Searching in Metric Spaces by Spatial Approximation *

Gonzalo Navarro
Dept. of Computer Science, University of Chile
Blanco Encalada 2120 - Santiago - Chile
gnavarro@dcc.uchile.cl

Abstract

We propose a new data structure to search in metric spaces. A metric space *is formed by a collection of objects and a* distance function *defined among them, which satisfies the triangular inequality. The goal is, given a set of objects and a query, retrieve those objects close enough to the query. The number of distances computed to achieve this goal is the complexity measure. Our data structure, called* sa-tree *("spatial approximation tree"), is based on approaching spatially the searched objects. We analyze our method and show that the number of distance evaluations to search among n objects is $o(n)$. We show experimentally that the* sa-tree *is the best existing technique when the metric space is high-dimensional or the query has low selectivity. These are the most difficult cases in real applications.*

1. Introduction

The concept of "approximate" searching has applications in a vast number of fields. Some examples are non-traditional databases (where the concept of exact search is of no use and we search for similar objects, e.g. databases storing images, fingerprints or audio clips); machine learning and classification (where a new element must be classified according to its closest existing element); image quantization and compression (where only some vectors can be represented and those that cannot must be coded as their closest representable point); 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); function prediction (where we want to search the most similar behavior of a function in the past so as to predict its probable future behavior); etc.

All those applications have some common characteristics. There is a universe U of *objects*, and a non-negative *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*

$$\begin{aligned} d(x,y) &= 0 \quad \Leftrightarrow \quad x = y \\ d(x,y) &= d(y,x) \\ d(x,z) &\leq d(x,y) + d(y,z) \end{aligned}$$

where the last one is called the "triangular inequality" and is valid for many reasonable similarity functions. The smaller the distance between two objects, the more "similar" they are. This distance is considered expensive to compute (think, for instance, in comparing two fingerprints). 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 instance). Later, given a new object from the universe (a *query* q), we must retrieve all similar elements found in the database. There are three typical queries of this kind:

(a) Retrieve all elements which are within distance r to q. This is, $\{x \in S \ / \ d(x,q) \leq r\}$.

(b) Retrieve the closest elements to q in S. This is, $\{x \in S \ / \ \forall y \in S, \ d(x,q) \leq d(y,q)\}$.

(c) Retrieve the k closest elements to q in S. This is, retrieve a set $A \subseteq S$ such that $|A| = k$ and $\forall x \in A, y \in S - A, d(x,q) \leq d(y,q)$.

Given a database of $|S| = n$ objects, all those 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 $\mathbb{R}^k$. There are effective methods for this case, such as kd-trees [3] or R-trees [9]. However, for roughly 20 dimensions or more those structures cease

*This work has been supported in part by Fondecyt grant 1990627.

to work well. We focus in this paper in general metric spaces, although the solutions are well suited also for k-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 is that the probability distribution of distances among elements has a very concentrated histogram (with larger mean as the dimension grows), difficulting the work of any similarity search algorithm [5, 7]. In the extreme case we have a space where $d(x,x) = 0$ and $\forall y \neq x,\ d(x,y) = 1$, where it is impossible to avoid a single distance evaluation at search time. 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 them work by discarding elements with the triangular inequality.

In this work we present a new data structure to answer similarity queries in metric spaces. We call it *sa-tree*, or "spatial approximation tree". It is based on a completely novel concept, namely to approach the query spatially, getting closer and closer to it, instead of the generally used technique of partitioning the set of candidate elements. We start by presenting an ideal data structure that, as we prove, cannot be built, and then design a tradeoff which can be built. We analyze the performance of the structure, showing that the number of distance evaluations is $o(n)$. We also experimentally compare our data structure against previous work, showing that it outperforms all the other schemes for high dimensions or queries with large radii.

2. Previous Work

Different tree structures have been proposed to filter out elements based on the triangular inequality. Burkhard-Keller Trees (bk-trees) [6] are designed for discrete distance functions: they select a pivot element p as the root of the tree, and put at child i the elements which are at distance i to the pivot. Each subtree is recursively built with the same technique until there are b elements or less, in which case the elements are simply stored in a "bucket" at the tree leaf. A type (a) query q with tolerance radius r is searched by measuring $d(p,q)$, reporting p if appropriate, and entering only into subtrees numbered $d(p,q) - r$ to $d(p,q) + r$. The rest are filtered out with the triangle inequality. The buckets reached are exhaustively compared against q.

Fixed Queries Trees (fq-trees) [2] are an evolution where the same pivot is used for all the nodes of the same level of the tree. In this case the pivot does not need to belong to the subtree. Many comparisons are saved in the backtracking process because only one different pivot per level exists. However, the tree is taller. A variant called Fixed Height fq-tree (fhq-tree) is also proposed where all the leaves are at the same depth h, regardless of the bucket size.

Vantage Point Trees (vp-trees) [13, 15] are designed for continuous distance functions. The root has two equal-size subtrees that divide the elements in closer to and farther from the root. This can be extended to m-ary trees (mvp-trees) [5, 4].

Generalized hyperplane trees (gh-trees) [13] use two pivots for each tree node and divide the space according to which of the two pivots is closer to each object. If this is generalized to an m-ary partition then a Geometric Near-neighbor Access Tree (gna-tree) is obtained [5], which makes a Voronoi-like partition of the space [1] among the m pivots at each node of the tree.

Finally, algorithms like AESA [14], LAESA [11, 10] and others [12, 8] are based in a common idea: k pivots are selected and each object is mapped to k coordinates which are its distance to the pivots. Later, the query q is also mapped and if it differs from an object in more than r along some coordinate then the element is filtered out by the triangle inequality. The rest of the elements are directly compared.

3. Spatial Approximation

We concentrate in this section on queries of the type (b). Instead of the known algorithms to solve approximate queries by dividing the set of candidates, we try a different approach here. In our model, we are always positioned at a given element of S and try to get "spatially" closer to the query (i.e. move to another element which is closer to the query than the current one). When this is no longer possible, we are positioned at the nearest element to the query in the set.

Those approximations are performed only via "neighbors". Each set element $a \in S$ has a set of neighbors $N(a)$, and we are allowed to move only to neighbors. The natural structure to represent this restriction is a directed graph. The nodes are the elements of the set and the neighbors are connected by an edge. More specifically, there is an edge from a to b if it is possible to move from a to b in a single step.

Once such graph is suitably defined, the search process for a query q is simple: start positioned at a random node a and consider all its neighbors. If no neighbor is closer to q than a, then report a as the closest neighbor to q. Otherwise, select some neighbor b closer to q than a and move to b. We can choose b as the neighbor which is closest to q or as the first one we find closer than a.

In order for that algorithm to work, the graph must contain enough edges. The simplest graph that works is the complete graph, i.e. all pairs of nodes are neighbors. However, this implies n distance evaluations just to check the neighbors of the first node! We prefer the graph which has the *least* possible number of edges and still allows to answer correctly all queries. This graph G must enforce the following property:

Condition 1: $\forall a \in S$, $\forall q \in U$, if $\forall b \in N(a), d(q,a) \leq d(q,b)$, then $\forall b \in S, d(q,a) \leq d(q,b)$.

This means that, given *any* possible element q, if we cannot get closer to q from a going to its neighbors, then it is because a is already the element closest to q in the whole set S. Expressed in this way it is the same that we already had, and therefore it is clear that if G satisfies Condition 1 we can search by spatial approximation. We seek a minimal graph of that kind.

This can be seen in another way: each $a \in S$ has a subset of U where it is the proper answer (i.e. the set of objects closer to a than to any other element of S). This is the exact analogous of a "Voronoi region" for Euclidean spaces in computational geometry [1]. The answer to the query q is the set element a which owns the Voronoi region where q lies. We need, if a is not the answer, to be able to move to another element closer to q. It is enough to connect each a with all its "Voronoi neighbors" (i.e. set elements whose Voronoi area shares a border with that of a), since if a is not the answer, then a Voronoi neighbor will be closer to q (this is exactly the Condition 1 just stated).

Consider the hyperplane between a and b (i.e. which divides the area of points x closer to a or closer to b). Each neighbor b we add to a will make the query to move from a to b provided q is in b's part of the hyperplane. Therefore, if (and only if) we add all the Voronoi neighbors to a, the only zone where the query would not move away from a will be exactly the area where a is the closest neighbor.

In a k-dimensional space, the minimal graph we seek corresponds to the classical Voronoi graph (where elements which are Voronoi neighbors are connected). The Voronoi graph (generalized to arbitrary spaces) is therefore the ideal answer in terms of space complexity, and should have good performance too.

Unfortunately, it is not possible to compute the Voronoi graph of a general metric space given only the set of distances among elements of S and no further indication of the structure of the space. This is because, given the set of $|S|^2$ distances, different spaces will have different Voronoi graphs. Moreover, it is not possible to prove that a single edge from any a to b nodes is not in the Voronoi graph. Therefore, the only superset of the Voronoi graph that works for an arbitrary metric space is the complete graph, and as explained this graph is useless. This outrules the data structure for general applications. We formalize this notion as a theorem.

Theorem: given a set S of elements in an unknown metric space U, and given the distances among each pair of elements in S, then for each $a, b \in S$ there exists a valid metric space U where a and b are connected in the Voronoi graph of S.

Proof: given the set of distances, we create a new element $x \in U$ such that $d(a,x) = M + \epsilon$, $d(b,x) = M$, and $d(y,x) = M + 2\epsilon$ for all others $y \in S$. This satisfies all triangle inequalities provided $\epsilon \leq 1/2 \min_{y,z \in S}\{d(y,z)\}$ and $M \geq 1/2 \max_{y,z \in S}\{d(y,z)\}$. Therefore, such an x may exist in U. Now, given the query $q = x$ and given that we are currently at element a, b is the element nearest to x and the only way to move to b without getting farther from q is a direct edge from a to b. See Figure 1. This argument can be repeated for any pair $a, b \in S$.

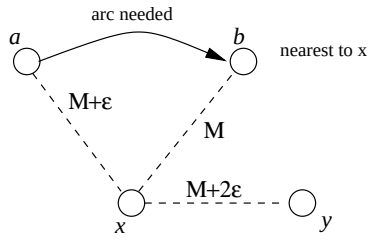

Figure 1. Illustration of the theorem.

4. The Spatial Approximation Tree

We make two crucial simplifications to the general idea to achieve a feasible solution. The resulting simplification answers only a reduced set of queries, but we show later how to solve the general case using the same structure.

(1) We do not start traversing the graph from a random node but from a fixed one, and therefore there is no need of all the Voronoi edges.

(2) Our graph will only be able to answer correctly queries $q \in S$, i.e. only elements already present in the database.

Given those simplifications, we can build the analogous to the Voronoi graph to search by spatial approximation queries of type (b). Actually, the result is not a graph but a tree, which we call the *sa-tree* ("spatial approximation tree"). Later, we show how to use this tree to search any query $q \in U$ (not only $q \in S$), for problems of type (a), (b) and (c) (not only (b)).

4.1. Construction Process

We select an element $a \in S$ to be the root of the tree. We then select a suitable set of neighbors $N(a)$ satisfying the following property:

Condition 2: (given a, S) $\quad \forall x \in S,\ x \in N(a) \Leftrightarrow \forall y \in N(a) - \{x\}, d(x,a) < d(x,y)$.

That is, the neighbors form a set such that any neighbor is closer to a than to any other neighbor. Notice that the set is defined in terms of itself in a non-trivial way. We want the smallest possible set $N(a)$.

Observe that if $d(x,a) \geq d(x,y)$ and y is already in $N(a)$, then x is not in $N(a)$. Therefore, we can define an "exclusion graph" where in the mentioned case y has an edge to x. However, there are loops in the exclusion graph. For instance, it may be the case that, by adding y to $N(a)$, x is excluded, and vice versa. The minimal set of neighbors sought is a maximal subset of the nodes with no edges among them. This is a particular case of the Independent Set problem, which is NP-complete. It is not immediate that Independent Set can be reduced to this problem but it also seems not easy to take advantage of our particular case.

However, simple heuristics which add more neighbors than necessary work well. We begin with the initial node a and its "queue" holding all the rest of S. Since we expect that closer nodes are more likely to be neighbors, we first sort the set of nodes by distance to a. Then, we start adding nodes to $N(a)$ (which is initially empty). Each time we consider a new node b, we see if it is closer to some element of $N(a)$ than to a itself. If that is not the case, we add b to $N(a)$. At the end we have a suitable set of neighbors. We now put each node not in $\{a\} \cup N(a)$ in the queue of its closest element of $N(a)$. Observe that this requires a second pass on the queue once $N(a)$ is fully determined.

We are done now with a, and process recursively all its neighbors, each one with the elements of its queue. Note that the resulting structure is not a graph but a tree, which can be searched for any $q \in S$ by spatial approximation for queries of type (b). The reason why it works is that, at search time, we can repeat what happened with q during the construction process until we reach q (this is because q is already in the tree). Being a tree, the space needed by this structure is $O(n)$. Figure 2 depicts the building process, where the first invocation is `Build(`$a, S - \{a\}$`)` with a a random element of the set S.

A problem with this structure is that it is difficult to add new elements, since the construction algorithm needs all elements. Each time a new element is inserted, we must go down the tree until the new element must become a neighbor of the current node. All the subtree must be rebuilt from scratch (since some nodes that went into another neighbor could prefer now to get into the new neighbor). An alternative is to have a queue per node with "extra" elements against which the query must be compared but have no subtree to follow. At periodic intervals, the index must be rebuilt to maintain the search efficiency.

```
Build (Node a, Queue of nodes Q)

N ← ∅        /* neighbors of a */
Sort Q by distance to a (closer first)
for v ∈ Q do
   if ∀b ∈ N, d(v,a) < d(v,b) then N ← N ∪ {v}
for b ∈ N do Q(b) ← ∅   /* subtrees queues */
for v ∈ Q − N − {a} do
   Let b ∈ N be the one minimizing d(v,b)
   Q(b) ← Q(b) ∪ {v}
for b ∈ N do        /* build subtrees */
   Add b as a child of a
   Build (b, Q(b))
```

Figure 2. Construction algorithm.

4.2. Searching

Of course it is of little interest to search only for elements $q \in S$. The tree we have described can, however, be used as a device to solve queries of any type for any $q \in U$. We start with type (a).

The key observation is that the answers to the query are elements $q' \in S$. So we use the tree to pretend that we are searching an element $q' \in S$. We do not know q', but using q we have some distance information: by the triangular inequality it holds that for any $x \in U$, $d(x,q) - r \leq d(x,q') \leq d(x,q) + r$, where r is the tolerance of our search.

Therefore, instead of simply going to the closest neighbor, we first determine the closest neighbor of q among $\{a\} \cup N(a)$ (say it is c). We then enter into *all* neighbors $b \in N(a)$ such that $d(q,b) \leq d(q,c) + 2r$. This is because the virtual element q' we are searching for can differ from q in at most r at any distance evaluation. In the way, we report all the nodes we have seen which are close enough to q. Therefore, what was originally conceived as a search by spatial approximation along a single path is combined now with backtracking, so that we search by a number of paths.

Figure 3 depicts the algorithm. Initially, a is the root of the tree. Notice that in the recursive case $d(a,q)$ is already known. Below we show an example of the search process, starting from p_{11} (tree root). Only p_9

is in the result, but all the bold edges are traversed.

```
Search (Node a, Query q, Radius r)

if d(a,q) ≤ r then Report a
N ← children nodes of a
mind ← min_{c∈{a}∪N} d(c,q)
for b ∈ N do
   if d(b,q) ≤ mind + 2r then Search (b,q,r)
```

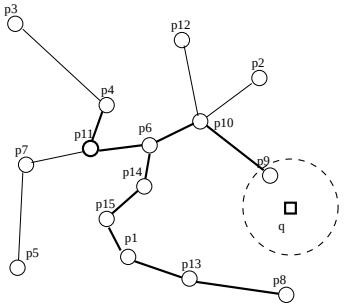

Figure 3. Search algorithm and example.

To solve queries of type (b), we start searching with $r = \infty$, and reduce r each time a new comparison is performed that gives a distance smaller than r. All the elements seen with the smallest distance found form the answer. In this case it is important to enter into each neighbor in order (closer neighbors first) to increase the chance of quickly reducing the tolerance r. Queries of type (c) are solved as a generalization of those of type (b): instead of just the closest neighbor we keep k closest neighbors and set r as the distance from q to the farthest among the k.

Finally, we can save some comparisons at query time by storing at each node a the maximum distance between a and any element in the subtree rooted by a. This information may show that it is not necessary to get into some subtrees at query time.

5. Analysis

We analyze now our *sa-tree* structure. Our analysis is simplified in many aspects, for instance it assumes that the distance distribution of nodes that go into a subtree is the same as in the global space. We also do not take into account that we sort the queue before selecting neighbors (the results are pessimistic in this sense, since it looks as if we had more neighbors). This analysis is done for a continuous distance function, although adapting it to the discrete case is immediate.

For the analysis that follows, we assume that the probability that two random elements are at distance x is p_x, where $\int_0^\infty p_x dx = 1$ (that is, p_x is the histogram of distances). We call P_x the probability that the distance is $< x$, i.e. $P_x = 1 - \int_x^\infty p_x dx$.

We select a random node as the root and determine which others are going to be neighbors. Imagine that a is the selected as root and b is an already present neighbor. The probability that a given node c is closer to a than to b is

$$A = \int_0^\infty p_x P_x \; dx$$

(where p_x refers to the possible values of $d(b,c)$ and P_x refers to $d(a,c)$). Therefore, if j neighbors are already present, the probability that we add another neighbor is that of being closer to a than to any neighbor, which is A^j. Calling X_j the random variable that counts the number of attempts to obtain the $(j+1)$-th neighbor given that there are already j, we have that X_j is hypergeometric with mean $1/A^j$. From scratch, we need $X_0 + X_1 + ... + X_{N-1}$ elements to obtain N neighbors. Since the expectation commutes with the sum, the average number of elements needed to obtain N neighbors is $\sum_{j=0}^{N-1} 1/A^j = (A^{-N} - 1)/(A^{-1} - 1)$.

We want to find which neighbor are we trying to add when the queue is exhausted, to determine how many neighbors we have on average. [1]. If the queue has n elements, we equate n with the previous expression to get that the average number of neighbors is

$$N(n) \;=\; \log_{1/A}(1 + n(A^{-1} - 1)) \;=\; \Theta(\log n)$$

although the constants depend on the probability distribution

This allows to determine some parameters of our index. For instance, since on average $\Theta(n/\log n)$ elements go into each subtree, the average depth of a leaf in the tree is

$$H(n) \;=\; 1 + H\left(\frac{n}{\log n}\right) \;=\; \Theta\left(\frac{\log n}{\log\log n}\right)$$

The construction cost is as follows (in terms of distance evaluations). The queue of n elements is compared against the root node. $\Theta(\log n)$ elements are selected as neighbors and then all the other elements are compared against the neighbors and are inserted into one queue. Then, all neighbors are recursively built.

$$B(n) = n\log n + \log(n) B\left(\frac{n}{\log n}\right) = \Theta\left(\frac{n\log^2 n}{\log\log n}\right)$$

The space needed by the index (number of links) is $O(n)$ because we have a tree.

[1] The exact solution is $N(n) = P_{n,0}$, where $P_{n,k} = A^k(1 + P_{n-1,k+1}) + (1 - A^k)P_{n-1,k}$ and $P_{0,k} = 0$. We have proved that $P_{n,k}$ is $O(\log n)$, but the proof is not included for lack of space.

We analyze the search times now. Since we enter into many neighbors, we must determine which is the amount of backtracking performed. The probability that, given the root a and j neighbors $v_1 ... v_j$, the element $c \in \{a, v_1, ..., v_j\}$ closest to q is at distance $\geq x$ from q is $P(d(q,c) \geq x) = P(d(q,a) \geq x) \times P(d(q,v_1) \geq x) \times ... \times P(d(q,v_j) \geq x) = (1 - P_x)^{j+1}$.

Therefore, the probability of entering into a given neighbor v_i is $P(d(q,v_i) \leq d(q,c) + 2r) = P(d(q,c) \geq d(q,v_i) - 2r) \leq P_{2r+\epsilon} + \int_{2r+\epsilon}^{\infty} p_x (1 - P_{x-2r})^{j+1} dx$, where the inequality holds for any $\epsilon \geq 0$ and becomes equality for $\epsilon = 0$. In the integral, p_x represents the possible values of $d(q,v_i)$. Now, since $j = \Theta(\log n) \leq s \ln n$ for some $s > 0$, we have that the probability is $P_{2r+\epsilon} + \int_{2r+\epsilon}^{\infty} p_x n^{s \ln(1-P_{x-2r})} dx \leq P_{2r+\epsilon} + n^{s \ln(1-P_\epsilon)} \int_{2r+\epsilon}^{\infty} p_x dx$, which is smaller than $P_{2r+\epsilon} + n^{-\alpha}$ for $\alpha = -s \ln(1 - P_\epsilon) > 0$. Hence, there is a constant part plus a negligible term. The constant part appears because all neighbors at distance $2r$ or less from q must be traversed no matter how close is the closest neighbor. Since there are $\Theta(\log n)$ neighbors, all them are compared against q, and on average we enter into $P_{2r+\epsilon}\Theta(\log n) + O(n^{-\alpha} \log n)$ of them. This makes the search cost $Q(n) = \log(n)(1 + P_{2r+\epsilon} Q(n/\log n))$, whose solution is

$$Q(n) = n^{1 - \Theta\left(\frac{\log(1/P_{2r+\epsilon})}{\log \log n}\right)} = n^{1 - \Theta(1/\log \log n)}$$

To give an idea of this complexity, we note that it is $o(n/\text{polylog}(n))$ but $\omega(n^x)$ for any $x < 1$. The effect of the dimensionality is present in P_{2r}. As the dimension is higher, we need that r approaches the mean of the distribution of distances in order to retrieve at least one element. But since the histogram is more and more concentrated and the mean larger, by the time the cummulative distribution P_r ceases to be zero, P_{2r} is almost one, since almost all the significant values are between P_r and P_{2r} [7]. This makes the exponent of n tend to 1 as the dimension grows.

6. Experimental Results

We have tested our *sa-tree* and previous work on a synthetic set of random points in a k-dimensional space. However, we have not used the fact that the space has coordinates, treating the points as abstract objects in an unknown metric space. This choice allows us to control the exact dimensionality we are working with, which is not so easy if the space is a general metric space or the points come from a real situation (where, despite that they are immersed in a k-dimensional space, their real dimension can be lower). Our tests use the Euclidean distance and four different dimensions: 5, 10, 15 and 20. For each dimension, we generated 6 groups of data sets, from $n = 50,000$ to $n = 300,000$ elements.

For each dimension, the height of the tree, average leaf depth and maximum arity of a node remain quite stable as n grows, showing a very small increment. For instance, on 10 dimensions they are (respectively) 12, 6.69 and 19 for $n = 50,000$; and 13, 8.10 and 23 for $n = 300,000$. The dimension has much more impact than the set size n, making the tree of smaller height and larger arity. For instance, for $n = 200,000$ the height, average leaf depth and arity are (respectively) 24, 12.89 and 9 for 5 dimensions; and 9, 5.51 and 67 for 20 dimensions.

In general, the *sa-tree* is more expensive to build than most other data structures. The construction times (averaged over 10 runs) are shown in Figure 4, measured in number of comparisons per element. The curves show the slightly superlinear behavior predicted in the analysis.

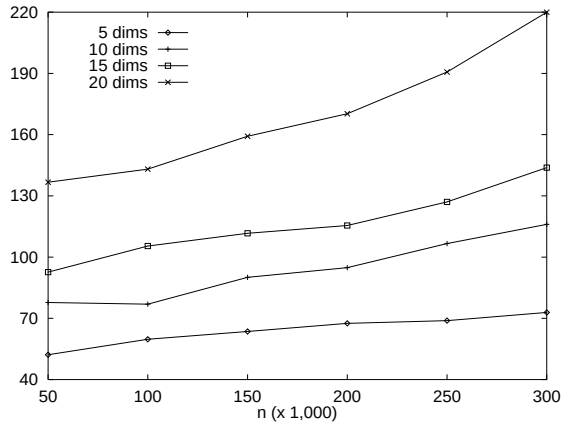

Figure 4. Comparisons per element to build the tree.

We consider search times now. We have performed range queries returning 0.01%, 0.1% and 1% of the total set size. This corresponds to more or less selectivity in the query (returning less elements is easier). The search radius to achieve each percentage grows with the dimension of the set, but remains stable as n grows. Figure 5 shows how the percentage of elements considered decreases as n grows, for different dimensions and selectivities (all the results have under 2% of error with 95% confidence). This shows that the number of comparisons is sublinear in the size of the set (as predicted) and that all the search times worsen as the dimension or the search radius grow.

We matched our cost model $t = an^{1 - c/\ln(\ln(n))}$ against the curves. The match is quite good, improving

for more dimensions. The values of c (corresponding to $\log(1/P_{2r})$) are as high as 1.79 for 0.01% selectivity in 5 dimensions and as low as 0.16 for 1% selectivity in 20 dimensions.

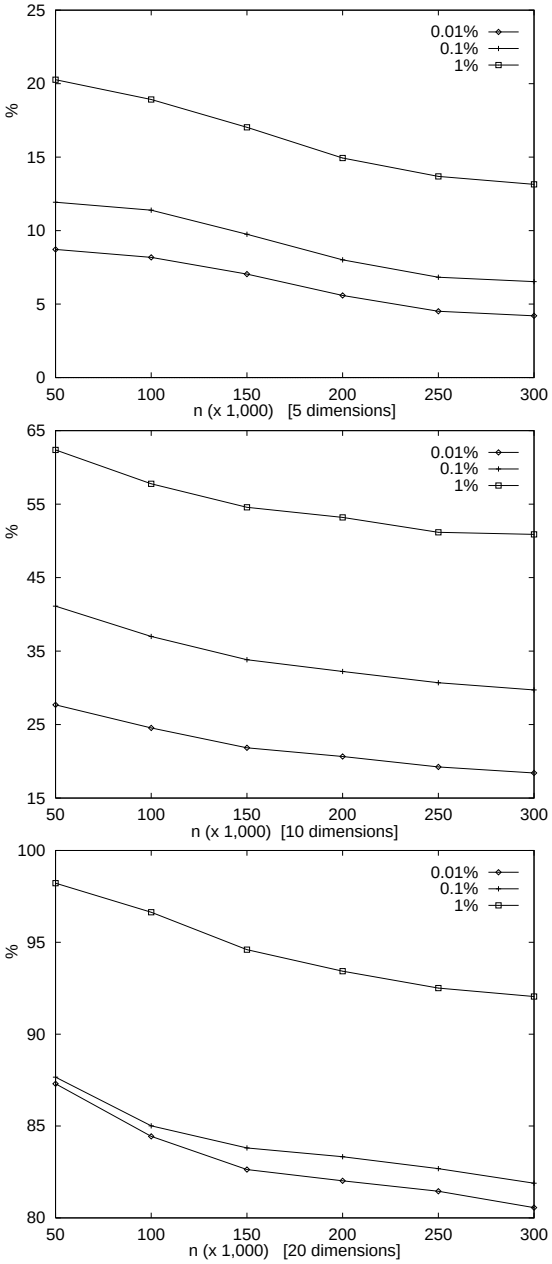

Figure 5. Set fraction traversed with the *sa-tree*.

Finally, Figure 6 compares our *sa-trees* against other data structures. This time we fix $n = 250,000$ and show how the results change with the dimension. We have tested bk-trees, fq-trees, fhq-trees, mvp-trees, gna-trees and k-pivots, manually selecting the best parameters for each structure. All the trees use buckets of size 1, which gives optimum performance. For bk-trees, fq-trees and fhq-trees we use slices at distances 0.15, 0.35, 0.45 and 0.55, as the dimension goes from 5 to 20. For mvp-trees the best arity was always 2 (i.e. vp-trees). For gna-trees we used arities of 4, 6, 10 and 14 as the dimension goes from 5 to 20.

The only case where we could not select the optimum setup is for fhq-trees and k-pivots. This is because their optimum needs so much memory that it cannot be achieved in practice. We have therefore limited the tree height (or number of pivots) so that their space requirement is 3 times that of *sa-trees*, which also matches the maximum space requirements of any other structure and is the maximum we can handle in our machine (64 Mb of RAM). This means using 12 pivots for k-pivots, and fhq-trees of heights 18, 21, 23 and 25 (growing with the dimension to use the same amount of memory). In all cases we selected the pivots and structure elements (tree roots, etc.) randomly, since there are no clear criteria in the source papers.

As seen in Figure 6, *sa-trees* become the best as the dimension grows or the query becomes less selective, the most difficult cases in practice.

7. Conclusions

We have presented a new data structure, the *sa-tree*, to search in metric spaces by approaching the query spatially rather than by reducing the set of candidates as in other approaches. As a byproduct we prove that no reasonable superset of the Voronoi graph of a metric space can be built using only the matrix of distances.

The *sa-tree* shows very good behavior on high dimensions (where the problem is more difficult) but is not so good when the problem is easier (low dimensions). This enables the possibility of designing hybrid schemes, such as replacing all the small enough subtrees (where the intrinsic dimension is lower) by another data structure better suited for that case. It is also possible to combine the *sa-tree* with a hierarchical clustering scheme, using the tree as a device to search among the clusters representatives at each level. We are currently working on those issues. We are also working on an extension of the *sa-tree*, an acyclic graph, where some redundancy is added to the tree in order to reduce backtracking in exchange for higher space requirements and construction cost. Finally, a problem still open is how to allow dynamic insertion and deletion of elements without degrading the performance. We are studying an alternative construction scheme where each element is inserted into the *first* neighbor closer than the root (instead of the closest neighbor). With this strategy we can pretend that the

new incoming element was the last one in the queue, which means that when it becomes a neighbor it can be simply added as the last neighbor. This allows to build the structure by successive insertions. Preliminary experimental results, however, indicate that the structure is very unbalanced and inferior to the current one.

Acknowledgements. We thank Edgar Chávez for fruitful discussions on the Voronoi graph and help with the experiments. We also thank Ricardo Baeza-Yates and an anonymous referee for their comments.

References

[1] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), 1991.

[2] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. CPM'94*, LNCS 807, pages 198–212, 1994.

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

[4] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. SIGMOD'97*, pages 357–368, 1997. Sigmod Record 26(2).

[5] S. Brin. Near neighbor search in large metric spaces. In *Proc. VLDB'95*, pages 574–584, 1995.

[6] W. Burkhard and R. Keller. Some approaches to best-match file searching. *CACM*, 16(4):230–236, 1973.

[7] E. Chávez and J. Marroquín. Proximity queries in metric spaces. In *Proc. WSP'97*, pages 21–36. Carleton University Press, 1997.

[8] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. SPIRE'99*, 1999.

[9] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD'84*, pages 47–57, 1984.

[10] L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbor classifier in metric spaces. *Patt. Recog. Lett.*, 17:731–739, 1996.

[11] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear preprocessing-time and memory requirements. *Patt. Recog. Lett.*, 15:9–17, 1994.

[12] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. PAMI*, 19(9):989–1003, 1997.

[13] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *IPL*, 40:175–179, 1991.

[14] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Patt. Recog. Lett.*, 4:145–157, 1986.

[15] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. SODA'93*, pages 311–321, 1993.

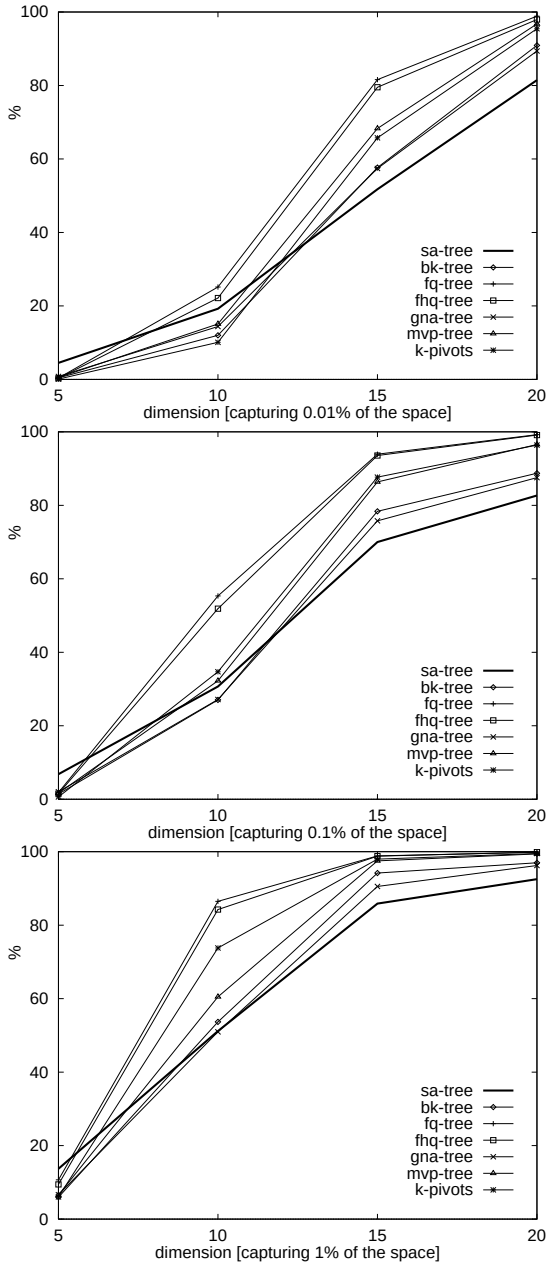

Figure 6. Set fraction traversed with the different data structures, for $n = 250,000$.