Efficient Reverse k-Nearest Neighbor Estimation

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

Institute for Computer Science
Ludwig-Maximilians Universität München
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, 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. In addition, many applications require a fast approximate answer of RkNN-queries. For these scenarios, it is important to generate a fast answer with high recall. In this paper, we propose the first approach for efficient approximative 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 an approximation of the nearest-neighbor-distances in order to prune the search space. We show that our method scales significantly better than existing non-approximative approaches while producing an approximation of the true query result with a high recall.

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 complementary problem to that of finding the k-nearest neighbors (kNN) of a query object. The goal of a reverse k-nearest neighbor query is to identify the "influence" of a query object on the whole data set. 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. Obviously, more efficient algorithms are required, and, thus, the RkNN problem has been studied extensively in the past few years (cf. Section 2).

As we will discuss in Section 2 these existing methods for RkNN search can be categorized into two classes, the hypersphere-approaches and the Voronoi-approaches. Usually, it is very difficult to extend Voronoi-approaches in order to apply them to general metric objects. Hypersphere-approaches extend a multidimensional index structure to store each ob-

ject along with its nearest neighbor distance. Thus, although most hypersphere-approaches are only designed for Euclidean vectors, these methods can usually be extended for general metric objects. 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. The only existing hypersphere-approach that is flexible w.r.t. the parameter k to some extend is limited by a parameter k_{max} which is an upper bound for the possible values of k. All these recent methods provide an exact solution for the RkNN problem. However, in many applications, an approximate answer for RkNN queries is sufficient especially if the approximate answer is generated faster than the exact one. Those applications usually need a solution for general metric objects rather than a solution limited to Euclidean vector data and, additionally, for handling RkNN queries for any value of k which is only known at query time.

One such sample application is a pizza company that wants to evaluate a suitable location for a new restaurant. For this evaluation, a RkNN query on a database of residents in the target district could select the set of residents that would have the new restaurant as its nearest pizza restaurant, i.e. are potential customers of the new restaurant. In addition, 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 one of the k-nearest pizza restaurant. In both cases, an approximate answer to the RkNN query is sufficient. Usually, the database objects in such an application are nodes in a traffic network (cf. Figure 1). Instead of the Euclidean distance, the network distance computed by graph algorithms like Dijkstra is used.

Another important application area of RkNN search in general metric databases is molecular biology. Researchers all over the world rapidly detect new biological sequences that need to be tested on originality and interestingness. When a new sequence is detected, RkNN queries are applied to large sequence databases storing sequences of biological molecules with known function. To decide about the originality of a newly detected sequence, the RkNNs of this sequence are computed and examined. Again, an approximate answer of the launched RkNN queries is sufficient. In addition, it is much more important to get quick results in order to enable interactive analysis of possible interesting sequences. 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 [DP03].

In general, the RkNN problem appears in many practical situations such as geographic information systems (GIS), traffic networks, adventure games, or molecular biology where the database objects are general metric objects rather than Euclidean vectors. In these application areas, RkNN queries are frequently launched where the parameter k can change from query to query and is not known beforehand. In addition, in many applications, the efficiency of the query execution is much more important than effectiveness, i.e. users want a fast response to their query and will even accept approximate results (as far as the number of false drops and false hits is not too high).

In this paper, we propose an efficient approximate solution based on the hypersphere-approach for the RkNN problem. Our solution is designed for general metric objects and

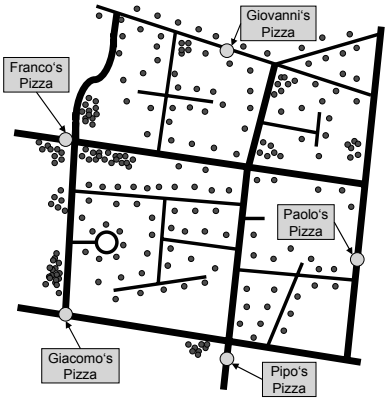


Figure 1: Evaluation of potential customers (small circles) for a new pizza restaurant (larger circles indicate competing pizza restaurants) using RkNN queries.

allows RkNN queries for arbitrary k. In contrast to the only existing approach, the parameter k is not limited by a given upper bounding parameter k_{max}. The idea is to use a suitable approximation of the kNN distances for each k of every object in order to evaluate database objects as true hits or true drops without requiring a separate kNN search. This way, we approximate the kNN distances of a single object stored in the database as well as the kNN distances of the set of all objects stored in a given subtree of our metric index structure. To ensure a high recall of our result set we need an approximation of the kNN distances with minimal approximation error (in a least square sense). We will demonstrate in Section 3 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 AMRkNN (Approximate Metric RkNN)-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, discusses related work, and points out our contributions. In Section 3 we introduce our novel AMRkNN-Tree in detail. Section 4 contains a comparative experimental evaluation. Section 5 concludes the paper.

2 Survey

2.1 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 [CPZ97] (or, if the objects are feature vectors, the R-Tree [Gut84] or the R*-Tree [BKSS90]) is used to speed-up the nearest neighbor queries.

2.2 Related Work

An approximative approach for reverse k-nearest neighbor search in higher dimensional space is presented in [SFT03]. A two-way filter approach is used to generate the results. Recently, in [XLOH05] two methods for estimating the kNN-distance from one known κNN-distance are presented. However, both methods are only applicable to Euclidean vector data, i.e. $\mathcal{D}$ contains feature vectors of arbitrary dimensionality d ($\mathcal{D} \in \mathbb{R}^d$).

All other approaches for the RkNN search are exact methods that usually produce considerably higher runtimes. Recent approaches can be classified as Voronoi-approaches or hypersphere-approaches.

Voronoi-approaches usually use the concept of Voronoi cells to prune the search space. The above-mentioned, approximate solution proposed in [SFT03] can be classified as Voronoi-based approach. In [SAA00], a Voronoi-based approach for reverse 1-nearest neighbor search in a 2D data set is presented. 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. Since the number of units in which the candidates are generated increases exponentially with d, this approach is only applicable for 2D data sets. Recently, in [TPL04] the first approach for RkNN search was proposed, that can handle arbitrary values of k. 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 kNN query must be launched. In general, Voronoi-based approaches can only be applied to Euclidean vector data because the concept of Voronoi cells does not exist in general metric spaces.

Hypersphere-approaches use the observation 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. In [KM00] an index structure called RNN-Tree is proposed for reverse 1-nearest neighbor search based on this observation. The RNN-Tree precomputes for each object p the distance to its 1-nearest neighbor, i.e. $nndist_1(p)$. The objects are not stored in the index itself. Rather, for each object p, the RNN-Tree manages a sphere with radius $nndist_1(p)$, i.e. the data nodes of the tree contain spheres around objects. The RdNN-Tree [YL01] extends the RNN-Tree by storing the objects of the database itself rather than circles around them. For each object p, the distance to p's 1-nearest neighbor, i.e. $nndist_1(p)$ is aggregated. 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. In general, 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. Both, the RNN-Tree and the RdNN-Tree, can be extended to metric spaces (e.g. by applying an M-Tree [CPZ97] instead of an R-Tree). However, since the kNN distance needs to be materialized, it is limited to a fixed k and cannot be generalized to answer RkNN-queries with arbitrary k. To overcome this problem, the MRkNNCoP-Tree [ABK$^+$06b] has been proposed recently. The index is conceptually similar to the RdNN-Tree but stores a conservative and progressive approximation for all kNN distances of any data object rather than the exact kNN distance for one fixed k. The only limitation is that k is upper-bounded by a parameter k_{max}. For RkNN queries with $k > k_{max}$, the MRkNNCoP-Tree cannot be applied [ABK$^+$06a]. The conservative and progressive approximations of any index node are propagated to the parent nodes. Using these approximations, the MRkNNCoP-Tree can identify a candidate set, true hits, and true drops. For each object in the candidate set, a kNN query need to be launched for refinement.

2.3 Contributions

Our solution is conceptually similar to that in [ABK$^+$06b] but extends this work and all other existing approaches in several important aspects. In particular, our method provides the following new features:

1. Our solution is applicable for RkNN search using any value of k because our approximation can be interpolated for any $k \in \mathbb{N}$. In contrast, most previous methods are limited to RkNN queries with one predefined, fixed k or $k \leq k_{max}$.

2. Our distance approximation is much smaller than the approximations proposed in recent approaches and, thus, produces considerably less storage overhead. As a consequence, our method leads to a smaller index directory resulting in significantly lower query execution times.

3. In contrast to several existing approaches, our method does not need to perform kNN queries in an additional refinement step. This also dramatically reduces query execution times.

4. Our distance approximations can be generated from a small sample of kNN distances (the kNN distances of any $k \in \mathbb{N}$ can be interpolated from these approximations). Thus, the time for index creation is dramatically reduced.

In summary, our solution is the first approach that can answer RkNN queries for any $k \in \mathbb{N}$ in general metric databases. Since our solution provides superior performance but approximate results, it is applicable whenever efficiency is more important than complete results. However, we will see in the experimental evaluation that the loss of accuracy is negligible.

3 Approximate Metric RkNN Search

As discussed above, the only existing approach to RkNN search that can handle arbitrary values of k at query time and can be used for any metric objects (not only for Euclidean feature vectors) is the MRkNNCoP-Tree [ABK$^+$06b] that extends the RdNN-tree by using conservative and progressive approximations for the kNN distances. This approach, however, is optimized for exact RkNN search and is limited to a its flexibility regarding the parameter k is limited by an additional parameter k_{max}. This additional parameter must be specified in advance, and is an upper bound for the value of k at query time. If a query is launched specifying a $k > k_{max}$, the MRkNNCoP-Tree cannot guarantee complete results. In our scenario of answering approximate RkNN queries, this is no problem. However, since the MRkNNCoP-Tree constraints itself to compute exact results for any query with $k \leq k_{max}$, it generates unnecessary overhead by managing conservative and progressive approximations. In general, an index for approximate RkNN search does not need to manage conservative and progressive approximations of the kNN distances of each object but only needs one approximation.

Thus, for each object, instead of two approximations (a conservative and a progressive) of the kNN distances which is bound by a parameter k_{max}, we store one approximation of the kNN distances for any $k \in \mathbb{N}$. This approximation is represented by a function, i.e. the approximated kNN distance for any value $k \in \mathbb{N}$ can be calculated by applying this function. Similar to existing approaches, we can use an extended M-Tree, that aggregates for

each node the one approximation of the 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 kNN distance for each node using this approximation in order to prune nodes analogously to the way we can prune objects. Since the approximation does not ensure completeness, the results may contain false positives and may miss some true drops. As discussed above, this is no problem since we are interested in an approximate RkNN search scenario.

In the following, we introduce how to compute an approximation of the kNN distances for arbitrary $k \in \mathbb{N}$. After that, we describe how this approximation can be integrated into an M-Tree. At the end of this section, we outline our approximate RkNN search algorithm.

3.1 Approximating the kNN Distances

A suitable model function for the approximation of our kNN distances for every $k \in \mathbb{N}$ should obviously be as compact as possible in order to avoid a high storage overhead and, thus, a high index directory.

In our case, we can assume that the distances of the neighbors of an object o are given as a (finite) sequence

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

for any $k_{max} \in \mathbb{N}$ 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 $f_o : \mathbb{N} \to \mathbb{R}$ with $f_o(k) \approx nndist_k(o)$. As discussed above, such a function should allow us to calculate an approximation of the kNN distance for any k, even for $k > k_{max}$ by estimating the corresponding values.

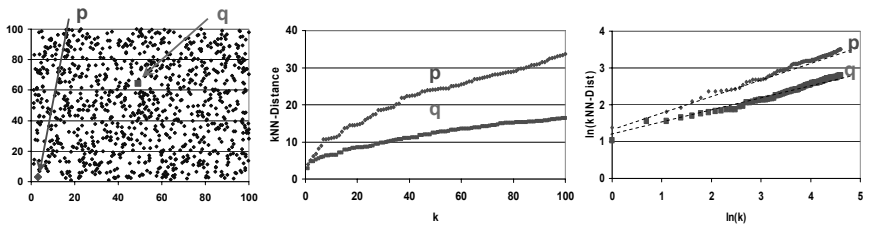
From the theory of self-similarity [Sch91] it is well-known that in most data sets the relationship between the number of objects enclosed in an arbitrary hypersphere and the scaling factor (radius) of the hypersphere (the same is valid for other solids such as hypercubes) approximately follows a power law:

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

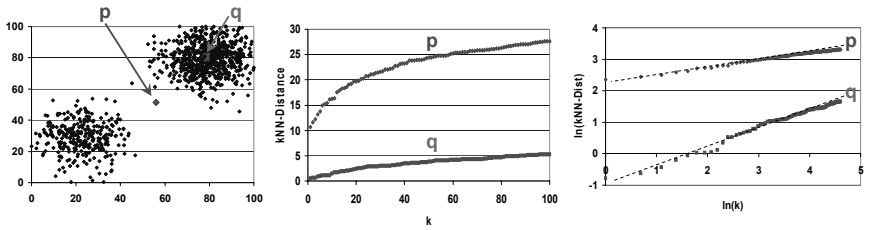
where ε is the scaling factor, $encl(\varepsilon)$ is the number of enclosed objects and d_f is the fractal dimension. The fractal dimension is often (but not here) assumed to be a constant which characterizes a given data set. Our kNN sphere around any object $o \in \mathcal{D}$ can be understood to be such a scaled hypersphere where the distance of the kNN is the scaling factor and k is the number of enclosed objects. Thus, it can be assumed that the kNN distances also follow the power law, i.e.

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

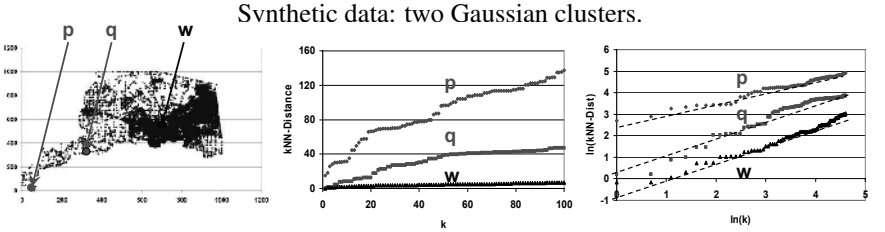
Transferred in log-log space (for an arbitrary logarithmic basis, e.g. for basis e), we have



Synthetic uniform data distribution.



Svnthetic data: two Gaussian clusters.



Real-world data set: Sacramento landmarks.

Figure 2: Illustration of the relationships between k and the kNN distance for different data distributions.

a linear relationship [Sch91]:

$$\log(nndist_k(o)) \propto \frac{1}{d_f} \cdot \log(k).$$

This linear relationship between k and the kNN distance in log-log space is illustrated for different sample data distributions and a sample 2D real-world data set[1] in Figure 2. Obviously this linear relationship is not perfect. However, as it can be anticipated from Figure 2, the relationship between $\log(k)$ and $\log(nndist_k(o))$ for any object o in a database of arbitrary distribution, exhibit a clear linear tendency.

From this observation, it follows that it is generally sensible to use a model function which is linear in log-log space — corresponding to a parabola in non-logarithmic space — for the approximation. Obviously, computing and storing a linear function needs considerably less overhead than a higher order function. Since we focus in this section on the approximation of the values of the kNN distance over varying k in a log-log sense, we consider

[1]The real-world data represents the spatial coordinates of landmarks in Sacramento, CA. The data originates from: http://www.census.gov

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.

Like in most other applications of the theory of self-similarity, we need to determine a classical regression line that approximates the true values of $nndist_k(o)$ with least square error. A conventional regression line $f_o(x) = m_o \cdot x + t_o$ would find the parameters (m_o, t_o) minimizing least square error:

$$\sum_{k=1}^{k_{max}} (y_k - (m_o \cdot \log k + t_o))^2 \rightarrow \min$$

where $y_k = \log nndist_k(o)$, which evaluates the well known formula of a regression line in 2D space. As indicated above, since this line is the best approximation of a point set, it is exactly the approximation of the kNN distances we want to aggregate. In other words, for each object $o \in \mathcal{D}$, we want to calculate the function $f_o(x) = m_o \cdot x + t_o$ that describes the regression line of the point set $\{(\log k, \log nndist_k(o)) \mid 1 \leq k \leq k_{max}\}$.

From the theory of linear regression, the parameters m_o and t_o can be determined as

$$m_o = \frac{(\sum_{k=1}^{k_{max}} y_k \cdot \log k) - k_{max} \cdot \bar{y} \cdot \frac{1}{k_{max}} \sum_{k=1}^{k_{max}} \log k}{(\sum_{k=1}^{k_{max}} (\log k)^2) - k_{max} \cdot (\frac{1}{k_{max}} \sum_{k=1}^{k_{max}} \log k)^2}$$

where $\bar{y} = \frac{1}{k_{max}} \sum_{k=1}^{k_{max}} \log nndist_k(o)$, and

$$t_o = \bar{y} - m_o \cdot \frac{1}{k_{max}} \sum_{k=1}^{k_{max}} \log k.$$

3.2 Aggregating the Approximations

So far, we have shown how to generate an accurate approximation for each object of the database. When using a hierarchically organized index structure, the approximation can also be used for the nodes of the index to prune irrelevant sub-trees. Usually, each node N of the index is associated with a page region representing a set of objects in the subtree which has N as root. In order to prune the subtree of node N, we need to approximate the kNN distances of all objects in this subtree, i.e. page region. If the distance between the query object q and the page region of N, called MINDIST, is larger than this approximation, we can prune N and thus, all objects in the subtree of N. The MINDIST is a lower

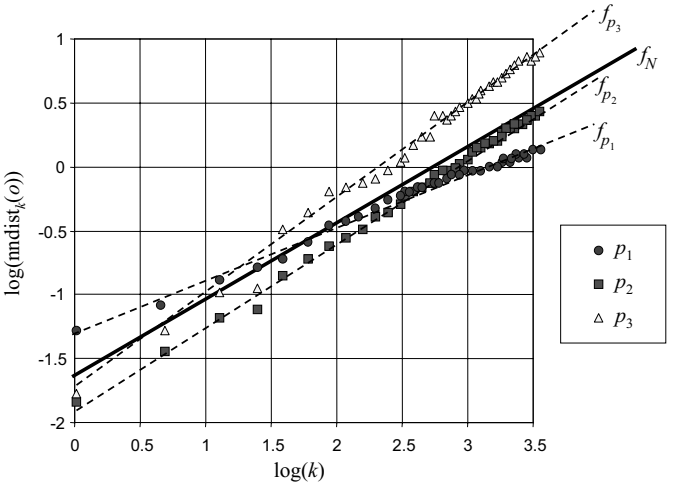


Figure 3: Visualization of the aggregated approximation f_N for a node N containing objects p_1, p_2, p_3.

bound for the distance of q to any of the objects in N. The aggregated approximation should again estimate the kNN distances of all objects in the subtree representing N with least squared error. This is a little more complex than a simple regression problem.

Obviously, given a data node N with $|N|$ data objects $o_i \in N$, the parameters of the optimal regression line $F_N(x) = m_N \cdot x + t_N$ that approximates the kNN distances of all objects in N can be determined as follows:

$$m_N = \frac{\sum\limits_{o_i \in N} \left(\sum\limits_{k=1}^{k_{max}} y_k^{o_i} \cdot \log k \right) - \frac{k_{max}}{|N|} \cdot \sum\limits_{o_i \in N} \bar{y}^{o_i} \cdot \frac{|N|}{k_{max}} \sum\limits_{k=1}^{k_{max}} \log k}{|N| \cdot \left(\sum\limits_{k=1}^{k_{max}} (\log k)^2 \right) - k_{max} \cdot \left(\frac{1}{k_{max}} \sum\limits_{k=1}^{k_{max}} \log k \right)^2}$$

and

$$t_N = \frac{1}{|N|} \sum_{o_i \in N} \bar{y}^{o_i} - m_o \cdot \frac{1}{k_{max}} \sum_{k=1}^{k_{max}} \log k,$$

where $y_k^{o_i} = \log nndist_k(o_i)$ and

$$\bar{y}^{o_i} = \frac{1}{k_{max}} \sum_{k=1}^{k_{max}} \log nndist_k(o_i).$$

The first equation can be reformulated as

$$m_N = \frac{\sum_{o_i \in N} (\sum_{k=1}^{k_{max}} y_k^{o_i} \cdot \log k) - \sum_{o_i \in N} \bar{y}^{o_i} \cdot \sum_{k=1}^{k_{max}} \log k}{|N| \cdot (\sum_{k=1}^{k_{max}} (\log k)^2) - \frac{1}{k_{max}} (\sum_{k=1}^{k_{max}} \log k)^2}$$

Thus, in order to generate an optimal approximation f_N for any directory node N with child nodes C_i, we need to aggregate $\sum_{o_i \in C_i} \sum_{k=1}^{k_{max}} y_k^{o_i}$ and $\sum_{o_i \in C_i} \bar{y}^{o_i}$ for each C_i. Thus, we store for each child nodes C_i two additional values

$$v_1 = \sum_{o_i \in C_i} \sum_{k=1}^{k_{max}} y_k^{o_i}$$

and

$$v_2 = \sum_{o_i \in C_i} \bar{y}^{o_i}$$

in order to compute the distance approximation of the parent node N. Obviously, the required storage overhead is negligible. On the other hand, we can now generate for each node N in the tree the optimal regression line for the kNN distances of all objects located in the subtree of N.

The idea of aggregating the kNN distance approximations for directory nodes is visualized in Figure 3. The approximation f_N of a node N representing objects p_1, p_2, p_3 is depicted. The regression line f_N approximates the kNN distances of p_1, p_2, p_3 with least square error.

We call the resulting index structure AMRkNN-Tree (Approximate Metric Reverse kNN-Tree). The original concepts of the AMRkNN-Tree presented here can be incorporated within any hierarchically organized index for metric objects. Obviously, our concepts can also be used for RkNN search in Euclidean data by integrating the approximation into Euclidean index structures such as the R-tree [Gut84], the R*-tree [BKSS90], or the X-tree [BKK96].

3.3 RkNN Search Algorithm

The algorithm for approximate RkNN queries on our novel AMRkNN-Tree is similar to the exact RkNN query algorithms of the RdNN-Tree and the MRkNNCoP-Tree. However, our index structure can answer RkNN queries for any k specified at query time. Let us point out that the value of k is not bound by a predefined k_{max} parameter, although the approximation of the kNN distances are computed by using only the first k_{max} values, i.e. the kNN distances with $1 \leq k \leq k_{max}$. The kNN distance for any $k > k_{max}$ can be extrapolated by our approximations in the same way as for any $k \leq k_{max}$. In addition, due

```
Approximate_RkNN_query(𝒟, q, k)
  // 𝒟 is assumed to be organized as AMRkNN-Tree
  queue := new Queue;
  insert root of AMRkNN-Tree into queue;
  while not queue.isEmpty()
    N := queue.getFirst();
    if N is node then
      if MINDIST(N, q) ≤ m_N · log k + t_N then
        insert all elements of N into queue;
      end if
    else // N is a point
      if log(dist(N, q)) ≤ m_N · log k + t_N then
        add N to result set;
      end if
    end if
  end while
```

Figure 4: Algorithm for approximate RkNN query.

to the use of a metric index structure, our AMRkNN-Tree is applicable to general metric objects.

Similar to the M-Tree concept, a node N of our AMRkNN-Tree is represented by its routing object N_o and the covering radius N_r. All objects represented by node N have a distance less than N_r to N_o. The logarithm of the aggregated kNN distance of a node N, denoted by $k\text{NN}_{agg}(N)$ can be determined from the approximation $f_N(x) = m_N \cdot x + t_N$ of N by

$$k\text{NN}_{agg}(N) = m_N \cdot \log k + t_N.$$

Note that the true (i.e. non-logarithmic) approximation of the aggregated kNN distance of N is $e^{k\text{NN}_{agg}(N)}$. To avoid unnecessary complex computations, we adapt the definition of the MINDIST between a node and a point to the logarithmic scale of $k\text{NN}_{agg}(N)$. Thus, the MINDIST of a node N and a query point q, denoted by MINDIST(N, q), is defined as

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

The pseudo code of the approximate RkNN query algorithm is depicted in Figure 4. A query q is processed by traversing the index from the root of the index to the leaf level. A node N needs to be refined if the MINDIST between q and N is smaller than the aggregated kNN distance approximation of n, i.e. MINDIST$(q, N) \leq k\text{NN}_{agg}(N)$. Those nodes, where the MINDIST to q is larger than their aggregated kNN distance approximation are pruned, i.e. if MINDIST$(N, q) > k\text{NN}_{agg}(N)$.

The traversal ends up at a data node. Then, all points p inside this node are tested using their approximation $f_p(x) = m_p \cdot x + t_p$. A point p is a hit if

$$\log(dist(N, q)) \leq m_N \cdot \log k + t_N.$$

Metric datasets

Name	# objects
Road network	18,236
Sequence	10,000

Euclidean datasets

Name	# objects	dimension
SEQUOIA	100,000	5
ColorMoments	68,040	9
CoocTexture	68,040	16

Table 1: Real-world datasets used for our experiments.

Otherwise, if $\log(dist(N,q)) > m_N \cdot \log k + t_N$, point p is a miss and should be discarded.

In contrast to other approaches that are designed for RkNN search for any k, our algorithm directly determines the results. In particular, we do not need to apply an expensive refinement step to a set of candidates. This further avoids a significant amount of execution time.

4 Evaluation

All experiments have been performed on Windows workstations with a 32-bit 4 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 dataset and reported the average results. 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.

We compared our AMRkNN-Tree with the index proposed in [ABK$^+$06b] that is designed for exact RkNN search in general metric spaces for any $k \leq k_{max}$ and the sequential scan. The approach in [ABK$^+$06b] claims to outperform all other approaches on general metric data as well as on Euclidean data. We will show, that our AMRkNN-Tree is much more efficient than this state-of-the-art approach on both general metric data and Euclidean data.

4.1 Datasets

Metric RkNN search. Our experiments were performed using two real-world datasets. The first one is a road network dataset derived from the city of San Juan, CA, which contains 18,236 nodes and 23,874 edges. The average degree of the nodes in this network is 2.61. The dataset 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 dataset consists of 10,000 protein sequences taken from SWISSPROT database[3], the Levenstein distance was used as similarity distance. For

[2] www.fh-oow.de/institute/iapg/personen/brinkhoff/generator/
[3] http://www.expasy.org/sprot/

both datasets we used an M-Tree with a node size of 4 KByte.

Euclidean RkNN search. We also integrated our concepts into an X-Tree [BKK96] in order to support RkNN search in Euclidean data. We used three real-world datasets for our experiments including a set of 5-dimensional vectors generated from the well-known SEQUOIA 2000 benchmark dataset and two "Corel Image Features" benchmark datasets from the UCI KDD Archive[4]. The first Corel Image dataset contains 9 values for each image ("ColorMoments"), the second Corel Image dataset contains 16-dimensional texture values ("CoocTexture"). The underlying X-Tree had a node size of 4 KByte.

The characteristics of the real-world datasets used for our evaluation are summarized in Table 1.

4.2 Comparison to competing approaches in Euclidean space

In Euclidean space, there exist two competitors PDE and kDE [XLOH05] as discussed in Section 2.2. In an initial setup, we compare the performance of our approach to both competing approaches by measuring the average kNN-distance error. For all experiments, we set $k_{max} = 100$. The κ parameter for the competing techniques was set to 50. Figure 5(a-c) depicts the error for varying parameter k. Because PDE and kDE store the exact distance for $k = \kappa$, the error for both techniques decreases when k converges to κ. For $k \neq \kappa$, the distance approximations of PDE and kDE are significantly worse than those of our approach. For the 16-dimensional Corel Image dataset, our AMRkNN approach outperforms the competing techniques by a factor between 4 and 6, for $k \leq 30$ resp. $k \geq 70$. In a next experiment, we evaluated the error for varying database size, as depicted in Figure 5(d). The results show that the quality of the distance approximations for all three techniques is almost independent from the database size, i.e. is not affected by the density of the dataset.

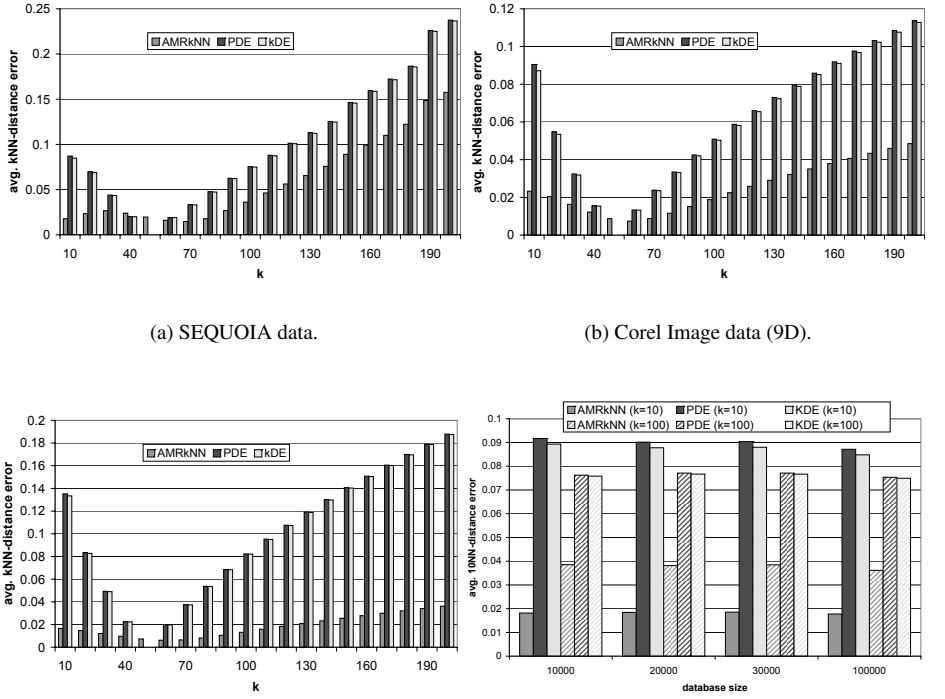
Because the quality of the distance approximations of the AMRkNN-Tree clearly outperforms the distance approximations of PDE and kDE for varying parameter k and varying database size, we do not take PDE and the kDE into account in the remaining experiments.

4.3 Runtime w.r.t. database size

We altered the number of database objects in order to evaluate the scalability of the competing methods w.r.t. the database size. Throughout all experiments, we set $k = 50$ and $k_{max} = 100$.

Metric RkNN search. A comparison of our novel index structure with the state-of-the-art approach applied to our real-world metric datasets is shown in Figure 6. It can be seen that our AMRkNN-Tree clearly outperforms the competing MRkNNCoP-Tree on the road

[4] http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html



(a) SEQUOIA data.

(b) Corel Image data (9D).

(c) Corel Image data (16D).

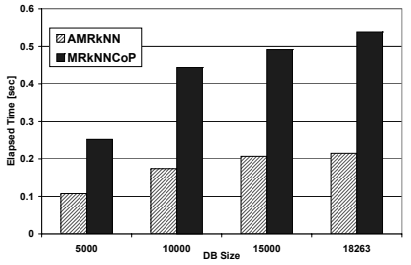
(d) SEQUOIA data.

Figure 5: Average kNN-distance error of competing methods w.r.t. parameter k (a, b, c) and database size (d) on Euclidean data.

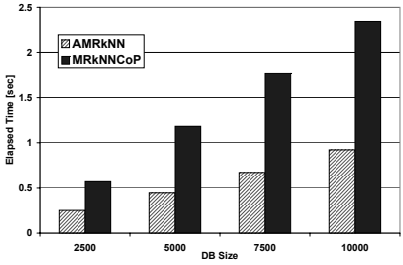
network dataset (cf. Figure 6(a)). The performance gain of our approach over the existing method also grows with increasing database size. Both approaches show a linear scalability w.r.t. the number of data objects, but the increase of runtime of our AMRkNN-Tree is smaller than the increase of runtime of the MRkNNCoP-Tree. The runtime of the sequential scan also grows linear with increasing number of database objects. It is not shown in Figure 6(a) for clearness reasons. In fact, we observed that the performance gain of our AMRkNN-Tree over the sequential scan grows with increasing database size from a factor of 150 to about 850.

A similar observation can be made on the dataset containing biological sequences. The results are illustrated in Figure 6(b). Again, the sequential scan is not shown due to clarity reasons.

Euclidean RkNN search. In Figure 7 a comparison of our novel index structure with the state-of-the-art approach applied to our real-world Euclidean datasets is presented. As it can be observed, our AMRkNN-Tree clearly outperforms the competing MRkNNCoP-Tree on all three datasets. In addition, the performance gain of our approach over the existing method also grows with increasing database size on all datasets. Both competing



(a) Road network dataset.



(b) Biological sequence dataset.

Figure 6: Scalability of competing methods w.r.t. the number of database objects on metric data (sequential scan is not shown for clarity reasons).

approaches show a linear scalability w.r.t. the number of data objects, but the increase of runtime of our AMRkNN-Tree is significantly smaller than the increase of runtime of the MRkNNCoP-Tree. The superiority of our AMRkNN-Tree is even more obvious on Euclidean data. The runtime of the sequential scan is also not shown in the charts presented in Figure 7 for clearness reasons. In fact, the sequential scan is outperformed by both methods by a factor of clearly over 100.

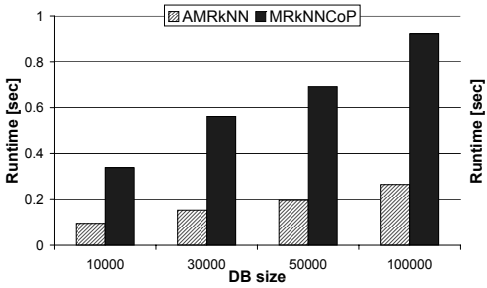
4.4 Runtime w.r.t. parameter k

We executed RkNN queries on a database with varying k and compared the scalability of both competing methods with the sequential scan. The parameter k_{max} was set to 100 for both approaches in all experiments.

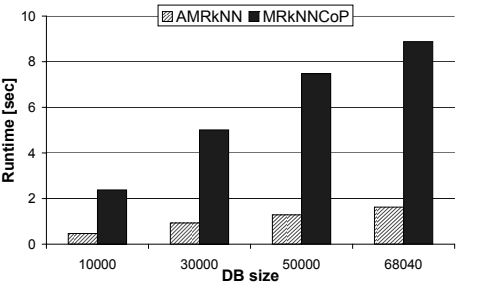
Metric RkNN search. The results of these experiments on the metric datasets are depicted in Figure 8. Applied to the road network dataset with 10,000 nodes, our novel AMRkNN-Tree clearly outperforms the current state-of-the-art approach (cf. Figure 8(a)). With increasing k, the performance gain of our method over the competitor further grows. The runtime of the sequential scan is independent of the choice of k and was observed at 140 seconds per query for any k. It is not shown in Figure 8(a) for clearness reasons.

A similar observation can be made when applying the competing methods to the dataset of 10,000 biological sequences. The results are illustrated in Figure 8(b). For clarity reasons, the runtime of the sequential scan (approx. 100 seconds) is again not shown. It can be observed that with increasing k, the performance gain of our method over the competitor is even stronger rising.

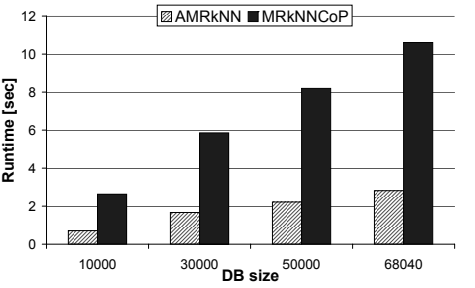
Euclidean RkNN search. The results of these experiments on the Euclidean datasets are depicted in Figure 9. All three datasets contained 50,000 objects. Applied to the SEQUOIA data, it can be seen that our approach scales linear with a very low slope. On the



(a) SEQUOIA data.



(b) Corel Image data (9D).



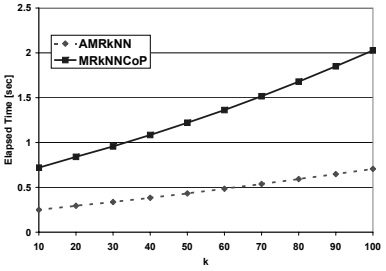
(c) Corel Image data (16D).

Figure 7: Scalability of competing methods w.r.t. parameter k on Euclidean data (sequential scan is not shown for clarity reasons).

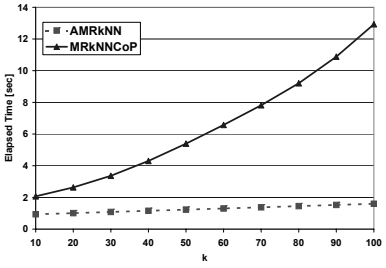
other hand, the MRkNNCoP-Tree exhibits a stronger rise of runtime. Similar observations can be made on the Corel Image datasets (cf. Figure 9(b) and Figure 9(c)). In summary, in almost all parameter settings, our novel AMRkNN-Tree is at least 4 times faster than the MRkNNCoP-Tree. The sequential scan scales constant for any value of k. The reported runtimes on the three Euclidean datasets of this naive solution are between 450 and 500 seconds. Those runtimes are not shown in Figure 9(a), Figure 9(b), and Figure 9(c) for clearness reasons.

4.5 Effectivness

The two probably most widespread concepts for measuring the effectivness are the *recall* and the *precision*. The recall measures the relative number of true hits reported as result, whereas precision measures the relative number of reported objects that are true hits. Usually, a user does not care so much about false positives, i.e. objects reported as hits that



(a) Road network dataset.



(b) Biological sequence dataset.

Figure 8: Scalability of competing methods w.r.t. parameter k on metric data (sequential scan is not shown for clarity reasons).

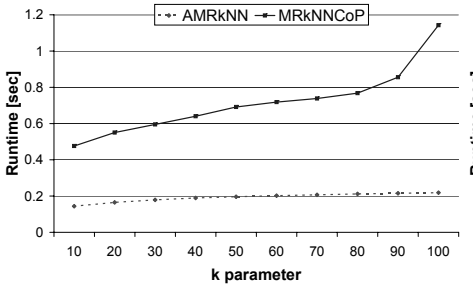
are true drops, as far as no true hits are missing. Thus, for measuring the quality of our approximate results, we focused on the recall. This measurement is the most important measurement to judge the quality of approximate results.

Metric RkNN search. We evaluated the effectiveness of our approximate RkNN search on our metric datasets. In this experiment, we set $k_{max} = 100$ and executed several RkNN queries for $10 \leq k \leq 200$. The results are depicted in Figure 10(a). As it can be seen, in almost all experiments, the recall is clearly above 90%. On the sequence dataset, the recall falls below 80% for low k values but rises significantly over 90% at about $k = 60$. This very accurate effectiveness is complemented by a rather high precision of the reported queries (between 80 - 97 %). It is worth mentioning, that the recall does not decrease significantly when answering RkNN queries with $k > k_{max}$. This observation confirms the claim that our AMRkNN-Tree is applicable to any $k \in \mathbb{N}$.

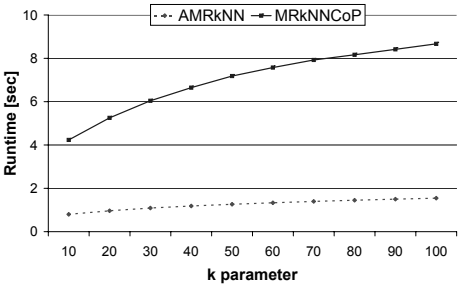
Euclidean RkNN search. A similar observation can be made when evaluating the recall of our method on the Euclidean datasets. Again we set $k_{max} = 100$ and executed several RkNN queries for $10 \leq k \leq 200$. The results are depicted in Figure 10(b). As it can be seen, for most parameter settings, the recall is clearly above 90%. Again we observed a rather high precision (between 80 - 98 %). We also want to point out that the recall does not decrease significantly when answering RkNN queries with $k > k_{max}$. Once again, this observation confirms the claim that our AMRkNN-Tree is applicable to any $k \in \mathbb{N}$.

5 Conclusions

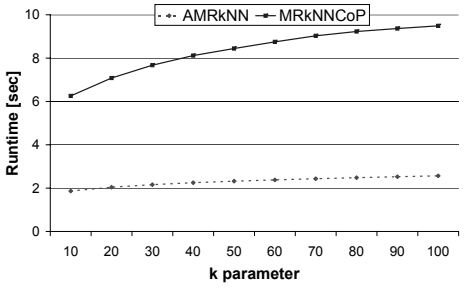
In this paper, we proposed the first solution for approximate RkNN search in general metric spaces for any $k \in \mathbb{N}$. Our approach is based on the observation known from the theory of self-similarity that the relationship between k and the kNN distance of any object is linear in log-log space. We proposed to calculate an approximation of the kNN



(a) SEQUOIA data.



(b) Corel Image data (9D).



(c) Corel Image data (16D).

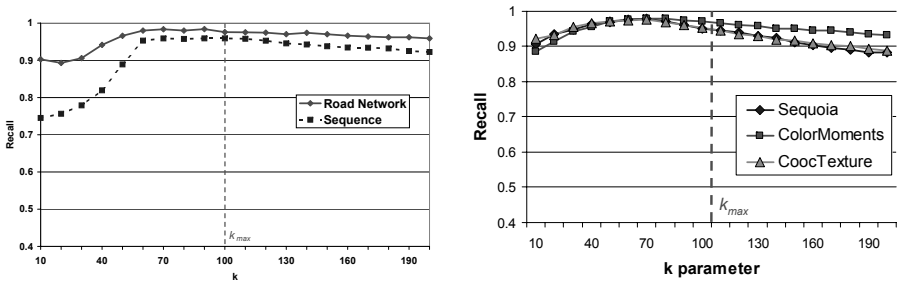
Figure 9: Scalability of competing methods w.r.t. parameter k on Euclidean data (sequential scan is not shown for clarity reasons).

distances of any database object by means of a regression line in the log-log space from a set of sample kNN distances. The kNN distance of any k can then be interpolated from this approximation. We showed how these approximations can be integrated into any hierarchically organized index structure (e.g. the M-Tree for metric objects or the R-Tree for Euclidean vectors) by propagating the approximations of child nodes into parent nodes. Our resulting index called AM$R k$NN-Tree has achieved significant performance boosts compared to existing approaches. In addition, our experiments showed that our performance gain caused only a negligible loss in accuracy.

For future work, we will examine parallel and distributed solutions to the RkNN problem.

References

[ABK+06a] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Approximate Reverse k-Nearest Neighbor Queries in General Metric Spaces. In *Proc. CIKM*, 2006.



(a) Metric datasets.

(b) Euclidean datasets.

Figure 10: Recall of our method on real-world datasets.

[ABK+06b] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Efficient Reverse k-Nearest Neighbor Search in Arbitrary Metric Spaces. In *Proc. SIGMOD*, 2006.

[BKK96] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. In *Proc. VLDB*, 1996.

[BKSS90] 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*, pages 322–331, 1990.

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proc. VLDB*, 1997.

[DP03] Chris Ding and Hanchuan Peng. Minimum Redundancy Feature Selection from Microarray Gene Expression Data. In *CSB03*, 2003.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. SIGMOD*, pages 47–57, 1984.

[KM00] F. Korn and S. Muthukrishnan. Influenced Sets Based on Reverse Nearest Neighbor Queries. In *Proc. SIGMOD*, 2000.

[SAA00] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. In *Proc. DMKD*, 2000.

[Sch91] M. Schroeder. *Fractals, Chaos, Power Laws: Minutes from an infinite paradise*. W.H. Freeman and company, New York, 1991.

[SFT03] Amit Singh, Hakan Ferhatosmanoglu, and Ali Saman Tosun. High Dimensional Reverse Nearest Neighbor Queries. In *Proc. CIKM*, 2003.

[TPL04] Yufei Tao, Dimitris Papadias, and Xiang Lian. Reverse kNN Search in Arbitrary Dimensionality. In *Proc. VLDB*, 2004.

[XLOH05] C. Xia, H. Lu, B. C. Ooi, and J. Hu. ERkNN: Efficient Reverse k-Nearest Neighbors Retrieval with Local kNN-Distance Estimation. In *Proc. CIKM*, 2005.

[YL01] Congjun Yang and King-Ip Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. ICDE*, 2001.