Speeding Up Permutation Based Indexing with Indexing

Karina Figueroa
Facultad de Ciencias Físico-Matemáticas
Universidad Michoacana
Mexico
Email: karina@fismat.umich.mx

Kimmo Fredriksson
Department of Computer Science
University of Kuopio
Finland
Email: kimmo.fredriksson@uku.fi

***Abstract*—A recent probabilistic approach for searching in high dimensional metric spaces is based on predicting the distances between database elements according to how they order their distances towards some set of distinguished elements, called permutants. In the preprocessing phase a set of permutants is chosen, and are sorted (permuted) by their distances against every database element. The permutations form the index. When a query is given, its corresponding permutation is computed, and — as similar elements will (probably) have a similar permutation — the database is compared in the order induced by the similarity between permutations. This works well but has relatively high CPU time due to computing the distances between permutations and (partially) sorting the database by the similarity. We improve this by identifying and solving this as another metric space problem. This avoids many distance computations between the permutants. The experimental results show that this works extremely well in practice.**

***Keywords*-metric space indexing; probabilistic algorithms; indexing permutations;**

I. Introduction

Proximity searching has a vast number of applications in numerous fields, such as audio, image and document databases, image quantization and compression, computational biology, and data mining. The common denominator between all of them is that we usually have a database of objects and a metric distance function defined between any two objects. Metric space indexing then means preprocessing the database so that subsequent queries can be efficiently answered without comparing the query against the whole database. The two fundamental types of queries are finding either the k nearest neighbors to the query element (k-NN query), or the set of elements within a fixed distance (r) to the query element (range query).

Some of the algorithms are very effective in low dimensional spaces. Unfortunately their performance become worse as the *intrinsic* dimension increases, and at certain point they degenerate to a linear scan over the database. The intrinsic dimension can be defined in many ways, but informally we can define it as the smallest dimensional vector space where the database elements can be mapped without distorting the pair-wise distances much. The problem of tackling with high dimensional spaces is so hard, that it is often called *the curse of dimensionality*. The hardness comes from the fact that the histogram of distances become very concentrated on high dimensions, or in other words, all the pair-wise distances between the database elements are almost equal, and thus contain little information.

The most promising, and currently the only one that is useful, technique for high dimensional spaces is to resort to *inexact searching*. The idea is to design a method that has a guarantee that the quality of the answer is within certain bounds (*approximate searching*) or with high probability retrieves most of the relevant elements (*probabilistic searching*), for the given query. I.e. these methods may miss some of the relevant elements, or report some that are not relevant. Fortunately, the answer is usually good enough for many applications. The common aspect of these algorithms is that they usually trade time for the quality of the answer.

In [1] a novel probabilistic technique was presented for proximity searching in metric spaces. This method has extremely good performance in high dimensional spaces. However, it makes sequential scan in a certain part of the searching process. In this paper we improve this subprocess by identifying and modelling the problem as a new metric space and another proximity searching problem.

The paper is organized as follows. In Sec. II we give the basic definitions and review some of the previous work in general, and in Sec. III the algorithm we are going to improve in particular. Our improvement is described in Sec. IV, and experimental results are given in Sec. V. Finally, Sec. VI concludes.

II. Definitions and Related Work

A metric space is a pair $(\mathbb{X}, d)$ where $\mathbb{X}$ is the universe of valid objects with a distance function $d : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$, satisfying: symmetry $d(x,y) = d(y,x)$, reflexivity $d(x,x) = 0$, strict positiveness $d(x,y) > 0 \iff x \neq y$, and the triangle inequality $d(x,y) \leq d(x,z) + d(z,y), \forall x,y,z \in \mathbb{X}$. The database is a finite subset of the valid objects $\mathbb{U} \subseteq \mathbb{X}$. Range query can then be given as a pair (q,r), where $q \in \mathbb{X}$, and the query retrieves the set $\{u \in \mathbb{U} \mid d(q,u) \leq r\}$. Similarly, k-NN query is given as (q,k), where k is a positive integer, and the query retrieves a set $S = \{u_1, \ldots, u_k\}$, $S \subset \mathbb{U}$, such that $\forall i, v \in \mathbb{U} \setminus S : d(q,u_i) \leq d(q,v)$.

Proximity searching in metric spaces usually are solved in two stages: preprocessing and query time. During the preprocessing an index is build and it is used during query time to avoid distance computations. Basically the state of the art in this area can be divided in two families [2]: pivot based algorithms and local partitioning (e.g. clustering) based algorithms.

A. Pivot Based Algorithms

In this family the index consists in a set of *pivots* $\{p_1, \ldots, p_k\} \subseteq \mathbb{U}$, which compute and keep (in a data structure, like tree usually) some (or all) distances

$$\{d(p_1, u), d(p_2, u), \ldots, d(p_k, u)\}, u \in \mathbb{U}.$$

Range query can then be solved by first computing $d(q, p_i)$, and then by the triangle inequality filter out every database element u that satisfies $|d(q, p_i) - d(u, p_i)| > r$. Many variations of this basic scheme exist, having different space and time trade-offs. Some representatives are [3], [4], [5], [6].

B. Compact Partition Based Algorithms

In this family the space is divided into compact zones (as small as possible). A set of objects (centers) $\{p_1, \ldots, p_k\} \subseteq \mathbb{U}$ are chosen and the rest are distributed into the k zones defined in various ways by the centers p_i. The index is formed by the centers, those elements in each zone and sometimes additional information about distances. Two basic variants exist. Each center p_i may have a corresponding "covering radius" r_i, and all database elements u belonging to the group i defined by the center p_i have $d(u, p_i) \le r_i$. Thus the range query can discard all elements in the group i if $d(q, p_i) > r + r_i$. Alternatively, every database element can be defined to belong to the group i if p_i is the closest center. Again, given that p_i is the closest center to q, we can discard the whole group j if $d(q, p_j) - r > d(q, p_i) + r$. Some examples are [7], [8], [9].

C. Approximate and Probabilistic Algorithms

In approximate algorithms one usually has a parameter ε, so that the retrieved elements are guaranteed to have a distance to the query at most $(1 + \varepsilon)$ times of what was asked for. This relaxation gives faster algorithms when ε increases [10], [11]. Probabilistic algorithms on the other hand state that the answer is correct with high probability. Some examples are [12], [13]. It is also possible to turn any exact algorithm into probabilistic one [14]. A recent probabilistic method is based on sorting the database in a particular order, and then traversing only a small fraction of the database in that order [1]. We cover this in detail in the next Section.

III. Permutation Based Algorithm

Let us define a subset $\mathbb{P} = \{p_1, p_2, \ldots, p_k\}$ (called permutants) of the database $\mathbb{U}$. Every element of the database sorts the permutants according to the distances to them, thus forming a permutation of $\mathbb{P}$. More formally, for an element $u \in \mathbb{X}$, its permutation Π_u of $\mathbb{P}$ satisfies $d(u, \Pi_u(i)) \le d(u, \Pi_u(i+1))$, where the elements at the same distance are taken in arbitrary but consistent order. We use $\Pi_u^{-1}(p_i)$ for the *rank* of an element p_i in the permutation Π_u. Note that two equal elements must have the same permutation, while two similar elements will (hopefully) have a similar permutation.

Basically, the permutation based algorithm is used to predict proximity between elements. The algorithm is very simple: In the preprocessing (offline) phase the permutation for every element in the database is computed. These are stored and form the index. When a query is given, its permutation is computed. The elements in the database are sorted by similarity between permutations and then compared against the query in this order, until some stopping criterion is met.

Similarity between two permutations can be measured for example by Kendall Tau, Spearman Rho, or Spearman Footrule metric [15]. All of them are metric. We use Spearman Rho because it is not expensive to compute and according to the authors in [1] it has a good performance to predict proximity between elements. (The square of the) Spearman Rho metric is defined as follows:

$$S_\rho(u, q) = S_\rho(\Pi_u, \Pi_q) = \sum_{i=1}^{k} |\Pi_u^{-1}(p_i) - \Pi_q^{-1}(p_i)|^2 \quad (1)$$

For example, let $\Pi_q = p_1, p_2, p_3, p_4, p_5$ be the permutation of the query, and $\Pi_u = p_3, p_2, p_1, p_5, p_4$ the permutation of an element $u \in \mathbb{U}$. According to (1), we have $S_\rho(\Pi_q, \Pi_u) = |1-3|^2 + |2-2|^2 + |3-1|^2 + |4-5|^2 + |5-4|^2 = 10$. This distance can be computed in $O(k)$ time [15]. Thus, during preprocessing stage we first compute the kn distances $d(u, p_i)$, and then compute and sort all the permutations for every element in the database. This costs $O(nk \log k)$ additional time, and requires $O(nk \log k)$ bits of space for the index.

At query time we first compute $d(q, p_i)$ for every $p_i \in \mathbb{P}$, compute the permutation Π_q, and sort $\mathbb{U}$ into increasing order according to $S_\rho(\Pi_u, \Pi_q)$ (the sorting can be done incrementally, as only some of the first elements are needed). Then $\mathbb{U}$ is traversed in that sorted order, evaluating $d(q, u)$ for each $u \in \mathbb{U}$. For range queries each u that satisfies $d(q, u) \le r$ is reported, and for k-NN queries the set of k smallest distances so far (and the corresponding elements) are maintained. The database traversal is stopped at some point, and the rest of the database elements are just ignored. This makes the algorithm probabilistic, as even if $S_\rho(q, u) < S_\rho(q, v)$ it does not guarantee that $d(q, u) < d(q, v)$, and our

Algorithm 1 Sequential-Range-Query(q, r, f)

```
1: INPUT: $q$ is a query and $r$ its radius, $f$ is the fraction of the database to traverse.
2: OUTPUT: Reports a subset of those $u \in \mathbb{U}$ that are at distance at most $r$ to $q$.
3: Let $A[1, n]$ be an array of tuples and $\mathbb{U} = \{u_1, \dots, u_n\}$
4: Compute $\Pi_q^{-1}$
5: for $i \leftarrow 1$ to $n$ do
6:     $A[i] \leftarrow \langle u_i, S_\rho(\Pi_{u_i}, \Pi_q) \rangle$
7: end for
8: SortIncreasing(A) // by second component of tuples
9: for $i \leftarrow 1$ to $f \cdot n$ do
10:     Let $\langle u, s \rangle = A[i]$
11:     if $d(q, u) \leq r$ then
12:         Report $u$
13:     end if
14: end for
```

Algorithm 2 Indexed-Range-Query(q, r, f)

```
1: INPUT: $q$ is a query and $r$ its radius, $f$ is the fraction of the database to traverse.
2: OUTPUT: Reports a subset of those $u \in \mathbb{U}$ that are at distance at most $r$ to $q$.
3: Let $A[1, f \cdot n]$ be an array of elements and $W$ an index of permutations
4: Compute $\Pi_q^{-1}$
5: $A \leftarrow$ NearestNeighbors($W, \Pi_q, f \cdot n$)
6: for $i \leftarrow 1$ to $f \cdot n$ do
7:     if $d(q, A[i]) \leq r$ then
8:         Report $u$
9:     end if
10: end for
```

stopping criterion may halt the search prematurely. On the hand, if the order induced by $S_\rho(q, u)$ is close to the order induced by the real distances $d(q, u)$, the algorithm performs well.

Algorithm 1 shows pseudo code for a range query (we will call this Sequential-Query algorithm), assuming that we are ready to traverse a fraction f of the whole database (i.e. fn elements). The efficiency and the quality of the answer obviously depend on f. A way to obtain good values for f is discussed in [1]. A naïve strategy for k-NN search could be to simply use $f = k/n$, and simply take the k first elements in the permutation order to be the (probabilistic) k nearest neighbors. The search makes $k + fn$ distance evaluations (using the original distance function $d(\cdot)$), i.e. first k evaluations to compute the query permutation, and then fn for the fn candidates. The extra CPU time for computing the S_ρ values and sorting them is $O(kn + n \log n)$, or just $O(kn + fn \log n)$ if the sorting is done incrementally, or by using a priority queue. Our goal is to improve this CPU complexity.

IV. Indexing Permutations

Permutation based algorithm is an excellent predictor of proximity, but the main problem is how to retrieve the most similar permutations quickly. If the fn elements need not to be traversed in order, we could simply select the fn smallest elements (in some order) using any $O(n)$ worst case selection algorithm. This improves the $O(kn + fn \log n)$ time to $O(kn + n) = O(kn)$, by just changing the line 8 in Algorithm 1. However, we want to do better than that. The real bottle-neck is the lines 5–7, which computes all the n Spearman ρ distances, taking $O(kn)$ time, while we would like to compute only about $O(fn)$ distances, i.e. only those that are most similar to the query permutation. This is because the algorithm never uses the $(1 - f)n$ largest distances (i.e. the values $A[fn + 1 \dots n]$ in Algorithm 1).

Our proposal is to consider the permutations of the database and the metric between permutations as a new metric space. In this new space we can use any (exact or inexact) algorithm for metric spaces to retrieve the most similar permutations avoiding to compare the whole permutations against the query. That is, in the preprocessing phase we build another index for the permutations, and then replace the lines 5–8 in Algorithm 1 with a nearest neighbor query that retrieves (and stores in A) the fn nearest permutations to Π_q. In the worst case this new algorithm makes the same n Spearman ρ distance evaluations as the base-line algorithm, costing $O(kn)$ time in total, but if the index is good, we can hope to improve this on average (in the best case our approach takes only $O(kfn)$ time to compute the relevant Spearman ρ distances). Algorithm 2 shows the generic pseudo code for the new range query algorithm. We will call this Indexed-Query algorithm. What remains is to fix the algorithm for the nearest neighbor search. For this we can use any exact indexing method, to compute exactly the same output as Sequential-Query algorithm. Alternatively, we can use the permutation based algorithm again, recursively. This latter approach introduces another probabilistic layer, and the output can differ from the basic method. However, this may be more efficient, as the intrinsic dimensionality of the permutations can be high. However, for this alternative one needs to use smaller set of permutants for the second index to be able to speed up the search.

In general, if we fix the number of permutants, the intrinsic dimension of the permutations grows as the dimension of the underlying real database grows, but is about a constant when the size of the database grows. Note however that the real distance prediction accuracy grows as the number of permutants is increased [1].

The growth of the intrinsic dimension of the permutations is shown in Figure 1, as a function of the real dimension of

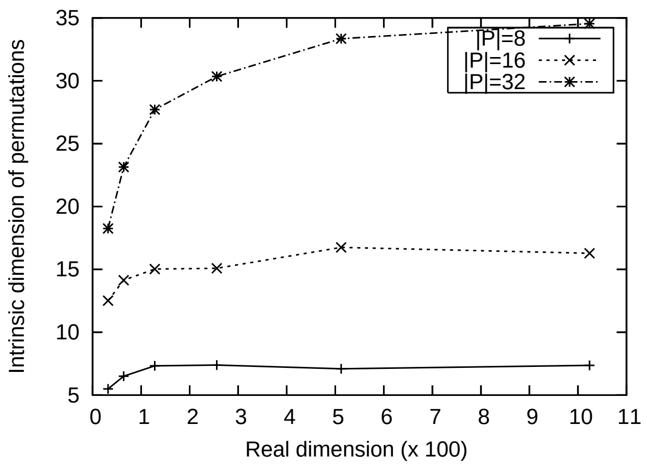

Figure 1. The intrinsic dimension as the function of the real dimension of the underlying database for $|\mathbb{P}| \in \{8, 16, 32\}$. The size of the database is 16,000 elements.

the database. The databases used were uniformly distributed sets in the unitary cube.

A. Comparison

If we use an exact metric space search algorithm as the second index (for permutations), then Sequential-Query algorithm and Indexed-Query algorithm compute exactly the same results, and make exactly the same distance evaluations in the original space, i.e. $k+fn$ of them. This cost obviously depends on the underlying distance function $d(\cdot)$. Let this cost be $O(c(k+fn))$, for some function c. The difference is that Indexed-Query algorithm computes less Spearman ρ distances in the permutation space. For the original Sequential-Query algorithm the cost of this is $O(kn + fn \log n)$, while for our Indexed-Query algorithm this can be substantially less, depending on how good our second index is (it can be e.g. $O(kn^{\alpha})$, for some $\alpha < 1$, but at least $O(kfn)$). Thus we can state that:

- if the distance function $d(\cdot)$ is very costly to compute, then $O(c(k+fn))$ may dominate $O(kn+fn \log n)$ (in Sequential-Query algorithm), and our improvement is negligible;
- if $d(\cdot)$ is cheap to compute, and/or if f is relatively small, then the Spearman ρ computations dominate Sequential-Query algorithm, and we can expect good speed-up by using Indexed-Query algorithm.

On the other hand, Sequential-Query algorithm is useful when $d(\cdot)$ is very costly to compute (as compared to the Spearman distances), and the dimensionality of the original space is very high, as this means that exact indexing methods degenerate to linear search. Hence, as Indexed-Query algorithm is cheaper w.r.t. computing the Spearman distances, we might add:

- Indexed-Query algorithm may be useful and competitive for lower dimensions as Sequential-Query algorithm.

V. Experimental Results

We have implemented our proposal and in the following report the performance for some artificial and real spaces. Experiments were ran on an Intel Core 2 Duo workstation with 2.2 GHz CPU and 4GB, 667 MHz of RAM with Mac OS X, version 10.5.5.

A. Setting and Rationale

As the performance and the accuracy of the base-line method Sequential-Query algorithm is well established in [16], and our method computes *exactly* the same results (assuming we use an exact method for the second index), we do not compare it against the competitors. Rather, we consentrate on our improvement, i.e. basically how to efficiently implement the line 5 of Indexed-Query algorithm. We tested our method for the k nearest neighbors problem. Note that for relatively small values of k (and f) and small dimensions, the line 5 of Indexed-Query algorithm, and especially lines 5–8 of Sequential-Query algorithm, dominate the total time in practice. However, in some cases the distance function for the actual database elements can be extremely costly to compute, making the cost of retrieving the most similar permutations negligible. Obviously this is not the domain of our interest.

B. Implementation

In our experiments line 8 of Sequential-Query algorithm is implemented as QuickSort for simplicity. In the original work [16] they used incremental sorting, which achieves better time but the practical difference is negligible for all but very small number of permutants, as the lines 5–7 dominate the time. This baseline algorithm is labeled as **sequential** in the plots.

To implement the line 5 in Indexed-Query algorithm, we tested the following variants:

bkt

Burkhard-Keller Tree [3], generalized for continuous distances using fixed width rings. We used bucket size 12 for the leaves, and ring size 3.

ght

Generalized Hyperplane Tree [7]. The original method used binary trees, while the implementation we used is generalized to arbitrary arity; we used the same arity as the dimension of the space.

mvp

Multi Vantage-Point tree [17], [6]. We used bucket size 12 and the tree arity was the same as the metric space dimension was.

sat

Spatial Approximation Tree [9].

lcluster

List of Clusters [18], with bucket size 12.

Unless otherwise stated, we used the same amount of permutants as was the dimension of the real space; for example, in dimension 8 we used 8 permutants, in dimension 12 we used 12 permutants, and so on.

The implementations of **bkt, ght, mvp, sat** and **lcluster** were taken from the SISAP library[1]. We note that the parameter values were not optimized for each problem instance, but simply used fixed values, in general trying to use the values suggested by the original papers. As is shown shortly, even this naïve approach leads to good results, but nevertheless there is probably room for improvement, especially in the cases where the number of permutants is high.

We note that all the indexes take asymptotically the same (linear) space, but the constant factors can vary, and in particular the original index is the most space efficient.

C. Experiments

Our first set of experiments were done using uniformly distributed point sets in the unitary cube, from 4 to 64 dimensions, under Euclidean distance. This allows us to precisely control the dimensionality of the space. We searched the 5-NN, using simply fractions of the form $f = k/n$ (i.e. f varies as the database size varies). Figure 2 shows the performance of using other similarity searching algorithm for retrieving the most similar permutations under this setting. Basically, it shows how much time we use to solve the lines 5–8 of Sequential-Query algorithm and line 5 of Indexed-Query algorithm. This by far dominates the overall time. (However, the negligible effect of including the distance evalutions in the original space to the total time is explicitly shown in our subsequent experiments on real databases.) The x-axis represents the dimension of the database examined, and the y-axis is the time per query (average). We used 500 queries for each point in the plot, and report the average time. In general, using **lcluster** as the second index is the best alternative, beating the baseline method by a large margin. For dimensions lower than about 20, all the exact algorithms perform very well as compared to the baseline method. This is especially interesting, as the original method was advocated mainly for large dimensions, or for spaces having otherwise costly distance evaluations, so that the relatively large CPU cost of computing the permutation distances could be amortized.

Our second experiment, reported in Figures 3 and 4 show the times used when the database size increases, for dimensions 16 and 128, correspondingly. Again, the number of permutants was the same as the dimension. For dimension 16, the results are very good, showing that using other metric space indexing algorithm (as second index) improve the

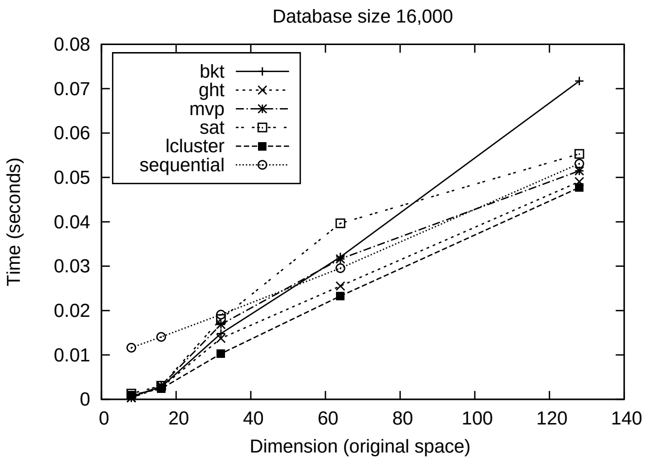

Figure 2. The search times using 16,000 elements. The amount of permutants was the same as the dimension (dimension 8, permutants 8 and so on).

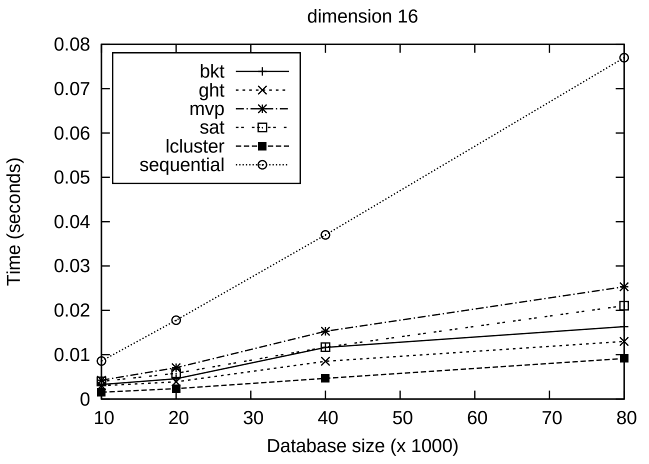

Figure 3. The search times for different database sizes (in thousands) in dimension 16.

time used by permutations-based algorithm. However, the relative difference against the baseline algorithm diminishes for larger dimensions, as seen in Figure 4. We note that we could trade accuracy for time, by using smaller number of permutants even for high dimensional spaces. This obviously works for the baseline algorithm as well, but our method benefits more on this.

Figures 5 and 6 show the times for k nearest neighbors queries for varying k. Figure 5 is for uniformly distributed data in unitary cubes in dimensions 16 and 32. Figure 6 is for 1024 dimensional Gaussian space. The data was generated with multivariate Gaussian distribution having 32 clusters, with variance of 0.09 between the center distribution. Again, using a second index to retrieve the similar permutations gives a significant speed-up. This is most visible for relatively small dimensions and small number of permutants. Our method is slowed down more by retrieving more of

[1]http://sisap.org/Metric_Space_Library.html

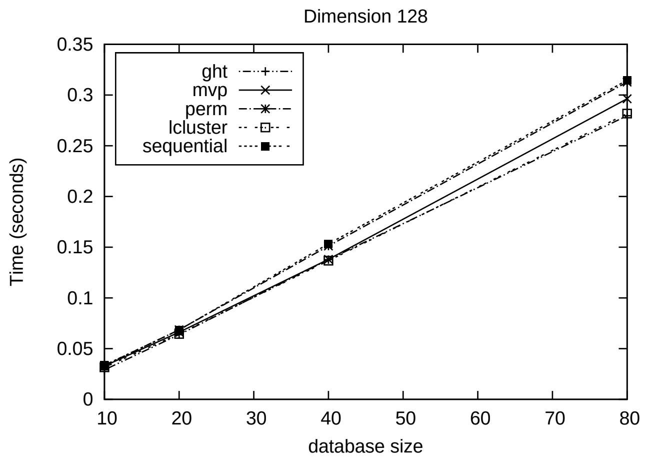

Figure 4. The search times for different database sizes (in thousands) in dimension 128.

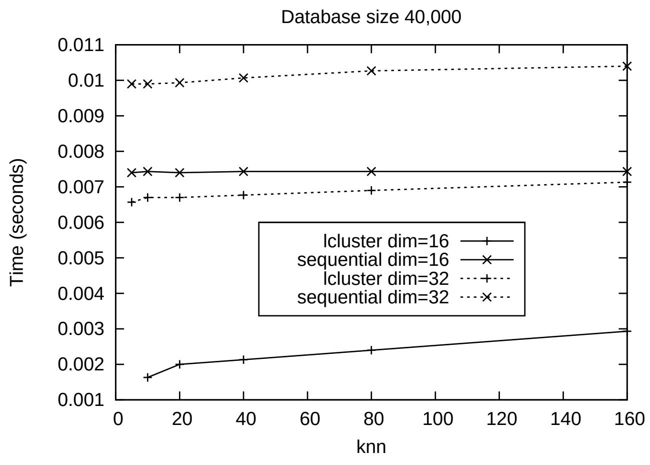

Figure 5. Search times for kNN queries, for different values of k, in dimensions 16 and 32, and 40,000 uniformly distributed elements in unitary cube.

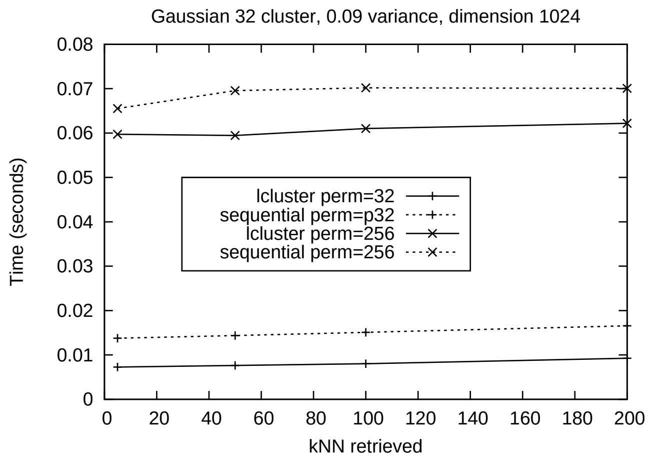

Figure 6. Search times for kNN queries, for different values of k and Gaussian data. The plots are for different number of permutants. The database size was 10,000 elements.

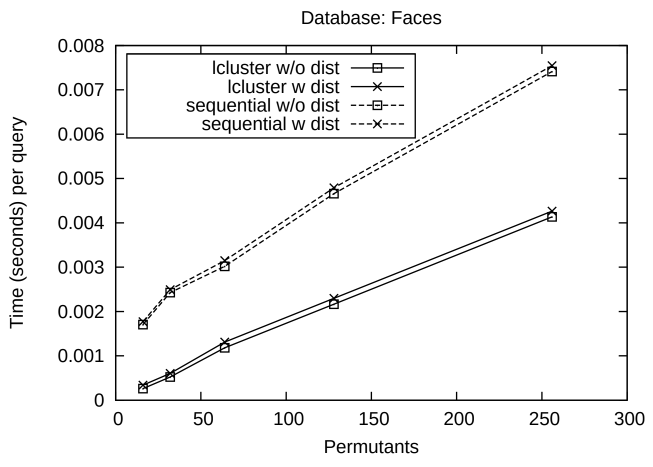

Figure 7. The search times for a real database (FERET). The x-axis is the number of permutants. The times are shown for with and without the distance computations in the real space.

the nearest neighbors, which is clear as for the baseline algorithm the time to retrieve the similar permutations is constant (as the function of k). Even so, the slow-down is quite mild for the k values shown.

We experimented with a real database using Euclidean distance. This database (FERET [19]^2) is composed of vectors of features extracted from real (face) images. The vectors have 761 components. The database size was a set of 762 faces, plus another (distinct) set of 254 faces used for queries. Figure 7 shows the time used to retrieve 5 nearest neighbors for different numbers of permutants. Again, using **lcluster** as the second index is the best alternative. Even for 256 permutants our approach is about two times faster than the original algorithm. This gap increases slightly for decreasing number of permutants. The times are shown for the the whole algorithm (including the distance evaluations in the original space), and only for retrieving the most similar permutation (i.e. without the distance evaluations in the real space). Notice that most of the time used was spent on retrieving the k most similar permutations, and the rest has negligible cost as claimed in this paper.

Finally, we experimented with strings (Spanish dictionary) using edit distance. This database is composed of 86061 words in Spanish. Figure 8 shows the time used to retrieve different number nearest neighbors using 32 permutants. We again used **lcluster** as the second index. Again, in this plot we show the time used for retrieving the k most similar permutations, without the edit distance computations (*w/o dist* in the plots), and also the total time (*w dist* in the plots). Again most of the main time was spent on retrieving the k most similar permutations.

^{2}http://sisap.org/Metric_Space_Library.html

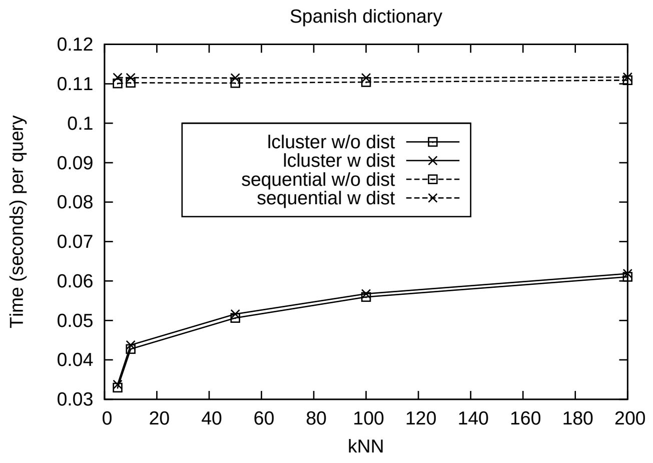

Figure 8. The search times for Spanish dictionary. The x-axis is the number kNN retrieved. The times are shown for with and without the edit distance computations.

D. Discussion

In our experiments we have not cared about the accuracy of the results (i.e. the accuracy of the retrieved k nearest neighbors from the actual database; the second level indexes are exact of course). Basically two things affect this; the number of permutants, and the fraction of the original database scanned. These are totally orthogonal to our technique. We simply used the same amount of permutants as the real dimension was. Nevertheless, this shows the general applicability of our method. In particular, our relative speed-up increases as the dimension or the number of permutants decreases. For moderate to high dimensions with small number of permutants our approach improves upon the original algorithm. For very small number of permutants (“low dimensions”, say, less than 20) our improvement is even better, but in this case the competition is also greater. Thus more work is needed to evaluate the speed and accuracy against traditional approaches that can be efficient as well for relatively low dimensions. The original algorithm is basically useful only for large dimensions, or if the distance function is otherwise costly to evaluate.

Also, for large number of permutants, one could trade space for time, by using e.g. AESA [5] or some more space efficient variants of it, such as [20], [21].

Another line of work is to apply some approximate or probabilistic algorithm for the second index. We ran some preliminary experiments with using the permutations based algorithm again. However, in order to reduce the search times, the number of permutants for the second layer must be substantially smaller than for the real database. This obviously reduces the accuracy of the method, and trying to compensate by scanning more of the database only flattens out any savings in the search time. However, these results are not yet conclusive.

VI. Conclusions

We have significantly reduced the CPU times of one of the best algorithms for searching in very high dimensional metric spaces. We were able to do this by identifying another metric space search problem in the algorithms internals. This search problem can be solved using any other existing algorithm for metric space similarity search. Our experimental results show that the approach is very effective in practice.

Acknowledgment

We wish to thank the anonymous reviewers for many helpful comments This work was partially supported by CONACyT (Mexico).

References

[1] E. Chávez, K. Figueroa, and G. Navarro, “Proximity searching in high dimensional spaces with a proximity preserving order,” in *MICAI 2005: Advances in Artificial Intelligence*, ser. LNCS, vol. 3789, 2005, pp. 405–414.

[2] E. Chávez, G. . Navarro, R. Baeza-Yates, and J. Marroquín, “Proximity searching in metric spaces,” *ACM Computing Surveys*, vol. 33(3), pp. 273–321, 2001.

[3] W. Burkhard and R. Keller, “Some approaches to best-match file searching,” *Comm. of the ACM*, vol. 16, no. 4, pp. 230–236, Apr. 1973.

[4] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu, “Proximity matching using fixed-queries tree,” in *Proceedings of 5th Conference on Combinatorial Pattern Matching (CPM’94)*, ser. LNCS 807. Springer–Verlag, 1994, pp. 198–212.

[5] E. Vidal, “An algorithm for finding nearest neighbors in (approximately) constant average time,” *Pattern Recognition Letters*, vol. 4, pp. 145–157, 1986.

[6] P. Yianilos, “Excluded middle vantage point forests for nearest neighbor search,” in *DIMACS Implementation Challenge, ALENEX’99*, 1999.

[7] J. Uhlmann, “Satisfying general proximity/similarity queries with metric trees,” *Information Processing Letters*, vol. 40, pp. 175–179, 1991.

[8] S. Brin, “Near neighbor search in large metric spaces,” in *Proceedings of 21st Conference on Very Large Databases (VLDB)*, 1995, pp. 574–584.

[9] G. . Navarro., “Searching in metric spaces by spatial approximation,” *The Very Large Databases Journal (VLDBJ)*, vol. 11(1), pp. 28–46, 2002.

[10] B. Bustos and G. . Navarro, “Probabilistic proximity search algorithms based on compact partitions,” *Journal of Discrete Algorithms (JDA)*, vol. 2, no. 1, pp. 115–134, 2003.

[11] K. Tokoro, K. Yamaguchi, and S. Masuda, “Improvements of tlaesa nearest neighbour search algorithm and extension to approximation search,” in *ACSC ’06: Proceedings of the 29th Australasian Computer Science Conference*, 2006, pp. 77–83.

[12] A. Singh, H. Ferhatosmanoglu, and A. Saman, “High dimensional reverse nearest neighbor queries,” in *CIKM International Conference on Information and Knowledge Management*. ACM, 2003, pp. 91–98.

[13] F. Moreno-Seco, L. Micó, and J. Oncina, “A modification of the LAESA algorithm for approximated k-nn clasification,” *Pattern Recognition Letters*, vol. 24, pp. 47–53, 2003.

[14] E. Chávez and G. . Navarro, “A probabilistic spell for the curse of dimensionality,” in *3rd Workshop on Algorithm Engineering and Experiments (ALENEX'01)*, ser. Lecture Notes in Computer Science, vol. 2153, 2001, pp. 147–160.

[15] R. Fagin, R. Kumar, and D. Sivakumar, “Comparing top k lists,” *SIAM J. Discrete Math*, vol. 17, no. 1, pp. 134–160, 2003.

[16] E. Chávez, K. Figueroa, and G. . Navarro, “Effective proximity retrieval by ordering permutations,” *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, vol. 30, no. 9, pp. 1647–1658, 2008.

[17] P. Yianilos, “Data structures and algorithms for nearest neighbor search in general metric spaces,” in *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, 1993, pp. 311–321.

[18] E. Chávez and G. Navarro, “A compact space decomposition for effective metric indexing,” *Pattern Recognition Letters*, vol. 26, no. 9, pp. 1363–1376, 2005.

[19] P. Phillips, H. Wechsler, J. Huang, and P. Rauss, “The FERET database and evaluation procedure for face recognition algorithms,” *Image and Vision Computing Journal*, vol. 16, no. 5, pp. 295–306, 1998.

[20] G. . Navarro, R. Paredes, and E. Chávez, “t-spanners as a data structure for metric space searching,” in *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, ser. Lecture Notes in Computer Science, vol. 2476. Springer, 2002, pp. 298–309.

[21] K. Figueroa and K. Fredriksson, “Simple space-time tradeoffs for AESA,” in *Proceedings of 6th Workshop on Efficient and Experimental Algorithms (WEA'07)*, ser. LNCS 4525. Springer–Verlag, 2007, pp. 229–241.