Using the k-Nearest Neighbor Graph for Proximity Searching in Metric Spaces *

Rodrigo Paredes[1] and Edgar Chávez[2]

[1] Center for Web Research, Dept. of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago, Chile.
raparede@dcc.uchile.cl
[2] Escuela de Ciencias Físico-Matemáticas, Univ. Michoacana, Morelia, Mich. México.
elchavez@fismat.umich.mx

Abstract. Proximity searching consists in retrieving from a database, objects that are *close* to a query. For this type of searching problem, the most general model is the *metric space*, where proximity is defined in terms of a *distance* function. A solution for this problem consists in building an *offline* index to quickly satisfy *online* queries. The ultimate goal is to use as few distance computations as possible to satisfy queries, since the distance is considered expensive to compute. Proximity searching is central to several applications, ranging from multimedia indexing and querying to data compression and clustering.
In this paper we present a new approach to solve the proximity searching problem. Our solution is based on indexing the database with the k-nearest neighbor graph (kNNG), which is a directed graph connecting each element to its k closest neighbors.
We present two search algorithms for both range and nearest neighbor queries which use navigational and metrical features of the kNNG graph. We show that our approach is competitive against current ones. For instance, in the document metric space our nearest neighbor search algorithms perform 30% more distance evaluations than AESA using only a 0.25% of its space requirement. In the same space, the pivot-based technique is completely useless.

1 Introduction

Proximity searching is the search for *close* or *similar* objects in a database. This concept is a natural extension of the classical problem of exact searching. It is motivated by data types that cannot be queried by exact matching, such as multimedia databases containing images, audio, video, documents, and so on. In this new framework the exact comparison is just a type of query, while close or similar objects can be queried as well. There exists a large number of computer applications where the concept of similarity retrieval is of interest. This applications include *machine learning and classification*, where a new element

* This work has been supported in part by the Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and CYTED VII.19 RIBIDI Project.

must be classified according to its closest existing element; *image quantization and compression*, where only some samples can be represented and those that cannot must be coded as their closest representable one; *text retrieval*, where we look for words in a text database allowing a small number of errors, or we look for documents which are similar to a given query or document; *computational biology*, where we want to find a DNA or protein sequence in a database allowing some errors due to typical variations; and *function prediction*, where past behavior is extrapolated to predict future behavior, based on function similarity. See [6] for a comprehensive survey on proximity searching problems.

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 or fingerprints).

We have a finite *database* of interest $\mathbb{U}$ of size n, which is a subset of the universe of objects $\mathbb{X}$ and can be preprocessed to build a search index.

A proximity query consists in retrieving objects from $\mathbb{U}$ which are close to a new object $q \in \mathbb{X}$. There are two basic proximity queries:

Range query $(q, r)_d$**:** Retrieve all elements in $\mathbb{U}$ which are within distance r to $q \in \mathbb{X}$. This is, $(q, r)_d = \{u \in \mathbb{U} \ / \ d(q, u) \leq r\}$.

Nearest neighbor query $NN_k(q)_d$**:** Retrieve the k closest elements in $\mathbb{U}$ to $q \in \mathbb{X}$. This is, $|NN_k(q)_d| = k$, and $\forall\ u \in NN_k(q)_d, v \in \mathbb{U} - NN_k(q)_d$, $d(u, q) \leq d(v, q)$.

There are some considerations about $NN_k(q)_d$. In case of ties we choose any k-element set that satisfies the query. The query *covering radius* cr_q is the distance from q towards the farthest neighbor in $NN_k(q)_d$. Finally, a $NN_k(q)_d$ algorithm is called *range-optimal* if it uses the same number of distance evaluations than a range search with radius the distance to the k-th closest element [11].

An *index* is a data structure built *offline* over $\mathbb{U}$ to quickly solve proximity queries *online*. Since the distance is considered expensive to compute the goal of an index is to save distance computations. Given the query, we use the index to discard as many objects from the database as we can to produce a small set of candidate objects. Later, we check exhaustively the candidate set to obtain the query outcome.

There are three main components in the cost of computing a proximity query using an index, namely: the number of distance evaluations, the CPU cost of side computations (other than computing distances) and the number of I/O operations. However, in most applications the distance is the leader complexity measure, and it is customary to just count the number of computed distances to compare two algorithms. This measure applies to both index construction and object retrieval. For instance, computing the cosine distance [3] in the document metric space takes 1.4 msecs in our machine (Pentium IV, 2 GHz), which is really costly.

An important parameter of a metric space is its intrinsic dimensionality. In $\mathbb{R}^D$ with points distributed uniformly the intrinsic dimension is simply D. In metric spaces or in $\mathbb{R}^D$ where points are not chosen uniformly, the intrinsic dimensionality can be defined using the distance histogram [6]. In practice, the proximity query cost worsens quickly as the space dimensionality grows. In fact, an efficient method for proximity searching in low dimensions may become painfully slow in high dimensions. For large enough dimensions, no proximity search algorithm can avoid comparing the query against all the database.

1.1 A Note on *k*-Nearest Neighbor Graphs

The *k-nearest neighbors graph* (kNNG) is a directed graph connecting each element to its k nearest neighbors. That is, given the element set $\mathbb{U}$ the kNNG is a graph $G(\mathbb{U}, E)$ such that $E = \{(u, v, d(u, v)), v \in NN_k(u)_d\}$, where each $NN_k(u)_d$ represent the outcome of the nearest neighbor query for each $u \in \mathbb{U}$.

The kNNG is interesting *per se* in applications like cluster and outlier detection [9, 4], VLSI design, spin glass and other physical process simulations [5], pattern recognition [8], and query or document recommendation systems [1, 2]. This contribution starts with the kNNG graph already built, we want to prove the searching capabilities of this graph. However, we show some specific kNNG construction algorithms for our present metric space application in [14].

Very briefly, the kNNG is a direct extension of the well known *all-nearest-neighbor* (ANN) problem. A naïve approach to build kNNG uses $\frac{n(n-1)}{2} = O(n^2)$ distance computations and $O(kn)$ memory. Although there are several alternatives to speed up the procedure, most of them are unsuitable for metric spaces, since they use coordinate information that is not necessarily available in general metric spaces. As far as we know, there are three alternatives in our context.

Clarkson generalized the ANN problem for general metric spaces solving the ANN by using randomization in $O(n(\log n)^2(\log \Gamma(\mathbb{U}))^2)$ expected time, where $\Gamma(\mathbb{U})$ is the distance ratio between the farthest and closest pair of points in $\mathbb{U}$ [7]. The technique described there is mainly of theoretical interest, because the implementation requires $o(n^2)$ space.

Later, Figueroa proposes build the kNNG by using a pivot-based index so as to solve n range queries of decreasing radius [10]. As it is well known, the performance of pivot-based algorithms worsen quickly as the space dimensionality grows, thus limiting the applicability of this technique.

Recently, we propose two approaches for the problem which exploit several graph and metric space features [14]. The first is based on recursive partitions, and the second is an improvement over the Figueroa's technique. Our construction complexity for general metric spaces is around $O(n^{1.27}k^{0.5})$ for low and medium dimensionality spaces, and $O(n^{1.90}k^{0.1})$ for high dimensionality ones.

1.2 Related Work

We have already made another attempt about using graph based indices for metric space searching by exploring the idea of indexing the metric space with a

t-spanner [12, 13]. In brief, a t-spanner is a graph with a bounded stretch factor t, hence the distance estimated through the graph (the length of the shortest path) is at most t times the original distance. We show that the t-spanner based technique has better performance searching real-world metric spaces than the obtained with the classic pivot-based technique. However, the t-spanner can require much space. With the kNNG we aim at similar searching performance using less space.

In the experiments, we will compare the performance of our searching algorithms against the basic pivot-based algorithm and AESA [15]. It is known that we can trade space for time in proximity searching in the form of more pivots in the index. So, we will compare our kNNG approach to find out how much memory a pivot-based algorithm need to use to be as good as the kNNG approach. Note that all the pivot-based algorithms have similar behavior in terms of distance computations, being the main difference among them the CPU time of side computations. On the other hand, we use AESA just like a baseline, since its huge $O(n^2)$ memory requirement makes this algorithm suitable only when n is small. See [6] for a comprehensive explanations of these algorithms.

1.3 Our Contribution

In this work we propose a new class of proximity searching algorithms using the kNNG as the data structure for searching $\mathbb{U}$. This is the first approach, up to the best of our knowledge, using the kNNG for metric searching purposes.

The core of our contribution is the use of the kNNG to estimate both an upper bound and a lower bound of the distance to the query from the database elements. Once we compute $d(q, u)$ for some u we can upper bound the distance from q to many database objects (if the graph is connected, we upper bound the distance to all the database objects). We can also lower bound the distance from the query to the neighbors of u. The upper bound allows the elimination of far-from-the-query elements whilst the lower bound can be used to test if an element can be in the query outcome.

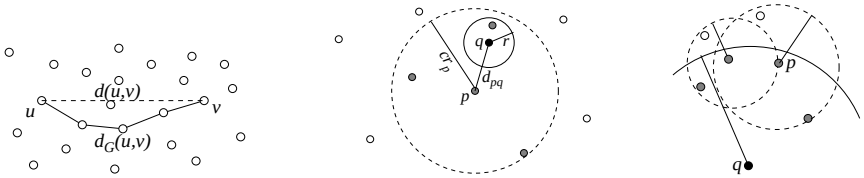
As we explain later (Sections 2 and 3), this family of algorithms have a large number of design parameters affecting its efficiency (not the correctness). We tried to explore all the parameters experimentally in Section 4.

We selected two sets of heuristics rising two metric range query algorithms, and building on top of them we designed two nearest neighbor search algorithms.

The experiments confirm that our algorithms are efficient in distance evaluations. For instance, in the document metric space with cosine distance our nearest neighbor query algorithms just perform 30% more distance evaluations than AESA, but only using a 0.25% of its space requirement. In the same space, the pivot-based technique is completely useless.

2 kNNG-based Range Query Algorithms

Given an arbitrary subgraph of the distance matrix of $\mathbb{U}$, one can upper bound the distance between two objects by using the shortest path between them.



(a) We upper bound the distance $d(u, v)$ with the length of the shortest path $d_G(u, v)$.

(b) If the ball (p, cr_p) covers (q, r), we make $C = (p, cr_p)$.

(c) If we find an object $p \in (q, r)_d$, we check its neighborhood.

Fig. 1. Using the kNNG features. In 1(a), approximating the distance in the graph. In 1(b), using the container. In 1(c), checking the neighborhood.

Formally, $d(u, v) \leq d_G(u, v)$ where $d_G(u, v)$ is the distance in the graph, that is, the length of the shortest path between the objects. Figure 1(a) shows this.

A generic graph-based approach for solving range queries consists in starting with a set of candidate nodes C of the provable smallest set containing $(q, r)_d$. A fair choice for an initial C is the whole database $\mathbb{U}$. Later, we iteratively extract an object u from C and if $d(u, q) \leq r$ we report u as part of $(q, r)_d$. Otherwise, we delete all the objects v such that $d_G(u, v) < d(u, q) - r$. Figure 2(a) illustrates this, we discard all the gray nodes because their distance estimations are small enough. We repeat the above procedure as long as C have candidate objects. In this paper we improve this generic approach using the kNNG properties.

Using Covering Radius. Notice that each node in kNNG has a covering radius cr_u (the distance towards its k-th neighbor). If the query ball $(q, r)_d$ is contained in $(u, cr_u)_d$ we can make $C = (u, cr_u)_d$, and proceed iteratively from there. Furthermore, we can keep track of the best fitted object, considering both its distance to the query and its covering radius using the equation $cr_u - d(u, q)$. The best fitted object will be the one having the largest difference, we call *container* this difference. Figure 1(b) illustrates this. So, once the container is larger than the searching radius we can make $C = (u, cr_u)_d$ as stated above. The probability of hitting a case to apply the above heuristic is low; but it is simply to check and the low success rate is compensate with the dramatic shrink of C when applied.

Propagating in the Neighborhood of the Nodes. Since we are working over a graph built by an object closeness criterion, if an object p is in $(q, r)_d$ it is likely that some of its neighbors are also in $(q, r)_d$. Moreover, since the out-degree of a kNNG is a small constant, spending some extra distance evaluations on neighbors of processed nodes do not add a large overhead to the whole process.

So, when we found an object belonging to $(q, r)_d$, it is worth to examine its neighbors, and, as with any other examination update the container. Note that every time we hit an answer we recursively check all of its neighbors. Special care must be taken to avoid multiple checks or cycles. Figure 1(c) illustrates this.

Note also that since we can lower bound the distance from the query to the neighbors of a processed object, we can discard some neighbors without directly computing the distance. Figure 2(b) illustrates this.

Working Evenly in All Graph Regions. Since we use path expansions from some nodes it is important to choose them scattered in the graph to avoid concentrating efforts in the same graph region. Otherwise, we will compute a path several times. A good idea is to select elements far apart from q and the previous selected nodes, because these nodes would have major potential of discarding non-relevant objects. Unfortunately, the selection of distant objects cannot be done by directly computing the distance to q. However, we can estimate "how visited" is some region. In fact, our two range query algorithms differ essentially in the way we select the next node to review.

2.1 First Heuristic for Metric Range Query (*k*NNG**RQ1).**

In this heuristic we prefer to start shortest path computations from nodes with few discarded neighbors, since these nodes have major discarding potential. Additionally, we also consider two criteria so as to break ties. The second criterion is to use nodes with small covering radius, and the third is that we do not want to restart elimination in an already visited node, and between two visited nodes we will choose the least traversed. So, to select a node, we consider the following:

1. How many neighbors already discarded has the node. Nodes with few discarded neighbors have major discarding potential, so they can reduce heavily the number of distance computations performed to solve the query.
2. The size of the covering radius. Objects having small covering radius, that is, very close neighbors, have major chance of discarding them (since if $cr_u < d(u,q) - r$, all its neighbors are discarded). Moreover, it is also likely that distance estimations computed from u would have tighter upper bounds.
3. The number of times the node was checked in a path expansion (when computing the graph distance). We prefer a node that it had been checked few times in order to scatter the search effort on the whole graph.

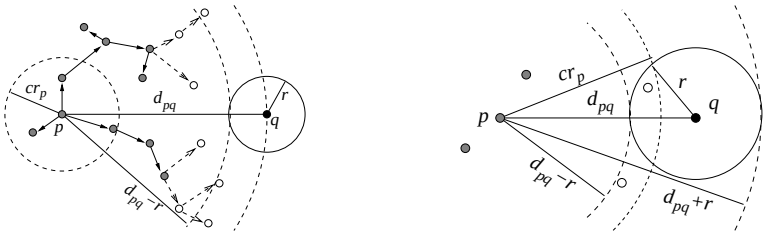
The above heuristics are condensed in Eq. (1).

$$p = \text{argmin}_{u \in C}\{|\mathbb{U}| \cdot (dn_u + f(u)) + \#visit\} \tag{1}$$

With $f(u) \in [0,1]$, $f(u) = \frac{cr_u - cr_{min}}{cr_{max} - cr_{min}}$, and $cr_{min} = \min_{u \in \mathbb{U}}\{cr_u\}$, $cr_{max} = \max_{u \in \mathbb{U}}\{cr_u\}$, and dn_u represents the number of discarded neighbors of u. Note that in Eq. (1) the leading term selects nodes with few discarded neighbors, the second term is the covering radius and the last term the number of visits.

The equation is computed iteratively for every node in the graph. For each node we save the value of Eq. (1) and every time we visit a node we update the heuristic value accordingly. Figure 2(a) illustrates this. Note that when we start a shortest path expansion we can discard some nodes (the gray ones), but for those that we cannot discard (the white nodes) we update their value of Eq. (1).

Please note that when we compute the graph distance (the shortest path between two nodes), we use a variation of Dijkstra's all shortest path algorithm which limits the propagation up to an estimation threshold, since a distance estimation grater that $d(u,q) - r$ cannot be used to discard nodes.



(a) The balls do not intersect each other. (b) The balls intersect each other.

Fig. 2. In 2(a), we extract gray objects which have a distance estimation lower that $d_{pq} - r$ and count visits to the white ones which have estimations lower than d_{pq}. In 2(b), we use p as a pivot discarding its gray neighbors when the distance from p towards them is not in $[d_{pq} - r, d_{pq} + r]$, else, we count the visit to the white nodes.

2.2 Second Heuristic for Metric Range Query (*k*NNG**RQ2).**

A different way to select a scattered element set is by using the graph distance. More precisely we assume that if two nodes are far apart according to the graph distance, they are also far apart using the original distance. The idea is to select the object with the largest sum of graph distances to all the previously selected objects. From other point of view, this heuristic tries to start shortest path computations from outliers.

3 *k*NNG-based Nearest Neighbor Queries

Range query algorithms naturally induce nearest neighbor searching algorithms. To this end, we use the following ideas:

- We simulate the nearest neighbor query using a range query of decreasing radius, which initial radius cr_q is ∞.
- We manage an auxiliary set of nearest neighbor candidates of q known up to now, so the radius cr_q is the distance from q to its furthest nearest-neighbor candidate.
- Each non-discarded object reminds its own lower bound of the distance from itself to the query. For each node its initial lower bound is 0.

Note that, each time we find and object u such that $d(u, q) < cr_q$, we replace the farthest nearest-neighbor candidate by u, so this can reduce cr_q. Note also that, if $d(u, q) < cr_q$ it is likely that some of the neighbors of u can also be relevant to the query, so we check all the u neighbors. However, since the initial radius is ∞ we change a bit the navigational schema. In this case, instead of propagating in the neighborhood, we start the navigation from the node u towards the query q by jumping from one node to another if the next node is closer to q to than the previous one. Figure 3 illustrates this. In the figure, we start in p, and we navigate towards q until we reach p_c.

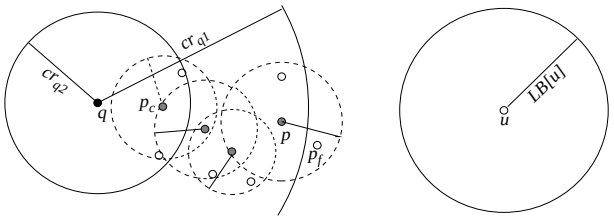


Fig. 3. If we find an object $p \in NN_k(q)_d$ we traverse through the graph towards q. Later, as cr_q decreases, it is possible to discard the node u when $LB[u] \geq cr_q$.

On the other hand, unlike range queries, we split the discarding process in two stages. In the first, we compute the lower bound of all the non-discarded nodes. In the second, we extract the objects such that their lower bound are big enough, that is, we discard u if $LB[u] > cr_q$. This is also illustrated in Figure 3. Note that, when we start in p the covering radius is cr_{q1}. However, upon we reach p_c the covering radius has been reduced to $cr_{q2} < LB[u]$, so we discard u.

$LB[u]$ is computed as the $\max_p\{d(p,q) - d_G(p,u)\}$, where p is any of the previously selected nodes. Note that $LB[u]$ allows us to delay the discarding of u until cr_q is small enough, even if we only update $LB[u]$ once.

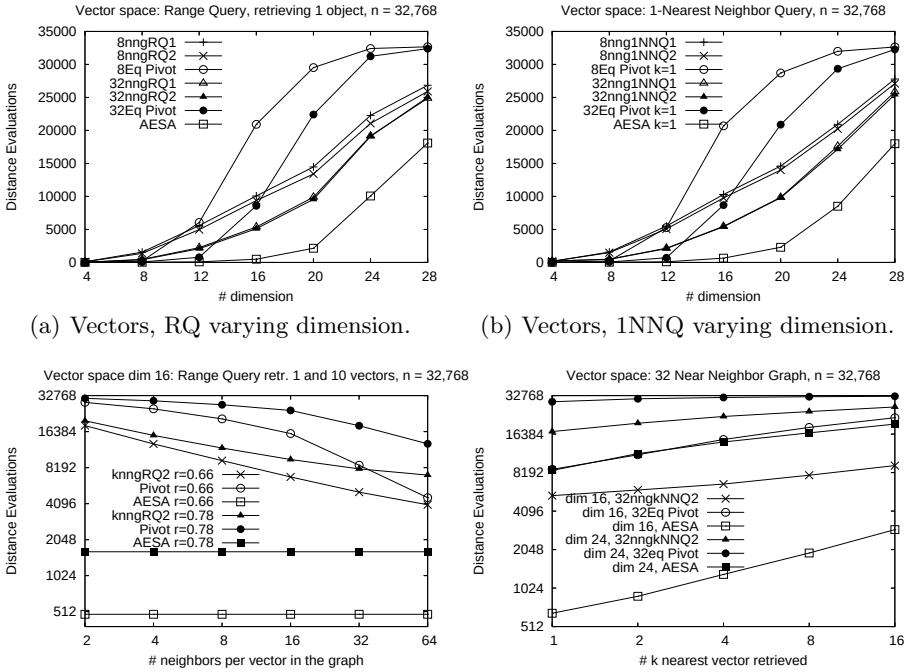
With these modifications we produce the algorithms kNNGk**NNQ1** which selects the next node according to Eq. (1), and kNNGk**NNQ2** which selects nodes far apart from each other.

4 Experimental results

We have tested our algorithms on synthetic and real-world metric spaces. The synthetic set consists of 32,768 points distributed uniformly in the D-dimensional unitary cube $[0,1]^D$, under the Euclidean distance. This space allows us to measure the effect of the space dimension D on our algorithms. Of course, we have not used the coordinates for discarding purposes, but just treated the points as abstract objects in an unknown metric space.

We also included two real-world examples. The first is a string metric space using the edit distance (the minimum number of character insertions, deletions and replacements needed to make two strings equal). The strings came from an English dictionary, where we index a random subset of 65,536 words. The second is a document metric space of 25,000 objects under the cosine distance. Both spaces are of interest in Information Retrieval applications.

Each point in the plots represents the average of 50 queries $q \in \mathbb{X} - \mathbb{U}$. For shortness we have called RQ the range query and NNQ the nearest neighbor query. We have compared our algorithms against AESA and a pivot-based algorithm (only in this case have we used range-optimal NNQs). For a fair comparison, we provided the same amount of memory for the pivot index and for our kNNG index (that is, we compare a kNNG index against a $1.5k$ pivot set size).



(a) Vectors, RQ varying dimension.

(b) Vectors, 1NNQ varying dimension.

(c) Vectors, RQ varying index size.

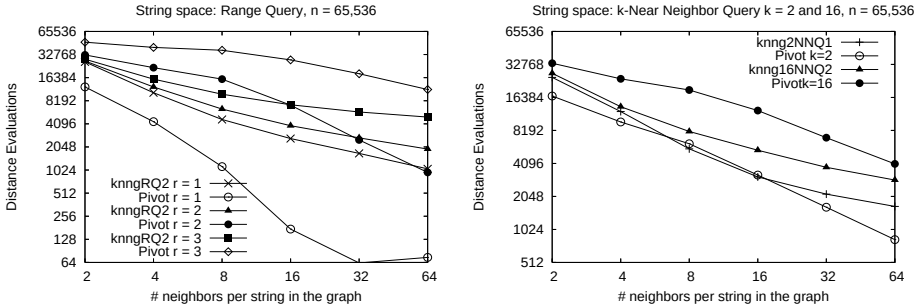
(d) Vectors, NNQ in dim 16 and 24.

Fig. 4. Distance evaluations in the vector space for RQ (left) and NNQ (right).

With our experiments we tried to measure the behavior of our technique varying the vector space dimension, the query outcome size (by using different radii in RQs or different number of retrieved neighbors in NNQs), and the graph size (that is, number of neighbors per object) to try different index size.

Figure 4 shows results in the vector space. Figure 4(a) shows RQs using radii that retrieve 1 object per query in average indexing the space with 8NNG and 32NNG graphs versus the dimension; and the Figure 4(b) shows the equivalent experiment for NNQs retrieving 1 neighbor. As can be seen from these plots, even though our NNQ algorithms are not range-optimal *per se*, they behave as if they were. Due to both RQ kNNG based variants behave very similar, we only show the better of them in the following plots in order to simplify the reading. We do the same in the NNQ plots.

Figure 4(c) shows RQs retrieving 1 and 10 vector in average per query versus the index size. Figure 4(d) shows NNQs over a 32NNG in dimension 16 and 24, versus the size of the query outcome. It is very remarkable that kNNG based algorithms are more resistant to both the dimension effect (Figures 4(a) and 4(b)) and the query outcome size (Figures 4(c) and 4(d)). As we can expect, the bigger the index size (that is, the more the neighbors in the kNNG), the better the searching performance (Figure 4(c)). Furthermore, all the plots in Figure 4



(a) Strings, RQ for $r = 1$, 2 and 3. (b) Strings, kNNQ for $k = 2$ and 16.

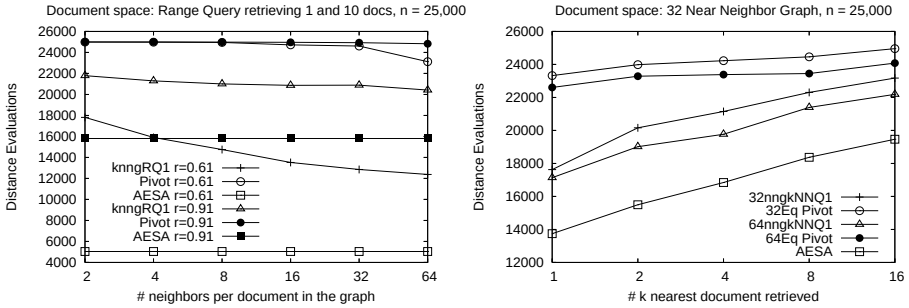
Fig. 5. Distance evaluations in the string space for RQ (left) and NNQ (right). In RQs, AESA needs 25, 106 and 713 distance evaluations for radii $r = 1$, 2 and 3 respectively. In NNQs, AESA needs 42 and 147 evaluations to retrieve 2 and 16 neighbors respectively.

show that our algorithms have better performance than the classic pivot based approach for medium and high dimension metric spaces, that is $D > 8$.

Figure 5 shows results in the string space. Figure 5(a) shows RQs using radii $r = 1$, 2, and 3, and Figure 5(b) shows NNQs retrieving 2 and 16 nearest neighbors, both of them versus the index size. They confirm that kNNG based search algorithms are resistant against the query result size, as expected from the synthetic space experiments. With radii $r = 1$, 2 and 3, we retrieve approximately 2, 29 and 244 strings per query in average, however the performance of our algorithms do not degrade so strongly as the pivot-based one. With radius 1 the pivot based technique has better performance than our algorithms. However, with radius $r = 2$ and 3, our algorithms outperform the pivot-based algorithm. In this figures, we do not plot the AESA results because it uses too few distances evaluations, however recall that the AESA index uses $O(n^2)$ memory which is impractical in most of the scenarios. Note that the difference between pivot range queries of radius 1 and the 2-nearest neighbor queries appears because there are strings that have much more than 2 neighbors at distance 1, for example the query word "cams" retrieves "jams", "crams", "cam" and seventeen others, so these words distort the average for radius 1. We also verify that, the bigger the index size, the better the performance.

Figure 6 shows results in the document space. Figure 6(a) shows RQs using radii $r = 0.61$ and 0.91 versus the index size. Figure 6(b) shows NNQs over a 32NNG versus the query outcome size. This space is particularly difficult to manage, please observe that the pivot-based algorithms check almost all the database. Even in this difficult scenario, our algorithms handle to retrieve object checking a fraction of the database. It is remarkable that in NNQs, our algorithms perform 30% more distance evaluation than AESA using only a 0.25% of its space requirement.

Note that in the three spaces, the grater the kNNG index size, the better the behavior of our algorithms. However, the search performance improves strongly



(a) Documents, RQ for $r = 0.61$ and 0.91. (b) Documents, kNNQ over a 32NNG.

Fig. 6. Distance evaluations in the document space for RQ (left) and NNQ (right).

as we add more space to the graph only when we use small indices, that is, kNNG graphs with few neighbors. Fortunately, our algorithms behave better than the classic pivot technique in low memory scenarios with medium or high dimensionality. According to our experiments for $k \leq 32$ we obtain better results than the equivalent memory space pivot-based algorithm in D-dimensional vector spaces of $D > 8$ and the document space. In the string space we obtain better results in RQ using radii $r > 2$ or in NNQ retrieving more than 4 nearest strings.

5 Conclusions

We have presented four metric space searching algorithms that use the k-nearest neighbor graph kNNG as a metric index.

Our algorithms have practical applicability in low memory scenarios for metric spaces of medium or high dimensionality. For instance, in the document metric space with cosine distance our nearest neighbor algorithm uses just 30% more distance computations than AESA only using a 0.25% of its space requirement. In same space, the pivot-based technique is completely useless.

The future work involves the development of range-optimal nearest neighbor queries and the researching of kNNG optimizations tuned for our metric applications. For instance, we want to explore other local graphs, like the *all range r graph* where we assign to each node all the nodes within distance r. This way also allow us to control the size of the neighbor ball.

Since our data structure can efficiently search for nearest neighbor queries, it is natural to explore an incremental construction of the graph itself. To do this end we need to solve *reverse* nearest neighbor problem with this data structure. Incremental construction is very realistic in many real-world applications.

References

1. 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.

2. R. Baeza-Yates, C. Hurtado, and M. Mendoza. Query recommendation usign query logs in search engines. In *Proc. EDBT Workshops'04*, LNCS 3268, pages 588–596, 2004.
3. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
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 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.
6. E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.
7. K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.
8. R. O. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.
9. D. Eppstein and J. Erickson. Iterated nearest neighbors and finding minimal polytopes. *Discrete & Computational Geometry*, 11:321–350, 1994.
10. K. Figueroa. An efficient algorithm to all k nearest neighbor problem in metric spaces. Master's thesis, Universidad Michoacana, Mexico, 2000. In Spanish.
11. G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report TR 4199, Dept. of Comp. Sci. Univ. of Maryland, Nov 2000.
12. G. Navarro and R. Paredes. Practical construction of metric t-spanners. In *Proc. ALENEX'03*, pages 69–81, 2003.
13. 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.
14. R. Paredes and G. Navarro. Practical construction of k nearest neighbor graphs in metric spaces. Technical Report TR/DCC-2005-6, Dept. of Comp. Sci. Univ. of Chile, May 2005. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/knnconstr.ps.gz`.
15. E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.