Kinetic and Dynamic Data Structures for Closest Pairs and All Nearest Neighbors[*]

Pankaj K. Agarwal[†] Haim Kaplan[‡] Micha Sharir[§]

July 13, 2007

Abstract

We present simple fully dynamic and kinetic data structures, which are variants of a dynamic 2-dimensional range tree, for maintaining the closest pair and all nearest neighbors for a set of n moving points in the plane; insertions and deletions of points are also allowed. If no insertions or deletions take place, the structure for closest pair uses $O(n \log n)$ space, and processes $O(n^2 \beta_{s+2}(n) \log n)$ critical events, each in $O(\log^2 n)$ time. Here s is the maximum number of times where the distances between any two specific pairs of points can become equal, $\beta_s(q) = \lambda_s(q)/q$, and $\lambda_s(q)$ is the maximum length of Davenport-Schinzel sequences of order s on q symbols. The dynamic version of the problem incurs a slight degradation in performance: If $m \geq n$ insertions and deletions are performed, the structure still uses $O(n \log n)$ space, and processes $O(mn\beta_{s+2}(n) \log^3 n)$ events, each in $O(\log^3 n)$ time.

Our kinetic data structure for all nearest neighbors uses $O(n \log^2 n)$ space, and processes $O(n^2 \beta^2_{s+2}(n) \log^3 n)$ critical events. The expected time to process all events is $O(n^2 \beta^2_{s+2}(n) \log^4 n)$, though processing a single event may take $\Theta(n)$ expected time in the worst case. If $m \geq n$ insertions and deletions are performed, then the expected number of events is $O(mn\beta^2_{s+2}(n) \log^3 n)$ and processing them all takes $O(mn\beta^2_{s+2}(n) \log^4 n)$. An insertion or deletion takes $O(n)$ expected time.

1 Introduction

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of n points, each moving independently in $\mathbb{R}^2$. Let $p_i(t) = (x_i(t), y_i(t))$ denote the position of p_i at time t, and set $P(t) = \{p_1(t), \ldots, p_n(t)\}$. We assume

[*]Work by Pankaj Agarwal has been supported by NSF under grants CCR- 00-86013, EIA-01-31905, CCR-02-04118, and DEB-04-25465, by ARO grant W911NF-04-1-0278 and by a grant from the U.S.–Israel Binational Science Foundation. Work by Haim Kaplan was partially supported by the Israeli Science Foundation (ISF) grant no. 975/06. Work by Micha Sharir was partially supported by NSF Grant CCR-00-98246, by a grant from the U.S.-Israeli Binational Science Foundation, by a grant from the Israel Science Fund, Israeli Academy of Sciences, for a Center of Excellence in Geometric Computing at Tel Aviv University, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

[†]Dept. Computer Science, Duke University, Durham, NC 27708-0129, USA. pankaj@cs.duke.edu

[‡]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel. E-mail: haimk@post.tau.ac.il

[§]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel, and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA. E-mail: michas@post.tau.ac.il

that each $x_i(\cdot), y_i(\cdot)$ is a semi-algebraic function of constant description complexity. The goal is to design a data structure that keeps track of the closest pair of points in P, and that can also support insertions and deletions of points into/from P, as well as changes in the flight plans of the moving points.

The *kinetic data structure* (KDS) framework, introduced by Basch et al. [10], proposes an algorithmic approach, together with several quality criteria, for maintaining a variety of geometric configurations determined by a set of objects, each moving along a semi-algebraic trajectory of constant description complexity (see below for a precise definition). Several interesting algorithms have been designed, using this framework, over the past few years, including algorithms for maintaining the convex hull of a set S of (moving) points in the plane [10], the closest pair in such a set [10], a point in the center region of such a set [3], kinetic planar subdivisions [2, 4, 7], kinetic medians and kd-trees [5], kinetic range searching [1], maintaining the extent of a moving point set [6], kinetic collision detection [9, 19, 22], shooting a moving target [12], kinetic discrete centers [16], kinetic connectivity for unit disks, rectangles, and hypercubes [18, 20], kinetic geometric spanners [21], and kinetic separation of convex polygons [23]; see [17] for a recent survey.

Typically, a geometric algorithm for computing such a configuration determined by a set S is designed for the *stationary* case, where the objects do not move. When the objects do move, the combinatorial representation of the configuration may change at certain *critical times*, when certain "events" occur (e.g., a new vertex of the convex hull may appear, an old vertex may disappear, the closest pair of points changes, etc.). The goal is to design a data structure that can efficiently keep track of these changes, and maintain (a discrete representation of) the correct configuration at all times. Thus the algorithm has to keep track of these critical events, and fix the configuration when they happen.

The crux in designing an efficient KDS is finding a set of *certificates* that, on one hand, ensure the correctness of the configuration currently being maintained, and, on the other hand, are inexpensive to maintain. When the motion starts, we can compute the closest failure time of any of the certificates, and insert these times into a global event queue. When the time of the next event in the queue matches the current time, we invoke the KDS repair mechanism, which fixes the configuration, and replaces the failing certificate(s) by new valid ones. In doing so, the mechanism will typically delete from the queue failure times that are no longer relevant, and insert new failure times into it.

To analyze the efficiency of a KDS, we distinguish between two types of events: *internal* and *external*. *External events* are events associated with a real (combinatorial) change in the configuration that we maintain, thus forcing a change in the output. *Internal events*, on the other hand, are events where some certificate fails, but the overall desired configuration still remains valid. These events arise because of our specific choice of the certificates, and are essentially an overhead incurred by the data structure. If the ratio between the number of internal events to the number of external events (in the worst case) is no more than polylogarithmic in the number of input objects, the KDS is said to be *efficient*.[1] Other parameters of the KDS that one would like to minimize are

[1](a) In the original setup of Basch et al. [10], a KDS is considered to be efficient, if the ratio between the worst-case number of internal events to the worst-case number of external events is bounded by an arbitrarily small power of the number of input objects. In our definition of efficient KDS, we only allow a degradation factor that is a polylogarithmic function of the number of input objects. We impose similar stringer restrictions on the other performance parameters of

the following.

- The processing time of a critical event by the repair mechanism. If this parameter is no more than polylogarithmic in the number of input objects, we say that the KDS is *responsive*.
- The maximum number of events at any fixed time in the data structure that are associated with one particular object. When this parameter is no more than polylogarithmic in the number of input objects, we say that the KDS is *local*. Locality typically implies that changes in flight plans of the moving points can be handled efficiently.
- The space used by the data structure. If this is larger than the number of input objects by at most a polylogarithmic factor, we say that the KDS is *compact*.

In addition, which is one of the central issues addressed in this paper (for the specific closest pair and all nearest neighbors problems), one might wish to design a KDS that is also *dynamic*, meaning that it can also efficiently support insertions and deletions of objects.

In their paper, Basch et al. [10] developed a KDS that maintains the closest pair in a set of n moving points in the plane, which meets all four standard criteria, namely, it is compact, efficient, local, and responsive. Specifically, their structure uses linear space; it processes $O(n^2\beta_{s+2}(n)\log n)$ events, each in $O(\log^2 n)$ time, where s is the number of times any two fixed pairs of points can attain equal distances, and $\beta_{s+2}(n)$ is as defined in the abstract. To achieve locality, their algorithm uses a fairly complicated set of certificates, to guarantee that each point participates in $O(\log n)$ certificates. Furthermore, Basch et al. have focused only on kinetization, and did not consider insertions and deletions of points, which seems hard to implement using their approach. The motivation for our work has been threefold: (i) to simplify the certificates used by [10] for the closest-pair problem, (ii) to obtain a dynamic algorithm that still meets the four quality criteria mentioned above, and that can also be extended to higher dimensions, and (iii) to extend the technique to the all-nearest-neighbors problem, a problem that has not yet been tackled in the KDS literature.

Further background. In the static and stationary scenario, the complete set of points is given when the algorithm starts, and the points do not move. The planar version of the static and stationary closest-pair problem has been solved in $O(n \log n)$ time by Shamos and Hoey [28] in 1975. A year later Bentley and Shamos [11] gave an $O(n \log n)$ algorithm for the d-dimensional case. Vaidya [31] describes an $O(n \log n)$ algorithm for computing the nearest neighbor of every point in a given set of n points in $\mathbb{R}^d$. All these algorithms can be implemented in the algebraic decision tree model, for which an $\Omega(n \log n)$ lower bound holds; see [13].

The problem of maintaining the closest pair in the dynamic but stationary scenario has also been studied extensively during the past fifteen years. The first algorithm supporting both insertions and deletions in polylogarithmic update time was given by Smid [30], who presents a data structure for points in d dimensions, that takes $O(n \log^d n)$ space and supports updates in $O(\log^d n \log\log n)$ amortized time per update. Later, Schwarz et al. [26] have given an algorithm that supports only

the structure. (b) Ideally, one would like to ensure a small ratio between the number of internal events and the number of external events that *actually take place*. This is considerably harder to achieve, and is not addressed in the earlier works too.

insertions, in $O(\log n)$ amortized time per insertion, again in any dimension. Finally, Bespamyatnikh [14] presented an algorithm that supports both insertions and deletions in $O(\log n)$ worst-case time per update, in any dimension.

Our results. Our first result is an efficient *dynamic* KDS for maintaining the closest pair in a set of moving points in the plane, which also supports insertions and deletions of points, and which can be extended to any dimension $d > 2$. The structure is constructed using standard off-the-shelf data structure components, our certificates are simpler than those of [10], and the performance of our algorithm (in the planar case) is comparable with that of [10].

We assume that each moving point p is given as a pair $(x_p(t), y_p(t))$ of semi-algebraic functions of time of *constant description complexity*. That is, each function is defined as a Boolean combination of a constant number of predicates involving polynomials of constant maximum degree.

Our solution is based on a simple geometric property of the nearest neighbor to a given point, stated in Lemma 2.1 below, a property which has also been noted in [10]. Using this property, we design a data structure, which in essence is a 2-dimensional range tree. It stores the points of P in a certain transformed coordinate system, along with certain additional information to facilitate the maintenance of the closest pair.

We present the algorithm in three stages. First, in Section 2, we describe the data structure for the *static* (no insertions and deletions) and *stationary* (no motion of the points) scenario. This leads to a simple alternative $O(n \log n)$ algorithm for computing the closest pair in a planar point set. We next show, in Section 4, how to make this structure kinetic, still disallowing insertions and deletions. The modified structure uses $O(n \log n)$ space, and processes $O(n^2 \beta_{s+2}(n) \log n)$ critical events, each in $O(\log^2 n)$ time. Here s is the maximum number of times where the distances between any two specific pairs of points can become equal, $\beta_s(q) = \lambda_s(q)/q$, and $\lambda_s(q)$ is the maximum length of Davenport-Schinzel sequences of order s on q symbols. Then, in Section 5, we modify the structure further, and turn it into a fully dynamic and kinetic structure. The dynamic version of the problem incurs a slight degradation in performance: If $m \geq n$ insertions and deletions are performed, the structure still uses $O(n \log n)$ space, and processes $O(mn\beta_{s+2}(n) \log^3 n)$ events, each in $O(\log^3 n)$ time. See Theorem 5.2 for a precise statement of the performance bounds.

An appealing feature of our solution, besides being the first fully dynamic and kinetic solution for this problem, is its simplicity. It is in fact a dynamic 2-dimensional range tree, on which we superimpose a heap-like tournament structure, in which only a small number of pairs compete for being the closest pair in P. This number is linear in the static-kinetic case, and is $O(n \log n)$ in the dynamic-kinetic case.

Our second result is a KDS for maintaining the nearest neighbor of every point in a set of moving points in plane, with the same assumption on the motions as before. We obtain this result using the same basic observation of Lemma 2.1. The skeleton of the data structure itself is the same two-dimensional range tree which we use for the closest pair, but with an additional level of kinetic tournaments. In order to attain a sharp upper bound on the number of events that may occur in the additional tournaments, and on the time to handle them, we use treaps [27] to implement both the primary and the secondary trees. We present this result in Section 6. It uses $O(n \log^2 n)$ expected space, and the expected number of events processed by the data structure is $O(mn\beta_{s+2}^2(n) \log^3 n)$, if $m \geq n$ insertions and deletions are performed. The expected time to process all the events is $O(mn\beta_{s+2}^2(n)\log^4 n)$, though a single event may take $O(n)$ expected time. See Theorems 6.9 and 7.2 for precise statements.

Another feature of our solutions is that they can be extended to arbitrary dimensions $d > 2$, using essentially the same machinery (a d-dimensional dynamic range tree combined with kinetic and dynamic tournament data structures). We present this extension in Section 8; the precise performance bounds are given in Theorems 8.2 and 8.3.

2 Closest Pair and Nearest Neighbors: Static and Stationary

Let P be a set of n "fixed" points in $\mathbb{R}^2$. We present a "warm-up" solution for the closet-pair and the all-nearest-neighbor problems. Although these problems are well studied and optimal algorithms exist [13], we derive specific solutions that are easy to extend to kinetic and dynamic scenarios. These structures rely on the following simple but crucial lemma, also proved in [10].

Partition the plane into six wedges $W_0, \ldots, W_5$ of angle $\pi/3$ each, with the origin as their common apex, where W_i spans the orientation $[(2i-1)\pi/6, (2i+1)\pi/6]$. Let b_i denote the unit vector in the direction of the bisector ray of W_i. Note that $W_{i+3} = -W_i$ and $b_{i+3} = -b_i$ (where addition of indices is modulo 6). For a point p and a wedge W_i, let $p + W_i$ denote the translation of W_i so that its apex is at p.

Lemma 2.1. *Let p be the point closest to q, and let W_i be the wedge such that $p + W_i$ contains q. Then*
$$(q-p)\cdot b_i = \min\bigl\{(w-p)\cdot b_i \mid w \in P \cap (p+W_i)\bigr\}.$$

Proof: Suppose to the contrary that there exists a point $w \in P \cap (p + W_i)$, such that $(w-p)\cdot b_i < (q-p)\cdot b_i$. See Figure 1. We have $|qw| \geq |pq|$, so $\angle pwq$ must be smaller than $\angle qpw$. However, $\angle qpw \leq \pi/3$, and $\angle pwq > \pi/3$, a contradiction. $\square$

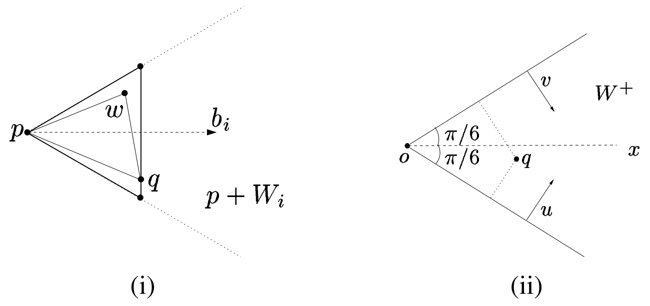

Figure 1. (i) p cannot be the point nearest to q. (ii) The u and v coordinates.

We restate Lemma 2.1 by saying that, if p is closest to q then q is the closest point to p (in $P \cap (p + W_i)$) in the b_i-direction. Symmetrically, if q is also the closest point to p then p is the closest point to q (in $P \cap (q + W_{i+3}) = P \cap (q - W_i)$) in the opposite $(-b_i)$-direction. We refer to such pairs of points as being *matched* in the b_i-direction.

Closest pair. Clearly, if (p, q) is a closest pair in P, then there exists a direction b_i such that p and q are *matched* in direction b_i. The algorithm keeps track of all these matched pairs of points, in each of the three directions b_0, b_1, and b_2, and selects the closest pair (in the Euclidean metric) among them. Note that, for each of the directions b_0, b_1, and b_2, a point p can participate in at most two matched pairs, once as the left point of the pair, and once as the right point of the pair. Thus, at any time, there are only $O(n)$ matched pairs.

Without loss of generality, we only consider matched pairs in direction b_0, i.e., the x-direction. Consider the two $\pi/3$ wedges $W^+ = W_0$, $W^- = W_3$ with the origin as an apex, whose bisector rays are, respectively, the positive and the negative portions of the x-axis. For simplicity of presentation, we regard W^+ and W^- as *open* wedges. For each point $q \in \mathbb{R}^2$, we set $W^+(q) := q + W^+$, $W^-(q) := q + W^-$. We thus wish to find all matched pairs of points (p, q) in the x-direction; that is, pairs (p, q) such that p lies to the left of q, q is the leftmost point of $P \cap W^+(p)$, and p is the rightmost point of $P \cap W^-(q)$. Let Π denote the set of these matched pairs.

To construct Π, we first map each point $p = (x_p, y_p) \in P$ to a point (u_p, v_p) in a new parametric plane, where $u_p = x_p + \sqrt{3}y_p$, and $v_p = x_p - \sqrt{3}y_p$. These coordinates are measured along axes that are orthogonal to the directions of the rays bounding the wedge W^+. See Figure 1 (ii).

Note that $q \in W^+(p)$ if and only if

$$u_q > u_p \quad \text{and} \quad v_q > v_p. \tag{1}$$

So all points q that may be matched with p are in the range given by (1), which is a translate of the positive quadrant in the uv-plane. To compute Π, we seek, for each point p, the point $q \in P$ that lies in that range and has the smallest x-coordinate. Since $x = (u + v)/2$ in the uv-plane, we want to find the point in the query quadrant which is extreme in the $(-1, -1)$-direction.

Motivated by this observation, we construct a 2-dimensional range tree T [13] on the transformed points of P, where the points are sorted in the primary tree by their u-coordinates, and in each secondary tree by their v-coordinates. We store a point in each leaf. Internal nodes do not contain points, but store keys to guide the search, and additional information that is described below. See Figure 2. We slightly abuse the notation, and use T to denote both the whole range tree and its primary tree. For a node w in the primary tree, we denote the set of points in the subtree rooted at w by $P(w)$, and denote the secondary tree associated with w by T_w. For a node $\xi \in T_w$, we denote the set of points in the subtree of ξ by $P(w, \xi)$. For an internal node ζ, either in the primary tree T or in some secondary tree, we denote by $\ell(\zeta)$ the left child of ζ, and by $r(\zeta)$ the right child of ζ.

Let w be a node in the primary tree T. The secondary tree T_w stores all points in $P(w)$ sorted by their v-coordinates. Within T_w, we refer to points that belong to $P(\ell(w))$ as *blue points*, and to points that belong to $P(r(w))$ as *red points*. By definition, the u-coordinate of each blue point is smaller than that of all the red points. However, when sorted by their v-coordinates, the blue and red points get mixed together. See Figure 2. We use the following observation, whose proof is straightforward.

Lemma 2.2. *Let w be a node in the primary tree T, and let ξ be a node in the secondary tree T_w. For each blue point $p \in P(w, \ell(\xi))$ and for each red point $q \in P(w, r(\xi))$, the wedge $W^+(p)$ contains q (and the wedge $W^-(q)$ contains p).*

For each node ξ in T_w, let $\mathsf{Red}(w, \xi)$ (resp., $\mathsf{Blue}(w, \xi)$) be the subset of red (resp., blue) points

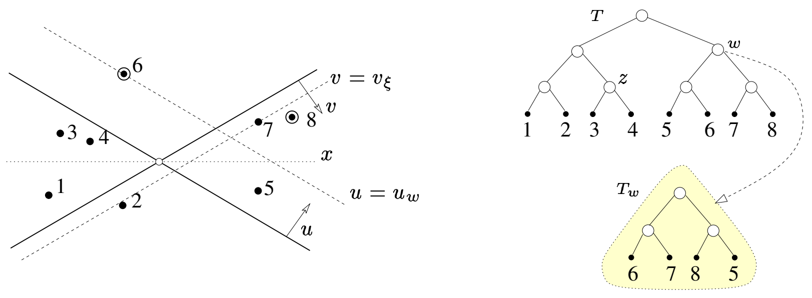

Figure 2. A set $P = \{1, \dots, 8\}$ of eight points in $\mathbb{R}^2$, the primary tree T storing the points sorted by their u-coordinates, and the secondary tree T_w of node w storing $P(w) = \{5, 6, 7, 8\}$ sorted by their v-coordinates. In T_w, 5, 6 are blue and 7, 8 are red points, and $\mathsf{blue}(w, \mathit{root}(T_w)) = 6$ and $\mathsf{red}(w, \mathit{root}(T_w)) = 8$. The closest pair (which is also matched in the x-direction) is $(3, 4) = \pi(z, \mathit{root}(T_z))$.

in $P(w, r(\xi))$ (resp., $P(w, \ell(\xi))$). Define $\mathsf{blue}(w, \xi)$ to be the point of maximum (resp., minimum) x-coordinate in $\mathsf{Blue}(w, \xi)$ (resp., $\mathsf{Red}(w, \xi)$); it is undefined if the set is empty. Consider the four quadrants defined by the lines $u = u_w$ and $v = v_\xi$, where u_w is the maximum u-coordinate of a point in $P(\ell(w))$, and v_ξ is the maximum v-coordinate of a point in $P(w, \ell(\xi))$. Then $\mathsf{Red}(w, \xi)$ (resp., $\mathsf{Blue}(w, \xi)$) lies in the first (resp., third) quadrant, and $\mathsf{red}(w, \xi)$ (resp., $\mathsf{blue}(w, \xi)$ is the point of $P(w, \xi)$ with the minimum (resp., maximum) x-coordinate (i.e., $(u + v)$-value) in this quadrant. See Figure 2. Let $\pi(w, \xi) = (\mathsf{blue}(w, \xi), \mathsf{red}(w, \xi))$, whenever both elements of the pair are defined. Set

$$\Pi^* = \{\pi(w, \xi) \mid \xi \in T_w, w \in T\}. \tag{2}$$

We have the following lemma.

Lemma 2.3. *Let p be the point closest to q so that $q \in W^+(p)$. Then there are nodes w and ξ such that $q = \mathsf{red}(w, \xi)$ and $p \in P(w, \ell(\xi))$. Symmetrically, if $q \in W^-(p)$, then there are nodes w and ξ such that $q = \mathsf{blue}(w, \xi)$ and $p \in P(w, r(\xi))$.*

Proof. We prove only the first part in which $q \in W^+(p)$; the proof of the second part is symmetric. Let w be the lowest common ancestor of p and q in T. Since $q \in W^+(p)$, we have $u_q > u_p$, so $p \in P(\ell(w))$ and $q \in P(r(w))$. It follows that in T_w, p is a blue point and q is a red point. Let ξ be the lowest common ancestor of p and q in T_w. Again, since $q \in W^+(p)$, we have $v_q > v_p$. Therefore, $p \in P(w, \ell(\xi))$ and $q \in P(w, r(\xi))$.

By Lemma 2.2, the wedge $W^+(p)$ contains all the red points in $P(w, r(\xi))$. It follows that q must be $\mathsf{red}(w, \xi)$, for otherwise there would exist a point q' in $W^+(p)$ with x-coordinate smaller than that of q, contradicting Lemma 2.1. $\square$

The following corollary follows from Lemmas 2.1 and 2.3.

Corollary 2.4. *For each matched pair $(p, q) \in \Pi$ there are nodes w and ξ such that $(p, q) = \pi(w, \xi)$. That is, $\Pi \subseteq \Pi^*$.*

Corollary 2.4 suggests the following procedure for computing the closest pair. We compute Π^* by constructing the range tree T and the pairs $\pi(w, \xi)$, for each primary node w and secondary node

$\xi \in T_w$. We then find the pair in Π^* with the minimum distance between its points. We apply a similar procedure to each of the two other bisector directions b_1, and b_2. The closest among the three resulting pairs is the closest pair in P. This completes the description of the static and stationary data structure for the closest pair problem. It uses $O(n \log n)$ storage, and the cost of constructing it, including the construction of the set Π^*, is $O(n \log n)$. The storage can be reduced to $O(n)$, if we construct T and the secondary trees incrementally, discarding portions that have already been fully processed.

All nearest neighbors. We now show how to use the range tree of the preceding analysis to find the nearest neighbor of each point $p \in P$. The strategy is to compute, for each $p \in P$, a candidate point $q \in W^+(p) \cup W^-(p)$, such that if the nearest neighbor of p lies in this union then it is equal to q (in certain cases, q may be undefined). By repeating this algorithm for the other two pairs of wedges, we assign to each point p at most three candidates q, one of which must be the nearest neighbor of p. We build the two-dimensional range tree as in the preceding algorithm. For each primary node w and secondary node $\xi \in T_w$, we compute the following:

(i) the points $\mathsf{red}(w, \xi)$ and $\mathsf{blue}(w, \xi)$;

(ii) $\beta(w, \xi)$, the point in $\mathsf{Blue}(w, \xi)$ closest to $\mathsf{red}(w, \xi)$;

(iii) $\varrho(w, \xi)$, the point in $\mathsf{Red}(w, \xi)$ closest to $\mathsf{blue}(w, \xi)$.

For each point p, define

$$\mathcal{C}(p) = \{\beta(w, \xi) \mid p = \mathsf{red}(w, \xi)\} \cup \{\varrho(w, \xi) \mid p = \mathsf{blue}(w, \xi)\}. \tag{3}$$

If $\mathcal{C}(p) \neq \emptyset$, then the candidate point q that we pick for p is the closest point to p among all points in $\mathcal{C}(p)$. The correctness of the algorithm follows from Lemma 2.1.

The running time of this algorithm is proportional to the sum of the sizes of all secondary subtrees, which is $O(n \log^2 n)$. The same technique can be used in higher dimensions too—see Section 8 for details. While yielding a suboptimal solution (Vaidya's algorithm runs in $O(n \log n)$ time in any dimension d), this technique is relatively easy to extend to the kinetic and dynamic scenarios, as will be described in the subsequent sections.

3 A Dynamic and Kinetic Tournament

In this section we review one of the main tools in our algorithm, adapted from Basch et al. [10], who present the following algorithm for maintaining the lowest point among a set of points moving along the y-axis.[2] Let $\mathcal{Q}$ be a completely balanced binary tree, with the points stored at its leaves (in an arbitrary order). For an internal node $v \in \mathcal{Q}$, let $P(v)$ denote the set of points in the subtree rooted at v. At any specific time t, each internal node v stores the lowest point among the points in

[2]We assume that when points collide, they simply go over each other and continue uninterruptedly with their individual trajectories.

$P(v)$ at time t. We call this lowest point the *winner* at v. Clearly, at any given time, the winner at v is the lower among the winner at the left child of v and the winner at the right child of v.

We associate a certificate with each internal node v, which asserts which of the two winners, at the left child and at the right child of v, is the winner at v. This certificate remains valid as long as (i) the winners at the children of v do not change, and (ii) the order along the y-axis between these two "sub-winners" does not change. The actual certificate caters only to the second condition; the first will be taken care of recursively. The failure time associated with this certificate is the next time when these two winners switch their order along the y-axis. We store all certificates in a heap, using the failure times as keys.[3] We call this heap of certificates the *event queue*.

As time progresses, the algorithm encounters *events*, in each of which some certificate fails. The algorithm keeps removing from the event queue the certificate with the minimum failure time, and replaces it with a new certificate, which takes into account the new order of the sub-winners at the corresponding node v, and whose failure time is the next time when the two points switch again their order. In addition, the new winner at v is propagated upwards to the ancestors of v, which may cause the algorithm to replace the certificates at some of these nodes too.

In more detail, the algorithm proceeds as follows. When a certificate associated with a node v fails at time t, a new winner q takes over the old winner p at v. The old winner p may have also been the winner at some ancestors of v, so, for each such ancestor, we change its winner to be q. For each ancestor node u of v whose winner has changed, we also change the certificate associated with its parent $p(u)$, where the new certificate confronts the new winner at u with the winner at the sibling of u. We remove the failure times of the old certificates from the event queue, and replace them by the failure times of the new certificates. All this takes $O(\log^2 n)$ time, and is dominated by the cost of performing $O(\log n)$ updates of the event queue. This implies that our data structure is *responsive* (in the terminology of [10]). It is also *compact* and *local*, as follows easily from the construction.

To bound the total number of events, we focus on a single node v, and bound the number of times when the winner at v can change. Clearly, the total number of winner changes at all nodes bounds the total number of events. (Note that creating a new certificate and inserting it into the event queue is not considered an event—the certificate may get deleted from the queue before it becomes the smallest element there. Only certificates that are removed by a **deletemin** operation are considered to be events.)

A bound on the total number of events times $O(\log^2 n)$ bounds the total time to process all events. However we can obtain a tighter bound by recalling that the time it takes to process an event at v is bounded by $O(\log n)$ times the number of ancestors of v that change their winner as a result of the event. Therefore, the total number of *winner changes* at all nodes times $O(\log n)$ bounds the total time for processing the events. (Again, as just remarked, there are potentially more winner changes than events.)

A winner change at v corresponds to a breakpoint in the *lower envelope* of the arrangement in the ty-plane, defined by the trajectories of the points in $P(v)$. If each such pair of trajectories intersect at most s times, then the complexity of the lower envelope that corresponds to v is at most $\lambda_s(|P(v)|) = |P(v)|\beta_s(|P(v)|)$, where $\lambda_s(n)$ is the maximum length of a Davenport-

[3]Any "regular" heap that supports **insert**, **delete**, and **deletemin** in $O(\log n)$ time is good for our purpose.

Schinzel sequence of order s on n symbols, and $\beta_s(n) = \lambda_s(n)/n$ is an extremely slowly growing function of n (see [29]). Summing these complexity bounds over all nodes v, we obtain that the overall number of winner changes, and therefore also the overall number of events, is at most $\sum_v |P(v)|\beta_s(|P(v)|) = O(n\beta_s(n)\log n)$. This is larger by a logarithmic factor than the maximum number of times the lowest point along the y-axis can indeed change, since this latter number is bounded by the complexity of the lower envelope associated with the root of $\mathcal{Q}$. It now follows from our discussion in the previous paragraph that the total time to process all the events is $O(n\beta_s(n)\log^2 n)$. In the terminology of [10], our data structure is thus also *efficient*.

Making the tournament dynamic. We next turn this static structure into a dynamic one, which also supports insertions and deletions of points. In principle, we can replace the static tree $\mathcal{Q}$ with any kind of dynamic balanced search tree data structure. However, for the analysis of the number of events to go through, we assume that $\mathcal{Q}$ is a *weight-balanced* $(BB(\alpha))$ *tree* [25] (see also [24]). This allows us to insert a new point anywhere we wish in $\mathcal{Q}$, and to delete any point from $\mathcal{Q}$, in $O(\log n)$ time. Each such insertion or deletion may change $O(\log n)$ certificates, along the corresponding search path, and therefore takes $O(\log^2 n)$ time, including the time for the structural updates of (rotations in) $\mathcal{Q}$; here n denotes the actual number of points in $\mathcal{Q}$, at the step where we perform the insertion or deletion. Again, most of the cost is incurred in accessing the event queue; updating the tournament structure itself takes only $O(\log n)$ time.[4]

We next bound the total number of events that may occur while inserting and deleting at most m points, at arbitrary locations, into a kinetic tournament $\mathcal{Q}$ that contains at most n points at any time. Each node in v is created during an insertion, and then exists in $\mathcal{Q}$ until the corresponding deletion. We refer to the period at which v exists in $\mathcal{Q}$ as the *lifetime* of v. We denote by $P(v)$ the *multiset* containing any point that is associated with a leaf of the subtree rooted at v during the lifetime of v. The multiplicity of a point p in $P(v)$ is the number of maximal connected time intervals at which p is stored at the subtree rooted at v. (Such transitions into and out of the subtree may happen when we perform rotations to rebalance the tree; see below.)

An argument analogous to the one given above for the static case, implies that the number of events at a node v (i.e., events where the certificate associated with v fails), is bounded by the number of winner changes at v. The number of winner changes in v is in turn bounded by $|P(v)|$ multiplied by $\beta_{s+2}(n)$.[5] Note that here we use β_{s+2} instead of β_s, since the lower envelope that we consider at each node v is now a lower envelope of *partial* functions [29]. Indeed, insertions and deletions of points from within the subtree of v, as well as rotations that may introduce new subtrees or remove subtrees from the subtree of v, may make the trajectory of a point appear in the arrangement in the ty-plane associated with v only during part of the lifetime of v.

Also, as in the static case, the time it takes to handle all the events is proportional to the number of winner changes at all nodes v multiplied by a factor of $O(\log n)$. So $O\left(\left(\sum_v |P(v)|\right)\beta_{s+2}(n)\log n\right)$ is an upper bound on the total time it takes to process all the events.

[4]Note that there is freedom in choosing the location where the new point is inserted, which we do not know how to exploit.

[5]We may use $\beta_{s+2}(n)$ rather than $\beta_{s+2}(m)$, by a standard argument that analyzes the total complexity of the envelope in the ty-plane by splitting it into $O(m/n)$ intervals along the t-axis, such that over each interval there are only $O(n)$ functions involved; see [29].

When inserting or deleting a node from a weight-balanced tree, we rebalance the tree by doing rotations at certain edges along the access path. See Figure 3.

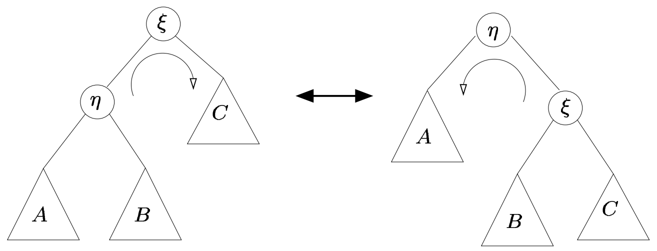

Figure 3. A rotation around the edge (ξ, η).

Suppose we perform $m \geq n$ insertions and deletions. If we ignore rotations for a moment, then each insertion increases $|P(v)|$ by one, only for nodes v on a single path, so this contributes $O(m \log n)$ to $\sum_v |P(v)|$. A rotation around an edge (ξ, η), where $\xi = p(\eta)$, changes $P(\xi)$ and $P(\eta)$ substantially. In particular, the set $P(\eta)$ grows substantially by inheriting the subtree rooted at the former sibling of η (and child of ξ). To complete the analysis, we have to bound the total growth in the sets $P(z)$ due to rotations. Here comes to the rescue a well known property of weight-balanced trees [15] (see also [24]), which asserts that, even if the cost of a rotation around (ξ, η) is proportional to $|P(\xi)|$, the total cost of any sequence of $O(m)$ operations is only $O(m \log n)$. This immediately implies that the sum of the size increases of the sets $P(z)$, due to rotations, is only $O(m \log n)$. The following theorem summarizes what we have just shown.

Theorem 3.1. *A sequence of m insertions and deletions into a kinetic tournament, whose maximum size at any time is n (assuming $m \geq n$), when implemented as a weight-balanced tree in the manner described above, generates at most $O(m\beta_{s+2}(n) \log n)$ events, for a total cost of $O(m\beta_{s+2}(n) \log^2 n)$. Each update takes $O(\log^2 n)$ worst-case time. One can construct (i.e., initialize) a kinetic tournament on n elements, at any fixed time, in $O(n)$ time.*

Remark. Note that the amortized analysis of rotations in a weight-balanced tree is used only for guaranteeing a near-linear bound on the total number of events. In contrast, the time bound for an update is *worst-case*, because, when we do a rotation in the weight-balanced tree, no rebuilding of secondary structures is needed, so the rotation takes $O(1)$ time.

4 KDS for Closest Pair

As the points of P vary continuously with time, we maintain the closest pair in P by keeping track of the combinatorial changes in the structure of T, and in the set Π of matched pairs. Without loss of generality, we limit the discussion to changes in the first range tree data structure, that uses the wedges W^+, W^- with the x-axis as a bisector. Here is an overview of our approach. We note that, as long as the u-order, v-order, and x-order of the points of P all remain unchanged, the structure of T also remains unchanged. Moreover, the set Π of matched pairs, and the larger set Π^* also do not change. We refer to swaps in which the u-order, v-order, or x-order of some pair of points of P changes, as *u-swaps*, *v-swaps*, or *x-swaps*, respectively.

We maintain the set Π explicitly in a dynamic kinetic tournament $\mathcal{Q}$ as described in Section 3. Specifically, for each pair (p, q) we define an item $\varphi_{p,q}$, which is the Euclidean distance between p and q, i.e.,

$$\varphi_{p,q}(t) = \|p(t) - q(t)\|. \tag{4}$$

Let $\Phi = \{\varphi_{p,q}(t) \mid (p, q) \in \Pi\}$. We construct the tournament $\mathcal{Q}$ on Φ. It follows that the winner at the root of $\mathcal{Q}$ corresponds to the closest pair. In between swaps, the set Φ is static, and changes of the closest pair are tracked by the kinetic tournament $\mathcal{Q}$ and its events.

To keep track of the swaps, we maintain the range tree T, and three auxiliary sorted lists L_u, L_v, L_x, where L_u (resp., L_v, L_x) stores the points of P sorted by their u-coordinates (resp., v-coordinates, x-coordinates). We implement L_u by threading the leaves of T, and we can implement L_v by threading the leaves of the secondary tree at the root of T. The list L_x is maintained separately. We maintain a collection of certificates, one for each pair of consecutive elements in each of the three lists; each certificate simply asserts that the corresponding pair obeys the respective order. We refer to these certificates as certificates of type (i), type (ii), and type (iii), respectively. The failure events of these certificates are then:

(i) Two consecutive points $p, q \in P$ in the u-order switch their positions (a u-swap). See Figure 4 (i).

(ii) Two consecutive points $p, q \in P$ in the v-order switch their positions (a v-swap).

(iii) Two consecutive points $p, q \in P$ in the x-order switch their positions (an x-swap). See Figure 4 (ii).

We add these certificates to the event queue in which we maintain the certificates of $\mathcal{Q}$; the key of each such certificate is its failure time.

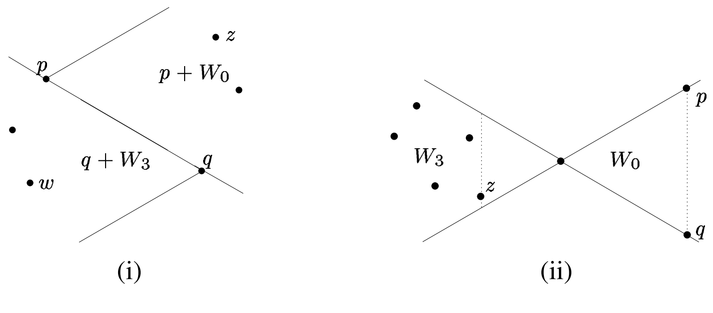

Figure 4. (i) A u-swap between p and q, with p moving down and q up; the old matched pair (p, z) is replaced by the new pair (p, q). (ii) An x-swap between p and q, with q moving to the left and p to the right; the old matched pair (z, p) is replaced by (z, q).

When a u-swap, v-swap, or x-swap between points p and q occurs, we update the tree T and the appropriate one among the lists L_u, L_v, and L_x. The swap may also change the set Π. Fortunately, any pair that starts or stops being matched due to the swap contains p or q, and the number of such pairs is $O(1)$. We use T to efficiently identify the changes in Π, and then perform the appropriate constantly many insertions and deletions into $\mathcal{Q}$. By the assumption on the motion of the points, the total number of swaps is $O(n^2)$. Hence, by Theorem 3.1, $\mathcal{Q}$ encounters $O(n^2\beta_{s+2}(n)\log n)$ events, which can be processed in overall time $O(n^2\beta_{s+2}(n)\log^2 n)$. Note that a swap may also change the set Π^*; the number of changes in this set that are caused by a swap might not be constant, though.

The following lemma summarizes our observations thus far.

Lemma 4.1. *The set Π of matched pairs changes only when two points p, q swap their positions in either the u-order, v-order, or x-order. In each of these events, any pair that starts or stops being matched contains p or q, and the number of such pairs is $O(1)$.*

We maintain at each node ξ of each secondary tree T_w, the points of maximal and minimal x-coordinates stored at $P(w, \xi)$. [6] This allows us to use T to answer queries of the form: For a given point p, find the leftmost point in $W^+(p)$, or find the rightmost point in $W^-(p)$. The first type of query specifies the quadrant $u > u_p$, $v > v_p$, and asks for the point of smallest x-coordinate (or $(u+v)$-value) in this quadrant. The second type of query specifies the quadrant $u < u_p$, $v < v_p$, and asks for the point of largest x-coordinate (or $(u+v)$-value) in this quadrant. Each of these queries can be answered in $O(\log^2 n)$ time, by standard techniques, using the information stored at the secondary nodes.

To update our range tree efficiently when swaps take place, we make each secondary tree a dynamic balanced search tree data structure that supports insertions and deletions. These updates are required when we encounter swaps in the u-order between consecutive elements—see below.

When we insert a point p into a secondary tree T_w, we may have to update the information associated with the nodes along the path from the root of T_w to the new point, some of which may participate in rebalancing rotations. For any node ξ on the update path in T_w, we add p to $P(w, \xi)$, which may now become the point with maximal or minimal x-coordinate in that set. Notice that, for a secondary node ξ in some T_w, it is straightforward to compute the x-maximal and x-minimal points in $P(w, \xi)$ from the corresponding values at the children of ξ. Therefore, it is easy to maintain these values, when we insert or delete a point and rebalance a secondary tree via rotations, in $O(\log n)$ time per such update. Since only $O(\log n)$ secondary trees are affected by any swap (see below), the cost of updating the overall tree structure after a swap is $O(\log^2 n)$. The same discussion applies in the case of deletions.

Handling u-swaps. Let $p, q \in P$ be the pair of points that switch their positions in the u-order. See Figure 4(i). This causes p and q to swap their positions in the primary tree T and in the list L_u. Let w be the lowest common ancestor of p and q in T. When we swap p and q, we have to delete p from the secondary tree $T_{w'}$ of every node w' on the path from w to the leaf that contained p before the swap, and add q to each such tree. Similarly, we have to delete q from, and add p to, every secondary tree $T_{w'}$, for w' on the path from w to the leaf containing q before the swap. Since we insert and delete p and q in $O(\log n)$ secondary trees, the total update time of these trees is $O(\log^2 n)$. Next, we swap p and q in L_u, and update the type (i) certificates associated with p, q,

[6]Here we do not need to explicitly maintain the pair $\pi(w, \xi) = (\mathsf{blue}(w, \xi), \mathsf{red}(w, \xi))$ at ξ, since we do not handle the set Π^* of these pairs. This set will come into play when we make the structure dynamic, in Section 5.

and their neighbors in L_u. Finally, we find the changes in the set Π of matched pairs, and update the items in $\mathcal{Q}$ accordingly. The procedure described below applies also for v-swaps and x-swaps, without any modifications.

In view of Lemma 4.1, only pairs involving p or q can start or stop being matched. To find these changes, we query T with the wedges $W^+(p)$, $W^-(p)$, $W^+(q)$, and $W^-(q)$, seeking the leftmost or rightmost point in each wedge, as appropriate. This yields at most four candidate pairs (each consisting of the apex (p or q) of the query wedge and the output point) for being new matched pairs. We check each of these pairs whether it is indeed matched. Let (p, z) be one of these pairs, say, with p lying to the left of z. We query in T with the left wedge $W^-(z)$. If the output is p, the pair is matched. Otherwise, it is not matched and we discard it.

We insert into $\mathcal{Q}$ the new matched pairs, and delete from it the old pairs that involve p or q if they are not in Π anymore. To find the old pairs easily, we maintain two pointers from each $p \in P$ to the (at most) two matched pairs that contain it, one as a left point and one as a right point. It is easy to maintain these links as we insert and delete pairs from the tournament.

The cost of this swap is twofold: the cost of updating T, and the cost of handling the tournament structure $\mathcal{Q}$. Updating T takes $O(\log^2 n)$ time. Querying T to compute the new matched pairs also takes $O(\log^2 n)$ time. Inserting and deleting a constant number of pairs into $\mathcal{Q}$ also takes $O(\log^2 n)$ time, as discussed in Section 3. Hence, the cost of a u-swap is $O(\log^2 n)$.

Handling v-swaps. When two points $p, q \in P$ switch their positions in the v-order, we have to swap them in L_v and any secondary tree that contains both of them. Specifically, let z be the lowest common ancestor of p and q in the primary tree T. Then p and q are stored in T_ζ for every ancestor ζ of z. For any such ancestor ζ (including z), we swap p and q in T_ζ. Fix one such tree T_ζ, and notice that p and q are stored at consecutive leaves of T_ζ. Let η be the lowest common ancestor in T_ζ of those two leaves. We swap p and q, and update the x-maximal and x-minimal points stored at each node ξ on the paths from the leaves containing p and q to η. Next, we update the certificates of type (ii), and their failure events, associated with p and q and their neighbors in L_v. Finally, we compute the new matched pairs in Π that arise from the swap (because x-maximal and x-minimal points may have changed), and update $\mathcal{Q}$, exactly as in the case of u-swaps. The total cost of the swap is, as above, $O(\log^2 n)$.

Handling x-swaps. Let p and q be the pair of points that switch their positions in the x-order, with p preceding q before the event takes place. This event does not cause any structural changes, neither in the primary T nor in any secondary tree, but it may change the x-maximal or the x-minimal points in any node ξ of any secondary tree T_w, such that the subtree of ξ contains both p and q.

More precisely, let z be the lowest common ancestor of p and q in the primary T. We need to process the swap of p and q in the secondary trees rooted at the ancestors of z, including z itself. Let w be a fixed ancestor of z, and let η be the lowest common ancestor of p and q in T_w. The subtrees of T_w containing both p and q are the subtrees rooted at ancestors of η (including η itself). Let ξ be one such ancestor. If the x-minimal point in $P(w, \xi)$ is p then we change it to q, and if the x-maximal point in $P(w, \xi)$ is q then we change it to p. Finally, we swap p and q in L_x, and update the certificates of type (iii), and their failure events, associated with p, q, and their neighbors in L_x.

We then compute the new matched pairs that arise from the swap, and update $\mathcal{Q}$, exactly as in the case of u-swaps. The total cost of the swap is, as above, $O(\log^2 n)$.

Theorem 4.2 below summarizes gives main result of this section.

Theorem 4.2. *The KDS for the closest pair described above has the following properties:*

(a) It processes $O(n^2\beta_{s+2}(n)\log n)$ events (thus the KDS is efficient).

(b) It takes $O(\log^2 n)$ time to process an event (thus the KDS is responsive).

(c) Each point p participates in a constant number of certificates of type (i), (ii), and (iii), and in $O(\log n)$ certificates of $\mathcal{Q}$, involving pairs of Π that contain p (thus the KDS is local).

(d) The KDS uses $O(n\log n)$ space (and is thus compact).

Proof. The proof of (b), (c), and (d) follows directly from the construction of the data structure and from the preceding analysis. We note that the size of the event queue is only $O(n)$. The bound on the number of events has been discussed in the paragraph preceding Lemma 4.1. □

Remark: The time bounds of Theorem 4.2 are the same as those of [10]. The space is larger by a $O(\log n)$ factor. However, the advantages of our KDS are that it is considerably simpler, is suitable for dynamization (see Section 5), and can easily be extended to higher dimensions (see Section 8).

5 Dynamizing the KDS for Closest Pair

In this section we show how to make the KDS described in Section 4 also support insertions and deletions of points. We have to be careful here, though, because an insertion or a deletion of a point may cause massive changes in the set Π, as is illustrated in Figure 5. We overcome the problem by maintaining the set Π^*, as defined in Section 2, rather than the set Π, in a kinetic and dynamic tournament $\mathcal{Q}$.

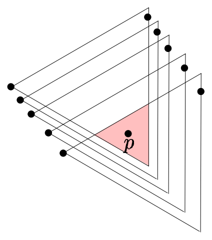

Figure 5. An insertion of a point p into the shaded region destroys $\Theta(n)$ pairs in Π, and deleting the point re-exposes them. In contrast, most of these pairs (those for which the v-coordinates of their points are both smaller than v_p) remain in Π^* when p is inserted or deleted.

At each node ξ of a secondary tree T_w, we store $\mathsf{red}(w,\xi)$ and $\mathsf{blue}(w,\xi)$. If $\pi(w,\xi)$ is defined, it generates an item in the tournament $\mathcal{Q}$ whose value at time t is the Euclidean distance between $\mathsf{red}(w,\xi)$ and $\mathsf{blue}(w,\xi)$ at time t. Since $|\Pi^*| = O(n\log n)$, $\mathcal{Q}$ uses $O(n\log n)$ storage. In contrast to Π, maintaining Π^* at an insertion or deletion can be made efficient, as we will shortly show.

Kinetization. Before describing the procedure for inserting or deleting a point, we consider the kinetization of the modified data structure. As in the case of Π, the set Π^* can change only at a u-swap, a v-swap, or an x-swap (or at an insertion or deletion, which we consider next). At any such swap, the tree T is updated as described in Section 4. The only changes in Π^* involve pairs stored along the paths where the updates of T and its secondary trees take place. Hence, only $O(\log^2 n)$ pairs have to be removed from Π^*, and $O(\log^2 n)$ new pairs have to be inserted. The corresponding updates in the tournament structure that represents Π^* can be naively performed in $O(\log^4 n)$ time per update, as described in Section 3.

We can slightly improve this cost, by noting that the nodes of the tournament Ω that are affected by the updates are those along the paths from the modified leaves to the root. In the worst case, the number of such nodes can be $O(\log^3 n)$, but we can reduce it to $O(\log^2 n)$, if we make the structure of the tournament Ω *identical* to that of the 2-tier range tree T. In fact, we embed the tournament Ω into the range tree T, as follows.

At each node ξ of a secondary tree T_w, we maintain $\pi(w, \xi)$ (whenever it is defined) and we also keep track of the closest pair among the pairs $\pi(w, \zeta)$, where ζ is a descendant of ξ in T_w. We denote this pair by $\pi^*(w, \xi)$, which is $\pi^*(w, \ell(\xi))$, $\pi^*(w, r(\xi))$, or $\pi(w, \xi)$. For each such node ξ we maintain a certificate that asserts which among $\pi^*(w, \ell(\xi))$, $\pi^*(w, r(\xi))$, and $\pi(w, \xi)$ is the closest pair. The failure times of these certificates are maintained in the event queue, where we also store the times of the upcoming u-swaps, v-swaps, and x-swaps. Similarly, each node $w \in T$ keeps track of the closest pair among the pairs $\pi(u, \zeta)$, where u is a descendant of w in T, and $\zeta \in T_u$. We denote this pair by $\pi^*(w)$, which is $\pi^*(\ell(w))$, $\pi^*(r(w))$, or $\pi^*(w, \eta)$, where η is the root of T_w. For each such node w we keep a certificate that asserts which among $\pi^*(\ell(w))$, $\pi^*(r(w))$, and $\pi^*(w, \eta)$ is the closest pair, and keep the failure times of these certificates in the same event queue. We call these two classes of certificates *tournament certificates*. Since the tournament is now embedded into T, we make the secondary trees weight-balanced trees, in accordance with the strategy used in Section 3.

When handling swaps, we update the tournament certificates at all nodes affected by the update. Since there are $O(\log^2 n)$ secondary nodes and $O(\log n)$ primary nodes affected by each such update, we handle each swap in $O(\log^3 n)$ time.

We handle a failure of a tournament certificate at a secondary node ξ in T_w, by updating $\pi^*(w, \xi)$ and propagating up this new closest pair. This may cause the tournament certificates of ancestors of ξ in T_w and ancestors of w in T to change, and therefore takes $O(\log^2 n)$ time. We handle a failure of a tournament certificate at a primary node w similarly.

Lemma 5.1. *The data structure processes $O(n^2 \beta_{s+2}(n) \log^3 n)$ events.*

Proof. We bound the total number of events due to failures of tournament certificates using the same technique as in Section 3. A failure of a tournament certificate at a secondary node ξ in T_w corresponds to a breakpoint in the lower envelope of the distance functions between components of pairs $\pi(w, \zeta)$, for all descendants ζ of ξ in T_w. The complexity of this envelope is proportional to the number of such functions times a factor of $\beta_{s+2}(n)$. Similarly, a failure of a tournament certificate at a primary node $w \in T$ corresponds to a breakpoint in the lower envelope of the distance functions between components of pairs $\pi(u, \zeta)$, over all descendants u of w in T, and $\zeta \in T_u$.

Let $\Phi(w,\xi)$ be the multiset of pairs (p,q) that are stored as $\pi(w,\zeta)$, for descendants ζ of ξ in T_w, where each pair (p,q) is counted with multiplicity equal to the number of maximal connected time intervals during which (p,q) is stored as $\pi(w,\zeta)$, over all nodes ζ as above. Similarly, let $\Phi(w)$ be the multiset of pairs (p,q) that are stored as $\pi(u,\zeta)$, over all descendants u of w and nodes ζ of T_u, where the multiplicity of a pair is defined as above. With this notation, the total number of events is $O(\beta_{s+2}(n))$ times

$$\sum_{w,\xi} |\Phi(w,\xi)| + \sum_{w} |\Phi(w)| \,. \tag{5}$$

Before the motion starts, the sum (5) is $O(n\log^2 n)$. Each swap may increase this sum by $O(\log^3 n)$. This is because each swap may change $O(\log^2 n)$ pairs $\pi(w,\zeta)$, and each such new pair contributes a new element to $O(\log n)$ sets $\Phi(w,\xi)$ and $\Phi(v)$, where ξ is an ancestor of ζ in T_w, and v is an ancestor of w in T. The u-swaps also increase $\sum_{w,\xi}|\Phi(w,\xi)|$ by a total of $O(n^2\log^2 n)$, due to rotations (this bound is a consequence of the fact that the secondary trees are maintained as weight-balanced trees). Summing up, we obtain that the total number of events due to failures of tournament certificates is bounded by $O(n^2\beta_{s+2}(n)\log^3 n)$. $\square$

It is easy to see that our modified data structure requires $O(n\log n)$ space. Furthermore, each point participates in at most $O(\log^2 n)$ tournament certificates, so the data structure is local.

Dynamization. We next turn to the implementation of insertions and deletions of points. For this, we make the primary tree a weight-balanced tree too. (We recall that this is the way in which standard dynamization of range trees is implemented [24].) When performing a rotation around an edge (ξ,η) in the primary tree, we have to rebuild the secondary trees T_ξ and T_η because a complete subtree moves from one tree to the other; see Figure 3. The weight-balanced representation allows us to amortize the work associated with such massive rebuildings. We also maintain the secondary trees as weight-balanced binary search trees, as above. Here rotations are less expensive, since they only entail pointer changes, and do not require any massive rebuilding, but we still need the weight-balanced mechanism to bound the total number of events. In addition to making the range trees dynamic, we also maintain a dynamic search tree over the list L_x. Notice that we already have such a search tree over L_u, which is our primary tree T, and we have a search tree over L_v, which is the secondary tree T_r associated with the root r of the primary tree.

To perform an insertion of a point p, we first insert it into the primary tree T, and then into all secondary trees on the search path from the root r of T to the primary leaf containing p. While performing these insertions, we update the tournament certificates of all nodes along the insertion paths in these trees. We also insert p into L_u, L_v, and L_x, using the search trees over these lists to locate the places where p should be inserted. We create new order certificates, associated with p and its neighbors in the lists L_u, L_v, and L_x, and delete the corresponding previous certificates.

For a rotation around an edge (ξ,η), where $\xi = p(\eta)$, in the primary tree, we rebuild T_ξ and T_η, as follows. Using the notation in Figure 3, the subtrees A, B, and C themselves are not affected by the rotation, so no update of the corresponding secondary trees is required. Updates are required in the new secondary trees T_ξ and T_η. For T_ξ, we merge the v-sorted lists of the elements of B and C into a common sorted list, and then (re)construct T_ξ over this list. Both steps take $O(|B|+|C|)$ time.

For T_η, we simply use the old secondary tree T_ξ as the new T_η. Hence, the properties of weight-balanced trees imply that the total cost of these rebuildings, during a sequence of m updates, is $O(m \log n)$.

It follows that the overall (amortized) cost of an insertion is dominated by the update of $O(\log^2 n)$ tournament certificates, which take $O(\log^3 n)$ time (using the optimization described above). Deletions are performed in an analogous manner.

The proof of the following theorem is analogous to the proof of Theorem 4.2.

Theorem 5.2. *The dynamic KDS for the closest pair, as described above, has the following properties.*

(a) The number of events during a sequence of m insertions and deletions into a KDS of size at most n at any time (assuming $m \geq n$), is $O(mn\beta_{s+2}(n)\log^3 n)$. This makes the KDS efficient.

(b) The time it takes to process an event is $O(\log^3 n)$ (thus the KDS is responsive).

(c) Each point participates in a constant number of order certificates, and in $O(\log^2 n)$ tournament certificates (thus the KDS is local).

(d) The KDS requires $O(n \log n)$ space (and is thus compact).

(e) An insertion or a deletion takes $O(\log^3 n)$ amortized time.

Proof. The proof of (b), (c), (d), and (e) follows directly from the construction of the data structure and from the analysis in Section 4, combined with the analysis given above. It remains to bound the total number of events, which we do as follows.

The number of failure events of order certificates in the lists L_u, L_v, L_x is $O(mn)$, because any newly inserted element can swap its position, in any of the three orders, with at most n *older* elements—those present at the time of insertion.

We bound the number of tournament events, as in Lemma 5.1, by charging them to breakpoints in lower envelopes and by bounding the sum in Equation (5). Each insertion, deletion, or swap increases this sum by $O(\log^3 n)$, including the amortized contribution of rotations, and therefore the number of tournament events is $O(mn\beta_{s+2}(n)\log^3 n)$.

To complete the proof of (a), we argue that the structure is efficient, by showing that, in this dynamic and kinetic setup, the number of closest pairs can be $\Theta(mn)$. A simple construction that shows this involves $n-1$ stationary points lying on the x-axis, and m additional points, where each new point p is inserted into the x-axis to the left of all stationary points, moves to the right and crosses each of the stationary points, and is then deleted. Hence, the data structure is efficient. □

6 KDS for Nearest Neighbors

This structure is an enhancement of the structure presented above for closest pairs. This enhancement is somewhat involved, though:

(a) It requires adding certain substructures to the nodes of the range tree of Section 4.

(b) In order to get a sharp bound on the number of events, we need to implement the primary and secondary trees as *treaps* [27], so our algorithm becomes randomized, and its performance bounds hold in expectation.

(c) The standard implementation of treaps stores an item at each node, rather than just at the leaves [27]. This requires some technical changes in the way in which the range-tree data structure and its auxiliary data are maintained.

We maintain at each secondary node ξ the points $\mathsf{blue}(w,\xi)$ and $\mathsf{red}(w,\xi)$ (whose definition slightly changes, because of the treap structure—see below). As in Section 4, the structure of the primary or a secondary tree changes only by u-swaps and v-swaps, and the points $\mathsf{blue}(w,\xi)$ and $\mathsf{red}(w,\xi)$ may also change as a result of an x-swap. The winner points $\beta(w,\xi)$ and $\varrho(w,\xi)$, however, may change even when no u-swap, v-swap, or x-swap occurs.

To keep track of the points $\beta(w,\xi)$ and $\varrho(w,\xi)$, for every primary node w and secondary node ξ, we store at each secondary node ξ two kinetic and dynamic tournaments, as described in Section 3. Let $a = \mathsf{red}(w,\xi)$ and $b = \mathsf{blue}(w,\xi)$ at some time t. The first tournament at ξ, denoted by $\mathcal{B}(w,\xi)$, contains the distances $\varphi_{p,a}$, defined in (4), for each point $p \in \mathsf{Blue}(w,\xi)$. The second tournament at ξ, denoted by $\mathcal{R}(w,\xi)$, contains the distances $\varphi_{q,b}$ for each point $q \in \mathsf{Red}(w,\xi)$. Thus $\beta(w,\xi)$ and $\varrho(w,\xi)$ are the respective winners of the kinetic tournaments $\mathcal{B}(w,\xi)$ and $\mathcal{R}(w,\xi)$.

In addition, for each point $p \in P$, we maintain another small kinetic and dynamic tournament, denoted $\mathcal{K}(p)$, which contains the distances $\varphi_{p,q}$, for each point $q \in \mathcal{C}(p)$, where $\mathcal{C}(p)$ is the set of candidate points defined in (3). The basic properties of the range tree T, and of nearest neighbors in planar point sets, established in Section 2, imply that the nearest neighbor of p is the winner of $\mathcal{K}(p)$ for one of the range trees corresponding to one of the three pairs of wedges.

The new tournaments $\mathcal{B}(w,\xi)$ and $\mathcal{R}(w,\xi)$ at secondary nodes may undergo massive changes during u-swaps, v-swaps, and x-swaps. Each time $\mathsf{red}(w,\xi)$ changes, we have to rebuild $\mathcal{B}(w,\xi)$ from scratch, since all trajectories of the items in $\mathcal{B}(w,\xi)$ change algebraically. Similarly, when $\mathsf{blue}(w,\xi)$ changes, we have to rebuild $\mathcal{R}(w,\xi)$ from scratch. In addition, a rotation around an edge (η,ξ) in T_w also requires rebuilding of $\mathcal{B}(w,\xi)$, $\mathcal{R}(w,\xi)$, $\mathcal{B}(w,\eta)$, and $\mathcal{R}(w,\eta)$, because the sets of points in the left and right subtrees of ξ and of η change in a massive manner (and also because $\mathsf{red}(w,\xi)$, $\mathsf{blue}(w,\xi)$, $\mathsf{red}(w,\eta)$, or $\mathsf{blue}(w,\eta)$, may change). To control the potential increase in the cost of performing swaps, due to rebuildings of the new tournaments, we use the scheme of Alexandron et al. [8], which stores T and each of its secondary trees as *treaps* (also known as *randomized search trees*) [27].

6.1 Treaps and the data structure

Here is a brief review of treaps and their basic properties; more properties will be established later, as ingredients for our analysis. A *treap* is a randomized search tree with optimal *expected* behavior. We associate with each node z in the treap a *rank*, denoted by $\mathrm{rank}(z)$, and a *priority*, denote by $\mathrm{priority}(z)$. [7] The i-th node encountered when we traverse the tree in symmetric order (or inorder,

[7]Note that priorities are associated with the *nodes* of the tree, rather than with the items that will reside at these nodes.

obtained by recursively traversing the left child, then the node itself, and then the right child) has rank i. The node of rank i stores the ith smallest item (the item of rank i) among the items stored in the treap. The priorities are random numbers, drawn independently and uniformly at random from an appropriate continuous distribution, so that, with probability 1, the set of priorities defines a random permutation of the nodes. The treap is a heap with respect to the *priorities*. That is, the priority of a node is larger than the priorities of its children. Note that, once we draw the priorities, the resulting treap is uniquely determined. The analysis of [27] shows that the expected depth of any node in a treap (over the draws of the priorities) is $O(\log n)$.

To insert a new item x into a treap, we create a new leaf ℓ, in a position determined by the rank of x. Then we draw a random priority for ℓ from the given distribution, and rotate ℓ up the tree, as long as its priority is larger than the priority of its parent. The implementation of a delete operation is similar: Let x be the item to be deleted and let v be the node containing x. We keep rotating the edge connecting v to its child of larger priority, until v becomes a leaf, and then remove v. Note that an insertion or a deletion changes the rank of all subsequent nodes by one, which, however, has no effect on the algorithm, because ranks are maintained only implicitly.

We maintain the primary tree T of our two-dimensional range tree as a treap. This requires a few minor and technical modifications of the structure, caused by the fact that now items are also stored at internal nodes of the tree. Specifically, the node $z \in T$ of rank k stores the point $\mu(z)$, which is the point with the k-th smallest u-coordinate. We refer to the priorities of nodes in this tree as *u-priorities*. We now denote by $P(z)$ the set of points stored at the nodes of the subtree rooted at $z \in T$, including $\mu(z)$ itself. Each secondary tree is maintained as a treap in a similar manner. We use a different independent set of priorities for each secondary tree, which we refer to as *v-priorities*. A node ξ of rank k in a secondary tree T_w stores the point $\mu(w, \xi)$ of the k-th smallest v-coordinate among all points of $P(w)$. We denote by $P(w, \xi)$ the set of points stored at the nodes of the subtree of T_w rooted at ξ, including $\mu(w, \xi)$ itself. Since the expected depth of a treap is $O(\log n)$, we obtain that any point belongs to an expected number of $O(\log n)$ subtrees of the primary tree, and to an expected number of $O(\log^2 n)$ subtrees of secondary trees.

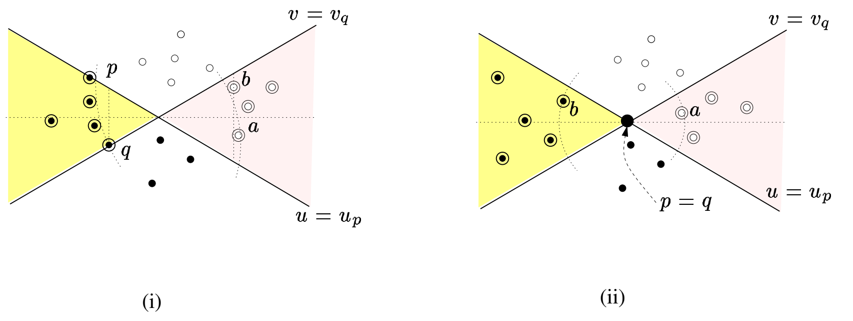

Figure 6. Points of $P(w, \xi)$, $p = \mu(w)$, $q = \mu(w, \xi)$; filled (hollow) circles denote the blue (red) points of $P(w, \xi)$. Points in $\mathsf{Red}(w, \xi)$ and $\mathsf{Blue}(w, \xi)$ are denoted by double circles. (i) $p \neq q$. Here $q = \mathsf{blue}(w, \xi)$, $b = \mathsf{red}(w, \xi)$, $q = \beta(w, \xi)$, and $a = \varrho(w, \xi)$. (ii) $p = q$. $\varrho(w, \xi) = a$, and $\beta(w, \xi) = b$.

Let w be a primary node. Since we now store points also at internal nodes, we have to redefine which points of T_w are red and which are blue, so as to guarantee that Lemma 2.3 still holds, thereby ensuring the correctness of the data structure. As before, each point in $P(\ell(w))$ is blue in T_w, and each point in $P(r(w))$ is red in T_w. The point $\mu(w)$, stored at w, is considered to be both red and blue.

Let ξ be a node in T_w. If $\mu(w,\xi) \neq \mu(w)$ then we redefine $\mathsf{Red}(w,\xi)$ (resp., $\mathsf{Blue}(w,\xi)$) to be the set of red (resp., blue) points in $P(w,r(\xi))$ (resp., $P(w,\ell(\xi))$) together with $\mu(w,\xi)$ if it is red (resp., blue). As earlier, $\mathsf{red}(w,\xi)$ is the point of the minimum x-coordinate in $\mathsf{Red}(w,\xi)$, and $\mathsf{blue}(w,\xi)$ is the point of the maximum x-coordinate in $\mathsf{Blue}(w,\xi)$.

Let $p = \mu(w)$ and $q = \mu(w,\xi)$. We say that node ξ is *special* if $p = q$. Let α_ξ be the intersection point of the lines $u = u_p$ and $v = v_q$. Note that if ξ is special then $\alpha_\xi = p = q$. The set $\mathsf{Red}(w,\xi)$ (resp., $\mathsf{Blue}(w,\xi)$) consists of the points in $P(w,\xi) \setminus \{\alpha_\xi\}$ contained in $W^+(\alpha_\xi)$ (resp., $W^-(\alpha_\xi)$), and $\mathsf{red}(w,\xi)$ (resp., $\mathsf{blue}(w,\xi)$) is the point of $P(w,\xi) \setminus \{\alpha_\xi\}$ with the minimum (resp., maximum) x-coordinate in this quadrant. See Figure 6. We now define the points $\varrho(w,\xi), \beta(w,\xi)$, and the candidate nearest neighbors generated at ξ. There are two cases:

Case A: ***ξ is not special, that is $p \neq q$.*** We define $\varrho(w,\xi)$ to be the point closest to $\mathsf{blue}(w,\xi)$ in $\mathsf{Red}(w,\xi)$, and $\beta(w,\xi)$ to be the point closest to $\mathsf{red}(w,\xi)$ in $\mathsf{Blue}(w,\xi)$. We maintain two tournaments at ξ: a tournament $\mathcal{R}(w,\xi)$ on the distances $\varphi_{a,b}$, over $a \in \mathsf{Red}(w,\xi)$, where $b = \mathsf{blue}(w,\xi)$, and another tournament $\mathcal{B}(w,\xi)$ on the distances $\varphi_{a,b}$, over $a \in \mathsf{Blue}(w,\xi)$, where $b = \mathsf{red}(w,\xi)$. Thus $\varrho(w,\xi)$ (resp., $\beta(w,\xi)$) is the winner of the tournament $\mathcal{R}(w,\xi)$ (resp., $\mathcal{B}(w,\xi)$). We add the point $\varrho(w,\xi)$ to $\mathcal{C}(\mathsf{blue}(w,\xi))$ and $\beta(w,\xi)$ to $\mathcal{C}(\mathsf{red}(w,\xi))$.

Case B: ***ξ is special, that is $p = q$.*** We define $\varrho(w,\xi)$ to be the point closest to $p = q = \mu(w,\xi)$ in $\mathsf{Red}(w,\xi)$, and $\beta(w,\xi)$ to be the point closest to $\mu(w,\xi)$ in $\mathsf{Blue}(w,\xi)$. We maintain two tournaments at ξ: a tournament $\mathcal{R}(w,\xi)$ on the distances $\varphi_{p,a}$, over $a \in \mathsf{Red}(w,\xi)$, and another tournament $\mathcal{B}(w,\xi)$ on the distances $\varphi_{p,b}$, over $b \in \mathsf{Blue}(w,\xi)$. Thus, as above, $\varrho(w,\xi)$ (resp., $\beta(w,\xi)$) is the winner of the tournament $\mathcal{R}(w,\xi)$ (resp., $\mathcal{B}(w,\xi)$). We add the points $\varrho(w,\xi), \beta(w,\xi)$ to $\mathcal{C}(\mu(w,\xi))$. We also add $\mu(w,\xi)$ to $\mathcal{C}(\mathsf{red}(w,\xi))$ and $\mathcal{C}(\mathsf{blue}(w,\xi))$.

The following lemma proves the correctness of our data structure.

Lemma 6.1. *If p is the nearest neighbor of q and $q \in W^+(p)$ or $q \in W^-(p)$, then $p \in \mathcal{C}(q)$.*

Proof. Suppose, without loss of generality, that $q \in W^+(p)$; the other case is symmetric. Let w be the lowest common ancestor of the nodes storing p and q in T. Then $p, q \in P(w)$ and p (resp., q) is blue (resp., red) in T_w; if p (or q) is $\mu(w)$, then it has both colors. Let ξ be the lowest common ancestor of the nodes storing p and q in T_w.

First, we claim that if $\mu(w) = \mu(w,\xi)$, then this point is either p or q. Indeed suppose $\mu(w) = \mu(w,\xi) = a \neq p, q$; see Figure 7 (i). Then p is a blue point in $P(w,\ell(\xi))$ and q is a red point in $P(w,r(\xi))$, thereby implying that $q \in W^+(a)$ and $p \in W^-(a)$. Consequently, $a \in W^+(p)$ and $x_a < x_q$. Hence, by Lemma 2.1, p cannot be the nearest neighbor of q, a contradiction which establishes the claim. The proof continues by considering the following three cases.

Case A: $\mu(w) = \mu(w,\xi) = p$. In this case, $\mathsf{Red}(w,\xi)$ is the set of red points in $P(w,r(\xi))$. Since $q \in W^+(p)$, $q \in \mathsf{Red}(w,\xi)$. Moreover, $\mathsf{Red}(w,\xi) \subset W^+(p)$, therefore by Lemma 2.1, $q = \mathsf{red}(w,\xi)$. Since the data structure adds $\mu(w,\xi)$ to $\mathcal{C}(\mathsf{red}(w,\xi))$ if ξ is special, $p \in \mathcal{C}(q)$. See Figure 7 (ii).

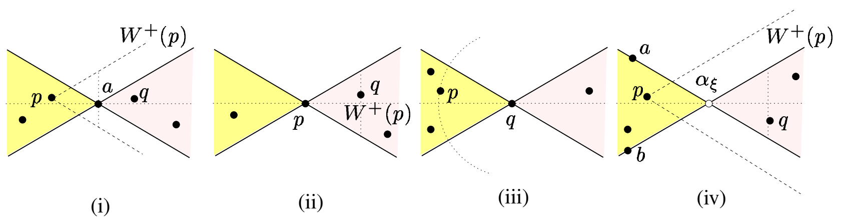

Figure 7. (i) $\mu(w) = \mu(w,\xi) = a \neq p, q$. (ii) $\mu(w) = \mu(w,\xi) = p$, $q = \mathsf{red}(w,\xi)$. (iii) $\mu(w) = \mu(w,\xi) = q$, $p = \beta(w,\xi)$. (iv) $\mu(w) \neq \mu(w,\xi)$.

Case B: **$\mu(w) = \mu(w,\xi) = q$.** In this case, $\mathsf{Blue}(w,\xi)$ is the set of blue points in $P(w,\ell(\xi))$. Since $p \in W^-(q)$, $p \in \mathsf{Blue}(w,\xi)$. Moreover, $p = \beta(w,\xi)$ because p is the nearest neighbor of $q = \mu(w,\xi)$ and thus the winner of the tournament built on the points in $\mathsf{Blue}(w,\xi)$. Since the data structure adds $\beta(w,\xi)$ to $\mathcal{C}(\mu(w,\xi))$ if ξ is special, $p \in \mathcal{C}(q)$. See Figure 7 (iii).

Case C: **$\mu(w) \neq \mu(w,\xi)$.** Let $a = \mu(w)$, $b = \mu(w,\xi)$, and α_ξ the intersection point of the line $u = u_a$ and $v = v_b$. Then $p \in W^-(\alpha_\xi)$ and $q \in W^+(\alpha_\xi)$, thereby implying that $p \in \mathsf{Blue}(w,\xi)$ and $q \in \mathsf{Red}(w,\xi)$. Moreover $\mathsf{Red}(w,\xi) \subset W^+(\alpha_\xi) \subset W^+(p)$, therefore, by Lemma 2.1, $q = \mathsf{red}(w,\xi)$ and p, the nearest neighbor of q, is $\beta(w,\xi)$. We can thus conclude that $p \in \mathcal{C}(q)$. See Figure 7 (iv).

$\square$

6.2 Kinetic maintenance of the structure

We now proceed to describe the details of the kinetic maintenance of the modified tree structure. Similar to the maintenance of the closest-pair KDS in Section 4, the critical events that affect the structure of T and its extreme blue and red points (that is, the points $\mathsf{blue}(w,\xi)$ and $\mathsf{red}(w,\xi)$), for $w \in T$ and $\xi \in T_w$) are the u-swaps, v-swaps, and x-swaps, defined as above. The tournaments maintained at the secondary nodes, as well as the tournaments $\mathcal{K}(p)$, may undergo discrete changes in between swaps. As in Section 4, to keep track of these swaps, we maintain three auxiliary sorted lists L_u, L_v, and L_x, and a collection of $O(n)$ certificates that specify the respective sorted orders of the points by their u-coordinates, v-coordinates, and x-coordinates.

Handling u-swaps. Let $p, q \in P$ be the pair of points that switch their positions in the u-order, so that p precedes q before the swap. This causes them to swap their (consecutive) positions in the primary tree T and in the list L_u. Since now T stores a point at each node, one of these points is an ancestor of the other. Assume that $q = \mu(w)$ and that p is stored at the rightmost leaf of $T_{\ell(w)}$. The case where p is an ancestor of q is handled in a fully symmetric manner. We swap p and q by making $\mu(w) := p$ and by storing q at the leaf that used to store p. This does not change the primary tree structure, but requires the following updates of secondary treaps.

When we swap p and q, we delete p from the secondary tree T_z of every node z on the path from $\ell(w)$ to the leaf that contained p before the swap, and add q to each such tree. In the treap T_w,

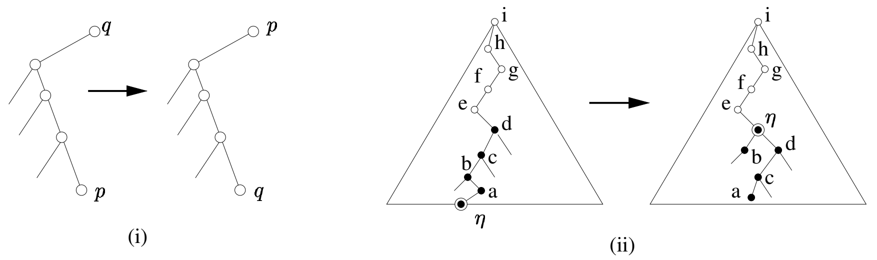

Figure 8. Updating T at a u-swap: (i) Swapping p and q in the primary treap. (ii) Inserting q into a secondary treap T_z, a new leaf η is created, which stores q; the filled (resp., hollow) nodes have lower (resp., higher) priority than that of η; the node η is rotated upwards until it becomes a child of e.

q was both blue and red, and p was blue before the swap. After the swap q is blue and p is both blue and red. This involves no structural changes in T_w, but it does affect the sets $\mathsf{Red}(w,\xi)$ and $\mathsf{Blue}(w,\xi)$, and the tournaments $\mathcal{B}(w,\xi)$ and $\mathcal{R}(w,\xi)$, at nodes ξ along the paths from the root of T_w to the nodes that store p and q, and thus also affect the corresponding extreme points $\mathsf{blue}(w,\xi)$ and $\mathsf{red}(w,\xi)$ and the winners $\beta(w,\xi)$ and $\varrho(w,\xi)$; see Figure 8 (i). Special care is needed at the nodes of T_w that store p and q—one of them stops being a special node and the other becomes special; see Section 6.1 for details.

We insert q to a secondary treap T_z, where z is a node on the rightmost path of the left subtree of w, using the insertion algorithm for a treap. Recall that the insertion puts q in a new node, say η, which is initially a leaf. It then propagates η upwards, using rotations, until $\text{priority}(\eta) < \text{priority}(p(\eta))$; see Figure 8 (ii). When we perform a rotation around an interim edge $(\eta, \xi = p(\eta))$ we recompute $\mathsf{blue}(z,\xi)$ and $\mathsf{red}(z,\xi)$, and we rebuild the tournaments $\mathcal{B}(z,\xi)$ and $\mathcal{R}(z,\xi)$. Once the final position of η is fixed, we update the tournaments $\mathsf{red}(z,\zeta)$, $\mathsf{blue}(z,\zeta)$ and $\mathcal{R}(z,\zeta)$, $\mathcal{B}(z,\zeta)$ at the ancestors ζ of η (after the rotations; hollow nodes in Figure 8 (ii)), as follows. Assume that q is blue after the swap (as in the case considered in Figure 8). For each ancestor node ζ of η such that η lies, say, in the left subtree of ζ we do the following: if the x-coordinate of q is larger than that of $\mathsf{blue}(z,\zeta)$, we set $\mathsf{blue}(z,\zeta)$ to q, and if ζ is not special we rebuild $\mathcal{R}(z,\zeta)$ on the distances $\varphi_{q,a}$, for $a \in \mathsf{Red}(z,\zeta)$. Furthermore, we set $b := \mathsf{red}(z,\zeta)$ if ζ is not special, and $b := \mu(z,\zeta)$ if ζ is special, and we add $\varphi_{b,q}$ to $\mathcal{B}(z,\zeta)$. The treatment of the case in which q is red is analogous. We delete a point p from a secondary treap T_z in a fully symmetric manner.

For each primary node z and secondary node ξ such that $\mathsf{red}(z,\xi)$, $\mathsf{blue}(z,\xi)$, $\varrho(w,\xi)$, or $\beta(w,\xi)$ changes we also make the derived modification to the tournaments $\mathcal{K}(p)$ for the affected points p. Finally, we swap p and q in L_u, and update the $O(1)$ order certificates associated with p, q, and their neighbors in L_u.

Handling v-swaps. As in the structure of Section 4, when two points $p, q \in P$ switch their positions in the v-order, we have to swap them in any secondary tree that contains them both, and in L_v. Specifically, let z be the lowest common ancestor of p and q in the primary tree T. For any ancestor y of z (including z), we swap p and q in T_y. Fix one such secondary treap T_y, and notice that p and q are stored at *consecutive* nodes (in symmetric order) of T_y, thus one of them is an extreme node in a subtree rooted at a child of the other, as in Figure 8(i). Let η be the lowest common ancestor of p and q in T_y. Assume that before the swap η stored q, and p was stored at the rightmost leaf of the left subtree of η. (The case when η holds p and q is stored at the leftmost leaf of the right subtree of η is symmetric.) We swap p and q, and update $\mathsf{blue}(y,\xi)$, $\mathsf{red}(y,\xi)$, $\mathcal{B}(y,\xi)$, and $\mathcal{R}(y,\xi)$, for each node ξ on the path from the leaf containing p to η. We perform an insertion and/or a deletion to/from $\mathcal{B}(y,\xi)$ in case $\mathsf{red}(y,\xi)$ does not change and either p or q is blue. If $\mathsf{red}(y,\xi)$ does change, and ξ is not special, we rebuild $\mathcal{B}(y,\xi)$. Symmetric updates are applied to $\mathcal{R}(y,\xi)$.

Note that either p or q is $\mu(z)$, and therefore in T_z either η or the leaf containing p is special. If η is special before the swap then it stops being special and the leaf that contained p and now contains q becomes special, and vice versa. We recompute $\mathsf{blue}(y,\xi)$, $\mathsf{red}(y,\xi)$, $\mathcal{B}(y,\xi)$, and $\mathcal{R}(y,\xi)$ for these nodes that change their status from special to non-special or vice versa.

For each primary node z and secondary node ξ such that $\mathsf{red}(z,\xi)$, $\mathsf{blue}(z,\xi)$, $\varrho(w,\xi)$, or $\beta(w,\xi)$ changes we also make the derived modification to the tournaments $\mathcal{K}(p)$ for the affected points p. Finally, we swap p and q in L_v, and update the $O(1)$ order certificates associated with p and q and their neighbors in L_v.

Handling x-swaps. Let p and q be a pair of points that switch their positions in the x-order, with p to the left of q before the event takes place. This event does not cause any structural changes in the primary treap, nor in any secondary treap, but it may change $\mathsf{red}(z,\xi)$ in any node ξ of any secondary treap T_z, such that the subtree of T_z rooted at ξ contains both p and q.

Let z be the lowest common ancestor of p and q in the primary treap. We need to process p and q at each ancestor of z, including z itself. Let y be a proper ancestor of z; note that neither p nor q is stored at y. Assume that p and q are both red in T_y, and let η be the lowest common ancestor of p and q in T_y. The subtrees of T_y containing both p and q are the subtrees rooted at ancestors of η (including η itself). At any such ancestor ζ, if $p = \mathsf{red}(y,\zeta)$ then we change $\mathsf{red}(y,\zeta)$ to be q and rebuild $\mathcal{B}(y,\zeta)$. Finally, we swap p and q in L_x, and update the $O(1)$ order certificates associated with p, q, and their neighbors in L_x. The case where both p and q are blue in T_y is analogous. If one is blue and the other is red in T_y (i.e., $y = z$), then we do nothing; this also covers the case when one of p, q is stored as $\mu(z)$.

6.3 Analysis

Some properties of treaps. Before analyzing the expected cost of the various swaps, we provide two related lemmas on the expected size of various substructures in a treap. The proofs are similar to those given in the original paper of Seidel and Aragon [27], but we present them in detail for the sake of completeness.

Lemma 6.2. *Let w be the node of rank k in a treap, and let $\mathcal{W} = \langle w_1, \ldots, w_l \rangle$ be the rightmost path starting from the left child w_1 of w and ending at the leaf w_l. Then, for any nonnegative function f,*

$$\mathbb{E}\left(\sum_{s=1}^{l} f(\mathrm{size}(w_s))\right) \le \frac{f(k-1)}{k} + 2\sum_{m=1}^{k-2} \frac{f(m)}{m^2},$$

where $\mathrm{size}(w)$ *is the size of the subtree rooted at w.*

Proof. For each i, let $w(i)$ denote the node of rank i in the treap. For each pair of indices i, j such that $i \le j < k$, consider the event $\mathbb{X}_{i,j}$, where $w(j)$ is the root of a subtree in T whose leftmost leaf is $w(i)$ and whose rightmost leaf is $w(k-1)$; clearly, in this case this subtree consists of all the vertices $w(l)$, for $i \le l \le k-1$. Note that when $\mathbb{X}_{i,j}$ takes place, $w(j)$ is a node on the path $\mathcal{W}$. Conversely, if $w(j)$ is a node on $\mathcal{W}$ then there exists an $i \le j$ such that $\mathbb{X}_{i,j}$ occurs; moreover, in any random instance of T, this i is unique. We then have

$$\mathbb{E}\left(\sum_{s=1}^{l} f(\text{size}(w_s))\right) \le \sum_{i=1}^{k-1}\sum_{j=i}^{k-1} \Pr(\mathbb{X}_{i,j}) \cdot f(k-i) \ . \tag{6}$$

For $i > 1$, $\mathbb{X}_{i,j}$ occurs if and only if among the nodes $w(s)$, $i-1 \le s \le k$, $w(i-1)$ and $w(k)$ have the two largest priorities, in either order, and $w(j)$ has the third largest priority. Hence,

$$\Pr[\mathbb{X}_{i,j}] = \frac{2(k-i-1)!}{(k-i+2)!} \le \frac{2}{(k-i)^3}.$$

The event $\mathbb{X}_{1,j}$ occurs if among the nodes $w(l)$, $1 \le l \le k$, $w(k)$ has the largest priority and $w(j)$ has the second largest priority. Hence,

$$\Pr[\mathbb{X}_{1,j}] = \frac{1}{k(k-1)}.$$

We can thus rewrite (6) as

$$\begin{aligned}\mathbb{E}\left(\sum_{s=1}^{l} f(\text{size}(w_s))\right) &\le \sum_{j=1}^{k-1} \frac{f(k-1)}{k(k-1)} + \sum_{i=2}^{k-1}\sum_{j=i}^{k-1} \frac{2f(k-i)}{(k-i)^3} \\ &= \sum_{i=2}^{k-1} \frac{2f(k-i)}{(k-i)^2} + \frac{f(k-1)}{k},\end{aligned}$$

which is exactly the inequality asserted in the lemma. □

In particular, for $f(s) = s$, Lemma 6.2 yields

$$\mathbb{E}\left(\sum_{j=1}^{l} \text{size}(w_j)\right) = O(\log n).$$

Lemma 6.3. *Let w be the root of a treap on n points, and let $\mathcal{W} = \langle w = w_1, \ldots, w_l \rangle$ denote the path from w to a node w_l of rank k. Then, for any nonnegative function f,*

$$\mathbb{E}\left(\sum_{s=1}^{l} f(\text{size}(w_s))\right) = O\left(f(n) + \sum_{m=1}^{n} \frac{f(m)}{m+1}\right),$$

where size(u) *is, as above, the size of the subtree rooted at node u.*

Proof. Let $w(i)$ denote the node of rank i in the treap, so, in particular, $w_l = w(k)$. For each triple of indices i, m, j such that $i \le m \le j$ and $i \le k \le j$, consider the event $\mathbb{X}_{i,m,j}$, in which $w(m)$ is the root of a subtree in T whose leftmost node is $w(i)$ and whose rightmost node is $w(j)$; clearly, in this case this subtree consists of all the vertices $w(s)$, for $i \le s \le j$, and, in particular, it contains $w(k)$. Note that when $\mathbb{X}_{i,m,j}$ occurs, $w(m)$ is a node on the path $\mathcal{W}$. Conversely, if $w(m)$ is a node on $\mathcal{W}$ then there exist indices $i \le m \le j$, where we also have $i \le k \le j$, such that $\mathbb{X}_{i,m,j}$ occurs. Moreover, in any random instance of T, these i and j are unique. We then have

$$\mathbb{E}\Bigg(\sum_{s=1}^{l} f(\text{size}(w_q))\Bigg) \le \sum_{i=1}^{k}\sum_{j=k}^{n}\sum_{m=i}^{j} \Pr(\mathbb{X}_{i,m,j}) \cdot f(j-i+1) \,. \tag{7}$$

We bound the right hand side by dividing the summation into four subsums.

Case A: $i > 1$ and $j < n$. In this case $\mathbb{X}_{i,m,j}$ occurs if and only if among the nodes $w(s)$, $i-1 \le s \le j+1$, $w(i-1)$ and $w(j+1)$ have the two largest priorities, in either order, and $w(m)$ has the third largest priority. Hence,

$$\Pr[\mathbb{X}_{i,m,j}] = \frac{2(j-i)!}{(j-i+3)!} \le \frac{2}{(j-i+1)(j-i+2)^2}.$$

Therefore

$$\begin{aligned}\sum_{i=2}^{k}\sum_{j=k}^{n-1}\sum_{m=i}^{j} \Pr(\mathbb{X}_{i,m,j}) \cdot f(j-i+1) \quad &\le \quad \sum_{i=2}^{k}\sum_{j=k}^{n-1}\sum_{m=i}^{j} \frac{2f(j-i+1)}{(j-i+1)(j-i+2)^2} \\ &= \quad \sum_{i=2}^{k}\sum_{j=k}^{n-1} \frac{2f(j-i+1)}{(j-i+2)^2} \le \sum_{i=1}^{n} \frac{2f(i)}{i+1}.\end{aligned} \tag{8}$$

Case B: $i = 1$ and $j < n$. The event $\mathbb{X}_{1,m,j}$, for $j < n$, occurs if among the nodes $w(s)$, $1 \le s \le j+1$, $w(j+1)$ has the largest priority and $w(m)$ has the second largest priority. Hence,

$$\Pr[\mathbb{X}_{1,m,j}] = \frac{1}{j(j+1)},$$

thereby implying that

$$\sum_{j=k}^{n-1}\sum_{m=1}^{j} \Pr(\mathbb{X}_{1,m,j}) \cdot f(j) \le \sum_{j=k}^{n-1}\sum_{m=1}^{j} \frac{f(j)}{j(j+1)} \le \sum_{j=1}^{n} \frac{f(j)}{j+1}. \tag{9}$$

Case C: $i > 1$ and $j = n$. As in the previous case, $\mathbb{X}_{i,m,n}$, for $i > 1$, occurs if among the nodes $w(s)$, $i-1 \le s \le n$, $w(i-1)$ has the largest priority and $w(m)$ has the second largest priority. Therefore,

$$\Pr[\mathbb{X}_{i,m,n}] = \frac{1}{(n-i+2)(n-i+1)},$$

and

$$\sum_{i=2}^{k}\sum_{m=i}^{n} \Pr(\mathbb{X}_{i,m,n}) \cdot f(n-i+1) \le \sum_{i=2}^{k}\sum_{m=i}^{n} \frac{f(n-i+1)}{(n-i+2)(n-i+1)} \le \sum_{i=1}^{n} \frac{f(i)}{i+1}. \tag{10}$$

Case D: $i = 1$ *and* $j = n$. Finally, $\mathbb{X}_{1,m,n}$ occurs if $w(m)$ has the highest priority overall (this is the event where $w(m)$ is the root), which implies that $\Pr[\mathbb{X}_{1,m,n}] = 1/n$. Therefore

$$\sum_{m=1}^{n} \Pr(\mathbb{X}_{1,m,n}) \cdot f(n) = \sum_{m=1}^{n} \frac{f(n)}{n} = f(n). \tag{11}$$

Summing (8)–(11), we obtain

$$\mathbb{E}\Big(\sum_{s=1}^{l} f(\text{size}(w_s))\Big) = O\Big(f(n) + \sum_{j=1}^{n} \frac{f(j)}{j+1}\Big),$$

as asserted. □

We now give the analysis of the data structure. We do not consider updates to the kinetic tournaments $\mathcal{K}(p)$ as it would be easy to verify that the cost of maintaining these tournaments is dominated by the cost of maintaining the tournaments $\mathcal{R}(w, \xi)$ and $\mathcal{B}(w, \xi)$.

The cost of a u-swap. Consider first the cost of rebuilding tournaments during a u-swap, and assume the setup depicted in Figure 8. At each of the affected secondary subtrees T_z (including T_w itself), the tournaments that might have to be rebuilt originate at nodes that lie on two paths from the root to two nodes of specific ranks. The cost of rebuilding a tournament at a node ξ is proportional to the size of the subtree rooted at ξ. Hence, the total expected cost of rebuilding tournaments at some secondary tree T_z is proportional to the expected sum of the sizes of the subtrees rooted at nodes lying along the path to p before it is deleted and along the path to q after it is inserted. By Lemma 6.3, with $f(x) = x$, this expected sum is $O(|T_z|)$. Plugging this bound into Lemma 6.2, the overall expected cost of rebuilding tournaments at a u-swap is $O(\log n)$.

The cost of the other steps that handle a u-swap is smaller—the cost of the actual updating of a secondary tree T_z is only $O(\log |T_z|)$, even if the cost of peforming a rotation around a node ξ is proportional to the size of the subtree rooted by ξ (See Theorem 7.1). This is subsumed by the preceding bound. We thus obtain the following lemma.

Lemma 6.4. *The expected cost of handling a u-swap is $O(\log n)$.*

The cost of a v-swap. Recall that a v-swap of two points p, q requires updates in secondary trees T_z, where z is a common ancestor of the nodes storing p and q. In each such tree T_z, p and q are consecutive, so one is an extreme node in a subtree rooted at a child of the other. Using Lemma 6.2, the expected cost of updating a secondary tree T_z, which consists of swapping p and q and rebuilding the appropriate tournaments, is $O(\log |T_z|)$. Plugging this into Lemma 6.3, with $f(x) = \log x$, we obtain:

Lemma 6.5. *The expected cost of handling a v-swap is $O(\log^2 n)$.*

Expected cost of all x-swaps. A single x-swap may be expensive, also in expectation (see below), but we show that the total cost of all x-swaps is small, arguing as follows. For a node $\xi \in T_w$, let $\Lambda(w, \xi)$ be the multiset of all pairs (a, b) of points of P, with $a \neq b$, such that, at some time t, $a \in \mathsf{Red}(w, \xi)$ and $b \in \mathsf{Blue}(w, \xi)$. The multiplicity of each pair (a, b) in $\Lambda(w, \xi)$ is equal to the number of maximal (connected) time intervals in which the above event occurs. Let $\Lambda = \bigcup_{w,\xi} \Lambda(w, \xi)$, where the union is over all primary nodes w and secondary nodes $\xi \in T_w$. The following lemma bounds the expected total size of tournaments that we rebuild while handling x-swaps.

Lemma 6.6. *The expected total size of tournaments that we rebuild while performing x-swaps is $O(|\Lambda|\beta_{s+2}(n))$.*

Proof. Fix a node ξ in a secondary treap T_w. Let B denote the multiset of all blue points that are ever stored either at the left subtree of ξ or at ξ itself. Each point is counted with multiplicity equal to the number of times it enters this set. Similarly, let R denote the multiset of all red points which are stored either at the right subtree of ξ or at ξ itself, where each point is counted with multiplicity equal to the number of times it enters this set. Associate with each point $b \in B$ the function $x(b(t))$, and let $\mathcal{U} = \mathcal{U}(t)$ denote the upper envelope of these (partial) functions, each defined over some connected t-interval. Since, by assumption, the x-coordinates of a fixed pair of points can become equal at most s times, it follows [29] that the number of breakpoints of $\mathcal{U}$ is at most $\lambda_{s+2}(|B|) = O(|B|\beta_{s+2}(n))$. Similarly, the number of breakpoints in the upper envelope defined by any subset $B' \subseteq B$ of the functions is $O(|B'|\beta_{s+2}(n))$.

Each such breakpoint corresponds to an x-swap at which we rebuild $\mathcal{R}(w, \xi)$. So if we sum the sizes of the tournaments $\mathcal{R}(w, \xi)$, measured at the times these breakpoints occur, we get a bound on the total size of red tournaments at the times when they are rebuilt at ξ. We can get a similar bound on the total size of blue tournaments at the times of their rebuilding. Assume that each time such a breakpoint occurs, we charge one unit to each point in $\mathcal{R}(w, \xi)$. We now bound the maximum number of such charges.

Fix a red point $a \in R$, within a fixed maximal time interval I in which a is in R. Let Ψ_a be the set of breakpoints of $\mathcal{U}$ that charge a; that is, breakpoints that occur within I. Let B_a^* denote the multiset of those blue points whose functions are incident to the breakpoints of Ψ_a. Clearly, the breakpoints of Ψ_a are also breakpoints of the upper envelope of $\{x(b(t)) \mid b \in B_a^*\}$. Hence we have $|\Psi_a| = O(|B_a^*|\beta_{s+2}(n))$. Now the total count of tournament changes under consideration is

$$\sum_{a \in R} |\Psi_a| = O\left(\sum_a |B_a^*|\beta_{s+2}(n)\right).$$

Since $\sum_a |B_a^*| \leq |\Lambda(w, \xi)|$, we conclude that $O(|\Lambda(w, \xi)|\beta_{s+2}(n))$ bounds that total size of red tournaments, measured at the time of their rebuilding, in ξ. Summing over all primary nodes w and secondary nodes ξ, and applying the same argument to blue tournaments, the lemma follows. $\square$

To apply Lemma 6.6, we have to bound $|\Lambda|$. New pairs in Λ are created during the handling of u-swaps and v-swaps. Consider first a u-swap of points p and q. There are two types of pairs that this swap creates: pairs that contain either p or q, and pairs created by the structural changes caused by rotations while deleting q and inserting p (or vice versa) into secondary trees. We refer to pairs of the first (resp., second) kind as *primary* (resp., *secondary*) pairs. The number of primary pairs is bounded by the expected total size of the secondary trees affected by the swap. By Lemma 6.2, this size is bounded by $O(\log n)$.

To bound the number of new secondary pairs, we first show that each insertion into or deletion from a secondary tree of size s creates an expected number of $O(s)$ new secondary pairs.

Lemma 6.7. *If a point is inserted into or deleted from a secondary tree T_w, then the expected increase in the value of $\sum_{\xi \in T_w} \Lambda(w, \xi)$ is $O(|T_w|)$.*

Proof. We analyze deletions in detail; the analysis of insertions is analogous and hence omitted. Assume that the point to be deleted from T_w resides at node ξ of rank m. We examine the rotations that bring ξ down, and bound the expected number of new pairs created by these rotations.

Let $\mathcal{X} = \langle \chi_1 = \ell(\xi), \chi_2, \ldots, \chi_g \rangle$ denote the rightmost path from $\ell(\xi)$ to a leaf, and let $\mathcal{Z} = \langle \zeta_1 = r(\xi), \zeta_2, \ldots, \zeta_h \rangle$ denote the leftmost path from $r(\xi)$ to a leaf, both defined before the rotations begin; see Figure 9. Each rotation in the deletion procedure to bring ξ down is performed along an edge on $\mathcal{X}$ or $\mathcal{Z}$. Let $B = \mathsf{Blue}(w, \xi)$ and $R = \mathsf{Red}(w, \xi)$. For $1 \le i \le g$, let $B_i = \mathsf{Blue}(w, \chi_i)$, and for $1 \le j \le h$, let $R_j = \mathsf{Red}(w, \zeta_j)$.

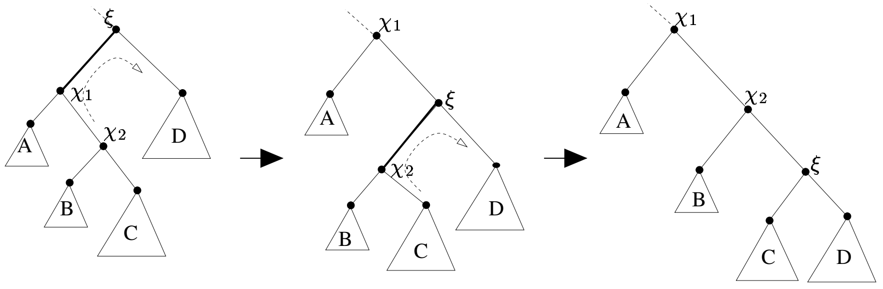

Figure 9. Rotating ξ down by a sequence of right rotations. D becomes part of T_{χ_i} after performing the right rotation along the edge $\xi\chi_i$.

Suppose we first perform right rotations along the edges of $\mathcal{X}$. As shown in Figure 9, a right rotation around the edge between ξ and its current left child $\ell(\xi) = \chi_i$ (starting with $i = 1$) changes the left child of ξ to χ_{i+1}, and ξ becomes the right child of χ_i. $T_{r(\xi)}$ remains unaffected by these rotations (see, e.g., the subtree D in Figure 9.) Since $T_{r(\xi)}$ was disjoint from the right subtree of T_{χ_i} before the rotation, but becomes part of it after the rotation, new pairs are generated in $\Lambda(w, \chi_i)$ by the rotation, namely, the pairs in $B_i \times R$. The only other pairs in Λ that can be generated by this rotation involve the point $p = \mu(w, \xi)$ stored at ξ; there are at most $|B_i|$ such pairs—see Figure 9. Hence, the right rotations introduce $\sum_{i=1}^{g}(|B_i \times R| + |B_i|) \le |B|(|R| + 1)$ new pairs to Λ (note that the B_i's are disjoint—see Figure 9).

Similarly, if we first perform left rotations along $\mathcal{Z}$ (before performing any right rotation), then the rotation around the edge (ξ, ζ_j) introduces $|B \times R_j| + |R_j|$ pairs to $\Lambda(w, \zeta_j)$, for a total of $(|B| + 1)|R|$ pairs. If right and left rotations are performed in any mixed order, then each right rotation creates only a subset of the pairs it would have created if performed before all left rotations, and the same holds for left rotations. Therefore, regardless of the order of the rotations, the total number of new pairs is

$$O((|B| + 1)(|R| + 1)) = O((|P(w, \ell(\xi))| + 1) \cdot (|P(w, r(\xi))| + 1));$$

here we are referring to the children of ξ before the rotations. It thus suffices to bound the expected value of this quantity.

Let $\xi(i)$ denote the node of rank i in the treap T_w, so in particular $\xi = \xi(m)$. For $i \le m \le j$, define $\mathbb{X}_{i,j}$ to be the indicator random variable of the event in which the node $\xi(m)$, storing m in T_w, is the lowest common ancestor of $\xi(i)$ and $\xi(j)$. The expected value of $(|P(w,\ell(\xi))|+1)\cdot(|P(w,r(\xi))|+1)$, for a fixed node $\xi = \xi(m)$, is

$$\sum_{i\le m}\sum_{j\ge m}\mathbb{E}(\mathbb{X}_{i,j})\,.$$

For $\mathbb{X}_{i,j}$ to be 1, $\xi(m)$ must have the largest priority among all nodes $\xi(k)$, for $i \le k \le j$. The probability of this event is clearly $1/(j-i+1)$. Summing up over all pairs (i,j) with $i \le m \le j$, we get that

$$\sum_{i\le m}\sum_{j\ge m}\mathbb{E}(\mathbb{X}_{i,j}) = \sum_{i\le m}\sum_{j\ge m}\frac{1}{j-i+1} \le \sum_{k=0}^{|T_w|}(k+1)\frac{1}{k+1} = O(|T_w|).$$

That is, we have shown that the expected increase in the sum $\sum_\xi |\Lambda(w,\xi)|$, caused by generating new secondary pairs, is $O(|T_w|)$. □

Combining this with Lemma 6.2, we obtain that the expected number of new secondary pairs is $O(\log n)$. Since we have $O(n^2)$ u-swaps altogether, it follows that the total contribution of u-swaps to Λ is $O(n^2\log n)$ pairs.

Consider now a v-swap of points p and q. Here there are no structural changes in any tree, and each newly created pair contains either p or q. By Lemma 6.2, with $f(x) = x$, the expected total size of the affected subtrees in a secondary treap T_u containing both p and q is $O(\log|T_u|)$. Applying Lemma 6.3 to the primary tree, we obtain that the expected number of new pairs created by a single v-swap is $O(\log^2 n)$. In total we have $O(n^2)$ v-swaps, which contribute an expected number of $O(n^2\log^2 n)$ new pairs to Λ.

Lemma 6.8. *The expected cost of all x-swaps is $O(n^2\beta_{s+2}(n)\log^2 n)$.*

Proof. Since the expected depth of a node in a treap is $O(\log n)$, it follows that, initially, the expected number of sets $\Lambda(w,\xi)$ in which a pair of points of P appears is $O(\log^2 n)$. So the expected initial size of Λ is $O(n^2\log^2 n)$. By the preceding discussion, the expected contribution to Λ by all u-swaps and v-swaps is also $O(n^2\log^2 n)$. Thus the expected size of Λ is $O(n^2\log^2 n)$. Combining this with Lemma 6.6, we obtain that the expected total size of tournaments, measured at the time of their rebuilding, is $O(n^2\beta_{s+2}(n)\log^2 n)$. By Theorem 3.1, the total time to rebuild these tournaments is also $O(n^2\beta_{s+2}(n)\log^2 n)$. This bounds the total time spent in handling all x-swaps. □

Expected cost of a single x-swap. Recall that the time spent in an x-swap of two points p and q is proportional to the expectation of the sum, over all secondary trees T_z that contain both p and q, of the sum of the sizes of the subtrees rooted at nodes along the path leading from the root to the lowest common ancestor of p and q. Applying Lemma 6.3, with $f(s) = s$, to such a secondary treap

T_z, we obtain that the expected contribution of T_z to this sum is $O(|T_z|)$. Now, applying Lemma 6.3 again in the primary tree T, we obtain that the expected total size of all tournaments affected by the swap is $O(n)$.

We summarize the result of this section in the following theorem.

Theorem 6.9. *Our KDS for maintaining the nearest neighbor of each point in a set of n moving points in the plane has the following properties.*

1. *The number of u-swaps is $O(n^2)$, and handling a u-swap takes $O(\log n)$ expected time.*
2. *The number of v-swaps is $O(n^2)$, and handling a v-swap takes $O(\log^2 n)$ expected time.*
3. *The number of x-swaps is $O(n^2)$, processing a single x-swap takes $O(n)$ expected time, and processing all x-swaps takes $O(n^2 \log^2 n\beta_{s+2}(n))$ expected time.*
4. *The number of tournament events is $O(n^2 \log^3 n\beta^2_{s+2}(n))$, and the total time required to handle them is $O(n^2 \log^4 n\beta^2_{s+2}(n))$.*
5. *The data structure requires $O(n \log^2 n)$ expected storage.*

In particular, the KDS is compact, efficient, responsive in an amortized sense, but in general not local.

Proof. The proof of (1), (2), and (3) follows from our assumption on the motion and from Lemmas 6.4, 6.5, 6.8, and the preceding discussion.

To bound the number of tournament events, recall, from the proof of (3), that, over the entire motion, all tournaments together contain $O(n^2 \log^2 n\beta_{s+2}(n))$ items (where each item is counted with multiplicity equal to the number of times it is inserted into the tournament). The bounds claimed in (4) now follow from Theorem 3.1. The expected storage required by the structure is dominated by the expected total size of all tournaments $\mathcal{B}(w, \xi)$, $\mathcal{R}(w, \xi)$. which is bounded by the expected sum of the sizes of all subtrees over all secondary trees. Since the expected depth of the primary tree and each of the secondary subtrees is $O(\log n)$, this expected sum is $O(n \log^2 n)$. Thus (5) follows. □

7 Dynamizing the KDS for Nearest Neighbors

Our kinetic data structure for nearest neighbors of Section 6 can in fact support insertions and deletions of points. We only need to add a dynamic search tree over the lists L_u, L_v, and L_x, as we did for the closest pair problem in Section 5. Here too the primary treap T can serve as the tree associated with L_u, and the secondary treap associated with the root of the primary treap can serve as the tree associated with L_v.

To perform an insertion of a point p, we first insert it into the primary treap T. For a rotation around an edge (z, w) in the primary tree (with w the former parent of z), we rebuild T_w and T_z, and the tournaments that they store. This takes $O(|T_w| \log n)$ expected time. Seidel and Aragon

[27] proved the following lemma (which is similar to Lemma 6.2); it shows that rotations in treaps are not that expensive in expectation.

Lemma 7.1. *Assume that a rotation around an edge $(x, y = p(x))$ in a treap takes $O(f(s))$ time, where s is the size of the subtree rooted at y. Then the expected time to perform an insertion or a deletion to/from the treap is*

$$O\left(\frac{f(n)}{n} + \sum_{1 \le s \le n} \frac{f(s)}{s^2}\right).$$

Lemma 7.1 implies that the insertion of p into the primary treap takes $O(\log^2 n)$ expected time.

Let z be the node containing p in the primary treap T. After inserting p into the primary treap, we insert p into every secondary treap T_u, where u is an ancestor of z in T. While inserting p into a secondary treap T_u, we also have to insert p into a tournament at each node on the path to p in T_u. Furthermore, if p becomes $\mathsf{red}(u, \xi)$ or $\mathsf{blue}(u, \xi)$, for some (non-special) node $\xi \in T_u$, then we have to rebuild the tournament $\mathcal{B}(u, \xi)$ or $\mathcal{R}(u, \xi)$, respectively. The node η containing p in T_z is special and we rebuild $\mathcal{B}(u, \eta)$ and $\mathcal{R}(u, \eta)$.

The expected time it takes to insert p into all secondary trees containing it is $O(\log^2 n)$. Updating tournaments, however, may be expensive. Nevertheless, this time is bounded by the expected sum of the sizes of all secondary subtrees containing p, which is $O(n)$, by Lemma 6.3. We summarize with the following theorem.

Theorem 7.2. *Suppose we make $m \ge n$ insertions and deletions to the kinetic and dynamic data structure for nearest neighbors described above, such that there are at most n points in the data structure at any fixed time. Then the following properties hold.*

1. *The number of u-swaps is $O(mn)$ and processing a u-swap takes $O(\log n)$ expected time.*
2. *The number of v-swaps is $O(mn)$ and processing a v-swap takes $O(\log^2 n)$ expected time.*
3. *The number of x-swaps is $O(mn)$, processing a single x swap takes $O(n)$ expected time, and processing all x-swaps takes $O(mn \log^2 n \beta_{s+2}(n))$ expected time.*
4. *The expected number of tournament events is $O(mn \log^3 n \beta_{s+2}^2(n))$, and their total expected cost is $O(mn \log^4 n \beta_{s+2}^2(n))$.*
5. *The data structure requires $O(n \log^2 n)$ space.*
6. *An insertion or a deletion takes $O(n)$ expected time.*

Proof. Arguing as in the proof of Theorem 5.2, the number of u-swaps, v-swaps, and x-swaps is $O(mn)$. It takes $O(n)$ expected time to insert or delete a point by the discussion preceding this theorem.

The rest of the proof is analogous to the proof of Theorem 6.9. The only difference is that we have to take into account the increase in Λ when we insert or delete a point p. Clearly there are $O(n)$ new primary pairs (containing p) in Λ. Secondary new pairs are created as a result of rotations in the primary treap, which cause rebuildings of secondary treaps. Each secondary treap T_w that is rebuilt may contribute $O(|T_w|^2)$ new pairs to Λ. However, Lemma 7.1 implies that the expected total number of new pairs is $O(n)$, so the increase of $|\Lambda|$ caused by insertions and deletions is $O(mn)$, and it therefore does not dominate the size of Λ. $\square$

8 Extension to Higher Dimensions

In this section, we extend the data structures of Sections 5 and 7 to fully dynamic and kinetic data structures for maintaining the closest pair and all nearest neighbors in a set P of n moving points in $\mathbb{R}^d$, for any $d \geq 3$. The extension is straightforward, and is based on the following generalization of the key geometric property, given in Lemma 2.1. The proof is essentially identical to the preceding proof, and is thus omitted.

Lemma 8.1. *Let p be the closest point to q, and let C be a cone of opening angle $\pi/3$, with apex at the origin, which contains $q - p$. Let b denote a vector in the direction of the symmetry axis of C (pointing into C). Then*

$$(q-p)\cdot b = \min\{(w-p)\cdot b \mid w \in P \cap (p + C)\}.$$

We tile $\mathbb{R}^d$ by a constant number of convex polyhedral cones, all having the origin o as their apex, such that each of these polyhedral cones is bounded by d facets, and is contained in a regular cone[8] of opening angle $\pi/3$ with apex o. Note that the number of polyhedral cones grows exponentially with d. As in the planar case, we may assume that, for each polyhedral cone W, its antipodal cone $-W$ also appears in the tiling. We describe the extension for closest pair, since the extension for all nearest neighbors is similar.

Let W be one of these polyhderal cones. Without loss of generality, assume that the symmetry axis of the $\pi/3$ cone that contains W is the positive x-axis. Clearly, Lemma 8.1 also holds for W. That is, if p, q is a closest pair at time t such that $q - p \in W$, then

$$(q-p)_x = \min\{(w-p)_x \mid w \in P \cap (p + W)\}.$$

That is, q is the leftmost point of $p + W$, and, symmetrically, p is the rightmost point of $q - W$. As in the planar case, we say that such a pair is *matched* (in the x-direction). Our strategy is thus to maintain $O(1)$ data structures, one for each pair $W, -W$ of cones in the tiling. For each cone W, its data structure maintains (a superset of) the set Π of all matched pairs, and runs a kinetic and dynamic tournament among them, to keep track of the closest pair in Π. The real closest pair is the pair with the smallest distance between its points, among the winners of these $O(1)$ tournaments. In complete analogy with the planar case, in the purely kinetic scenario we can maintain the actual set Π, whereas in the kinetic and dynamic scenario, we need to maintain a slightly larger superset Π^*, which we will shortly define.

Fix a cone W, and assume, as above, that its "symmetry axis" is the positive x-axis. Let $e^{(1)}, \ldots, e^{(d)}$ be vectors orthogonal to the facets of W and pointing into W. For each point $p \in \mathbb{R}^d$,

[8]That is, a cone of the form $\{x \mid \angle(x, u) \leq \alpha\}$, for some vector u and angle α.

define $u_p^{(i)} = p \cdot e^{(i)}$, for $i = 1, \ldots, d$. Clearly, $q - p \in W$ if and only if $u_q^{(i)} > u_p^{(i)}$, for $i = 1, \ldots, d$. We thus apply the same strategy as in the planar case: We construct a d-dimensional range tree T, where the i-th level of the tree stores points according to their $u^{(i)}$-order. At the bottom level of the structure, each node ξ records the points with the smallest and largest x-coordinates that are stored in the subtree rooted at ξ. By querying the structure with a point p, or, more precisely, with the orthant $u^{(i)} > u_p^{(i)}$, for $i = 1, \ldots, d$, we can find the leftmost point $q \in p + W$, in time $O(\log^d n)$. This allows us to construct the set Π of all $O(n)$ matched pairs, in $O(n \log^d n)$ time, and to run a tournament on these pairs, using exactly the same structure as described in Section 3. (Note that this tournament structure is independent of d.)

As in the planar case, we can alternatively use the larger set Π^* of pairs, defined as follows. For each node ξ of the bottom level of T, we form a pair (p, q), where p (resp., q) is the rightmost (resp., leftmost) point that is stored at the left (resp., right) subtree of each of the nodes $\xi^{(1)}, \xi^{(2)}, \ldots, \xi^{(d)} = \xi$ in the d levels of the structure, whose respective trees contain ξ. As in the planar case, it is easy to see that $\Pi \subseteq \Pi^*$. The size of Π^* is $O(n \log^{d-1} n)$, and it can initially be constructed in time $O(n \log^{d-1} n)$, during the construction of T, in a straightforward bottom-up manner. (Thus, in the static and stationary case, we obtain an algorithm for the closest pair that runs in time $O(n \log^{d-1} n)$. There are, however, faster algorithms for this scenario, such as the one of Vaidya [31].) To compute and maintain Π^*, we need to store a more refined information in the nodes of the bottom level of T, which extends, in an obvious manner, the blue/red pointers maintained in the 2-dimensional tree of Section 4.

Each of the sets Π and Π^* remains unchanged as long as the orders of the points of P in each of the coordinates $x, u^{(1)}, \ldots, u^{(d)}$ remain unchanged. Hence, the critical events that the range tree T has to keep track of are the $O(n^2)$ swaps of consecutive points in any one of these orders. In addition, the tournament structure maintains its own set of critical events, exactly as in the planar case.

Consider first the purely kinetic scenario (no insertions or deletions). Here we maintain only the set Π. When a swap between two points p, q takes place, we update T in an analogous manner to that described in Section 4. To make the updates efficient, we maintain each subtree of T in each of the levels $2, \ldots, d$ as a dynamic weight-balanced $BB(\alpha)$ tree. Note that rebalancing rotations may require that the relevant subtrees be completely rebuilt in the deeper levels. Using the properties of $BB(\alpha)$ trees, an update of T takes $O(\log^d n)$ *amortized* time. In the kinetic and dynamic scenario, the first level of T is also maintained as a weight-balanced tree.

In the purely kinetic scenario, we query T after the update with p and q, and find the $O(1)$ new pairs of Π that the swap has generated, as well as the $O(1)$ old pairs that have to be deleted. We then update the tournament structure accordingly. The cost of handling the tournament structure is negligible in this case. We summarize the performance of the structure in the following theorem.

Theorem 8.2. *In the purely kinetic scenario, the KDS for the closest pair in $\mathbb{R}^d$ described above has the following properties.*

1. *The number of events that it processes is $O(n^2 \beta_{s+2}(n) \log n)$ (thus the KDS is* efficient*).*
2. *The (amortized) time it takes to process an event is $O(\log^d n)$ (thus the KDS is* responsive*, in an amortized sense).*

3. *At any time, each point p participates in a constant number of certificates of types (i), (ii), and (iii), and pairs that p belongs to participates in $O(\log n)$ certificates of $\mathcal{Q}$ (thus the KDS is* local).

4. *The structure requires $O(n \log^{d-1} n)$ space (and is thus* compact).

In the kinetic and dynamic scenario, we need to maintain the larger set Π^*, for which we need to maintain the refined information at the bottom-level nodes of T. Here each swap generates an amortized number of $O(\log^d n)$ updates of Π^*, which are then fed into the tournament structure. As in the planar case, naive implementation using an external tournament will result in $O(\log^{d+1} n)$ certificates changing, which will then require $O(\log^{d+2} n)$ time to process (largely consumed by updating the event queue). However, if we embed the tournament into the range tree, we can reduce the (still amortized) number of certificates changing by a swap to $O(\log^d n)$, and their processing cost to $O(\log^{d+1} n)$. The total number of internal critical events that the tournament keeps track of is $O(mn\beta_{s+2}(n) \log^{d+1} n)$, by arguments analogous to the ones given in Section 5. Summarizing, we have:

Theorem 8.3. *The dynamic KDS for the closest pair in $\mathbb{R}^d$, as described above, has the following properties.*

1. *The number of events during a sequence of m insertions and deletions into a KDS of size at most n at any time (assuming $m \geq n$), is $O(mn\beta_{s+2}(n) \log^{d+1} n)$. This makes the KDS* efficient.

2. *The amortized time it takes to process an event is $O(\log^{d+1} n)$ (thus the KDS is* responsive).

3. *Each point participates in $O(\log^d n)$ certificates (thus the KDS is* local).

4. *The KDS requires $O(n \log^{d-1} n)$ space (and is thus* compact).

5. *An insertion or a deletion takes $O(\log^{d+1} n)$ amortized time.*

Finally, we consider the extension of our data structure for all nearest neighbors to higher dimensions. For this we need to construct a d-dimensional dynamic range tree using treaps. The analysis in Sections 6 and 7 extends to higher dimensions in a straightforward (albeit tedious) manner. Omitting all further details, we obtain the following extension of Theorem 7.2.

Theorem 8.4. *Let P be a set of moving points in $\mathbb{R}^d$. to which we also make $m \geq n$ insertions and deletions of points, so that there are at most n points in the set at any fixed time. One can then construct a KDS that maintains all nearest neighbors in P, which also supports these insertions and deletions, and which satisfies the following properties.*

1. *The number of $e^{(i)}$-swaps, for $1 \leq i \leq d$, is $O(mn)$ and processing an $e^{(i)}$-swap takes $O(\log^d n)$ expected time.*

2. *The number of x-swaps is $O(mn)$, processing a single x-swap takes $O(n)$ expected time, and processing all x-swaps takes $O(mn\beta_{s+2}(n) \log^d n)$ expected time.*

3. *The expected number of tournament events is $O(mn\beta_{s+2}^2(n)\log^{d+1} n)$, and their total expected cost is $O(mn\beta_{s+2}^2(n)\log^{d+2} n)$.*

4. *The data structure requires $O(n\log^d n)$ space.*

5. *An insertion or a deletion takes $O(n)$ expected time.*

Acknowledgment

We thank Danny Feldman for pointing out that our technique works in any dimension.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson, Indexing moving points, *J. Comp. Sys. Sci.* 66 (2003), 207–243.

[2] P. K. Agarwal, J. Basch, M. de Berg, L. Guibas, and J. Hershberger, Lower bounds for kinetic planar subdivisions, *Discrete Comput. Geom.* 24 (2000), 721–733.

[3] P. K. Agarwal, M. de Berg, J. Gao, L. Guibas, and S. Har-Peled, Staying in the middle: Exact and approximate medians in R^1 and R^2 for moving points, *Proc. 16th Annu. Canadian Conf. Comput. Geom.*, 2005.

[4] P. K. Agarwal, J. Erickson, and L. Guibas, Kinetic binary space partitions for intersecting segments and disjoint triangles, *Proc. 9th Annu. ACM-SIAM Sympos. Discrete Algo.* 1998, 107–116,

[5] P. K. Agarwal, J. Gao, and L. Guibas, Kinetic medians and *kd*-trees, *Proc. 10th European Sympos. Algo.*, 2002, 5–16.

[6] P. K. Agarwal, L. Guibas, J. Hershberger, and E. Veach, Maintaining the extent of a moving point set, *Discrete Comput. Geom.* 26 (2001), 353–374.

[7] P. K. Agarwal, L. Guibas, T. M. Murali, and J. S. Vitter, Cylindrical static and kinetic binary space partitions. *Comput. Geom. Theory Appls.* 16 (2000), 103–127.

[8] G. Alexandron, H. Kaplan and M. Sharir, Kinetic and dynamic data structures for convex hulls and upper envelopes, *Comput. Geom. Theory Appls.* 36 (2007), 144–158.

[9] J. Basch, J. Erickson, L. Guibas, J. Hershberger, and L. Zhang, Kinetic collision detection between two simple polygons, *Comput. Geom. Theory Appls.* 27 (2004), 211–235.

[10] J. Basch, L. J. Guibas, and J. Hershberger, Data structures for mobile data, *J. Algorithms* 31(1) (1999), 1–28.

[11] J. Bentley and M. Shamos, Divide-and-conquer in higher-dimensional space. *Proc. 8th Annu. ACM Sympos. Theory Comput.*, 1976, 220–230.

[12] M. de Berg, Kinetic dictionaries: how to shoot a moving target, *Proc. 11th European Sympos. Algo.*, 2003, 172–183.

[13] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd Edition, Springer Verlag, Heidelberg, 2000.

[14] S. Bespamyatnikh, An optimal algorithm for closest-pair maintenance, *Discrete Comput. Geom.* 19 (1998), 175–195.

[15] N. Blum and K. Mehlhorn, On the average number of rebalancing operations in weight-balanced trees, *Theoret. Comput. Sci.* 11 (1980), 303–320.

[16] J. Gao, L. Guibas, J. Hershberger, L. Zhang, and A. Zhu, Discrete mobile centers, *Discrete Comput. Geom.* 30 (2003), 45–63.

[17] L. Guibas, Modeling motion, in *Handbook of Discrete and Computational Geometry,* 2nd Ed., (J. Goodman and J. O'Rourke, eds.), Chapman and Hall/CRC, 2004, 1117–1134.

[18] L. Guibas, J. Hershberger, S. Suri, and L. Zhang, Kinetic connectivity for unit disks. *Discrete Comput. Geom.* 25 (2001), 591–610.

[19] J. Hershberger, Kinetic collision detection with fast flight plan changes, *Inform. Process. Lett.* 92 (2004), 287–291.

[20] J. Hershberger and S. Suri, Simplified kinetic connectivity for rectangles and hypercubes, *Proc. 12th Annu. ACM-SIAM Sympos. Discrete Algo.*, 2001, 158–167.

[21] M. I. Karavelas and L. Guibas, Static and kinetic geometric spanners with applications, *Proc. 12th Annu. ACM-SIAM Sympos. Discrete Algo.*, 2001, 168–176.

[22] D. Kirkpatrick, J. Snoeyink, and B. Speckmann, Kinetic collision detection for simple polygons, *Internat. J. Comput. Geom. Appls.* 12 (2002), 3–27.

[23] D. Kirkpatrick and B. Speckmann, Separation sensitive kinetic separation structures for convex polygons, *Proc. Japanese Sympos. Discrete Comput. Geom.*, 2001, 222–236,

[24] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer Verlag, Berlin 1984.

[25] J. Nievergelt and E. M. Reingold, Binary search trees of bounded balance, *SIAM J. Comput.* 2 (1973), 33–43.

[26] C. Schwarz, M. Smid, and J. Snoeyink, An optimal algorithm for the on-line closest-pair problem, *Algorithmica* 12 (1994), 18–29.

[27] R. Seidel and C. R. Aragon, Randomized search trees, *Algorithmica* 16 (1996), 464–497.

[28] M.I. Shamos and D. Hoey, Closest-point problems, *Proc. 16th IEEE Sympos. Foundat. Comp. Sci.*, 1975, 151–162.

[29] M. Sharir and P.K. Agarwal, *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, New York, 1995.

[30] M. Smid, Maintaining the minimal distance of a point set in polylogarithmic time, *Discrete Comput. Geom.* 7 (1992), 415–431.

[31] P.M. Vaidya, An $O(n \log n)$ algorithm for the all nearest neighbor problem, *Discrete Comput. Geom.* 4 (1989), 101–115.