Kinetic Reverse k-Nearest Neighbor Problem *

Zahed Rahmati† Valerie King‡ Sue Whitesides§

Abstract

This paper provides the first solution to the kinetic reverse k-nearest neighbor (RkNN) problem in $\mathbb{R}^d$, which is defined as follows: Given a set P of n moving points in arbitrary but fixed dimension d, an integer k, and a query point $q \notin P$ at any time t, report all the points $p \in P$ for which q is one of the k-nearest neighbors of p.

1 Introduction

The *reverse k-nearest neighbor* (RkNN) problem is a popular variant of the k-nearest neighbor (kNN) problem and asks for the influence of a query point on a point set. Unlike the kNN problem, the exact number of reverse k-nearest neighbors of a query point is not known in advance. The RkNN problem is formally defined as follows: Given a set P of n points in $\mathbb{R}^d$, an integer k, $1 \leq k \leq n-1$, and a query point $q \notin P$, find the set RkNN(q) of all p in P for which q is one of k-nearest neighbors of p. Thus RkNN$(q) = \{p \in P : \ |pq| \leq |pp_k|\}$, where $|.|$ denotes Euclidean distance, and p_k is the k^{th} nearest neighbor of p among the points in P. The *kinetic RkNN* problem is to answer RkNN queries on a set P of moving points, where the trajectory of each point $p \in P$ is a function of time. Here, we assume the trajectories are polynomial functions of maximum degree bounded by some constant s.

Related work. The reverse k-nearest neighbor problem was first posed by Korn and Muthukrishnan [14] in the database community, and then considered extensively in this community due to its many applications, *e.g.*, decision support systems, profile-based marketing, traffic networks, business location planning, clustering and outlier detection, and molecular biology [14, 15, 16]. The reverse k-nearest neighbor queries for a set of continuously moving objects has also attracted the attention of the database community; see [9] and references therein. Examples of moving objects

*This work was partially supported by a British Columbia Graduate Student Fellowship and by NSERC discovery grants.

†Department of Computer Science, University of Victoria, Canada, `rahmati@uvic.ca`

‡Department of Computer Science, University of Victoria, Canada, `val@uvic.ca`

§Department of Computer Science, University of Victoria, Canada `sue@uvic.ca`

include players in multi-player game environments, soldiers in a battlefield, tourists in dangerous environments, and mobile devices in wireless ad-hoc networks.

To our knowledge, in computational geometry, there exist two data structures [17, 10] that give solutions to the RkNN problem. Both of these solutions answer RkNN queries for a set P of stationary points and both only work for $k = 1$. Maheshwari *et al.* (2002) [17] gave a data structure to solve the R1NN problem in $\mathbb{R}^2$. Their data structure, which supports insertions and deletions of points, creates an arrangement of largest empty circles centered at the points of P and answers R1NN queries by point location in the arrangement. Their data structure uses $O(n)$ space and $O(n \log n)$ preprocessing time, and an R1NN query can be answered in time $O(\log n)$. Cheong *et al.* (2011) [10] considered the R1NN problem in fixed dimension $\mathbb{R}^d$, where $d = O(1)$. Their method, which uses a compressed quadtree, partitions space into cells such that each cell contains a small number of candidate points. To answer an R1NN query, their solution finds a cell that contains the query point and then checks all the points in the cell. Their approach uses $O(n)$ space and $O(n \log n)$ preprocessing time, and can answer an R1NN query in $O(\log n)$ time; it seems that the approach by Cheong *et al.* can be extended to answer RkNN queries with preprocessing time $O(kn \log n)$, space $O(kn)$, and query time $O(\log n + k)$.

For a set P of n stationary points, one can report all the 1-nearest neighbors in time $O(n \log n)$ [20], and all the k-nearest neighbors, for any $k \geq 1$, in time $O(kn \log n)$ [13], where the neighbors are reported in order of increasing distance from each point; reporting the unordered set takes time $O(n \log n + kn)$ [6, 11, 13].

For a set of moving points, there are two kinetic data structures [2, 19] to maintain all the k-nearest neighbors, but they only work for $k = 1$.

Our contribution. We provide the *first* solution to the kinetic RkNN problem for *any* $k \geq 1$ in *any* fixed dimension d. To answer an RkNN query for a query point $q \notin P$ at any time t, we partition the d-dimensional space into a constant number of cones around q, and then among the points of P in each cone, we examine the k points having shortest projections on the cone axis. We obtain $O(k)$ candidate points for q such that q might be one of their k-nearest neighbors at time t. To check which if any of these candidate points is a reverse k-nearest neighbor of q, we maintain the k^{th} nearest neighbor p_k of each point $p \in P$ over time. By checking whether $|pq| \leq |pp_k|$ we can easily check whether a candidate point p is one of the reverse k-nearest neighbors of q at time t.

For a set P of n continuously moving points in $\mathbb{R}^d$, where the trajectory of each point is a polynomial function of at most constant degree s, we provide a simple kinetic approach to answer RkNN queries on the moving points. In the preprocessing step, we introduce a method for reporting all the k-nearest neighbors for all the points $p \in P$ in order of increasing distance from p. For $k = \Omega(\log^{d-1} n)$, both our method and the method of Dickerson and Eppstein [13] give the same complexity, but in our view, our method is simpler in practice.

In order to answer RkNN queries, our kinetic approach maintains all the k-nearest neighbors over time. This is the first KDS for maintenance of all the k-nearest neighbors in $\mathbb{R}^d$, for any $k \geq 1$. Our KDS uses $O(n \log^d n + kn)$ space and $O(n \log^d n + kn \log n)$ preprocessing time, and processes $O(\phi(s,n) * n^2)$ events, each in amortized time $O(\log n)$. Here, $\phi(s,n)$ is the complexity of the k-level of a set of n partially-defined polynomial functions, such that each pair of them intersects at most s times. The current bounds on $\phi(s,n)$ are as follows.

$$\phi(s,n) = \begin{cases} O(n^{3/2}\log n), & \text{for } s = 2 \text{ [8];} \\ O(n^{5/3}\text{poly}\log n), & \text{for } s = 3 \text{ [7];} \\ O(n^{31/18}\text{poly}\log n), & \text{for } s = 4 \text{ [7];} \\ O(n^{161/90-\delta}), & \text{for } s = 5, \text{ for some constant } \delta > 0 \text{ [8];} \\ O(n^{2-1/2s}), & \text{for odd } s \text{ [7];} \\ O(n^{2-1/2(s-1)}), & \text{for even } s \text{ [7].} \end{cases}$$

At any time t, an RkNN query can be answered in time $O(\log^d n + k \log\log n)$. Note that if an event occurs at the same time t, we first spend amortized time $O(\log n)$ to update all the k-nearest neighbors, and then we answer the query.

Outline. Section 2 provides two key lemmas, and in fact introduces a new supergraph, namely the *k-Semi-Yao graph*, of the k-nearest neighbor graph. In Section 3, we show how to report all the k-nearest neighbors. Section 4 gives a (kinetic) data structure for answering RkNN queries on moving points, where the trajectory of each point is a bounded-degree polynomial. Also included in this section is an analysis of our kinetic data structure in terms of the kinetic data structure performance criteria. Section 5 concludes.

2 Key Lemmas

Partition the plane around the origin o into six wedges, $W_0, ..., W_5$, each of angle $\pi/3$ (see Figure 1(a)). Denote by $W_l(p)$ the translation of wedge W_l, $0 \leq l \leq 5$, such that its apex moves from o to point p (see Figure 1(b)). Denote by x_l (resp. $x_l(p)$) the vector along the bisector of W_l (resp. $W_l(p)$) directed outward from the apex at o (resp. p). Denote the reflection of $W_l(p)$ through p by $W_{l'}(p)$. Note that $l' = (l+3) \bmod 6$; see Figure 1(b).

Consider the i^{th} nearest neighbor p_i of p. Denote by $L(P \cap W_l(p_i))$ the list of the points in $P \cap W_l(p_i)$, sorted by increasing order of their x_l-coordinates (projections). The following lemma provides a key insight.

Lemma 1 *Let p_i be the i^{th} nearest neighbor of p among a set P of points in $\mathbb{R}^2$, and let $W_l(p_i)$ be the wedge of p_i that contains p. Then point p is among the first i points in $L(P \cap W_l(p_i))$.*

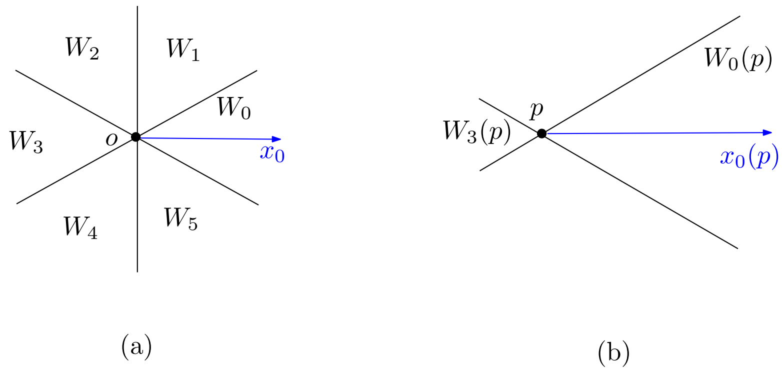


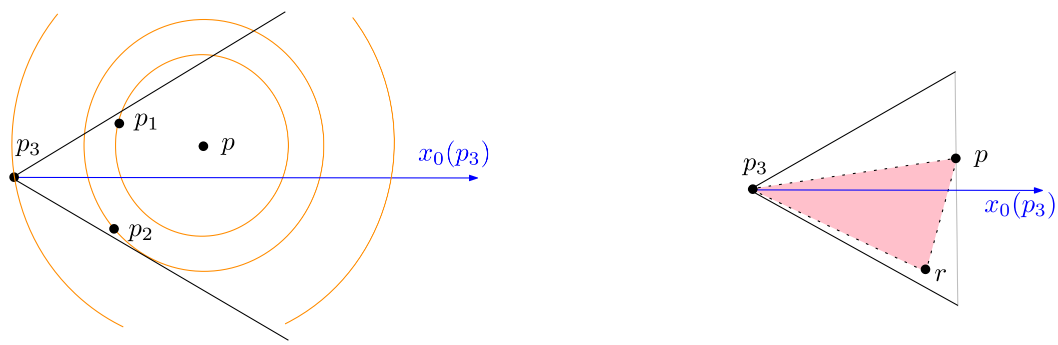
Figure 1: (a) A Partition of the plane into six wedges with common apex at o. (b) A translation of W_0 that moves apex to p. The wedge $W_0(p)$ is the reflection through p of $W_3(p)$ and vise-versa.

Proof. Let $P' = P \backslash \{p_1, ..., p_{i-1}\}$. Then the point p_i is the closest point to p among the points in P'; see Figure 2(a) below. We now prove by contradiction that the point p has the minimum x_l-coordinate among the points in $P' \cap W_l(p_i)$: Assume there is a point $r \in P$ inside the wedge $W_l(p_i)$ whose x_l-coordinate is less than the x_l-coordinate of p; see Figure 2(b) for an example where $i = 3$. Consider the triangle pp_ir. Since p_i is the closest point to p among the points in P', $|pp_i| < |pr|$ which implies that the angle $\angle pp_ir > \angle prp_i$. This is a contradiction, because $\angle pp_ir \leq \pi/3$ and $\angle prp_i > \pi/3$.

Now we add the points $p_1, ..., p_{i-2}$, and p_{i-1} to the point set P'. Consider the worst case scenario that all these $i-1$ points insert inside the wedge $W_l(p_i)$, and that the x_l-coordinates of all these points are less than the x_l-coordinate of p. Then the point p is still among the first i points in the sorted list $L(P \cap W_l(p_i))$. □

The *k-nearest neighbor graph* (k-NNG) of a point set P is constructed by connecting each point in P to all its k-nearest neighbors. If we connect each point $p \in P$ to the first k points in the sorted list $L(P \cap W_l(p))$, for $l = 0, ..., 5$, we obtain what we call the *k-Semi-Yao graph* (k-SYG). Lemma 1 gives a necessary condition for p_i to be the i^{th} nearest neighbor of p: the point p is among the first i points in $L(P \cap W_l(p_i))$, where l is such that $p \in W_l(p_i)$. Therefore, the edge set of the k-SYG covers the edges of the k-NNG. In summary, we have the following.

Lemma 2 *The k-NNG of a set P of points in* $\mathbb{R}^2$ *is a subgraph of the k-SYG of the set P.*



(a) (b)

Figure 2: Point p_3 is the 3rd nearest neighbor of p. After deleting the points p_1 and p_2, point p_3 is the closest point to p; among the points in $W_0(p_3)$, p has the minimum length projection on the bisector $x_0(p_3)$.

3 Reporting All k-Nearest Neighbors

Here we give a simple method for reporting all the k-nearest neighbors via a construction of the k-SYG.

Let C be a *right circular cone* in $\mathbb{R}^d$ with opening angle θ with respect to some given unit vector v. Thus C is the set of points $x \in \mathbb{R}^d$ such that the angle between $\overrightarrow{ox}$ and $\overrightarrow{v}$ is at most $\theta/2$. The angle between any two rays inside C emanating from the apex o is at most θ. From now on, we assume $\theta = \pi/3$.

Now consider a *polyhedral cone* inscribed in the right circular cone C where the polyhedral cone is formed by the intersection of d distinct half-spaces, bounded by $f_1, ..., f_d$, passing through the apex of C. Assuming d is arbitrary but fixed, the d-dimensional space around the origin o can be tiled by a constant number of polyhedral cones $W_0, ..., W_{c-1}$ [1, 2]. Denote by C_l the associated right circular cone of the polyhedral cone W_l. Let x_l be the vector in the direction of the symmetry of C_l. Denote by $W_l(p)$ the translation of the wedge (polyhedral cone) W_l where o moves to p.

A similar approach and analysis as that in Section 2 can be easily used to state (key) Lemmas 1 and 2 for a set of points in $\mathbb{R}^d$.

To construct the k-SYG efficiently, we need a data structure to perform the following operation efficiently: For each $p \in P$ and any of its wedges $W_l(p)$, $0 \leq l \leq c-1$, find the first k points in $L(P \cap W_l(p))$. Such an operation can be performed by using *range tree* data structures. For each wedge W_l with apex at origin o, we construct an associated d-dimensional range tree $\mathcal{T}_l$ as follows.

Consider a particular wedge W_l with apex at o. The wedge W_l is the intersection of d half-spaces $f_1^+, ..., f_d^+$ bounded by $f_1, ..., f_d$ (see Figure 3). Let $\hat{u}_j$ denote the

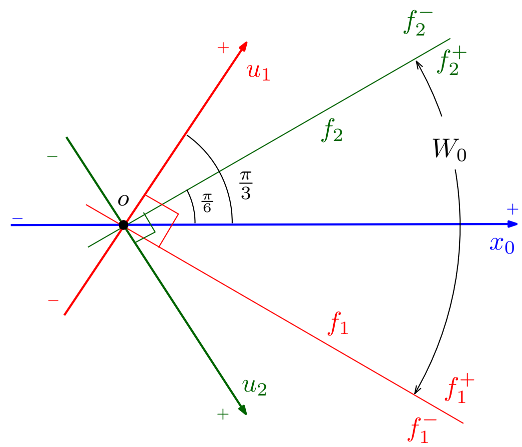


Figure 3: The wedge W_0 in $\mathbb{R}^2$ is bounded by f_1 and f_2. The coordinate axes u_1 and u_2 are orthogonal to f_1 and f_2.

normal to f_j pointing to f_j^+. We define d coordinate axes u_j, $j = 1, ..., d$, through $\hat{u_j}$, where $\hat{u_j}$ gives the respective directions of increasing u_j-coordinate values.

The range tree $\mathcal{T}_l$ is a regular d-dimensional range tree based on the u_j-coordinates, $j = 1, ..., d$. The points at level j are sorted at the leaves according to their u_j-coordinates (for more details about range trees, see Chapter 5 of [5]). From Theorem 5.8 in [5], any d-dimensional range tree, *e.g.*, $\mathcal{T}_l$, uses $O(n \log^{d-1} n)$ space and can be constructed in time $O(n \log^{d-1} n)$; for any point $r \in \mathbb{R}^d$, the points of P inside the query wedge $W_l(r)$ whose sides are parallel to f_j, $j = 1, ..., d$, can be reported in time $O(\log^d n + z)$, where z is the cardinality of the set $P \cap W_l(r)$. In particular, in time $O(\log^d n)$ one can determine a set of $O(\log^d n)$ internal nodes v at level d of $\mathcal{T}_l$, such that $P \cap W_l(r) = \bigcup_v P(v)$, where $P(v)$ is the set of points at the leaves of subtree rooted at v.

Now we add a new level to $\mathcal{T}_l$, based on the coordinate x_l. Let $\mathcal{C}_l(p)$ be the set of the first k points in $L(P \cap W_l(p))$. To find $\mathcal{C}_l(p)$ in an efficient time, we use the level $d+1$ of $\mathcal{T}_l$, which is constructed as follows: For each internal node v at level d of $\mathcal{T}_l$, we create a list $L(P(v))$ sorted by increasing order of x_l-coordinates of the points in $P(v)$. For the set P of n points in $\mathbb{R}^d$, the range tree $\mathcal{T}_l$, which now is a $(d+1)$-dimensional range tree, uses $O(n \log^d n)$ space and can be constructed in time $O(n \log^d n)$.

The following lemma establishes the processing time for obtaining a $\mathcal{C}_l(p)$.

Lemma 3 *Given $\mathcal{T}_l$, the set $\mathcal{C}_l(p)$ can be found in time $O(\log^d n + k \log\log n)$.*

Proof. The proof is by construction. Recall that the set $P \cap W_l(p)$ is the union of $O(\log^d n)$ sets $P(v)$, where v ranges over internal nodes at level d of $\mathcal{T}_l$. Consider the associated sorted lists $L(P(v))$.

We construct a priority queue on the first elements of these $O(\log^d n)$ sorted lists $L(P(v))$ in time $O(\log^d n)$.

By repeating the following two steps k times we can find $\mathcal{C}_l(p)$:

- Delete the element $\hat{p}$ with highest priority from the priority queue, and
- insert the next element into the priority queue from the sorted list $L(P(v_j))$, where v_j is such that $\hat{p} \in P(v_j)$.

Since d is fixed and the size of the priority queue is $O(\log^d n)$, all together these k iterations take $O(k \log\log n)$ time. □

By Lemma 3, we can find all the $\mathcal{C}_l(p)$, for all the points $p \in P$. This gives the following lemma.

Lemma 4 *Using a data structure of size $O(n \log^d n)$, the edges of the k-SYG of a set of n points in fixed dimension d can be reported in time $O(n \log^d n + kn \log\log n)$.*

Next, suppose we are given the k-SYG and we want to report all the k-nearest neighbors. Let E_p be the set of edges incident to the point p in the k-SYG. By sorting these edges in non-decreasing order according to their Euclidean lengths, which can be done in time $O(|E_p| \log |E_p|)$, we can find the k-nearest neighbors of p ordered by increasing distance from p. Since the number of edges in the k-SYG is $O(kn)$ and each edge pp' belongs to exactly two sets E_p and $E_{p'}$, the time to find all the k-nearest neighbors, for all the points $p \in P$, is $\sum_p O(|E_p| \log |E_p|) = O(kn \log n)$.

From the above discussion and Lemmas 2 and 4, the following results.

Theorem 1 *For a set of n points in fixed dimension d, our data structure can report all the k-nearest neighbors, in order of increasing distance from each point, in time $O(n \log^d n + kn \log n)$. The data structure uses $O(n \log^d n + kn)$ space.*

4 RkNN Queries on Moving Points

We are given a set P of n continuously moving points, where the trajectory of each point in P is a polynomial function of bounded degree s. To answer RkNN queries on the moving points, we must keep a valid range tree and track all the k-nearest neighbors during the motion. This section first shows how to maintain a (ranked-based) range tree, and then provides a KDS for maintenance of the k-SYG, which in fact gives a supergraph of the k-NNG over time. Using the kinetic k-SYG, we can easily maintain all the k-nearest neighbors over time. Finally we show how to answer RkNN queries on the moving points.

Kinetic RBRT. Let u_j, $1 \leq j \leq d$, be the coordinate axis orthogonal to the half-space f_j of the wedge W_l, $0 \leq l \leq c-1$ (see Figure 3). Abam and de Berg [1] introduced a variant of the range tree, namely the *ranked-based range tree* (RBRT), which has the following properties. Denote by $\mathcal{T}_l$ the RBRT corresponding to the wedge W_l.

- $\mathcal{T}_l$ can be described as a set of pairs $\Psi_l = \{(B_1, R_1), ..., (B_m, R_m)\}$ such that:
  - For any two points p and q in P where $q \in W_l(p)$, there is a unique pair $(B_i, R_i) \in \Psi_l$ such that $p \in B_i$ and $q \in R_i$.
  - For any pair $(B_i, R_i) \in \Psi_l$, if $p \in B_i$ and $q \in R_i$, then $q \in W_l(p)$ and $p \in W_{l'}(q)$; here $W_{l'}(q)$ is the reflection of $W_l(q)$ through q.

  The Ψ_l is called a *cone separated pair decomposition* (CSPD) for P with respect to W_l. Each pair (B_i, R_i) is generated from an internal node v at level d of the RBRT $\mathcal{T}_l$.

- Each point $p \in P$ is in $O(\log^d n)$ pairs of (B_i, R_i), which means that the number of elements of all the pairs (R_i, B_i) is $O(n \log^d n)$.

- For any point $p \in P$, all the sets B_i (resp. R_i) where $p \in B_i$ (resp. $p \in R_i$) can be found in time $O(\log^d n)$.

- The set $P \cap W_l(p)$ is the union of $O(\log^d n)$ sets R_i, where $p \in B_i$.

- When the points are moving, $\mathcal{T}_l$ remains unchanged as long as the order of the points along axes u_j, $1 \leq j \leq d$, remains unchanged.

- When a u-swap event occurs, meaning that two points exchange their u_j-order, the RBRT $\mathcal{T}_l$ can be updated in worst-case time $O(\log^d n)$ without rebalancing operations.

4.1 Kinetic k-SYG

Here we give a KDS for the k-SYG, for any $k \geq 1$, extending [18].

To maintain the k-SYG, we must track the set $\mathcal{C}_l(p)$ for each point $p \in P$. So, for each $1 \leq i \leq m$, we need to maintain a sorted list $L(R_i)$ of the points in R_i in ascending order according to their x_l-coordinates over time. Note that each set R_i is some $P(v)$, the set of points at the leaves of the subtree rooted at some internal node v at level d of $\mathcal{T}_l$. To maintain these sorted lists $L(R_i)$, we add a new level to the RBRT $\mathcal{T}_l$; the points at the new level are sorted at the leaves in ascending order according to their x_l-coordinates. Therefore, in the modified RBRT $\mathcal{T}_l$, in addition to the u-swap events, we handle new events, called *x-swap events*, when two points exchange their x_l-order. The modified RBRT $\mathcal{T}_l$ behaves like a $(d+1)$-dimensional RBRT. From the last property of an RBRT above, when a u-swap event or an x-swap event occurs, the RBRT $\mathcal{T}_l$ can be updated in worst-case time $O(\log^{d+1} n)$.

Denote by $\ddot{p}_{l,k}$ the k^{th} point in $L(P \cap W_l(p))$. To track the sets $\mathcal{C}_l(p)$, for all the points $p \in P$, we need to maintain the following over time.

- A set of $d+1$ *kinetic sorted lists* $L_j(P)$, $j = 1, ..., d$, and the $L_l(P)$ of the point set P. We use these kinetic sorted lists to track the order of the points in the coordinates u_j and x_l, respectively.

- For each B_i, a sorted list $L(B'_i)$ of the points in B'_i, where $B'_i = \{(p, \ddot{p}_{l,k})|\ p \in B_i\}$. The order of the points in $L(B'_i)$ is according to a *label* of the second points $\ddot{p}_{l,k}$. This sorted list $L(B'_i)$ is used to answer the following query efficiently: Given a query point q and a B_i, find all the points $p \in B_i$ such that $\ddot{p}_{l,k} = q$.

- The k^{th} point $r_{i,k}$ in the sorted list $L(R_i)$. We track the values $r_{i,k}$ in order to make necessary changes to the k-SYG when an x-swap event occurs.

Handling u-swap events. W.l.o.g., let $q \in W_l(p)$ before the event. When a u-swap event between p and q occurs, the point q moves outside the wedge $W_l(p)$; after the event, $q \notin W_l(p)$. Note that the changes that occur in the k-SYG are the deletions and insertions of the edges incident to p inside the wedge $W_l(p)$.

Whenever two points p and q exchange their u_j-order, we do the following updates.

- We update the kinetic sorted list $L_j(P)$. Each swap event in a kinetic sorted list can be handled in time $O(\log n)$.

- We update the RBRT $\mathcal{T}_l$ and if a point is deleted or inserted into a B_i, we update the sorted list $L(B'_i)$. Since each insertion/deletion to $L(B'_i)$ takes $O(\log n)$ time, and since each point is in $O(\log^d n)$ sets B_i, this takes $O(\log^{d+1} n)$ time.

- We update the values of $r_{i,k}$. After updating the RBRT $\mathcal{T}_l$, point q might be inserted or deleted from some R_i and change the values of $r_{i,k}$. So, for all R_i where $q \in R_i$, before and after the event, we do the following. We check whether the x_l-coordinate of q is less than or equal to the x_l-coordinate of $r_{i,k}$; if so, we take the successor or predecessor point of $r_{i,k}$ in $L(R_i)$ as the new value for $r_{i,k}$. This takes $O(\log^{d+1} n)$ time.

- We query to find $\mathcal{C}(p)$. By Lemma 3, this takes $O(\log^d n + k \log\log n)$ time.

- If we get a new value for $\ddot{p}_{l,k}$, we update all the sorted lists $L(B'_i)$ such that $p \in B_i$. This takes $O(\log^{d+1} n)$ time.

Considering the complexity of each step above, and assuming the trajectory of each point is a bounded degree polynomial, the following results.

Lemma 5 *Our KDS for maintenance of the k-SYG handles $O(n^2)$ u-swap events, each in worst-case time $O(\log^{d+1} n + k \log\log n)$.*

Handling x-swap events. When an x-swap event between two consecutive points p and q with p preceding q occurs, it does not change the elements of the pairs (B_i, R_i) of the CSPD Ψ_l. Such an event changes the k-SYG if both p and q are in the same $W_l(w)$, for some $w \in P$, and $w_{l,k} = p$.

We apply the following updates to our KDS when two points p and q exchange their x_l-order.

1. We update the kinetic sorted list $L_l(P)$; this takes $O(\log n)$ time.
2. We update the RBRT $\mathcal{T}_l$, which takes $O(\log^{d+1} n)$ time.
3. We find all the sets R_i where both p and q belong to R_i and such that $r_{i,k} = p$. Also, we find all the sets R_i where $r_{i,k} = q$. This takes $O(\log^d n)$ time.
4. For each R_i, we extract all the pairs $(w, \ddot{w}_{l,k})$ from the sorted lists $L(B'_i)$ such that $\ddot{w}_{l,k} = p$. Note that each change to the pair $(w, \ddot{w}_{l,k})$ is a change to the k-SYG.
5. For each w, we update all the sorted lists $L(B'_i)$ where $(w, \ddot{w}_{l,k}) \in B'_i$: we replace the previous value of $\ddot{w}_{l,k}$, which is p, by the new value q.

Denote by χ_k the number of exact changes to the k-SYG of a set of moving points over time. For each found R_i, the fourth step takes $O(\log n + \xi_i)$ time, where ξ_i is the number of pairs $(w, \ddot{w}_{l,k})$ such that $\ddot{w}_{l,k} = p$. For all these $O(\log^d n)$ sets R_i, this step takes $O(\log^{d+1} n + \sum_i \xi_i)$ time, where $\sum_i \xi_i$ is the number of exact changes to the k-SYG when an x-swap event occurs. Therefore, for all the $O(n^2)$ x-swap events, the total processing time for this step is $O(n^2 \log^{d+1} n + \chi_k)$.

The processing time for the fifth step is a function of χ_k. For each change to the k-SYG, this step spends $O(\log^{d+1} n)$ time to update the sorted lists $L(B'_i)$. Therefore, the total processing time for all the x-swap events in this step is $O(\chi_k * \log^{d+1} n)$.

From the above discussion and an upper bound for χ_k in Lemma 6, Lemma 7 results.

Lemma 6 *The number of changes to the k-SYG of a set of n moving points, where the trajectory of each point is a polynomial function of at most constant degree s, is $\chi_k = O(\phi(s, n) * n)$.*

Proof. Fix a point $p \in P$ and one of its wedges $W_l(p)$. There are $O(n)$ insertions/deletions into the wedge $W_l(p)$ over time. The x_l-coordinates of these points create $O(n)$ partial functions.

The k-SYG changes if a change to $\ddot{p}_{l,k}$ occurs. The number of all changes to $\ddot{p}_{l,k}$ is equal to $\phi(s,n)$, the complexity of the k-level of partially-defined polynomial functions of bounded degree s.

Therefore, considering all the $n = |P|$ points, the number of changes to the k-SYG is within a linear factor of $\phi(s,n)$: $\chi_k = O(\phi(s,n) * n)$. $\square$

Lemma 7 *Our KDS for maintenance of the k-SYG handles $O(n^2)$ x-swap events with a total cost of $O(\phi(s,n) * n \log^{d+1} n)$.*

From Lemmas 5 and 7, the following theorem results.

Theorem 2 *For a set of n moving points in $\mathbb{R}^d$, where the trajectory of each point is a polynomial function of at most constant degree s, our k-SYG KDS uses $O(n \log^d n)$ space and handles $O(n^2)$ events with a total cost of $O(kn^2 \log\log n + \phi(s,n) * n \log^{d+1} n)$.*

4.2 Kinetic All k-Nearest Neighbors

Given a KDS for maintenance of the k-SYG (from Theorem 2), a supergraph of the k-NNG, this section shows how to maintain all the k-nearest neighbors over time. For maintenance of the k-nearest neighbors of each point $p \in P$, we only need to track the order of the edges incident to p in the k-SYG according to their Euclidean lengths. This can easily be done by using a kinetic sorted list. The following theorem summarizes the complexity of our kinetic approach.

Theorem 3 *For a set of n moving points in $\mathbb{R}^d$, where the trajectory of each point is a polynomial of at most constant degree s, our KDS for maintenance of all the k-nearest neighbors, ordered by distance from each point, uses $O(n \log^d n + kn)$ space and $O(n \log^d n + kn \log n)$ preprocessing time. Our KDS handles $O(\phi(s,n) * n^2)$ events, each in $O(\log n)$ amortized time.*

Proof. Let $E_p(t)$ be the set of edges incident to point $p \in P$ in the k-SYG at time t. Let $L(E_p(t))$ denote a kinetic sorted list that maintains the edges in $E_p(t)$ sorted by their Euclidean lengths.

Let m_p be the number of insertions/deletions to the set $E_p(t)$ over time. Since the cardinality of $E_p(t)$ is $O(n)$, each insertion into a kinetic sorted list $L(E_p(t))$ can cause $O(n)$ swaps. Each change, *e.g.*, inserting/deleting an edge pq, to the k-SYG creates two insertions/deletions in the kinetic sorted lists $L(E_p(t))$ and $L(E_q(t))$; this implies that $\sum_p m_p = O(\chi_k)$. By Lemma 6, the kinetic sorted lists handle a total of $O(n \sum_p m_p) = O(\phi(s,n) * n^2)$ events. Each event in a kinetic sorted list is handled in time $O(\log n)$. Thus from this and Theorem 2, the total processing time for swap events is $O(kn^2 \log\log n + \phi(s,n) * n \log^{d+1} n + \phi(s,n) * n^2 \log n) = O(\phi(s,n) * n^2 \log n)$. $\square$

KDS performance criteria. The KDS framework [4] measures the performance of a KDS by four standard criteria, which we now apply to our KDS for maintenance of all the k-nearest neighbors in $\mathbb{R}^d$.

- **Efficiency:** This is the ratio of the number of events that a KDS processes to the number of exact changes to the attribute of interest over time. The exact number of changes for maintenance of all the k-nearest neighbors can be computed as follows. Fix a point $p \in P$. The distances of the $n-1$ points of $P \backslash \{p\}$ to p as functions of time create $2s$-intersecting curves, meaning that each pair intersects at most $2s$ times. The number of changes to the i^{th} nearest neighbor p_i of p equals $\Phi(2s, n-1)$, the complexity of the i-level of the $n-1$ $2s$-intersecting curves. Thus the number of changes to the k-nearest neighbors $p_1, ..., p_k$ of p is $O(\Phi(2s, n) * k)$. The total for all points $p \in P$ is $O(\Phi(2s, n) * kn)$. Since the number of events in our KDS is $O(\phi(s, n) * n^2)$, the efficiency of our KDS is $O(\frac{n}{k})$.

- **Responsiveness:** This is the cost of updating the KDS when an event occurs. In our KDS each event can be handled in amortized time $O(\log n)$. Thus the responsiveness of our KDS is $O(\log n)$ on average.

- **Locality:** The number of updates to a KDS when a point changes its trajectory gives the locality of the KDS. In our KDS, for each two consecutive elements in each of the kinetic sorted lists $L_j(P)$, $L_l(P)$, and $L(E_p(t))$, we have a boolean function of time, called a *certificate*. Each certificate has a failure time, the time when the two consecutive elements exchange their order. If a point changes its trajectory, we update a constant number of these certificates in the kinetic sorted lists $L_j(P)$ and $L_l(P)$. Since the number of edges in the k-SYG is $O(kn)$, if a point changes its trajectory, the number of updates to the certificates in the kinetic sorted lists $L(E_p(t))$ is $O(k)$ on average. Therefore, the locality of our KDS is $O(k)$ on average.

- **Compactness:** This is the number of certificates in the KDS. Since the number of certificates of the kinetic sorted lists $L_j(P)$ and $L_l(P)$ is $O(n)$, and the number of certificates of the kinetic sorted lists $L(E_p(t))$ is $O(kn)$, the compactness of our KDS is $O(kn)$.

Therefore, we can obtain the following.

Lemma 8 *In terms of the KDS performance criteria, the "efficiency", "responsiveness", "locality", and "compactness" of our KDS are $O(nk)$, $O(\log n)$ on average, $O(k)$ on average, and $O(kn)$, respectively.*

4.3 RkNN Queries

Suppose we are given a query point $q \notin P$ at some time t. To find the reverse k-nearest neighbors of q, we seek the points in $P \cap W_l(q)$ and find $\mathcal{C}_l(q)$, the set of the first k points in $L(P \cap W_l(q))$. The set $\cup_l \mathcal{C}_l(q)$ contains $O(k)$ candidate points for q such that q might be one of their k-nearest neighbors. In time $O(\log^d n)$ we can find a set of R_i where $P \cap W_l(q) = \sum_i R_i$. From Lemma 3, and since we have sorted lists $L(R_i)$ at level $d+1$ of $\mathcal{T}_l$, the $O(k)$ candidate points for the query point q can be found in worst-case time $O(\log^d n + k \log\log n)$. Now we check whether these candidate points are the reverse k-nearest neighbors of the query point q at time t or not; this can be easily done by application of Theorem 3, which in fact maintain the k^{th} nearest neighbor p_k of each $p \in P$. Therefore, checking a candidate point can be done in $O(1)$ time by comparing distance $|pq|$ to distance $|pp_k|$. This implies that checking which elements of $\mathcal{C}_l(q)$, for $l = 0, ..., c-1$, are reverse k-nearest neighbors of the query point q takes time $O(k)$.

If a query arrives at a time t that is simultaneous with the time when one of the $O(\phi(s,n) * n^2)$ events occurs, our KDS first spends time $O(\log n)$ in an amortized sense to handle the event, and then spends time $O(\log^d n + k \log\log n)$ to answer the query. Thus we have the following.

Theorem 4 *Consider a set P of n moving points in $\mathbb{R}^d$, where the trajectory of each one is a bounded-degree polynomial. The number of reverse k-nearest neighbors for a query point $q \notin P$ is $O(k)$. Our (kinetic) data structure uses $O(n \log^d n + kn)$ space and $O(n \log^d n + kn \log n)$ preprocessing time. At any time t, an RkNN query can be answered in time $O(\log^d n + k \log\log n)$. If an event occurs at time t, the KDS spends $O(\log n)$ time in an amortized sense on updating itself.*

5 Discussion and Conclusion

In the kinetic setting, where the trajectories of the points are polynomials of bounded degree, to answer the RkNN queries over time we have provided a KDS for maintenance of all the k-nearest neighbors. Our KDS is the first KDS for maintenance of all the k-nearest neighbors in $\mathbb{R}^d$, for any $k \geq 1$. It processes $O(\phi(s,n) * n^2)$ events, each in time $O(\log n)$ in an amortized sense. An open problem is to design a KDS that processes less than $O(\phi(s,n) * n^2)$ events.

Arya *et al.* [3] have a kd-tree implementation to approximate the nearest neighbors of a query point that is in use by practitioners [12] who have found challenging to implement the theoretical algorithms [6, 11, 13, 20]. Since to report all the k-nearest neighbors ordered by distance from each point our method uses multidimensional range trees, which can be easily implemented, we believe our method may be useful in practice.

Acknowledgments. We thank Timothy Chan for his remarks on the best current bounds on the complexity of the k-level of partially-defined bounded-degree polynomials.

References

[1] Abam, M.A., de Berg, M.: Kinetic spanners in $\mathbb{R}^d$. Discrete & Computational Geometry 45(4), 723–736 (2011)

[2] Agarwal, P.K., Kaplan, H., Sharir, M.: Kinetic and dynamic data structures for closest pair and all nearest neighbors. ACM Transactions on Algorithms 5, 4:1–37 (2008)

[3] Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. Journal of the ACM 45(6), 891–923 (1998)

[4] Basch, J., Guibas, L.J., Hershberger, J.: Data structures for mobile data. Journal of Algorithms 31, 1–19 (1999)

[5] Berg, M.d., Cheong, O., Kreveld, M.v., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd edn. (2008)

[6] Callahan, P.: Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In: Proceedings of the 34th Annual Symposium on Foundations of Computer Science (FOCS '93). pp. 332–340. IEEE Computer Society, Washington, DC, USA (1993)

[7] Chan, T.M.: On levels in arrangements of curves, ii: A simple inequality and its consequences. Discrete & Computational Geometry 34(1), 11–24 (2005)

[8] Chan, T.M.: On levels in arrangements of curves, iii: further improvements. In: Proceedings of the 24th annual Symposium on Computational Geometry (SoCG '08). pp. 85–93. ACM, New York, NY, USA (2008)

[9] Cheema, M.A., Zhang, W., Lin, X., Zhang, Y., Li, X.: Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. The VLDB Journal 21(1), 69–95 (2012)

[10] Cheong, O., Vigneron, A., Yon, J.: Reverse nearest neighbor queries in fixed dimension. International Journal of Computational Geometry and Applications 21(02), 179–188 (2011)

[11] Clarkson, K.L.: Fast algorithms for the all nearest neighbors problem. In: Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS '83). pp. 226–232. IEEE Computer Society, Washington, DC, USA (1983)

[12] Connor, M., Kumar, P.: Fast construction of k-nearest neighbor graphs for point clouds. IEEE Transactions on Visualization and Computer Graphics 16(4), 599–608 (2010)

[13] Dickerson, M.T., Eppstein, D.: Algorithms for proximity problems in higher dimensions. International Journal of Computational Geometry and Applications 5(5), 277–291 (1996)

[14] Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00). pp. 201–212. ACM, New York, NY, USA (2000)

[15] Kumar, Y., Janardan, R., Gupta, P.: Efficient algorithms for reverse proximity query problems. In: Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '08). pp. 39:1–39:10. ACM, New York, NY, USA (2008)

[16] Lin, J., Etter, D., DeBarr, D.: Exact and approximate reverse nearest neighbor search for multimedia data. In: Proceedings of the 2008 SIAM International Conference on Data Mining (SDM '08). pp. 656–667. SIAM (2008)

[17] Maheshwari, A., Vahrenhold, J., Zeh, N.: On reverse nearest neighbor queries. In: Proceedings of the 14th Canadian Conference on Computational Geometry (CCCG '02). pp. 128–132 (2002)

[18] Rahmati, Z., Abam, M.A., King, V., Whitesides, S.: Kinetic data structures for the Semi-Yao graph and all nearest neighbors in $\mathbb{R}^d$. In: Proceedings of the 26th Canadian Conference on Computational Geometry (CCCG '14) (2014)

[19] Rahmati, Z., King, V., Whitesides, S.: Kinetic data structures for all nearest neighbors and closest pair in the plane. In: Proceedings of the 29th Symposium on Computational Geometry (SoCG '13). pp. 137–144. ACM, New York, NY, USA (2013)

[20] Vaidya, P.M.: An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. Discrete & Computational Geometry 4(2), 101–115 (1989)