Further Dynamic Computational Geometry[1]

— Extended Abstract —

Mordecai J. Golin[2] Christian Schwarz[3] Michiel Smid[3]

1 Introduction

Atallah [2] introduced the field of Dynamic Computational Geometry in which he assumed that the inputs to his problems were not static objects, but points which moved over time with what he defined as k-motion. A point p moves with k-motion in d-dimensional space if $p(t) = \sum_{l=0}^{k} \mathbf{C}_l t^l$ where t is a time parameter and the $\mathbf{C}_l$ are constant d-dimensional vectors. Thus, $\mathbf{C}_0$ is the initial position of p at time $t = 0$.

In his paper Atallah studied the dynamics of a few of the basic problems in computational geometry when the input points have k-motion. As an example, he examined the problem of identifying the closest (furthest) pair in a set of n points moving with k-motion. He proved that the closest (furthest) pair changes at most $O(\lambda_{2k}(n^2))$ times as t increases to infinity and that the sequence of these changes can be computed in $O(dk^2n^2 + \lambda_{2k}(n^2)\log n)$ time, where $\lambda_s(n)$ is the maximal size of an (n, s)-Davenport-Schinzel sequence. (The function $\lambda_s(n)$ is slightly superlinear for any constant s.) He also showed that if the points are planar ($d = 2$), then the points on their convex hull change only $O(n\lambda_{4k}(n))$ times and gave efficient algorithms for computing the intervals of time in which a particular point appears on the convex hull. In a more recent article, Fu and Lee [4] have shown how to extend Atallah's ideas in order to compute the Voronoi diagram of planar points under k-motion.

In this paper we examine a number of other problems in computational geometry — such as finding minimum spanning trees, proximity questions and range reporting — under the k-motion model. We provide algorithms for constructing data structures that support *efficient queries*. As an example, one of the data structures that we present supports answering queries of the type "What is the minimum spanning tree of the n points for a specific time t?" in time $O(n)$ (which is obviously optimal). See Table 1.

In the rest of this paper, S is a set of points that are in k-motion. We assume that one memory location is needed to store a real number. Consequentially, evaluating the location of a point at time t requires $O(dk)$ time and storing the input equations requires $O(dkn)$ space. We furthermore follow the practice of [2] and [4] and assume that it is possible to find a real root of an order k polynomial in constant time, and to find all of the (at most k) real roots in $O(k)$ time. The square of the distance of two points, which will be used frequently, can be computed in time $O(dk^2)$. To make the presentation of our results simpler, we assume that d and k are at most n, meaning e.g. that $\log(d \cdot k \cdot n) = O(\log n)$.

2 Nearest neighbor problems

We consider the following query problems:
All nearest/furthest neighbors: Report $\bigcup_{p \in S}$ (nearest/furthest neighbor(p)) at time t.
m-th nearest neighbor: Let p be one of the n moving points. Given a query time t and a number $1 \leq m \leq n-1$, report the m-th nearest neighbor of p in $S \setminus \{p\}$ at time t.
All m-th nearest neighbors: Given a query time t and a number $1 \leq m \leq n-1$, report $\bigcup_{p \in S}$ (m-th nearest neighbor(p)) at time t.

Atallah gave an algorithm for the nearest/furthest neighbor problem. Let p be a fixed point of S. For every other point q in S, let $f_{pq}(t)$ be the polynomial in t of degree $2k$ that expresses the square of the distance between p and q. Consider the lower envelope of these functions, where q varies over all points in $S \setminus \{p\}$. The envelope consists of portions of the functions $f_{pq}(t)$, ordered w.r.t. time. Each portion of the lower envelope corresponds to an interval $[t_1, t_2]$ such that for any $t \in [t_1, t_2]$, the value $\min_q (f_{pq}(t))$ is attained by the same function $f_{pq'}(t)$. I.e., we have a partition of the positive time axis $[0, \infty]$ into intervals of fixed answer (if we

[1]This research was supported by the European Community, Esprit Basic Research Action Number 7141 (ALCOM II).
[2]INRIA Rocquencourt, 78153 Le Chesnay, France, email: golin@margaux.inria.fr. The research of this author was also partially supported by NSF grant CCR-891852.
[3]Max-Planck-Institut für Informatik, W-6600 Saarbrücken, Germany, email: {schwarz,michiel}@mpi-sb.mpg.de.

Problem	*Preprocessing Time*	*Space*	*Query Time*
All Nearest/Furthest Neighbors	$dk^2n^2 + n\lambda_{2k}(n)\log n$	$n\lambda_{2k}(n)$	n
m-Nearest Neighbor	$dk^2n + kn^2\log n$	kn^2	$\log n$
All m-Nearest Neighbors	$dk^2n^2 + kn^3\log n$	kn^3	n
Minimum Spanning Tree	$dk^2n^2 + kn^4\log n$	kn^4	n
Proximity Counting	$dk^2n^2 + kn^4\log n$	kn^4	$\log n$
Reporting	$dk^2n^2 + kn^4\log n$	$\lambda_{2k}(n^2)\log n$	$\log n + A$
Maxima Counting	$dkn^2\log n$	dkn^2	$\log n$
Reporting	$dkn^2\log n$	dkn^2	$\log n + A$
Range Query Counting	$dkn\log n$	dkn	$\log n$
Reporting	$dkn\log n$	dkn	$\log n + A$

Table 1: Summary of this paper's results. The input points have dimension d and move with k-motion; A denotes the size of the answer. $\lambda_k(n)$ denotes the maximum size of an (n,k)-Davenport-Schinzel sequence.

assume that motion starts at time $t = 0$). Since the nearest neighbor of p at time t is the point q' such that $f_{pq'}(t) = \min_{q\in S\setminus\{p\}}(f_{pq}(t))$, finding the nearest neighbor of p at time t reduces to locating the interval $[t_1, t_2]$ containing t. The lower envelope has size $O(\lambda_{2k}(n))$ and can be computed in time $O(dk^2n + \lambda_{2k}(n)\log n)$ by the divide&conquer algorithm of Atallah [2]. Storing the interval endpoints of the envelope in a binary tree, nearest neighbor queries for point p can be answered in time $O(\log\lambda_{2k}(n)) = O(\log n)$.

To solve the all nearest neighbor problem, we have to do the above for every point p in S. Doing this, the size of the structure and the preprocessing time are multiplied by n and it is not satisfying to answer queries in time $O(n\cdot\log n)$. However, since the query time t is the same for every point p, we can store the lower envelopes for the different points in one data structure: As mentioned above, the lower envelope for a point p induces a partition of the positive time axis $[0,\infty]$ into intervals where answers stay fixed. We store the intervals of all n envelopes in an interval tree [5]. The tree contains $O(n\lambda_{2k}(n))$ intervals. Each interval is labeled by a pair (p,q') denoting that in this interval, q' is the nearest neighbor of p. Note, that by construction, the number of intervals containing the value t is, for any t, exactly n. Then, we can report the nearest neighbor for every point in S in time $O(n)$: we report all the intervals containing t in the interval tree. This takes time $O(\log \#\text{intervals} + \text{size of the answer})$ [5], which is $O(\log(n\lambda_{2k}(n)) + n) = O(n)$.

Of course, we can use the same approach for the all furthest neighbor problem: Use the same algorithm for the upper envelope.

Theorem 1 *For the dynamic all nearest (furthest) neighbor problem there exists a data structure of size $O(n\lambda_{2k}(n))$ that can be built in time $O(dk^2n^2 + n\lambda_{2k}(n)\log n)$, such that the nearest (furthest) neighbor of each point at a query time t can be reported in time $O(n)$.*

To solve the m-th nearest neighbor problem for a specific point p, we need to store the whole arrangement defined by the functions $f_{pq}(t)$, $q \in S\setminus\{p\}$ and not only its lower envelope. We do this by using persistent data structures [3]. The data structure is built as follows. Compute the at most $2k\binom{n-1}{2}$ intersections of the functions $f_{pq}(t), q \neq p$, and sort them w.r.t. time t. Then, make a sweep over time. Between two intersections, the vertical order of the $n-1$ curve segments remains fixed. We store these segments in a data structure for ordered lists that allows queries of the form "return the m-th element of the list" in $O(\log n)$ time, e.g. a binary search tree storing the elements of the list. The elements are stored, together with their ranks, in the leaves. Note that then, exchanging two list elements causes only $O(1)$ changes in the structure. At each intersection of two curves, two adjacent elements of the list flip. We perform this update in a persistent data structure. Having processed all intersections, we have obtained a data structure of size $O(\text{size of list} + \#\text{intersections}) = O(n + kn^2) = O(kn^2)$ such that a query of the version of the list that exists at time t can be answered in time proportional to the query time of the original (non-persistent) data structure for the list. The preprocessing time is dominated by the time needed to compute the distances and to sort the intersections, i.e. $O(dk^2n + kn^2\log n)$. We have the following

Theorem 2 *For the dynamic m-th nearest neighbor problem there exists a data structure of size $O(kn^2)$ that can be built in time $O(dk^2n + kn^2\log n)$, such that the m-th nearest neighbor of the point $p \in S$ at a query time t can be reported in time $O(\log n)$.*

Of course, this can be extended to the all m-th nearest neighbor problem, analogous to the extension for $m = 1$ (nearest neighbor) and $m = n - 1$ (furthest neighbor) in Theorem 1. We have

Theorem 3 *For the dynamic all m-th nearest neighbor problem there exists a data structure of size $O(kn^3)$ that can be built in time $O(dk^2n^2 + kn^3 \log n)$, such that the m-th nearest neighbor of each point at a query time t can be reported in time $O(n)$.*

3 The dynamic euclidean minimum spanning tree problem

Consider the complete graph $G(t)$ for the points in S at time t. Its nodes are the points at this time and its edges are the line segments joining pairs of points. The EMST is a spanning tree of $G(t)$. In this section, we want to build a data structure for the points in S, such that for any query time t, we can report the EMST at this time.

It follows immediately from Kruskal's algorithm that the EMST is fully determined by the ordering of the edge lengths. (See any standard algorithms book.) Therefore, the EMST can only change if the ordering of the edge lengths changes, i.e., if two adjacent edges are flipped in the ordering.

Assume for simplicity that at any time there can be at most one flip. Or, equivalently, the $\binom{n}{2}$ functions representing the distances between pairs of points have the property that no three of them intersect in one point.

Suppose that edges e and e', with lengths $|e|$ and $|e'|$, respectively, flip at time t. That is, there is an $\epsilon > 0$, such that during the time interval $[t - \epsilon, t + \epsilon]$, e and e' are adjacent in the edge ordering, and, furthermore, $|e| < |e'|$ at time $t - \epsilon$, $|e| = |e'|$ at time t, and $|e| > |e'|$ at time $t + \epsilon$.

Lemma 1 *If at time $t-\epsilon$, e and e' both are not in the EMST, or e and e' both are in the EMST, or e is not in the EMST and e' is in the EMST, then the EMST does not change during the time interval $[t - \epsilon, t + \epsilon]$.*

Because of this lemma, we only have to consider the case where, at time $t - \epsilon$, e is in the EMST and e' is not. Define the *fundamental cycle* of a non-tree edge e' as the cycle of the tree that arises when we add e' to it.

Lemma 2 *Suppose that at time $t - \epsilon$, e is in the EMST and e' is not. If e is not a part of the fundamental cycle of e', then the EMST does not change during the time interval $[t - \epsilon, t + \epsilon]$. Otherwise, the EMST changes at time t. At that time, edge e is replaced by edge e'.*

At this moment, we know exactly how the EMST changes over time. How do we implement the algorithm? At the start, we construct the $\binom{n}{2}$ functions each of which describes the square of the distance of a pair of points, as a function of t. Then we compute the intersections of these functions, and sort them w.r.t. their time parameters. Since the functions have degree $2k$, there are at most $2kn^4$ intersections, each corresponding to a flip of two edges.

Given these intersection points, we can make a sweep over time. Each time we encounter an intersection, we check the conditions of the above two lemmas. If the EMST changes, we delete one edge and insert the new one. Checking the conditions and inserting/deleting edges can be done using Dynamic Trees [8].

We have seen how to keep track of the changes of the EMST. Now we describe the data structure that will be used to answer queries. We shall use an interval tree [5], where each interval is labeled by an edge of the graph. Interval $[i, j]_r$ is in the tree if and only if edge r is part of the EMST from time i to time j. Then, a query at time t is simply solved by reporting all the intervals containing the value t. This takes time $O(\log n + A)$, which is $O(n)$, since $A = n - 1$.

During the sweep over the intersections of the curves, we compute, for each edge e, a list $L(e)$ containing the time intervals when e is in the EMST. At the end of the sweep, we insert these intervals into the interval tree. This gives a data structure of size $O(kn^4)$, since there are only that many flips, and each flip consists of adding and removing only one edge. The dynamic tree operations implementing a flip take logarithmic time, and the interval tree can be built in $O(kn^4 \log n)$ time. Also, the initial computation of the distances and sorting of the intersection points take time $O(dk^2n^2)$ and $O(kn^4 \log n)$, respectively. Therefore, the preprocessing time is $O(dk^2n^2 + kn^4 \log n)$.

We have proved the following

Theorem 4 *For the dynamic EMST problem, there exists a data structure of size $O(kn^4)$ that can be built in time $O(dk^2n^2 + kn^4 \log n)$, such that at any time t, the EMST at that time t can be reported in time $O(n)$.*

4 Proximity problems

In this section, we want to build data structures for the points in S, such that for any query time t and any distance r, we can (i) count the number of pairs of points which are at most r apart from each other at time t or (ii) report all such pairs. We shall first describe a simple method that works for both versions of the problem and then improve the storage bound for the reporting problem.

As in the EMST problem, we construct the $\binom{n}{2}$ (squared) distance functions, compute their pairwise intersections and sort them w.r.t. their time parameters. Drawing these functions in one graph where t is shown at the horizontal axis and the distances are shown at the vertical axis, our problem has now turned into the following: Given an arrangement of $\binom{n}{2}$ polynomial curves in the plane, report (count) all curves lying below point (t, r^2).

This problem is implicitly solved by Agarwal et. al. [1]. They treat the problem of Jordan arcs that intersect a halfplane which they transform (by dualization) to the problem given above in order to argue about the space complexity of their algorithm. Of course, the ideas can be used to solve our problem directly. We just give a sketch here, a detailed description can be found in [6].

The simple method is a straighforward extension of the point location method of Sarnak and Tarjan [7], and it works for reporting and counting.

The data structure is built analogously to the one used for the m-th nearest neighbor problem in Section 2. Given the intersections of the curves, we make a sweep over time. Between two intersections, the vertical order of the curve segments remains fixed. We store these segments in a data structure for ordered lists, e.g. a balanced binary search tree. At each intersection of two curves, two adjacent elements of the list flip. We perform this update in a persistent data structure. Having processed all intersections, we have obtained a data structure of size $O(\text{size of list} + \#\text{intersections}) = O(n^2 + kn^4) = O(kn^4)$ such that a query of a version of the list that exists at time t can be answered in the same time as the original (non-persistent) data structure for the list, plus the time to find the right version to search in, which is $O(\log \#\text{versions}) = O(\log(kn^4)) = O(\log n)$.

Therefore, let us first ignore that the lists are persistent when explaining the point location algorithm. Intuitively, to answer a point location query for point (t, r^2) in the subdivision defined by the curves, we first have to find the version of the list at time t, i.e. the intersections t_1, t_2 such that $t_1 \leq t \leq t_2$. This takes time $O(\log n)$. In this list of curve segments, we locate the point using its vertical coordinate r^2.

Since we store an ordered list of segments in a balanced search tree, locating a point $p = (t, r^2)$ vertically in this list, i.e. finding the segment that lies directly below p, takes $O(\log n)$ time. Location of point $p = (t, r^2)$ is then solved by locating the point vertically in the list which is valid at time t. Note, however, that the different lists are stored implicitly in the persistent data structure. Therefore, from the discussion of persistence above, the total point location time is $O(\log n)$.

Reporting all curves below a point is straightforward: we just traverse the list from the segment directly below the query point downwards to the first (lowest) segment, reporting the curves to which these segments belong. To solve the counting problem, we associate the number of curves below a segment with each segment stored in a list. We have the following

Theorem 5 *The proximity problem at time t and distance r can be solved by a data structure that can be built in time $O(dk^2n^2 + kn^4 \log n)$, uses $O(kn^4)$ storage, and answers report and count queries in $O(\log n + A)$ and $O(\log n)$ time, respectively.*

For the reporting version, this result can be improved w.r.t. storage requirements. Consider the data structure given above and ignore at the moment that the various lists are stored in a persistent data structure. We have q lists, each of size $m = \binom{n}{2}$, where $q = O(km^2) = O(kn^4)$ is the number of intersections among the m curves. The reporting problem was solved by locating point (t, r^2) in the list corresponding to interval $[t_1, t_2]$ such that $t \in [t_1, t_2]$, followed by reporting all elements below the point. To locate (t, r^2) vertically in a list, a binary search tree was used. The data structure was essentially the same for counting and reporting.

Now, if we only want to solve the reporting problem, we can use a simpler method: we do not need to locate (t, r^2) in a binary search tree; we just have to find the correct list corresponding to interval $t_1 \leq t \leq t_2$, start at the beginning of the list (containing the lowest curve segment), walk along the list and stop after we have reached a segment that lies above (t, r^2).

The idea for the improvement [1, 6] is now as follows. During the preprocessing, we use the original sequence $\mathcal{Z} = Z_1, \ldots, Z_q$ of lists to create a considerably smaller sequence $\mathcal{Z}' = Z'_1, \ldots, Z'_u$ of lists which will be used for the queries. In $\mathcal{Z}$, each intersection of two curves causes two adjacent list elements to flip.

In the smaller sequence $\mathcal{Z}'$, we do not create a new list for every flip, but only for flips at certain positions $b_1 \leq \ldots \leq b_l$, called *borders*. That is, we walk through the sequence $\mathcal{Z}$ of lists, and if there is a flip at some border position from Z_i to Z_{i+1}, and we have already created lists $Z'_1, \ldots, Z'_j$, $j \leq i$, we create a new list Z'_{j+1} by flipping the elements which are at the border position in Z_i. Note that in the list Z'_j, these elements need not be adjacent, since in the lists of $\mathcal{Z}'$, we "forget" the flips that are not at border positions. As a result we obtain a sequence $\mathcal{Z}'$ where elements are ordered across borders and the order between two borders is lost. The sequence $\mathcal{Z} = Z_1, \ldots, Z_q$ is slimmed down to a sequence $\mathcal{Z}' = Z'_1, \ldots, Z'_u$ that corresponds to a subsequence $Z_{i_1}, \ldots, Z_{i_u}$ of $\mathcal{Z}$ as follows: $Z_1 = Z_{i_1}$, flips occur at border positions in $Z_{i_j}, 2 \leq j \leq u$, and at non-border positions in $Z_{i_j+1}, \ldots, Z_{i_{j+1}-1}, 1 \leq j \leq u$. We say that the lists $Z_{i_j}, \ldots, Z_{i_{j+1}-1}$ (and the time interval covered by them) are *represented by* Z'_j. Therefore, the number of lists in $\mathcal{Z}'$ depends on the total number of flips that can occur at border positions. The new search algorithm is as follows.

First, find the list representing time t in the sequence $\mathcal{Z}' = Z'_1, \ldots, Z'_u$, say Z'_i, in $O(\log u)$ time. Then, from the beginning of Z'_i, walk up reporting all curves lying below $p = (t, r^2)$, until a curve is found that is not below p . Walk further to the next border position, say b_j, checking each curve.

Since the elements are ordered across borders, we are sure that curves beyond position b_j cannot contribute to the answer. Note that we can afford to check elements that do not contribute to the answer without affecting the asymptotic query time if their number is proportional to the size of the answer.

The goal is now to choose positions $b_1, \ldots, b_l$ in the preprocessing algorithm such that (i) l is small, (ii) there are not many flips at these positions, and (iii) the positions are not too far apart from each other.

Let the set of exchanges of positions j and $j+1$ in the lists of the sequence $\mathcal{Z}$ be denoted by *j-flip*.

Lemma 3 ([1, 6]) *We can choose border positions $b_1, \ldots, b_l$ such that $|b_i\text{-flip}| = O(\lambda_{2k}(m))$ for $1 \leq i \leq l$, $b_l \geq m/3$, and $2 \leq b_i/b_{i-1} < 3$ for $2 \leq i \leq l$.*

Theorem 6 *The size of the structure is $O(\lambda_{2k}(n^2)\log n)$, and the preprocessing time is $O(dk^2n^2 + kn^4 \log n)$.*

Proof: From Lemma 3, we have to store $O(\lambda_{2k}(m)\log m)$ lists, since this is the total number of flips across borders. Since we store the lists persistently and the size of each (original) list is m, we need $O(m + \lambda_{2k}(m)\log m) = O(\lambda_{2k}(m)\log m)$ space. Finally, we have $m = \binom{n}{2}$. The additional preprocessing that is needed in comparison to the simple method described before is still dominated by the time to compute the curves and all their intersections, i.e. $O(dk^2n^2 + kn^4 \log n)$. ∎

Theorem 7 *The query time of the structure is $O(\log n + A)$, as in the simple method.*

Proof: See the algorithm given above. Finding the right list to search in (the lists are kept separately only conceptually, since we store them in a persistent data structure) takes time $O(\log u) = O(\log n)$, since $u = O(\lambda_{2k}(n^2)\log n)$. Then we report answers until we encounter an element that is not part of the answer. Let b_i be the last border position that was crossed before this event. Then we have $A \geq b_i$. We proceed until we reach the next border position b_{i+1}. Therefore, the total time to search the list is $O(b_{i+1})$. From Lemma 3, $b_{i+1}/b_i < 3$, and therefore $b_{i+1} < 3b_i \leq 3A$. ∎

5 The dynamic maxima problem

We want a data structure such that at any query time t, we can report the maximal elements of S at this time, or count their number.

A point $p = (p^1, \ldots, p^d)$ dominates point $q = (q^1, \ldots, q^d)$ if $p \neq q$ and $p^i \geq q^i$ for all $1 \leq i \leq d$. The *maximal points* in S are those points that are not dominated by any other point in S.

Let p be a fixed point of S. We want to compute the time intervals during which p is maximal. For each point $q \neq p$ in S, we compute the time intervals during which p is dominated by q, as follows. The differences $p^i(t) - q^i(t), 1 \leq i \leq d$, are polynomials of degree at most k. For each $1 \leq i \leq d$, compute the at most $\lceil (k+1)/2 \rceil \leq k$ intervals during which $p^i(t) - q^i(t) \leq 0$. This gives at most dk intervals. The intersection of these intervals yields all time intervals during which p is dominated by q, i.e., p is not maximal in S. Computing the complement of these intervals gives at most $dk+1$ time intervals during which p is not dominated by q.

Having done this for each point q, we have $n-1$ lists of disjoint intervals, one list for each point $q \neq p$. Each list consists of at most $dk+1$ intervals. Compute the intersection of all these intervals. This intersection consists of at most $(n-1)(dk+1)$ intervals, during which point p is maximal in the set S.

Repeat this for each point p of S, and store the resulting collection of at most $n(n-1)(dk+1)$ intervals in an interval tree. With each interval, store the name of the point that is maximal during it.

To report all maximal points at a given time t, search in the interval tree for all intervals that contain t, and report the points corresponding to them. These points give the maximal points at time t. The counting version of the interval tree enables us to count the number of maximal points at any given time t.

Theorem 8 *For the dynamic maxima reporting problem there exists a data structure of size $O(dkn^2)$ that can be built in time $O(dkn^2 \log n)$, such that for any t, all A maximal points at time t can be reported in time $O(\log n + A)$. For the corresponding counting problem there exists a data structure using equal space and preprocessing, such that the number of maximal points at time t can be reported in time $O(\log n)$.*

6 The dynamic range query problem

Let $\mathcal{I} = I_1 \times \ldots \times I_d$ be a hyperrectangle. We want a data structure for the following query problem: Given a query time t, report all points in S that are contained in $\mathcal{I}$ at time t.

Again, we fix a point p. We want to compute the time intervals during which p is contained in $\mathcal{I}$.

For each $1 \leq i \leq d$, do the following: Let $I_i = [a, b]$. We want to find all time intervals during which $a \leq p^i(t) \leq b$. This can easily be done by computing the roots of the polynomials $p^i(t) - a$ and $p^i(t) - b$. This gives at most k pairwise disjoint intervals.

Having done this for each i, we compute the intersection of these intervals. This gives at most dk intervals during which p is contained in the query rectangle.

We do this for each point in S and store the resulting collection of at most dkn intervals in an interval tree. With each interval, we store the name of the point corresponding to it.

To report all points that are contained in $\mathcal{I}$ at a given time t, search in the interval tree for all intervals that contain t, and report the points corresponding to them. Of course, as in the maxima counting problem, we can also count the number of points that are contained in $\mathcal{I}$ at time t.

Theorem 9 *For the dynamic range query problem there exists a data structure of size $O(dkn)$ that can be built in time $O(dkn \log n)$, such that all A points that are contained in the hyperrectangle $\mathcal{I}$ at a query time t can be reported in time $O(\log n + A)$. For the corresponding counting problem there exists a data structure of equal size and preprocessing, such that the number of points that are contained in the hyperrectangle $\mathcal{I}$ at a query time t can be reported in time $O(\log n)$.*

Acknowledgements: We would like to thank Marc van Kreveld for very helpful discussions.

References

[1] P.K. Agarwal, M. van Kreveld and M. Overmars. *Intersection queries for curved objects.* Proc. 7th Annual ACM Symp. on Computational Geometry (1991), 41-50.

[2] M.J. Atallah. *Dynamic computational geometry.* Proc. 24th FOCS (1983), 92-99.

[3] J. Driscoll, N. Sarnak, D.D. Sleator and R.E. Tarjan. *Making data structures persistent.* J. of Computer and System Sciences **38** (1989), 86-124.

[4] J.-J. Fu and R.C.T. Lee. *Voronoi diagrams of moving points in the plane.* International J. of Computational Geometry & Applications **1** (1991), 23-32.

[5] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, 1985.

[6] M. van Kreveld. *New Results on Data Structures in Computational Geometry.* Ph.D. Thesis, University of Utrecht, The Netherlands, 1992.

[7] N. Sarnak and R.E. Tarjan. *Planar point location using persistent search trees.* Comm. of the ACM **29** (1986), 669-679.

[8] D.D. Sleator and R.E. Tarjan. *A data structure for dynamic trees.* J. of Computer and System Sciences **26** (1983), 362-391.