Practical Construction of k-Nearest Neighbor Graphs in Metric Spaces *

Rodrigo Paredes[1], Edgar Chávez[2], Karina Figueroa[1,2], and Gonzalo Navarro[1]

[1] Center for Web Research, Dept. of Computer Science, University of Chile.
[2] Escuela de Ciencias Físico-Matemáticas, Univ. Michoacana, Mexico.
{raparede,kfiguero,gnavarro}@dcc.uchile.cl, elchavez@umich.mx

Abstract. Let $\mathbb{U}$ be a set of elements and d a distance function defined among them. Let $NN_k(u)$ be the k elements in $\mathbb{U}-\{u\}$ having the smallest distance to u. The k-nearest neighbor graph (kNNG) is a weighted directed graph $G(\mathbb{U}, E)$ such that $E = \{(u, v), v \in NN_k(u)\}$. Several kNNG construction algorithms are known, but they are not suitable to general metric spaces. We present a general methodology to construct kNNGs that exploits several features of metric spaces. Experiments suggest that it yields costs of the form $c_1 n^{1.27}$ distance computations for low and medium dimensional spaces, and $c_2 n^{1.90}$ for high dimensional ones.

1 Introduction

Let $\mathbb{U}$ be a set of elements and d a distance function defined among them. Let $NN_k(u)$ be the k elements in $\mathbb{U} - \{u\}$ having the smallest distance to u according to the function d. The *k-nearest neighbor graph* (kNNG) is a weighted directed graph $G(\mathbb{U}, E)$ connecting each element to its k-nearest neighbors, thus $E = \{(u, v), v \in NN_k(u)\}$. Building the kNNG is a direct generalization of the *all-nearest-neighbor* (ANN) problem, so ANN corresponds to the 1NNG construction problem. kNNGs are central in many applications: cluster and outlier detection [14, 4], VLSI design, spin glass and other physical process simulations [6], pattern recognition [12], query or document recommendation systems [3], and others.

There are many kNNG construction algorithms which assume that nodes are points in $\mathbb{R}^D$ and d is the Euclidean or some L_p Minkowski distance. However, this is not the case in several kNNG applications. An example is collaborative filters for Web searching, such as query or document recommendation systems, where kNNGs are used to find clusters of similar queries, to later improve the quality of the results shown to the final user by exploiting cluster properties [3].

To handle this problem one must resort to a more general model called *metric spaces*. A metric space is a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is the universe of objects and d is a distance function among them that satisfies the triangle inequality.

Another appealing problem in metric spaces is similarity searching [8]. Given a finite metric database $\mathbb{U} \subseteq \mathbb{X}$, the goal is to build an index for $\mathbb{U}$ such that later, given a query object $q \in \mathbb{X}$, one can find elements of $\mathbb{U}$ close to q using as few distance computations as possible. See [8] for a comprehensive survey.

* Supported in part by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile; and CONACyT, Mexico.

We have already demonstrated kNNG searching capabilities in general metric spaces [20], where we give kNNG-based search algorithms with practical applicability in low-memory scenarios, or metric spaces of medium or high dimensionality. Hence, in this paper we focus on a metric kNNG construction methodology, and propose two algorithms based on such methodology. According to our experimental results, they have costs of the form $c_1 n^{1.27}$ distance computations for low and medium dimensionality spaces, and $c_2 n^{1.90}$ for high dimensionality ones. Note that a naive construction requires $O(n^2)$ distance evaluations.

1.1 A summary of metric space searching

Given the universe of objects $\mathbb{X}$, a metric space is a pair $(\mathbb{X}, d)$, where $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ is any distance function in $\mathbb{X}$ that is symmetric and satisfies the triangle inequality. Some examples are $(\mathbb{R}^D, L_p)$, the space of strings under the edit distance, or the space of documents under the cosine distances.

The metric database is a finite set $\mathbb{U} \subseteq \mathbb{X}, n = |\mathbb{U}|$. A similarity query is an object $q \in \mathbb{X}$, and allows two basic types: the *Range query* (q, r) retrieves all objects $u \in \mathbb{U}$ such that $d(u, q) \leq r$; and the *k-Nearest neighbor query* $NN_k(q)$ retrieves the k objects in $\mathbb{U}$ closest to q according to the distance d. A $NN_k(q)$ algorithm is called *range-optimal* [16] if it uses the same number of distance evaluations as the equivalent range query whose radius retrieves exactly k objects. We call this radius *covering radius*.

An *index* $\mathcal{I}$ is a data structure built over $\mathbb{U}$ using some cells from the whole $\mathbb{U} \times \mathbb{U}$ distance matrix. $\mathcal{I}$ permits solving the above queries without comparing q with each element in $\mathbb{U}$. There are two kinds of indices: pivot based and compact partition based. Search algorithms use $\mathcal{I}$ and some distance evaluations to discard – using the triangle inequality – as many objects as they can, to produce a small candidate set $\mathcal{C}$ that could be relevant to q. Later, they exhaustively check $\mathcal{C}$ by computing distances from q to each candidate to obtain the query result.

As the distance is considered expensive to compute, it is customary to use the number of distance evaluations as the complexity measure both for index construction and object retrieving. For instance, each computation of the cosine distance takes 1.4 msecs in our machine (Pentium IV of 2 GHz). This is really costly even compared with the operations introduced by the graph, such as the shortest path computation using Dijkstra's algorithm.

Many authors agree that the proximity query cost worsens quickly as the *intrinsic dimensionality* of the space grows. This is known as the *curse of dimensionality*. Although there is not and accepted criterion to define the intrinsic dimensionality in a metric space, a general agreement is that spaces with low variance and large mean in their distance histograms have a large intrinsic dimension.

1.2 Related work on kNNG construction

The naive approach to construct kNNGs uses $\frac{n(n-1)}{2} = O(n^2)$ distance computations and $O(kn)$ memory. For each $u \in \mathbb{U}$ we compute the distance to all the others, selecting the k lowest-distance objects. However, there are alternatives to

speed up the procedure. The proximity properties of the Voronoi diagram [2] or its dual, the Delaunay triangulation, allow solving the problem more efficiently. The ANN problem can be optimally solved in $O(n \log n)$ time in the plane [13] and in $\mathbb{R}^D$ for any fixed D [9, 22], but the constant depends exponentially on D. In $\mathbb{R}^D$, kNNGs can be built in $O(nk \log n)$ time [22] and even in $O(kn+n \log n)$ time [5, 6, 11]. Approximation algorithms have also been proposed [1]. However, these alternatives, except the naive one, are unsuitable for metric spaces, as they use coordinate information that is not necessarily available in general metric spaces.

Clarkson states the first generalization of ANN to metric spaces [10], where the problem is solved using randomization in $O(n \log^2 n \log^2 \Gamma(\mathbb{U}))$ expected time, where $\Gamma(\mathbb{U})$ 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 $O(n \log^4 n)$ 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 $\Omega(n^2)$ space for high dimensions, which is too much for practical applications.

In [15], another technique for general metric spaces is given. It solves n range queries of decreasing radius by using a pivot-based index. As it is well known, the performance of pivot-based algorithms worsens quickly as the dimension of the space grows, limiting the applicability of this technique. Our pivot based algorithm (Section 2.4) can be seen as an improvement over this technique.

Recently, Karger and Ruhl present the *metric skip list* [18], an index that uses $O(n \log n)$ space and can be constructed with $O(n \log n)$ distance computations. The index answers $NN_1(q)$ queries using $O(\log n)$ distance evaluations with high probability. Later, Krauthgamer and Lee introduce *navigating nets* [19], another index that can be constructed also with $O(n \log n)$ distance computations, yet using $O(n)$ space, and which gives an $(1 + \epsilon)$-approximation algorithm to solve $NN_1(q)$ queries in time $O(\log n)+(1/\epsilon)^{O(1)}$. Both of them could serve to solve the ANN 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.

2 Our methodology

We are interested in practical kNNG construction algorithms for general metric spaces. This problem is equivalent to solve n $NN_k(u)$ queries for all $u \in \mathbb{U}$. Thus, a straightforward solution has two stages: the first is to build some known metric index $\mathcal{I}$ [8], and the second is to use $\mathcal{I}$ to solve the n queries. However, this basic scheme can be improved if we take into account these observations:

- We are solving queries for all the elements in $\mathbb{U}$, not for general objects in $\mathbb{X}$. If we solve the n queries jointly we can share costs through the whole process. For instance, we can avoid some calculations by using the symmetry of d.
- We can upper bound some distances by computing shortest paths over the kNNG under construction, maybe avoiding their actual computation. So, we can use the very kNNG in stepwise refinements to improve the second stage.

2.1 The ingredients of the recipe

The main data structure. Along all the algorithm, we use the *Neighbor Heap Array* (*NHA*) to store the kNNG under construction. *NHA* can be regarded as the union of *priority queues* NHA_u, of size k, for all $u \in \mathbb{U}$. At any point in the process NHA_u will contain the k elements closest to u known up to then, and their distances to u. Formally, $NHA_u = \{(x_{i_1}, d(u, x_{i_1})), \ldots, (x_{i_k}, d(u, x_{i_k}))\}$ sorted by decreasing $d(u, x_{i_j})$ (i_j is the j-th neighbor identifier).

For each $u \in \mathbb{U}$, we initialize $NHA_u = \{(\perp, \infty), \ldots, (\perp, \infty)\}$, $|NHA_u| = k$. Let $curCR_u = d(u, x_{i_1})$ be the current covering radius of u, that is, the distance from u towards its current farthest neighbor candidate in NHA_u.

In the first stage, every distance computed to build the index $\mathcal{I}$ populates *NHA*. In the second, we refine *NHA* with the following distance computations. We must ensure that $|NHA_u| = k$ upon successive additions. Hence, if we find some object v such that $d(u, v) < curCR_u$, before adding $(v, d(u, v))$ to NHA_u we extract the farthest candidate from NHA_u. This progressively reduces $curCR_u$ from ∞ to the real covering radius. At the end, *NHA* stores the kNNG of $\mathbb{U}$.

Using *NHA* as a graph. Once we calculate $d_{uv} = d(u, v)$, if $d_{uv} \geq curCR_u$ we discard v as a candidate for NHA_u. Also, due to the triangle inequality we can discard all objects w such that $d(v, w) \leq d_{uv} - curCR_u$. Unfortunately, we do not necessarily have stored $d(v, w)$. However, we can upper bound $d(v, w)$ with the sum of edge weights traversed in the shortest paths over *NHA* from v to all $w \in \mathbb{U}$, $d_{NHA}(v, w)$. So, if $d_{uv} \geq curCR_u$, we also discard all objects w such that $d_{NHA}(v, w) \leq d_{uv} - curCR_u$.

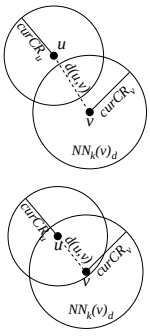
***d* is symmetric.** Every time a distance $d_{uv} = d(u, v)$ is computed, we check both $d_{uv} < curCR_u$ for adding (v, d_{uv}) to NHA_u, and $d_{uv} < curCR_v$ for adding (u, d_{uv}) to NHA_v. This can both reduce $curCR_v$, and cheapen the future query for v, even when we are solving neighbors for another object.

$\mathbb{U}$ is fixed. Assume we are solving query $NN_k(u)$, we have to check some already solved object v, and $curCR_u \leq curCR_v$. Then, if $u \notin NN_k(v) \Rightarrow d(u, v) \geq curCR_v$, so $v \notin NN_k(u)$. Otherwise, if $u \in NN_k(v)$, then we already computed $d(u, v)$. Then, in those cases we avoid to compute $d(u, v)$. Fig. 1(a) illustrates.

Check Order Heap (*COH*). We create the priority queue $COH = \{(u, curCR_u), u \in \mathbb{U}\}$ to complete $NN_k(u)$ queries in increasing $curCR_u$ order, because a small radius query has larger discriminative power and produces candidates that are closer to the query u. This reduces the CPU time and – as d is symmetric – could increase the chance of improving candidate sets in *NHA* for other objects v. This, in turn, could reduce $curCR_v$ and change the position of v in COH.

The recipe. We split the process into two stages. The first is to build $\mathcal{I}$ to preindex the objects. The second is to use $\mathcal{I}$ and all the ingredients to solve the $NN_k(u)$ queries for all $u \in \mathbb{U}$. Fig. 1(b) depicts the methodology.

For practical reasons, we allow that our algorithms use at most $O(n(\log n + k))$ memory both to index $\mathbb{U}$ and to store the kNNG under construction.



(a) $\mathbb{U}$ is fixed.

K**NN** (Integer k, ObjectSet $\mathbb{U}$)
Stage 1: Initialize *NHA* and construct the index $\mathcal{I}$
1. **For each** $u \in \mathbb{U}$ **Do** $NHA_u \leftarrow \{(\perp, \infty), \ldots, (\perp, \infty)\}$ // k pairs
2. Create $\mathcal{I}$, all computed distances populate symmetrically *NHA*

Stage 2: Complete the $NN_k(u)$ for all $u \in \mathbb{U}$
3. $COH \leftarrow \{(u, curCR_u), u \in \mathbb{U}\}$
4. **For each** $(u, curCR_u) \in COH$, in increasing $curCR_u$ order **Do**
5. Create the candidate set $\mathcal{C}$ according to $\mathcal{I}$ // exclude NHA_u
6. **While** $\mathcal{C} \neq \emptyset$ **Do**
7. $c \leftarrow$ extract a candidate from $\mathcal{C}$
8. **If** "$\mathbb{U}$ is fixed" does not apply for u and c **Then**
9. $d_{uc} \leftarrow d(u, c)$, try to insert c into NHA_u
10. try to insert u into NHA_c, update c in COH (symmetry)
11. use *NHA* as a graph and $\mathcal{I}$ to discard objects from $\mathcal{C}$
12. **Return** *NHA* as a graph

(b) Sketch of the methodology.

Fig. 1. In 1(a), assume we are solving u, v is already solved, and $curCR_u \leq curCR_v$. On the top, if $u \notin NN_k(v) \Rightarrow d(u, v) \geq curCR_v \geq curCR_u$. On the bottom, if $u \in NN_k(v)$, we already computed $d(u, v)$. Then, in those cases we avoid computing $d(u, v)$. In Fig. 1(b), we sketch the methodology.

2.2 The resulting algorithms

Based on our methodology, we propose two kNNG construction algorithms focused on decreasing the total number of distance computations. They are:

1. Recursive partition based algorithm: In the first stage, we build a preindex by performing a recursive partitioning of the space. In the second stage, we complete the $NN_k(u)$ queries using the order induced by the partitioning.
2. Pivot based algorithm: In the preindexing stage, we build the pivot index. Later, we complete the $NN_k(u)$ queries by performing range-optimal queries.

The experiments confirm that these algorithms are efficient. For instance, in the string space, the pivot-based algorithm requires CPU time of the empirical form $c_t n^{1.85}$, and $c_d n^{1.26}$ in distance computations. In the high-dimensional document space, the recursive partition-based algorithm requires empirically $cn^{1.955}$ both in distance computations and CPU time.

2.3 Recursive partition based algorithm

This algorithm is based on using a preindex slightly different to the *Bisector Tree* (*BST*) [17]. We call our modified *BST* the *Division Control Tree* (*DCT*), which is a binary tree representing the shape of the partitioning. The *DCT* node structure is $\{p, l, r, pr\}$, which represents the parent, left and right children, and partition radius of the node, respectively. The partition radius is the distance from the node towards the farthest node of its partition. (With respect to the *BST* structure, we have added the pointer p to easily navigate trough the tree.)

For simplicity we use the same name for the node and for its representative in the DCT. Then, given a node $u \in \mathbb{U}$, u_p, u_l, and u_r, refer to nodes that are the parent, left child, and right child of u in the DCT, respectively, and also to their representative nodes in $\mathbb{U}$. Finally, u_{pr} refers to the partition radius of u.

In this algorithm, we use $O(kn)$ space to store the NHA and $O(n)$ to store the DCT. The remaining memory is used as a cache of computed distances, CD, whose size is limited to $O(n \log n)$. Thus, every time we need to compute a distance, we check if it is present in CD, in which case we return the stored value. Note that the $CD \subset \mathbb{U}^2 \times \mathbb{R}^+$ can also be seen as graph of all stored distances. The criterion to insert distances into CD depends on the stage (see later). Once we complete the $NN_k(u)$, we remove its adjacency list from CD.

First stage: construction of DCT. We partition the space recursively to construct the DCT, and populate symmetrically NHA and CD with all the computed distances. The DCT is built as follows. Given the node $root$ and the set S, we choose children objects l and r from S. Then, we generate two subsets: S_l, objects nearest to l, and S_r, objects nearest to r. Finally, we compute both partition radii. The recursion follows with (l, S_l) and (r, S_r), finishing when $|S| < 2$. Once we finish the division, leaves in the DCT have partition radii 0. The DCT root is fictitious, having no equivalent in $\mathbb{U}$, and partition radius ∞.

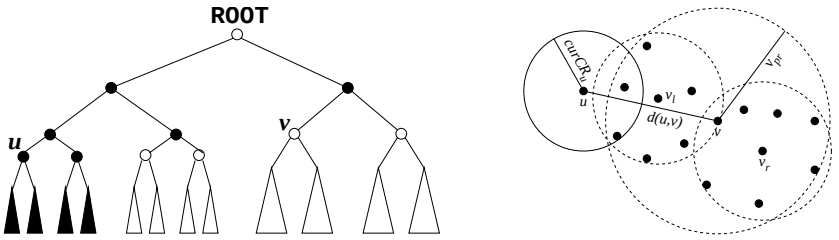
Since the DCT has n nodes, its expected height is $2 \ln n$ (the DCT construction is statistically identical to populating a binary search tree). For each DCT level, each node computes two distances towards the splitting nodes, which accounts for $2n$ distances per level. So, we expect to compute $4n \ln n$ distances in the partitioning. As we store 2 edges per distance, we need to store $8n \ln n$ in CD. Hence, we fix the maximum space of CD as $8n \ln n = O(n \log n)$.

Solving $NN_k(u)$ queries with DCT. The construction of DCT ensures that every node has already computed distances to all of its ancestors, its ancestor's siblings, and its parent descent. Then, to finish the $NN_k(u)$ query, it is enough to check whether there are relevant objects in all the descendants of u's ancestors' siblings. This corresponds to white nodes and subtrees in Fig. 2(a).

Nevertheless, the DCT allows us to avoid some work. Assume we are checking whether v is relevant to u, and the balls $(u, curCR_u)$ and (v, v_{pr}) do not intersect each other, then we discard v and its partition. Otherwise, we recursively check children v_l and v_r. Fig. 2(b) illustrates this.

Hence, in the candidate set $\mathcal{C}$, it suffices to manage the set of ancestors' siblings, and if it is not possible to discard the whole sibling's partition we add its children into $\mathcal{C}$. Since it is more likely to discard small partitions, we process $\mathcal{C}$ in order of increasing radius. This agrees with the intuition that the partition radius of u's parent's sibling is likely the smallest of $\mathcal{C}$, and that some of its descendants could be relevant to u.

Second stage: Completing the queries. As CD can be seen as a graph, we use $NHA \cup CD$ to upper bound distances: when $d(u, v) \geq curCR_u$, we discard objects w such that their shortest path $d_{NHA \cup CD}(v, w) \leq d(u, v) - curCR_u$. We do this by adding them to $\mathcal{C}$ marked as `EXTRACTED`.



(a) Induced check sequence. (b) DCT can discard partitions.

Fig. 2. Using the DCT to solve $NN_k(q)$ queries. In 2(a), u has been compared with all black nodes and all the descent of its parent. To finish the query, we just process white nodes and subtrees. In 2(b), as $d(u,v) \leq curCR_u + v_{pr}$ the partition of v intersects the ball $(u, curCR_u)$, so we recursively check children v_l and v_r. As v_r's partition does not intersect the ball $(u, curCR_u)$, we discard v_r and its partition. However, we continue the checking on v_l's partition as it intersects the ball $(u, curCR_u)$.

In this stage, if we have available space in CD, we cache all the computed distances small enough so as to get into their respective queues in NHA, since these distances can be used in future symmetric queries. Note that adding distances to CD without considering the space limitation could increase its size beyond control, as it is shown by the following average case analysis. With probability $\frac{n-k}{n}$, a random distance is greater than the k-th shortest one (thus, not stored), and with probability $\frac{k}{n}$ it is lower, then it is stored in CD using one cell. The base case uses k cells for the first distances. Then, the recurrence for the average case of edge insertions for each NHA_u is: $T(n,k) = T(n-1,k) + \frac{k}{n}$, $T(k,k) = k$. We obtain $T(n,k) = k(H_n - H_k + 1) = O(k \log \frac{n}{k})$. As we have n priority queues, if we do not consider the limitation, we could use $O(nk \log \frac{n}{k})$ memory cells, which can be an unpractical memory requirement.

Finally, we combine all of these ideas to complete the $NN_k(u)$ queries for all nodes in $\mathbb{U}$. We begin by creating the priority queue COH. Then, for each node u picked from COH we do the following. We add the edges of NHA_u to CD_u, where CD_u refers to the adjacency list of u in CD. (Due to the size limitation it is likely that some of the u's current neighbors do not belong CD_u.) Then, we compute shortest paths from all u's ancestors discarding as many objects as we can. Then, we finish the query $NN_k(u)$, and finally delete CD_u.

To finish the query $NN_k(u)$, we start adding all u's ancestors to $\mathcal{C}$. Later, we take objects c from $\mathcal{C}$ in increasing c_{pr} order, and process c according one of the following rules:

1. If c was already marked as `EXTRACTED`, we add its children $\{c_l, c_r\}$ to $\mathcal{C}$;
2. If "$\mathbb{U}$ is fixed" applies for c and u, and $d(u,c) \notin CD$, we add $\{c_l, c_r\}$ to $\mathcal{C}$; or
3. If we have $d(u,c)$ stored in CD, we retrieve it, else we compute it and use "d is symmetric". Then, if $d(u,c) < curCR_u + c_{pr}$, we have region intersection, so we add $\{c_l, c_r\}$ to $\mathcal{C}$. Next, we use $NHA \cup CD$ as a graph computing shortest paths from c to discard as many object as we can.

2.4 Pivot-based algorithm

Pivot-based algorithms have good performance in low dimensional spaces, but worsen quickly as the dimension grows. However, our methodology compensates this failure in medium and high dimensions. In this algorithm we use $O(kn)$ space in *NHA* and $O(n \log n)$ space to store the pivot index.

First stage: construction of the pivot index. We select at random a set of pivots $\mathcal{P} = \{p_1, \ldots, p_{|\mathcal{P}|}\} \subseteq \mathbb{U}$, and store a table of $|\mathcal{P}|n$ distances $d(p_j, u),\ j \in \{1, \ldots, |\mathcal{P}|\},\ u \in \mathbb{U}$. We give the same space in bytes to the table as that of the cache of distances and the division control tree of the recursive based algorithm. Then, in our implementation we use $|\mathcal{P}| = 12 \ln n + 2.5 = O(\log n)$.

Solving $NN_k(u)$ queries with the pivot table. To perform a range-optimal query for u we use $\mathcal{C}$ as an array to store maximum lower bounds of distances from u to other objects. Because of the triangle inequality, for each $v \in \mathbb{U}$ and $p \in \mathcal{P}$, $|d(v,p) - d(u,p)|$ is a lower bound of $d(u,v)$. Let $\mathcal{C}_v = \max_{p \in \mathcal{P}}\{|d(v,p) - d(u,p)|\}$. So, we can discard non-relevant objects v such that $\mathcal{C}_v \geq curCR_u$.

Then, we store $\mathcal{C}$ values in a priority queue $Sort\mathcal{C} = \{(v, \mathcal{C}_v), v \in \mathbb{U} - (\mathcal{P} \cup NHA_u \cup \{u\})\}$. For each object v picked from $Sort\mathcal{C}$ by ascending $\mathcal{C}_v$, we check if $\mathcal{C}_v < curCR_u$. In such case, when "$\mathbb{U}$ is fixed" applies for u and v we avoid the distance computation and process the next node, else we compute the distance $d_{uv} = d(u,v)$. So, if $d_{uv} < curCR_u$ we add v to NHA_u (this could reduce $curCR_u$). Also, using " d is symmetric" we can refine NHA_v and consequently update v in COH. Finally, we use *NHA* as a graph computing shortest path from v to extract from $Sort\mathcal{C}$ as many object as we can. Each $NN_k(u)$ query finishes when we reach a node v such that $\mathcal{C}_v \geq curCR_u$, or $Sort\mathcal{C}$ gets empty.

Second stage: Completing the queries. Since pivots $p \in \mathcal{P}$ compute distances towards all objects, once we compute the table, they have already solved their k-nearest neighbors. So, we only have to complete $n - |\mathcal{P}|$ range-optimal queries for objects $u \in \mathbb{U} - \mathcal{P}$. Notice that because of the symmetry of d, these objects already have candidates in their respective queues in *NHA*.

3 Experimental results

We have tested our algorithms on synthetic and real-world metric spaces. The first synthetic set is formed by 65,536 points uniformly distributed in the metric space $([0,1]^D, L_2)$ (the unitary real D-dimensional cube with Euclidean distance). This space allows us to measure the effect of the space dimension D on our algorithms. The second set is formed by 65,536 points in a 20-dimensional space with Gaussian distribution forming 256 clusters randomly placed in $([0,1]^{20}, L_2)$. We consider three standard deviations to make more crisp or more fuzzy clusters ($\sigma = 0.1,\ 0.2,\ 0.3$). Of course, we have not used the fact that vectors have coordinates, but have treated them as abstract objects.

The first real-world set is the string metric space under the edit distance, a discrete function that measures the minimum number of character insertions,

deletions and replacements needed to make the strings equal. We index a random subset of 65,536 words taken from an English dictionary. The second set is the document space under the cosine distance, a function that measures the angle between two documents when they are represented as vectors in a high-dimensional vector model. We index a random subset of 1,215 English documents taken from the TREC-3 collection.

Experiments were run on an Intel Pentium IV of 2 GHz and 512 MB of RAM. We measure distance evaluations and CPU time. For shortness we have called the basic kNNG construction algorithm KNNb, the recursive partition based algorithm KNNrp, and the pivot based algorithm KNNpiv. We are not aware of any published kNNG practical implementation for general metric spaces.

We summarize our experimental results in Fig. 3, where we show distance computations per element, and Table 1 for the least square fittings computed with R [21]. The dependence on k turns out to be so mild that we neglect k in the fittings, thus, costs have the form cn^{α}. Even though in Table 1 we explicit the constant c, from now on, we only refer to the exponent α.

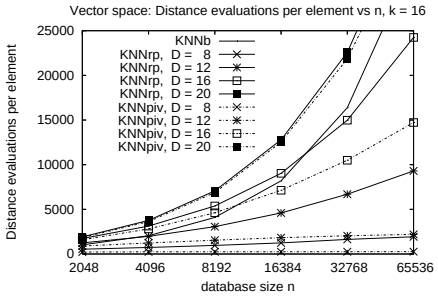
Figs. 3(a), 3(b) and 3(c) show experimental results for $\mathbb{R}^D$. Fig. 3(c) shows that, as D grows, the performance of our algorithms degrade, phenomenon known as the *curse of dimensionality*. For instance, for $D = 4$, KNNpiv uses $cn^{1.10}$ distance evaluations, but for $D = 24$, it is $cn^{1.96}$ distance evaluations. Notice that a metric space with dimensionality $D > 20$ is considered as intractable [8]. Fig. 3(a) shows that for all dimensions our algorithms are subquadratic in distance evaluations, instead of KNNb which is always cn^2. For low and medium dimensions ($D \leq 16$) ours have better performance than KNNb, being KNNpiv the best of ours. Moreover, for lower dimensions ($D \leq 8$) ours are only slightly superlinear. Fig. 3(b) shows a sublinear dependence on k for all dimensions, however, KNNpiv is more sensitive to k than KNNrp. Also, the dependence on k diminishes as long as D grows, although it is always monotonically increasing on k. Finally, it is shown that for $k \leq 4$, our algorithms behave better than KNNb, even in high dimensional spaces (KNNpiv in $D = 20$).

Figs. 3(e) and 3(f) show results in Gaussian space. For crisp clusters ($\sigma = 0.1$) the performance of our algorithms improves significantly, even for high values of k. It is interesting to note that for $k \leq 8$ our algorithms are more efficient than KNNb for the three variances. Again, KNNpiv has the best performance.

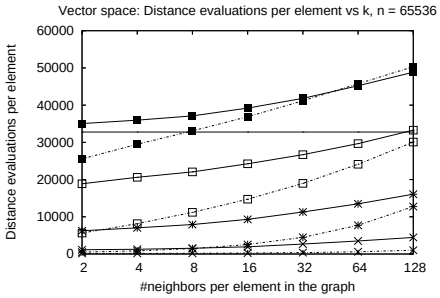
Figs. 3(g) and 3(h) show results for strings. The plots show that both KNNrp and KNNpiv are subquadratic for all $k \in [2, 128]$. For instance, for $n = 65,536$, KNNrp costs 28%, and KNNpiv just 8%, of KNNb to build the 32NNG.

Finally, Fig. 3(d) shows that our methodology save lots of work in the high-dimensional document space. For instance, for $n = 1,215$, KNNrp costs 63%, and KNNpiv costs 67%, of KNNb to build the 8NNG. These two last results show that our methodology is also practical in real-world situations.

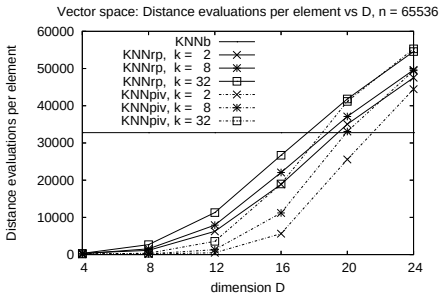
All of these conclusions are confirmed in Table 1. We remark that in some practical conditions (vectors in $[0, 1]^D$ with $D \leq 8$ and $k \leq 32$ and Gaussian vectors with $\sigma = 0.01$ and $k \leq 8$), KNNpiv also has better performance than KNNb in CPU time. This is important since the Euclidean distance is very cheap



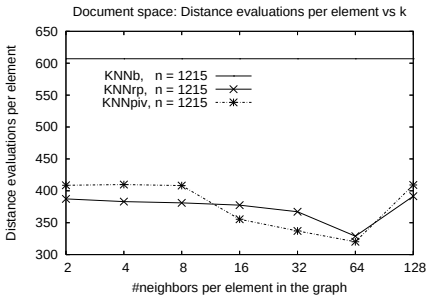
(a) In $\mathbb{R}^D$, dependence on n.



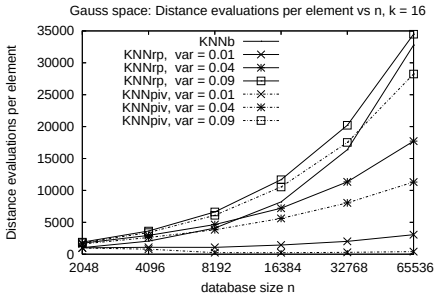
(b) In $\mathbb{R}^D$, dependence on k.



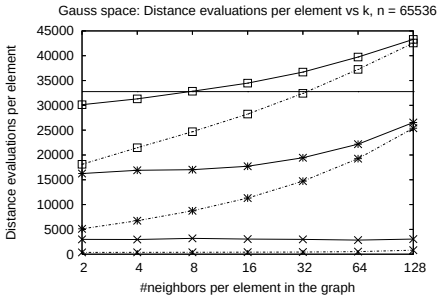
(c) In $\mathbb{R}^D$, dependence on D.



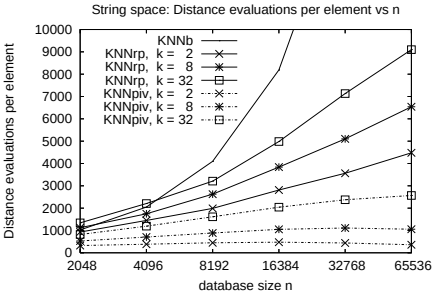
(d) In Document space, dependence on k.



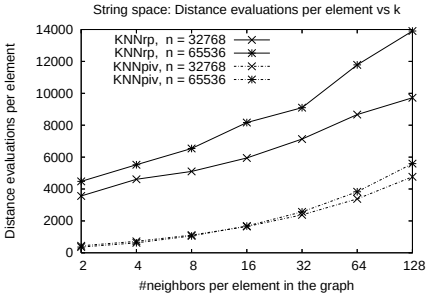
(e) In Gaussian space, dependence on n.



(f) In Gaussian space, dependence on k.



(g) In String space, dependence on n.



(h) In String space, dependence on k.

Fig. 3. Distance evaluations per node during kNNG construction. Fig. 3(b)/3(f) follows the legend of Fig. 3(a)/3(e).

Space	KNNrp Dist. evals.	KNNrp CPU time	KNNpiv Dist. evals.	KNNpiv CPU time
$[0,1]^4$	$10.0n^{1.32}$	$0.311n^{2.24}$	$56.1n^{1.09}$	$0.787n^{2.01}$
$[0,1]^8$	$32.8n^{1.38}$	$0.642n^{2.11}$	$168n^{1.06}$	$15.5n^{1.69}$
$[0,1]^{12}$	$15.1n^{1.59}$	$1.71n^{2.03}$	$116n^{1.27}$	$20.1n^{1.79}$
$[0,1]^{16}$	$5.06n^{1.77}$	$0.732n^{2.14}$	$12.1n^{1.64}$	$6.87n^{1.97}$
$[0,1]^{20}$	$2.32n^{1.88}$	$0.546n^{2.18}$	$2.48n^{1.87}$	$2.77n^{2.10}$
$[0,1]^{24}$	$1.34n^{1.96}$	$0.656n^{2.16}$	$1.23n^{1.96}$	$1.29n^{2.16}$
$[0,1]^D$	$0.455e^{0.19D}n^{1.65}$	$0.571e^{0.01D}n^{2.14}$	$0.685e^{0.23D}n^{1.48}$	$0.858e^{0.11D}n^{2.15}$
Gaussian $\sigma = 0.1$	$74.7n^{1.33}$	$1.13n^{2.07}$	$1260n^{0.91}$	$63.5n^{1.63}$
Gaussian $\sigma = 0.2$	$7.82n^{1.71}$	$1.13n^{2.09}$	$16.3n^{1.60}$	$8.70n^{1.94}$
Gaussian $\sigma = 0.3$	$2.97n^{1.85}$	$0.620n^{2.17}$	$3.86n^{1.81}$	$3.78n^{2.06}$
String	$21.4n^{1.54}$	$1.09n^{2.09}$	$99.9n^{1.26}$	$10.8n^{1.85}$
Document	$0.425n^{1.95}$	$193n^{1.96}$	$0.840n^{1.86}$	$364n^{1.87}$

Table 1. KNNrp and KNNpiv least square fittings for distance evaluations and CPU time for all the metric spaces. CPU time measured in microseconds.

to compute. Note that KNNrp and KNNpiv turn out to be clearly subquadratic on distances evaluations when considering the exponential dependence on D.

Note that in the metric space context, superquadratic CPU time in side computations is not as important as a subquadratic number of computed distances. In fact, in the document space, KNNrp and KNNpiv perform better in CPU time that KNNb, showing that in practice the leading complexity (computing distances) is several orders of magnitude larger than other side computations such as traversing pointers or scanning the pivot table.

4 Conclusions

We have presented a general methodology to construct the k-nearest neighbor graph (kNNG) in general metric spaces. Based on our methodology we give two algorithms. The first is based on a recursive partitioning of the space (KNNrp), and the second on the classic pivot technique (KNNpiv). Our methodology considers two stages: the first indexes the space, and the second completes the kNNG using the index and some metric and graph optimizations.

Experimental results confirm the practical efficiency of our approach in vectorial metric spaces of wide dimensional spectrum ($D \leq 20$), and real-world metric spaces. For instance, in the string space, our algorithms achieve empirical CPU time of the form $c_t n^{1.85}$, and $c_d n^{1.26}$ in distance computations; and in the high-dimensional document space, they reach empirical $cn^{1.87}$ both in distance computations and CPU time. In low dimensional metric spaces, our algorithms behave even better. KNNpiv is in general better than KNNrp for small and moderate k values, yet KNNrp is less sensitive to larger k values or higher dimensional spaces.

Future work involves developing another kNNG constructing algorithm based on the list of clusters [7] so that we can also obtain good performance in higher dimensional metric spaces. We are also researching how to enhance the data structure to allow dynamic insertions/deletions in reasonable time, so as to maintain an up-to-date set of k-nearest neighbors for each element in the database.

Acknowledgement We wish to thank Georges Dupret and Marco Patella for their valuable comments.

References

1. S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In *Proc. SODA'94*, pages 573–583, 1994.
2. F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), 1991.
3. R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query clustering for boosting web page ranking. In *Proc. AWIC'04*, LNCS 3034, pages 164–175, 2004.
4. M. Brito, E. Chávez, A. Quiroz, and J. Yukich. Connectivity of the mutual k-nearest neighbor graph in clustering and outlier detection. *Statistics & Probability Letters*, 35:33–42, 1996.
5. P. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proc. FOCS'93*, pages 332–340, 1993.
6. P. Callahan and R. Kosaraju. A decomposition of multidimensional point sets with applications to k nearest neighbors and n body potential fields. *JACM*, 42(1):67–90, 1995.
7. E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.
8. E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.
9. K. Clarkson. Fast algorithms for the all-nearest-neighbors problem. In *Proc. FOCS'83*, pages 226–232, 1983.
10. K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.
11. M. Dickerson and D. Eppstein. Algorithms for proximity problems in higher dimensions. *Computational Geometry Theory and Applications*, 5:277–291, 1996.
12. R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.
13. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.
14. D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete & Computational Geometry*, 11:321–350, 1994.
15. K. Figueroa. An efficient algorithm to all k nearest neighbor problem in metric spaces. Master's thesis, Universidad Michoacana, Mexico, 2000. In Spanish.
16. G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report TR 4199, Dept. of Comp. Sci. Univ. of Maryland, Nov 2000.
17. I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Trans. Software Eng.*, 9(5):631–634, 1983.
18. D. R. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics. In *Proc. STOC'02*, pages 741–750, 2002.
19. R. Krauthgamer and J. Lee. Navigating nets: simple algorithms for proximity search. In *Proc. SODA'04*, pages 798–807, 2004.
20. R. Paredes and E. Chávez. Using the k-nearest neighbor graph for proximity searching in metric spaces. In *Proc. SPIRE'05*, LNCS 3772, pages 127–138, 2005.
21. R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004.
22. P. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. *Discrete & Computational Geometry*, 4:101–115, 1989.