A Hybrid Data Structure for Searching in Metric Spaces *

Norma Herrera y Nora Reyes – {nherrera,nreyes}@unsl.edu.ar
Dpto. de Informática – Universidad Nacional de San Luis – Tel.: 02652-420822-257 - Fax: 02652-430224

Edgar Chávez – elchavez@fismat.umich.mx
Escuela de Ciencias Físico-Matemáticas Universidad Michoacana de San Nicolás de Hidalgo - México

1. Introduction and motivation

The concept of "approximate" searching has applications in a vast number of fields. Some examples are non-traditional databases (e.g. storing images, fingerprints or audio clips, where the concept of exact search is of no use and we search instead for similar objects), text searching, information retrieval, machine learning and classification, image quantization and compression, computational biology, and function prediction.

All those applications have some common characteristics. There is a universe U of *objects*, and a nonnegative *distance function* $d : U \times U \longrightarrow \mathbb{R}^+$ defined among them. This distance satisfies the three axioms that make the set a *metric space*: strict positiveness ($d(x, y) = 0 \Leftrightarrow x = y$), symmetry ($d(x, y) = d(y, x)$) and triangle inequality ($d(x, z) \leq d(x, y) + d(y, z)$). The smaller the distance between two objects, the more "similar" they are. We have a finite *database* $S \subseteq U$, which is a subset of the universe of objects and can be preprocessed (to build an index, for example). Later, given a new object from the universe (a *query* q), we must retrieve all similar elements found in the database. There are two typical queries of this kind:

Range query: Retrieve all elements within distance r to q in S. This is, $\{x \in S\ ,\ d(x, q) \leq r\}$.

Nearest neighbor query (k-NN): Retrieve the k closest elements to q in S. That is, a set $A \subseteq S$ such that $|A| = k$ and $\forall x \in A, y \in S - A, d(x, q) \leq d(y, q)$.

The distance is considered expensive to compute (think, for instance, in comparing two fingerprints). Hence, it is customary to define the complexity of the search as the number of distance evaluations performed, disregarding other components such as CPU time for side computations, and even I/O time. Given a database of $|S| = n$ objects, queries can be trivially answered by performing n distance evaluations. The goal is to structure the database such that we perform less distance evaluations.

A particular case of this problem arises when the space is a set of d-dimensional points and the distance belongs to the Minkowski L_p family: $L_p = (\sum_{1 \leq i \leq d} |x_i - y_i|^p)^{1/p}$. There are effective methods to search on d-dimensional spaces, such as kd-trees [1] or R-trees [4]. However, for roughly 20 dimensions or more those structures cease to work well. We focus our work in general metric spaces, although the solutions are well suited also for d-dimensional spaces.

It is interesting to notice that the concept of "dimensionality" can be translated to metric spaces as well: the typical feature in high dimensional spaces with L_p distances is that the probability distribution of distances among elements has a very concentrated histogram, making the work of any similarity search algorithm more difficult [2, 3]. We say that a general metric space is high dimensional when its histogram of distances is concentrated.

There are a number of methods to preprocess the set in order to reduce the number of distance evaluations. All those structures work on the basis of discarding elements using the triangle inequality, and most use the classical divide-and-conquer approach (which is not specific of metric space searching). Many of these indexes are based on pivots (Section 2).

*This work has been partially supported by RIBIDI Project CYTED VII.19.

The Dynamic Spatial Approximation Tree (*dsa-tree* for short) is a recently proposed data structure for searching in metric spaces[5], which is based on a novel concept: rather than dividing the search space, approach the query spatially, that is, start at some point in the space and get closer and closer to the query. It has been shown that the *dsa-tree* behaves better than the other existing structures on metric spaces of high dimension or queries with low selectivity, which is the case in many applications. The *dsa-tree*, have a parameter to be tuned by the user of each application, that is the maximum permitted arity. The rule of thumb is that low arities are good for low dimensions or small search radii. The main benefit of the *dsa-tree* is that it is dynamic. That is, once it is built, it is easy to add new elements to it. This makes the *dsa-tree* suitable for dynamic applications such as multimedia databases.

Pivoting algorithms use $(O(kn))$ space for k pivots. If we are able to use unbounded memory pivoting algorithms can beat any other data structure. Unlike pivoting algorithms, the *dsa–tree* cannot use more memory to improve query costs.

In this work describe a hybrid index for metric space searching built on top of the *dsa–tree*. This new data structure inherites all the properties of the *dsa-tree* and makes better use of the available memory.

We also describe further improvements, generalizations and optimizations being carried out within our research group.

2. Previous Work

Algorithms to search in general metric spaces can be divided in two large areas: pivot-based and clustering algorithms. (See [3] for a more complete review.)

Pivot-based algorithms. The idea is to use a set of k distinguished elements ("pivots") $p_1 ... p_k \in S$ and storing, for each database element x, its distance to the k pivots ($d(x,p_1) ... d(x,p_k)$). Given the query q, its distance to the k pivots is computed ($d(q,p_1) ... d(q,p_k)$). Then the *distance by pivots* between $x \in S$ and q gets defined as $\mathcal{D}(x,q) = \max_{p_i} |d(x,p_i) - d(q,p_i)|$. It can be seen that $\mathcal{D}(x,q) \leqslant d(x,q)$ for all $a \in S,\ q \in \mathcal{U}$. This is used to avoid distance evaluations. Each x such that $\mathcal{D}(x,q) > r$ can be discarded because we deduce $d(x,q) > r$ without actually computing $d(x,q)$. All the elements that cannot be discarded this way are directly compared against q.

Usually pivoting schemes perform better as more pivots are used, this way beating any other index. They are, however, better suited to "easy" metric spaces [3]. In hard spaces they need too many pivots to beat other algorithms.

Local Partitioning Algorithms The second trend consists in dividing the space in zones as compact as possible, normally recursively, and storing a representative point ("center") for each zone plus a few extra data that permits quickly discarding the zone at query time. Two criteria can be used to delimit a zone.

The first one is the *Dirichlet Domain*, where we select a set of centers and put each other point inside the zone of its closest center. The areas are delimited by hyperplanes and the zones are analogous to Voronoi regions in vector spaces. Let $\{c_1 \dots c_m\}$ be the set of centers. At query time we evaluate $(d(q,c_1), \dots, d(q,c_m))$, choose the closest center c and discard every zone whose center c_i satisfies $d(q,c_i) > d(q,c) + 2r$, as its Dirichlet Domain cannot intersect with the query ball.

The second criterion is the *covering radius* $cr(c_i)$, which is the maximum distance between c_i and an element in its zone. If $d(q,c_i) - r > cr(c_i)$, then there is no need to consider zone i.

3. Dynamic Spatial Approximation Trees

In this section we briefly describe *dsa-trees*, in particular the version called *timestamp with bounded arity* [5], on top of which we build.

3.1. Insertion Algorithm

The *dsa–tree* is built incrementally, via insertions. The tree has a maximum arity. Each tree node a stores a timestamp of its insertion time, $time(a)$, and its covering radius, $R(a)$, which is the maximum distance to any element in its subtree. Its set of children is called $N(a)$, the *neighbors* of a. To insert a new element x, its point of insertion is sought starting at the tree root and moving to the neighbor closest to x, updating $R(a)$ in the way. We finally insert x as a new (leaf) child of a if (1) x is closer to a than to any $b \in N(a)$, and (2) the arity of a, $|N(a)|$, is not already maximal. Neighbors are stored left to right in increasing timestamp order. Note that the parent is always older than its children.

3.2. Range Search Algorithm

The idea is to replicate the insertion process of elements to retrieve. That is, we proceed as inserting q but keep in mind that relevant elements may be at distance up to r from q, so in each decision for simulating the insertion of q we permit a tolerance of $\pm r$.

Note that when an element x was inserted, a particular node a may not be its parent when its arity was already maximal. So, at query time, we must choose the minimum distance to x only among $N(a)$. Also note that, when x was inserted, elements with higher timestamp were not yet present in the tree, so x could choose its closest neighbor only among older elements. Hence, we consider the neighbors $\{b_1, \ldots, b_k\}$ of a from oldest to newest, disregarding a, and perform the minimization as we traverse the list. We use timestamps to reduce the work inside older neighbors.

4. Dynamic Spatial Approximation Trees with Pivots

The idea behind the data structure is to enrich *dsa–trees* with the ability of working with pivots, besides its own spatial approximation approach, so as to get a hybrid data structure.

Pivoting techniques can trade memory space for query time, but they perform well on easy spaces only. A *dsa–tree*, on the other hand, is suitable for searching spaces of medium difficulty. However, it uses a fixed amount of memory, being unable of using additional memory to improve query time. The novel idea is to obtain a hybrid data structure getting the best of both worlds enriching *dsa–trees* with pivots. As expected, the result is better than both building blocks.

4.1. Insertion Algorithm

The new data structure is also built incrementally, via insertions. The tree has a maximum arity. Like the *dsa–tree*, each tree node a stores a timestamp of its insertion time, $time(a)$, and its covering radius, $R(a)$. Its set of children is called $N(a)$, the *neighbors* of a. To insert a new element x, its point of insertion is sought starting at the tree root and moving to the neighbor closest to x, updating $R(a)$ along the way. We finally insert x as a new (leaf) child of a if (1) x is closer to a than any $b \in N(a)$, and (2) the arity of a, $|N(a)|$, is not already maximal. If x is closer to a than any $b \in N(a)$, but the arity of a is already maximal, instead of choosing the closer neighbor of a to x and going downward into the tree (like the *dsa-tree* should do), we put X in a common overflow area. For each element stored in this overflow area we maintain a list of pairs *(node, distance)*, with all the distances evaluated during insertion, as pivots.

We choose different pivots for each element in the overflow area, such that *we do not need any extra distance evaluations against pivots*, either at insertion or search time. Recall that, after we find the insertion point of a new element x, say a, if x cannot become a neighbor of a because the arity of a is maximal, x has been compared against all of his ancestors in the tree, all the siblings of its ancestors, and all the neighbors of a. At query time, when we reach node x, some distances between q and the aforementioned elements have also been computed. So, we can use these elements as pivots to obtain better search performance, without introducing extra distance computations.

4.2. Range Search Algorithm

The search process can be divided in two parts: the first is to search into the tree, and the second is searching into the common overflow area.

During the range search into the tree we replicate the insertion process of elements to retrieve, like the original *dsa-tree*. That is, we act as if we wanted to insert q but keep in mind that relevant elements may be at distance up to r from q, so in each decision for simulating the insertion of q we permit a tolerance of $\pm r$. So it may be that relevant elements were inserted in different children of the current node, and backtracking is necessary. We maintain the pairs *(node,distance)*, of all distances evaluated between q and the elements into the tree for use in the second part of the search (the overflow search).

At query time, unlike the *dsa-tree*, we can choose the minimum distance to x among $\{a\} \cup N(a)$. Note also that, when x was inserted, elements with higher timestamp were not yet present in the tree, so x could choose its closest neighbor only among older elements. Hence, we consider the neighbors $\{b_1, \ldots, b_k\}$ of a from oldest to newest, considering a, and perform the minimization as we traverse the list. We use timestamps to reduce the work inside older neighbors, exactly as in the *dsa–tree* range searching.

The second part of a range search is to search into the overflow area. The elements in it are not organized, so we traverse them in a list. We can avoid the distance computation between an element x and the query q for two reason: (1) the query q did not enter by the same path x did (2) the pivot distance is greater than r (that is $\mathcal{D}(q, x) > r$). In other case we must compare q against x, and we report x if it is near enough.

It can be noticed that we take advantage of work done during the first part of the search, not only using the distances previously evaluated (like distances to pivots), but also using the information obtained by the traversal of the tree. Moreover, we compute $\mathcal{D}(q, x)$ at search time without additional distance evaluations.

5. Conclusions and Future Work

In this work we have proposed a hybrid scheme for similarity searching in metric spaces. Such scheme is built on top of *dsa–tree*, by adding a common overflow area associating each element stored in this area with the distances computed during insertion into the *dsa–tree*. A metric range search proceeds in two parts. The first one is almost the same metric range search of the *dsa–tree*. The second one is a sequential scan in the overflow area, using pivots to prune the search space for free and additionally using the information obtained during the search into the tree.

As work in progress, already being carried out within our research group, we studying:

- The deletion problem. It is easy to delete elements allocated from the common overflow area, but it is difficult to delete an element from the tree, because any element in the overflow area could be used it as pivot.
- The secondary memory implementation of the hybrid *dsa–tree*. Here both the number of distance computations and random disk accesses will be optimized.
- Comparing against pure pivoting, and pure clustering.
- Characterizing the intrinsic dimension of the metric space by *natural* arity of the *dsa-tree* on a sample of the space.
- Characterizing the *good* roots of the *dsa-tree*. Not every point will behave the same as a root, Or does it?

Referencias

[1] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.

[2] S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[3] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[5] G. Navarro and N. Reyes. Fully dynamic spatial approximation trees. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, LNCS 2476, pages 254–270. Springer, 2002.