An Unbalanced Approach to Metric Space Searching *

Edgar Chávez † **Verónica Ludueña** ‡ **Nora Reyes** ‡

Departamento de Informática, Universidad Nacional de San Luis.‡
Facultad de Cs. Físico-Matemáticas, Universidad Michoacana de México †
elchavez@umich.mx vlud@unsl.edu.ar nreyes@unsl.edu.ar

Abstract

Proximity queries (the searching problem generalized beyond exact match) is mostly modeled as metric space. A metric space consists of a collection of objects and a distance function defined among them. The goal is to preprocess the data set (a slow procedure) to quickly answer proximity queries. This problem have received a lot of attention recently, specially in the pattern recognition community.

The **Excluded Middle Vantage Point Forest** (*VP–forest*) is a data structure designed to search in high dimensional vector spaces. A *VP–forest* is built as a collection of *balanced* **Vantage Point Trees** (*VP–trees*).

In this work we propose a novel two-fold approach for searching. Firstly we extend the *VP–forest* to search in metric spaces, and more importantly we test a counterintuitive modification to the *VP–tree*, namely to unbalance it. In exact searching an unbalanced data structure perform poorly, and most of the algorithmic effort is directed to obtain a balanced data structure. The unbalancing approach is motivated by a recent data structure (the *List of Clusters*) specialized in high dimensional metric space searches, which is an extremely unbalanced data structure (a linked list) outperforming other approaches.

1. Introduction and motivation

The concept of "approximate" searching has applications in a vast number of fields. Some examples are next-generation 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 $\forall x \in A, y \in S - A, d(x, q) \leq d(y, q)$.

*This work has been partially supported by RIBIDI Project CYTED VII.19 and CONACyT grant 36911-A

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 to compute a minimal amount of distances in answering proximity queries.

There are a number of methods to preprocess the set (to index the data set) in order to reduce the number of distance evaluations [2]. All those structures work on the basis of discarding elements using the triangle inequality, and most of them use the classical divide-and-conquer approach (which is a general algorithmic approach).

The *VP–forest* is a data structure supporting *worst case* sublinear time searches in high dimensional vector spaces for fixed-radius nearest-neighbor queries. Worst case performance depends of the dataset but is not affected by the distribution of queries.

The *VP–tree* was proposed in an early paper by Peter Yianilos [3], which is a straightforward implementation of a general multidimensional search.

The above data structures are balanced. Perhaps following general algorithmic criteria, and strongly motivated by exact searching problems. However it has been proved in [1] that unbalancing is indeed a good idea to save distance computations.

In this work we will test the unbalancing hypothesis in this particular data structure.

2. VP–tree and VP–forest

In this section we briefly describe both data structures, the *VP–forest* and the *VP–tree*, more details can be seen in [3, 4].

The *VP–tree)* organize a database S with n elements allowing metric-range and nearest-neighbor queries with a expected sublinear complexity for a fixed distribution. Performance depends on the dataset and on the assumed distribution of queries. The *VP–forest* is a related data structure supporting *worst case* sublinear time searches for fixed-radius (τ) nearest neighbors queries. Worst case performance depends on the dataset but is not affected by the distribution of queries.

The dataset is organized into a forest of $O(n^{1-\rho})$ trees, each of depth $O(logn)$. Here ρ may be viewed as depending on τ, the distance measure, and on the dataset. The radius of interest τ is an input to the organization process and the result is a data structure specialized to answer queries within this distance.

Each element of the dataset occurs in exactly one tree so that the entire forest remains linear space. Searches follow a single root-leaf path in each tree. There in no backtracking when the search is limited to neighbors within distance τ. Along its way *every* neighbor within τ is necessarily encountered. The effect of the query is to guide the descent through each tree.

The general idea behind *VP–forest* is easily understood. A *VP–tree* recursively divide the dataset using as a ruler the distance to a fixed *vantage point* or *pivot*. At each node a, roughly central, fixed threshold is used in the distribution of values. Elements below this threshold are assigned to, say, the left child, and those above to the right child. Elements near to the threshold lead to backtracking during search.

The above backtracking problem is eliminated in the *VP–forest*, by deleting such elements from the tree. Once the tree is completed, the throwed elements are organized into a different tree with the same procedure, resulting in another (smaller) bucket of elements. This continues until all the elements in the dataset belong to a tree. This effectively eliminates backtracking, because elements near to the threshold

are recursively deleted, and this threshold lies near the middle of the distribution of values. This data structure is referred as an *excluded middle vantage point forest*.

The ideas and construction sketched can be formalized by the following definition.

Definition 1 *Consider an ordered set $S = \{x_1 \dots x_n\}$ and a value $m \in [0, 1]$. Let $w = \lfloor mn \rfloor$ and $a = \lfloor (n - w)/2 \rfloor$. Then, the m-split of S consists of left, middle, and right subsets defined by:*

$$\begin{aligned} L &= \{x_i | i \leq a\} \\ M &= \{x_i | i > a, i \leq a + w\} \\ R &= \{x_i | i > a + w\} \end{aligned}$$

That is, a balanced 3-way partition of S with a central proportion of approximately m.

3. Balancing versus Unbalancing

We learn in any elementary book of algorithms, as one of the first lessons, that balanced data structures (trees in general) provide the best performance. Indeed, as a tree becomes more unbalanced it becomes more similar to a linked list, and the cost raises from $O(\log n)$ to $O(n)$.

However, all the concept of balancing is based on the implicit assumption of *exact searching*: We have a search query and want to find its exact replica in the tree. Hence, we enter only one branch of the tree, and therefore a balanced tree minimizes the cost. More sophisticated queries such as range searching are still based on the assumption that there exist a total linear order on the keys. Hence, these queries are reduced to a couple of exact searches to find the extremes of the range of interest.

None of these assumptions is valid in proximity searching. The only tool to organize a data structure on metrics spaces is the distances among elements. Many proposals still manage to design tree data structures, where a total linear order is imposed by sorting the elements according to their distances to the root. Probably influenced by a strong algorithmic background, most authors try as well to obtain a balanced data structure by splitting the range of distances so that the subtrees have the same size, as the *VP–tree* does.

The real problem with this approach appears when one considers the type of search carried out on these balanced trees. As explained, the search is not exact, but it has a tolerance radius r which is fixed at query time and is insensitive to the slices assigned by the tree. Low dimensional metric spaces have an histogram of distances which is more uniform than those of high dimensional spaces. In low dimensional spaces, therefore, the query is compared against the root and a range of the histogram is selected. This range contains a reasonably small fraction of the distribution and therefore the problem is reduced well along the iterations. Moreover, since the histogram is not concentrated, a partition where the subtree have the same number of elements yields slices of approximately the same width, and therefore the search enters into a reasonable number of subtrees. Consider now a high dimensional space. All the histogram is concentrated in a small range, where the query also lies with high probability when compared to the root of the tree. Hence a large proportion of the elements will now be selected by the query range. This is the basic reason that makes searching in high dimensional spaces so difficult.

However, balancing the trees adds an extra inefficiency to this. As the histogram becomes more concentrated, the slices to partition the elements in equal size groups become thinner. Since the search radius stays the same, it will intersect more slices and the search will need to enter more subtrees. This shows why the search model for proximity queries makes balanced trees a poor choice for high dimensional metric spaces.

A tree where the slices have fixed width avoids this last problem. Since the width is independent on the dimension of the space, the search will not enter more subtrees of a node as the dimension grows. However, a new consequence shows up when fixed slices are used: The subtrees corresponding to the slices containing the core of the distribution will have much more elements that the rest, ane therefore the tree will be more and more unbalanced as the dimensions grows.

As the tree becomes more unbalanced and hence taller as the dimensions grows, the leaves of the tree will know their (approximately) distance to more pivots: from $O(\log n)$ in a balanced tree to $O(n)$ in a very unbalanced tree. Also a random query will be compared against more pivots as it traverses the tree. this effect is very similar to having a large number k of pivots in plain pivot-based algorithms. Unlike those, however, this unbalanced tree takes always linear space. The main problem is its construction cost, which moves from $O(n \log n)$ to $O(n)$ as the tree loses balance.

To summarize, unbalancing permits, in essence, to have a large number of pivots without incrementing the space cost (the price is paid in construction cost). Hence we can reach the optimum number of pivots, which grows with the dimension.

4. Unbalancing a VP–forest

The idea behind our proposal is to improve the *VP–forest* searching performance in high dimensional spaces, unbalancing the *VP–trees* that compound it.

As *VP–forest* does, we recursively divide the dataset in three parts, but the parts sizes are not similar. Hence, the trees will not be balanced. We choose an element p of S as pivot, and we consider the set ordered by distance to p and a value $m \in [0, 1]$. Let $w = \lfloor mn \rfloor$ and $a = \lfloor (n - w)/k \rfloor$, where k can be a value not equal to 2. Then the m-split of S consists again of left, middle, and right subsets defined by:

$$\begin{aligned} L &= \{x_i | i \leq a\} \\ M &= \{x_i | i > a, i \leq a + w\} \\ R &= \{x_i | i > a + w\} \end{aligned}$$

Note that it is a 3-way partition of S with a central part M with a proportion of m, but L and R parts can have very dissimilar proportion of element, depending on the chosen value of k. The particular case when $k = 2$ will produce a *VP–forest* balanced.

The simpler way to define this 3-way partition of S, using p as a pivot, is considering to obtain a distance value d_m to p such that the number of elements whose distance to p is lower than $d_m - \tau$ is approximately a (those form L). R would be the elements at distance greater than $d_m + \tau$ to p and M would be the elements whose distance to p belong to the range $[d_m - \tau, d_m + \tau]$.

At each node we put the element p of S, chosen as pivot, and L and R parts are assigned to its left and right child, and recursively we continue on both parts. Once the tree is completed, all the elements discarded, that is those of the M part for p and those discarded during the construction of the left and right subtrees, are organized into a tree in the same way, resulting another (smaller) set of discarded elements. This continues until the forest is completely built. We can observe that all the trees will be unbalanced, because all of them have different sizes between left and right subtrees.

When we search an element q with radius r, in our proposed structure, we compare q against the root element p of the first tree. If $d_m - \tau \leq d(p, q) \pm r \leq d_m + \tau$ we do not need to enter in this tree, because we known that all the elements relevant to the query q must be in another tree. If $d(p, q) \pm r < d_m + \tau$, or if $d(p, q) \pm r > d_m + \tau$, we continue recursively the searching process only on the left, or right, subtree of p respectively . In any other case, when the ball query intersect more than one zone, we should enter into

more than one subtree, and/or more than one tree. The worst case is when the ball query intersect every zone of p, because in this case we must enter in each subtree of p and we need to continue searching on the following trees. However, in the next paths of the process we will consider another pivot that it should permits us to avoid entering in some branches.

Hence, although we unbalance the trees, we also effectively eliminates backtracking if the query radius r is lower than τ, because elements of M parts are recursively deleted.

5. Conclusions and Future Work

In this work we have proposed to unbalance a data structure already known for similarity searching in metric spaces, in order to improve its searching performance in high dimensional spaces.

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

- Experimenting with different metric spaces of low and high intrinsic dimensionality.
- Comparing against *List of Clusters* [1], that is a unbalanced data structure for searching in metric spaces and very resistant to the intrinsic dimensionality of the data set.
- Comparing against data structures that use pure pivoting.
- Characterizing the good unbalancing for the intrinsic dimension of the metric space.
- Characterizing good values of the radius τ for the queries that could be considered in each space.
- Analyzing if there is another way to make unbalanced a *VP–forest*, and if it is possible, comparing it against our proposal, in order to obtain the best way.

References

[1] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*. To appear.

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

[3] 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)*, pages 311–321, 1993.

[4] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, Baltimore, MD, 1999.