vol. 3404 of Lecture Notes in Computer Science, Springer, Berlin, 2005, pp. 508–520.
- [33] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, J. I. Munro, An optimal cache-oblivious priority queue and its application to graph algorithms, *SIAM Journal on Computing* 36 (6) (2007) 1672–1695.
- [34] G. S. Brodal, R. Fagerberg, Cache oblivious distribution sweeping, in: P. Widmayer, F. T. Ruiz, R. M. Bueno, M. Hennessy, S. Eidenbenz, R. Conejo (Eds.), Proceedings of the 29th International Colloquium on Automata, Languages and Programming, Vol. 2380 of Lecture Notes In Computer Science, Springer, Berlin, 2002, pp. 426–438.
- [35] L. Arge, The buffer tree: A technique for designing batched external data structures, *Algorithmica* 37 (1) (2003) 1–24.
- [36] G. N. Frederickson, A data structure for dynamically maintaining rooted trees, *Journal of Algorithms* 24 (1) (1997) 37–65.
- [37] N. Zeh, I/O-efficient algorithms for shortest path related problems, Ph.D. thesis, School of Computer Science, Carleton University (2002).
- [38] F. Gieseke, Algorithmen zur Konstruktion und Ausdünnung von Spanner-Graphen im Cache-Oblivious-Modell, Tech. Rep. TR07-3-005, Universität Dortmund, Fachbereich Informatik, in German (Jul. 2007).
- [39] G. S. Brodal, R. Fagerberg, Funnel heap - a cache oblivious priority queue, in: P. Bose, P. Morin (Eds.), Proceedings of the 13th International Symposium on Algorithms and Computation, Vol. 2518 of Lecture Notes in Computer Science, Springer, Berlin, 2002, pp. 219–228.
- [40] P. B. Callahan, S. R. Kosaraju, Faster algorithms for some geometric graph problems in higher dimensions, in: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, Association for Computing Machinery, 1993, pp. 291–300.
- [41] S. Arya, D. M. Mount, M. Smid, Randomized and deterministic algorithms for geometric spanners of small diameter, in: Proceedings of the 35th IEEE Symposium on Foundations of Computer Science, 1994, pp. 703–712.