Answer-Pairs and Processing of Continuous Nearest-Neighbor Queries[*]

A. Prasad Sistla Ouri Wolfson Bo Xu
Department of Computer Science
University of Illinois at Chicago
{sistla,wolfson,boxu}@uic.edu

Naphtali Rishe
School of Computing and Information Sciences, Florida International University
rishen@cis.fiu.edu

ABSTRACT

We consider the problem of evaluating the continuous query of finding the *k* nearest objects with respect to a given moving point-object O_q among a set of *n* moving point-objects. The query returns a sequence of answer-pairs, namely pairs of the form (*I*, *S*) such that *I* is a time interval and *S* is the set of objects that are closest to O_q during *I*. Existing work on this problem lacks complexity analysis due to limited understanding of the maximum number of answer-pairs. In this paper we analyze the lower bound and the upper bound on the maximum number of answer-pairs. Then we consider two different types of algorithms. The first is off-line algorithms that compute a priori all the answer-pairs. The second type is on-line algorithms that at any time return the current answer-pair. We present the algorithms and analyze their complexity using the maximum number of answer-pairs.

Categories and Subject Descriptors
D.3.3 [**Database Management**]: Systems – *query processing.*

General Terms
Algorithms, Performance, Theory.

Keywords
Nearest-neighbor queries, continuous queries, moving objects databases, complexity analysis, kinetic data structure.

1. INTRODUCTION

A C*k*NN query is a query that continuously finds the *k* nearest neighbors with respect to a given moving point-object O_q among a set of *n* moving point-objects. The query returns a sequence of answer-pairs, namely pairs of the form (*I*, *S*) such that *I* is a time interval and *S* is the set of *k* objects that are nearest to O_q during *I*. C*k*NN queries see many applications in mobile computing environments. For example, in a road network, the C*k*NN query continuously provides a driver with the locations of the *k* nearest vehicles from her location. The query enables the driver to be aware of the vehicles that are blocked by a truck, a turn, a blind zone, etc. In a digital battlefield, a vehicle may use the C*k*NN query to monitor the *k* nearest hostile (or friendly) vehicles for surveillance (or for support).

The processing of C*k*NN queries has been extensively studied in the database community (see e.g., [3, 4]). However, existing studies lack complexity analysis due to limited understanding of the maximum number of answer-pairs. These studies either do not provide algorithm complexity analysis (such as [3]), or treat the number of answer-pairs as an input size that is independent of *n* (such as [4]). In order to enable complexity analysis for C*k*NN query processing, our paper bounds the value of $\beta_k(n)$ which represents the maximum number of pairs in an answer to the C*k*NN problem with *n* moving objects. Throughout the paper we assume that *k* is a constant. We study both the lower bound and the upper bound of $\beta_k(n)$.

We start with the linear motion model, in which all the objects, including O_q, start from different points and move with a constant velocity-vector in a multi-dimensional space. We adopt a standard strategy which transforms the problem into the Time-SquareDistance space, where the square distance between a moving object O_i and the query object O_q as a function of time is a parabola curve. With this transformation, $\beta_k(n)$ is upper bounded by the maximum number of edges in the *k-level* of an arrangement of *n* parabola curves ([8]). It is well known that the maximum number of edges in the *k*-level is O(*n*) when *k* is a constant (see e.g. [6]). However, most of the existing literature only gives order-statistics results. Theorem 3.1 in [2] implies an exact upper bound of $8k(n\text{-}k)+1$. In this paper we give an exact upper bound of $k(2n-k-1)+1$. Since $k(2n-k-1)<8k(n\text{-}k)$ when $k<<n$, and for most real applications this is indeed the case, our bound is tighter.

Now the question is whether the O(*n*) upper bound is attainable. In other words, what is the lower bound of $\beta_k(n)$. In this paper we show that O(*n*) is also the lower bound of $\beta_k(n)$. We show this by constructing a feasible configuration in terms of the motion of objects such that $\beta_k(n)$ is equal to $2(n-k)+1$.

Then we bound $\beta_k(n)$ for a second motion model, called the piecewise linear model, in which the objects change their velocities a finite number of times denoted by *m*. In this case, each object is represented by a sequence of connected parabola-segments in the Time-SquareDistance space. Using Theorem 2.4

[*] Research supported by NSF DGE-0549489, IIS-0957394, CNS-0821345, HRD-0833093, IIP-0829576, IIP-0931517, CNS-1057661, IIS-1052625, and CNS-0959985.

in [8], we can easily prove an O($nm2^{\alpha(nm)}$) upper bound for $\beta_k(n)$ where α is the functional inverse of Ackermann's function (see [8] for the definition of Ackermann's function). However, we prove a tighter bound of O($nm\alpha(n)$), utilizing the fact that the parabola-segments of each individual object are one-by-one connected. We further show that the lower bound of $\beta_k(n)$ for the piecewise linear model is $2m(n-k)+1$=O(nm). Given the fact that $\alpha(n)$ is at most 4 for any practical value of n, we can say that the bound of O($nm\alpha(n)$) is nearly tight as it only adds an almost-constant factor to the lower bound.

In the second part of the paper, we study the processing of C*k*NN queries. We consider two different processing styles, namely off-line processing and on-line processing. The off-line processing computes a priori all the answer-pairs. The on-line processing at any time returns the current answer-pair. For off-line processing, there is an existing simple divide-and-conquer algorithm which gives the solution for the linear model in O($n\log n$) time (see Theorem 2.6 in [8]). For the piecewise linear model, when k=1, the algorithm introduced in [1] can be used to give the solution in O($nm\log(nm)$) time. This algorithm treats the parabola-segments of each individual object as independent functions. In this paper we present an algorithm which gives the solution for an arbitrary k value in O($nm\alpha(n)\log n$) time. Our algorithm utilizes the fact that the parabola-segments of an object belong to the same function. Since $\alpha(n)$ is almost a constant, our algorithm is practically more efficient than the algorithm in [1]. Furthermore, observe that the time complexity of our algorithm is only higher by a factor of $\alpha(n)\log n$ than the lower bound which is the maximum number of answer-pairs.

For on-line processing, we develop a kinetic data structure, called *object heap*. This data structure allows updates like insertion of a new object, deletion of an existing object, and velocity-vector change of an existing object. We analyze the complexity of object heap under two different conditions, depending on whether there are updates or not. In either case, we assume that initially each object moves linearly until updated. When there are no updates, the *cumulative complexity* of object heap, i.e., the total cost for returning all answer-pairs (Recall that on-line processing returns one answer-pair each time), is O($n\log^2 n$). This is the same as that of *kinetic tournament* [5] and *kinetic heap* [5,7], two classical kinetic data structures for monitoring the nearest neighbor (i.e., k=1)[1]. However, object heap is better at handling updates. In object heap, insertion, deletion, and velocity-vector change can be carried out in O($\log n$) time each. In kinetic tournament and kinetic heap, these operations require O($\log^2 n$) time each. We prove that if each object can change its velocity-vector at most m times, then the cumulative complexity of our on-line algorithm is O($nm\alpha(n)\log^2 n$) which is higher than the lower bound by a factor of $\alpha(n)\log^2 n$. Furthermore, our data structure satisfies all the four quality criteria defined in [5] for a kinetic data structure, namely responsiveness, compactness, locality, and efficiency. Intuitively, these criteria mean that the processing does not look too much ahead because there may be updates coming in that will invalidate such look-ahead.

[1] Better complexity is known for kinetic heap when the distance functions are pseudo-lines (i.e., any pair of distance functions intersect each other at most once) (see [7]). In our problem, the square-distance functions are parabolas. Observe that each parabola can be cut into two pseudo-lines. However, this will make the square-distance functions partially defined whereas the analysis in [7] assumes totally defined functions.

In summary, the main contributions of this paper are the following:

1. We give exact numbers that upper bound and lower bound the number of answer-pairs in a C*k*NN query for the linear model for a given constant k. These bounds prove that the O(n) bound to the maximum number of answer-pairs is tight.

2. We prove that the upper bound to the maximum number of answer-pairs in a C*k*NN query for the piecewise linear model is O($nm\alpha(n)$) and the lower bound is $2m(n-k)+1$.

3. We introduce an algorithm that computes answer-pairs for the piecewise linear model. The algorithm improves the complexity from the existing O($nm\log(nm)$) to O($nm\alpha(n)\log n$).

4. We introduce a kinetic data structure for on-line processing which has the cumulative complexity of O($nm\alpha(n)\log^2 n$) if each object can change its velocity-vector at most m times. The complexity of an update operation improves from the existing result by a factor of $\log n$.

Table 1.1. Summary of the results.

problem			**previous result**	**our result**
number of answer-pairs	linear	lower	not studied	$2(n-k)+1$
		upper	$8k(n-k)+1$	$k(2n-k-1)+1$
	piecewise linear	lower	not studied	$2m(n-k)+1$
		upper	O($nm2^{\alpha(nm)}$)*	O($nm\alpha(n)$)
off-line processing, piecewise linear			O($nm\log(nm)$)*	O($nm\alpha(n)\log n$)
on-line processing	single update		O($\log^2 n$)	O($\log n$)
	cumulative complexity with m velocity-changes per object		not studied	O($nm\alpha(n)\log^2 n$)

* Results obtained by a straightforward application of existing techniques.

The rest of this paper is structured as follows. Section 2 analyzes the bound on the number of answer-pairs. Section 3 discusses query processing. Section 4 concludes the paper and discusses future work.

2. MAXIMUM NUMBER OF ANSWER-PAIRS

Subsection discusses the number of answer-pairs in the case where objects move linearly. Subsection 2.2 discusses the number of answer-pairs in the case where objects move piecewise linearly.

2.1 Number of Answer-pairs with Linear Motion

We consider query processing at a *query object* O_q that is moving. O_q has an *objects database* that stores the motion information of each other *data object* O_1, O_2,…, O_n that is also moving. The

CkNN query requests for every point in time the k nearest neighbors of O_q among all the data objects. All the objects move in a linear manner in a multi-dimensional space called the *motion space*. That is, each object moves along a straight line with a constant speed. All the objects start at negative infinity and continue to positive infinity. Then the square of the distance d_i between O_q and a data object O_i is a quadratic function of time t: $d_i^2(t) = at^2 + bt + c$, where a, b, and c are parameters dependent on the velocities and initial locations of O_q and O_i. The coefficient a is non-negative and thus the $d_i^2(t)$ function is convex. The $d_i^2(t)$ function is a parabola in the Time-SquareDistance space [3] as illustrated in Figure 2.1. For convenience of presentation, in the rest of this paper we use object O_i and its square-distance function $d_i^2(t)$ interchangeably when there is no confusion.

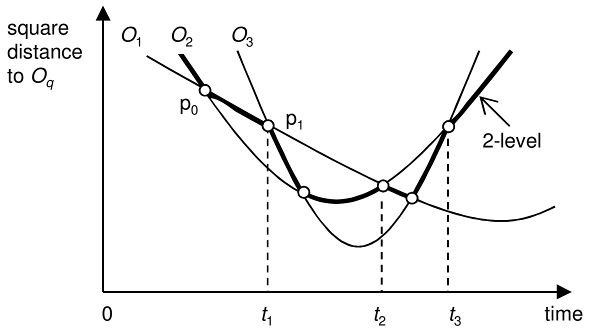


Figure 2.1. Example of answer-pairs in the Time-SquareDistance space

The CkNN query is issued at time 0. At that time O_q has a set of k nearest neighbors, but as time progresses, the kNN set may change. A time interval during which the kNN set remains unchanged is referred to as an *answer interval*. An *answer-pair* is a pair consisting of an answer interval and its associated kNN set. Answer-pairs are required to be non-redundant in that, for any pairs (I, S) and (I', S'), the sets S and S' are distinct if I and I' are contiguous. As an example, in Figure 2.1, if k=2, then the answer-pairs are:

$<[0,t_1),\{O_1, O_2\}>$,

$<[t_1,t_2),\{O_2, O_3\}>$,

$<[t_2,t_3),\{O_1, O_3\}>$,

$<[t_3, \infty),\{O_1, O_2\}>$

Denote by $I_k(G)$ the number of answer-pairs for an instance G of the objects database for the CkNN query. Define

$\beta_k(n)$=max$\{I_k(G)|G$ is an instance of the objects database with n objects$\}$

In words, $\beta_k(n)$ is the maximum number of answer-pairs for the CkNN query among all the objects database instances with n objects. First we examine the upper bound of $\beta_k(n)$. Observe that for any objects database instance, the number of answer-pairs is upper bounded by the number of times the k-th nearest neighbor changes. This is because each change of the answer set is always caused by a change of the k-th nearest neighbor whereas a change of the k-th nearest neighbor does not necessarily cause a change of the answer set. Specifically, the answer set changes only when there is a switch of order between the k-th nearest neighbor and the (k+1)st nearest neighbor. In the Time-SquareDistance space, such a switch occurs when a k-th lowest curve is crossed by a curve from above. For example, in Figure 2.1, at point p_1, the 2nd lowest curve O_1 is crossed by O_3 from above, and thus the answer set is changed at p_1. Specifically, O_1 is removed from the answer set, and O_3 is added to the answer set. On the other hand, the answer set does not change when a k-th lowest curve is crossed by a curve from below. For example, in Figure 2.1, at point p_0, the 2nd lowest curve O_2 is crossed by O_1 from below, and thus the answer set is not changed at p_0.

According to the above observation, the number of answer-pairs is upper bounded by the number of pieces in the envelope formed by the k-th lowest curves. The latter number has been studied in the context of arrangements of curves and is formally defined as follows (see [8]). Given a set Γ of curves where each curve is a continuous and univariate function, the arrangement $A(\Gamma)$ of Γ is the planar subdivision induced by the curves in Γ. That is, $A(\Gamma)$ is a planar map the *vertices* of which are the pair-wise intersection points of the curves in Γ and the *edges* of which are maximal connected portions of the curves that do not contain a vertex. As an example, Figure 2.1 shows the arrangement of three objects O_1, O_2, and O_2, where white circles are vertices. The *level* of a point p in $A(\Gamma)$ is the number of curves in Γ not above p, and the level of an edge $e \in A(\Gamma)$ is the common level of all the points lying in the relative interior of e[2]. The *k-level* of $A(\Gamma)$ is the union of all edges in $A(\Gamma)$ whose level is $k-1$. The *length* of the k-level is the number of edges in the k-level. In Figure 2.1, the thick curve shows the 2-level in the arrangement of O_1, O_2, and O_3; its length is 7.

It is well known that the length of the k-level in an arrangement of n parabolas, each with an axis of symmetry that is parallel to the y-axis, is O(n) for any constant k (see e.g. [6]). However, most of the existing literature only gives order-statistics results. Theorem 3.1 in [2] implies an exact bound of $8k(n-k)+1$. In the following lemma we give a tighter exact bound.

Theorem 2.1. *Assume that the query object and data objects all move linearly. Then for any constant k* $\beta_k(n) \le k(2n-k-1)+1$.

Before proving Theorem 2.1, we introduce some definitions and denotations. Given the arrangement of data objects in the Time-SquareDistance space, an intersection point in the arrangement is referred to as a *≤k-level intersection* (respectively *>k-level intersection*) if it is an endpoint of an edge the level of which is smaller than or equal to k (respectively greater than k). Let p be an intersection of two objects A and B. We use *p.time* to denote the time coordinate of the intersection. p is associated with another two attributes, namely its downward and upward. A is the *downward* of p, and A *downward-crosses* B at p, if A is farther than B immediately before *p.time* (and thus closer than B immediately after *p.time*). In this case B is the *upward* of p, and B *upward-crosses* A at p. Clearly, for any two objects A and B, A downward-crosses B at most once and A upward-crosses B at most once.

[2] In the computational geometry literature, the level of a point p is defined to be the number of curves lying strictly below p. In this paper we define the level to be the number of curves not above p, so that the k-level corresponds to the k-th nearest neighbor.

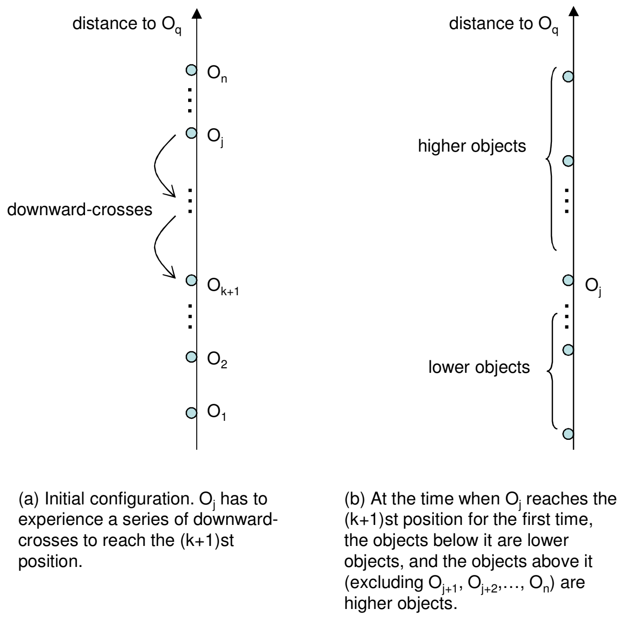
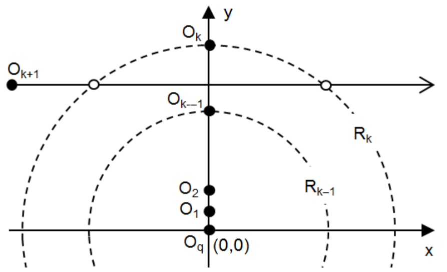

distance to O_q
O_n
O_j
downward-crosses
O_{k+1}
O_2
O_1
(a) Initial configuration. O_j has to experience a series of downward-crosses to reach the (k+1)st position.
distance to O_q
higher objects
O_j
lower objects
(b) At the time when O_j reaches the (k+1)st position for the first time, the objects below it are lower objects, and the objects above it (excluding O_{j+1}, O_{j+2},…, O_n) are higher objects.

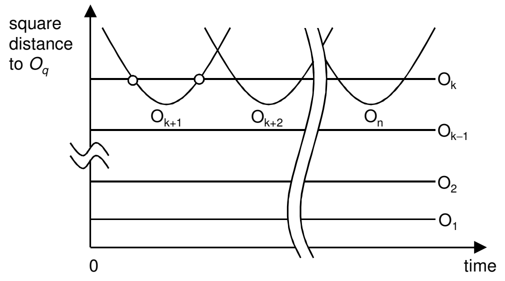
Figure 2.2. The auxiliary figure for the proof of Theorem 2.1.

Proof of Theorem 2.1. We prove by showing that the number of $\leq k$-level intersections is at most $k(2n-k-1)$. Without loss of generality, we assume that at the time when the query is processed, O_1 is the first nearest neighbor, O_2 is the second nearest neighbor, …, and O_n is the n-th nearest (and the farthest) neighbor (see Figure 2.2(a)). We construct the arrangement in the same order. That is, we add O_1 first, and then add O_2, and so on. We show that for any integer $k<j\leq n$, the addition of O_j can introduce at most $2k$ $\leq k$-level intersections to the final arrangement of the n objects. To do this, we divide the intersections introduced by O_j to the arrangement of the first j objects into two groups, namely the $\leq k$-level intersections and $>k$-level intersections. We call these two groups *lower intersections* and *higher intersections* respectively. Observe that, due to the later introduction of O_{j+1}, O_{j+2}, and so on, some of the lower intersections may become $>k$-level intersections in the final arrangement. However, none of the higher intersections may become $\leq k$-level intersections in the final arrangement. Thus the number of $\leq k$-level intersections introduced by O_j to the final arrangement is upper bounded by the number of lower intersections. In the following we show that the number of lower intersections is at most $2k$.

If O_i never reaches the k-th position, then the statement clearly holds. If O_j does reach the k-th position, then consider the first time it reaches the (k+1)st position. Denote that time by T. Observe that O_j has to experience a series of downward-crosses to reach the (k+1)st position. We refer to the objects that are closer to O_q than O_j at time T as the *lower objects* and the objects that are farther away from O_q at time T as the *higher objects*. Clearly there are k lower objects. In Figure 2.2(b), the objects left to O_j are the lower objects. The objects right to O_j, excluding O_{j+1}, O_{j+2},…, O_n, are the higher objects. Notice that the set of lower objects and that of higher objects pertain to time T.



(a) Configuration in the motion space.



(b) Arrangement in the Time-SquareDistance space.

Figure 2.3. The auxiliary figure for the proof of Proposition 2.2

Assume that object O_j participates in d lower intersections with the higher objects after time T. Obviously, O_j can make at most $2k$ lower intersections with the lower objects. Observe that for each lower intersection that O_j makes with an higher object after time T, that higher object will move to the left of O_j and it can never move to the right of O_j again, because the downward-cross has already been consumed before time T.

Now observe that if more than $k-d$ of the lower objects make 2 lower intersections with O_j, it means that an higher object upward crosses O_j after time T; thus this is impossible. Therefore at most $k-d$ of the lower objects make 2 lower intersections with O_j and each one of the rest makes at most one lower intersection with O_j. Thus if the number of lower intersections of higher objects with O_j is d, then the maximum number of lower intersections of lower objects with O_j is $2k-d$. Thus the maximum number of lower intersections is $2k$.

So far we have studied the case of $k<j\leq n$. Using similar argument we can show that in the case of $j\leq k$ the number of $\leq k$-level intersections introduced by O_j to the final arrangement is $2(j-1)$.

In summary, the total number of $\leq k$-level intersections in the final arrangement is:

$$2k(n-k)+\sum_{j=1}^{k}2(j-1)=k(2n-k-1)$$

The length of the k-level is upper bounded by the number of $\leq k$-level intersections plus one. Thus the number of answer pairs is at most $k(2n-k-1)+1$.□

For some k's, the exact number given by Theorem 2.1 is tight. For example, when k=1, the number of answer-pairs is equal to the length of the 1-level which is tightly bounded by $2n-1$ according to Theorems 2.1 and 3.1 in [8]. Theorem 2.1 gives the same bound in this case. The order given by Theorem 2.1 (i.e., O(n)) is tight for all values of k, as shown by the following lemma.

Proposition 2.2. *Assume that the query object and data objects all move linearly. Then for any constant k* $\beta_k(n) \geq 2(n-k)+1$.

Proof: We construct a feasible case in which the number of answer-pairs is linear in n. Let objects O_q, O_1, O_2,…, and O_k be static in a 2D plane of the motion space such that O_i is the i-th nearest neighbor of O_q, as shown in Figure 2.3(a). Denote by R_i the circle the center of which is the location of O_q and the radius of which is the distance between O_i and O_q. Let object O_{k+1} move in the same 2D plane such that its route intersects R_k twice but does not intersect R_{k-1}. Let O_{k+2} have the same route as O_{k+1} and move behind O_{k+1} such that it enters R_k after O_{k+1} leaves R_k. Construct the same for O_{k+3} and so on. Figure 2.3(b) shows the arrangement of the objects in the Time-SquareDistance space. It is easy to see that the number of answer-pairs is $2(n-k)+1$. □

Corollary 2.3. *Assume that the query object and the data objects all move linearly. Then for any constant k* $\beta_k(n)=\Theta(n)$.

2.2 Number of Answer-pairs with Piecewise Linear Motion

Now we assume that objects move piecewise linearly in the motion space. That is, an object moves along a straight line with a constant speed from point a to point b. At point b the object changes its velocity-vector and moves to point c; there it changes the velocity-vector again and moves to point d, etc. Observe that in the Time-SquareDistance space in this model each object is represented by a connected sequence of parabola-segments (see Figure 2.4). It is easy to see that if the motion of O_q has m linear pieces and that of each O_i also has m linear pieces, then O_i has at most $2m-1$ parabola-segments in the Time-SquareDistance space.

According to Theorem 2.5 of [8], the length of the k-level in an arrangement of N parabola-segments is $O(N2^{\alpha(N)})$ where α is the functional inverse of Ackermann's function[3]. Based on this result, $\beta_k(n)= O(nm2^{\alpha(nm)})$ because there are at most $n(2m-1)$ parabola-segments totally for all objects in the Time-SquareDistance space. However, this derivation treats the parabola-segments of each individual object as independent segments. It does not utilize the fact that they are one-by-one connected. With the connectivity property taken into account, we obtain a tighter bound in the following lemma.

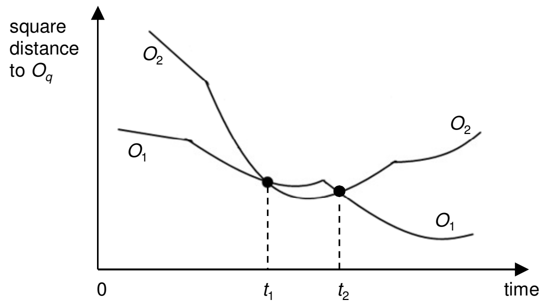


Figure 2.4. In the piece-wise linear model each object is represented by a connected sequence of parabola-pieces.

[3] See [8] for the definition of Ackermann's function.

Theorem 2.4: *Assume that the query object and the data objects all move piecewise linearly, where each object can have at most m linear pieces. Then* $\beta_k(n)=O(nm\alpha(n))$.

Due to the extremely fast growth of Ackermann's function, its inverse $\alpha(n)$ grows extremely slowly, and is at most 4 for any practical value of n.

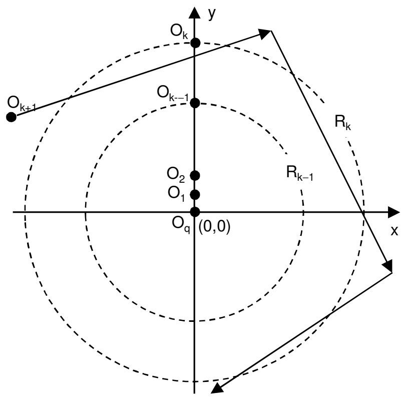
Proof idea: The proof is inspired by Corollary 3.4 in [8]. The Corollary implies that the length of the k-level in an arrangement of n piecewise linear functions, where each function has m linear pieces, is $O(nm\alpha(n))$. It can be shown that Corollary 3.4 in [8] holds for piecewise pseudo-linear functions as well. A piecewise pseudo-linear function is one that has one-by-one connected pseudo-linear pieces; two pseudo-linear pieces intersect at most once. We then cut each parabola into two pseudo-linear pieces using its axis of symmetry. Thus the $O(nm\alpha(n))$ bound follows. □

The following proposition gives a lower bound of $\beta_k(n)$ in the piecewise linear model.

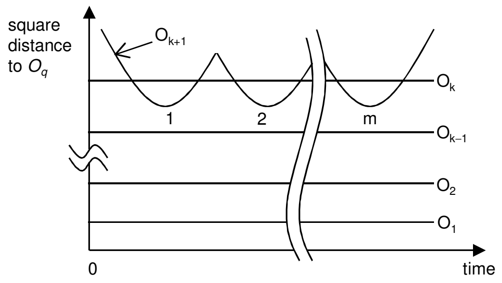
Proposition 2.5: *Assume that the query object and the data objects all move piecewise linearly, where each object can have at most m linear pieces. Then for any constant k* $\beta_k(n) \geq 2m(n-k)+1$.

Proposition 2.5 tells us that the bound of $O(nm\alpha(n))$ is very tight because it only adds an almost-constant factor to the lower bound.

Proof: For the motion in Figure 2.5(a) the curves arrangement is as in Figure 2.5(b), and the details are as in Proposition 2.2. □



(a) Configuration in the motion space



(b) Arrangement in the Time-SquareDistance space

Figure 2.5. The auxiliary figure for the proof of Proposition 2.5.

Corollary 2.6. *Assume that the query object and the data objects all move piecewise linearly, where each object can have at most m linear pieces.* $\beta_k(n)=O(nm\alpha(n))$ and $\beta_k(n)=\Omega(nm)$.

3. Query Processing

In subsection 3.1 we discuss off-line processing and in subsection 3.2 we discuss on-line processing.

3.1 Off-line Processing

When k=1, the processing of a CkNN query reduces to the construction of the lower envelope in the arrangement of the square-distance functions of the n data objects (i.e., the $d_i^2(t)$'s). In the linear model, the lower envelope can be constructed using a simple and efficient divide-and-conquer algorithm in O($n$log$n$) time (see Theorem 2.6 in [8]). This algorithm divides the objects database into two subsets S_1 and S_2, each of size at most $\lceil n/2 \rceil$, computes the lower envelopes of S_1 and S_2 recursively, and merges the two envelopes to obtain the lower envelope of the objects database.

The same paradigm can be used to compute the lower envelope in the piecewise linear model. In this case, merging the lower envelopes of S_1 and S_2 still takes time proportional to the sum of $|S_1|$ and $|S_2|$ as described in [8]. Due to Theorem 2.4, $|S_1|+|S_2|=O(nm\alpha(n))$. Thus the complexity in the piece-wise linear model is $T(n)=2T(\frac{n}{2})+O(nm\alpha(n))$, which is O($nm\alpha(n)logn$).

When k>1, the answer pairs can be computed in the same fashion. Details are omitted due to space limitations.

3.2 On-line Processing

In subsection 3.2.1 we describe an existing on-line processing data structure called *kinetic tournament* and discuss its shortcoming. From 3.2.2 to 3.2.5 we present our on-line processing data structure which overcomes this shortcoming.

3.2.1 Kinetic Tournament

Consider the case in which k=1 and all the objects move linearly. The query processing in this case translates to the problem of dynamically maintaining the lower envelope of a set of parabolas in the Time-SquareDistance space. In computation geometry this problem is known as *kinetic minimum maintenance*. The authors of [5] introduce a solution to this problem, which is called a *kinetic tournament*. The idea is to use a simple divide-and-conquer strategy. The algorithm partitions the data objects into two approximately equal-sized groups (arbitrarily), and recursively maintains the minimum of each group. A final comparison at the top level yields the global minimum. If viewed from the bottom up, this is exactly a tournament for computing the global winner. Each comparison of two data objects is associated with an event that describes when this comparison will be violated in the future. When a violation happens, the new winner is produced and is percolated up the tournament tree, until it is either defeated or declared the global winner.

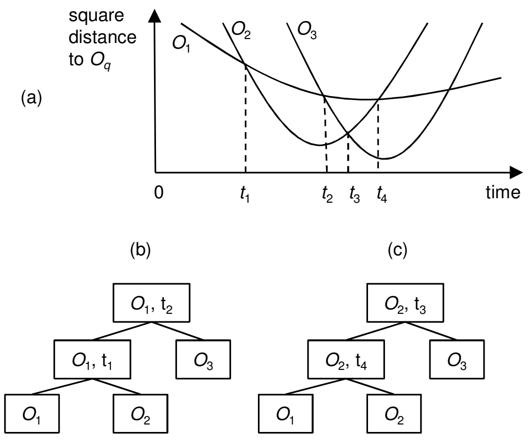


Figure 3.1. Kinetic tournament

Figure 3.1 gives an example of kinetic tournament. 3.1(a) shows three data objects O_1, O_2, and O_3 in the Time-SquareDistance space. 3.1(b) shows the tournament tree constructed at time 0. Two events are scheduled at this moment: (i) O_1 crosses O_2 at time t_1; (ii) O_1 crosses O_3 at time t_2. Since $t_1<t_2$, the O_1-crossing-O_2 event is triggered earlier, at t_1. In response to this event, the winner between O_1 and O_2 is changed, and the new winner is percolated up (see 3.1(c)). The O_1-crossing-O_3 event is eliminated, and two new events are scheduled to be triggered, at time t_4 and t_3 respectively.

Now examine the complexity of kinetic tournament on processing each event. The processing of an event involves the percolation of the new winner. Because a tournament tree is balanced, the percolation visits at most O(log$n$) nodes. A visit to each of these nodes may result in elimination of an existing event and creating of a new event. If we use a priority queue to store the relevant events (at most O(n)), then the elimination or creation of an event takes O(log$n$) time. Thus, the percolation takes O(log^2n) time.

As analyzed above, in kinetic tournament, O(log$n$) time is spent on accessing the event queue when processing an event. Indeed, kinetic tournament schedules an event for each internal node in the tournament tree, even though some of these events will never be triggered. In the Figure 3.1 example, the O_1-crossing-O_3 event is never triggered because it becomes useless after O_2 claims the global winner. In fact we only need to schedule one event which is the earliest time at which the current global winner may be replaced. In this way we eliminate the cost of accessing the event queue. This is the main idea of our on-line processing data structure which is presented in the following subsection.

3.2.2 The Object Heap Data Structure

Our kinetic data structure is called *object heap* and is organized as follows. As before, we assume that we have a query object O_q and a set S of data objects such that $O_q \notin S$. We assume that there is a unique id with each object in S. Let $t\geq 0$ be a time instance. An object heap H over the set S at time t is a full binary tree such that each node x in it stores two items *x.object* and *x.time* satisfying the following conditions. First, for any internal node x, let *ObjectSet*(x) denote the set of objects stored at the leaves of the sub-tree rooted at x. For a leaf node x, *x.time*=∞. For an internal node x, *x.time*≥t and *x.object* is the closest object to O_q during the time interval $[t, x.time]$ among all the objects in *ObjectSet*(x). For any internal node x, we let *x.left*, *x.right* represent respectively the left and right child of x. Also, *x.parent* represents the parent of x. Figure 3.2 shows the object heap at time 0 for the arrangement of Figure 3.1(a). From the heap we see that O_1 is the closest object until t_1.

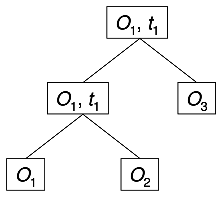


Figure 3.2. The object heap at time 0 for the arrangement of Figure 3.1(a).

An object heap is constructed using the procedure *BuildObjectHeap*(S,O_q,*Ctime*) given below. The procedure takes the set of objects S, the query object O_q and time *Ctime* and builds an object heap at time *Ctime*. *CloserObject*(O_1,O_2,t), *NextTime*(O_1,O_2,t) are functions that take two objects O_1, O_2 and time t as parameters and returns values as defined below. *CloserObject*(O_1,O_2,t) returns the object closest to O_q among the two objects O_1, O_2 at time t. *NextTime*(O_1,O_2,t) returns the earliest time $t'>t$ such that *CloserObject*(O_1,O_2,t') is different from *CloserObject*(O_1,O_2,t), i.e. when one of the two objects becomes closer than the other; if no such t' exists then it returns ∞. The BuildObjectHeap procedure first initializes the heap so that each object in S is stored at a leaf node. Then it processes nodes in decreasing order of their levels. For each internal node, it sets its object to be the closest object to O_q at time *Ctime* among the objects stored at its two children; it sets its time to be the minimum of the time values in its children and the next time when one of them is going to overtake the other to become closer to O_q. In this procedure, min is a function that returns the minimum of three values.

```
BuildObjectHeap($S$,$O_q$,$Ctime$)
Initialize();
For each node $x$ in decreasing values of the level of $x$
    If $x$ is an internal node
        $O_1$ := $x.left.object$;
        $O_2$ := $x.right.object$;
        $x.object$ := $CloserObject(O_1, O_2, Ctime)$;
        $x.time$ := min($x.left.time$, $x.right.time$, $NextTime(O_1,O_2,Ctime)$)
```

It is fairly obvious to see that the worst case time complexity of *BuildObjectHeap*() is $O(n)$ where n is the number of objects in S. Assuming that the start time is zero, the initial object heap is built by invoking the above procedure with the parameter value of *Ctime* set to zero. It is to be noted that if r is the root node of the object heap then *r.object* is the closest object until the time *r.time*. Thus, at time *r.time*, the object heap needs to be readjusted. We call such a readjustment as an *implicit update*.

3.2.3 Algorithm for Implicit Updates

We classify the internal nodes of an object heap as *cross nodes* and *minimal nodes* as follows. An internal node x is called a cross node, if *x.time* is less than both *x.left.time* and *x.right.time*. All internal nodes other than cross nodes are called minimal nodes. If x is a cross node and O_1,O_2 are the objects at its children and *x.object*=O_1, then up to the time *x.time* object O_1 is the closest to O_q among objects in *ObjectSet*(x) and at time *x.time*, O_2 will be the closest object among these objects. If x is a minimal node, then there is a child y of x such that *x.time*=*y.time*.

Roughly speaking, the algorithm for implicit update works as follows. If the root node r is a cross node then it sets *r.object* to be the object, among its children, that is different from the current value of *r.object* and sets *r.time* to be the minimum of the times at its children. If the root node r is a minimal node then it traverses along a path of internal nodes $x_1,\ldots,x_g$ such that $x_1=r$, x_g is a cross node and all nodes $x_1,\ldots,x_{g-1}$ are minimal nodes and the *time* value on all these nodes is *r.time*. After reaching x_g, it updates the object and time value at this node, and retraces the path back to the root node updating the object and time values on each of these nodes appropriately. The object heap of Figure 3.3 results when we perform implicit update on the object heap of Figure 3.2 at time t_1. The modified object heap shows that object O_2 is the closest object from time t_1 up to time t_2.

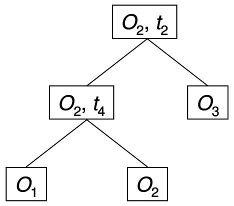


Figure 3.3. The object heap resulting from an implicit update on the object heap of Figure 3.2.

The above algorithm is accomplished by the recursive procedure *ImplicitUpdate*(x) given below. This procedure acts as follows. If x is a cross node and O_1,O_2 are objects stored in its two children and *x.object*=O_1 then it sets *x.object* to O_2 and sets *x.time* to be the minimal of the times of its children and returns. Otherwise, it recursively invokes the procedure on a child y such that *y.time*=*x.time*. This recursive invocation may change the object and time at node y. Thus, when invocation on y returns, it reevaluates which of the objects at its children is the closest object and sets *x.object* to that object and resets *x.time* and returns. Note that the actual implicit update is carried out by invoking this procedure with the root r at time *r.time*.

```
ImplicitUpdate(x)
    $y$ := $x.left$; $z$ := $x.right$;
    If $x.time<y.time$ and $x.time<z.time$ (//$x$ is a cross node)
        If $x.object = y.object$
            $x.object$ := $z.object$;
        Else $x.object$ := $y.object$;
        $x.time$ := $NextTime(y,z,x.time)$;
        return;
    If $x.time = y.time$
        $ImplicitUpdate(y)$;
    If $x.time = z.time$
        $ImplicitUpdate(z)$;
    $x.object$ := $CloserObject(y,z,x.time)$;
    $x.time$ := min($y.time$, $z.time$, $NextTime(y,z,x.time)$);
    return
```

Note that for an object heap at time t, with root r, $t<r.time$, *r.time* is less than or equal to *x.time* for every node x in it, and the

structure satisfies the heap property until *r.time*. At time *r.time*, we invoke the procedure *ImplicitUpdate*(r). It is fairly straightforward to show that after invocation of *ImplicitUpdate*(r) at time *r.time*, the structure continues to be an object heap.

We make the assumption, called *distinct-sibling-times* assumption, that for every pair of internal nodes y, z, that are siblings, $y.time \neq z.time$. At the end of this subsection we will discuss what happens when this assumption does not hold. Under the distinct-sibling-times assumption, when *ImplicitUpdate*(r) is invoked, the algorithm will travel along a single path in the object heap. Hence, the complexity of executing *ImplicitUpdate*(r) is $O(h)$ where h is the height of the object heap. Since $h \leq \log n+1$, we see that the complexity of execution of the ImplicitUpdate is $O(\log n)$. Furthermore, at any point in time there is only one event to monitor which is the expiration of the root node. Now the following theorem shows that there can be at most $O(n\log n)$ implicit updates, i.e., after at most $n\log n$ updates, $r.time=\infty$ and hence the object heap does not need any further implicit updates. Thus the object heap is efficient in the sense that the number of implicit updates is only $\log n$ larger than the maximum number of times that the answer set may change.

Theorem 3.1: *Let T be an object heap at time 0 with root r, then after at most 2nlogn updates, r.time=∞.*

Proof: Observe that whenever an implicit update is performed, the object heap is traversed starting from the root r until a cross node x is reached. At the cross node x, the value of *x.object* and *x.time* are updated. It should be easy to see that the time when this update is carried out, i.e., at this time $x.time=r.time$, and at this time the closest object to O_q among *ObjectSet*(x) changes and hence it is a split point for this set of objects. Since there are at most $2n_1-1$ split points for *ObjectSet*(x) where n_1 is the number of objects in *ObjectSet*(x), we see that the number of implicit updates on T that stop at x is bounded by $2n_1-1$. Let l be the level number of x. Now the total number of implicit updates that stop at a node at level l is bounded by twice the number of objects stored in the sub-trees whose root is at level l. Since the number of all such objects is the total number of objects, we see that the total number of implicit updates that stop at a level l node is bounded by $2n$. Since there are $\log n$ levels, we see that the total number of implicit updates is bounded by $2n\log n$. Clearly, after at most such implicit updates $r.time=\infty$. □

Since there are $O(n\log n)$ implicit updates and each such update takes time $O(\log n)$, we see that the cumulative complexity of performing these updates is $O(n\log^2 n)$.

If the distinct-sibling-times assumption does not hold then an invocation of *ImplicitUpdate*(r) may travel on multiple paths, and hence the complexity may be higher than $O(h)$; however, its complexity can be shown to be only $O(n)$. Furthermore, the cumulative complexity over a number of invocations until $r.time=\infty$, can easily be shown to be still $O(n\log^2 n)$.

3.2.4 Explicit Updates and Piecewise Linear Model

Now we show how explicit updates can be implemented efficiently using object heap. We consider three types of updates, namely insertion of a new object, deletion of an existing object, and velocity-vector change of an existing object.

Addition. Assume that we have an object heap H with root node r storing n objects. In this heap which has $2n-1$ nodes, each leaf node will have a sibling. Assume that the new object to be inserted is O' at time *Ctime* which is the current time. We can assume that *Ctime* < *r.time*. We allocate two new nodes, say y_1 and y_2. Let x be the first leaf node in H, i.e., the leftmost leaf node at the lowest numbered level (note that if the height of H is h, then there can be leaf nodes both at level $h-1$ and h). Also let O'' be the object in node x. Now we add the two nodes y_1, y_2 as children of x and place objects O' and O'' in these two nodes. This makes x as an internal node. Now we perform, the following operation, called *float*(z) with argument x. This operation *float*(z), is a recursive operation, which acts as follows. It sets *z.object* to the object among its children that is closest to the query object O_q at time *Ctime*, i.e., to the object given by the function *CloserObject*(O_1, O_2, *Ctime*) and sets *z.time* to be the value given by the function *NextTime*(O_1, O_2, *Ctime*) where O_1, O_2 are the objects *z.left.object* and *z.right.object*, respectively. After this, if z is the root then it stops; else it recursively invokes *float* on the parent of z. It is not difficult to see that the resulting structure H is an object heap at time *Ctime* containing the $n+1$ objects. It's complexity is clearly $O(\log n)$.

Consider the object heap of Figure 3.2. Assume that $t_1>1$. When we insert a new object O_4 at time 1, the following heap results (see Figure 3.4). Note that O_1 is still the closest object till t_1. Also O_3 is the closer of O_3 and O_4 until t_5. Here t_5 is assumed to be greater than t_1.

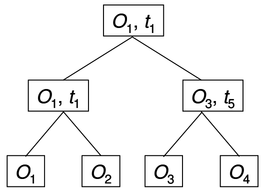


Figure 3.4. The object heap resulting from an explicit update on the object heap of Figure 3.2.

Deletion. Now we describe the deletion of an object from H. Assume that the object to be deleted is the object in the last leaf node, i.e. the right most node in the lowest level (i.e., at level h). Let y_2 be this node and let its sibling be y_1 and let x be their parent. It is not difficult to see that x is the last internal node of H. We delete both the nodes y_1 and y_2 and place the object in y_1 in the node x. By this, H loses two nodes and x becomes a leaf node. Now, we invoke *float* operation on the parent z of x. After this, the resulting structure will satisfy the object heap property. Clearly, the complexity of this operation is $O(\log n)$.

Now consider the deletion of an object O' in an arbitrary leaf node y. To do this, we first delete the object O'' in the last leaf node of H using the above procedure. In the resulting object heap, let y' be the leaf node containing object O'' It is possible that y' is different from y. This occurs if y was the last but one leaf node in H. Now we replace object O' in y' by O'' and perform the *float* operation starting from the parent of y'. It should be easy to see that the resulting structure is an object heap containing all objects of H excepting O'. Clearly, the complexity of this whole procedure is $O(\log n)$.

Velocity-vector Change. When the speed and/or the direction of an existing object O′ changes, the square-distance function $d^2_{O'}(t)$ changes. In this case, find the leaf node y that contains O' and update $d^2_{O'}(t)$. After this, we perform the *float* operation starting from the parent of y. The complexity of this update is also $O(\log n)$.

Now let us consider the piecewise linear model in which each object can change its velocity-vector at most m times and we analyze the cumulative complexity of our on-line algorithm for maintaining the 1NN set and handling velocity-vector changes. The cumulative complexity is bounded as follows. Since the complexity for handling each velocity-vector update is $O(\log n)$, the total complexity of these updates is $O(nm\log n)$. Note that implicit updates are carried out between velocity-vector changes as before, as and when needed. Now, we bound the cumulative complexity of implicit updates. Observe that each object has a piecewise linear motion with at most m pieces throughout the time. Using Theorem 2.4, it is not difficult to see that the total number of implicit updates that stop at a particular internal node x is $O(n_1 m\alpha(n_1))$ where n_1 is the number of objects stored in the sub-tree rooted at x. Using the same argument as given in Theorem 3.1 for the linear case, we see that the total number of implicit updates that stop at an internal node at some level say l is $O(nm\alpha(n))$. Since there are at most $\log n$ levels, we see that the total number of implicit updates is $O(nm\alpha(n)\log n)$. Since the cost of each implicit update is $O(\log n)$, we see that the cumulative time complexity of implicit updates is $O(nm\alpha(n)\log^2 n)$. This is also the cumulative complexity of all the updates both implicit as well as explicit.

3.2.5 Extension to k>1

For k>1, each internal node x in the object heap stores two items *x.set* and *x.time* such that *x.set* is the set of the k closest objects to O_q up to time *x.time* among all the objects in *ObjectSet*(x). To achieve this goal, we modify the procedure *BuildObjectHeap*(S,O_q,*Ctime*) as follows. For each internal node x, x.set is set to be the kNN set at time *Ctime* among the objects stored at its two children; x.time is set to be the minimum of the time values in x's children and the next time when the kNN set among the objects stored at x's two children is going to change.

Now consider how to process an implicit update that is triggered at time t. Similar to k=1 case, the implicit update algorithm traverses along a path of internal nodes $x_1,\ldots,x_g$ such that $x_1=r$, x_g is a cross node and all nodes $x_1,\ldots,x_{g-1}$ are minimal nodes and the *time* value on all these nodes is *r.time*. After reaching x_g, it sets x_g*.set* to be the kNN set at time t among the objects stored at x_g's two children and sets x_g.time to be the minimum of the time values in x_g's children and the next time when x_g*.set* is going to change.

Explicit updates can be extended in the same fashion. It is not difficult to see that, when k is a constant, the complexity of our on-line algorithm for k>1 is the same as that for k=1.

4. Conclusion and Future Work

The results developed in this paper are summarized in Table 1.1 in section 1. In the rest of this section we discuss the future work.

C*k*NN with location uncertainty. Due to continuous motion, communication delays and positioning errors, it is normal that there is uncertainty associated with the locations of moving objects. In this case, a C*k*NN query may ask for objects that are *possibly* the k nearest neighbors, or the objects that are *definitely* the k nearest neighbors. In terms of the location uncertainty model, the uncertainty region at any point in time can be a circle, an ellipse, a line segment, a fan area, etc. We will study the complexity and processing of C*k*NN queries under various query semantics and various location uncertainty models.

Indexing in the Time-SquareDistance space. In the existing literature, indexes are built in the motion space. Would it be more efficient if we build indexes in the Time-SquareDistance? That is, we index the distance curves using a spatial indexing structure such as a quadtree. A quick observation is that the distance curves are invariable in time unless there are updates. On the other hand, the locations of moving objects change continuously in the motion space even if there are no updates. Thus intuitively an index structure in the Time-SquareDistance space would be more stable than one in the motion space.

5. References

[1] J. Hershberger. Finding the upper envelope of n line segments in O(nlogn) time. *Information Processing Letters*, 33(4):169-174, 1989.

[2] K. L. Clarkson and P. W. Shor. Applications of Random Sampling in Computational Geometry, II. *Discrete and Computational Geometry*, 4(1):387-421, 1989.

[3] Y. Li, J. Yang, and J. Han: Continuous K-Nearest Neighbor Search for Moving Objects. SSDBM 2004: 123-126.

[4] G. Iwerks, H. Samet, K. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. *VLDB*, 2003.

[5] J. Basch and L. J. Guibas. Data Structures for Mobile Data. Journal of Algorithms, 31:1-28, 1999.

[6] P. K. Agarwal and M. Sharir. Arrangements and Their Applications. In Handbook of Computational Geometry, edited by J. R. Sack and J. Urrutia, North-Holland, 2000.

[7] G. D. da Fonseca and C. M. H. de Figueiredo. Kinetic heap-ordered trees: tight analysis and improved algorithms. Information Processing Letters 85(3):165-169, 2003.

[8] P. K. Agarwal and M. Sharir. Davenport-Schinzel Sequences and Their Geometric Applications. In Handbook of Computational Geometry, edited by J. R. Sack and J. Urrutia, North-Holland, 2000.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. McGrwHill Publishers, 2001.