The VLDB Journal manuscript No.
(will be inserted by the editor)

Searching in Metric Spaces by Spatial Approximation *

Gonzalo Navarro[1]

Dept. of Computer Science, University of Chile. Blanco Encalada 2120 - Santiago - Chile. e-mail: gnavarro@dcc.uchile.cl

The date of receipt and acceptance will be inserted by the editor

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 triangle inequality. The goal is, given a set of objects and a query, retrieve those objects close enough to the query. The complexity measure is the number of distances computed to achieve this goal. Our data structure, called *sa-tree* ("spatial approximation tree"), is based on approaching spatially the searched objects, that is, getting closer and closer to them, 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 hard to search or the query has low selectivity. These are the most important unsolved 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 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 "triangle 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. 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.

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

Note that in large databases the I/O cost is assumed to be the most important complexity measure, as CPU costs tend to be negligible compared to disk access costs. This may or may not be the case in metric spaces. Some distance functions are so expensive to compute in terms of CPU time (think, for example, of comparing two fingerprints or two documents) that the overall search time, even for a large database that does not fit in main memory, is dominated by the number of distance evaluations performed rather than by the total number of disk pages read. So the axiom of considering only I/O costs may fail in this type of databases, depending on the relationship between the cost to compute a distance and the cost to read an object from disk.

A particular case of metric space searching is that of vector spaces, where the elements are D-dimensional points and their distance belongs to the Minkowski L_r family: $L_r = (\sum_{1 \le i \le D} |x_i - y_i|^r)^{1/r}$. 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: $L_\infty = \max_{1 \le i \le D} |x_i - y_i|$.

There are effective methods to search on D-dimensional spaces, such as kd-trees [3,2] or R-trees [17]. However, for roughly 20 dimensions or more those structures cease to work well. We focus in this paper on general metric spaces, although the solutions are well suited also for D-dimensional spaces. It is interesting to notice that the concept of "dimensionality" is related to "easiness" or "hardness" for searching a D-dimensional space: higher dimensional spaces have a probability distribution of distances among elements whose histogram is more concentrated and with larger mean. This makes the work of any similarity search algorithm more difficult (this is discussed for example in [33,6,9,14]). In the extreme case we have a space where $d(x, x) = 0$ and $\forall y \neq x,\ d(x, y) = 1$, where the query has to be exhaustively compared against every element in the set. We will extend this idea by saying that a general metric space is "harder" than other when its histogram of distances is more concentrated than the other.

Figure 1 gives some intuition on why more concentrated histograms yield harder metric spaces. Let p be a database element and q a query. The triangle inequality implies that every element x such that $|d(q, p) - d(p, x)| > r$ cannot be at distance r or less from q, so we could discard x. However, in a concentrated histogram the distances between two random distances are closer to zero and hence the probability of discarding an element x is lower.

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 triangle 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 concept completely different from existing methods, 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 hard metric spaces (concentrated histograms) or hard queries (large radii, i.e., low selectivity).

There are many interesting applications whose space is hard. Some examples are ranking documents for information retrieval or finding similar words for spelling purposes. 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 numerous applications that resort to metric space searching in their back end, 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 large 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 [8].

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 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. Incremental construction is discussed in Section 7. We draw our conclusions in Section 8. A partial and less mature earlier version of this work appeared in [22].

2 Previous Work

Algorithms to search in general metric spaces can be divided in two large areas: pivot-based and clustering algorithms. (See [14] for a more complete review.)

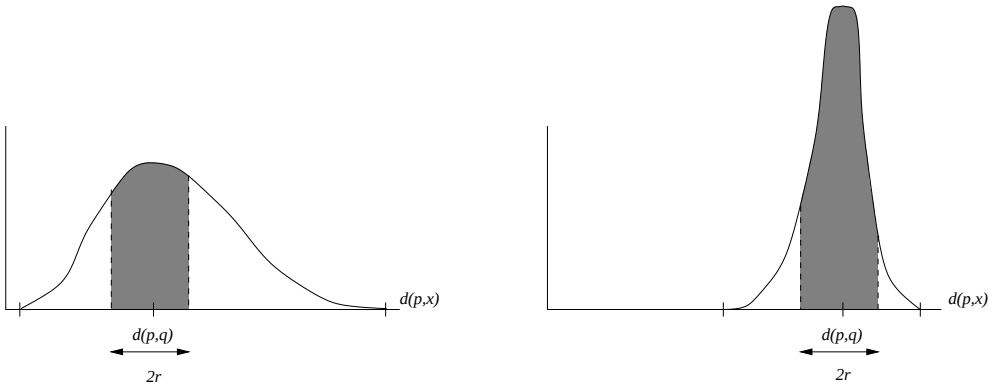


Fig. 1 A flatter (left) versus a more concentrated (right) histogram. The latter implies harder to search metric spaces because the triangle inequality permits discarding less elements (the non grayed area).

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 triangle inequality that $d(q,x) > r$ and therefore do not need to explicitly evaluate $d(q,x)$. All the other elements that cannot be eliminated using this rule are directly compared against the query.

Algorithms such as *aesa* [32], *laesa* [21], *spaghettis* and variants [10,24], *fq-trees* and variants [7], and *fq-arrays* [11], 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 triangle 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 [4,29], *metric trees* [31], *tlaesa* [20], and *vp-trees* and variants [33,5,34].

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 $\{c_1 \dots c_m\}$ be the set of centers. At query time we evaluate $(d(q,c_1), \dots, d(q,c_m))$, choose the closest center c and discard every zone whose center c_i satisfies $d(q,c_i) > d(q,c) + 2r$, as its 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 [31,27], and *Voronoi trees* [16,25]. Some using only covering radii are the *M-trees* [15] and *lists of clusters* [12]. One using both criteria is the *gna-tree* [6].

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 [30] 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 triangle 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). See also [19] for alternative ideas (albeit for vector spaces).

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 the sense of finding closer and closer elements to it.

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. From now on we will speak of graph (or tree) nodes and database elements (or objects) indistinctly.

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:

Property 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 Property 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][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 Property 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 element b we add as a neighbor of a will allow the search to move from a to b provided q is in b's side 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 element.

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 2 shows an example.

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, given only the distances. 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 1 *Given the distances between pairs of elements in a finite subset S of an unknown metric space U, then for each $a, b \in S$ there exists a choice for 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 the 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

[1] The proper name in a general metric space is "Dirichlet domain" [6].

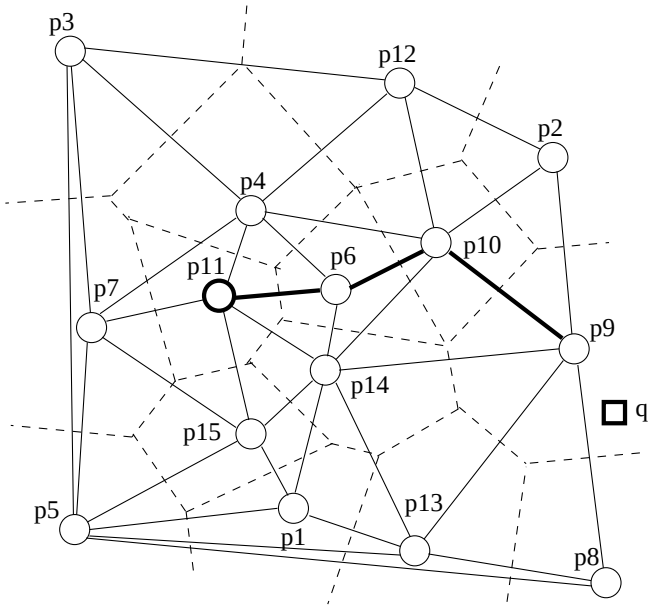


Fig. 2 An example of the search process with a Delaunay graph (solid edges) corresponding to a Voronoi partition (areas delimited by dashed lines). We start from p_{11} and reach p_9, the node closest to q, moving always to neighbors closer and closer to q.

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 3). This argument can be repeated for any pair $a, b \in S$. □

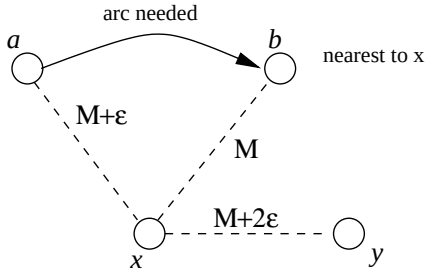


Fig. 3 Illustration of the theorem.

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:

Property 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 "$\Leftarrow$" 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 "$\Rightarrow$" part aims at obtaining as adding only the necessary neighbors.

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 that add more than the minimum possible neighbors 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 Property 2 is satisfied thanks to the fact that we have considered the elements in order of increasing distance to a. The "$\Leftarrow$" part of the Property is clearly satisfied because any element satisfying the clause on the right is inserted in $N(a)$. The "$\Rightarrow$" 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 mechanism consists in comparing q against $\{a\} \cup N(a)$. If a is closest to q, then a is the answer, otherwise we continue the search by the subtree of the closest element to q in $N(a)$.

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 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 at a. The way to use this information is made clear in Section 4.2.

Figure 4 depicts the construction process.

BuildTree(Node a, Set of nodes S)

```
1.   N(a) ← ∅      /* neighbors of a */
2.   R(a) ← 0      /* covering radius */
3.   Sort S by distance to a (closer first)
4.   For v ∈ S Do
5.        R(a) ← max(R(a), d(v,a))
6.        If ∀b ∈ N(a), d(v,a) < d(v,b) Then
7.             N(a) ← N(a) ∪ {v}
8.   For b ∈ N(a) Do S(b) ← ∅     /* subtrees */
9.   For v ∈ S − N(a) Do
10.       Let c ∈ N(a) be the one minimizing d(v,c)
11.       S(c) ← S(c) ∪ {v}
12.  For b ∈ N(a) Do BuildTree(b, S(b))
```

Fig. 4 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, some of the $d(v,c)$ at line 10 are 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.

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 triangle inequality it holds that for any $x \in U$, $d(x,q) - r \leq d(x,q') \leq d(x,q) + r$.

When we knew the q we were searching for, we went directly to the neighbor of a closest to q. Now, we are searching for the unknown q' and are not certain of which is the neighbor of a closest to q'. Hence, we have to explore several possible neighbors. Some neighbors, fortunately, can be deduced to be irrelevant, as q' cannot have chosen them at construction time if it holds $d(q,q') \leq r$.

Instead of just going to the closest neighbor, we first determine the closest neighbor c of q among $\{a\} \cup N(a)$. So, for any b in $\{a\} \cup N(a)$, we know that $d(c,q) \leq d(b,q)$. However, as explained, it is possible that $d(c,q') \geq d(b,q')$ and therefore we will not find q' by entering only the tree of c. Instead, we must enter into *all* the 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 other words, a neighbor b such that $d(q,b) > d(q,c) + 2r$ satisfies $d(q',b) \geq d(q,b) - r > d(q,c) + r \geq d(q',c)$, so q' could not have been inserted in the subtree of b. In any other case, we are not sure and must enter the subtree of b.

A different way to regard this process is to lower bound the distance between q and any node x in the subtree of b. By the triangle inequality we have $d(x,q) \geq d(x,c) - d(q,c)$ and $d(x,q) \geq d(q,b) - d(x,b)$. Summing up both inequalities and keeping in mind that $d(x,b) \leq d(x,c)$ and $d(q,c) \leq d(q,b)$, we obtain $2d(x,q) \geq (d(q,b) - d(q,c)) + (d(x,c) - d(x,b)) \geq d(q,b) - d(q,c)$. Therefore $d(x,q) \geq (d(q,b) - d(q,c))/2$. If the latter term is larger than r, we can safely discard every x in the subtree of b. The condition is therefore $(d(q,b) - d(q,c))/2 > r$, or $d(q,b) > d(q,c) + 2r$.

The process guarantees that we compare q against every node that cannot be proved to be far away enough from q. Hence, by reporting every node q' that was compared against q and for which $d(q,q') \leq r$ holds, we are sure to report every relevant element.

As can be seen, 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 5 illustrates the search process.

The search algorithm can be improved a bit further. When we search for an element $q \in S$ (that is, an exact search for a tree node), we follow a single path from the root to q. At any node a' in this path, we choose the closest to q among $\{a'\} \cup N(a')$. Therefore, if the search is currently at tree node a, we have that q is closer to a than to any ancestor a' of a and also any neighbor of a'. Hence, if we call $A(a)$ the set of ancestors of a (including a), we have that, at search time, we can avoid entering

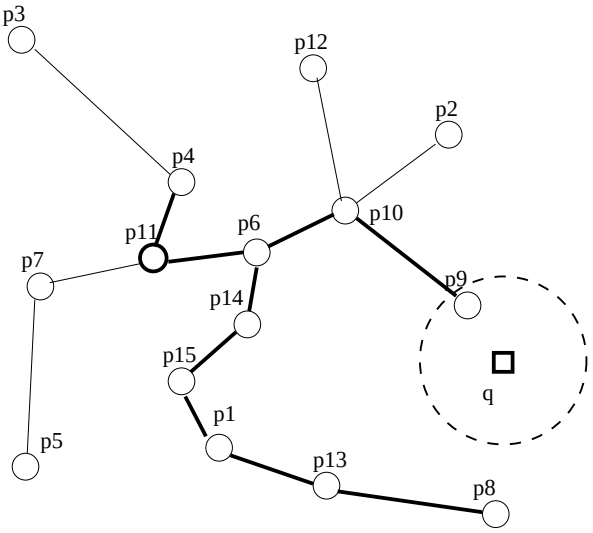


Fig. 5 An example of the search process, starting from p_{11} (tree root). Only p_9 is in the result, but all the bold edges are traversed.

any element $x \in N(a)$ such that

$$d(q,x) > 2r + \min\{d(q,c),\ c \in \{a'\} \cup N(a'),\ a' \in A(a)\}$$

because we can show using the triangle inequality that no q' with $d(q,q') \leq r$ can be stored inside x. This condition is a stricter version of the original condition $d(q,x) > 2r + \min\{d(q,c),\ c \in \{a\} \cup N(a)\}$.

We use this observation as follows. At any node b of the search we keep track of the minimum distance $mind$ to q seen up to now across this path, including neighbors. We enter only neighbors that are not farther than $mind + 2r$ from q.

Finally, the covering radius $R(a)$ is used to further reduce the search cost. We never enter into a subtree rooted at a where $d(q,a) > R(a) + r$, since this implies $d(q',a) > R(a)$ for any q' such that $d(q,q') \leq r$. The definition of $R(a)$ implies that q' cannot belong to the subtree of a. Figure 6 depicts the algorithm.

RangeSearch(Node a, Query q, Radius r, Dist. $mind$)

```
1.  If d(a,q) ≤ R(a) + r Then
2.      If d(a,q) ≤ r Then Report a
3.      mind ← min {mind} ∪ {d(q,c), c ∈ N(a)}
4.      For b ∈ N(a) Do
5.          If d(b,q) ≤ mind + 2r Then
6.              RangeSearch(b,q,r,mind)
```

Fig. 6 Algorithm to search q with radius r in a *sa-tree*. It is firstly invoked as RangeSearch(a,q,r,$d(a,q)$), where a is the root of the tree. Notice that in the recursive invocations $d(a,q)$ is already computed.

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 that 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 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 [30] can be adapted to our data structure. We have a priority queue of subtrees, most promising first. Initially, we insert the *sa-tree* root in the data structure. Iteratively, we extract the most promising subtree root, process it, and insert all the roots of its subtrees in the queue. This is repeated until the queue becomes empty or its most promising subtree root can be discarded (i.e., its "promise value" is bad enough).

The most elegant measure of how promising is a subtree is a lower bound to the distance between q and any element in the subtree. Once this lower bound exceeds r we can stop the whole process. We have indeed two possible lower bounds:

1. Since we find the closest neighbor c and then enter into any other neighbor b such that $d(q,b) - d(q,c) \leq 2r$, we have that we would not have entered the subtree rooted at b if $(d(q,b) - d(q,c))/2 \leq r$ did not hold. In fact, this c is taken over the neighbors of any ancestor.
2. By the lower bound to the distance between q and an element in the subtree we have $d(q,b) - R(b) \leq r$.

Since r is reduced along the search, a node b may seem useful at the moment it is inserted in the priority queue and useless later, when it is extracted from the queue to be processed. So we store together with b the maximum of the three lower bounds, and use this maximum to sort the subtrees in the priority queue, smaller first. As we extract subtrees from the queue, we check whether their value exceeds r, in which case we stop the whole process as all the remaining subtrees are known to contain irrelevant elements. Note that we need to keep track of $mind = m$ separately. Finally, note that children nodes inherit the lower bound of their parents.

Figure 7 depicts the algorithm.

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

1.   Q ← {(a, max(0, d(q,a) − R(a)), d(q,a))}    /* promising subtrees */
2.   A ← ∅    /* best answer so far */
3.   r ← ∞
4.   While Q is not empty Do
5.       (b,t,m) ← element in Q with smallest t , Q ← Q − {(b,t,m)}
6.       If t > r Then Return the answer A    /* global stopping criterion */
7.       A ← A ∪ {(b, d(q,b))}
8.       If |A| = k + 1 Then
9.           (c, maxd) ← element in A with largest maxd , A ← A − {(c, maxd)}
10.      If |A| = k Then
11.          (c, maxd) ← element in A with largest maxd , r ← maxd
12.      m ← min {m} ∪ {d(c,q), c ∈ N(b)}
13.      For v ∈ N(b) Do
14.          Q ← Q ∪ (v, max(t, (d(q,v) − m)/2, d(q,v) − R(v)), m)
15.  Return the answer A
```

Fig. 7 Algorithm to search the k nearest neighbors of q in a *sa-tree*. A is a priority queue of pairs *(node,distance)* sorted by increasing *distance*. Q is a priority queue of triples *(node,lbound,mind)* sorted by increasing *lbound*.

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 reality is very good. 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 hard spaces and $O(n^\alpha)$ $(0 < \alpha < 1)$ in easy spaces.

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 side of the hyperplane, never from the side of the hyperplane of another neighbor (which is the same to say that neighbors are closer to a than to each other). Second, in easy spaces (e.g. low dimensional vector spaces) 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 (i.e. attempts) 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 (Appendix A), 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 1.35. Recall also that there is a constant part that should be especially relevant in easy spaces. 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

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

which is obtained by unrolling (see Appendix B).

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.

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

which is solved in detail in Appendix B.

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 $X = D_i - \min(D_0, \ldots, D_j)$, 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 $X = D_i - \min(D_0, \ldots, D_j) \leq 2r$. The probability of such a fact is $F(2r)$.

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 $\Theta(n/\log n)$. Hence if we call $T(n)$ the search cost with n elements, then the following recurrence holds

$$T(n) = \log n + \log n\, F(2r) T\left(\frac{n}{\log n}\right)$$

which can be solved by unrolling (see details in Appendix B) to get

$$T(n) = \Theta\left(nF(2r)^{\log_{\log n} n}\right) = \Theta\left(n^{1-\frac{\log(1/F(2r))}{\log\log n}}\right) = \Theta\left(n^{1-\Theta(1/\log\log n)}\right)$$

This shows the sublinearity with respect to n. On the other hand, as the search radius increases or the hardness increases, $F(2r)$ becomes closer to 1 and the cost becomes closer to linear.

On the other hand, note that when the space is easy (e.g. vector spaces with dimension smaller than $O(\log n)$), $N(n)$ is closer to a constant because there cannot be too many neighbors. In this case the analysis yields $T(n) = O(n^\alpha)$ 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: every coordinate was chosen uniformly and independently in $[0,1)$. 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 (difficulty) 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.

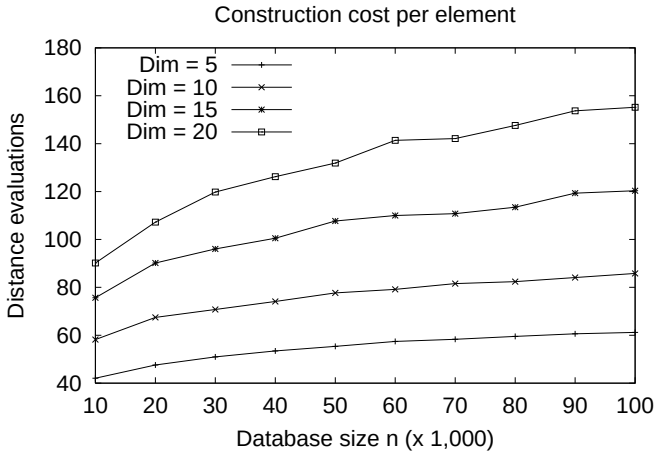
6.1 Construction Cost and Tree Shape

Our first experiment aims at measuring the construction cost of the *sa-tree*, as well as the shape of the resulting tree. Figure 8 shows how the cost grows as n increases. We show the number of evaluations per element, which according to the analysis is $O(\log^2 n/\log\log n)$. 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.

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 9.

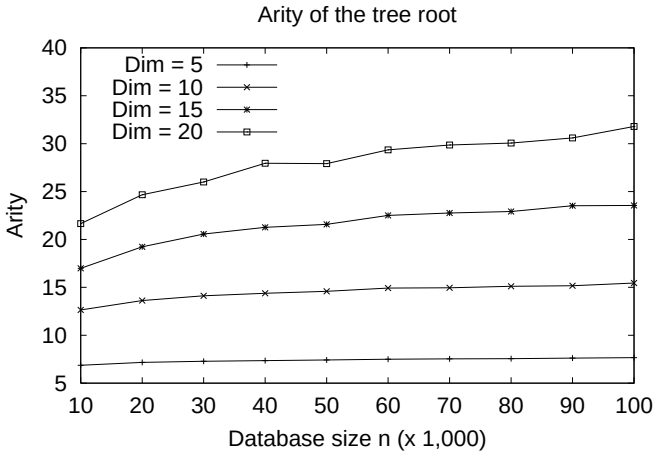
Let us now focus on the average leaf depth of the trees. The analysis predicts $O(\log n/\log\log n)$. Again, we have obtained a very good approximation, with relative error well below 1%, with the model $a + b\ln n/\ln\ln n$. This time the constant b decreases with the dimension. Figure 10 shows the results.

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



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

Fig. 8 Construction cost, measured in number of distance evaluations per element. The cost grows with n and with the dimension of the database. On the bottom, the formula obtained by least squares and the relative error.



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

Fig. 9 Arity of the tree root. It grows with n and with the dimension of the database. On the bottom, the formula obtained by least squares and the relative error.

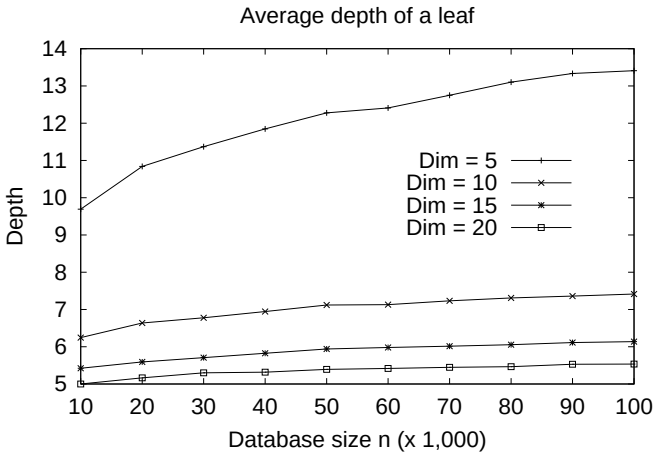
dimension of the set grows. As the experiments show, the trees get fatter and shorter for higher dimensions, and consequently they are harder to build.

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* [6], 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

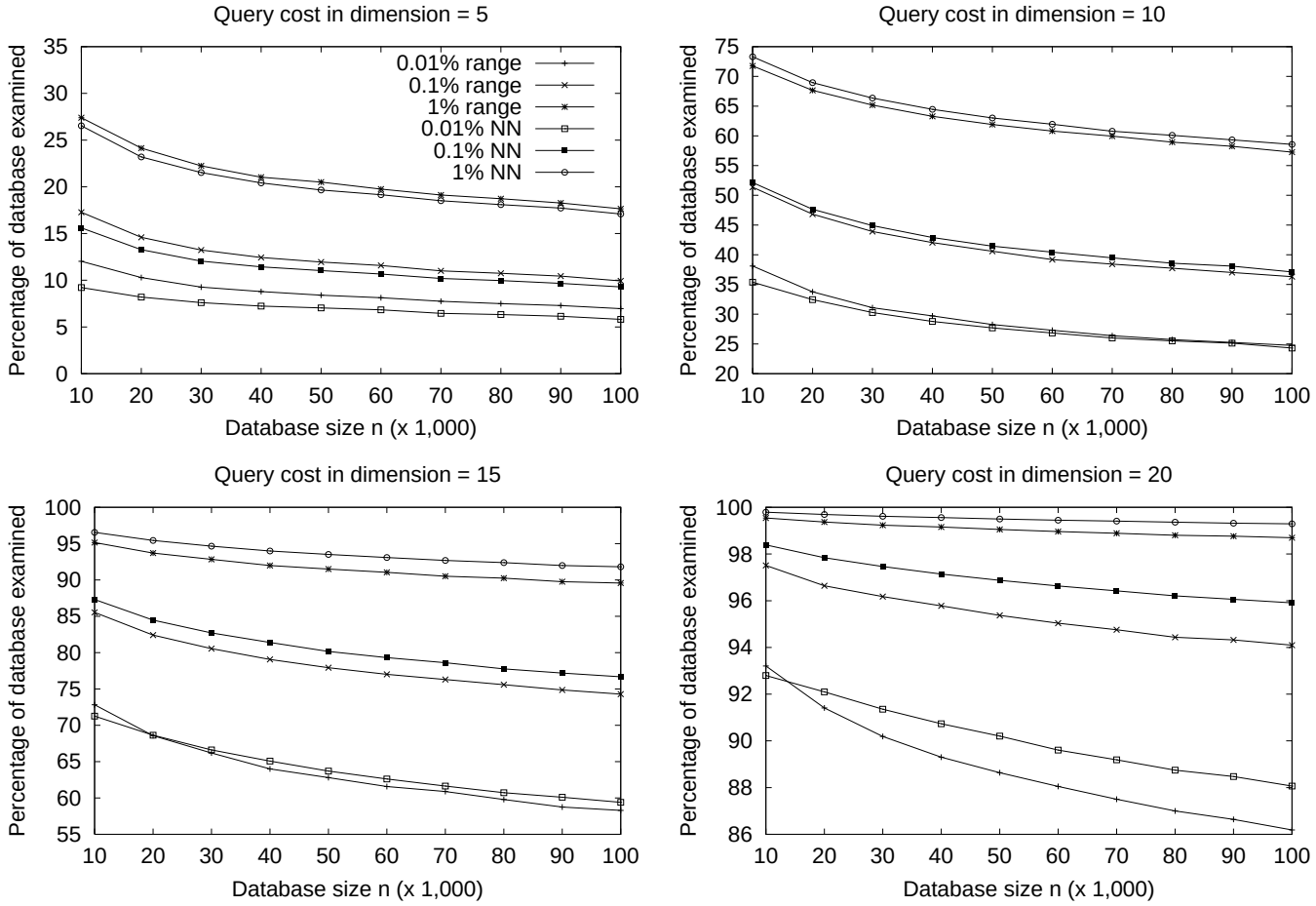
We consider now the cost of searching the index. We have tried both range and nearest neighbor searching. For range searching, we have selected manually the radii that recover 0.01%, 0.1% and 1% of the set. For nearest neighbor searching, we have directly requested to retrieve that number of elements. 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 11 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 always below 1%). 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 is quite close to the corresponding range search.



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

Fig. 10 Average leaf depth in the tree. It grows with n and decreases with the dimension of the database. On the bottom, the formula obtained by least squares and the relative error.



Dimension	range 0.01%	range 0.1%	range 1%
5	$5.911n^{1-0.938/\ln\ln n}$ (.007)	$8.479n^{1-0.939/\ln\ln n}$ (.008)	$6.413n^{1-0.760/\ln\ln n}$ (.005)
10	$9.582n^{1-0.776/\ln\ln n}$ (.003)	$6.831n^{1-0.622/\ln\ln n}$ (.004)	$3.759n^{1-0.398/\ln\ln n}$ (.004)
15	$3.828n^{1-0.399/\ln\ln n}$ (.003)	$2.437n^{1-0.251/\ln\ln n}$ (.002)	$1.501n^{1-0.109/\ln\ln n}$ (.001)
20	$1.668n^{1-0.140/\ln\ln n}$ (.002)	$1.275n^{1-0.064/\ln\ln n}$ (.001)	$1.062n^{1-0.015/\ln\ln n}$ (.000)

Fig. 11 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.

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* [32] needs $O(n^2)$ space) or construction time (e.g. *aesa* [32] and the *list of clusters* [12] need $O(n^2)$ construction time). To make a fair comparison we consider the amount of memory or construction time required. 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*.
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 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.

A compact implementation of our data structure needs: for every object, a leaf/nonleaf bit plus an object identifier (17 bits are enough for 100,000 elements); for every internal node, a covering radius (32 bits, both considering a floating point number or the number of bits to distinguish among $n^2/2$ distances when $n = 100,00$), a pointer to the first child (17 bits) and the number of children (5 bits is more than enough, both analytically and in our experiments). This permits a breath-first representation of the *sa-tree* on an array. In our experiments, there are at most 6 leaves per internal node (this also matches analytical predictions), so in total we need about $27n$ bits.

On the other hand, we need 32 bits to represent a distance, so the minimal space for k pivots is $32k$ bits. Hence, Pivot(s) is equivalent to using $k = s$ pivots.

Clusters(t): is the scheme proposed in [12]. This structure takes linear space and it is shown to behave better than *sa-trees* in hard spaces. However, for this to happen it is necessary to pay a quadratic construction cost, which is unrealistic even compared to our (already expensive) construction cost.

The data structure consists of a list of balls of m elements. The first ball is formed by a center c_1 and the $m-1$ elements closest to c_1. Those m elements are not considered when building the rest of the list. For the second ball another center is chosen and the ball contains the $m-1$ elements closest it, and so on. At search time every center c_i is compared against q in sequence. 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 m of $817/t$, $582/t$, $415/t$ and $322/t$ for dimensions 5, 10, 15 and 20, respectively.

Gna-tree: is a simplification of the structure proposed in [6]. A set of m centers is selected at random and the rest are sent to the subtree of their closest center. The subtrees are built recursively. At search time the query is compared against the m centers and enters into the closest, c, and into those whose Voronoi region have intersection with the query ball (i.e. $d(q, c_i) \leq d(q, c) + 2r$). Covering radii are used as well to increase pruning. 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. Our experiments with the full-fledged structure proposed in [6] show that our simplification is indistinguishable in performance.

Figure 12 shows a comparison between *sa-trees* and the idealized pivoting algorithms. As it can be seen, the *sa-tree* tolerates better harder spaces or larger radii. A pivoting index using four times the 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 hardness or the search radius grow, pivoting algorithms need more and more memory in order to compete. In hard spaces or large search radii, pivoting algorithms cannot compete even when they take 64 times the amount of memory required by *sa-trees*.

Figure 13 shows a comparison between *sa-trees* and clustering algorithms. These algorithms tolerate better harder spaces 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 hardness or the search radii grow: 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 (or Levenshtein) distance, defined as the number of character insertions, deletions and substitutions needed to convert one string into the other. This distance is discrete and has many applications in text retrieval, signal processing and computational biology [23]. The particular case of a dictionary is of interest in spelling applications.

The second metric space is that of documents under the cosine similarity measure [8]. We took the 25,960 documents of the FR (Federal Register) collection of TREC-3 [18]. We took the vocabulary of each document (considering letters and digits and mapping them to lower case) and created for each document a vector where each vocabulary word is a coordinate. If the vocabulary word t_i appears f_{ij} times 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 largely used in Information Retrieval applications, and it is quite expensive to compute.

In the space of words under the edit distance, 5 bits suffice to store a distance, and hence the space taken by the *sa-tree* is equivalent to that of 5 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 radii 1 to 4, which

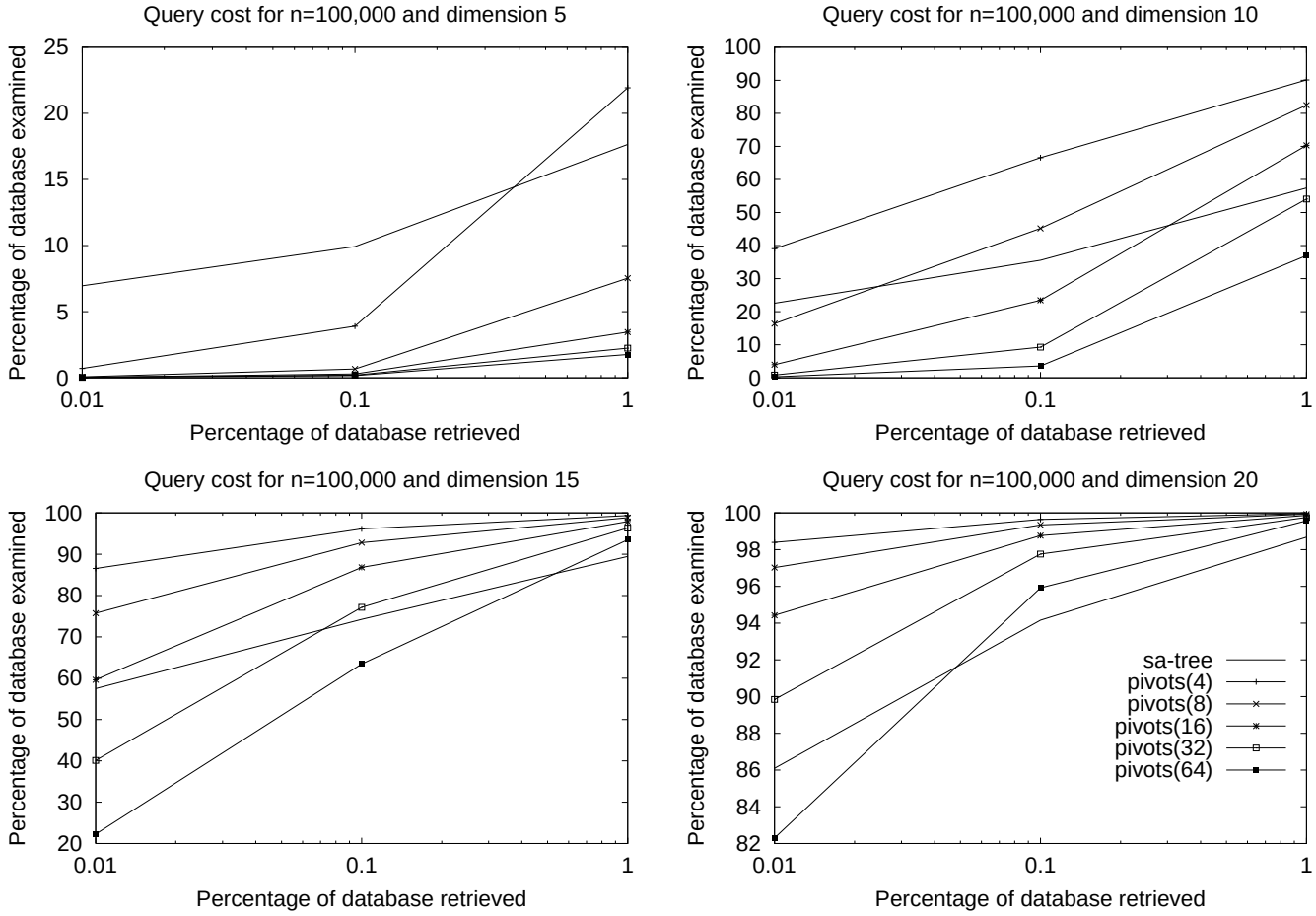


Fig. 12 Comparison between the cost of range searching using the *sa-tree* and an idealized pivoting algorithm. We show each dimension separately and the cost for growing radius (i.e. queries that retrieve 0.01%, 0.1% and 1% of the database).

retrieved 0.00354%, 0.0300%, 0.258% and 1.515% 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 one pivot. The *gna-tree* gives its best results with arity 4 (in fact, the arity makes little difference). A *list of clusters* of equivalent construction cost uses clusters of size 109, as the *sa-tree* needed 118.63 comparisons per element. We show the results of searching with radii retrieving 1 to 16 elements apart from the query itself. 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. Moreover, this space has very high dimension.

Figure 14 shows the results. In the space of words, the *sa-tree* outperforms the *gna-tree*. A pivoting algorithm needs 8 times more space to beat *sa-trees* when the search radius becomes large (3 or 4). Lists of clusters need to pay 4 times the construction cost of *sa-trees* in order to achieve better efficiency.

In the space of documents, pivots (even using 64 of them) and *gna-trees* perform poorly. The only competitor for the *sa-tree* is the *list of clusters*. When retrieving very few elements, they need 8 times more construction time to beat *sa-trees*.

As it can be seen, *sa-trees* provide a good tradeoff between efficiency and space/construction cost. It is necessary to pay much more space or construction time to beat them when the space is hard or the search radius is large. These are the most important unresolved 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

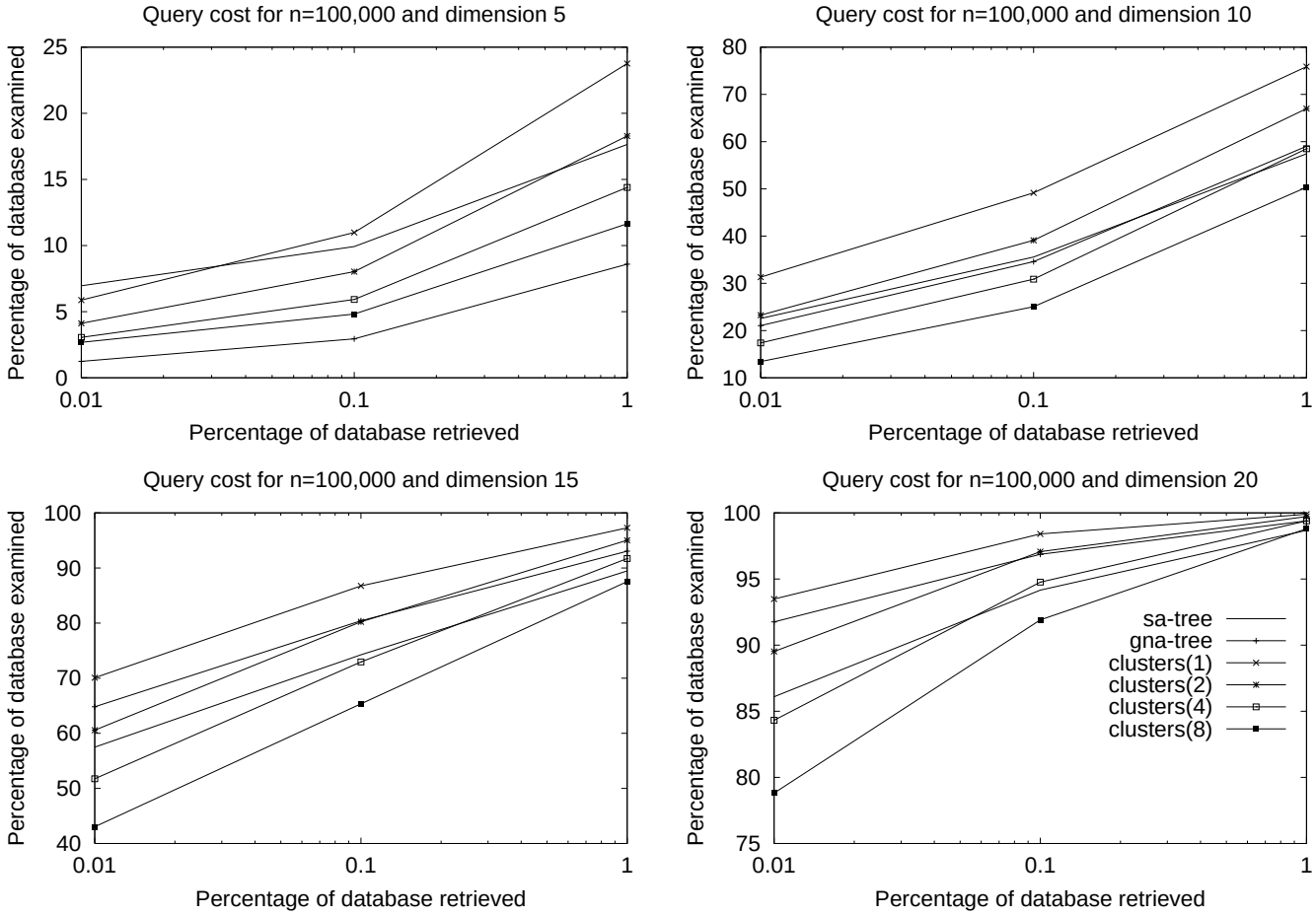


Fig. 13 Comparison between the cost of range searching using the *sa-tree* and other clustering algorithms. We show each dimension separately and the cost for growing radius (i.e. queries that retrieve 0.01%, 0.1% and 1% of the database).

went into another neighbor could prefer now to get into the new neighbor.

We have studied several alternatives that permit an efficient incremental construction, that is, by successive insertions [26]. We present below those that have worked better. We show some experimental results that make it clear that a dynamic *sa-tree* is feasible. Moreover, we have found that their performance may even be better than the standard structure in some cases. A deep analysis of these facts is our current focus [28].

7.1 Timestamping

We keep a timestamp of the insertion time of each element. When inserting a new element, we add it as a neighbor at the appropriate point but omit rebuilding the tree. This makes construction cost by successive insertions very close to that of a static construction.

Let us consider that neighbors are added at the end of the list, 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 in order. We perform the minimization (*mind* in Figure 6) as we traverse the neighbors. That is, we enter into the subtree of v_1 whenever $d(q, v_1) \leq d(q, a) + 2r$; into the subtree of v_2 whenever $d(q, v_2) \leq \min(d(q, a), d(q, v_1)) + 2r$; and in general into the subtree of v_i whenever $d(q, v_i) \leq \min(d(q, a), d(q, v_1), \ldots, d(q, v_{i-1})) + 2r$.

This works because between the insertion of v_i and v_{i+j} new elements may have appeared 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}.

Up to now we do not really need timestamps but just to keep the neighbors sorted by insertion time. Yet a more sophisticated scheme is to effectively use the timestamps to reduce the work done inside older neighbors. Say that v_i cannot be discarded by an older sibling or by the parent, that is $d(q, v_i) \leq \min(d(q, a),\ d(q, v_1), \ldots, d(q, v_{i-1})) + 2r$. So we have to enter into v_i even if $d(q, v_i) > d(q, v_{i+j}) + 2r$ for some younger sibling v_{i+j}. However, only the elements with timestamp older than that of v_{i+j} should be considered

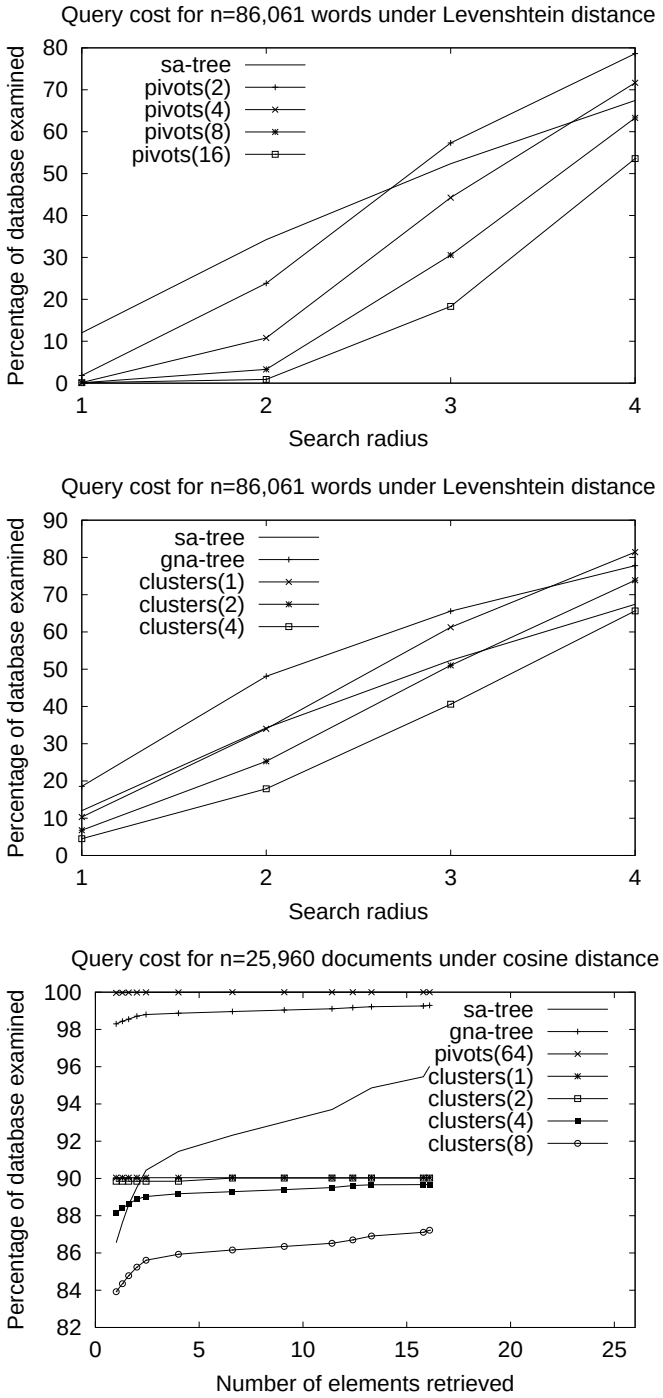


Fig. 14 Comparison between the cost of range searching using the *sa-tree* and other algorithms. On top the space of words for pivoting algorithms, in the middle the space of words for clustering algorithms, and on the bottom the space of documents.

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. So for every v_i such that $d(q,v_i) \leq \min(d(q,a),\ d(q,v_1),\ldots,d(q,v_{i-1})) + 2r$, we compute the oldest timestamp t among the set $\{v_{i+j},\ d(q,v_i) > d(q,v_{i+j}) + 2r\}$, and stop the search inside v_i at nodes whose timestamp is newer than t.

Let us now consider nearest neighbor searching. An equivalent view of the above restriction focuses on the maximum allowed radius instead of maximum allowed timestamp, as follows. Let v_i be as above and y be a child of v_i. Node y must be considered if every sibling v_{i+j} of v_i such that $d(q,v_i) > d(q,v_{i+j}) + 2r$ is newer than y. Which is the same, y must be considered if $r_y = \max(d(q,v_i) - d(q,v_{i+j}))/2 \leq r$, where the maximization is done over those v_{i+j} older than y. Hence, r_y is a lower bound on r.

Assume that we are currently processing node v_i and insert its children y in the priority queue (Figure 7). We compute r_y as before and include it as a new lower bound on r (recall that we have already four lower bounds, Section 4.3).

7.2 Inserting at the Fringe

Another alternative is as follows. We can relax Property 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 by others close enough.

This seems to make the search time slightly worse, but the cost is marginal. The benefit is that we are not forced anymore to put a new inserted element x as a neighbor of a, even when Property 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. However, we have to permit the reconstruction of small subtrees so as to avoid that the tree becomes almost a linked list. So we permit inserting x as a neighbor when the size of the subtree to rebuild is small enough. This leads to a tradeoff between insertion cost and quality of the tree at search time.

Note that this scheme could be of interest for mapping the *sa-tree* to disk, as we can define the size of the subtree as that of a disk page, so reorganizations of the tree occur only inside one (leaf) disk page. Once a

disk page ceases to be a leaf, it becomes read-only. (This may have interest also for concurrent access to the data structure.) Furthermore, we can control the arity of the nodes, which can also be of use to have a regular structure at the internal nodes: note that we can artificially increase the arity of the tree by adding as neighbors elements that could be inserted into another neighbor. The exact tradeoff between few versus many neighbors is not totally understood yet, so having the choice of adding or not adding an element as a neighbor permits studying the optimality of the structure.

7.3 Experimental Results

We have performed some additional tests to show the practical performance of the alternatives discussed for incremental construction of the *sa-tree*. The experimental setup follows that of Section 6. We have chosen three of the metric spaces: 100,000 random vectors in dimension 15 under Euclidean distance, 86,061 strings under edit distance and 1,263 documents under cosine similarity. For each of these, we have measured the static and incremental construction cost, as well as the search performance of the structures built. This is by no means an exhaustive study but a set of tests to demonstrate the feasibility of the incremental construction.

In the tests that follow, "static" refers to the static construction as explained in the main body of the paper, "timestamping" to the timestamping technique, and "fringe(t)" to the mechanism of inserting at the fringe, on subtrees of size tn or less (where n is the size of the *final* set, e.g. Fringe(0.10%) on the vectors space rebuilds trees under 100 nodes). The dynamic versions build the tree by successive insertions.

Figure 15 and 16 compare the construction and search times, respectively. As can be seen, the dynamic versions are quite competitive. Timestamping costs a bit more at construction and search time (the difference is more significant for strings). Fringe(t) may cost even less than the static version at construction time, if t is small enough. As t grows its construction time quickly doubles that of the static version, although the benefit at search time of a more costly construction is barely noticeable. It is interesting that the fringe method can behave even better than the static method at search time. The reason is that in this case the tree is forced to be of smaller arity, which is advantageous for easier metric spaces or queries with smaller radii. This shows that, although the *sa-tree* adapts automatically to the dimension of the set, it does not necessarily find the best choice of arity (which is impossible because the best arity depends on the search radius). So the fringe method permits an optimization that is not possible in the static version.

Note that, in the case of documents, timestamping costs much less than the static method. The same happens to fringe(t), but this may be because since the n value here is about 1/4 of the other spaces, rebuilding a subtree of the same percentage implies in practice rebuilding smaller subtrees. We have tried with fringe(0.3) but the construction cost went up to 1,800 evaluations per element. All their performances at search time, however, are slightly worse than the static version (even for fringe(0.4)).

Method	Metric space		
	Vectors	Strings	Docs.
Static	120.35	72.43	118.63
Timestamp	120.50	90.15	72.81
Fringe(0.05%)	101.21	69.70	49.44
Fringe(0.10%)	157.76	95.36	68.053
Fringe(0.20%)	252.15	131.45	104.91

Fig. 15 Construction time comparison between the static and dynamic versions, in terms of number of distance computations per element.

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 hard spaces (i.e. with concentrated histogram of distances) or hard queries (i.e. those of large search radii). These are the unsolved cases in proximity searching.

Some issues for future work 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 when the search problem is more difficult but is inferior to others when the problem is easier. Moreover, it cannot trade space for query time as pivoting schemes do. This enables the possibility of designing hybrid schemes that get the best of both cases. 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. We are already pursuing this line.
- It is interesting to try to reduce the backtracking, although our attempts up to now have failed. One

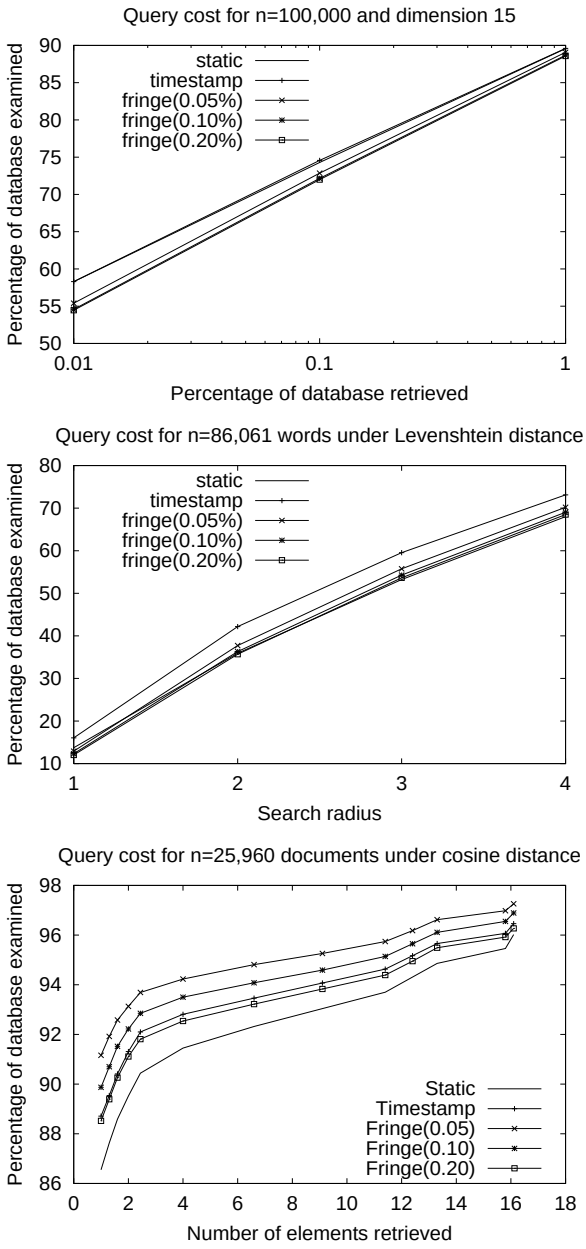


Fig. 16 Query time comparison between the static and dynamic versions, in terms of percentage of the database examined.

choice is to define a tolerance radius R and insert each element into its closest neighbor and any other that differs from it by at most R. The backtracking can now be done with tolerance $2(r - R)$. The structure is now a DAG (directed acyclic graph), not a tree, and its construction is much more complicated. In particular, our attempts lead to a (large) set of (small) DAGs rather than to a single DAG, and the search complexity on this set is not promising. A single DAG should work much better.

– We have presented solutions that permit dynamic insertions at low cost without degrading the performance, and in some cases even improving it. However, still more work is necessary to fully understand them. Some choices have even open the door to optimize the structure by choosing whether or not to add a neighbor. On the other hand, deletions have to be handled in order to have a fully dynamic data structure. This is our main focus at the moment [28].
– Secondary memory issues have not been considered yet. 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 technique of inserting at the fringe (Section 7.2), not only because the top of the tree may be read-only and the reorganizations be restricted to the leaf pages, but also because we can control the maximum arity of the tree so as to make the neighbors fit in a disk page. Our project [28] aims at a fully dynamic structure that can be efficiently handled in secondary memory.
– It would be interesting to build approximate or probabilistic algorithms based on this structure, as they have proved to be of great interest in extremely difficult metric spaces using other data structures that typically work well only on easier spaces [13]. We are also pursuing this line.
– 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 would permit us to reach any element from any other by always the distance among them. Although we have proved that such a structure cannot be built, other simplified structures, different from the *sa-tree*, could be created based on the spatial approximation approach.

Acknowledgements We are indebted to Gisli Hjaltason for finding tricky mistakes and proposing improvements in the search algorithms. We also wish to thank Nora Reyes for providing the code of the dynamic versions of the *sa-tree*.

References

1. F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.
2. J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
3. J. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, 5(4):333–340, 1979.
4. W. Burkhard and R. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, 1973.
5. 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).
6. 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.

7. 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.
8. R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.
9. 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.
10. 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.
11. 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.
12. 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.
13. E. Chávez and G. Navarro. A probabilistic spell for the curse of dimensionality. In *Proc. 3rd Workshop on Algorithm Engineering and Experiments (ALENEX'01)*, pages 147–160, LNCS 2153, 2001.
14. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 2001. To appear.
15. 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.
16. F. Dehne and H. Nolteimer. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987. Pergamon Journals.
17. 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.
18. 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.
19. G. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.
20. 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.
21. 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.
22. 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.
23. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys* 33(1):31–88, 2001.
24. 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.
25. H. Nolteimer. Voronoi trees and applications. In *Proc. International Workshop on Discrete Algorithms and Complexity*, pages 69–74, 1989.
26. G. Navarro and N. Reyes. Dynamic spatial approximation trees. In *Proc. XXI Conference of the Chilean Computer Science Society (SCCC'01)*, 2001. To appear. IEEE CS Press.
27. 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.
28. N. Reyes. *Dynamic data structures for searching metric spaces*. MSc. Thesis, Univ. Nac. de San Luis, Argentina, 2001. In progress. G. Navarro, advisor.
29. M. Shapiro. The choice of reference points in best-match file searching. *Communications of the ACM*, 20(5):339–343, 1977.
30. J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. Manuscript, 1991.
31. J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.
32. E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.
33. 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.
34. P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000.

A Average Number of Neighbors

We show in this appendix that the average number of neighbors (tree children) of a node, given n candidates, is $\Theta(\log n)$. Given the insertion process, if we have already k neighbors, then the probability that a new candidate becomes a new neighbor is a^k, for some $0 < a < 1$ (we use $1/2^k$ in the body of the paper but this is a simplification, so we prefer to be more general here). Hence, if we call $P_{n,k}$ the average number of new neighbors that are added from n candidates given that there are already k neighbors, the following recurrence holds:

$$P_{n+1,k} = a^k(1+P_{n,k+1})+(1-a^k)P_{n,k}\ , \qquad P_{0,k} = 0 \quad (1)$$

and we are interested in obtaining $P_{n,0}$.

It is possible to solve this recurrence exactly by using generating functions, more precisely $P_k(z) = \sum_{n\geq 0} P_{n,k} z^n$. The result is $P_{n,k} =$

$$1+\sum_{i=1}^{n}\sum_{k=1}^{n} a^{k(k+1)/2} \sum_{r_1+\ldots+r_k=n-k} \binom{n-k}{r_1,\ldots,r_k} \prod_{j=1}^{k}(1-a^j)^{r_j}$$

which is quite difficult to grasp.

We take a different approach. From the simplified analysis in the paper, we suspect that $P_{n,0} = \Theta(\log n)$, and therefore try to prove it by induction on n. The problem is to guess a formula for the general case $P_{n,k}$, which is especially difficult because it has to be decreasing on k and $P_{0,k} = 0$ must hold. After some attempts, we arrive at

$$P_{n,k} \leq c\ln(n+1) - \min(k, d\ln(n+1))$$

for positive c and d. This satisfies the base case $n = 0$ for all k. Now, we replace in Recurrence (1) the $P_{n,*}$ values by our bound and try to obtain the same bound for $P_{n+1,*}$. We have to prove

$$\begin{aligned} & a^k(1 + c\ln(n+1) - \min(k+1, d\ln(n+1))) \\ & +(1-a^k)(c\ln(n+1) - \min(k, d\ln(n+1))) \\ & \leq c\ln(n+2) - \min(k, d\ln(n+2)) \end{aligned}$$

which must be split in four cases:

1. $k \leq d\ln(n+1) - 1$. In this case we have to prove

   $$a^k(1 + c\ln(n+1) - (k+1)) + (1-a^k)(c\ln(n+1) - k) \leq c\ln(n+2) - k$$

   which simplifies to $c\ln(n+1) \leq c\ln(n+2)$, always true.
2. $d\ln(n+1) - 1 \leq k \leq d\ln(n+1)$. In this case we have to prove

   $$a^k(1 + c\ln(n+1) - d\ln(n+1)) + (1-a^k)(c\ln(n+1) - k) \leq c\ln(n+2) - k$$

   which simplifies to

   $$a^k + c\ln(n+1) - a^k(d\ln(n+1) - k) \leq c\ln(n+2)$$

   Now, since $k \geq d\ln(n+1) - 1$ we have $a^k \leq a^{d\ln(n+1)-1} = (n+1)^{d\ln a}/a$ (because $0 < a < 1$). Also, the expression $d\ln(n+1) - k$ is positive. So we can pessimistically try to prove

   $$\frac{1}{a}(n+1)^{d\ln a} \leq c\,(\ln(n+2) - \ln(n+1))$$

   and since $\ln(x) = \int_1^x dz/z$, we have that $\ln(n+2) - \ln(n+1) = \int_{n+1}^{n+2} dz/z$, which can be proven to lie between $1/(n+2)$ and $1/(n+1)$ by a simple geometric argument. Hence, we pessimistically try to prove

   $$\frac{1}{a}(n+1)^{d\ln a} \leq \frac{c}{n+2}$$

   which is true provided

   $$d \geq \frac{1}{\ln(1/a)}\,\frac{\ln(n+2)}{\ln(n+1)} + \frac{1}{\ln(n+1)}\left(1 - \frac{\ln c}{\ln(1/a)}\right) \quad (2)$$
3. $d\ln(n+1) \leq k \leq d\ln(n+2)$. In this case we have to prove

   $$\begin{aligned} & a^k(1 + c\ln(n+1) - d\ln(n+1)) \\ & +(1-a^k)(c\ln(n+1) - d\ln(n+1)) \\ & \leq c\ln(n+2) - k \end{aligned}$$

   which simplifies to

   $$a^k + c\ln(n+1) - d\ln(n+1) \leq c\ln(n+2) - k$$

   and we pessimistically replace a^k by $a^{d\ln(n+1)} = (n+1)^{d\ln a}$, as well as k by $d\ln(n+2)$. We are left with

   $$\begin{aligned} & (n+1)^{d\ln a} + d(\ln(n+2) - \ln(n+1)) \\ & \leq c(\ln(n+2) - \ln(n+1)) \end{aligned}$$

   which can again be pessimistically simplified to

   $$(n+1)^{d\ln a} + \frac{d}{n+1} \leq \frac{c}{n+2}$$

   from where we obtain a condition on c:

   $$c \geq \frac{n+2}{n+1}\,d + (n+2)(n+1)^{d\ln a} \quad (3)$$
4. $k \geq d\ln(n+2)$. In this case we have to prove

   $$\begin{aligned} & a^k(1 + c\ln(n+1) - d\ln(n+1)) \\ & +(1-a^k)(c\ln(n+1) - d\ln(n+1)) \\ & \leq c\ln(n+2) - d\ln(n+2) \end{aligned}$$

   which is simplified to

   $$a^k + c\ln(n+1) - d\ln(n+1) \leq c\ln(n+2) - d\ln(n+2)$$

   which is very similar to the previous case. Making the same pessimistic simplifications we arrive to $c \geq (n+2)^{1+d\ln a} + d(n+2)/(n+1)$ which is a bit less stringent than Eq. (3) (recall that $d\ln a < 0$).

We have succeeded to prove the hypothesis, and we analyze now the resulting conditions obtained for c and d. A pessimistic bound in Eq. (2) is to assume $c \geq 1$ and set

$$d = \frac{1 + \log_{1/a}(n+2)}{\ln(n+1)}$$

and by replacing this into Eq. (3) we get the surprisingly simple condition

$$c = \frac{n+2}{n+1}\,d + a$$

By replacing this c value into our initial hypothesis we get

$$\begin{aligned} P_{n,0} &\leq c\ln(n+1) \\ &= \frac{n+2}{n+1}\,(1 + \log_{1/a}(n+2)) + a\ln(n+1) \\ &= O(\log n) \end{aligned}$$

which is asymptotically $(a+1/\ln(1/a))\ln n$. If $a=1/2$ this is $1.35\log_2 n$.

We have to prove now that $P_{n,0} = \Omega(\log n)$. This time we need a lower bounding formula. The way we found is as follows: we prove that

$$P_{n,k} \geq \text{ if } \left(k \leq \log_{1/a}\frac{n+1}{c}\right) \text{ then } c\ln(n+1) \text{ else } dk$$

which satisfies $P_{n,0} \geq c\ln(n+1)$ provided $c \leq 1$. The proof by induction is split in three cases now:

1. $k+1 \leq \log_{1/a}((n+1)/c)$. In this case we have to prove

   $$a^k(1+c\ln(n+1))+(1-a^k)c\ln(n+1) \geq c\ln(n+2)$$

   which, using the same techniques as before, can be pessimistically simplified to

   $$a^k \geq \frac{c}{n+1}$$

   which is true given the condition of Case 1.
2. $\log_{1/a}((n+1)/c)-1 \leq k \leq \log_{1/a}((n+1)/c)$. In this case we need to prove

   $$a^k(1+d(k+1))+(1-a^k)c\ln(n+1) \geq c\ln(n+2)$$

   which can be pessimistically simplified to

   $$a^k(1+d(k+1))-a^k c\ln(n+1) \geq \frac{c}{n+1}$$

   and, given that under this case we have $c/(n+1) \leq a^k \leq c/(a(n+1))$, can be pessimistically further simplified to

   $$d(k+1) \geq \frac{c\ln(n+1)}{a}$$

   and using again that $k \geq \log_{1/a}((n+1)/c)-1$, we arrive at a condition on d

   $$d \geq \frac{c\ln(n+1)\ln(1/a)}{a\ln((n+1)/c)} \qquad (4)$$

3. $k \geq \log_{1/a}((n+1)/c)$. In this case we have to prove

   $$a^k(1+d(k+1))+(1-a^k)dk \geq dk$$

   which is immediate.

We analyze now the conditions on c and d. We had $c \leq 1$ initially, so we set $c=1$ as the best lower bound. From Eq. (4), we obtain $d = \ln(1/a)/a$. Hence, we have been able to prove $P_{n,0} \geq \ln(n+1)$, and therefore $P_{n,0} = \Theta(\log n)$.

B Recurrences of the Form $f(n) = g(f(n/\log n))$

We show first that the solution to the recurrence $H(n) = 1 + H(n/\log n)$ is $\Theta(\log n/\log\log n)$. For exactness, let us state the recurrence

$$H(n) = 1 + H\left(\frac{n}{\lceil \log_c n\rceil}\right)$$

and $H(c)=0$. Let us call $N=n$ the initial n value and assume for simplicity that $N=c^K$. In the interval $N/c < n \leq N$ we have $\lceil\log_c n\rceil = K$. Hence, in this range the recurrence is

$$H(n) = 1 + H(n/K) = i + H(n/K^i)$$

and this is true until it holds $n/K^i = N/c$, or $i = \log_K c$. Hence

$$H(N) = \log_K c + H(N/c)$$

now we repeat the argument in the area $N/c^2 < n \leq N/c$, where $\lceil\log_c n\rceil = K-1$ to obtain

$$\begin{aligned} H(N) &= \log_K c + \log_{K-1} c + H(N/c^2) \\ &= \ln c \times \sum_{i=2}^{K}\frac{1}{\ln i} = \ln c \int_2^K \frac{dx}{\ln x} + O(1) \end{aligned}$$

where for the last term we unrolled the recurrence. Finally,

$$\int_2^K \frac{dx}{\ln x} = \frac{K}{\ln K} + O\left(\frac{K}{\ln^2 K}\right)$$

(easy to get with any mathematical package) shows that

$$H(N) = K/\log_c K + O(1) = \log_c N/\log_c\log_c n + O(1)$$

Since we proved it for infinitely many values of N and the function does not grow fast enough between a pair of those values, it follows $H(n) = \Theta(\log n/\log\log n)$.

The solution for $B(n) = n\log n + \log n\ B(n/\log n)$ follows the same steps. This time we arrive to

$$\begin{aligned} B(N) &= Kc^K\log_K c + (K-1)c^K\log_{K-1} c + c^2 H(N/c^2) \\ &= N\ln c\sum_{i=2}^{K}\frac{i}{\ln i} = N\ln c\int_2^K \frac{x\,dx}{\ln x} + O(1) \end{aligned}$$

where

$$\int_2^K \frac{x\,dx}{\ln x} = \frac{K^2}{2\ln K} + O\left(\frac{K^2}{\ln^2 K}\right)$$

(again with the help of a mathematical package) shows that $B(N) = NK^2/(2\log_c K) + O(1) = N\log_c^2 N/(2\log_c\log_c n) + O(1)$. Hence $B(n) = \Theta(n\log^2 n/\log\log n)$.

Finally, the most complicated recurrence is $T(n) = \log n + F(2r)\log n\ T(n/\log n)$. Let us call $x = F(2r)$ to

abbreviate. With the same assumptions of the previous cases we have that for $N/c < n \leq N$

$$\begin{aligned} T(n) &= K + xKT(n/K) \\ &= K\frac{(xK)^i - 1}{xK - 1} + (xK)^i T(n/K^i) \\ &= \Theta(cx^{\log_K c}) + cx^{\log_K c} T(N/c) \end{aligned}$$

now considering the next area $N/c^2 < n \leq N/c$ we have

$$\begin{aligned} T(N) &= \Theta(cx^{\log_K c}) + \Theta(c^2 x^{\log_K c + \log_{K-1} c}) \\ &\quad + c^2 x^{\log_K c + \log_{K-1} c} T(N/c^2) \end{aligned}$$

so by unrolling we get

$$T(N) = \Theta\left(\sum_{i=1}^{K-1} c^i x^{\ln c \sum_{j=K-i+1}^{K} \frac{1}{\ln j}}\right)$$

Given our previous results,

$$\begin{aligned} \sum_{j=K-i+1}^{K} \frac{1}{\ln j} &= \int_{K-i}^{K} \frac{dx}{\ln x} + O(1) \\ &= \frac{K}{\ln K} - \frac{K-i}{\ln(K-i)} + O(1) \end{aligned}$$

so

$$\begin{aligned} T(N) &= \Theta\left(\sum_{i=1}^{K-1} c^i x^{\frac{K}{\log_c K} - \frac{K-i}{\log_c (K-i)}}\right) \\ &= \Theta\left(c^K x^{\frac{K}{\log_c K}} \sum_{i=1}^{K-1} \frac{1}{c^i x^{\frac{i}{\log_c i}}}\right) \end{aligned}$$

and since the last summation is clearly $O(1)$, we get

$$\begin{aligned} T(N) &= \Theta\left(c^K x^{\frac{K}{\log_c K}}\right) = \Theta\left(N x^{\frac{\log_c N}{\log_c \log_c N}}\right) \\ &= \Theta\left(N^{1 - \frac{\log_c (1/F(2r))}{\log_c \log_c N}}\right) \end{aligned}$$

hence $T(n) = \Theta(n^{1-\Theta(1/\log\log n)})$.