Approximate direct and reverse nearest neighbor queries, and the *k*-nearest neighbor graph

Karina Figueroa †
Facultad de Ciencias Físico Matemáticas
Universidad Michoacana
Morelia, Mexico
e-mail: karina@fismat.umich.mx

Rodrigo Paredes ‡
Dept. of Computer Science
University of Chile
Santiago, Chile
e-mail: raparede@dcc.uchile.cl

***Abstract*—Retrieving the *k-nearest neighbors* of a query object is a basic primitive in similarity searching. A related, far less explored primitive is to obtain the dataset elements which would have the query object within their own *k*-nearest neighbors, known as the *reverse k-nearest neighbor* query. We already have indices and algorithms to solve *k*-nearest neighbors queries in general metric spaces; yet, in many cases of practical interest they degenerate to sequential scanning. The naive algorithm for reverse *k*-nearest neighbor queries has quadratic complexity, because the *k*-nearest neighbors of all the dataset objects must be found; this is too expensive. Hence, when solving these primitives we can tolerate trading correctness in the solution for searching time. In this paper we propose an efficient approximate approach to solve these similarity queries with high retrieval rate. Then, we show how to use our approximate *k*-nearest neighbor queries to construct (an approximation of) the *k-nearest neighbor graph* when we have a fixed dataset. Finally, combining both primitives we show how to *dynamically maintain* the approximate *k*-nearest neighbor graph of the objects currently stored within the metric dataset, that is, considering both object insertions and deletions.**

I. Introduction

Given an object q and a dataset $\mathbb{U}$ of size n, the *k-nearest neighbor query* ($NN_k(q)$) retrieves the k elements from $\mathbb{U}$ closest to q. This primitive is a building block for a large number of problems in a wide number of application areas. For instance, in pattern classification, the nearest-neighbor rule can be implemented with $NN_1(q)$'s [1]. $NN_k(q)$'s are also a fundamental tool in cluster and outlier detection [2], [3], image segmentation [4], query or document recommendation systems [5], and so on.

A related, but far less explored primitive is the *reverse k-nearest neighbor query* ($RNN_k(q)$), that is, given an object q finding the dataset elements which have q within their own k-nearest neighbors (kNNs). This primitive is quite expensive to compute since we need the kNNs for several database objets (or maybe, for them all) in order to verify the correctness of the query outcome. This similarity query has interesting applications. For instance, let us consider the problem of placing a new supermarket in a given location. We could perform several reverse 1NN queries in order to find a place such that many residential areas in that location would find the new outlet as their nearest choice. Reverse kNN queries can also be used in profile-based marketing, cluster and outlier detection, geographic information systems, traffic networks, adventure games, or molecular biology (see [6], [7] for further details). Finally, in this paper we use reverse kNN queries for dynamic kNN graphs.

As can be seen, direct and reverse k-nearest neighbors are fair choices for several problems. Also, the kNN approach is simple, has a small number of parameters to tune up, has zero training time, can be adapted to database changes over time, and has excellent classification performance.

Despite the advantages of the kNN approach, in real-world applications it is seldom used outside some toy examples, such as considering small databases or in low-dimensional vector-spaces. This is because real-world data are medium or high dimensional, or have no coordinates at all, for instance strings. In these cases, one needs to resort to the *metric space search* model, where objects are treated as black boxes and the similarity among them is computed with a metric (comprehensive surveys and books are [8]–[11]).

When we model similarity as a metric space, we are already approximating the real retrieval need of users. In fact, given a dataset, we can use several distance functions, each of them considering some aspects of objects and neglecting others. Likewise, when we design a model to represent real-life objects, we usually lose some information. Think, for instance, in the vector representation of a document. This representation does not consider either positions of the words composing the document, the document structure, or the semantic. Moreover, even if we find the proper metric and a lossless object representation, there are high-dimensional metric spaces where solving similarity queries requires reviewing almost all the dataset no matter what strategy we use. In addition, in many applications, the efficiency of the query execution is much more important than effectiveness. That is, users want a fast response to their queries and will even accept approximate results (as far as the number of false drops and hits are moderate). This has given rise to a

† Partially funded by CONACyT, Mexico.
‡ Partially funded by MEC grant TIN2006-15071-C03-03, Spain. Part of this work was made while visiting the LBD, Univ. of Coruña, Spain.

new approach to the similarity search problem: We try to find the objects relevant to a given query with high probability. An intuitive notion of what this approach aims to is that it attempts not to miss many relevant objects at query time.

Our contribution is based on the fact that a $NN_k(q)$ defines a dataset search order given by the distance to the query. All we need to solve a $NN_k(q)$ are the first k elements in this order, while the remaining ones are not considered at all. Calculating this order in the original metric space is expensive as it implies computing n distances. Our idea is to use an alternative search order, which is cheaper to compute than and is rather similar to the real order, especially with respect to the elements we care the most, the first ones. The fundamental idea of the alternative order was presented in [12] for metric range searching. In this paper we use the alternative search order to efficiently solve direct and reverse kNN queries with high probability. Then, we apply these primitives to construct *k-nearest neighbor graphs* (kNNGs) and to update the kNNG upon object insertions and deletions.

II. Related work

1) A summary of metric space searching: A metric space is a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is the universe of valid objects and $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+ \cup \{0\}$ is distance function defined among them. Objects in $\mathbb{X}$ do not necessarily have coordinates. The distance function gives us a dissimilarity criterion to compare objects from the universe. Thus, the smaller the distance between two objects, the more "similar" they are. Function d satisfies the following properties: symmetry $d(x, y) = d(y, x)$, reflexivity $d(x, x) = 0$, strict positiveness $d(x, y) > 0 \iff x \neq y$, and the triangle inequality $d(x, y) \leq d(x, z) + d(z, y)$, $\forall\ x, y, z \in \mathbb{X}$.

Typically, we have a finite *database* or *dataset* $\mathbb{U}$ of size n, which is a subset of the universe. Later, given an object $q \in \mathbb{X}$, a proximity query consists in retrieving objects from $\mathbb{U}$ relevant to q. There are two basic proximity queries or primitives: The *range query* (q, r) retrieves all the elements in $\mathbb{U}$ which are within distance r to q. That is, $(q, r) = \{x \in \mathbb{U}, d(x, q) \leq r\}$. The *k-nearest neighbor query* $NN_k(q)$ retrieves the k elements from $\mathbb{U}$ closest to q. That is, $NN_k(q)$ such that $\forall\ x \in NN_k(q)$, $y \in \mathbb{U} \setminus NN_k(q)$, $d(q, x) \leq d(q, y)$, and $|NN_k(q)| = k$ (in case of ties we choose any k-element set satisfying the condition). The covering radius $cr_{q,k}$ of a query $NN_k(q)$ is the distance from q towards the farthest neighbor in $NN_k(q)$.

In this paper, we are also dealing with the *reverse k-nearest neighbor query* $RNN_k(q)$, which retrieves the elements from $\mathbb{U}$ having q in their own k-nearest neigbors. That is, $RNN_k(q)$ such that $\forall\ x \in RNN_k(q) \Rightarrow d(q, x) \leq cr_{x,k}$. This primitive does not necessarily retrieve k objects.

Given the dataset $\mathbb{U}$, range and kNN queries can be trivially answered by performing n distance evaluations. On the other hand, reverse kNN queries requires n^2 distance evaluations. However, as the distance is assumed to be expensive to compute (think, for instance, in comparing two fingerprints), it is customary to define the complexity of the search as the number of distance evaluations performed, disregarding other components such as CPU time for side computations and even I/O time. Thus, the ultimate goal is to build *offline* an index in order to speed up *online* queries.

2) k-Nearest neighbors: Due to its importance in several application fields, the literature on kNN searching is abundant. The most efficient algorithms are focused on 1NN queries in vector spaces. The techniques used are standard kd-trees for dimension two and R-trees for dimension up to four, as described in [13]. So, the problem could be considered solved for low-dimensional vector-spaces. For high-dimensional vectors, in [14] the authors present a probabilistic algorithm using dimensionality reduction techniques that retrieves the nearest neighbor about 50% of the times. Dimensionality reduction techniques work when the data lie in a lower dimensional manifold compared to the representational (number of coordinates) dimension, a fortunate condition which is not always present. Finally, using metric indices, we solve kNN queries using $O(n^\alpha)$ distance computations, where $\alpha \leq 1$ is a parameter depending on both the particular metric space and the index used. Yet, there are several cases where traditional metric search algorithms degenerate to sequential scanning.

3) k-Nearest neighbor graphs: The kNNG of the set $\mathbb{U}$ is a weighted directed graph $G(\mathbb{U}, E)$ connecting each element $u \in \mathbb{U}$ to its kNNs, thus $E = \{(u, v), v \in NN_k(u)\}$, where in this case $NN_k(u) \subseteq \mathbb{U} \setminus \{u\}$. Beside the applications already mentioned for kNN queries, kNNGs can also be used in metric space searching [15], [16], VLSI design, spin glass and other physical process simulations [17], and so on.

The kNNG can be constructed by solving a $NN_k(u)$ for each $u \in \mathbb{U}$. There are techniques to speed up the procedure in vector spaces [17]–[22]. Most of them assume that nodes are points in $\mathbb{R}^D$ and that d is Euclidean or some Minkowski distance, which is not the case in several applications where kNNGs are required nor is suitable for general metric spaces (in [19] the vector space limitation is eliminated but the algorithm demands polynomial space to be implemented).

We also have alternatives for general metric spaces [15], [23]–[27]. In [23], the problem is solved using randomization in $O\big(n \log^2 n \log^2 \Gamma(\mathbb{U})\big)$ expected time. Here, $\Gamma(\mathbb{U})$ is the distance ratio between the farthest and closest pairs of points in $\mathbb{U}$. The author argues that in practice $\Gamma(\mathbb{U}) = n^{O(1)}$, in which case the approach is $O\big(n \log^4 n\big)$ time. However, the analysis needs a sphere packing bound in the metric space. Otherwise the cost must be multiplied by "sphere volumes", that are also exponential on the dimensionality. Moreover, the algorithm needs $\Omega\big(n^2\big)$ space for high dimensions, which is too much for practical applications.

In [25], the authors present the *metric skip list*. It uses $O(n \log n)$ space and can be constructed with $O(n \log n)$ distance computations. It answers 1NN queries using

$O(\log n)$ distance evaluations with high probability. Later, in [26], other authors introduce *navigating nets*. It can also be constructed with $O(n \log n)$ distance computations, yet using $O(n)$ space. It gives an $(1+\varepsilon)$-approximation algorithm to solve 1NN queries in time $O(\log n) + (1/\varepsilon)^{O(1)}$. Both indices could serve to solve the 1NNG problem with $O(n \log n)$ distance computations but not to build kNNGs. In addition, the hidden constants are exponential on the intrinsic dimension, which makes these approaches useful only in low-dimensional metric spaces.

In [24], the author uses a FQTrie [28] in order to speed up the n $NN_k(q)$'s. Later, this idea is improved in [15], [27], where the authors propose a general kNNG construction methodology for metric spaces. They also plug into the methodology a generic pivot-based index and a variant of the BST [29], in order to obtain two concrete algorithms.

A dynamic kNNG construction is a hard task since every object insertion/deletion can affect several sets of kNNs in the current graph. We are unaware of any work on this topic.

4) Reverse k-nearest neighbors: The techniques described in [14], [30]–[32] support reverse 1NN queries, which are too restrictive for our purposes. The works in [7], [33] are specific for the metric space scenario and support reverse kNN queries for values of $k > 1$. However, we do not consider them in the experimental evaluation (Section IV), as they are exact approaches focused in low dimensional metric spaces, and optimized in order to reduce I/Os. (Ours is approximated, focused in high dimensional spaces, and optimized in order to reduce distance evaluations.) Thus, a experimental comparison is not fair or meaningful.

In [6] the authors claim to be the first solution for approximate reverse kNN searching in general metric spaces for any k. This solution assumes that the number k of objects enclosed in an arbitrary hypersphere centered in an object $u \in \mathbb{U}$ follows a power low $k \propto cr_{u,k}^{d_f}$, where d_f is the fractal dimension of the space. Then, they solve a regression model $\ln cr_{u,k} \propto \frac{\ln k}{d_f}$ for each object $u \in \mathbb{U}$ so as to estimate $cr_{u,k}$, given the parameter k and the object u.

5) The permutation index: Let $\mathbb{A} \subseteq \mathbb{U}$ be a subset of *anchors*. Each element $u \in \mathbb{U}$ induces a *preorder* $\leq_u$ of the anchors given by the distance to u, defined as $y \leq_u z \iff d(u,y) \leq d(u,z)$, for any anchor pair $y, z \in \mathbb{A}$. The relation $\leq_u$ is a preorder and not an order because some anchors can be at the same distance of u, and then it could be possible to find two anchors $y \neq z$ such that $y \leq_u z \ \wedge \ z \leq_u y$.

Let $\Pi_u = i_1, i_2, \ldots, i_{|\mathbb{A}|}$ be the permutation of u, where anchor $a_{i_j} \leq_u a_{i_{j+1}}$. Anchors at the same distance take an arbitrary but consistent order. Every object in $\mathbb{U}$ computes its preorder of $\mathbb{A}$ and associates it to a permutation, which is stored in the index. Thus, the index needs $n|\mathbb{A}|$ space.

The crux of this index is that two equal objects must have the same permutation, while similar objects will hopefully have similar permutations. So if Π_u is similar to Π_q we expect that u is close to q. Thus, we have changed the problem from searching $\mathbb{U}$ to searching the permutation set.

At query time, we compute Π_q and compare it with all the permutations stored in the index. So, we traverse $\mathbb{U}$ in the order $\leq_{\Pi_q}$ induced by Π_q (by increasing permutation dissimilarity). If we limit the number of distance computations we obtain a probabilistic search algorithm. Fortunately, the order $\leq_{\Pi_q}$ induced is extremely promissory, as reported in [12] for range queries.

Similarity between the permutations of q and u can be measured by *Kendall Tau* (K_τ), *Spearman Footrule* (*SF*), or *Spearman Rho* (S_ρ) metric [34], among others. K_τ can be seen as the number of swaps that a bubble-sort-like algorithm has to do in order to make two permutations equal. Using $\Pi^{-1}(i_j)$ to denote the position of anchor a_{i_j} in the permutation Π, *SF* and S_ρ are defined as follows:

$$\begin{aligned} SF(\Pi_u, \Pi_q) &= \sum_{j\,=\,[1,|\mathbb{A}|]} |\Pi_u^{-1}(i_j) - \Pi_q^{-1}(i_j)|, \\ S_\rho(\Pi_u, \Pi_q) &= \sqrt{\sum_{j\,=\,[1,|\mathbb{A}|]} |\Pi_u^{-1}(i_j) - \Pi_q^{-1}(i_j)|^2}, \end{aligned}$$

as S_ρ is monotonous, we use S_ρ^2. For example, let $\Pi_q = (42153)$ and $\Pi_u = (32154)$ be the query and object $u \in \mathbb{U}$ permutations, respectively. So, $K_\tau(\Pi_u, \Pi_q) = 7$, $SF(\Pi_u, \Pi_q) = 8$, and $S_\rho^2(\Pi_u, \Pi_q) = 16$.

III. Our proposal

An obvious procedure to solve the $NN_k(q)$ is to report the first k objects in the order $\leq_q$, yet we need n distance evaluations to compute it. In this paper, we use the permutation index to estimate the order $\leq_q$ and solve kNN queries and related problems with high retrieval rate. Let us introduce the concept of *dominating order*.

Definition 1 (dominating order): Order $\leq_s$ dominates order $\leq_t$ at level C/k if the first C elements in order $\leq_s$ contain the first k elements in order $\leq_t$.

We intensively use the order $\leq_{\Pi_q}$ induced by the permutation of q as an "almost" dominant order for $\leq_q$ so as to solve kNN related problems. Even though we cannot guarantee that $\leq_{\Pi_q}$ is a truly dominant order for $\leq_q$, in practice we verify that for a level $C/k = O(1)$ we solve $NN_k(q)$'s and $RNN_k(q)$'s with high probability (that is, high retrieval rate), even in high dimensional spaces. Both the domination level and the function to measure similarity between permutations will be experimentally determined in Section IV. We assume we already have the permutation index and we use an abstract permutation similarity function PS. All the pseudocodes are given in Fig. 1.

1) k-Nearest neighbor queries: We compute the query permutation Π_q and then select the first C objects in the order $\leq_{\Pi_q}$. Next, we compute the distances between these objects and q and return the k-closest objects. **Approx**kNN implements this. It needs $|\mathbb{A}| + C$ distance computations.

```
Approx$k$NN(Obj $q$, Int $k$)
1. $\Pi_q \leftarrow$ compute the permutation of $q$ // $|\mathbb{A}|$ evals.
2. $pDist \leftarrow \{(u, PS(\Pi_q, \Pi_u)), u \in \mathbb{U}\}$
3. $\mathcal{C} \leftarrow$ selectSort($pDist, C$) // by perm. sim.
4. $distCq \leftarrow \{(c, d(c,q)), c \in \mathcal{C}\}$ // $C$ evals.
5. $k\mathrm{NN}_q \leftarrow$ selectSort($distCq, k$) // by dist. to $q$
6. Return ($k\mathrm{NN}_q, \Pi_q, distCq$)
```

```
ApproxR$k$NN(Obj $q$, Int $k$)
1. $\Pi_q \leftarrow$ compute the permutation of $q$ // $|\mathbb{A}|$ evals.
2. $pDist \leftarrow \{(u, PS(\Pi_q, \Pi_u)), u \in \mathbb{U}\}$
3. For $i \in [1, C]$ Do // $C + C^2$ dist. evals. over all
4.    $c \leftarrow$ select($pDist, i$) // by perm. sim.
5.    $k\mathrm{NN}_c \leftarrow$ Approx$k$NN$\mathbb{U}$($c, k$) // $C$ evals.
6.    If $d(q,c) \leq$ cov. rad. of $k\mathrm{NN}_c$ Then Report $c$// 1 ev.
```

```
Approx$k$NNG(Objs $\mathbb{U}$, Int $k$)
1. $E \leftarrow \emptyset$
2. For each $u \in \mathbb{U}$ Do // $nC$ dist ev. overall
3.    $E \leftarrow E \cup \{(u,v), v \in$ Approx$k$NN$\mathbb{U}$(u,k)$\}$ // $C$ evs.
4. Return($\mathbb{U}, E$)
```

```
RemoveNode($k$NNG $(\mathbb{U}, E)$, Index $\mathcal{I}$, Obj $u$)
1. $k\mathrm{NN}_u \leftarrow$ get the current adjacency of $u$ from $E$
2. $E \leftarrow E \setminus \{(u,v), v \in k\mathrm{NN}_u\}$ // remov. $u$ adja.
3. For each $v \in \mathbb{U}$ Do $E \leftarrow E \setminus \{(v,u)\}$//remov. rev. edg.
4. $\mathbb{U} \leftarrow \mathbb{U} \setminus \{u\}$,  $\mathcal{I}$.remove($u$)
```

```
Approx$k$NN$\mathbb{U}$(Obj $q$, Int $k$)
1. retrieve $\Pi_q$ from the index
2. $pDist \leftarrow \{(u, PS(\Pi_q, \Pi_u)), u \in \mathbb{U} \setminus \{q\}\}$
3. $\mathcal{C} \leftarrow$ selectSort($pDist, C$) // by perm. sim.
4. $distCq \leftarrow \{(c, d(c,q)), c \in \mathcal{C}\}$ // $C$ evals.
5. Return selectSort($distCq, k$) // by dist. to $q$
```

```
ApproxR$k$NNG(Obj $u$, Permut $\Pi_u$, Set $dCu$, Int $k$,
              Edges $E$)
1. For each $(c, d_{c,u}) \in dCu$ Do // up to $C^2$ evs. over all
2.    $k\mathrm{NN}_c \leftarrow$ get the current adjacency of $c$ from $E$
3.    If $d_{c,u} \leq$ covering radius of $k\mathrm{NN}_c$ Then Report $c$
4.    Else If $|k\mathrm{NN}_c| < k$ Then // check. $c$ adja. for restor.
5.       $E \leftarrow E \setminus \{(c,v), v \in k\mathrm{NN}_c\}$ // remov. old $c$ adja.
6.       $k\mathrm{NN}_c \leftarrow$ Approx$k$NN$\mathbb{U}$($c, k$) // $C$ evals.
7.       $E \leftarrow E \cup \{(c,v), v \in k\mathrm{NN}_c\}$ // restor. $c$ adja.
8.       If $d_{c,u} \leq$ cov. radius of $k\mathrm{NN}_c$ Then Report $c$
```

```
AddNode($k$NNG $(\mathbb{U}, E)$, Index $\mathcal{I}$, Obj $u$, Int $k$)
1. ($k\mathrm{NN}_u, \Pi_u, dCu$) $\leftarrow$ Approx$k$NN($u, k$) // $|\mathbb{A}|$+$C$ evs.
2. R$k\mathrm{NN}_u \leftarrow$ ApproxR$k$NNG($u, \Pi_u, dCu, k, E$)//0–$C^2$ evs.
3. $\mathbb{U} \leftarrow \mathbb{U} \cup \{u\}$,  $\mathcal{I}$.add($u, \Pi_u$)
4. For each $v \in$ R$k\mathrm{NN}_u$ Do // remove incident edges
5.    If $v$ has $k$ neighbors Then // if necessary
6.       let $w$ be $v$'s farthest neighbor,   $E \leftarrow E \setminus (v,w)$
7. $E \leftarrow E \cup \{(u,v), v \in k\mathrm{NN}_u\}$ // adding $u$ adjacency
8. $E \leftarrow E \cup \{(v,u), v \in$ R$k\mathrm{NN}_u\}$ // complet. other adjas.
```

Figure 1. Pseudocodes for kNN related problems. selectSort(set, k) retrieves the first k elements in set in increasing order, and select(set, k) retrieves the k-th element of set.

We consider a variant **Approx**kNN$\mathbb{U}$ for the special case when the query belongs to $\mathbb{U}$, which is used as an auxiliary method by other functions. In this variant we retrieve Π_q from the index, thus it only needs C distance computations.

2) Reverse k-nearest neighbor queries: We compute Π_q and select the first C objects in $\leq_{\Pi_q}$. Then, for each candidate c in C, we check if $d(q,c)$ is lower than the covering radius of c for kNNs in $\mathbb{U} \setminus \{c\}$. We do this in **Approx**RkNN, where we use **Approx**kNN$\mathbb{U}$. This algorithm needs $|\mathbb{A}| + C + C^2$ distance computations.

3) k-Nearest neighbor graph: For each $u \in \mathbb{U}$ we solve a kNN query retrieving neighbors from $\mathbb{U} \setminus \{u\}$ using **Approx**kNN$\mathbb{U}$. This cost nC distance computations (and extra $n|\mathbb{A}|$ distance computations if we need to construct the index). It is implemented in **Approx**kNNG.

4) Dynamic k-nearest neighbor graph: For this sake, we have to handle both object insertions into and deletions from the kNNG. When deleting an object, some nodes in the kNNG lose a neighbor, so we also need to restore them by performing kNN queries. However, as the most important property of the kNNG is that the objects in the adjacency list of a given object *are* the closest ones, we have chosen to perform a lazy kNNG restoration when adding new nodes.

Adding new objects into the kNNG: Given a new object $u \notin \mathbb{U}$, we compute the $NN_k(u)$ to determine its neighbors, but this time, besides retrieving the set of kNNs, we also get the permutation Π_u of u, and the set dCu of distances computed when reviewing the first C objects in the order induced by the permutation Π_u. Then, we compute the $RNN_k(u)$ to know which objects already in the kNNG must update their kNN adjacency lists. In this step, we perform a partial restoration of the graph by calling **Approx**RkNN**G**. Finally, we have to add u into $\mathbb{U}$, Π_u into the permutation index, and update the kNNG. This is implemented in **AddNode**, which costs $|\mathbb{A}| + C$ distance computations plus the ones needed to partially restore the graph, that is from zero to C^2 extra distance computations.

Function **Approx**RkNN**G** reuses Π_u and dCu. For each candidate c in dCu, it retrieves the current adjacency of c to check whether u is a reverse kNN of c (using the covering radius of c), in which case it reports c and continues with the next candidate. Otherwise, if c has *less* than k neighbors, it is still possible that u will be a reverse neighbor of c. Thus, we restore its adjacency using the variant **Approx**kNN$\mathbb{U}$ (this cost C distance computations) and perform the check again. This way, we delay the restoration (and its cost) as much as we can in the process of inserting and deleting objects. Actually, if the process only considers insertions of new objects, we never need to restore the kNNG.

Removing objects from the kNNG: We do this in **RemoveNode**, which simply extracts the node from both $\mathbb{U}$, the graph, and the permutation index; and also extracts its adjacency list and all the edges pointing to it from the graph edge set. This operation uses zero distance computations.

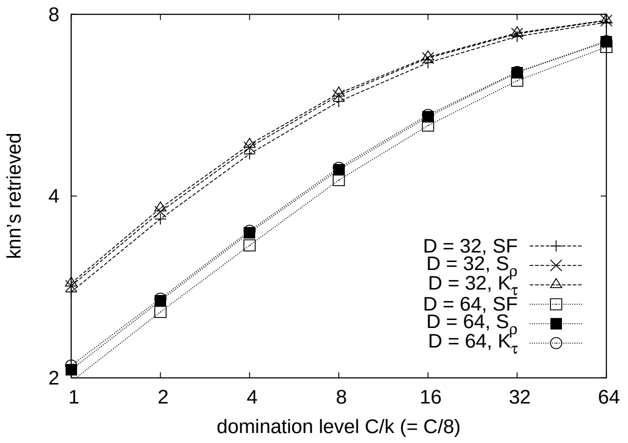
To remove an object o from the index we extract its permutation. Yet, if o is an anchor we need to do more work. The simplest option is to take away its identifier from every permutation. This does not alter the order in the permutations, but could degrade the search performance. So, upon several anchor deletions, it is necessary to restore the index, that is, choosing a new anchor set $\mathbb{A}$ and recompute all the permutations. This maintenance process can be made offline. Nevertheless, in this paper we neglect this situation.

IV. Experimental evaluation

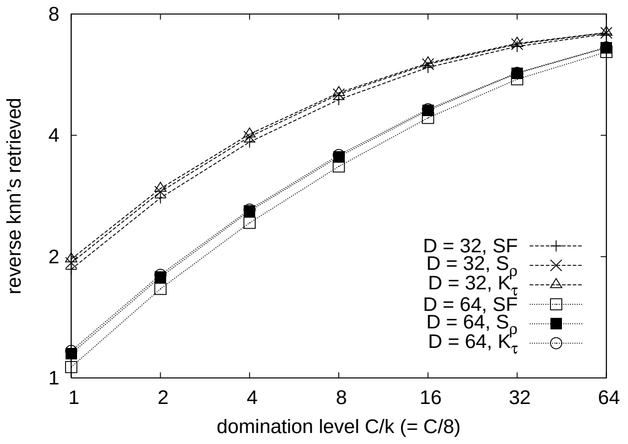
We have tested our kNN approach on a synthetic and a real-world metric space. The synthetic dataset is formed by 10,000 vectors uniformly distributed in the metric space $([0,1]^D, L_2)$ (the unitary real D-dimensional cube with Euclidean distance), for $D = 32$ and 64. For this value of dimension the problem is considered untreatable for exact techniques. Of course, we have not used the fact that vectors have coordinates, but have treated them as abstract objects.

The real-word dataset is composed by face images obtained from several sources: Kanade (`vasc.ri.cmu.edu/-NNFaceDetector/`); PIE_F_SE, PIE_NF, and PIE_T (`web.mit.edu/emeyers/www/face_databases.html`); BioID (`www.bioid.com/downloads/facedb/index.php`); and CAS-PEAL (`www.jdl.ac.cn/peal/index.html`). In order to standardize this set, face images were re-projected using PCA, generating 51,246 feature vectors with 2,152 components. We use L_2 in order to compare the feature vectors. We have indexed 20,000 randomly chosen face images and picked other 100 for the queries.

We run three experimental series. The first one is devoted to fix the parameters of our approach, namely, the function to measure similarity between permutations, the domination level, and the number of anchors. (This is because we do not have theoretical tools to estimate performance.) In this series, we also test the performance of the approximate direct and reverse kNN queries. The second series shows brief results respect to the construction of the kNNG. Finally, the third series deals with the face images. In the plots we show how many objects in the query outcome are correctly found, that is, we compare the query outcome with the real answer of the proximity query. In general, instead of speaking about distance computations we refer to the domination level. (To obtain the number of distances computed one needs to multiply k by the domination level.)



(a) $([0,1]^D, L_2)$, 8NN, 64 anchors, 5,000 objects.



(b) $([0,1]^D, L_2)$, R8NN, 64 anchors, 5,000 objects.

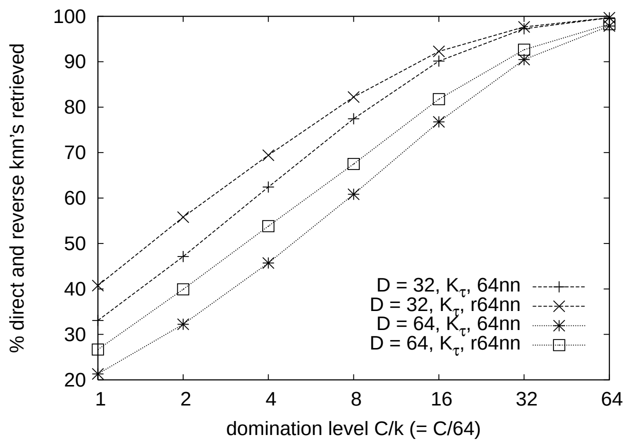
Figure 2. Studying the permutation similarity functions. Note the logscales.

The experiments were run on an Intel Core 2 Duo of 2.2 GHz, 4 MB of cache, 4 GB of RAM, and local disk, running Mac OS X 10.5.6. The algorithms were coded in C, and compiled with `gcc` version 4.0.1.

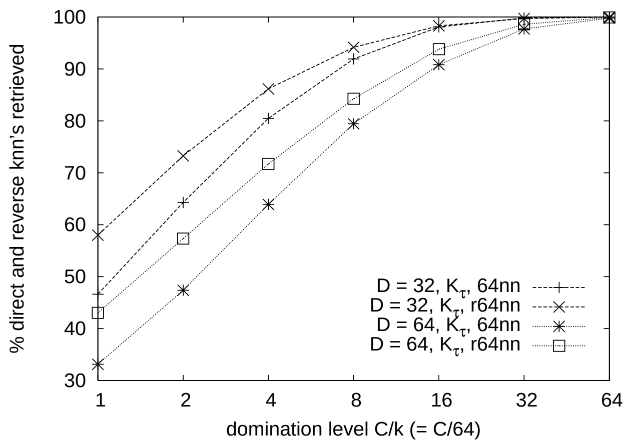
A. Parameter tuning, and direct and reverse kNN queries

We start by studying the prediction performance of the functions to measure permutation similarity. We run a test similar to the one in [12, Fig. 2]. We do direct and reverse 8NN queries using 64 anchors for 5,000 objects. We use a smaller dataset as we perform direct and reverse queries for the 5,000 objects (thus, in average a reverse 8NN query retrieves 8 objects). The results are show in Fig. 2. As expected from [12], SF, S_ρ, and K_τ have similar prediction power, the later being the most accurate one. Hence, in the following experiments we only use K_τ. Furthermore, when the space dimensionality increases, it is more difficult to solve similarity queries, as expected.

Now, we want to fix the domination level C/k. Plots in Fig. 2 also show that we obtain reasonable good results from



(a) $([0,1]^D, L_2)$, 64NN's and R64NN's, 64 anchors, 10,000 objects.



(b) $([0,1]^D, L_2)$, 64NN's and R64NN's, 128 anchors, 10,000 objects.

Figure 3. Studying the domination level and the size of anchor set. Note the logscales.

a domination level $C/k = 32$, specially in kNN queries. This is corroborated by the results in Fig. 3(a), where we test direct and reverse 64NN queries (over the full 10,000-vector dataset, averaging over 100 queries, so this time we show percentage of retrieval). In fact, for $k = 64$ we can use a lower domination level (for instance, 16). Nevertheless, the increasing of the dimensionality has a negative effect in the performance of our approach, but we can control this by increasing the number of anchors. In Fig. 3(b) we repeat the test doubling the number of anchors (we use 128), showing good retrieval results both in direct and reverse queries. We give some figures to illustrate the point: In dimension 64, with domination level $C/k = 16$ and 64 anchors (Fig. 3(a)) our technique retrieves 76.8% and 81.8% of the direct and reverse 64NNs, respectively; but if we use 128 anchors, our approach retrieves 90.8% and 93.8% of the direct and reverse 64NNs, respectively.

With regard to CPU time, our approach needs moderate time to solve these queries. For instance, in dimension 32, $k = 32$, $n = 10,000$, and 64 anchors, we need 0.52 and 21.5 seconds in order to solve direct and reverse kNN queries, respectively. A detailed CPU time study will be deferred to the extended version of this paper.

We run the same tests in dimensions D = 8 and 16. After tuning the size of the anchor set to 128 and the domination level to $C/k = 8$, we obtain almost complete retrieval.

B. k-Nearest neighbor graphs

We have already shown that our approach has good performance when solving direct and reverse kNN queries. This suggests good results in the kNNG construction process, since it is based in solving a single kNN query per object in the dataset (and each one needs $C = O(k)$ distance computations). Therefore, given the dataset $\mathbb{U}$ of size n, the whole process needs $nC = O(nk)$ distance evaluations in order to obtain the approximate kNNG. Hence, it is interesting to know how good is the approximation.

For this sake, we show in Table I the percentage of k-nearest neighbors properly computed in the graph (% kNN), and the ratio between the average covering radius of the approximate and the real kNNG (cr ratio). In this case we use 10,000 objects and 128 anchors. For $k = 8$, we use level $C/k = 32$, and for $k = 64$, level 16. (Remember that the number of distance evaluations can be easily derived by multiplying the domination level by k.)

We recover more than 98% of the true kNNs in dimension D = 32 using a reasonable number of distance computations. This dimensionality is considered as untreatable for traditional exact techniques. Similar results are obtained for $D = 64$ (more than 91% of true neighbors in the graph). Finally, it can be seen that the degradation of the approximate kNNG with respect to the real one is negligible. In fact, the average covering radius of the approximate kNNG has increased less than 4% for small values of k.

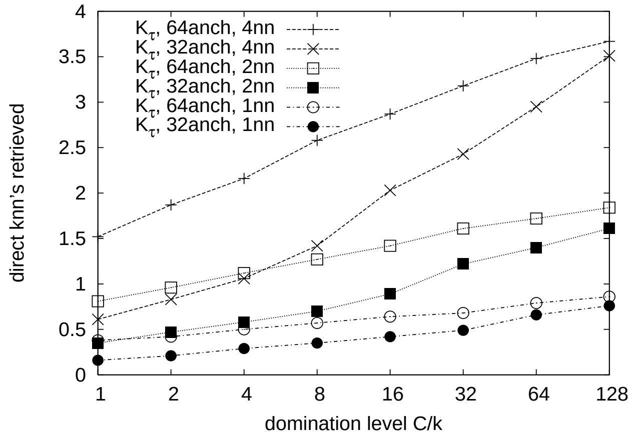
In the extended version of this paper we will also show how much varying these measures when inserting and deleting objects from the graph. However, we expect that these numbers remain constant.

C. Face images

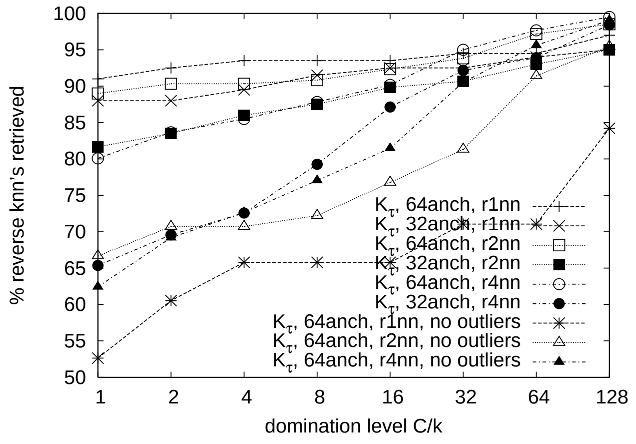
Finally, we perform a brief test in this dataset of 20,000 objects having representational dimensionality 2,152. We have considered anchor sets of sizes 32 and 64 (which are moderate when considering the high dimensionality). Fig. 4(a) shows good results for kNN queries. In fact, for 2NN and 4NN queries we retrieve around 86% of the

Table I
QUALITY PERFORMANCE IN THE CONSTRUCTION OF kNNGS

D	***k* = 8, *C/k* = 32**		***k* = 64, *C/k* = 16**	
	% kNN	cr ratio	% kNN	cr ratio
32	98.1	1.035	98.8	1.052
64	91.1	1.023	92.1	1.083



(a) Space of face images, kNN queries.



(b) Space of face images, reverse kNN queries.

Figure 4. Direct and reverse kNN queries in the face space. Note the logscales.

nearest neighbors using level $C/k = 64$ and 64 anchors. Finally, these experimental results confirm improvements in the retrieval rate when increasing the size of the anchor set and the domination level.

In Fig. 4(b), we plot the percentage of reverse kNNs properly retrieved. In this difficult case, we have excellent retrieval rate, which can be even improved by increasing the anchor set size or the domination level (and thus, the number of distance evaluations performed in the query). We suspect that the superior reverse kNN retrieval rate in this space is explained by the presence of outliers, that is, face images with no reverse neighbors. So, we repeat the experiment excluding outliers and we obtain a lower, but still good, rate. Certainly, this deserves more research.

V. Conclusions

We have presented a new approximate approach to solve several k-nearest neighbor (kNN) related problems in general metric spaces.

Our contribution is based on the following observation: A kNN query defines a search order in the metric dataset $\mathbb{U}$, of size n. This order is the sequence of object identifiers when they are sorted in increasing distance to the query. All we need to solve a kNN query are the first k elements in this sequence. Calculating this order is expensive in the original metric space as it implies computing n distances. Our idea is to use an alternative order, much cheaper to compute, yet it yields a rather similar sequence. This alternative order is obtained with the *permutation index* [12]. In this index, we choose a set of objects, the *anchors*, from the dataset $\mathbb{U}$. Then, each object in the dataset computes the distance to all of the anchors and stores in the index the permutation of anchor identifiers in increasing order of distance.

To solve the kNN query we compute the alternative order. Then, we select its first $C > k$ elements, and refine this subset so as to obtain an approximation to the true kNN answer. We have experimentally shown that one can conveniently choose $C/k = O(1)$, for reasonable values of the constant in the big-O notation. With a similar technique we can solve the related, but more difficult problem of computing *reverse* kNN queries using $O(k^2)$ distance computations.

With these low complexity bounds it is possible to foresee a large number of applications that may benefit with these algorithms. For instance, we also apply these primitives to construct *k-nearest neighbor graphs* (kNNGs), and also to update the kNNG upon object insertions and deletions.

In order to illustrate the effectiveness of our approach, we can say that in dimension 64, with $C/k = 16$ and 128 anchors, our technique retrieves 91% and 94% of the direct and reverse 64NNs, respectively. Of course, we can obtain even better results if we increase the value of C/k or the size of the anchor set.

With respect to the kNNG, using 128 anchors and $C/k = 32$, we recover more than 98% of the true k-nearest neighbors in dimension 32 using a reasonable amount of distance computations. Also, the expansion of the covering radii in the approximate graph is negligible, less than an 4% when compared with the covering radii of the real kNNG.

Future work involves the exploration of other alternative orders when solving kNN problems. Another interesting trend is to speed up *reverse* $k$nn queries by incorporating the regression model of [6] in order to efficiently estimate the covering radii of objects in the dataset. This way, instead of spend distance computations in order to compute the covering radii, we can use them in order to review more elements in the database, to hopefully improve our results when retrieving reverse k-nearest neighbors.

References

[1] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*. John Wiley & Sons, 1973.

[2] M. Brito, E. Chávez, A. Quiroz, and J. Yukich, "Connectivity of the mutual k-nearest neighbor graph in clustering and outlier detection," *Stat. Prob. Lett.*, vol. 35, pp. 33–42, 1997.

[3] D. Eppstein and J. Erickson, "Iterated nearest neighbors and finding minimal polytopes," *Discr. Comput. Geom.*, vol. 11, pp. 321–350, 1994.

[4] N. Archip, R. Rohling, P. Cooperberg, H. Tahmasebpour, and S. K. Warfield, "Spectral clustering algorithms for ultrasound image segmentation," in *Proc. 8th MICCAI, part II*, ser. LNCS, vol. 3750. Springer, 2005, pp. 862–869.

[5] R. Baeza-Yates, C. Hurtado, and M. Mendoza, "Query clustering for boosting web page ranking," in *Proc. 2nd AWIC*, ser. LNCS, vol. 3034. Springer, 2004, pp. 164–175.

[6] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz, "Efficient reverse *k*-nearest neighbor estimation," *Inform., Forsch. Entwickl.*, vol. 21, no. 3–4, pp. 179–195, 2007.

[7] Y. Tao, M. L. Yiu, and N. Mamoulis, "Reverse nearest neighbor search in metric spaces," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 9, pp. 1239–1252, 2006.

[8] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín, "Proximity searching in metric spaces," *ACM Comp. Surv.*, vol. 33, no. 3, pp. 273–321, 2001.

[9] G. Hjaltason and H. Samet, "Index-driven similarity search in metric spaces," *ACM Trans. Database Syst.*, vol. 28, no. 4, pp. 517–580, 2003.

[10] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. New York: Morgan Kaufmann, 2006.

[11] P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity Search - The Metric Space Approach*, ser. Advances in Database Systems. Springer, 2006, vol. 32.

[12] E. Chávez, K. Figueroa, and G. Navarro, "Effective proximity retrieval by ordering permutations," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 9, pp. 1647–1658, 2008.

[13] C. Böhm, S. Berchtold, and D. A. Keim, "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 322–373, 2001.

[14] A. Singh, H. Ferhatosmanoğlu, and A. Ş. Tosun, "High dimensional reverse nearest neighbor queries," in *Proc 12th CIKM*. ACM, 2003, pp. 91–98.

[15] R. Paredes, "Graphs for metric space searching," Ph.D. dissertation, University of Chile, Chile, 2008, Dept. of Computer Science Tech Report TR/DCC-2008-10. Available at `www.dcc.uchile.cl/TR/2008/TR_DCC-2008-010.pdf`.

[16] R. Paredes and E. Chávez, "Using the *k*-nearest neighbor graph for proximity searching in metric spaces," in *Proc. 12th SPIRE*, ser. LNCS, vol. 3772. Springer, 2005, pp. 127–138.

[17] P. Callahan and R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *Journal of the ACM*, vol. 42, no. 1, pp. 67–90, 1995.

[18] P. Callahan, "Optimal parallel all-nearest-neighbors using the well-separated pair decomposition," in *Proc. 34th IEEE FOCS*. IEEE CS Press, 1993, pp. 332–340.

[19] K. Clarkson, "Fast algorithms for the all-nearest-neighbors problem," in *Proc. 24th IEEE FOCS*. IEEE CS Press, 1983, pp. 226–232.

[20] M. Dickerson and D. Eppstein, "Algorithms for proximity problems in higher dimensions," *Comput. Geom. Theory Appl.*, vol. 5, no. 5, pp. 277–291, 1996.

[21] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[22] P. Vaidya, "An $O(n \log n)$ algorithm for the all-nearest-neighbor problem," *Discr. Comput. Geom.*, vol. 4, pp. 101–115, 1989.

[23] K. Clarkson, "Nearest neighbor queries in metric spaces," *Discr. Comput. Geom.*, vol. 22, no. 1, pp. 63–93, 1999.

[24] K. Figueroa, "An efficient algorithm to all k nearest neighbor problem in metric spaces," Master's thesis, Universidad Michoacana, Mexico, 2000, in Spanish.

[25] D. R. Karger and M. Ruhl, "Finding nearest neighbors in growth-restricted metrics," in *Proc. 34th ACM STOC*. ACM, 2002, pp. 741–750.

[26] R. Krauthgamer and J. R. Lee, "Navigating nets: simple algorithms for proximity search," in *Proc. 15th ACM-SIAM SODA*. SIAM, 2004, pp. 798–807.

[27] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro, "Practical construction of k-nearest neighbor graphs in metric spaces," in *Proc. 5th WEA*, ser. LNCS, vol. 4007. Springer, 2006, pp. 85–97.

[28] E. Chávez and K. Figueroa, "Faster proximity searching in metric data." in *Proc. 3rd MICAI*, ser. LNAI, vol. 2972. Springer, 2004, pp. 222–231.

[29] I. Kalantari and G. McDonald., "A data structure and an algorithm for the nearest point problem," *IEEE Trans. Softw. Eng.*, vol. 9, no. 5, pp. 631–634, 1983.

[30] F. Korn and S. Muthukrishnan, "Influence sets based on reverse nearest neighbor queries," in *Proc. 20th ACM SIGMOD*. ACM, 2000, pp. 201–212.

[31] A. Maheshwari, J. Vahrenhold, and N. Zeh, "On reverse nearest neighbor queries," in *Proc. 14th CCCG*, 2002, pp. 128–132.

[32] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao, "Reverse nearest neighbors in large graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 4, pp. 540–553, 2006.

[33] E. Achtert, H. P. Kriegel, P. Kröger, M. Renz, and A. Züfle, "Reverse k-nearest neighbor search in dynamic and general metric databases," in *Proc. 12th EDBT*. ACM, 2009, pp. 886–897.

[34] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," *SIAM J. Discr. Math.*, vol. 17, no. 1, pp. 134–160, 2003.