In Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD06), Chicago (IL), U.S.A., 2006

Efficient Reverse k-Nearest Neighbor Search in Arbitrary Metric Spaces

Elke Achtert, Christian Böhm, Peer Kröger, Peter Kunath,
Alexey Pryakhin, Matthias Renz

Institute for Computer Science
University of Munich
Oettingenstr. 67, 80538 Munich, Germany
{achtert,boehm,kroegerp,kunath,pryakhin,renz}@dbs.ifi.lmu.de

ABSTRACT

The reverse k-nearest neighbor (RkNN) problem, i.e. finding all objects in a data set the k-nearest neighbors of which include a specified query object, is a generalization of the reverse 1-nearest neighbor problem which has received increasing attention recently. Many industrial and scientific applications call for solutions of the RkNN problem in arbitrary metric spaces where the data objects are not Euclidean and only a metric distance function is given for specifying object similarity. Usually, these applications need a solution for the generalized problem where the value of k is not known in advance and may change from query to query. However, existing approaches, except one, are designed for the specific R1NN problem. In addition — to the best of our knowledge — all previously proposed methods, especially the one for generalized RkNN search, are only applicable to Euclidean vector data but not for general metric objects. In this paper, we propose the first approach for efficient RkNN search in arbitrary metric spaces where the value of k is specified at query time. Our approach uses the advantages of existing metric index structures but proposes to use conservative and progressive distance approximations in order to filter out true drops and true hits. In particular, we approximate the k-nearest neighbor distance for each data object by upper and lower bounds using two functions of only two parameters each. Thus, our method does not generate any considerable storage overhead. We show in a broad experimental evaluation on real-world data the scalability and the usability of our novel approach.

1. INTRODUCTION

A reverse k-nearest neighbor (RkNN) query returns the data objects that have the query object in the set of their k-nearest neighbors. It is the complimentary problem to that of finding the k-nearest neighbors (kNN) of a query object and has been studied extensively in the past few years for Euclidean data ([7], [10], [12], [9]). The goal of a reverse k-nearest neighbor query is to identify the "influence" of a query object on the whole data set. For example, consider a decision support system with the goal of choosing the location for a new store. Given several choices for the new location, the strategy is to pick the location that can attract the most customers. A RkNN query would return the customers who would be likely to use the new store because of its geographical proximity. The RkNN problem appears in many practical situations such as geographic information systems (GIS), traffic networks, or molecular biology where the database objects are general metric objects rather than Euclidean vectors. In these application areas, the database objects are polygons or sequences and an arbitrary metric distance function is defined on these objects to evaluate object similarity. For example, the increasing progress in telecommunication techniques and location tracking systems like GPS, extends significantly the scope for location based service applications. Beside the nearest neighbor search, the reverse nearest neighbor query is one of the most important query types for location based services, e.g. in applications where stationary or moving objects agree to provide some kind of service to each other. Objects or individuals usually want to request services from their nearest neighbors. Conversely, the objects or individuals which provide some services may be interested in the number of expected service requests, or which objects could be interesting candidates to provide the services and maybe where they are actually located. For example, all filling stations of one company in a town want to provide their own advertisements to all cars of which they are the nearest neighbor.

Another example consisting of pizza restaurants and potential customers is depicted in Figure 1. To keep down costs when carrying out an advertising campaign, it would be profitable for a restaurant owner to send menu cards only to those customers which have his restaurant as the nearest pizza restaurant.

Usually, the objects are nodes in a traffic network. Instead of the Euclidean distance, graph algorithms like Dijkstra have to be applied. Another important application area of RkNN search in general metric databases is molecular biol-

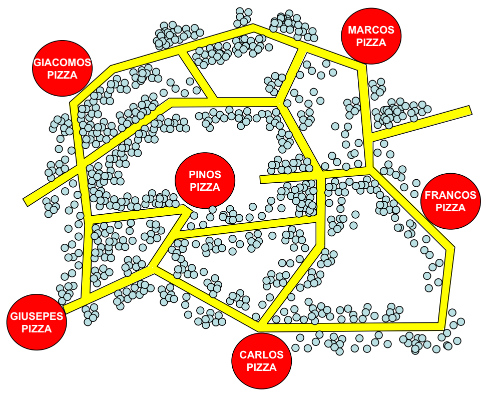


Figure 1: RkNN query example consisting of pizza restaurants (large dots) and potential customers (small dots).

ogy. Here, the detection of new sequences call for efficient solutions of the general RkNN problem in sequence databases. These databases usually contain a large number of biological sequences. Researchers all over the world steadily detect new biological sequences that need to be tested on originality and interestingness. To decide about the originality of a newly detected sequence, the RkNNs of this sequence are computed and examined. Usually, in this context, the similarity of biological sequences is defined in terms of a metric distance function such as the edit distance or the Levenstein distance. More details on this application of RkNN search in metric databases can be found in [5].

Although the reverse k-nearest neighbor problem is the complement of the k-nearest neighbor problem, the relationship between kNN and RkNN is not symmetric and the number of the reverse k-nearest neighbors of a query object is not known in advance. A naive solution of the RkNN problem requires $O(n^2)$ time, as the k-nearest neighbors of all of the n objects in the data set have to be found. Thus, more efficient algorithms are required.

As we will discuss in Section 3 the well-known methods for reverse k-nearest neighbor search can be categorized into two classes, the hypersphere-approaches and the Voronoi-approaches. All these approaches are only designed for Euclidean vector data but cannot be applied to general metric objects. Hypersphere-approaches such as [12] extend a multidimensional index structure to store each object along with its nearest neighbor distance and, thus, actually store hyperspheres rather than points. In contrast, the Voronoi-approaches such as [11] store the objects in conventional multidimensional index structures without any extension and compute a Voronoi cell during query processing. In principle, the possible performance gain of the search operation is much higher in the hypersphere approaches while only Voronoi-approaches can be extended to the reverse k-nearest neighbor problem with an arbitrary $k > 1$ in a straightforward way. However, this approach is not extendable to general metric spaces since it relies on explicitly computing Voronoi hyperplanes which are complex to compute in arbitrary metric spaces.

In this paper, we propose an efficient solution based on the hypersphere approach for the RkNN problem with an arbitrary k not exceeding a given threshold parameter k_{max} for general metric objects. The idea is not to store the true nearest neighbor distances for each k of every object separately but rather to use suitable approximations of the set of nearest neighbor distances. This way, we approximate both, the kNN distances of a single object stored in the database as well as the k-nearest neighbor distances of the set of all objects stored in a given subtree of our metric index structure. To ensure the completeness of our result set (i.e. to guarantee no false dismissals) we need a conservative approximation which never under-estimates any k-nearest neighbor distance but also approximates the true k-nearest neighbor distances of a single object or a set of objects with minimal approximation error (in a least squares sense). To reduce the number of candidates that need to be refined, we additionally store a progressive approximation of the kNN distances which is always lower or equal to the real kNN distances. Thus, only objects that have a kNN distance to a given query object which is between the lower bound (progressive approximation) and the upper bound (conservative approximation) need to be refined. We will demonstrate in Section 4 that the k-nearest neighbor distances follow a power law which can be exploited to efficiently determine such approximations. Our solution requires a negligible storage overhead of only two additional floating point values per approximated object. The resulting index structure called MRkNNCoP (Metric reverse kNN with conservative and progressive approximations)-Tree can be based on any hierarchically organized, tree-like index structure for metric spaces. In addition, it can also be used for Euclidean data by using a hierarchically organized, tree-like index structure for Euclidean data.

The remainder of this paper is organized as follows: Section 2 introduces preliminary definitions and Section 3 discusses related work. In Section 4 we introduce our new index structure, the MRkNNCoP-Tree, for efficient reverse k-nearest neighbor search in general metric spaces in detail. Section 5 contains an extensive experimental evaluation. Section 6 concludes the paper and proposes some directions of future work.

2. PROBLEM DEFINITION

Since we focus on the traditional reverse k-nearest neighbor problem, we do not consider recent approaches for related or specialized reverse nearest neighbor tasks such as the bichromatic case, mobile objects, etc.

In the following, we assume that $\mathcal{D}$ is a database of n metric objects, $k \leq n$, and *dist* is a metric distance function on the objects in $\mathcal{D}$. The set of *k-nearest neighbors* of an object q is the smallest set $NN_k(q) \subseteq \mathcal{D}$ that contains at least k objects from $\mathcal{D}$ such that

$$\forall o \in NN_k(q), \forall \hat{o} \in \mathcal{D} - NN_k(q) : dist(q,o) < dist(q,\hat{o}).$$

The object $p \in NN_k(q)$ with the highest distance to q is called the *k-nearest neighbor* (kNN) of q. The distance $dist(q,p)$ is called *k-nearest neighbor* distance (kNN distance) of q, denoted by $nndist_k(q)$.

The set of *reverse k-nearest neighbors* (RkNN) of an object q is then defined as

$$RNN_k(q) = \{p \in \mathcal{D} \mid q \in NN_k(p)\}.$$

The naive solution to compute the reverse k-nearest neighbor of a query object q is rather expensive. For each object $p \in \mathcal{D}$, the k-nearest neighbors of p are computed. If the k-nearest neighbor list of p contains the query object q, i.e.

$q \in NN_k(p)$, object p is a reverse k-nearest neighbor of q. The runtime complexity of one query is $O(n^2)$. It can be reduced to an average of $O(n \log n)$ if an index such as the M-Tree [4] (or, if the objects are feature vectors, the R-Tree [6] or the R*-Tree [2]) is used to speed-up the nearest neighbor queries.

3. RELATED WORK

All previously proposed methods are only applicable to Euclidean vector data, i.e. $\mathcal{D}$ contains feature vectors of arbitrary dimensionality d ($\mathcal{D} \in \mathbb{R}^d$).

In [7] an index structure called RNN-Tree is proposed for reverse 1-nearest neighbor search. The basic idea is that if the distance of an object p to the query q is smaller than the 1-nearest neighbor distance of p, p can be added to the result set. This saves a nearest neighbor query w.r.t. p. Thus, the RNN-Tree stores for each object p the distance to its 1-nearest neighbor, i.e. $nndist_1(p)$. In particular, the RNN-Tree does not store the data objects itself but for each object p a sphere with radius $nndist_1(p)$. Thus, the data nodes of the tree contain spheres around objects rather than the original objects. The spheres are approximated by minimal bounding rectangles (MBRs). Since the tree suffers from a high overlap of the data MBRs and, thus, from a high overlap of the directory MBRs, [7] propose to use two trees: (1) a traditional R-Tree-like structure for nearest neighbor search (called NN-Tree) and (2) the RNN-Tree for reverse 1-nearest neighbor search.

The RdNN-Tree [12] extends the RNN-Tree by combining the two index structures (NN-Tree and RNN-Tree) into one common index. It is also designed for reverse 1-nearest neighbor search. For each object p, the distance to p's 1-nearest neighbor, i.e. $nndist_1(p)$ is precomputed. In general, the RdNN-Tree is a R-Tree-like structure containing data objects in the data nodes and MBRs in the directory nodes. In addition, for each data node N, the maximum of the 1-nearest neighbor distance of the objects in N is aggregated. An inner node of the RdNN-Tree aggregates the maximum 1-nearest neighbor distance of all its child nodes. A reverse 1-nearest neighbor query is processed top down by pruning those nodes N where the maximum 1-nearest neighbor distance of N is greater than the distance between query object q and N, because in this case, N cannot contain true hits anymore. Due to the materialization of the 1-nearest neighbor distance of all data objects, the RdNN-Tree needs not to compute 1-nearest neighbor queries for each object.

A geometric approach for reverse 1-nearest neighbor search in a 2D data set is presented in [10]. It is based on a partition of the data space into six equi-sized units where the gages of the units cut at the query object q. The nearest neighbors of q in each unit are determined and merged together to generate a candidate set. This considerably reduces the cost for the nearest-neighbor queries. The candidates are then refined by computing for each candidate c the nearest neighbor; if this nearest neighbor is q then c is added to the result. Since the number of units in which the candidates are generated increases exponentially with d, this approach is only applicable for 2D data sets.

An approximative approach for reverse k-nearest neighbor search in higher dimensional space is presented in [9]. A two-way filter approach is used to generate the results. However, the method cannot guarantee the completeness of the result but trade off a loss of accuracy for a gain of performance.

Recently, in [11] the first approach for reverse k-nearest neighbor search was proposed, that ensures complete results. The method uses any hierarchical tree-based index structure such as R-Trees to compute a nearest neighbor ranking of the query object q. The key idea is to iteratively construct a Voronoi cell around q from the ranking. Objects that are beyond k Voronoi planes w.r.t. q can be pruned and need not to be considered for Voronoi construction. The remaining objects must be refined, i.e. for each of these candidates, a k-nearest neighbor query must be launched.

3.1 Contributions

Most recent methods for the reverse k-nearest neighbor search suffer from the fact that they are only applicable to $k = 1$ or at least a fixed value of k. The most generic approach is that of using Voronoi cells [11]. Since it does not rely on precomputed kNN distances, it can handle queries with arbitrary values for k. However, the approach proposed in [11] relies on the computation of the Voronoi (hyper-)plane which only exists in Euclidean vector spaces. In general metric spaces, the hyperplanes that separate the Voronoi cells are hard to compute. So far, there exist only methods for R-tree like index structures. Thus, these approaches cannot be extended for metric databases.

To the best of our knowledge, this paper is the first contribution to solve the generalized RkNN search problem for arbitrary metric objects. In particular, our method provides the following new features:

1. It can be applied to general metric objects, i.e. databases containing any type of complex objects as long as a metric distance function is defined on these objects.

2. It is applicable to the generalized RkNN problem where the value of k is specified at query time.

4. kNN DISTANCE APPROXIMATIONS FOR RkNN SEARCH

As discussed above, the Voronoi approach is not applicable to general metric spaces because in these spaces, the explicit computation of hyperplanes is not possible. As a consequence, we want to base our generic index for reverse kNN search on the ideas of the RdNN-Tree which is only applicable for Euclidean vector data and the specialized case of $k = 1$. The generalizations to enable the application of metric objects is rather straightforward. Instead of using an Euclidean index structure such as the R-Tree [6] or R*-Tree [2] we can use a metric index structure such as the M-Tree [4]. However, in order to solve the generalized RkNN problem where the value for k could be different from query to query is much more challenging. The key idea of the RdNN-Tree is to precompute the kNN distance for an *a priori* fixed value[1] of k for each object. If a query object q has a larger distance to an object o than the kNN distance of o, we can safely drop o. Otherwise, object o is a true hit. However, as indicated above, this approach requires to specify the value of k in advance.

In general, there are two possibilities to generalize the RdNN-Tree in order to become independent of k. First, we could store the kNN distances of each object for each $k \leq k_{max}$, where k_{max} is the maximum value of k depending on the application. Obviously, this approach results in

[1]originally for $k = 1$

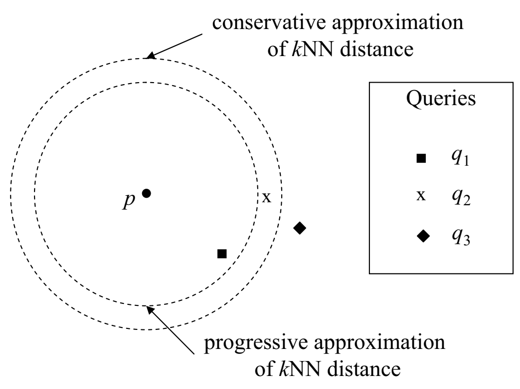

conservative approximation of kNN distance
Queries
q_1
q_2
q_3
p
x
progressive approximation of kNN distance

Figure 2: Using conservative and progressive approximation for RkNN search.

high storage costs and, as a consequence, in a very large directory of the index tree. Thus, the response times will be significantly enlarged. Second, we can conservatively approximate the kNN distances of each object by one distance, i.e. the k_{max}NN distances. However, if k_{max} is considerably higher than the k needed in a given query, the pruning power of the k_{max}NN distance in order to identify true drops will obviously be decreased dramatically. As a consequence, the response time will also be significantly increased due to this bad kNN distance estimation and the resulting poor pruning. In addition, for each object o where $dist(o,q) \leq dist_{k_{max}}(o)$ a refinement is needed since it cannot be ensured that $dist(o,q) < dist_k(o)$, i.e. for each of these objects o another kNN query needs to be launched.

Here, we propose a novel index structure that overcomes these problems by approximating the kNN distances of each object for all $k \leq k_{max}$ using a function. We store for each object $o \in \mathcal{D}$ such a function that conservatively approximates the kNN distances of o for any $k \leq k_{max}$. The approximation is always greater or equal to the real distances. This conservative approximation allows to identify objects that can be safely dropped because they cannot be true hits. For the remaining objects, we need a refinement step inducing a kNN query for each candidate.

In fact, we can even further reduce the number of candidates, i.e. the number of necessary kNN queries for refinement by storing also a progressive approximation of the kNN distances of each $o \in \mathcal{D}$ for any $k \leq k_{max}$. The progressive approximation is always lower or equal to the real distances and, thus, enables to identify true hits. Only objects that have a distance to the query q less or equal than the conservative approximation and greater than the progressive approximation need to be refined, i.e. induce a kNN query.

The idea of using conservative and progressive approximations is illustrated in Figure 2. If q_1 is the query object, p is a true hit and need not to be refined because the progressive approximation ensures that $dist(p,q_1) < nndist_k(p)$. If q_3 is the query object, p can be dropped safely because the progressive approximation ensures that $dist(p,q_3) > nndist_k(p)$. If q_2 is the query object, p is a candidate that needs refinement, i.e. we must launch an exact kNN query around p.

Thus, for each object, instead of the kNN distance(s) of a given value of k or all possible values of k, we simply store two approximation functions. We can use an extended M-Tree, that aggregates for each node the maximum of all conservative approximations and the minimum of all progressive approximations of all child nodes or data objects contained in that node. These approximations are again represented as functions. At runtime, we can estimate the maximum kNN distance for each node using the approximation in order to prune nodes analogously to the way we can prune objects. The resulting candidate objects can be identified as true hits or candidates that need further pruning using the progressive approximation.

In the following, we introduce how to compute a conservative approximation of the kNN distances for arbitrary $k \leq k_{max}$ (cf. Section 4.1). We then sketch how a progressive approximation can be generated analogously (cf. Section 4.2). After that, we describe how these approximations can be integrated into an M-Tree. The resulting structure is called MRkNNCoP-Tree (Metric RkNN with conservative and progressive approximations — cf. Section 4.3). At the end of this section, we outline our novel RkNN search algorithm (cf. Section 4.4).

4.1 Conservative Approximation of k-NN Distances

As discussed above, a conservative approximation of the kNN distances of each data object is needed in order to determine those objects that can be safely dropped because they cannot be part of the final result.

First, we have to address the problem to select a suitable model function for the conservative approximation of our k-nearest neighbor distances for every $k \leq k_{max}$. In our case, the distances of the neighbors of an object o are given as a sequence

$$NNdist(o) = \langle nndist_1(o), nndist_2(o), ...nndist_{k_{max}}(o)\rangle$$

and this sequence is ordered by increasing k. Due to monotonicity, we also know that $i < j \Rightarrow nndist_i(o) \leq nndist_j(o)$. Our task here is to describe the discrete sequence of values by some function $appx_o : \mathbb{N} \rightarrow \mathbb{R}$ with $appx_o(k) \approx nndist_k(o)$. As the approximation function is required to be conservative we have the additional constraint $appx_o(k) \geq nndist_k(o)$.

From the theory of self-similarity [8] it is well-known that in most data sets the relationship between the number of objects enclosed in an arbitrary hypersphere and the scaling factor (radius) of the hypersphere (the same is valid for other solids such as hypercubes) approximately follows a power law:

$$encl(\varepsilon) \propto \varepsilon^{d_f}$$

where ε is the scaling factor, $encl(\varepsilon)$ is the number of enclosed objects and d_f is the fractal dimension. The fractal dimension is often (but not here) assumed to be a constant which characterizes a given data set. Our k-nearest neighbor sphere can be understood to be such a scaled hypersphere where the distance of the k-nearest neighbor is the scaling factor and k is the number of enclosed objects. Thus, it can be assumed that the k-nearest neighbor distances also follow the power law and form approximately a line in log-log space (for an arbitrary logarithmic basis) [8], i.e.:

$$\log(nndist_k(o)) \propto \frac{\log(k)}{d_f}.$$

From this observation, it follows that it is generally sensible to use a model function which is linear in log-log space

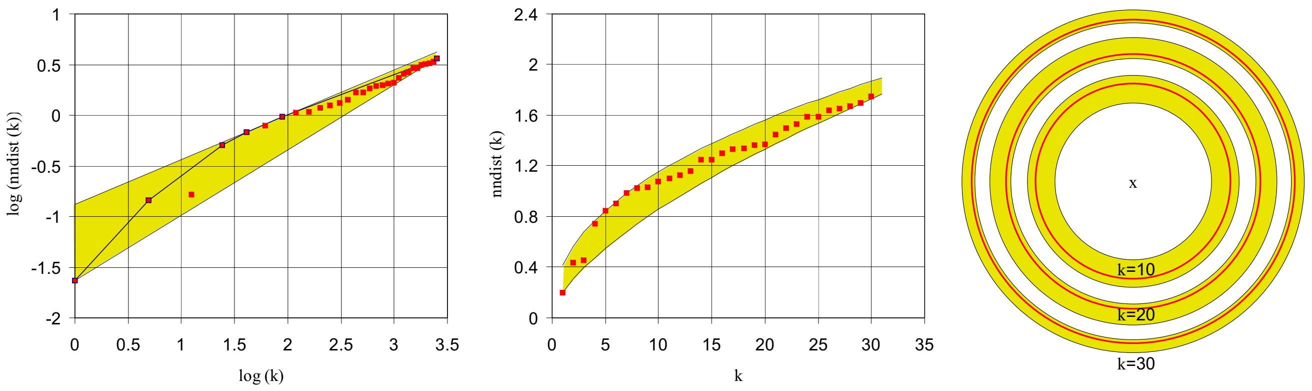


Figure 3: Visualizations of the conservative and progressive approximations of the k-nearest neighbor distances of a sample object for different values of k.

— corresponding to a parabola in non-logarithmic space — for the approximation. Obviously, computing and storing a linear function needs considerable less overhead than a higher order function. Since we focus in this section on the approximation of the values of the k-nearest neighbor distance over varying k in a log-log sense, we consider the pairs $(\log(k), \log(nndist_k(o))$ as points of a two-dimensional vector space (x_k, y_k). These points are not to be confused with the objects stored in the database (e.g. the object o the nearest neighbors of which are considered here) which are general metric objects. Whenever we speak of *points* (x, y) or *lines* $((x_1, y_1), (x_2, y_2))$ we mean points in the two-dimensional log-log space where $\log(k)$ is plotted along the x-axis and $\log(nndist_k(o))$ for a given general metric object $o \in \mathcal{D}$ is plotted along the y-axis.

In most other applications of the theory of self-similarity it is necessary to determine a classical regression line without any constraint conditions, approximating the true values of $nndist_k(o)$ with least square error. A conventional regression line would find the parameters (m, t) of the linear function $y = m \cdot x + t$ minimizing least square error:

$$\sum_{1 \le k \le k_{max}} ((m \cdot x_k + t) - y_k)^2 = \min$$

which evaluates the well known formula of a regression line in 2D space. This line, however, is not a conservative approximation of a point set. In order to guarantee no false dismissals we need here a line which minimizes the above condition while observing the constraint that all actual y_k-values, i.e. $\log(nndist_k)$, are less or equal than the line, i.e. $y_k \le m \cdot x_k + t$, and we will derive a method with a linear time complexity in the number k_{max} of k-nearest neighbor distances to be approximated (provided that the distances are ordered according to increasing k). We formally state this optimization problem:

Optimization Goal. The optimal conservative approximation of a sequence $NN_k(o)$ of k-nearest neighbor distances of an object o is a line

$$L_{opt}(o) = (m_{opt}, t_{opt}) : y = m_{opt} \cdot x + t_{opt}$$

in the log-log space. This line defines the following approximation function:

$$\log(appx_o(k)) = m_{opt} \cdot \log(k) + t_{opt}$$

with the following constraints:

C1. $L_{opt}(o)$ is a conservative approximation of y_k, i.e. $y_k \le m_{opt} \cdot x_k + t_{opt}$ $(\forall k : 1 \le k \le k_{max})$.

C2. $L_{opt}(o)$ minimizes the mean square error, i.e.

$$\sum_{1 \le k \le k_{max}} (m_{opt} \cdot x_k + t_{opt} - y_k)^2 = min$$

An example is visualized in Figure 3. Here we have a sequence of k-nearest neighbor distances for $1 \le k \le k_{max} = 30$ which are depicted as squares in the left diagram in a log-log space and in the middle diagram in non-logarithmic space. The corresponding visualization of the conservative and progressive approximations in the data space for $k = 10, 20, 30$ are depicted on the right hand side in Figure 3. In the left diagram the optimal line for the conservative approximation is the upper limit of the shaded area. In Section 4.2 we will also introduce the progressive approximation which is the lower limit of the shaded area. These two limiting lines correspond to the parabolic functions depicted in the middle diagram where we can directly determine the conservative and progressive approximations for a given k. On the right hand side in Figure 3, we have an object x together with its actual k-nearest neighbor distance for $k = \{10, 20, 30\}$. The three shaded shapes mark the conservative and progressive approximations for these three k-values.

We develop our method to determine the approximation function in three steps. At first, we show that the line must interpolate (pass through) either two neighboring points or one single point of the upper part of the convex hull of the set to be approximated. Second, we show how to optimize a regression line (m_A, t_A) under the constraint that it has to interpolate one given *anchor point* A (in our method, only points of the upper convex hull have to be considered as anchor points). As a third step, we show that the line obtained in the second step is the optimal line according to our optimization goal if and only if both the successor and the predecessor of A in the upper convex hull are under the line (m_A, t_A).

Moreover, we will show the following: If the predecessor exceeds the line, then the optimum line cannot pass through one of the successors of A. *Vice versa*, if the successor exceeds line (m_A, t_A) the optimum line cannot pass through one of the predecessors of A. This statement gives us the justification to search for a suitable anchor point using a

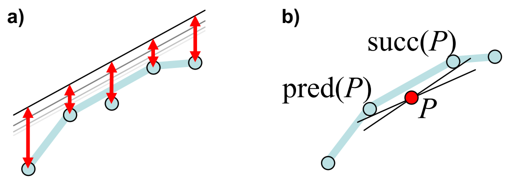


Figure 4: Constraints on the optimal line: Line intersects at least a) one point of the approximated set (Lemma 1) b) one point of the UCH (Lemma 2)

bisection search (in contrast to linear search the bisection search improves the total runtime but not the complexity of the overall method which will be shown to be linear in k_{max}). Finally, we show that the global optimum is either found directly by the bisection search or interpolates the last two points considered in the bisection search.

Step 1. First, we show that the line must interpolate either two neighboring points or one single point of the upper convex hull of the approximated point set. The upper convex hull (UCH) is a sequence

$$UCH = \langle (x_{k_1}, y_{k_1}), (x_{k_2}, y_{k_2}), ..., (x_{k_u}, y_{k_u}) \rangle$$

composed from u points ($2 \le u \le k_{max}$). UCH always starts with the first point and ends with the last point to be approximated, i.e. $k_1 = 1; k_u = k_{max}$. It contains the remaining points in ascending order, i.e. $k_i < k_j \Leftrightarrow i < j$, and due to monotonicity we know also $x_{k_i} < x_{k_j}$ and $y_{k_i} < y_{k_j}$. UCH forms a poly-line composed from one or more (exactly $u-1$) line segments $S_i = ((x_{k_i}, y_{k_i}), (x_{k_{i+1}}, y_{k_{i+1}}))$ where the most important property is that the slope $s(S_i) = (y_{k_{i+1}} - y_{k_i})/(x_{k_{i+1}} - x_{k_i})$ of the line segments S_i is strictly monotonically decreasing ($i < j \Leftrightarrow s(S_i) > s(S_j)$), i.e. the segments form a right turn. UCH is the maximal sequence with this property, and, therefore, the composed UCH line forms an upper limit of the points to be approximated. The upper convex hull has some nice properties which are important to solve our optimization problem. First, as we will prove in Lemma 1, the optimal conservative approximation must interpolate one or two points of the convex hull. Second, and more importantly for the complexity of our method, it facilitates the test whether or not the constraint is fulfilled that all approximated points are below a given line. We will see later that only two points of UCH have to be tested. In contrast, if we do not know the UCH, all the points $(x_k, y_k), 1 \le k \le k_{max}$ of the complete approximated set need to be tested.

It is easy to see that an optimal line must interpolate at least one point of the approximated set:

LEMMA 1. *The optimal line L_{opt} must interpolate at least one point of the set of conservatively approximated points.*

PROOF. Assume L_{opt} is above all points of the data set but does not touch one of the points of the data set. Then we could move the line downwards (leaving it parallel to the original line) until it touches one of the points without violating the constraints. When moving downwards, the distance of the line to every point decreases. Therefore the original line cannot be optimal. □

This is visualized in Figure 4a. In our next lemma, we show that it is not possible that the approximating line interpolates only points which are not in UCH. In other words, it cannot happen that none of the points that are interpolated by L_{opt} are not in UCH.

LEMMA 2. *Any line L interpolating a point $(x_k, y_k), 1 \le k \le k_{max}$ which fulfills the constraint that all approximated points are upper bound by it must interpolate at least one point $P \in UCH$.*

PROOF. Let us assume, that L interpolates only one point $P \notin UCH$. Let $pred(P)$ be the last point before P which is member of UCH and $succ(P)$ be the first point after P in UCH. According to the definition of UCH, the two segments $S_1 = (pred(P), P)$ and $S_2 = (P, succ(P))$ form a left turn (otherwise, P would have to be member of UCH), i.e. $s(S_1) < s(S_2)$. The slope of L must be less than the slope of S_1 and greater than the slope of S_2 to upper bound both $pred(P)$ and $succ(P)$. This is not possible as the slope of S_1 is less than that of S_2 (left turn). □

The idea of the proof is visualized in Figure 4b. From Lemma 2 and the obvious observation that no straight line can interpolate more than two points of UCH (which form a right turn), we know that L_{opt} must interpolate one or two points of UCH. We will show later that both cases occur, indeed. In Figure 4, the 4 points of UCH are marked with darker frames and additionally connected by lines. This example also visualizes how the conservative approximation interpolates two points of UCH.

UCH can be determined from the ordered set of points $(x_k, y_k), 1 \le k \le k_{max}$ in linear time by sequentially putting the points onto a stack and deleting the second point after the top-of-stack whenever the slope is increasing among the three points on the top-of-stack. Obviously, the determination of UCH can be done in $O(k_{max})$.

Step 2. From Step 1 we know that our optimal line interpolates at least one point of UCH. Our next building block for the generation of the conservative approximation is the derivation of a regression line under the constraint that one given point is interpolated. We call this point the *anchor point*. Note that, until now, we have not yet shown how to select this anchor point among the points in UCH. For the moment, we can assume that we do this optimization for every point in UCH, but we will show later how the anchor can be determined in a more directed way using bisection search. Furthermore, note that the optimization described in this step does not necessarily yield a line which fulfills all constraints. Our final method will do so, but the method here need not yet meet this requirement.

Given an anchor point $A = (x_A, y_A)$ and an approximated point set S, we call a line $L_A = (m_A, t_A)$ *anchor-optimal* w.r.t. A and S if it interpolates A and approximates S with least square error. As we know that our anchor point (x_A, y_A) is interpolated, in our optimization problem (to select the optimal line $y = m \cdot x + t$) we have only one variable (say m) as degree of freedom, because t is fixed by the constraint $y_A = m \cdot x_A + t$. Therefore, we can integrate the constraint into our line formula:

$$y = mx - mx_A + y_A$$

We search for that line which minimizes the sum of squared distances from the actual points to be approximated:

$$\sum_{1 \le k \le k_{max}} (mx_k - mx_A + y_A - y_k)^2 = min$$

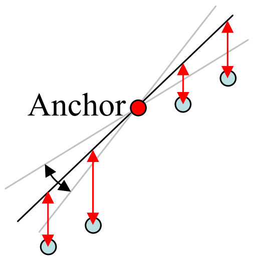


Figure 5: Illustration of an anchor point.

An example is visualized in Figure 5 where we can see the anchor point as well as the distances from an arbitrary line which interpolates the anchor point. A necessary condition for a minimum is that the derivative vanishes:

$$\frac{\partial}{\partial m}\left(\sum_{1\le k\le k_{max}}(mx_k - mx_A + y_A - y_k)^2\right) = 0$$

There exists only one solution $m = m_A$ to this equation with

$$m_A = \frac{\sum_k (y_k - y_A)(x_k - x_A)}{\sum_k (x_k - x_A)^2} =$$
$$= \frac{k_{max}x_A y_A - y_A(\sum_k x_k) + (\sum_k x_k y_k) - x_A(\sum_k y_k)}{k_{max}x_A^2 + (\sum_k x_k^2) - 2x_A(\sum_k x_k)}$$

This local optimum is a minimum, and there are no further local minima or maxima (except $m = \pm\infty$). Although the second formula for m_A looks complex at the first glance, it is very efficient to evaluate: the "expensive" sum-terms are independent from the anchor point A and need, thus, to be evaluated only once. If m_A is determined for more than one anchor point, those terms which need evaluation of all x_k and y_k must be evaluated only once. We will show in the next section that, in general, we need to evaluate m_A for a number of anchor points which is logarithmic in the number of points in the convex hull, and, therefore, is in the worst case also in $O(\log k_{max})$.

Step 3 So far, we know that our optimal line interpolates at least one point of the UCH (Step 1). Together with Step 2, we know how to compute the optimal line: we need to construct an anchor optimal line given the right anchor point. We also know that this anchor point is part of the UCH. The remaining task is to find this right anchor point of the UCH. In the third step, we will show that the correct anchor point and, thus, the global optimum approximation can be found efficiently by means of bisection search. The goal is to find that point $(x_{opt}, y_{opt}) \in UCH$ which is intersected by the global-optimum approximation-line. The search starts by the median $M_{UCH} = (x_{k_m}, y_{k_m})$ of the UCH which we first take as anchor point and compute its local optimum line (m_{k_m}, t_{k_m}). By means of the location of (m_{k_m}, t_{k_m}), we decide in which direction the bisection search has to proceed to find the correct anchor point. Thereby we distinguish three cases:

1. Both the predecessor and successor of M_{UCH} in the UCH, i.e. $pred(M_{UCH})$ and $succ(M_{UCH})$ are not above (m_{k_m}, t_{k_m}) (i.e. below or exactly on it).

2. The predecessor $pred(M_{UCH})$ is above (m_{k_m}, t_{k_m}).

3. The successor $succ(M_{UCH})$ is above (m_{k_m}, t_{k_m}).

As state above, the search starts at the median of the UCH, i.e. at M_{UCH}. At each step of the bisection search, we examine the three cases for the current point $(x_{k_m}, y_{k_m}) \in UCH$ with its local optimum line (m_{k_m}, t_{k_m}). As we will see in the next lemma, in the first case we can stop our search, because (m_{k_m}, t_{k_m}) is the global optimum. In the second case we proceed the search with the corresponding predecessor (x_{k_p}, y_{k_p}) of (x_{k_m}, y_{k_m}). If the corresponding predecessor has been already considered during the bisection search we can stop the search and the global optimum line passes through (x_{k_p}, y_{k_p}) and (x_{k_m}, y_{k_m}). In the third case we proceed the search with the successor of (x_{k_m}, y_{k_m}) analogous to the second case. In this way, the global optimum approximation can be found due to the following lemma:

Lemma 3. *Let $A \in UCH$ be the anchor point of an anchor optimal line (m_A, t_A), i.e. the line interpolate A and approximates the points with least square error. Furthermore, let (m_{A2}, t_{A2}) be another line which also passes through A and additionally passes through any successor $A^+ \in UCH$ of A, whereas $m_{A2} < m_A$. Then the global optimum line passes through A or any predecessors $A^- \in UCH$ of A.*

Proof. Let $A \in UCH$ be the anchor point of a line (m_A, t_A) which is anchor optimal w.r.t. A. Furthermore, let (m_{A2}, t_{A2}) be another line which pass through two points A and any successor $A^+ \in UCH$ of A, whereas $m_{A2} < m_A$. In the following we assume, that the global optimum line pass through A^+ but pass not through A. Let this line be $(m_{A'}, t_{A'})$, as depicted in Figure 6. Following the above assumption, the sum of squared distances from the actual points to line (m_{A2}, t_{A2}) must be greater than the squared distances to line $(m_{A'}, t_{A'})$. In the following, we will show that this assumption does not hold.

Let us first group all points into three sets S_0, S_1 and S_2. Whereas, S_0 denotes the set of all preceding points of A, S_1 denotes the set of all points succeeding A (A included) but preceding A_+ and S_2 denotes the set of all points succeeding A_+ (A_+ included). In the following we consider two additional lines (m_{A1}, t_{A1}) and (m_{A3}, t_{A3}), such that (m_{A1}, t_{A1}) pass through A and (m_{A3}, t_{A3}) pass through A_+. Furthermore, without loss of generality, let the slopes of (m_{A1}, t_{A1}) and (m_{A3}, t_{A3}) be in such a way, that $m_{A1} \le m_A$ and $m_{A1} - m_{A2} = m_{A2} - m_{A3}$ and $m_A \ge m_{A1} \ge m_{A2} \ge m_{A3} \ge m_{A'}$ (cf. Figure 6).

Let I_i^A be the sum of all squared distances from the points in S_i to the line (m_A, t_A), i.e.

$$I_i^A = \Sigma_{(x,y)\in S_i} dist((x,y),(m_A,t_A)),$$

where

$$dist((x,y),(m_A,t_A)) = (m_A x - m_A x_A + y_A - y)^2.$$

Then, the sum of the squared distances between all points and a line (m_A, t_A) is equal to the sum $I_0^A + I_1^A + I_2^A$. If we assume that (m_A, t_A) is local optimal according to its anchor point A and $m_A \ge m_{A1} \ge m_{A2}$, then

$$I_0^A + I_1^A + I_2^A \le I_0^{A1} + I_1^{A1} + I_2^{A1} \le I_0^{A2} + I_1^{A2} + I_2^{A2}.$$

The latter inequality is equivalent to:

$$I_0^{A2} - I_0^{A1} \ge I_1^{A1} - I_1^{A2} + I_2^{A1} - I_2^{A2}$$

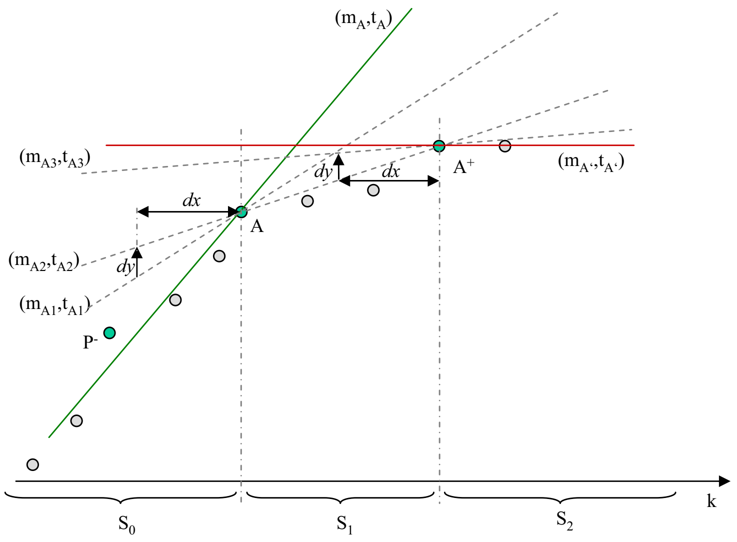


Figure 6: Illustration of the proof of Lemma 3: Monotonicity of the error of the conservative approximation of kNN distances

By means of the theorems on intersecting lines and the assumption that $m_{A1} - m_{A2} = m_{A2} - m_{A3}$, the following statements hold:

1. $I_0^{A3} - I_0^{A2} \geq I_0^{A2} - I_0^{A1}$
2. $I_0^{A3} - I_0^{A2} \geq I_0^{A2} - I_0^{A1}$
3. $I_1^{A1} - I_1^{A2} \geq 0$
4. $I_1^{A3} - I_1^{A2} \geq 0$

With 1) the following statement holds:

$$I_0^{A3} - I_0^{A2} \geq I_1^{A1} - I_1^{A2} + I_2^{A1} - I_2^{A2}$$

With 2) this leads to

$$I_0^{A3} - I_0^{A2} \geq I_1^{A1} - I_1^{A2} + I_2^{A2} - I_2^{A3}$$

Due to 3) and 4), the inequality can be resolved to:

$$I_0^{A3} - I_0^{A2} + I_1^{A3} - I_1^{A2} \geq I_2^{A2} - I_2^{A3}$$

which is equivalent to

$$I_0^{A2} + I_1^{A2} + I_2^{A2} \leq I_0^{A3} + I_1^{A3} + I_2^{A3} \leq I_0^{A'} + I_1^{A'} + I_2^{A'}$$

This means, that the sum of squared distances from the actual points to line (m_{A2}, t_{A2}) is lower than the squared distances to line (m_{A3}, t_{A3}) which contradicts the assumption. $\square$

Summary: The Optimization Algorithm

The algorithm which computes the optimal conservative approximation line is depicted in Figure 7. It requires as input an object o, the sequence $NNdist(o)$ of kNN distances of o which should be approximated and the upper limit k_{max}. The algorithm reports the line $L_{opt} = (m, t)$ corresponding to the line $y = m \cdot x + t$ which denotes the optimal conservative approximation line. The algorithm shown in Figure 7 consists of two main parts.

In the first part we compute the UCH of the kNN distances in $NBNdist(o)$ in the log-log space, i.e. of the points $(\log k, \log nndist_k(o))$. For this task, we apply a modification of Graham's scan algorithm for the convex hull [1] which parses all kNN-distances from $k = 1$ to k_{max} and buffers references to those kNN-distances within a stack which build a conservative right-curved sequence denoting the upper convex hull of the points $(\log k, \log nndist_k(o))$ in log-log space. This step is performed because we have shown that at least one point of the convex hull must be interpolated by L_{opt} (cf. Lemma 2).

In the second main part we perform a bisection search for the optimum approximation line. Our algorithm starts with the complete UCH. It selects the median point of the UCH as the first anchor point and computes its anchor optimal line (aol). By inspecting its direct predecessor and successor, respectively, it distinguishes between 3 cases: (1) Both neighbor points are below aol: Then the global optimum is reached. (2) The right neighbor point (successor) is above the aol: We proceed recursively with the right half of the UCH (this time considering the median of the right half). Case (3), left neighbor above aol is handled analogously. The slope of the computed line is used to identify the search space of the subsequent search step (as substantiated by Lemma 3). In each step of this algorithm, the problem size is divided by two. Finally, the parameters (m, t) of the computed line are reported.

4.2 Progressive Approximation of kNN Distances

As discussed above, a progressive approximation of the kNN distances of each data object can be used to determine true hits. Analogously to the optimal conservative approximation, the optimal progressive approximation of a sequence $NN_k(o)$ of k-nearest neighbor distances of an object o is a line $L_{opt}(o)$ in log-log space. The only difference is that the line $L_{opt}(o)$ must satisfy a *progressive* constraint, i.e. constraint C1 in Section 4.1 is changed as follows:

$$y_k \geq m_{opt} \cdot x_k + t_{opt} \ (\forall 1 \leq k \leq k_{max}).$$

We can generate the progressive approximation analogously as described in Section 4.1. The difference is that we have to consider the lower convex hull instead of the

```
optimize($o$, $NNdist(o)$, $k_{max}$)
    // First Part:
    compute upper convex hull $UCH$ of the points
        $(\log k, \log nndist_k(o))$ according to [1];
    // Second Part:
    while $UCH$ still contains unmarked points do
        $(x_a, y_a)$ = median of $UCH$;
        compute anchor optimal line $(m_a, t_a)$
            w.r.t. anchor point $(x_a, y_a)$;
        mark $(x_a, y_a)$;
        $(x_p, y_p) = pred((x_a, y_a))$; // predecessor in $UCH$
        $(x_s, y_s) = succ((x_a, y_a))$; // successor in $UCH$
        if $y_p \leq m_a \cdot x_p + t_a$ and $y_s \leq m_a \cdot x_s + t_a$ then
            // global optimum found
            return $(m_a, t_a)$;
        else if $y_p > m_a \cdot x_p + t_a$ then
            // examine predecessor
            if $(x_p, y_p)$ is already marked then
                $m_p = (y_a - y_p)/(x_a - x_p)$; $t_p = y_p - x_p \cdot m_p$;
                return $(m_p, t_p)$;
            else // proceed with left side of UCH
                $UCH$ = left side of $UCH$;
        else if $y_s > m_a \cdot x_s + t_a$ then
            // examine successor
            if $(x_s, y_s)$ is already marked then
                $m_s = (y_a - y_s)/(x_a - x_s)$; $t_s = y_s - x_s \cdot m_s$;
                return $(m_s, t_s)$;
            else // proceed with right side of UCH
                $UCH$ = right side of $UCH$;
    end while
```

Figure 7: Finding the optimal approximation.

upper convex hull. Obviously, the resulting progressive approximation line must be below all real kNN distances.

4.3 Aggregating the Approximations

So far, we have shown how to generate a conservative and progressive approximation for each object of the database. However, the conservative and progressive approximations can also be used for the nodes of the index to prune irrelevant sub-trees. Similar to the RdNN-Tree, we need to aggregate for each data node the maximum kNN distances of the objects within that node. For this aggregation, the conservative approximations must be used. In addition, we could aggregate the minimum kNN distance of the objects within the node in order to detect true hits. For this aggregation, the progressive approximation should be used. However, in most cases the progressive aggregation does not pay off because for a node it is not selective enough.

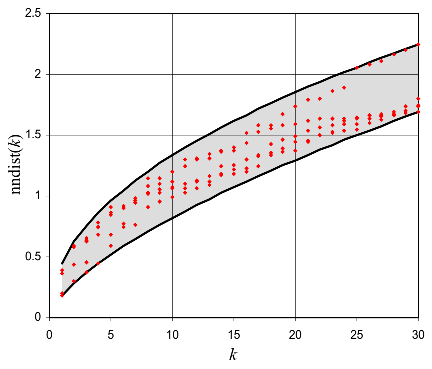
We build the conservative approximation of a data node N by conservatively approximating the conservative approximation lines $L_{opt}(o_i) = (m_i, t_i)$ of all data objects $o_i \in N$. The resulting approximation line is defined by the points $(\log(1), y_1)$ and $(\log k_{max}, y_{k_{max}})$, where y_1 is the maximum of all lines at $k = 1$ and $y_{k_{max}}$ is the maximum of all lines at k_{max}, formally

$$y_1 = \max_{o_i \in N} m_i \cdot \log 1 + t_i = \max_{o_i \in N} t_i,$$

$$y_{k_{max}} = \max_{o_i \in N} m_i \cdot \log k_{max} + t_i.$$

The progressive approximation can be determined analogously. Figure 8 illustrates both concepts.

The approximation information can be propagated to parent, i.e. directory, nodes in a straightforward way. The resulting storage overhead is negligible because the additional information stored at each node is limited to two values for



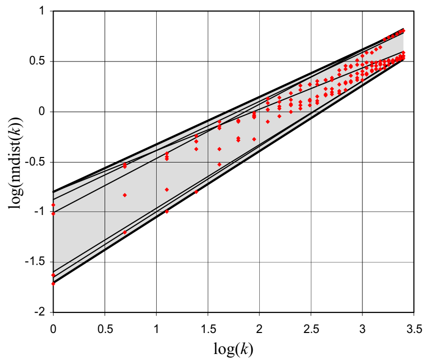


Figure 8: Aggregated approximation lines.

the conservative approximation. Nevertheless, the kNN distance for *any* value of k can be estimated at each node.

We call the resulting index structure MRkNNCoP (Metric reverse kNN with conservative and progressive approximations) - Tree. The original concepts of the MRkNNCoP-Tree presented here can be incorporated within any hierarchically organized index for metric objects. Let us note that the concepts can obviously also be used for RkNN search in Euclidean data. In that case, we need to integrated the proposed approximations into Euclidean index structures such as the R-tree [6], the R*-tree [2], or the X-tree [3].

4.4 RkNN Search Algorithm

The search algorithm for RkNN queries on our MRkNNCoP-Tree is similar to that of the RdNN-Tree. However, our index structure can answer RkNN queries for any k specified at query time and, due to the use of a metric index structure, is applicable to general metric objects.

The pseudo code of the query algorithm is depicted in Figure 9. A query q is processed by starting at the root of the index. The index is traversed such that the nodes having a smaller mindist to q than their aggregated kNN distance approximations are refined. Those nodes, where the mindist to q is larger than their aggregated kNN distance approximation are pruned. A node N of a M-Tree is represented by its routing object N_o and the covering radius N_r. All objects represented by N have a distance less than N_r to N_o. The mindist of a node N and a query point q, denoted by $mindist(N, q)$, is the maximum of the distance of q to N_o minus the covering radius N_r (if $dist(q, N_o) > N_r$) and zero (if $dist(q, N_o) \leq N_r$), formally:

$$mindist(N, q) = \max\{dist(q, N_o) - N_r, 0\}.$$

```
RkNN_query($\mathcal{D}$, $q$, $k$)
  // Assumption: $\mathcal{D}$ is organized as MR$k$NNCoP
  $queue$ := new Queue;
  insert root of $\mathcal{D}$ into $queue$;
  while not $queue$.isEmpty() {
    $N$ := $queue$.getFirst();
    if $N$ is node then {
      if $mindist(N,q) \leq m_N \cdot \log k + t_N$ then
        insert all elements of $N$ into $queue$;}
    else // $N$ is a point
      if $dist(N,q) < m^p_N \cdot \log k + t^p_N$ then
        add $N$ to result set;
      else if $m^c_N \cdot \log k + t^c_N > dist(N,q)$
        add $N$ to candidate set;
  end while
```

Figure 9: The RkNN search algorithm.

The aggregated kNN distance of a node N, denoted by $k\text{NN}_{agg}(N)$ can be determined from the conservative approximation $L_{opt}(N) = (m_N, t_N)$ of N by
$k\text{NN}_{agg}(N) = m_N \cdot \log k + t_N$.

Thus, we can prune a node N with approximation $L_{opt}(N) = (m_N, t_N)$ if $mindist(N,q) > m_N \cdot \log k + t_N$.

The traversal ends up at a data node. Then, all points p_i inside this node are tested using their conservative approximation $L_{con}(p_i) = (m^c_{p_i}, t^c_{p_i})$ and their progressive approximation $L_{prog}(p_i) = (m^p_{p_i}, t^p_{p_i})$. A point p can be dropped if $dist(p,q) > m^c_p \cdot \log k + t^c_p$. Otherwise, if $dist(p,q) < m^p_p \cdot \log k + t^p_p$, point p can be added to the result. Last but not least, if $m^c_p \cdot \log k + t^c_p > dist(p,q) > m^p_p \cdot \log k + t^p_p$, point p is a candidate that need an exact kNN query as refinement. Using this strategy, we get a set of true hits and a set of candidates from the RkNNCoP-Tree. The set of candidates are refined using a batch kNN search as proposed in [12].

5. EVALUATION

All experiments have been performed on Windows workstations with a 32-bit 3.2 GHz CPU and 2 GB main memory. We used a disk with a transfer rate of 50 MB/s, a seek time of 6 ms and a latency delay of 2 ms. In each experiment we applied 100 randomly selected RkNN queries to the particular data set and reported the overall result. The runtime is presented in terms of the elapsed query time including I/O and CPU-time. All evaluated methods have been implemented in Java.

5.1 Metric RkNN Search

Since there is no recent approach for RkNN search in general metric spaces, we compared our MRkNNCoP-Tree with two already discussed variants. The first variant, denoted by "MRkNN-Max", stores for each object of the database the kNN distance for one arbitrary $k_{max} \geq 1$. Obviously, this approach has less storage overhead than the MRkNNCoP-Tree but needs expensive refinement steps if the parameter k of the query differs from the precomputed k_{max} value. The second variant, denoted by "MRkNN-Tab", stores all kNN distances for $k = 1 \ldots, k_{max}$ in a table for each data object and each node of the tree, respectively. The advantage of this approach is that only true hits are computed, i.e. we do not need any refinement step. However, the distance table becomes quite large for increasing k_{max} values. This leads to a smaller branching factor of the tree nodes. Thus, the

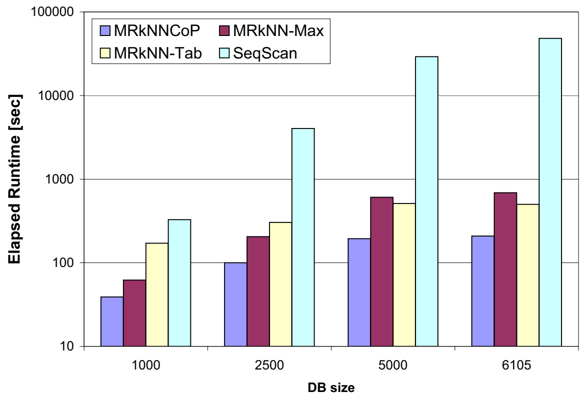


Figure 10: Runtime w.r.t. database size on Oldenburg data set.

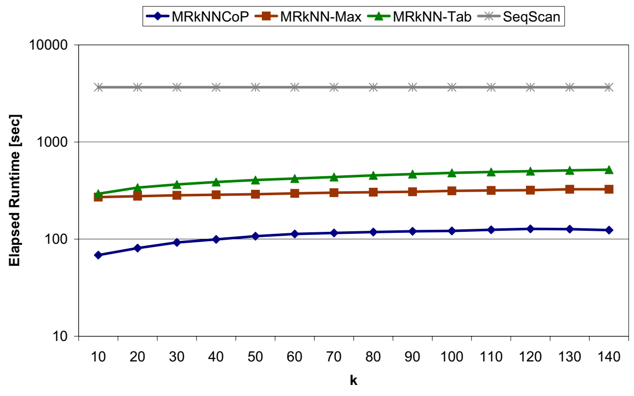


Figure 11: Runtime w.r.t. parameter k on Oldenburg data set (2.500 data objects).

tree is higher suffering from large directory traversal overhead. A third competitor of our MRkNNCoP-Tree is the sequential scan, denoted as "SeqScan".

Our experiments where performed using two real-world data sets. The first one is a road network data set derived from the city of Oldenburg, which contains 6,105 nodes and 7,035 edges. The average degree of the nodes in this network is 2.63. The data set is online available [2]. The nodes of the network graph were taken as database objects from which subsets of different size were selected to form the test data set. For the distance computation we used the shortest-path distance computed by means of the Djikstra algorithm. The second data set consists of protein sequences taken from SWISSPROT database, the Levenstein distance was used as similarity distance. Due to space limitations, no figures of the results of the second data set are depicted.

Runtime w.r.t. database size. In Figure 10 the runtime of the competitive algorithms w.r.t. varying data base size is illustrated in a log-scale. The parameter k was set to $k = 50$, while $k_{max} = 100$. It can be observed that our MRkNNCoP approach clearly outperforms the simple "MRkNN-Max" and "MRkNN-Tab" approaches. This is

[2] www.fh-oow.de/institute/iapg/personen/brinkhoff/generator/

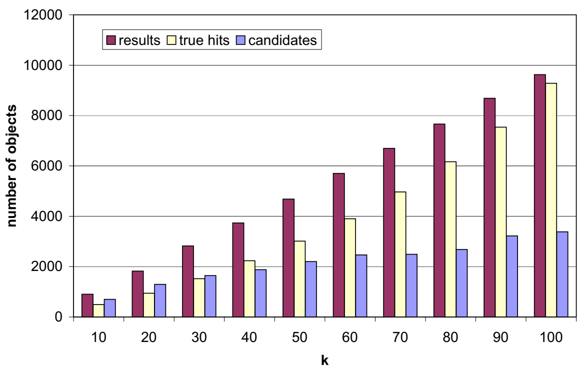


Figure 12: Pruning capability w.r.t. parameter k on Oldenburg data set (5.000 data objects).

due to the already discussed shortcomings of the two naive approaches. "MRkNN-Max" suffers from a poor pruning capability, whereas "MRkNN-Tab" suffers from a large directory and, thus, from a costly tree traversal. Similar results were observed on the second metric data set.

Runtime w.r.t. parameter k. A similar result can be observed when comparing the runtime of MRkNNCoP and its competitors w.r.t. varying parameter k (cf. Figure 11). In this experiment we set $k_{max} = 150$. Again, the runtime axis is in log-scale to visualize the sequential scan. Obviously, the sequential scan is almost independent from k and lies significantly above the runtime of the other techniques. Again, the MRkNNCoP-Tree performs significantly better than the naive approaches. We also made similar observations on the second real-world data set.

Pruning capabilities. Figure 12 shows the pruning capability of MRkNNCoP w.r.t. k on the Oldenburg data set. Compared with the size of the result set only a small number of candidates has to be refined, i.e. the conservative approximation yields a sound upper bound. Furthermore, the number of true hits we get from our progressive approximation increases with increasing k. For example, for $k \geq 70$ the pruned true hits are more than 75% of the result set. For these objects no expensive refinement step is necessary, thus our progressive approximation provides a very efficient lower bound for the kNN distance. We observed similar results on our second metric data set.

5.2 Euclidean RkNN Search

We also integrated our concepts into an X-Tree [3] in order to support RkNN search in Euclidean data. We made the same experiments as presented already above for metric data using three real-world data sets. It turned out that our approach even outperforms recent RkNN search methods that are specialized for Euclidean data. The used data sets include a set of 5-dimensional vectors generated from the well-known SEQUOIA 2000 benchmark data set and two "Corel Image Features" benchmark data sets from the UCI KDD Archive, one contains 9 values for each image, the other data set contains 16-dimensional texture values. The underlying X-Tree had a node size of 2 KByte.

Naive approaches. Again, in our first experiment we compared our approach with the "MRkNN-Max" and the "MRkNN-Tab" approaches, both also based on an X-Tree. The experiment was performed on the sequoia data set consisting of 20.000 objects. The results are not shown due to

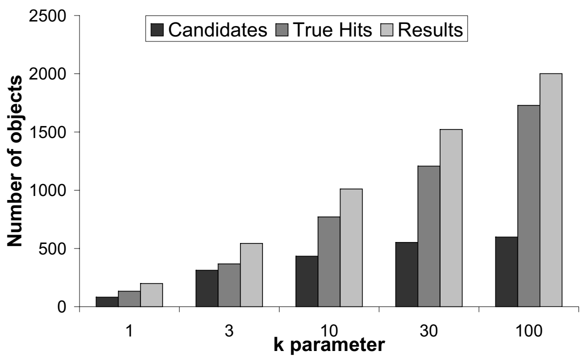


Figure 15: Pruning capability w.r.t. parameter k on color texture data set.

space limitations. The MRkNN-Max approach stores the k_{max}NN distance for $k_{max} = 100$. Again, it turns out that our approach has much better runtime performance than the other two techniques due to the already discussed shortcomings of these naive approaches. We observed similar results for the two other data sets.

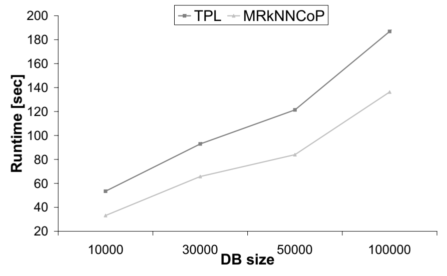
Runtime w.r.t. database size. We compared our approach with the Voronoi-based approach [11] (short "TPL") and the sequential scan (short "SeqScan"). TPL is the only existing approach for the generalized RkNN search. In Figure 13 the runtime of the three algorithms w.r.t. varying data base size is illustrated. The parameter k was set to $k = 10$. For clearness reasons, the sequential scan is not visualized for all database sizes. It can be observed that our MRkNNCoP approach also outperforms the existing TPL approach. For example the performance gain is more than 40% for the 9-dimensional data set. This is due to the fact that MRkNNCoP needs substantially less refinement steps than TPL.

Runtime w.r.t. parameter k. A comparison of the runtime of MRkNNCoP and TPL w.r.t. varying parameter k is depicted in Figure 14. In this experiment the database size was set to 30.000 objects. The runtime of the sequential scan is almost independent from k and lies significantly above the runtime of the two other techniques. Due to clearness reasons it is again not visualized. Again, the MRkNNCoP performs better than TPL. It is also worth noting that the performance gap between MRkNNCoP and TPL increases with increasing parameter k.

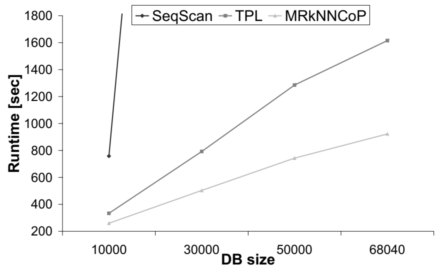
Pruning capabilities. Figure 15 shows the pruning capability of MRkNNCoP w.r.t. k on the 16-dimensional color texture data set (30.000 objects). Compared with the size of the result set only a small number of candidates has to be refined, i.e. the conservative approximation yields a sound upper bound. Furthermore, the number of true hits we get from our progressive approximation is about 66% to 86% of the result set and increases with increasing k. For these objects no expensive refinement step is necessary, thus our progressive approximation provides a very efficient lower bound for the kNN distance. We show only the color texture data set, as the results on the two other data sets are similar.

6. CONCLUSIONS

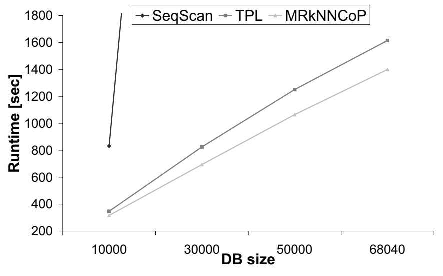
In this paper we proposed the MRkNNCoP-Tree the first index structure for reverse k-nearest neighbor (RkNN) search in general metric spaces where the value for k is specified at query time. Our index is based on the pruning power of



(a) Sequoia (5-dimensional)

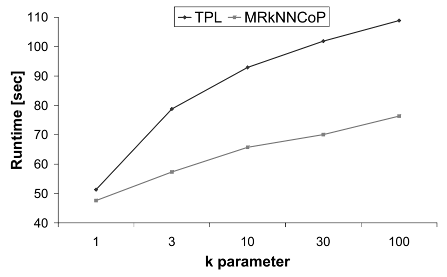


(b) Color moments (9-dimensional)

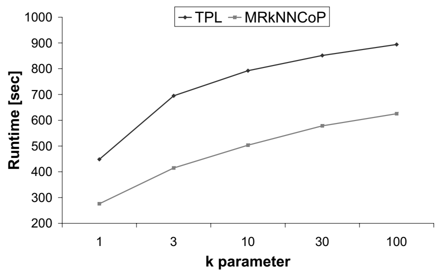


(c) Color texture (16-dimensional)

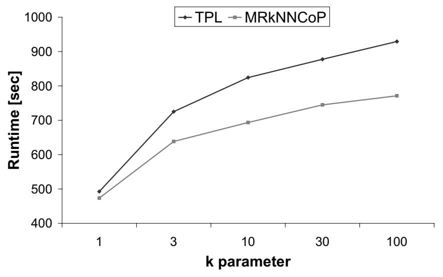
Figure 13: Comparison of runtime w.r.t. database size.



(a) Sequoia (5-dimensional)



(b) Color moments (9-dimensional)



(c) Color texture (16-dimensional)

Figure 14: Comparison of runtime w.r.t. parameter k.

the kNN distances of the database points. We proposed to approximate these kNN distances by a simple function conservatively and progressively in order to avoid a significant storage overhead. We demonstrated how these approximation functions can efficiently be determined and how any tree-like metric index structure can be built with this aggregated information. In our broad experimental evaluation, we illustrated that our MRkNN-CoP-Tree efficiently supports the generalized RkNN search in arbitrary metric spaces. In particular, our approach yields a significant speed-up over the sequential scan and naive indexing solutions. In addition, we demonstrated that our proposed concepts are also applicable for Euclidean RkNN search. We have shown that our MRkNNCoP-Tree even outperforms the only existing solution for RkNN search for Euclidean vector data.

For future work, we will examine data structures for parallel and distributed RkNN queries in Euclidean and general metric spaces.

Acknowledgement

We are grateful with Yufei Tao, Dimitris Papadias, Xiang Lian and Xiaokui Xiao providing us the source code of their Reverse kNN Search in [11].

7. REFERENCES

[1] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9, 1979.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD*, 1990.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-Tree: An index structure for high-dimensional data. In *Proc. VLDB*, 1996.

[4] P. Ciaccia, M. Patella, and P. Zezula. M-Tree: an efficient access method for similarity search in metric spaces. In *Proc. VLDB*, 1997.

[5] C. Ding and H. Peng. Minimum redundancy feature selection from microarray gene expression data. In *CSB03*, 2003.

[6] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD*, 1984.

[7] F. Korn and S. Muthukrishnan. Influenced sets based on reverse nearest neighbor queries. In *Proc. SIGMOD*, 2000.

[8] M. Schroeder. *Fractals, Chaos, Power Laws: Minutes from an infinite paradise.* W.H. Freeman and company, New York, 1991.

[9] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High dimensional reverse nearest neighbor queries. In *Proc. CIKM*, 2003.

[10] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *Proc. DMKD*, 2000.

[11] Y. Tao, D. Papadias, and X. Lian. Reverse kNN search in arbitrary dimensionality. In *Proc. VLDB*, 2004.

[12] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. ICDE*, 2001.