I/O-Efficient Well-Separated Pair Decomposition and its Applications*

(Extended Abstract)

Sathish Govindarajan[1], Tamás Lukovszki[2], Anil Maheshwari[3], and Norbert Zeh[3]

[1] Duke University, gsat@cs.duke.edu
[2] University of Paderborn, Heinz-Nixdorf-Institut, tamas@hni.upb.de
[3] Carleton University, {maheshwa,nzeh}@scs.carleton.ca

Abstract. We present an external memory algorithm to compute a well-separated pair decomposition (WSPD) of a given point set P in $\mathbb{R}^d$ in $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory, where N is the number of points in P, and $sort(N)$ denotes the I/O complexity of sorting N items. (Throughout this paper we assume that the dimension d is fixed). We also show how to dynamically maintain the WSPD in $O(\log_B N)$ I/O's per insert or delete operation using $O(N/B)$ blocks of external memory. As applications of the WSPD, we show how to compute a linear size t-spanner for P within the same I/O and space bounds and how to solve the K-nearest neighbor and K-closest pair problems in $O(sort(KN))$ and $O(sort(N+K))$ I/Os using $O(KN/B)$ and $O((N+K)/B)$ blocks of external memory, respectively. Using the dynamic WSPD, we show how to dynamically maintain the closest pair of P in $O(\log_B N)$ I/O's per insert or delete operation using $O(N/B)$ blocks of external memory.

1 Introduction

Many geometric applications require computations involving the set of all distinct pairs of points (and their distances) from a set P of N points in d-dimensional Euclidean space (e.g. nearest neighbor for each point). Voronoi diagrams and multi-dimensional divide and conquer are the traditional techniques used for solving several distance based geometric problems, especially in two and three dimensions. But in d dimensions, the worst case size of Voronoi diagrams can be $\Omega(N^{\lfloor d/2 \rfloor})$, and divide and conquer will require an exponent in the polylog factor depending on the dimension. Callahan and Kosaraju [6] introduced the WSPD data structure to cope with higher dimensional geometric problems. It consists of a binary tree T whose leaves are the points in P, with internal nodes representing the subsets of points within the subtree, and a list of "well-separated" pairs of subsets of P, each of which is a node in T. Intuitively a pair $\{A,B\}$ is well separated, if the distance between A and B is significantly greater than the distance between any two points within A or B. It turns out that for many problems it is sufficient to perform only a constant number of operations on pair $\{A,B\}$ instead of performing $|A||B|$ separate operations on the corresponding pairs of points. Moreover for fixed d, a WSPD of $O(N)$ pairs of subsets can be constructed. This has resulted in fast sequential, parallel,

* Research supported by NSERC, NCE GEOIDE, and by DFG-SFB376.

and dynamic algorithms for a number of problems on point-sets. Here we extend these results to external memory.

Previous Work In the Parallel Disk Model (PDM) [16], there is an external memory consisting of D disks attached to a machine with internal memory size M. Each of the D disks is divided into blocks of B consecutive data items. Up to D blocks, at most one per disk, can be transferred between internal and external memory in a single I/O-operation. The complexity of an algorithm in the PDM model is the number of I/O operations it performs. Relevant research work in the external memory (EM) setting can be found in the survey of Vitter [15]. In the PDM model it has been shown that sorting an array of size N takes $sort(N) = \Theta((N/DB)\log_{(M/B)}(N/B))$ I/Os [16, 15]. Scanning an array of size N takes $scan(N) = \Theta(N/DB)$ I/Os.

For the geometric problems discussed in this paper, the best resources will be [10] for WSPD and its applications and [14] for results on proximity problems. We omit the discussion on the state of the art, importance and significance of these problems here due to the lack of space, and refer the reader to these references.

New Results In this paper we present external memory algorithms for the following problems for a set P consisting of N points in d-dimensional Euclidean space: In Sections 3 and 4, we present an algorithm to compute the WSPD data structure for P; it requires $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory. In Section 5, we present an algorithm to dynamically maintain the WSPD in $O(\log_B N)$ I/O's per insert/delete operation using $O(N/B)$ blocks of external memory. In Section 6.1, we present an algorithm to compute a linear size spanner for P; it requires $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory. Choosing spanner edges carefully, we can guarantee a diameter of $2\log N$ for the constructed spanner. We also show that $\Omega(\min\{N, sort(N)\})$ I/Os are required to compute any t-spanner of a given point set, for any $t > 1$. In Section 6.2, we present an algorithm to compute K-nearest neighbors, i.e. for each point $a \in P$, compute the K-nearest points to a in $P - \{a\}$; this takes $O(sort(KN))$ I/Os using $O(KN/B)$ blocks of external memory. In Section 6.3, we present an algorithm to compute K-closest pairs, i.e. report the K smallest interpoint distances in P; this takes $O(sort(N+K))$ I/Os using $O((N+K)/B)$ blocks of external memory. In Section 6.4, we present an algorithm to dynamically maintain the closest pair of P in $O(\log_B N)$ I/O's per insert or delete operation using $O(N/B)$ blocks of external memory.

In [5] an $O(\log_B N)$ algorithm for the dynamic closest pair problem in higher dimensions is given. Their approach uses the Topology B-tree data structure. For the remaining problems, our results are the only efficient external memory algorithms known in higher dimensions. For the K-nearest neighbor and K-closest pair problems, optimal algorithms were presented in [12] for the case where $d = 2$ and $K = 1$. In [2] it is shown that computing the closest pair of a point set requires $\Omega(sort(N))$ I/Os, which implies the same lower bound for the general problems we consider in this paper. I/O-efficient construction of fault-tolerant spanners and bounded degree spanners for point sets in the plane and of spanners for polygonal obstacles in the plane has been discussed in [13].

2 Preliminaries

For a given point set P, the *bounding rectangle* $R(P)$ is the smallest rectangle containing all points in P, where a rectangle is the Cartesian product $[x_1,x_1'] \times [x_2,x_2'] \times \cdots \times [x_d,x_d']$ of a set of closed intervals. The *length* of R in dimension i is $l_i(R) = x_i' - x_i$. The maximum and minimum lengths of R are $l_{\max}(R) = \max\{l_i(R) : 1 \le i \le d\}$ and $l_{\min}(R) = \min\{l_i(R) : 1 \le i \le d\}$. When all lengths of R are equal, R is a cube; we denote it's side length by $l(R) = l_{\max}(R) = l_{\min}(R)$. Let $i_{\max}(R)$ be the dimension such that $l_{i_{\max}}(R) = l_{\max}(R)$. For a point set P, let $l_i(P) = l_i(R(P))$, $l_{\max}(P) = l_{\max}(R(P))$, $l_{\min}(P) = l_{\min}(R(P))$, and $i_{\max}(P) = i_{\max}(R(P))$. Let $d(x,y)$ denote the Euclidean distance between points x and y. For point sets X and Y, let $d(X,Y) = \min\{d(x,y) : x \in X \wedge y \in Y\}$. Given a *separation constant s*, we say that two point sets A and B are well-separated if $R(A)$ and $R(B)$ can be enclosed in two d-balls of radius r such that the distance between the two balls is at least sr. We define the *interaction product* $A \otimes B$ of two point sets A and B as $A \otimes B = \{\{a,b\} : a \in A \wedge b \in B \wedge a \neq b\}$. A *well-separated realization* of $A \otimes B$ is a set $\{\{A_1,B_1\},\ldots,\{A_k,B_k\}\}$ with the following properties [9]:

(R1) $A_i \subseteq A$ and $B_i \subseteq B$, for $1 \le i \le k$,
(R2) $A_i \cap B_i = \emptyset$, for $1 \le i \le k$,
(R3) $(A_i \otimes B_i) \cap (A_j \otimes B_j) = \emptyset$, for $1 \le i < j \le k$,
(R4) $A \otimes B = \bigcup_{i=1}^{k} A_i \otimes B_i$.
(R5) A_i and B_i are well-separated, for $1 \le i \le k$.

Intuitively, this means that for every pair $\{a,b\}$ of points $a \in A$ and $b \in B$, there is a unique pair $\{A_i,B_i\}$ such that $a \in A_i$ and $b \in B_i$ and that for any pair $\{A_i,B_i\}$ the distance between the points in one of the sets is small compared to the distance between the sets. We can use a binary tree T to define a partition of a point set P into subsets. In particular, there is a leaf in T for each point in P. An internal node represents the set of points associated with its descendant leaves. We refer to a node representing a subset $A \subseteq P$ as node A. A leaf representing point $a \in P$ is referred to as leaf a. The parent of a node A in T is denoted by $p(A)$. We say that a realization of $A \otimes B$ *uses* a tree T if all sets A_i and B_i in the realization are nodes in T. A *well-separated pair decomposition* of a point set P is a binary tree T associated with P and a well-separated realization of $P \otimes P$ that uses T. A *split* of a point set P is a partition of P into two non-empty point sets lying on either side of a hyperplane perpendicular to one of the coordinate axes, and not intersecting any points in P. A *split tree* T of P is a binary tree whose nodes are associated with subsets of P, defined as follows: If $P = \{x\}$, T contains a single node x. Otherwise, we use a split to partition P into two subsets P_1 and P_2; T consists of two split trees for point sets P_1 and P_2 whose roots are the children of the root node P of T. For a node A in T, we define the *outer rectangle* $\hat{R}(A)$ as follows: For the root P, let $\hat{R}(P)$ be an open d-cube centered at the center of $R(P)$ with $l_i(\hat{R}(P)) = l_{\max}(P)$. For all other nodes A, the hyperplane used for the split of $p(A)$ divides $\hat{R}(p(A))$ into two open rectangles. Let $\hat{R}(A)$ be the one that contains A. A *fair split* of A is a split of A where the hyperplane splitting A is at distance at least $l_{\max}(A)/3$ from each of the two boundaries of $\hat{R}(A)$ parallel to it. A split tree formed using only fair splits is called a *fair split tree*. A *partial fair split tree* is defined in the same way; but the leaves may represent point sets instead of single points.

3 Constructing a WSPD

In this section we assume that we are given a fair split tree $T = (V,E)$ for a point set P. Also, let every interior node A of T be labeled with its bounding rectangle $R(A)$. Every leaf is labeled with the point it represents. Given this input, we show how to find a well-separated pair decomposition of P I/O-efficiently. In the output, every pair $\{A_i,B_i\}$ in the WSPD will be represented by the IDs of the two nodes A_i and B_i in T.

Our approach is based on [9], which we sketch next. Given a postorder traversal of T, denote the postorder number of a node A in T by $\eta(A)$. Define an ordering $\prec$ on the nodes of T as $A \prec B$ if and only if either $l_{\max}(A) < l_{\max}(B)$ or $l_{\max}(A) = l_{\max}(B)$ and $\eta(A) < \eta(B)$. Given the sequence of nodes in T sorted by $\succ$, define $\nu(A)$ to be the position of node A in this sequence.

Denote the internal memory algorithm of [9] to construct a well-separated realization of $P \otimes P$ by COMPUTEWSR. It takes the tree T as an input. For every internal node X of T, let A and B be the two children of X. Then COMPUTEWSR calls a procedure FINDPAIRS with argument (A,B). Given a pair (A',B') as an input, procedure FINDPAIRS does the following: First it ensures that $B' \prec A'$, by swapping A' and B' if necessary. If A' and B' are well-separated, the pair $\{A',B'\}$ is added to the well-separated realization. Otherwise, let A'_1 and A'_2 be the children of A' in T. The procedure invokes itself recursively with pairs (A'_1,B') and (A'_2,B') as arguments.

The recursive invocation pattern of procedure FINDPAIRS can be described using the concept of *computation trees*. The root of such a tree T' is a pair $\{A,B\}$ such that A and B are the children of an internal node X of T. A node $\{A',B'\}$ in T' is a leaf if A' and B' are well-separated. Otherwise, let $B' \prec A'$ and A'_1 and A'_2 be the two children of A'; then node $\{A',B'\}$ has two children $\{A'_1,B'\}$ and $\{A'_2,B'\}$ in T'. Tree T' represents the recursive invocations of procedure FINDPAIRS made to compute a realization of $A \otimes B$. Every leaf of T' corresponds to a pair $\{A_i,B_i\}$ in the well-separated realization of $P \otimes P$. In [9] it is shown that there are only $O(N)$ pairs in this realization, which implies that the total size of all computation trees is $O(N)$.

We simulate COMPUTEWSR in external memory by applying *time-forward processing* [11] to the forest of computation trees. The difficulty is that this forest is not known in advance, and we have to generate it on the fly while processing it.

We use the split tree T to guide the processing of F. Assume that we are given the forest $F = (V_F,E_F)$ of computation trees and the fair split tree $T = (V,E)$. We sort the nodes of T by the $\succ$-relation and define a mapping $\mu : V_F \to V$ as $\mu(\{A,B\}) = A$, assuming w.l.o.g. that $B \prec A$. This mapping naturally defines an edge set $\mu(E_F)$ between the vertices in V as $\mu(E_F) = \{(\mu(v),\mu(w)) : (v,w) \in E_F\}$. Note that $\mu(E_F)$ may be a multiset. Given this mapping μ we process F by applying time-forward processing to the multigraph $\mu(F)$. In particular, we process the nodes of $\mu(F)$ sorted by $\succ$, which guarantees that we process the source vertex of every edge before the target vertex. Indeed, for any edge (A,A') in $\mu(E_F)$, either $\{A,A'\}$ is a vertex in F with $A' \prec A$ or A' is a child of A, which also implies that $A' \prec A$.

As F is *not* given beforehand, we have to show how to generate the edges of $\mu(F)$ while processing $\mu(F)$. For every node $A \in V$ with children A_1 and A_2, we store a label $\lambda(A)$ consisting of $\nu(A_1)$, $\nu(A_2)$, $R(A_1)$, and $R(A_2)$. Given an edge $(\{A,B\},\{A',B\}) \in E_F$, where A' is a child of A in T, the information $\phi(\mu(\{A,B\}))$ sent along edge

$(\mu(\{A,B\}),\mu(\{A',B\}))$ consists of $\nu(A')$, $\nu(B)$, $R(A')$, and $R(B)$. Now consider a node $A \in V$ receiving some tuple describing pair $\{A,B\}$ as input. We have to show how to generate the two children $\{A_1,B\}$ and $\{A_2,B\}$ of $\{A,B\}$ in F (if any) as well as the two edges $(A,\mu(\{A_1,B\}))$ and $(A,\mu(\{A_2,B\}))$ from this input and the label $\lambda(A)$. From the received information, decide whether A and B are well-separated. If they are, add $\{A,B\}$ to the well-separated realization, and proceed to the next input received by A. Otherwise, as the input has been sent to A, $B \prec A$. Hence, $\{A_1,B\}$ and $\{A_2,B\}$ are indeed the two children of $\{A,B\}$ in F. Generate the description of these two pairs from the received information about B and the information about A_1 and A_2 stored in $\lambda(A)$. Compare $\nu(A_1)$ and $\nu(A_2)$ with $\nu(B)$ to compute $\mu(\{A_1,B\})$ and $\mu(\{A_2,B\})$. Queue $\{A_1,B\}$ with priority $\nu(\mu(\{A_1,B\}))$ and $\{A_2,B\}$ with priority $\nu(\mu(\{A_2,B\}))$.

To bound the I/O-complexity of this procedure, observe that we queue a constant amount of information per edge in F. As there are $O(N)$ edges in F, applying known results [11, 1] about time-forward processing, we obtain the following lemma.

Lemma 1. *Given a set P of N points in $\mathbb{R}^d$, a fair split tree T of P with $O(N)$ nodes and a separation constant $s > 0$, a well-separated realization of P can be computed in $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory. Together with T this gives a well-separated pair decomposition of P.*

4 Constructing a Fair Split Tree

Our algorithm to construct a fair split tree T for a given point set P follows the framework of the optimal PRAM algorithm in [4].[1] The idea is to construct T recursively. First we construct a partial fair split tree T' whose leaves have size $O(N^\alpha)$ for some constant $1 - \frac{1}{6d} \leq \alpha < 1$. Then we recursively build split trees for the leaves, proceeding with an optimal internal memory algorithm for every leaf whose size is at most M. We will show how to implement one such recursive step in $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory, which leads to the following result.

Lemma 2. *Given a set P of N points in $\mathbb{R}^d$, a fair split tree for P can be computed in $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory.*

From Lemmas 1 and 2 we obtain the following result.

Theorem 1. *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 $O(sort(N))$ I/Os using $O(N/B)$ blocks of external memory.*

To construct a partial fair split tree T', we first construct a compressed version T_c as in [4]. First each dimension is partitioned into slabs containing N^α points each. Every rectangle R associated with a node in T_c satisfies the following three invariants: (1) In each dimension at least one side of R lies on a slab boundary; (2) If R' is the largest rectangle contained in R all of whose sides fall onto slab boundaries, then either $l_i(R') = l_i(R)$ or $l_i(R') \leq \frac{2}{3} l_i(R)$; (3) $l_{\min}(R) \geq \frac{1}{3} l_{\max}(R)$. Now if we want to split rectangle R associated with an internal node v into smaller rectangles, we split R along its longest

[1] We do not simulate the PRAM algorithm as this would lead to a much higher I/O complexity.

side according to the following three cases: (1) $l_{\max}(R) = l_{i_{\max}}(R')$: We find the slab boundary that comes closest to splitting R into equal-sized pieces. If this slab boundary is at distance at least $\frac{1}{3}l_{\max}(R)$ from either side of R, we split R along this slab boundary. Otherwise, we split R into two equal-sized pieces. This case produces the two resulting rectangles R_1 and R_2. (2) $l_{\max}(R') \geq \frac{8}{81}l_{\max}(R)$: If $l_{i_{\max}}(R') \geq \frac{1}{3}l_{\max}(R)$, we split R along the side of R' that is not shared with R. Otherwise, we find the unique integer j such that $y = \frac{2}{3}\left(\frac{4}{3}\right)^j l_{\max}(R')$ lies in the interval $\left(\frac{1}{2}, \frac{2}{3}\right] l_{\max}(R)$ and split along a hyperplane that is at distance y from the slab boundary shared by R' and R. This case produces the rectangle containing R'. The other rectangle is being ignored for the time being; we reattach it later. (3) $l_{\max}(R') < \frac{8}{81}l_{\max}(R)$: In this case, R and R' share a unique corner. We construct a d-cube sharing the same corner with R and with side length $\frac{3}{2}l_{\max}(R')$. This case produces the cube C. Later we have to construct a sequence of fair splits cutting R down to C.

The construction of T_c takes $O(sort(N))$ I/Os using standard external-memory graph techniques. The reattachment of the missing Case 2 leaves takes sorting and scanning. Next we describe how to expand compressed edges corresponding to a Case 3 split.

Such a Case 3 edge corresponds to a rectangle R that has been "shrunk" to a d-cube C with the following properties: (P1) C and R share exactly one corner, and C is contained in R. (P2) Any rectangle contained in R and not intersecting C contains at most $O(N^\alpha)$ points. (P3) $l(C) < \frac{8}{81}l_{\max}(R)$ and $l_{\min}(R) \geq \frac{1}{3}l_{\max}(R)$.

Properties P1 and P3 guarantee that there exists a sequence of fair splits that produces C from R. Property P2 guarantees that for every split in this sequence, the rectangle not containing C can be made a leaf of T'. We show how to construct these sequences of fair splits for all Case 3 edges in $O(sort(N))$ I/Os. We also have to assign the points in P to the containing leaves of T', as this information is needed for the next level of recursion in the fair split tree construction algorithm.

Every point of P is contained in some leaf rectangle or in some region $R \setminus C$, where (R,C) is a compressed edge produced by Case 3. We compute an assignment of the points in P to these regions similarly to the corresponding step in the optimal PRAM-algorithm in [4]. This takes a constant number of sorting and scanning steps.

To replace each edge corresponding to Case 3 by a sequence of fair splits, we simulate one phase of the sequential algorithm of [9]. We sort the points in $R \setminus C$ in each dimension, producing a sorted list L_i^R of points for each dimension i. Consider the current rectangle R which we want to split in dimension $i_{\max}$, producing rectangles R_1 and R_2, where R_1 contains the cube C. If $l_{\max}(R) > 3l(C)$, split R in half. Otherwise, choose a hyperplane containing a side of C for the split. We use list $L_{i_{\max}}^R$ to decide whether there are any points in R_2. If so, we perform a split producing two new nodes R_1 and R_2 of T'' and assigning the points in R_2 to the leaf R_2. Otherwise, shrink R to R_1, not producing any new nodes. Shrinking R does not cost any I/Os as the coordinates of R are maintained in internal memory. This approach ensures that a linear number of splits are performed in the number of points in $R \setminus C$. Thus, in total only $O(N)$ splits for all edges (R,C) to be expanded are performed. We have to show how to maintain the lists L_i^R representing the set of points in the current rectangle R.

Assume that we split R in dimension i and that this produces two new rectangles R_1 and R_2. R_1 contains the cube C and R_2 does not. We scan L_i^R from the tail until we find the first point that is in R_1. All points after this position in L_i^R are not in R_1 and are therefore removed and put into the point list of leaf R_2. As R_1 is going to act as rectangle R for the next split, we need to delete all points in R_2 from all lists L_j^R. The above scan takes care of deleting these points from L_i^R. However, we cannot afford to delete these points from all the other lists L_j^R, $j \neq i$, because this would cause many random accesses or would require a scan over all these lists per split. Instead we delete points lazily. When splitting the rectangle R_1 in dimension j, which produces rectangles R_3 and R_4, where R_3 contains C, we scan L_j^R from the tail until we find the first point in R_3. For each point visited by this scan we test whether it is in R_4. If it is, it is appended to the point set of leaf R_4. Otherwise, as it is not in R_3 and not in R_4, it must be contained in a leaf produced by a previous split and can therefore be discarded. In total, we scan every list at most once, for a total of $O(scan(N))$ I/Os.

Up to this point, the construction algorithm has in fact computed a "super-tree" T'' of T', as some leaves produced by the algorithm may be empty. Even though some leaves of T'' are empty, the size of T'' is $O(N)$ [4]. Thus, given the assignment of point sets to the leaves of T'' as computed above, it takes $O(sort(N))$ I/Os using standard external memory graph techniques to remove empty leaves and all internal nodes such that all descendant leaves of at least one of its two children are empty. The result is the desired partial fair split tree T'.

5 Dynamic Well-Separated Pair Decomposition

Dynamic Fair-Split Tree We present an I/O-efficient algorithm for dynamically maintaining the fair-split tree using $O(\log_B N)$ I/Os per insert or delete operation. We follow the approach of [8].

The main idea is to maintain a rectangle $\tilde{R}(A)$, such that $R(A) \subseteq \tilde{R}(A)$, for each node $A \in T$ that is not modified during updates. A *fair cut* of any rectangle $\bar{R}$ is defined as partitioning the rectangle using an axis-parallel hyperplane that is at distance at least $\frac{1}{3} l_{\max}(\bar{R})$ from its nearest parallel sides. Note that a fair cut of $\tilde{R}(A)$ satisfies the fair split condition on $R(A)$, since $l_{\max}(R(A)) \leq l_{\max}(\tilde{R}(A))$ and $R(A) \subseteq \tilde{R}(A)$ at any moment in time. Let $R \rightarrow R'$ indicate that R' can be constructed from R by a sequence of fair cuts.

We maintain a binary tree T under insertions and deletions in which each node satisfies the following invariants:

(I1) For all internal nodes A with children A_1 and A_2, there exists a *fair cut* that partitions $\tilde{R}(A)$ into rectangles R_1 and R_2 such that $R_1 \rightarrow \tilde{R}(A_1)$ and $R_2 \rightarrow \tilde{R}(A_2)$.
(I2) For all leaves a, $\tilde{R}(a) = a$.

It is easy to see that T is a fair-split tree. Invariant I1 ensures the existence of a fair cut, which satisfies the fair split property.

Insertion: When we want to insert a point a, we find the deepest node A in T such that $a \in \tilde{R}(A)$. Let R_1 and A_1 have the same meaning as in Invariant I1 and assume that $a \in R_1$. We insert a new node B, which replaces A_1 as a child of A. We make A_1 and a children of B. We show how to construct $\tilde{R}(B)$ satisfying Invariant I1.

Consider Invariant I1 for node A. Since $R_1 \to \tilde{R}(A_1)$, there exists a sequence of fair cuts that construct $\tilde{R}(A_1)$ from R_1. Clearly the last rectangle R in the sequence of cuts that satisfies $a \in R$ can be assigned as $\tilde{R}(B)$. However, this does not give us an efficient way to find $\tilde{R}(B)$, since the sequence of cuts may be long. In [8] it is shown that given the deepest node A, $\tilde{R}(B)$ can be computed in constant time.

We can compute A using $O(\log_B N)$ I/Os by posing it as a deepest-intersect query on the Topology B-tree [5].

Deletion: To delete a point a, delete the leaf a and compress the parent node $p(a)$. Note that this preserves the invariants.

Theorem 2. *The fair-split tree of a point set P can be maintained using $O(N/B)$ disk blocks and $O(\log_B N)$ I/Os per insert or delete operation.*

Maintaining the pairs dynamically We follow the approach of [8]. We use the following characterization of well-separated pairs. An ordered pair of nodes $(A,B), A, B \in T$ is well-separated if and only if it satisfies the following conditions:

1. The point sets A and B are well-separated,
2. Nodes $p(A)$ and B are not well-separated, and
3. $l_{\max}(R(B)) < l_{\max}(R(p(A))) \leq l_{\max}(R(p(B)))$.

It is easy to construct examples where the insertion of a new point can add a linear number of well-separated pairs. Thus, we cannot do a trivial update of the pairs upon insertion or deletion. The main idea of the approach is to anticipate all but a constant number of pairs that would be added when a new point is inserted. So, when a point is inserted or deleted, we need to update only a constant number of pairs. In [8] it is shown that this invariant can be maintained only if the point distribution of P is uniform. If it is not uniform, we add dummy points to make it uniform. In [8] it is shown that it is necessary to add only $O(1)$ dummy points per well-separated pair.

When a point p is inserted, we compute a c-approximation of the distance between p and its nearest neighbor in P, for some constant $c = f(s)$. In [5] it is shown how to compute the approximate nearest neighbor in $O(\log_B N)$ I/Os using a Topology B-tree.

Let d_p be the approximate distance computed above. It can be shown that the interactions between p and points $\{q \in P \mid d(q,p) > d_p\}$ are already covered in the current WSPD. We need to compute well-separated pairs that account for interactions between p and points within distance d_p from p. The number of such pairs needed is $O(1)$. We can compute these pairs as follows:

Extract all nodes $A \in T$ whose bounding rectangles $R(A)$ overlap the cube C centered at p with length $l(C) = d_p$ and satisfy the conditions of a WSPD defined earlier in this subsection. This can be done by filtering C through a Topology-B-tree that corresponds to the fair-split tree. Since the number of pairs extracted is $O(1)$, we can perform this extraction using $O(\log_B N)$ I/O's.

We need to add dummy points to make sure that the region covering the newly computed pairs (A,B) is uniform. Note that for the existing pairs, we have already added the required dummy points. A straightforward way is to add a mesh of suitably spaced points covering the region around $R(A)$ and $R(B)$. We insert the dummy points and compute the pairs corresponding to them as described above. The total number of dummy points added is $O(1)$ and the number of pairs added to the WSPD is $O(1)$. Thus, the entire process requires $O(\log_B N)$ I/Os. The deletion of a point p can be taken care of using a similar procedure.

Theorem 3. *Let P be a set of N points in $\mathbb{R}^d$. The well-separated pair decomposition of P with respect to a fair-split tree T can be maintained under insertions and deletions using $O(\log_B N)$ I/Os per update and $O(N/B)$ disk blocks.*

6 Applications of the WSPD

6.1 *t*-Spanners

Given a point set P, let $G = (P,E)$ be a graph whose edges have weights corresponding to the Euclidean distance between their two endpoints. G is called a t-spanner for P if for any two points p and q in P, the shortest path from p to q in G is at most t times longer than the Euclidean distance between p and q. In [9] it is shown that the following graph $G = (P,E)$ is a t-spanner of linear size for the point set P: For every node $A \in T$ choose a representative $q(A) \in A$. For every pair $\{A_i, B_i\}$ in the given WSPD of P, add an edge $\{q(A_i), q(B_i)\}$ to E. In [3] it is shown that one can compute a spanner of diameter at most $2\log N$ by choosing representatives carefully. In particular, we partition the edges in T into heavy and light edges. Given a node A with children A_1 and A_2, edge (A,A_1) is heavy and edge (A,A_2) is light if $|A_1| \geq |A_2|$. Otherwise, edge (A,A_2) is heavy and edge (A,A_1) is light. Every node of T is contained in a unique chain of heavy edges starting at a leaf. For node A, let $q(A)$ be the leaf whose chain contains A.

This assignment of representatives corresponds to sending representatives up the tree; at every node A receiving representative $q(A_1)$ and $q(A_2)$ from its children, we choose $q(A)$ as the one sent along the heavy edge. The decision which edge is heavy can be made by sending sizes $|A_1|$ and $|A_2|$ along with the representative $q(A_1)$ and $q(A_2)$. This can be done using time-forward processing.

Theorem 4. *Given a set P of N points in $\mathbb{R}^d$, it takes $O(sort(N))$ I/Os and $O(N/B)$ blocks of external memory to compute a t-spanner G of linear size and diameter at most $2\log N$ for P.*

The next theorem states that the I/O-complexity of the spanner construction is practically optimal.

Theorem 5. *It takes $\Omega(\min\{N, sort(N)\})$ I/Os to compute a t-spanner, $t > 1$, for a point set P of N points in $\mathbb{R}^d$.*

Proof. We prove the lemma for $d = 1$. The proof generalizes to higher dimensions by placing all points on a straight line in $\mathbb{R}^d$. Given a list X of items $x_1, \ldots, x_N$ and a permutation $\sigma : [1,N] \to [1,N]$, we are finally interested in computing the sequence $x_{\sigma(1)}, \ldots, x_{\sigma(N)}$. Let σ be given as a sequence $S = \langle \sigma(1), \ldots, \sigma(N) \rangle$. We map data item x_i to the point $i + \frac{1}{t}$ and $\sigma(i)$ to the point $\sigma(i)$. Any t-spanner, $t > 1$, of the resulting point set must contain edges $(\sigma(i), x_{\sigma(i)})$. We first remove all edges that are not of this type and then reverse the permutation performed on elements $\sigma(i)$ by the spanner construction algorithm to arrange the remaining edges in the order $(\sigma(1), x_{\sigma(1)}), \ldots, (\sigma(N), x_{\sigma(N)})$. This reversal can be done by recording the I/Os performed by the spanner construction algorithm and playing this I/O-sequence backward, exchanging the meaning of read and write operations.

The construction of the point set takes $O(scan(N))$ I/Os. The reversal of the permutation takes $O(T_S(N))$ I/Os, where $T_S(N)$ is the I/O-complexity of the spanner algorithm. In total, we can compute any permutation in $O(T_S(N))$ I/Os, so that $T_S(N) = \Omega(perm(N)) = \Omega(\min\{N, sort(N)\})$. □

6.2 *K*-Nearest Neighbors

In this section we show how to compute the K-nearest neighbors for every point $p \in P$. The construction follows the sequential algorithm in [9] for this problem.

Lemma 3 ([9]). *Let $\{A, B\}$ be a pair in a well-separated realization of $P \otimes P$ with separation $s > 2$. If there is a point $b \in B$ that is a K-nearest neighbor of a point $a \in A$, then $|A| \leq K$.*

Given a set B, let O_B be the center of $R(B)$. Then we divide the space around B into a constant number of cones with apex O_B such that for any two points a and a' in the same cone, $\angle aO_Ba' < \frac{s}{s+1}$.

Lemma 4 ([9]). *Let a point set X and a cone c with apex O_B be given, such that for any two points a and a' in c, $\angle aO_Ba' < \frac{s}{s+1}$. Let $X_c = (x_1, \ldots, x_l)$ be the set of points in X that lie in c, sorted by distance from O_B. For $i > K$, no point in B can be a K-nearest neighbor of x_i.*

Based on Lemma 3 the algorithm in [9] first extracts all pairs $\{A_i, B_i\}$ with $|A_i| \leq K$. This can be done in $O(sort(N) + scan(KN))$ I/Os using time-forward processing and a reverse preorder traversal of T. For a node B in T, let $\{A'_1, B\}, \ldots, \{A'_q, B\}$ be the set of pairs such that $|A'_i| \leq K$. Note that the sets A'_i are pairwise disjoint. We store the set $f(B) = \bigcup_{i=1}^{q} A'_i$ as a candidate set at node B in T. The construction of sets $f(B)$ takes sorting and scanning and thus $O(sort(KN))$ I/Os. Then the set of candidates is narrowed down by a top-down procedure in T, which can be realized using time-forward processing. At the root node B of T, the algorithm partitions the space around O_B into cones as described above and uses K-selection in each cone to find the K points in $f(B)$ that fall into this cone and are closest to O_B. By Lemma 4, all other points in $f(B)$ cannot have a K-nearest neighbor in B. Call the resulting set of points $N(B)$. As $N(B)$ contains at most K points per cone and there are $O(1)$ cones, $|N(B)| = O(K)$. The set $N(B)$ is passed on to the children B_1 and B_2 of B. The points possibly having a closest neighbor in B_i are in the set $f(B_i) \cup N(B)$. Thus, the algorithm applies the same K-selection algorithm to this set to construct $N(B_i)$ and pass it on to B_i's children. As we send only $O(K)$ points along every edge of T, this takes $O(sort(KN))$ I/Os.

Finally every leaf b stores a set $N(b)$ of size $O(K)$ such that b is a potential K-nearest neighbor for the points in $N(b)$. The total size of all these sets is $O(KN)$. Now we build sets $N'(a)$ for all points $a \in P$ such that $b \in N'(a)$ if and only if $a \in N(b)$ and apply

K-selection to each set $N'(a)$ separately to find the K-nearest neighbors of point a. This takes sorting of KN points and a standard K-selection algorithm, hence, $O(sort(KN))$ I/Os.

Theorem 6. *Given a set P of N points in $\mathbb{R}^d$, it takes $O(sort(KN))$ I/Os and $O(KN/B)$ blocks of external memory to find the K-nearest neighbors for all points in P.*

6.3 *K*-Closest Pairs

Given a well-separated realization of $P \otimes P$, let the pairs $\{A_i, B_i\}$ be sorted by increasing distances $d(R(A_i), R(B_i))$. For any such pair, $|A_i \times B_i| = |A_i||B_i|$. To find the K closest pairs, we first determine the smallest i such that $\sum_{j=1}^{i} |A_j||B_j| \geq K$ and retrieve all pairs $\{A, B\}$ such that $d(R(A), R(B)) \leq (1 + 4/s)r$, where $r = d(R(A_i), R(B_i))$. We extract the set of pairs $\{a, b\}$ such that $a \in A$ and $b \in B$ for some pair $\{A, B\}$ that we retrieved.

In [7] it is shown that the K closest pairs are among the extracted point pairs and that the total number of extracted point pairs is $O(N + K)$. Given this set of point pairs, we apply K-selection again to find the K closest ones. Given all point pairs, this takes $O(scan(N + K))$ I/Os. We have to show how to extract all candidate pairs $\{a, b\}$.

Sorting the pairs $\{A_i, B_i\}$ by their distances $d(R(A_i), R(B_i))$ between the bounding rectangles takes $O(sort(N))$ I/Os. Then it takes a single scan to extract all pairs with $d(R(A), R(B)) \leq (1 + r/s)r$. We extract the points in sets A_i and B_i in the same way as in the previous section and sort these at most $O(N + K)$ points to guarantee that the points in the same pair $\{A_i, B_i\}$ are stored consecutively. We can now construct the set of all candidate point pairs by scanning the resulting point list. We obtain the following result.

Theorem 7. *Given a set P of N points in $\mathbb{R}^d$, it takes $O(sort(N + K))$ I/Os and $O((N + K)/B)$ blocks of external memory to compute the K closest pairs in P.*

6.4 Dynamic Closest Pair

We show how to maintain the closest pair of points in P under insertion and deletion. We make the following simple observation:

Lemma 5. *Let P be a point set of n points in $\mathbb{R}^d$. Let (a, b), $a, b \in P$ be the closest pair in P. Then, the pair $\{\{a\}, \{b\}\}$ belongs to the well-separated pair decomposition of P.*

Proof. Let (A, B) be the pair in the well-separated decomposition such that $a \in A, b \in B$ and b is a nearest neighbor of a. Applying Lemma 3, we have $|A| = 1$. Similarly, a is a nearest neighbor of b. By Lemma 3, we have $|B| = 1$. □

Denote the well-separated pairs of the form $\{\{a\}, \{b\}\}$ as singleton pairs. We maintain the singleton pairs in a B-tree based on the distance between them. The closest pair is the singleton pair with minimum distance.

During an insert or delete operation, update the well-separated pairs and insert or delete the singleton pairs in the B-tree, respectively. Since only $O(1)$ well-separated pairs change during an insert or delete operation, we obtain the following theorem.

Theorem 8. *Let P be a point set of n points in $\mathbb{R}^d$. The closest pair in P can be maintained using $O(\log_B N)$ I/Os per insert or delete operation.*

Acknowledgments

We would like to thank Pankaj Agarwal, Pat Morin, and Michiel Smid for helpful discussions.

References

1. Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345, 1995.
2. Lars Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. AMS, 1999.
3. S. Arya, D. M. Mount, and M. Smid. Randomized and deterministic algorithms for geometric spanners of small diameter. In *35th IEEE Symposium on Foundations of Computer Science (FOCS'94)*, pages 703–712, 1994.
4. P. B. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proc. FOCS'93*, pages 332–341, 1993.
5. P. B. Callahan, M. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. WADS'95*, LNCS 955, pages 381–392, 1995.
6. P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point sets with applications to *k*-nearest-neighbors and *n*-body potential fields. In *Proc. STOC'92*, pages 546–556, 1992.
7. P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proc. SODA'93*, pages 291–300, 1993.
8. P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and *n*-body potential fields. In *Proc. SODA'95*, pages 263–272, 1995.
9. 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.
10. Paul B. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1995.
11. Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1995.
12. Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory computational geometry. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, November 1993.
13. T. Lukovszki, A. Maheshwari, and N. Zeh. I/O-efficient spanner construction in the plane. `http://www.scs.carleton.ca/~nzeh/Pub/spanners00.ps.gz`, 2000.
14. Michiel Smid. Closest-point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 877–936. North-Holland, 2000.
15. J. S. Vitter. External memory algorithms. In *Proceedings of the 17th Annual ACM Symposium on Principles of Database Systems*, June 1998.
16. J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.