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, i.e., getting closer and closer to it, rather than the classical divide-and-conquer approach of other data structures. We analyze our method and show that the number of distance evaluations to search among n objects is sublinear. 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. As a practical advantage, our data structure is one of the few that do not need to tune parameters, which makes it appealing for use by non-experts.

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 instead for similar objects, e.g. databases storing images, fingerprints or audio clips); text retrieval (where we look for words and phrases in a text database allowing a small number of typographical or spelling errors, or we look for documents which are similar to a given query or document); 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); 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 nonnegative *distance function* $d : U \times U \longrightarrow R^+$ defined among them. This distance satisfies the

*This work has been supported in part by Fondecyt grant 1-000929.

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. 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. This is, $\{x \in S \ , \ d(x,q) \leq r\}$.

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

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 their distance belongs to the Minkowski L_r family: $L_r = (\sum_{1 \leq i \leq d} |x_i - y_i|^r)^{1/r}$. The best known special cases are $r = 1$ (Manhattan distance), $r = 2$ (Euclidean distance) and $r = \infty$ (maximum distance). This last distance deserves an explicit formula: $L_\infty = \max_{1 \leq i \leq d} |x_i - y_i|$.

There are effective methods to search on d-dimensional spaces, such as kd-trees [Ben79, Ben75] or R-trees [Gut84]. However, for roughly 20 dimensions or more those structures cease to work well. We focus in this paper 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_r distances 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 [Bri95, CM97, CNBYM01]. 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. Some are tailored to continuous and others to discrete distance functions. All those structures work on the basis of discarding elements using the triangular inequality (there is no way to avoid a single distance evaluation if the triangular inequality does not hold).

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 different 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 dimensional spaces or queries with large radii.

There are many interesting applications whose space is high dimensional. On the other hand, one can argue that large radii may return too many results if one considers the particular case of the end user of a database, so all the interesting cases are of very small selectivity. However, there are lots of applications where it is necessary to retrieve a relatively large portion of the database. Even in data retrieval applications, the similarity criterion may be just a first step from where we obtain a set of candidates which are further filtered with more complex criteria before delivering a small set of answers to the final user. This is indeed the ranking method of many existing systems for textual information retrieval [BYRN99].

The *sa-tree*, unlike other data structures, does not have parameters to be tuned by the user of each application. This makes it very appealing as a general purpose data structure for metric searching, since any non-expert seeking for a tool to solve his/her particular problem can use it as a black box tool, without the need of understanding the complications of an area he/she is not interested in. Other data structures have many tuning parameters, hence requiring a big effort from the user in order to obtain an acceptable performance.

This work is organized as follows. In Section 2 we cover the main We have found only one antecedent of our idea in the literature: in [Cla99] they start with this concept of spatially approaching the query and end up with a probabilistic algorithm based on a randomized data structure for searching the nearest neighbor of the query. Our results are more general and rely on deterministic algorithms.

previous work. In Section 3 we present the ideal data structure and prove that it cannot be built. In Section 4 we propose the simplified structure. The structure is analyzed in Section 5. Section 6 shows experimental results verifying the analysis and comparing the structure against others. Some alternatives that permit incremental construction are discussed in Section 7. We draw our conclusions in Section 8. A partial and less mature earlier version of this work appeared in [Nav99].

2 Previous Work

Algorithms to search in general metric spaces can be divided in two large areas: pivot-based and clustering algorithms. (See [CNBYM01] 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 $(d(q,p_1)...d(q,p_k))$. Now, if for some pivot p_i it holds that $|d(q,p_i) - d(x,p_i)| > r$, then we know by the triangular inequality that $d(q,x) > r$ and therefore do not need to explicitly evaluate $d(x,p)$. All the other elements that cannot be eliminated using this rule are directly compared against the query.

Algorithms such as *aesa* [Vid86], *laesa* [MOV94], *spaghettis* and variants [CMBY99, NN97], *fq-trees* and variants [BYCMW94], and *fq-arrays* [CMN01], are almost direct implementations of

this idea, and differ basically in their extra structure used to reduce the CPU cost of finding the candidate points, but not in the number of distance evaluations performed.

There are a number of tree-like data structures that use this idea in a more indirect way: they select a pivot as the root of the tree and divide the space according to the distances to the root. One slice corresponds to each subtree (the number and width of the slices differs across the strategies). At each subtree, a new pivot is selected and so on. The search performs a backtrack on the tree using the triangular inequality to prune subtrees, that is, if a is the tree root and b the root of a children corresponding to $d(a,b) \in [x_1, x_2]$, then we can avoid entering in the subtree of b whenever $[d(q,a) - r, d(q,a) + r]$ has no intersection with $[x_1, x_2]$. Data structures using this idea are the *bk-tree* and its variants [BK73, Sha77], *metric trees* [Uhl91b], *tlaesa* [MOC96], and *vp-trees* and variants [Yia93, BO97, Yia00].

Clustering 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 *Voronoi area*, where we select a set of centers and put each other point inside the zone of its closest center. The areas are limited by hyperplanes and the zones are analogous to Voronoi regions in vector spaces. Let $\{c_1 \ldots c_m\}$ be the set of centers. At query time we evaluate $(d(q,c_1), \ldots, d(q,c_m))$, choose the closest center c and discard every zone whose center c_i satisfies $d(q,c_i) > d(q,c) + 2r$, as its Voronoi area cannot have intersection 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.

The techniques can be combined. Some using only hyperplanes are the *gh-trees* and variants [Uhl91b, NVZ92], and *Voronoi trees* [DN87, Nol89]. Some using only covering radii are the *M-trees* [CPZ97] and *lists of clusters* [CN00]. One using both criteria is the *gna-tree* [Bri95].

To answer 1-NN queries, we simulate a range query with a radius that is initially $r = \infty$, and reduce r as we find closer and closer elements to q. At the end, we have in r the distance to the closest elements and have seen them all. Unlike a range query, we are now interested in quickly finding close elements in order to reduce r as early as possible, so there are a number of heuristics to achieve this. One of the most interesting is proposed in [Uhl91a] for metric trees, where the subtrees are stored in a priority queue in a heuristically promising ordering. The traversal is more general than a backtracking. Each time we process the most promising subtree, we may add its children to the priority queue. At some point we can preempt the search using a cutoff criterion given by the triangular inequality.

k-NN queries are handled as a generalization of 1-NN queries. Instead of a closest element, a priority queue of the k closest elements known is maintained. The r value is now that of the element among the k current candidates which is farthest from q. Each new candidate is inserted in the heap and may displace the farthest one out of the queue (hence reducing r for the rest of the algorithm).

Note that all the previous work aims at dividing the database, inheriting from the classical divide-and-conquer ideas of searching typical data (e.g. binary search trees). We propose in this

paper a new approach which is specific of spatial searching. Rather than dividing the set of candidates along the search, we try to start at some point in the space and get closer to the query q in a spatial sense.

3 The Spatial Approximation Approach

We concentrate in this section on 1-NN queries (at the end we will solve all types of queries). Instead of the known algorithms to solve proximity 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.

This approximation is performed only via "neighbors". Each element $a \in S$ has a set of neighbors $N(a)$, and we are allowed to move directly only to neighbors. The natural structure to represent this restriction is a directed graph where the nodes are the elements of S and they have direct edges to their neighbors. That is, 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 element 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 last node! For this reason and also to minimize the space required by the structure, we prefer the graph which has the *least* possible number of edges and still allows answering correctly all queries. This graph $G = (S, \{(a,b),\ a \in S,\ b \in N(a)\})$ 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. 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 [Aur91][1]. The answer to the query q is the element $a \in S$ 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 $a \in S$ with all its "Voronoi neighbors" (i.e. elements of S whose Voronoi area share 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).

[1] The proper name in a general metric space is "Dirichlet domain" [Bri95].

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 allow the search 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, then the only zone where the query would not move away from a will be exactly the area where a is the closest neighbor.

Therefore, in a vector space, the minimal graph we seek corresponds to the classical Delaunay triangulation (a graph where the elements which are Voronoi neighbors are connected). The Delaunay graph, generalized to arbitrary spaces, would be therefore the ideal answer in terms of space complexity, and it should permit fast searching too. Figure 1 shows an example.

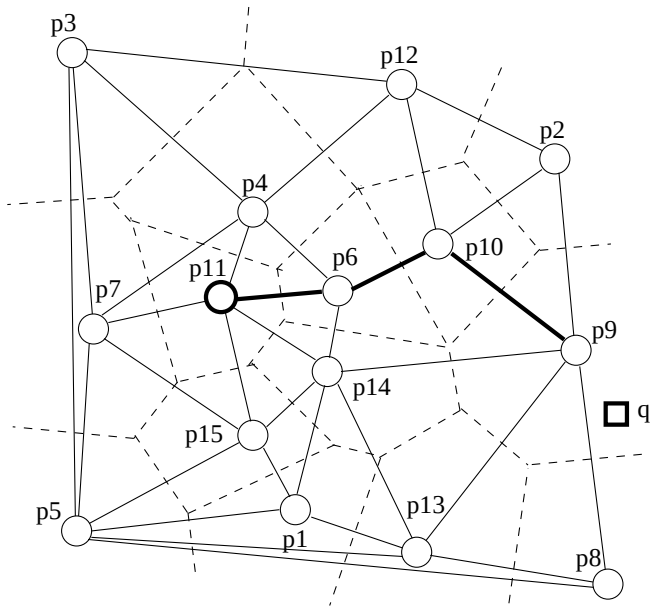


Figure 1: An example of the search process with a Delaunay graph (solid edges) corresponding to a Voronoi partition (areas delimited by dashed lines). We start from p_{11} and reach p_9, the node closest to q, moving always to neighbors closer and closer to q.

Unfortunately, it is not possible to compute the Delaunay 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 graphs. Moreover, it is not possible to prove that a single edge from any node a to b is not in the Delaunay graph. Therefore, the only superset of the Delaunay 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 Delaunay 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 every other $y \in S$. 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, we have that 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 2). This argument can be repeated for any pair $a, b \in S$.

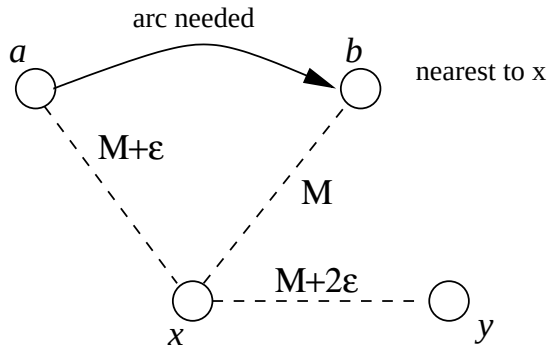


Figure 2: Illustration of the theorem.

4 The Spatial Approximation Tree

We make two crucial simplifications to the general idea so as to achieve a feasible solution. The resulting simplification answers only a reduced set of queries, namely 1-NN queries for $q \in S$, which is no more than exact searching. However, we show later (Section 4.2) how to combine the spatial approximation approach with backtracking so as to answer any query $q \in U$ (not only $q \in S$), for both range queries and nearest neighbor queries.

(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.

4.1 Construction Process

We select a random 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) $\quad \forall x \in S, \ x \in N(a) \Leftrightarrow \forall y \in N(a) - \{x\}, \ d(x,y) > d(x,a)$.

That is, the neighbors of a form a set such that any neighbor is closer to a than to any other neighbor. The "only if" ($\Leftarrow$) part of the definition guarantees that if we can get closer to any $b \in S$ then an element in $N(a)$ is closer to b than a, because we put as direct neighbors all those elements that are not closer to another neighbor. The "if" part ($\Rightarrow$) aims at puting as few neighbors as possible.

Notice that the set $N(a)$ is defined in terms of itself in a non-trivial way and that multiple solutions fit the definition. For example, if a is far from b and c and these are close to each other, then both $N(a) = \{b\}$ and $N(a) = \{c\}$ satisfy the definition.

Finding the smallest possible set $N(a)$ seems to be a nontrivial combinatorial optimization problem, since by including an element we need to take out others (this happens between b and c in the example of the previous paragraph). However, simple heuristics which add more neighbors than necessary work well. We begin with the initial node a and its "bag" holding all the rest of S. We first sort the bag 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 this point we have a suitable set of neighbors. Note that Condition 2 is satisfied thanks to the fact that we have considered the elements in order of increasing distance to a. The "only if" part of the Condition is clearly satisfied because any element satisfying the clause on the right is inserted in $N(a)$. The "if" part is more delicate. Let $x \neq y \in N(a)$. If y is closer to a than x then y was considered first. Our construction algorithm guarantees that if we inserted x in $N(a)$ then $d(x,a) < d(x,y)$. If, on the other hand, x is closer to a than y, then $d(y,x) > d(y,a) \geq d(x,a)$ (that is, a neighbor cannot be removed by a new neighbor inserted later).

We now must decide in which neighbor's bag we put the rest of the nodes. We put each node not in $\{a\} \cup N(a)$ in the bag of its closest element of $N(a)$ (*best-fit* strategy). Observe that this requires a second pass 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 bag. 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 nearest neighbor queries. The reason why this works is that, at search time, we repeat exactly what happened with q during the construction process (i.e. we enter into the subtree of the neighbor closest to q), until we reach q. This is is because q is present in the tree, i.e., we are doing an exact search.

Finally, we save some comparisons at search time by storing at each node a its covering radius, i.e. the maximum distance $R(a)$ between a and any element in the subtree rooted by a. The way to use this information is made clear in Section 4.2.

Figure 3 depicts the construction process.

4.2 Range 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 range queries with radius r.

The key observation is that, even if $q \notin S$, 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 since $d(q,q') \leq r$, we can obtain from q some distance information regarding q': by the triangular inequality it holds that for any $x \in U$, $d(x,q) - r \leq d(x,q') \leq d(x,q) + r$.

Therefore, instead of simply going to the closest neighbor, we first determine the closest neighbor c of q among $\{a\} \cup N(a)$. 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 by at most r at any distance evaluation, so it could have been inserted inside such b nodes. In the process, we report all the nodes q' we have seen which are close enough to q.

Moreover, notice that, in an exact search for a $q \in S$, the distances between q and the nodes we traverse gets reduced as we step down the tree. That is,

```
BuildTree (Node a, Set of nodes S)

N(a) ← ∅        /* neighbors of a */
R(a) ← 0        /* covering radius */
Sort S by distance to a (closer first)
for v ∈ S do
   R(a) ← max(R(a), d(v,a))
   if ∀b ∈ N(a), d(v,a) < d(v,b) then N(a) ← N(a) ∪ {v}
for b ∈ N(a) do S(b) ← ∅      /* subtrees */
for v ∈ S − N(a) do
   Let c ∈ N(a) be the one minimizing d(v,c)
   S(c) ← S(c) ∪ {v}
for b ∈ N(a) do BuildTree (b, S(b))      /* build subtrees */
```

Figure 3: Algorithm to build the *sa-tree*. It is firstly invoked as `BuildTree`$(a, S - \{a\})$ where a is a random element of the set S. Note that, except for the first level of the recursion, we already know all the distances $d(v,a)$ for every $v \in S$ and hence do not need to recompute them. Similarly, $d(v,c)$ at line 9 is already known from line 6. The information stored by the data structure is the root a and the $N()$ and $R()$ values of all the nodes.

Observation 1: Let $a, b, c \in S$ such that b descends from a and c descends from b in the tree. Then $d(c,b) \leq d(c,a)$.

The same happens, allowing a tolerance of $2r$, with a range search with radius r. That is, for any b in the path from a to q' it holds that $d(q',b) \leq d(q',a)$, so $d(q,b) \leq d(q,a) + 2r$. Hence, while at first we need to enter into all the neighbors $b \in N(a)$ such that $d(q,b) - d(q,c) \leq 2r$, when we enter into those b the tolerance is not $2r$ anymore but it gets reduced to $2r - (d(q,b) - d(q,c))$.

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. This is the price of not being able to build a true spatial approximation graph. Figure 4 illustrates the search process.

The covering radius $R(a)$ is used to reduce the search cost as follows. We never enter into a subtree such that $d(q,a) > R(a) + r$, since there cannot be useful elements there. Figure 5 depicts the algorithm.

4.3 Nearest Neighbor Searching

We can also perform nearest neighbor searching by simulating a range search where the search radius is reduced as we get more and more information. To solve 1-NN queries, we start searching with $r = \infty$, and reduce r each time a new comparison is performed which gives a distance smaller than r. We finally report the closest element seen along all the search. For k-NN queries we store all the time a priority queue with the k closest elements to q we have seen up to now. The radius r is the distance between q and its farthest candidate in the queue (∞ if we still have less than k

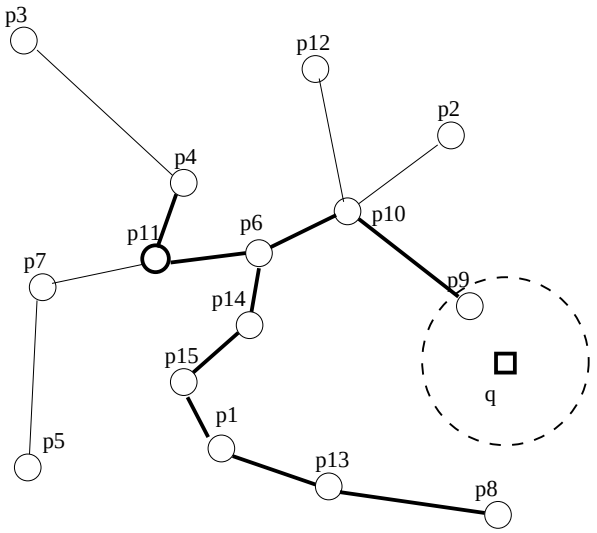


Figure 4: An example of the search process, starting from p_{11} (tree root). Only p_9 is in the result, but all the bold edges are traversed.

candidates). Each time a new candidate appears we insert it into the queue, which may displace another element and hence reduce r. At the end, the queue contains the k closest elements to q (recall Section 2).

In a normal range search with fixed r, the order in which we backtrack in the tree is unimportant. This is not the case now, as we would like to quickly find elements close to q so as to reduce r early. A general idea proposed in [Uhl91a] can be adapted to our data structure. We have a priority queue of subtrees, most promising first. Initially, we insert the whole *sa-tree* in the data structure. Iteratively, we extract the most promising subtree, process its root, and insert all its subtrees in the queue. This is repeated until the queue gets empty or its most promising subtree can be discarded (i.e., its promise value is bad enough).

How promising is a subtree rooted at a and how can it be discarded is measured, in our case,

```
RangeSearch (Node a, Query q, Radius r, Tolerance t)

if d(a,q) ≤ r then Report a
if d(a,q) ≤ R(a) + r then
   mind ← min{d(c,q), c ∈ {a} ∪ N(a)}
   for b ∈ N(a) do
      if d(b,q) − mind ≤ t then RangeSearch (b,q,r,t − (d(b,q) − mind))
```

Figure 5: Algorithm to search q with radius r in a *sa-tree* under the best-fit strategy. It is firstly invoked as RangeSearch(a,q,r,$2r$), where a is the root of the tree. Notice that in the recursive invocations $d(a,q)$ is already computed.

in two possible ways:

1. By the lower bound to the distance between q and an element in the subtree, namely $d(q,b) - R(b) \leq r$.

2. By the fact that we find the closest neighbor c and then enter into any other neighbor b such that $d(q,b) - d(q,c) \leq 2r$, i.e., $(d(q,b) - d(q,c))/2 \leq r$. As in fact the tolerance is $2r$ at the beginning and it gets reduced across the search, we add up the reductions. That is, let t be the accumulated differences between the distances to the selected neighbor and the closest neighbor. Then we continue the search as long as $t + (d(q,b) - d(q,c)) \leq 2r$. Hence the new limit is $(t + d(q,b) - d(q,c))/2$

Since r gets reduced along the search, we can store together with b the two values, $d(q,b) - R(b)$ and $(t + d(q,b) - d(q,c))/2$, and avoid entering into those where any of these two value is larger than r. From these two criteria, we must choose one as our "primary" criterion, so as to sort the queue by it and stop the whole process when its value is larger than r. The other can be used as a "secondary" criterion, just to avoid entering some subtrees but not sorting by it and hence not using it as a global stopping criterion. Our experiments show that the second option is slightly better as the primary criterion. Figure 6 depicts the algorithm.

```
NN-Search (Tree a, Query q, Neighbors wanted k)

Q ← {(a, 0, d(q,a) − R(a))}     /* promising subtrees */
A ← ∅      /* best answer so far */
r ← ∞
while Q is not empty
   (b, t, d) ← element in Q with smallest t  ,  Q ← Q − {(b, t, d)}
   if t > r then Return the answer A     /* global stopping criterion */
   if d ≤ r then     /* secondary criterion */
      A ← A ∪ {(b, d + R(b))}
      if |A| = k + 1 then
         (c, d') ← element in A with largest d'  ,  A ← A − {(c, d')}
      if |A| = k then
         (c, d') ← element in A with largest d'  ,  r ← d'
      c ← closest to q among N(b)
      for each v ∈ N(b), Q ← Q ∪ (v, (t + d(q,v) − d(q,c))/2, d(q,v) − R(v))
Return the answer A
```

Figure 6: Algorithm to search the k nearest neighbors of q in a *sa-tree*. A and Q are priority queues of pairs *(subtree, distance1)* and *(element, distance1, distance2)*, respectively, delivering the smallest *distance*1 first.

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 bag before selecting neighbors (the results are pessimistic in this sense, since it looks as if we had more neighbors). As seen in the experiments however, the fitting with real data is excellent. This analysis is done for a continuous distance function, although adapting it to the discrete case is immediate.

Our results can be summarized as follows. The *sa-tree* needs linear space $O(n)$, reasonable construction time $O(n \log^2 n / \log\log n)$ and sublinear search time $O(n^{1-\Theta(1/\log\log n)})$ in high dimensions and $O(n^\alpha)$ $(0 < \alpha < 1)$ in low dimensions.

5.1 Construction Cost and Tree Shape

Let us consider first the construction process. 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 simply $1/2$ because the situation is symmetric: if we draw a hyperplane at the same distance from a and b, then c can equally lie at either side of the hyperplane.

If j neighbors are already present, the probability that we add another neighbor is that of being closer to a than to any neighbor. If we assume that all the hyperplanes are independent, then this probability is $1/2^j$. This is a simplification for several reasons. First, the neighbors are chosen from a's part of the hyperplane, never from the part of the hyperplane of another neighbor (which is the same to say that neighbors are closer to a than to each other). Second, in low dimensions it is not possible to set up too many different hyperplanes because the space becomes filled.

Since each attempt to obtain the $(j+1)$-th neighbor has a probability of success of $1/2^j$, we have a hypergeometric process with mean 2^j. The total number of attempts to obtain N neighbors is a sum of hypergeometric variables with means 2^0, 2^1, and so on. Since the mean commutes with the sum, the average number of attempts necessary to obtain N neighbors is $\sum_{j=0}^{N-1} 2^j = 2^N - 1$. Inverting, we have that with n elements (attempt) we obtain on average $\log_2(n+1)$ neighbors. This is a lower bound because we are taking the inverse of the average instead of the average of the inverse, and the inverse function is concave down. It is possible, although tedious, to prove that in fact the average number of neighbors is

$$N(n) \quad = \quad \Theta(\log n)$$

under our simplifications stated above. The constant is between 1.00 and 3.28. Recall also that there is a constant part that should be specially relevant in low dimensions. However, for our analysis $\Theta(\log n)$ suffices.

This allows determining 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

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

The construction cost is as follows (in terms of distance evaluations). The bag 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 bag. Then, all neighbors are recursively built.

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

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

5.2 Query Time

We analyze the search time now. Since we enter into many neighbors, we must determine which is the amount of backtracking performed. Let $D_0, \ldots, D_j$ random variables corresponding to the distances $D_0 = d(a,q)$ and $D_i = d(v_i, q)$, where v_i is the i-th neighbor of q. Let us call $f(x)$ the probability density function of $D_i - \min(D_0, \ldots, D_j)$, for any D_i corresponding to a neighbor. It is clear that $f(x) > 0$ only when $x \geq 0$. We also call $F(y) = \int_0^y f(y)dy$ its cumulative distribution.

Now, we will enter into neighbor i whenever $X = D_i - \min(D_0, \ldots, D_j) \leq t$, where t is the tolerance (initially $2r$). The probability of such a fact is $F(t)$. Moreover, if $X \leq t$ we enter into the neighbor with tolerance $t - X$.

There are $\Theta(\log n)$ neighbors, and we enter into each one with the same probability. The size of the set inside a neighbor is $O(n/\log n)$. Hence if we call $T(n,t)$ the search cost with n elements and tolerance t (initially $t = 2r$), then the following recurrence holds

$$T(n,t) \quad = \quad \log n \; + \; \log n \int_0^t f(x) T\left(\frac{n}{\log n}, t - x\right) \; dx$$

which is hard to solve exactly. We set the inductive hypothesis of $T(n,t) = cn^\alpha g(t)$ for $0 < \alpha < 1$ and an increasing function $g(t)$. Trying to prove the inductive thesis yields

$$T(n,2r) \quad = \quad O\left(n^{1 - \frac{g(2r)}{\int_0^{2r} f(x)g(2r-x)dx} \frac{1}{\log\log n}} \; g(2r)\right) \quad = \quad O\left(n^{1-\Theta(1/\log\log n)}\right)$$

for any function $g(t)$. The complexity in terms of n is more or less settled, but finding it in terms of the search radius involves obtaining the optimal $g(t)$, i.e. the one minimizing the constant

$$\frac{1}{g(2r)} \int_0^{2r} f(x)g(2r-x) \; dx$$

which is not trivial. Fast growing functions work better, for example $g(t) = s^t$ yields $1/\int_0^t f(x)s^{-x}dx$, independent of r, but it is not clear which is the optimum s.

On the other hand, note that when the intrinsic dimension is small compared to $O(\log n)$, $N(n)$ is closer to a constant because there cannot be too many neighbors. In this case the analysis yields $O(n^\alpha)$ for constant $0 < \alpha < 1$. We prefer, however, to stick to the more conservative complexity.

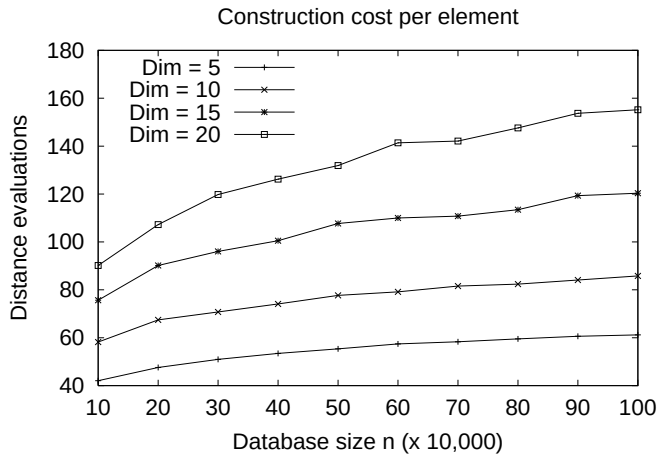
6 Experimental Results

We have tested our *sa-tree* and previous work on a synthetic set of random points in a d-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 d-dimensional space, their real dimension can be lower). Our tests use the Euclidean distance (L_2) and four different dimensions: 5, 10, 15 and 20. For each dimension, we generated 10 incremental groups of data sets, from $n = 10,000$ to $n = 100,000$ elements. Later, when comparing our data structure against others, we show some real metric spaces too.

The results were averaged over 100 index constructions (recall that the construction algorithm is randomized) and 100 queries run over each index. Hence, each data point about the structure itself or its construction is an average over 100 iterations, while each data point about query costs is an average over 10,000 iterations.

6.1 Construction Cost and Tree Shape

Our first experiment aims at measuring the construction cost of the *sa-tree*, as well as the shape of the resulting tree. Figure 7 shows how the cost grows as n increases. We show the number of evaluations per element, which according to the analysis is $O(\log^2 / \log\log n)$. A least squares estimation shows an excellent fitting with this analysis (better for low dimensions, as in higher dimension there is more variance), with an accompanying constant factor that seems to depend linearly on the dimension.

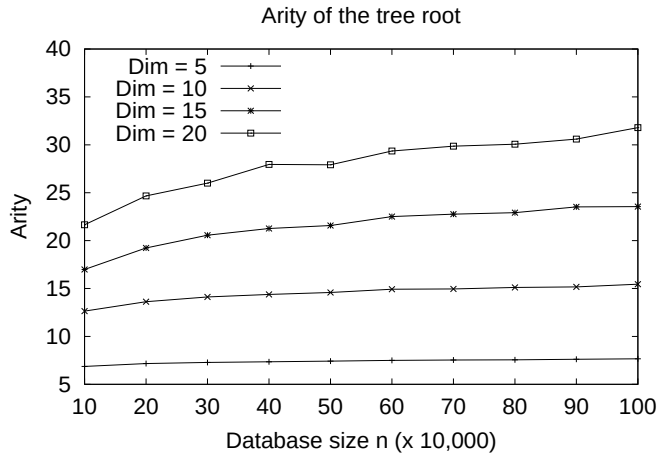


Dim	Approximation	Error
5	$1.126\frac{\ln(n)^2}{\ln\ln n}$	0.007
10	$1.569\frac{\ln(n)^2}{\ln\ln n}$	0.008
15	$2.155\frac{\ln(n)^2}{\ln\ln n}$	0.025
20	$2.722\frac{\ln(n)^2}{\ln\ln n}$	0.049

Figure 7: Construction cost, measured in number of distance evaluations per element. The cost grows with n and with the dimension of the database. On the right, the formula obtained by least squares and the relative error.

We consider now the arity of the tree root. The analytical prediction, $O(\log n)$, fits again very

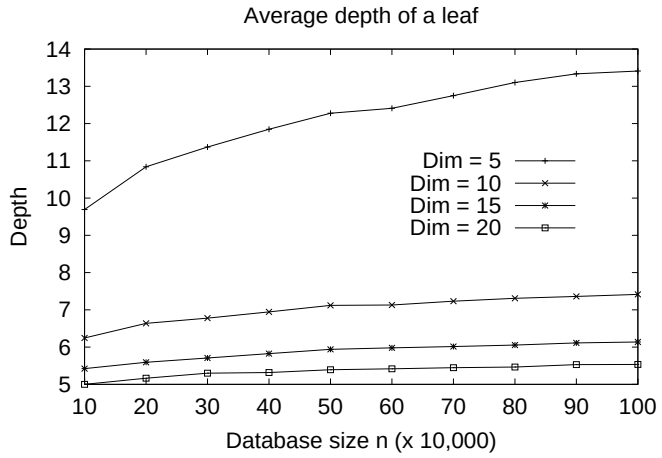
well with the experiments. Using a model of the form $a + b \ln n$ we obtain relative errors below 1%. The constant b seems to grow exponentially with the dimension. The results are shown in Figure 8.



Dim	Approximation	Error
5	$3.892 + 0.327 \ln n$	0.002
10	$2.126 + 1.154 \ln n$	0.005
15	$-8.965 + 2.481 \ln n$	0.007
20	$-16.971 + 4.194 \ln n$	0.009

Figure 8: Arity of the tree root. It grows with n and with the dimension of the database. On the right, the formula obtained by least squares and the relative error.

Let us now focus on the average leaf depth of the trees. The analysis predicts $O(\log n / \log \log n)$. Again, we have obtained a very good approximation, with relative error well below 1%, with the model $a + b \ln n / \ln \ln n$. This time the constant b decreases with the dimension. Figure 9 shows the results.



Dim	Approximation	Error
5	$-17.857 + 6.630 \frac{\ln n}{\ln \ln n}$	0.006
10	$-2.283 + 2.058 \frac{\ln n}{\ln \ln n}$	0.003
15	$-0.082 + 1.319 \frac{\ln n}{\ln \ln n}$	0.003
20	$1.136 + 0.934 \frac{\ln n}{\ln \ln n}$	0.002

Figure 9: Average leaf depth in the tree. It grows with n and decreases with the dimension of the database. On the right, the formula obtained by least squares and the relative error.

The results show that our analysis is quite accurate, despite the simplifications made. We have been able to predict how the tree behaves as a function of the database size n. However, the experiments give additional information on an aspect that we could not capture analytically,

namely the behavior of the trees as the dimension of the set grows. As the experiments show, the trees get fatter and shorter for higher dimensions, and consequently they are harder to build.

This phenomenon is interesting because it shows how the *sa-tree* adapts itself to the dimension of the data without need of external tuning, a feature that very few data structures posess. Other articles, such as that of *gna-trees* [Bri95], suggest to use a larger arity for higher dimensions and to reduce the arity in lower levels of the tree, but all this occurs naturally in *sa-trees*.

6.2 Querying Cost

We consider now the cost of searching the index. We have tried both range and nearest neighbor searching. For range searching, we have selected manually the radii that recover 0.01%, 0.1% and 1% of the set. For nearest neighbor searching, we have directly requested to retrieve that number of elements. As our algorithm for nearest neighbor searching is a range search algorithm that adjusts the radius as it gets more and more information on the set, we expect that nearest neighbor searching takes more time than range searching in order to retrieve the same amount of elements. How close is the time with respect to range searching gives us an idea of how good is the heuristic.

Figure 10 shows the results, in terms of percentage of the set traversed for a query. Several observations are in order. First, note that the sublinearity is clear. Moreover, our analysis holds with extreme accuracy using the model $an^{1-b/\ln\ln n}$ (the relative error is 0.5% at most). Second, the results worsen fast as the dimension or the search radius grows, which is reflected in a reduction of the constant b. Third, note that the nearest neighbor search algorithm has a cost close but sometimes noticeable higher than that of range searching.

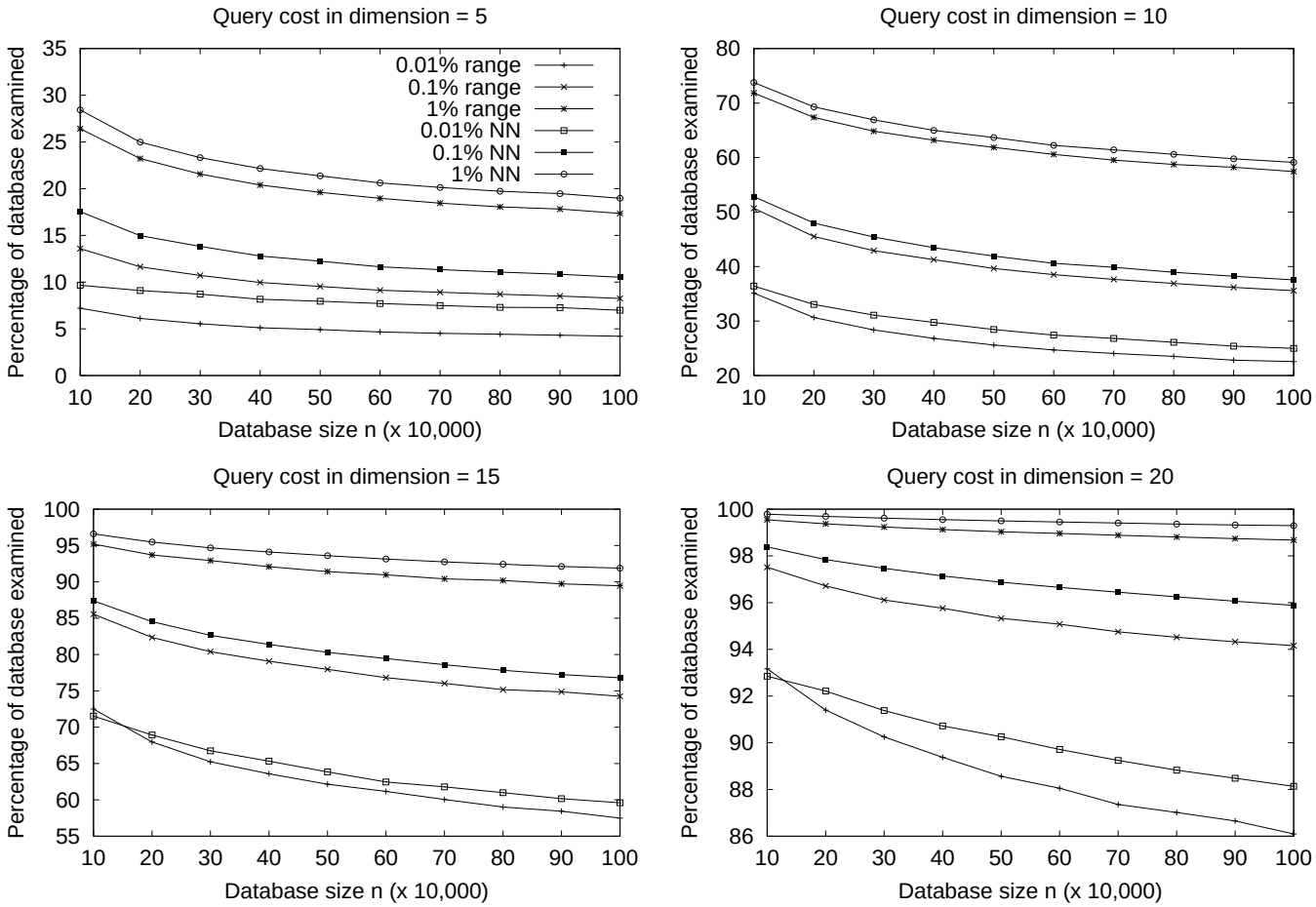
6.3 Comparison against Others

Finally, we compare our *sa-trees* against other data structures. This time we fix $n = 100,000$ and show how the results change with the dimension. We also show the case of real-world metric spaces.

There are too many proposals to compare them all, so we have selected a small set of good representatives. Some structures do behave better than our *sa-tree*, but at the expense of impractical amounts of memory (e.g. *aesa* [Vid86] needs $O(n^2)$ space) or construction time (e.g. *aesa* [Vid86] and the *list of clusters* [CN00] need $O(n^2)$ construction time). To make a fair comparison we fix the amount of memory or construction time that we permit and limit these structures accordingly. The structures chosen are:

Pivot(s): is a generic pivoting algorithm, where we limit the amount of space permitted to s times that of our *sa-tree*. A reasonable implementation shows that our data structure takes space equivalent to storing 4 pivots per element of the set. Hence Pivot(s) is equivalent to using $k = 4s$ pivots.

The specific algorithm consists of executing the first k steps of *aesa*, i.e. choosing a pivot p from the remaining set of elements and discarding every candidate element x such that $|d(q,x) - d(q,p)| > r$. This is better than fixing the k pivots in advance as done by many pivoting algorithms, because it is well known that better results are obtained by choosing the pivots from the remaining set. Some tree schemes permit adapting the pivot to the remaining set, at the cost of not using all the information given by their distances. So in fact we are



Dimension	range 0.01%	range 0.1%	range 1%
5	$3.801n^{1-0.957/\ln\ln n}$ (.005)	$5.141n^{1-0.877/\ln\ln n}$ (.004)	$5.726n^{1-0.742/\ln\ln n}$ (.002)
10	$9.392n^{1-0.792/\ln\ln n}$ (.002)	$6.851n^{1-0.627/\ln\ln n}$ (.002)	$3.748n^{1-0.397/\ln\ln n}$ (.002)
15	$3.939n^{1-0.407/\ln\ln n}$ (.003)	$2.475n^{1-0.255/\ln\ln n}$ (.002)	$1.522n^{1-0.112/\ln\ln n}$ (.002)
20	$1.674n^{1-0.140/\ln\ln n}$ (.002)	$1.272n^{1-0.064/\ln\ln n}$ (.001)	$1.063n^{1-0.016/\ln\ln n}$ (.000)

Figure 10: Percentage of the set traversed when searching using the *sa-tree*. Each plot considers a different dimension, showing range and nearest neighbor queries that retrieve 0.01%, 0.1% and 1% of the database. On the bottom, least squares estimations for the range queries, with the relative error in parenthesis.

simulating an algorithm which has the best of both worlds: we assume that we need only the space for k fixed pivots, that we can use all the information they yield, and that we are able to choose those pivots at query time and yet have all the $d(p_i, x)$ precomputed.

Clusters(t): is the scheme proposed in [CN00]. This structure takes linear space and it is shown to behave better than *sa-trees* in high dimensions. However, for this to happen it is necessary to pay a quadratic construction cost, which is unrealistic even compared to our (relatively expensive) construction cost.

The data structure consists of a list of balls containing a center and the $m-1$ elements closest to it. For the second ball we exclude the elements of the first, and so on. At search time every center c_i is compared against q. Its ball is discarded if $d(q, c_i) - r > cr(c_i)$, otherwise it is exhaustively searched (we can stop traversing the list of centers if $d(q, c_i) + r \leq cr(c_i)$). The construction cost needs $n^2/(2m)$ distance evaluations and the optimum m is constant. For a fair comparison, the parameter t will indicate how many times was the construction cost of the *list of clusters* superior to that of the *sa-tree*. Given our constructions costs, this implies cluster sizes of $817/t$, $582/t$, $415/t$ and $322/t$ for dimensions 5, 10, 15 and 20, respectively.

Gna-tree: is the structure proposed in [Bri95]. A set of m centers is selected and the rest are sent to their Voronoi region. The structure is built recursively inside each region. The covering radius is used too. This structure uses linear space and a construction time close to ours, so we do not put a parameter on it. Rather, we choose manually the best m for each case, which turns out to be 4 for 5 and 10 dimensions and 16 for 15 and 20 dimensions.

Figure 11 shows a comparison between *sa-trees* and the idealized pivoting algorithms. As it can be seen, the *sa-tree* tolerates better higher dimensions or larger radii. A pivoting index using the same amount of memory as the *sa-tree* is faster only for 5 dimensions and a radius that retrieves less than 0.1% of the database. As the dimension or the search radius grows, pivoting algorithms need more and more memory in order to compete. In high dimensions or large search radii, pivoting algorithms cannot compete even when they take 16 times the amount of memory required by *sa-trees*.

Figure 12 shows a comparison between *sa-trees* and clustering algorithms. These algorithms tolerate better high dimensions and large search radii, with a growth rate similar to that of *sa-trees*. Our structure is better than *gna-trees* for more than 10 dimensions. *Lists of clusters*, on the other hand, need more and more times the construction time of *sa-trees* to beat them as the dimension or the search radii grows: 2 times in 5 dimensions, 4 times in 10 dimensions, 4 to 8 times in 15 dimensions and 8 times in 20 dimensions.

Finally, we show a couple of real life metric spaces. The first one is a dictionary of 86,061 Spanish words under the edit distance, defined as the number of character insertions, deletions and replacements needed to convert one string into the other. This distance is discrete and has many applications in text retrieval, signal processing and computational biology [Nav01]. The particular case of a dictionary is of interest in spelling applications.

The second metric spaces is that of documents under the cosine similarity measure [BYRN99]. We took 1,263 documents of about 1 Mb each from the TREC-3 collections [Har95], namely from the collections AP, DOE, FR, WSJ and ZIFF. We took the vocabulary of each document (considering

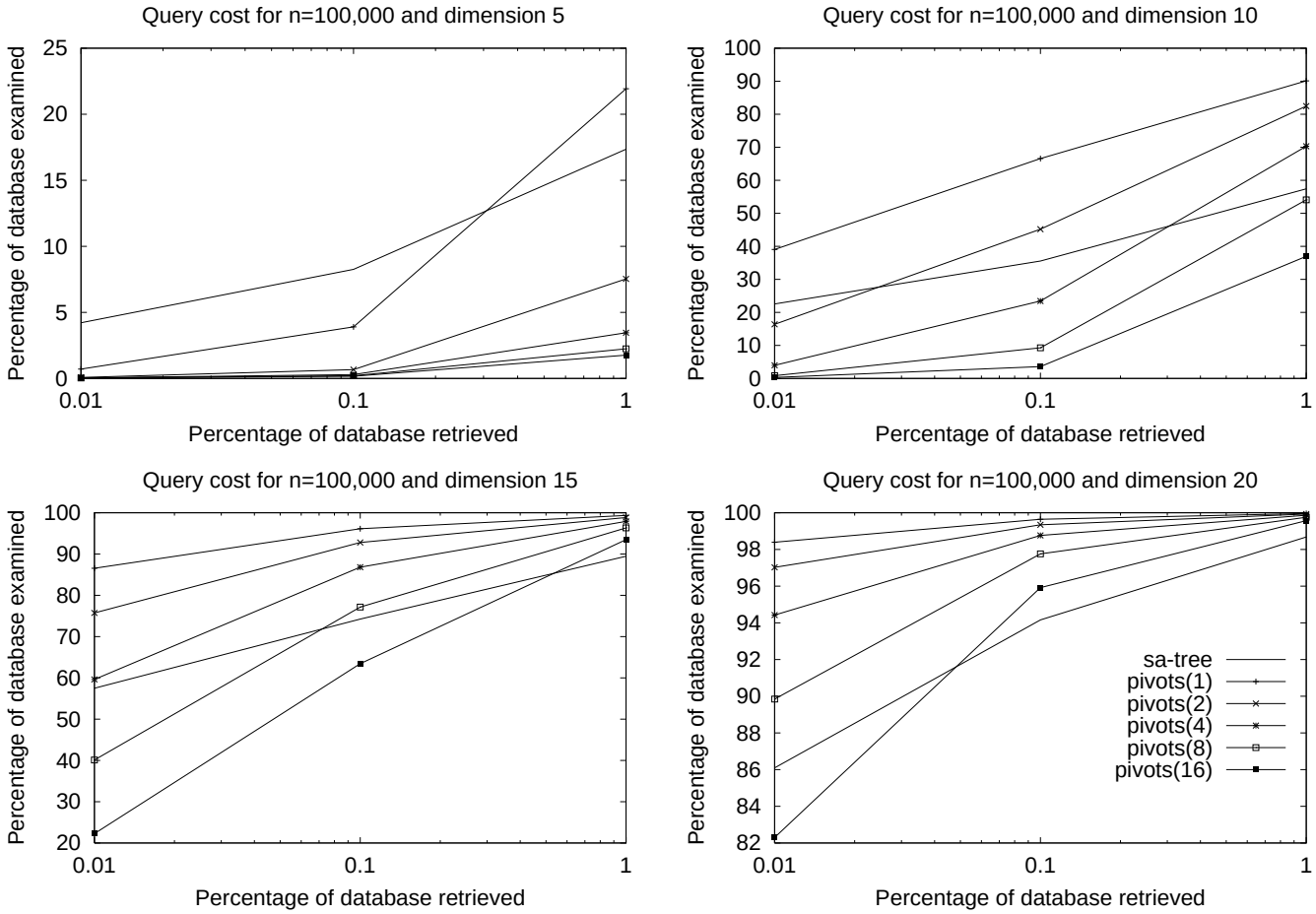


Figure 11: Comparison between the cost of range searching using the *sa-tree* and an indealized pivoting algorithm. We show each dimension separately and the cost for growing radius (i.e. queries that retrieve 0.01%, 0.1% and 1% of the database).

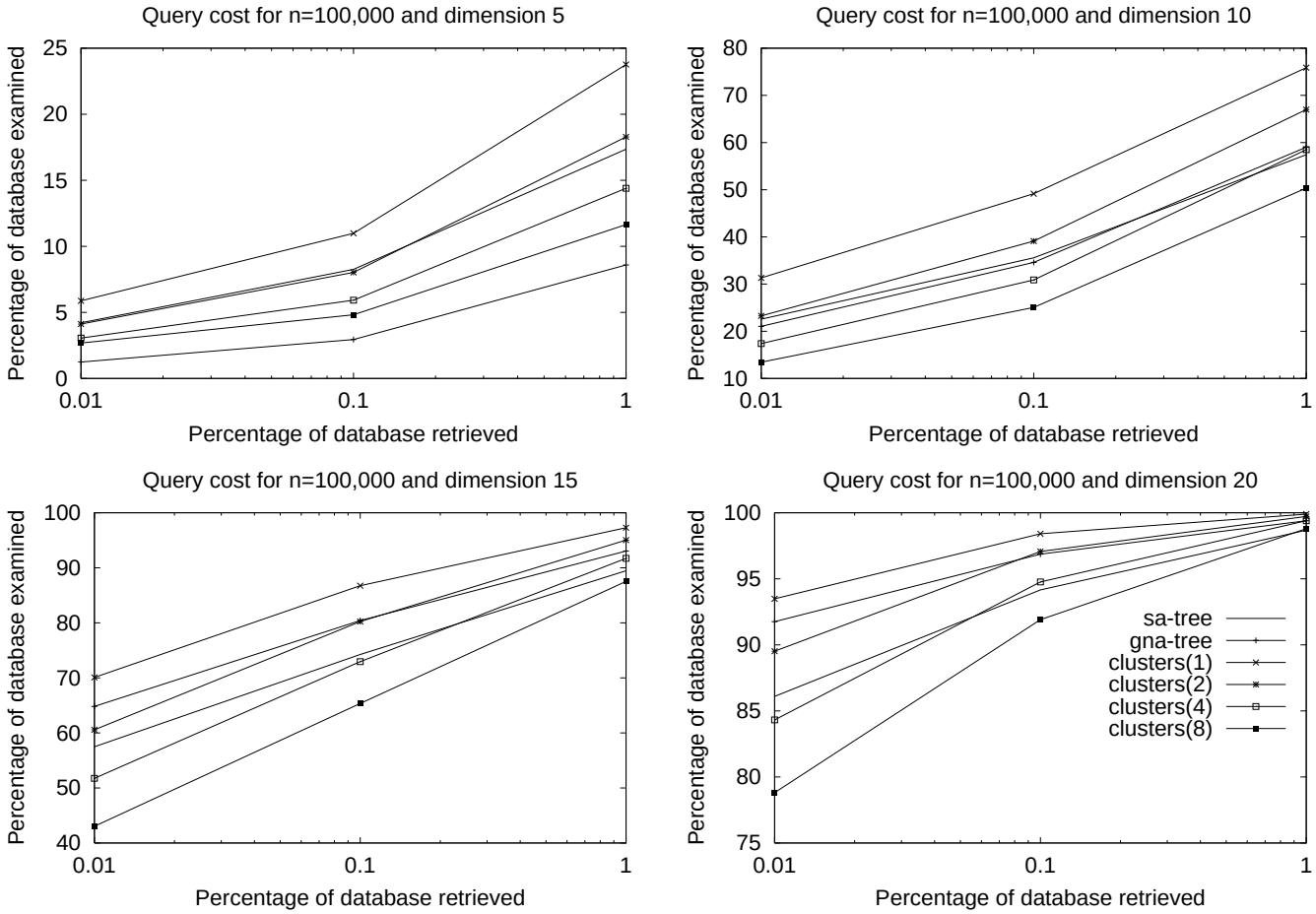


Figure 12: Comparison between the cost of range searching using the *sa-tree* and other clustering algorithms. We show each dimension separately and the cost for growing radius (i.e. queries that retrieve 0.01%, 0.1% and 1% of the database).

letters and digits and mapping them to lower case) and created for each document a vector where each vocabulary word is a coordinate and each document is a point. If the vocabulary word t_i appears f_{ij} in document d_j and it appears in n_i documents out of a total of N, then the value of document d_j at the coordinate t_i is $f_{ij} \ln(N/n_i)$. The distance is the angle between the vectors, i.e., the inverse cosine of the dot product between the two normalized vectors. This distance is of great use in Information Retrieval applications, and it is quite expensive to compute.

In the space of words under the edit distance, a byte suffices to store each distance, and hence the space taken by the *sa-tree* is equivalent to that of 10 pivots. The *gna-tree* gives its best results with arity 6. A *list of clusters* of equivalent construction cost uses clusters of size 594, as the *sa-tree* needed 72.43 comparisons per element. We show the results of searching with radius 1 to 4, which retrieved 0.00343%, 0.0286%, 0.245% and 1.460% of the set, respectively.

In the space of documents under the cosine similarity, the distance is a real number and hence we assume that the space taken by the *sa-tree* is equivalent to that of 4 pivots. The *gna-tree* gives its best results with arity 8. A *list of clusters* of equivalent construction cost uses clusters of size 14, as the *sa-tree* needed 45.05 comparisons per element (note that the clusters would be bigger if we had more elements in the set). We show the results of searching with radius retrieving 0.1%, 0.5% and 5% of the set. Each distance evaluation involves reading about 400 Kb from disk, so it is really expensive. For this reason we contented ourselves with building the indexes only once, and querying it 100 times.

Figure 13 shows the results. In the space of words, the *sa-tree* clearly outperforms the *gna-tree*. A pivoting algorithm needs 4 and even 8 times more space to beat *sa-trees* when the search radius becomes large (3 or 4). The lists of clusters needs to pay 4 times the construction cost of *sa-trees* in order to achieve better efficiency.

In the space of documents, the *sa-tree* outperforms again the *gna-tree*. A curious phenomenon occurs with the *list of clusters*: a cluster size of 14 seems to be too small, and we find the optimum at size 28 (with half the construction time of the *sa-tree*). However, our structure is superior for any choice of cluster size. (Previous datasets were too large to permit us reaching the optimal point; the smaller clusters were always better.) With respect to pivots, the *sa-tree* is quickly improved when searching with very small radii, but in order to beat it at searching with larger radii it is necessary to spend 8 times more memory.

As it can be seen, *sa-trees* provide a good tradeoff between space or construction cost. It is necessary to pay much more space or construction time to beat them when the dimension is high or the search radius is large. These are the most difficult cases in practice.

7 Incremental Construction

The *sa-tree* is a structure whose construction algorithm needs to know all the elements of S in advance. In particular, it is difficult to add new elements under the *best-fit* strategy once the tree is already built. Each time a new element is inserted, we must go down the tree by the closest neighbor until the new element must become a neighbor of the current node a. All the subtree rooted at a must be rebuilt from scratch, since some nodes that went into another neighbor could prefer now to get into the new neighbor.

Permitting the insertion of new elements into an already built *sa-tree* is the main issue in our

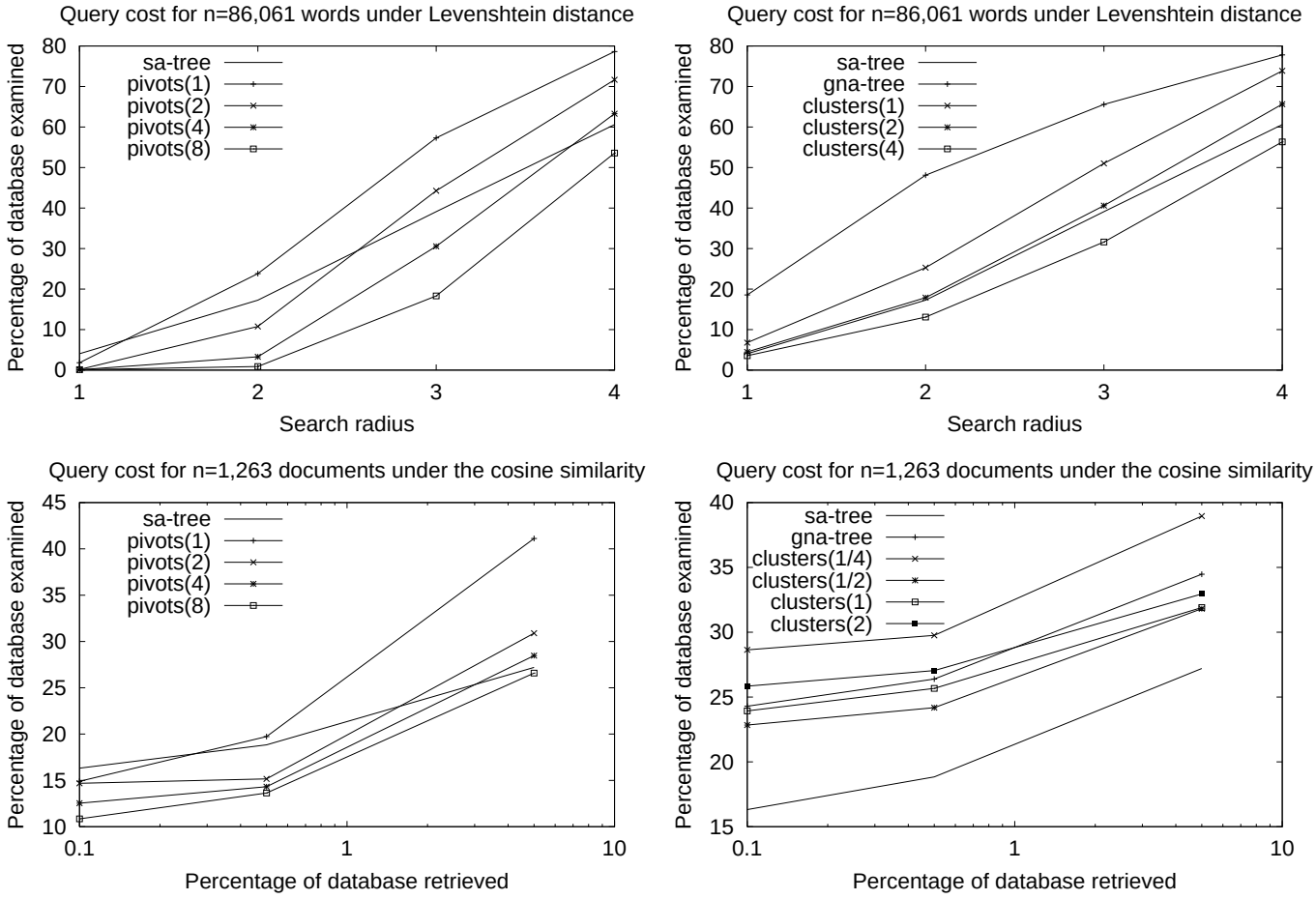


Figure 13: Comparison between the cost of range searching using the *sa-tree* and other algorithms. On top the space of words and on the bottom that of documents. On the left we compare against pivoting algorithms and on the right against clustering algorithms.

ongoing work [Rey01]. In this section we comment some alternatives we are working on.

7.1 Rebuilding the Subtree

The naive approach is that, once a new element x being inserted has to become a neighbor of a tree node a, we rebuild the whole subtree rooted at a. This may or may not be too costly depending on how close are we typically from the tree leaves at the moment of rebuilding the subtree. In general we expect to be quite close to the leaves, as the tree has sublogarithmic depth and the probability of becoming a neighbor is quite low (it grows exponentially with the depth).

It is possible to analyze the expected cost of this choice. Let us assume that we enter at the root of a tree of n elements with a new element. There are $\Theta(\log n)$ neighbors. Hence the probability of becoming a new neighbor is at most $1/2^{\Theta(\log n)} = n^{-\alpha}$ for some $\alpha > 0$ (recall Section 5). With this probability our new element becomes a neighbor and we have to rebuild the whole tree of size n. Otherwise we go to the closest neighbor and work on a tree of size $\Theta(n/\log n)$. The average size of the tree to rebuild is then

$$R(n) \;=\; n^{-\alpha}\, n \;+\; (1-n^{-\alpha})\, R\left(\frac{n}{\log n}\right) \;\leq\; n^{1-\alpha} + R\left(\frac{n}{\log n}\right) \;=\; O\left(n^{1-\alpha}\right)$$

which, given the cost $B(n) = O(n \log^2 n/\log\log n)$ of rebuilding a subtree of size n, gives an update cost of $O(n^{1-\alpha}\log^2 n/\log\log n)$. This shows that, on average, we only rebuild a small part of the tree, but the update cost is far away from the ideal logarithmic time.

7.2 Overflow Bags

We can have an overflow bag per node with "extra" neighbors against which the query must be directly compared but have no subtree to follow. When the new element x must become a neighbor of a, we put it in the overflow bag of a. Each time we reach a at query time, we also compare q against its overflow bag and report any element near enough.

At periodic intervals, the structure must be rebuilt from scratch in order to maintain a reasonable search efficiency. For example we may rebuild a subtree when its overflow bag exceeds a given size. The main question is which is the tradeoff in practice between reconstruction cost and query cost. The more often we rebuild the tree, the smaller are the overflow bags and the query time improves. Note that the naive strategy of Section 7.1 is a particular case of overflow bags with zero tolerance.

7.3 A First-Fit Strategy

Yet another solution is to change our *best-fit* strategy to put elements inside the bags of the neighbors of a at construction time. An alternative strategy, *first-fit*, is to put each node in the bag of the first neighbor closer than a. The determination of $N(a)$ and the assignment of the other elements to their bag in $N(a)$ can now be done in one pass.

With the first-fit strategy, however, we can easily add more elements by pretending that the new incoming element x was the last one in the bag, which means that when it becomes a neighbor of a it can be simply added as the last neighbor of a, and there were no later elements that had the chance of getting into x. This allows building the structure by successive insertions.

The range search under the first-fit construction strategy is a little different. We consider the neighbors $\{v_1, \ldots, v_k\}$ of a in order. We perform the minimization while we traverse the neighbors. That is, we enter into the subtree of v_1 if $d(q, v_1) \leq d(q, a) + 2r$; we enter into the subtree of v_2 if $d(q, v_2) \leq \min(d(q, a), d(q, v_1)) + 2r$; and in general we enter into the subtree of v_i if $d(q, v_i) \leq \min(d(q, a), d(q, v_1), \ldots, d(q, v_{i-1})) + 2r$. This is because a neighbor v_{i+j} can never take out an element from v_i.

Our preliminary experiments, however, show that the *first-fit* strategy works much worse in practice because of its asymmetry (the first subtrees are much larger than the last ones, and they are searched more often). Therefore, although this solution is elegant, it is not really promising.

7.4 Keeping Track of History

An alternative that has resemblances with the two previous, but seems more promising, consists in keeping a timestamp of the insertion time of each element. When inserting a new element, we add it as a neighbor at the appropriate point using *best-fit* and do not rebuild the tree. Let us consider that neighbors are added at the end, so by reading them left to right we have increasing insertion times. It also holds that the parent is always older than its children.

At search time, we consider the neighbors $\{v_1, \ldots, v_k\}$ of a from oldest to newest. We perform the minimization while we traverse the neighbors, exactly as in Section 7.3. This is because between the insertion of v_i and v_{i+j} there may have appeared new elements that preferred v_i just because v_{i+j} was not yet a neighbor, so we may miss an element if we do not enter into v_i because of the existence of v_{i+j}.

Note that, although the search process is the same as under *first-fit*, the insertion puts the elements into their closest neighbor, so the structure is balanced.

Up to now we do not really need timestamps but just keeping the neighbors sorted. Yet a more sophisticated scheme is to use the timestamps to reduce the work done inside older neighbors. Say that $d(q, v_i) > d(q, v_{i+j}) + 2r$. We have to enter into v_i because it is older. However, only the elements with timestamp smaller than that of v_{i+j} should be considered when searching inside v_i; younger elements have seen v_{i+j} and they cannot be interesting for the search if they are inside v_i. As parent nodes are older than their descendants, as soon as we find a node inside the subtree of v_i with timestamp larger than that of v_{i+j} we can stop the search in that branch, because its subtree is even younger.

7.5 Inserting at the Leaves

Another promising alternative we are considering is as follows. We can relax Condition 2 (Section 4.1), whose main goal is to guarantee that if q is closer to a than to any neighbor in $N(a)$ then we can stop the search at that point. The idea is that, at search time, instead of finding the closest c among $\{a\} \cup N(a)$ and entering into any $b \in N(a)$ such that $d(q, b) \leq d(q, c) + 2r$, we exclude the subtree root $\{a\}$ from the minimization. Hence, we *always* continue to the leaves by the closest neighbor and others close enough.

This makes the search time slightly worse, but our preliminary experiments show that in practice the difference is negligible. The benefit is that we are not forced anymore to put a new inserted element x as a neighbor of a, even when Condition 2 would require it. That is, at insertion time,

even if x is closer to a than to any element in $N(a)$, we have the choice of not putting it as a neighbor of a but inserting it into its closest neighbor of $N(a)$. At search time we will reach x because the search and insertion processes are similar.

This freedom opens a number of new possibilities that deserve a much deeper study, but an immediate consequence is that we can insert always at the leaves of the tree. Hence, the tree is read-only in its top part and it changes only in the fringe. Our insertion cost becomes sublogarithmic (proportional to the average leaf depth). However, it is not clear how good is to do this all the time, since we are not optimizing the neighbors. It is possible that if we build all the tree in this way the search quality gets affected. An intermediate alternative is to permit inserting x as a neighbor when the size of the subtree to rebuild is small enough, which leads to a tradeoff between insertion cost and quality of the tree at search time. For example we can permit rebuilding subtrees of size $O(\log n/(\log\log n)^2)$ only, and hence the insertion cost becomes logarithmic.

An ideal scenario would be to estimate how much we improve the search time by inserting x as a neighbor of the current node a versus how costly is it to rebuild the tree.

8 Conclusions

We have presented a new data structure, the *sa-tree*, to search in metric spaces. Our idea is to approach the query spatially rather than by dividing the set of candidates as in other approaches. We first show that the ideal structure for spatial approximation cannot be built and then propose a structure which provides a reasonable trade-off by combining spatial approximation with backtracking. We show analytically that the number of distance evaluations at search time is $o(n)$, and present experimental evidence showing that our structure outperforms all the others on high-dimensional spaces or queries with low selectivity. These are the harder cases in proximity searching.

Some issues for future work which we are already pursuing follow.

- We have made some heuristic decisions in order to find a data structure that can be built in reasonable time, e.g. selecting the root at random or using a simple heuristic to select a set of neighbors $N(a)$. It may be possible to find better solutions that improve the search time.

- The *sa-tree* outperforms the other structures on high dimensions (where the problem is more difficult) but is inferior to others when the problem is easier (low dimensions). Moreover, it cannot trade space for query time as pivoting schemes do. 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. A simple twist is to store the distance of each node to its k ancestors in the tree, so as to use them as pivots to prune the search space. This does not require more distance evaluations at construction or at query time, but it increases the index space by kn distances.

- It is interesting to try to reduce the backtracking, although our attempts up to now have failed. A first choice is to refine a tolerance radius R and insert each element into its closest neighbor and any other that differs from it in at most R. The backtracking can now be done with tolerance $2(r-R)$. The problem is that the structure is now a DAG (directed

acyclic graph), not a tree, and that they may appear loops in the construction, a problem we have solved by restarting a new DAG with the elements involved in the loop. This has lead to a large number of small DAGs, each of which has to be traversed almost completely (the sublinearity plays against us in this case). Another choice is to put as neighbor any element whose difference of distances between the root and the closest neighbor is at most R, and reduce the backtracking to $2(r - R)$ again. The problem is that the balance between number of neighbors and backtracking is delicate: too many neighbors are generated and the efficiency is reduced.

- A problem still open is how to allow dynamic insertions and deletions of elements without degrading the performance. This is our main focus at the moment [Rey01]. We have presented a number of possible solutions for dynamic insertions, some of which are promising. Deletions seem more difficult but always can be treated by marking the nodes as deleted and making periodic rebuilds. The average cost of a deletion is anyway sublinear, $O(n^{1-\alpha})$ for $0 < \alpha < 1$, just like the insertion.

- Secondary memory issues have not been considered. A simple solution is to try to store whole subtrees in disk pages so as to minimize the number of pages read at search time. This has an interesting relationship with the last proposal for dynamic insertions (Section 7.5), not only because we can insert always at the leaves but also because we can control the maximum arity of the tree so as to make the neighbors fit in a disk page.

- It would be interesting to build approximate or probabilistic algorithms based on this structure, as they have proved to be of great interest in high dimensional metric spaces using other data structures that typically work well only on low dimensional spaces [CN01].

- Our data structure was born in the quest for a more powerful structure, which we could call a *spatial approximation graph*. Such a directed graph should permit us to reach any element from each other by always reducing the distance to it. Fast algorithms to build and exploit this structure and a mathematical characterization of it is interesting by itself. Moreover, other simplified structures, different from the *sa-tree*, could be created based on the spatial approximation approach.

References

[Aur91] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Ben79] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, 5(4):333–340, 1979.

[BK73] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, 1973.

[BO97] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. ACM Conference on Management of Data (SIGMOD'97)*, pages 357–368, 1997. Sigmod Record 26(2).

[Bri95] S. Brin. Near neighbor search in large metric spaces. In *Proc. of the 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.

[BYCMW94] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Conference on Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[CM97] E. Chávez and J. Marroquín. Proximity queries in metric spaces. In *Proc. 4th South American Workshop on String Processing (WSP'97)*, pages 21–36. Carleton University Press, 1997.

[CMBY99] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. 6th South American Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 38–46. IEEE CS Press, 1999.

[CMN01] E. Chávez, J. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001. Kluwer. To appear.

[CN00] E. Chávez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proc. 7th South American Symposium on String Processing and Information Retrieval (SPIRE'00)*, pages 75–86. IEEE CS Press, 2000.

[CN01] E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *Proc. 3rd Workshop on Algorithm Engineering and Experiments (ALENEX'01)*, LNCS, 2001. To appear.

[CNBYM01] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 2001. To appear.

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Conference on Very Large Databases (VLDB'97)*, pages 426–435, 1997.

[DN87] F. Dehne and H. Nolteimer. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987. Pergamon Journals.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM Conference on Management of Data (SIGMOD'84)*, pages 47–57, 1984.

[Har95] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[MOC96] L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbor classifier in metric spaces. *Pattern Recognition Letters*, 17:731–739, 1996. Elsevier.

[MOV94] 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. *Pattern Recognition Letters*, 15:9–17, 1994. Elsevier.

[Nav99] G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. 6th South American Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 2001. To appear.

[NN97] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

[Nol89] H. Nolteimer. Voronoi trees and applications. In *Proc. International Workshop on Discrete Algorithms and Complexity*, pages 69–74, 1989.

[NVZ92] H. Nolteimer, K. Verbarg, and C. Zirkelbach. Monotonous Bisector* Trees – a tool for efficient partitioning of complex schenes of geometric objects. In *Data Structures and Efficient Algorithms*, LNCS 594, pages 186–203, 1992.

[Rey01] N. Reyes. *Dynamic data structures for searching metric spaces.* MSc. Thesis, Univ. Nac. de San Luis, Argentina, 2001. In progress. G. Navarro, advisor.

[Sha77] M. Shapiro. The choice of reference points in best-match file searching. *Communications of the ACM*, 20(5):339–343, 1977.

[Uhl91a] J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. Manuscript, 1991.

[Uhl91b] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991. Elsevier.

[Vid86] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[Yia93] 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.

[Yia00] P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000.