Practical Construction of k Nearest Neighbor Graphs in Metric Spaces *

Rodrigo Paredes and Gonzalo Navarro

Center for Web Research, Dept. of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago, Chile.
{raparede,gnavarro}@dcc.uchile.cl

Abstract. Let $\mathbb{U}$ be a set of elements and d a distance function defined among them. Let $NN_k(u)_d$ be the k elements in $\mathbb{U} - \{u\}$ which have the smallest distance to u. The k-nearest neighbors graph (kNNG) is a directed graph $G(\mathbb{U}, E)$ such that $E = \{(u, v, d(u, v)), v \in NN_k(u)_d\}$. We focus on the metric space context, so d is a metric. Several kNNGs construction algorithms are known, but they are not suitable to general metric spaces. We present two practical algorithms to construct kNNGs that exploit several features of metric spaces, obtaining time costs of the form $O(n^{1.63..2.24}k^{0.02..0.59})$, and using $O(n^{0.91..1.96}k^{0.04..0.66})$ distance computations.

1 Introduction

Let $\mathbb{U}$ be a set of elements and d a distance function defined among them. Let $NN_k(u)_d$ be the k elements in $\mathbb{U}-\{u\}$ which have the smallest distance towards u according to the function d. The *k-nearest neighbors graph* (kNNG) is a directed graph connecting each element to its k nearest neighbors, that is, $G(\mathbb{U}, E)$ such that $E = \{(u, v, d(u, v)), v \in NN_k(u)_d\}$. The kNNG is a direct generalization of the well known *all-nearest-neighbor* (ANN) problem, so ANN can be seen as the 1NNG problem. kNNGs are central in many applications, for instance cluster and outlier detection [13, 5], VLSI design, spin glass and other physical process simulations [7], and pattern recognition [11].

On the other hand, a metric space is a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is an object set and d is a function that measures the distance among them. The smaller the distance between two objects, the more "similar" they are. Given a finite metric space database $\mathbb{U} \subseteq \mathbb{X}$ of size n, the goal is to index $\mathbb{U}$ such that, given a query object $q \in \mathbb{X}$, one can find the elements of $\mathbb{U}$ close to q with as few distance computations as possible. See [8] for a comprehensive survey.

In some kNN applications objects have no coordinates, but they belong to abstract metric spaces. For instance in query or document recommendation systems, kNN based techniques are used to find clusters, and later exploit their properties to propose options to the final user [3, 4]. Unfortunately, most of the

* This work has been supported in part by the Millennium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

kNNG algorithms use object coordinates, so they are unsuitable to general metric spaces. We face this problem proposing two kNNG construction algorithms that exploit several features of metric spaces, obtaining time costs of the form $O(n^{1.63..2.24}k^{0.02..0.59})$ and using $O(n^{0.91..1.96}k^{0.04..0.66})$ distance computations.

The kNNG $G(\mathbb{U}, E)$ can serve as an index for searching metric spaces as well, as done in [16, 17]. In this case, each object in $\mathbb{U}$ is represented by a vertex in G, and the index is composed by E, that is, all the edges from the objects towards their k nearest neighbors.

1.1 A summary of metric spaces

Proximity/similarity queries can be formalized using the metric space model, where a distance function $d(x, y)$ is defined for every object pair in $\mathbb{X}$. Objects in $\mathbb{X}$ do not necessarily have coordinates (for instance, strings and images).

The distance function d satisfies the metric properties: $d(x, y) \geq 0$ (positiveness), $d(x, y) = d(y, x)$ (symmetry), $d(x, y) = 0$ iff $x = y$ (reflexivity), and $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality). The distance is considered expensive to compute (for instance, when comparing two documents).

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 can be of two basic types: (a) Range query $(q, r)_d$: retrieve all objects in $\mathbb{U}$ that are within distance r to q; and (b) k-Nearest neighbor query $NN_k(q)_d$: retrieve the k objects in $\mathbb{U}$ closest to q. The *covering radius* is the distance from q towards the farthest neighbor in $NN_k(q)_d$.

An *index* is a data structure built over $\mathbb{U}$ that permits solving the above queries without comparing q against every element in $\mathbb{U}$. There are two kinds of indices: pivot based and compact partition based. Search algorithms use the index and some distance evaluations to discard as many objects as they can to produce a small candidate set that could be relevant to q. Later, they exhaustively check the candidate set computing distances from q towards each candidate to obtain the query result. A k nearest neighbor query algorithm is called *range-optimal* if it uses the same amount of distance evaluations than the equivalent range query that retrieves exactly k objects [15].

There are three main components in the cost of computing a proximity query using an index: the number of distance evaluations, the CPU cost of the side computations (other than computing distances) and the number of I/O operations. However, the cost of computing a distance is so significant in many applications that it is customary to use the number of distance evaluations as the complexity measure both for index construction and for object retrieving.

The proximity query cost worsens quickly as the dimension of the space grows, which is called the *curse of dimensionality*. In $\mathbb{R}^D$ with points chosen randomly, the dimension is simply D. In metric spaces or in $\mathbb{R}^D$ where points are not uniformly distributed, the dimension can be defined using distance histogram properties [8]. In general terms, the dimension grows as the histogram concentrates.

1.2 Previous work in k nearest neighbor graphs

The naive approach to build kNNGs uses $\frac{n(n-1)}{2} \in O(n^2)$ distance computations and uses $O(kn)$ memory. For each $u \in \mathbb{U}$ we compute the distance towards all the others, and select the k lowest distance objects. However, there are several 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. ANN can be optimally solved in $O(n \log n)$ time in the plane and in $\mathbb{R}^D$ for any fixed D [9, 12, 18], but the constant depends exponentially on D. In $\mathbb{R}^D$, kNNG can be solved in $O(nk \log n)$ time [18] and even in $O(n(k + \log n))$ time [6, 7, 9]. Approximation algorithms for the problem have also been proposed [1]. Nevertheless, none of these alternatives, excepting the naive one, is suitable for metric spaces, since all use coordinate information that is not necessarily available in general metric spaces.

The generalization of ANN for general metric spaces was first stated in [10], where the problem is solved using randomization in $O(n(\log n)^2(\log \Gamma(\mathbb{U}))^2)$ expected time, with $\Gamma(\mathbb{U})$ the ratio between the distances among the farthest and closest pairs of points in $\mathbb{U}$. The author argues that in practice $\Gamma(\mathbb{U}) = \text{poly}(n)$, in which case the approach is $O(n(\log n)^4)$ time. However, the analysis needs a sphere packing bound in the metric space. Otherwise the cost must be multiplied by "sphere volumes", that is, extra factors that are similar to those that, in $\mathbb{R}^D$, are exponential on D. Moreover, the algorithm needs $\Omega(n^2)$ space for high dimensions, which is too much space for practical applications.

In [14], another technique for general metric spaces is given. It uses a pivot-based index in order to solve n range queries of decreasing radius. As it is well known, the performance of pivot-based algorithms worsen quickly as the dimension of the space grows, thus limiting the applicability of this technique. Our pivot based algorithm (Section 4) is an improvement over this technique.

1.3 Our contribution at a glance

We are interested in kNNG construction for general metric spaces. In this context, a natural approach to the kNNG construction has two stages. The first is to build an index of $\mathbb{U}$ using less than $\frac{n(n-1)}{2}$ distance computations, which we call the *preindex*. The second is to use the preindex to solve n k-nearest neighbor queries to obtain each $NN_k(u)_d$ subset for every $u \in \mathbb{U}$. This basic scheme can be improved if we take into account some observations:

- We can use the kNNG under construction to improve the second stage.
- We are solving queries for elements in $\mathbb{U}$, not for general objects in $\mathbb{X}$.
- We can solve the n queries jointly to share the costs through the whole process.
- There are several metric indices to preindex the metric space.
- We can use range-optimal k nearest neighbor queries.

In this work, we propose two kNNG construction algorithms that use a small preindex in order to decrease the number of distance computations. These are:

1. A recursive partition based algorithm: In the first stage, we build a preindex by performing a recursive partition of the space. In the second stage, we complete the k nearest neighbors using the order induced by the partitioning.
2. A pivot based algorithm: In the preindexing stage, we build a pivot index. Later, we complete the k nearest neighbors by performing range-optimal queries improved with metric and graph considerations.

Our algorithms use $O(n(k + \lg n))$ space and are subquadratic in distance evaluations. We evaluate them experimentally as a function of n, k, and the dimension of the space. We are not aware of any other practical kNNG implementation for metric spaces. The experiments confirm that our algorithms are efficient in CPU time and subquadratic in distance evaluations. For instance, in the string metric space with edit distance we obtain empirical time costs of the form $O(n^{1.85..2.10}k^{0.12..0.29})$ and $O(n^{1.26..1.54}k^{0.20..0.48})$ distance computations.

2 Some ingredients of the recipe

We explain here some elements that are common to our algorithms.

Two stages algorithm: We split the process into two stages: the first is the preindexing, and the second is to use the preindex to solve all the k nearest neighbor queries.

Neighbor Heap Array: Along all the algorithm, for each $u \in \mathbb{U}$ we maintain a *priority queue* NHA_u of size k. 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 $curCovR_u = d(u, x_{i_k})$ be an alias for the distance value towards the current farthest element of NHA_u, that is, the covering radius of the current k nearest neighbors of u.
We have to ensure that $|NHA_u| = k$ after successive additions. To achieve this, when we add some object v such that $d(u, v) < curCovR_u$, before actually adding $(v, d(u, v))$ to NHA_u we extract its farthest object. This will progressively reduce $curCovR_u$ from its initial value ∞ to the real $NN_k(u)_d$ covering radius. We first populate all the NHA_u queues with whatever elements u was compared to. Later, we will add more and more elements to them as the algorithm progresses, making their respective $curCovR_u$ radii decrease. At the end, in NHA we have the kNNG of $\mathbb{U}$.

Using *NHA* as a graph: Once we calculate $d(u, v)$, if $d(u, v) \geq curCovR_u$ we can discard v as a candidate for NHA_u, and moreover, we can discard all objects w such that $d(v, w) \leq d(u, v) - curCovR_u$ because of the triangle inequality. Unfortunately, we do not necessarily have stored $d(v, w)$. However, we can compute shortest paths over NHA from v to all $w \in \mathbb{U}$, and due to the triangle inequality, the sum of the edges traversed in this path, $d_{NHA}(v, w)$, is an upper bound to $d(v, w)$. So, if $d(u, v) \geq curCovR_u$, we also discard all objects w such that $d_{NHA}(v, w) \leq d(u, v) - curCovR_u$.

Exploiting the symmetry of d: Since d is a symmetric function, every time a $d(u, v)$ is computed, we check both $d(u, v) < curCovR_u$ to add $(v, d(u, v))$ to NHA_u, and $d(u, v) < curCovR_v$ to add $(u, d(u, v))$ to NHA_v. This can reduce $curCovR_v$, and can cheapen the future query for v.

Check Order Heap: When the second stage begins we create a priority queue $COH = \{(u, curCovR_u), u \in \mathbb{U}\}$ to process objects as they are extracted from COH in increasing $curCovR_u$ order. The goal is to solve the easiest queries first, both to reduce the CPU time and to increase the chance of reducing other $curCovR_v$. This is because a small radius query has larger discriminative power and produces candidates v that are closer to the query u, so it is also possible that these candidates reduce their respective $curCovR_v$.

$\mathbb{U}$ is fixed: Suppose that we are solving u, we are going to check a solved object v, and $curCov_u \leq curCov_v$. Then, if $u \notin NN_k(v)_d \Rightarrow d(u, v) \geq curCovR_v$, so, necessarily $v \notin NN_k(u)_d$. On the other hand, if $u \in NN_k(v)_d$, then we already computed $d(u, v)$. Therefore, in both cases we avoid to compute $d(u, v)$.

Limiting the space to store distances: We allow that our algorithms use at most $O(n \log n)$ extra memory to index $\mathbb{U}$.

3 Recursive partition based algorithm

This algorithm uses all the common ingredients, and some others. We use a cache of computed distances, *CD*, so that, every time we have to compute a distance, we check if it is present in *CD*, in which case we return the stored value. $CD \subset \mathbb{U}^2 \times \mathbb{R}^+$ can be seen as the graph of all stored distances. In the first stage, *CD* is an undirected graph since for each distance we store both directed edges. In the second stage, if a distance is sufficiently small so as to belong to *NHA*, we store it in *CD*, but only for unsolved nodes. However, adding distances to *CD* could increase its size beyond control, so we limit $|CD| < 8n \ln n$, and as soon as we finish the query for some node, we delete its adjacency list from *CD*. Therefore, in the second stage *CD* becomes a directed graph. The memory limitation comes from an estimation of how many edges we store in *CD* during the division procedure, which is explained soon.

3.1 First stage: Preindex construction

The first stage consists in partitioning the space recursively. During the partition we symmetrically populate *NHA* and *CD* with all the computed distances. The goal is to construct the *Division Control Tree*, *DCT*.

DCT is a binary tree that represents the shape of the partitioning and gives the partition radius for each node. The partition radius is the distance from the node towards the farthest node of its partition. The *DCT* node structure is $\{p, l, r, pr\}$, which represents the node parent, its left and right children, and its partition radius, respectively. For simplicity we use the same name for the node $u \in \mathbb{U}$ and for its representative in *DCT*. Then, given a node $u \in \mathbb{U}$, u_p,

u_l, and u_r, refer to the nodes that are the parent, left child, and right child of u in DCT, respectively, and also to their representative nodes in $\mathbb{U}$. Finally, u_{pr} refers to the partition radius of u.

DCT is constructed by the **division** procedure. This procedure receives an object set S, a root node, and DCT. To split S, we take two far away objects, u and v, and then generate two subsets: S_u, objects nearer to u, and S_v, objects nearer to v. Later, we compute the u and v partition radii, and finally we update the root in the children and their partition radii in DCT. To choose u and v, we take a sample of $|S|/2$ object pairs at random and pick the farthest pair. Later, the recursion follows with (u, S_u) and (v, S_v), and it finishes when $|S| < 2$. Once we finish the division, leaves in DCT have partition radii 0. The DCT root is a fictitious node: it is the only one that does not have an equivalent in $\mathbb{U}$, it has partition radius ∞, and its children are the two nodes of the first division.

Since the DCT has n nodes, its expected height is $2 \ln n$ (the DCT construction is statistically identical to populate a binary search tree). For each DCT level, each node computes two distances towards 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 have the $8n \ln n$ space limitation. Therefore, in this algorithm we use the extra $O(n \log n)$ space to store both the DCT and CD.

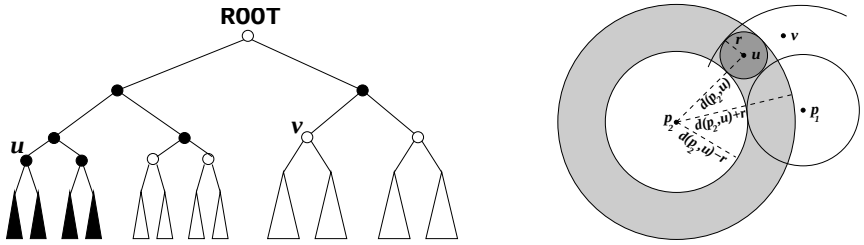
3.2 Using the DCT in the second stage

The DCT allows us to save distance evaluations in the second stage. Assume we are checking whether v is relevant to u. If $d(u, v) \geq curCovR_u + v_{pr}$ we discard all the descent of v, because there is no intersection between the space region centered in u with radius $curCovR_u$ and the one centered in v with radius v_{pr}.

Furthermore, the DCT gives us a mechanism to complete the k nearest neighbor query. Due to the division procedure, we are sure that u has already computed distances to all of its ancestors, its ancestor's siblings, and its parent descent. Then, to finish the query for u, we need to verify whether there exist relevant objects in the ancestor's sibling descents. Figure 1(a) illustrates this. However, since the DCT allows us to discard whole descents, it is enough with managing the set of ancestor's sibling children. This set is composed by u's cousins, u's parent cousins, and so on, and we call it CS (cousin set). Actually, we store u's ancestor siblings, not cousins, and if it is not possible to discard the sibling (and its descent), then we add its children (the cousins).

Note that, when we start processing CS, partition radii of u's cousins are likely to be the smallest of CS, and it is also probable that these cousins, or their descendants, are relevant to u, since these nodes share most of u's branch in DCT, and the partitioning is made according to node closeness. Thus, we process CS using a priority queue sorted by increasing partition radius.

We also avoid distance evaluations using $CD \cup NHA$ as a graph. If $d(u, v) \geq curCovR_u$, we discard all objects such that $d_{CD \cup NHA}(v, w) \leq d(u, v) - curCovR_u$. We do this by marking them as `EXTRACTED`. This task is performed by **ex-**



(a) In DCT, u has been compared with all black nodes and to all the descent of its parent. To finish the query, we have to process the white nodes and subtrees.

(b) The ring of elements not discarded by pivot p_2. p_1 cannot discard v, however p_2 can.

Fig. 1. Using DCT to finish the query (left). The pivot discard condition (right).

tractFrom, which is a shortest paths algorithm with propagation limited to $d(u, v) - curCovR_u$.

3.3 Second stage: The whole picture

We use all of these ideas in the second stage in order to complete the k nearest neighbors for all nodes $u \in \mathbb{U}$. To do this, we create the priority queue COH to process objects by picking them from COH in increasing $curCovR$ order. For each node u, we add the edges of NHA_u to CD_u (note that, due to the CD size limitation, it is possible that we do not already have all the current neighbors in CD_u). Second, we call **extractFrom** for all u's ancestors, marking as `EXTRACTED` as many objects as we can. Then, we finish the query using the procedure **neighborComplete**. Finally, we delete CD_u.

The **neighborComplete** procedure receives the node u to be processed. First, we build the priority queue CS with all u's ancestor siblings. Later, we take objects c from CS in increasing order of c_{pr}, and process c according to the following rules:

1. If c was already marked as `EXTRACTED`, we add its children to CS, and continue with the next cousin.
2. If c was already solved, $curCovR_c \geq curCovR_u$, and $d(u, c) \notin CD$, we add its children to CS, and continue with the next cousin.
3. If we have $d(u, c)$ precomputed in CD, we retrieve it, else we compute it. If $d(u, c) < curCovR_u + c_{pr}$, we have region intersection, so we add the c's children to CS. Later, if $d(u, c) > curCovR_u$, we call **extractFrom** to mark objects as `EXTRACTED` from c limited to $d(u, c) - curCovR_u$.

Note that, if we calculate the amount of distance we could insert in NHA, we obtain an upper bound for the expected size of $|CD|$. The initial memory is used to store the first k distances. Later, with probability $\frac{n-k}{n}$, a random distance is greater than the k-th shortest one, and, with probability $\frac{k}{n}$ it is lower. We only store distances in the second case, using one cell for each distance.

Then, the recurrence for the average case of edge insertions for each NHA_u is: $T(n,k) = T(n-1,k)+1\cdot\frac{k}{n}, T(k,k) = k$. So, we obtain that $T(n,k) = k(1+\lg\frac{n}{k}) \in O(k\lg\frac{n}{k})$. As we have n priority queues, we have $O(nk\lg\frac{n}{k})$ memory. As can be seen, we really need to limit the size of CD to a reasonable one.

4 Pivot based algorithm

Pivot based algorithms have good performance in low dimensional spaces, but their performance worsens quickly as the dimension grows. So we use all the common ingredients to compensate this failure in medium and high dimensions. In this algorithm we use the whole extra $O(n\lg n)$ space to store a pivot index.

Pivot algorithms select a set of objects, the pivots, $\mathcal{P} = \{p_1,\ldots,p_{|\mathcal{P}|}\} \subseteq \mathbb{U}$, and store a table of $|\mathcal{P}|n$ distances $d(p_j,u),\ j \in \{1\ldots|\mathcal{P}|\},\ u \in \mathbb{U}$. Then, in the first stage we build the pivot table. To determine $|\mathcal{P}|$, we give the same space in bytes to the table as that of the cache of computed distances of the recursive based algorithm, so $|\mathcal{P}|n = |CD| + |DCT|$. Therefore, $|\mathcal{P}|$ is $O(\log n)$.

Since pivots compute distances towards all the objects, once we compute the table, they have solved their k nearest neighbors. On the other hand, due to the symmetry of d, objects in $\mathbb{U}-\mathcal{P}$ already have candidates to k nearest neighbors in NHA, then we complete $n-|\mathcal{P}|$ range-optimal queries for them in the second stage.

We want to solve first the objects which have the lowest $curCovR_u$, so we sort objects $u \in \mathbb{U}-\mathcal{P}$ in COH according to their $curCovR_u$. Note that the k nearest neighbor candidate sets are refined as the algorithm computes more and more distances, so we reposition objects u in COH during the algorithm upon any distance computation that modifies some NHA_u. Even though this does not save distance computations, it reduces the CPU time.

To perform a range-optimal query for u, we need an array, $Diff$, and a priority queue, $SortDiff$. Because of the triangle inequality, for each $v \in \mathbb{U}$, $|d(v,p_j) - d(u,p_j)|$ is a lower bound of $d(u,v)$. Let $Diff_v$ be the maximum lower bound of $d(u,v)$ using all the pivots, then $Diff_v = \max_{p\in\mathcal{P}}\{|d(v,p) - d(u,p)|\}$. We can discard v if $Diff_v \geq curCovR_u$. Figure 1(b) shows the concept graphically. In this case $Diff_v = \max\{|d(v,p_1) - d(u,p_1)|, |d(v,p_2) - d(u,p_2)|\}$ $= |d(v,p_2) - d(u,p_2)| > r$, thus v is discarded by p_2. Later, we create a priority queue $SortDiff = \{(v, Diff_v), v \in \mathbb{U} - (\mathcal{P} \cup NHA_u \cup \{u\})\}$ sorted by increasing order of $Diff_v$ to process each object v. Upon $SortDiff$ is created, we start to pick objects v from $SortDiff$ from smaller to larger $Diff_v$. For each object v we compute the distance $d(u,v)$ and if $d(u,v) < curCovR_u$ we add v to NHA_u. If $Diff_v > curCovR_u$ we stop processing $SortDiff$ and are done with u.

Note that, it is possible that $curCovR_u$ will decrease during the $SortDiff$ reviewing. We can improve the CPU time if we take into account two facts. First, it is not always necessary to calculate the maximum difference for each node v. In practice to discard v, it is enough to find some lower bound greater than $curCovR_u$, not necessarily the maximum. Thus we can learn that $Diff_u > curCovR_v$ without fully compute the maximum of $Diff_u$ formulae. Second, it

is not necessary to add all objects in $\mathbb{U} - (\mathcal{P} \cup NHA_u \cup \{u\})$ to $SortDiff$, if $Diff_v \geq curCovR_u$ we already know that v will not be reviewed.

During the reviewing of $SortDiff$, we use the fact that $\mathbb{U}$ is fixed to avoid some distance computations. On the other hand, if we cannot avoid the computation, for the objects v that we evaluate the distance $d(u, v)$, we use the symmetry of d to update NHA_v and replace v in COH if applicable, and later, in **extractFrom** we use NHA as a graph to discard from $SortDiff$ all the objects w such that $d_{NHA}(v, w) \leq d(u, v) - curCovR_u$.

5 Experimental results

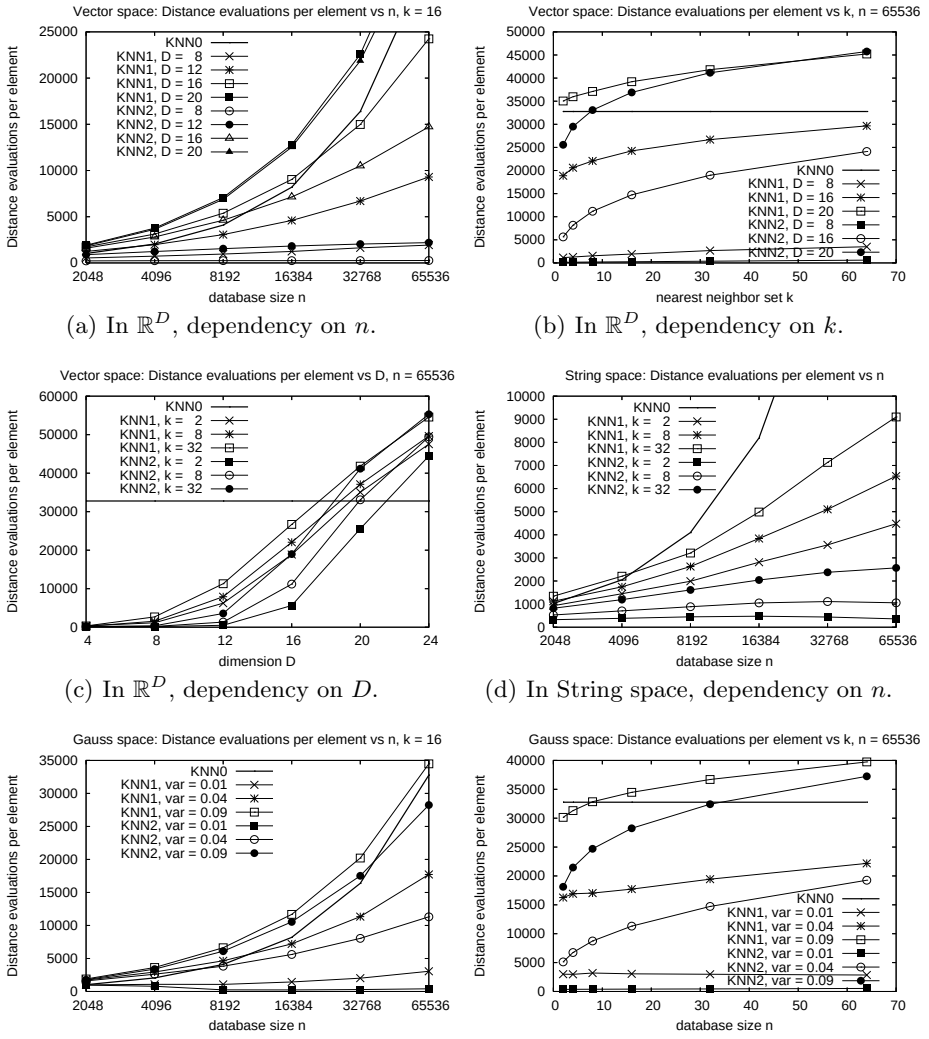
We have tested our algorithms on synthetic and real-world metric spaces. The first synthetic set is formed by 65,536 points uniformly distributed in the metric space of $[0, 1]^D$ (that is, the unitary real D-dimensional cube). This space allows us to measure the effect of the space dimension D on our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

The real-world set is the string metric space using the edit distance (a discrete function that measures the minimum number of character insertions, deletions and replacements needed to make the strings equal). The strings come from an English dictionary, where we index a random subset of 65,536 words.

The experiments were run on an Intel Pentium IV of 2 GHz and 512 MB of RAM. We measure distance evaluations and CPU time for each algorithm. For shortness we have called KNN0 the basic kNNG construction algorithm, KNN1 the recursive partition based algorithm, and KNN2 the pivot based algorithm.

We summarize our experimental results for the three metric spaces in Figure 2 and Table 1. Figure 2 shows distance computations per element. Table 1 shows our least squares fittings using the models $DistEval = O(n^\alpha k^\beta)$ and $time = O(n^\alpha k^\beta)$.

Figures 2(a), 2(b) and 2(c) show experimental results for $\mathbb{R}^D$. Figure 2(a) shows that for all dimensions KNN1 and KNN2 are subquadratic in distance evaluations, instead of KNN0 which is always $O(n^2)$. For low and medium dimensions, $D < 16$, they have better performance than KNN0 for all dimensions $D < 20$. Moreover, for lower dimensions, $D < 8$, they are only slightly superlineal. For all dimensions $D \leq 20$ KNN2 has better performance than the others. Figure 2(b) shows a sublinear dependency on k for all dimension for both algorithms, however, KNN2 is more sensitive to k than KNN1. Also, it shows that for $k \leq 4$, our algorithms behave better than KNN0. Figure 2(c) shows that, as D grows, the performance of KNN1 and KNN2 degrades. This phenomenon is known as the *curse of dimensionality*. For instance, for $D = 4$, KNN2 is



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

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

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

(d) In String space, dependency on n.

(e) In Gaussian space, dependency on n.

(f) In Gaussian space, dependency on k.

Fig. 2. Distance evaluations during construction per node as a function of database size n, nearest neighbor set size k and the dimension D.

$O(n^{1.10}k^{0.13})$ distance evaluations, but, for $D = 24$, it is $O(n^{1.96}k^{0.04})$ distance evaluations.

Figures 2(e) and 2(f) show that 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 KNN0 for the three variances. They also show that KNN2 has the best performance.

Space	KNN1 Dist. evals.	KNN1 CPU time	KNN2 Dist. evals.	KNN2 CPU time
$[0,1]^4$	$O(n^{1.32}k^{0.25})$	$O(n^{2.24}k^{0.23})$	$O(n^{1.10}k^{0.13})$	$O(n^{2.01}k^{0.59})$
$[0,1]^8$	$O(n^{1.38}k^{0.30})$	$O(n^{2.12}k^{0.17})$	$O(n^{1.06}k^{0.48})$	$O(n^{1.70}k^{0.40})$
$[0,1]^{12}$	$O(n^{1.59}k^{0.19})$	$O(n^{2.03}k^{0.15})$	$O(n^{1.27}k^{0.55})$	$O(n^{1.79}k^{0.36})$
$[0,1]^{16}$	$O(n^{1.77}k^{0.10})$	$O(n^{2.14}k^{0.08})$	$O(n^{1.64}k^{0.27})$	$O(n^{1.97}k^{0.18})$
$[0,1]^{20}$	$O(n^{1.89}k^{0.06})$	$O(n^{2.18}k^{0.03})$	$O(n^{1.87}k^{0.10})$	$O(n^{2.10}k^{0.07})$
$[0,1]^{24}$	$O(n^{1.96}k^{0.03})$	$O(n^{2.16}k^{0.02})$	$O(n^{1.96}k^{0.04})$	$O(n^{2.16}k^{0.04})$
Gaussian $\sigma = 0.1$	$O(n^{1.33}k^{0.20})$	$O(n^{2.07}k^{0.18})$	$O(n^{0.91}k^{0.66})$	$O(n^{1.63}k^{0.48})$
Gaussian $\sigma = 0.2$	$O(n^{1.71}k^{0.13})$	$O(n^{2.10}k^{0.09})$	$O(n^{1.60}k^{0.31})$	$O(n^{1.94}k^{0.22})$
Gaussian $\sigma = 0.3$	$O(n^{1.85}k^{0.07})$	$O(n^{2.17}k^{0.04})$	$O(n^{1.81}k^{0.14})$	$O(n^{2.06}k^{0.10})$
String	$O(n^{1.54}k^{0.20})$	$O(n^{2.10}k^{0.12})$	$O(n^{1.26}k^{0.48})$	$O(n^{1.85}k^{0.29})$

Table 1. KNN1 and KNN2 empirical models for distance evaluations and CPU time for the three metric spaces.

Figure 2(d) shows the results in the string space. The graphic shows that both KNN1 and KNN2 are subquadratic for all $k \in [2, 32]$. For instance, for $n = 65536$, KNN1 costs 28% of KNN0 to build the 32NNG, and KNN2 just 8% of KNN0. This shows that our algorithm is also practical in real-world situations. Again, KNN2 shows better performance than KNN1.

All of these conclusions (for three spaces) 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 $k \leq 8$, KNN2 has better performance than KNN0 in CPU time. This is important since the Euclidean distance is very cheap to compute.

6 Conclusions

We have presented two kNNGs construction algorithms that consider and exploit several features of metric spaces. Our algorithms have two stages: the first indexes the space through a recursive partition of the space (KNN1) or a basic pivot technique (KNN2), and the second completes the k nearest neighbors using the indices and some metric and graph optimizations.

We have shown experimental results that confirm the practical efficiency of our algorithms in metric spaces of wide dimensional spectrum ($2 \leq D \leq 20$). For low dimensions ($D \leq 8$) our algorithms are slightly superlineal in distance evaluations. For higher dimensions they are subquadatics. For instance, in the space of vectors in $[0,1]^{12}$, KNN1 is $O(n^{1.59})$ distance evaluations and KNN2 is $O(n^{1.27})$ distance evaluations. We also have this performance in crisp and medium cluster Gaussian spaces and in the metric space of strings.

We have to remark that most of the KNN2 improvements to overcome the *curse of dimensionality* are by-products of the KNN1 development. Even though both of our algorithms are by far better than the naive one, KNN2 shows more promise for our future work, which involves using kNNGs to solve

proximity queries. Another challenge is to enhance the data structure to allow 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 Edgar Chávez, Marco Patella and Georges Dupret 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 *AWIC*, LNCS 3034, pages 164–175, 2004.
4. R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query recommendation usign query logs in search engines. In *EDBT Workshops*, LNCS 3268, pages 588–596, 2004.
5. 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.
6. P. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. In *Proc. FOCS'93*, pages 332–340, 1993.
7. 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.
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. R. O. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.
12. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.
13. D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete & Computational Geometry*, 11:321–350, 1994.
14. K. Figueroa. An efficient algorithm to all k nearest neighbor problem in metric spaces. Master's thesis, Universidad Michoacana, Mexico, 2000. In Spanish.
15. G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report TR 4199, Department of Computer Science, University of Maryland, November 2000.
16. G. Navarro and R. Paredes. Practical construction of metric t-spanners. In *Proc. ALENEX'03*, pages 69–81, 2003.
17. G. Navarro, R. Paredes, and E. Chávez. t-Spanners as a data structure for metric space searching. In *Proc. SPIRE'02*, LNCS 2476, pages 298–309, 2002.
18. P. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. *Discrete & Computational Geometry*, 4:101–115, 1989.