A Dynamic Pivoting Algorithm Based on Spatial Approximation Indexes

Diego Arroyuelo

Department of Informatics, Universidad Técnica Federico Santa María
Yahoo! Labs Santiago, Chile
darroyue@inf.utfsm.cl

Abstract. Metric indexes aim at reducing the amount of distance evaluations carried out when searching a metric space. Spatial approximation trees (*sa-trees* for short), in particular, are efficient data structures, which have shown to be competitive in metric spaces of medium to high difficulty, or queries with low selectivity. *Sa-trees* can be also made dynamic, and can use the available space to improve the query performance adding pivot information. In this paper we extend previous work on dynamic *sa-trees* with pivots, and show how the pivot information can be used to a full extent to improve the search performance. The result is a technique that allows one to traverse a dynamic *sa-tree* without necessarily comparing all traversed nodes against the query object. As a result, the novel algorithm makes a much better use of the available space, yielding a saving of distance computations of about 10% to 70%, compared with previous *sa-tree* schemes that use pivot information.

1 Introduction

The classical way of searching a database has been that of finding those database records whose search attribute (or *key*) has a given value. However, this is not suitable when searching non-traditional databases, such as multimedia databases (e.g., image, video, or audio), multidimensional vector spaces (which has applications in GIS), and digital libraries, among others. In such cases, one would want to find the database objects that are "similar" to a given query object.

The similarity search problem is usually modeled as *proximity searches in metric spaces*. In the *metric space model* [2], there is a universe $\mathcal{U}$ of objects, and a positive real-valued distance function $d : \mathcal{U} \times \mathcal{U} \to \mathbb{R}^+$ defined among them. The distance between two objects models their similarity: the smaller the distance is, the more similar the objects are. We assume that the distance satisfy the three axioms that make the set a metric space:

Strict positiveness: $\forall x, y \in \mathcal{U},\ d(x, y) = 0 \Leftrightarrow x = y$;
Symmetry: $\forall x, y \in \mathcal{U},\ d(x, y) = d(y, x)$; and
Triangle inequality: $\forall x, y, z \in \mathcal{U},\ d(x, z) \leqslant d(x, y) + d(y, z)$.

The triangle inequality property is used to save comparisons in a proximity query. The distance function is usually expensive to compute, hence we define the search complexity as the number of distance evaluations performed.

A.J. Machado Traina et al. (Eds.): SISAP 2014, LNCS 8821, pp. 70–81, 2014.
DOI: 10.1007/978-3-319-11988-5_7

We are given (in advance to queries) a database $S \subseteq \mathcal{U}$ of size $|S| = n$. Proximity-search algorithms are allowed to build an *index* of the database, avoiding exhaustive searches at query time [2]. Building an index is usually an expensive process. However, this cost is amortized after enough queries have been issued. At query time, given a query object $q \in \mathcal{U}$, we must retrieve all similar elements found in S. There are two typical queries of this kind:

Range Queries: retrieve all elements in S within distance r to q. That is, the set $\{x \in S, \quad d(x,q) \leqslant r\}$.

Nearest-Neighbor Queries: retrieve the k closest elements to q in S. That is, a set $A \subseteq S$ such that $|A| = k$ and $\forall x \in A, y \in (S - A), \quad d(x,q) \leqslant d(y,q)$.

In this paper we focus on range queries only.

Algorithms to search in general metric spaces can be divided into two large areas [2]: *pivoting* algorithms, and *compact partitions* algorithms. Pivoting algorithms are better suited for low dimensional (or easy) metric spaces, whereas compact partitions algorithms deal better with high dimensional (or hard) metric spaces. Although pivoting algorithms can use the available memory to improve the query performance, they need to use more memory to beat the latter as dimension grows. On the other hand, indexes based on compact partitions use a fixed amount of memory and cannot be improved by giving them more space.

Since a time ago, there are also data structures that combine both approaches, as for instance the memory-adaptive dynamic spatial approximation trees from [1]. These are basically dynamic spatial approximation trees (*dsa-trees*) [3], on which pivot information is added. Hence, they are able to trade memory space for a better query performance. However, pivots on *dsa-trees* are not used to prune the search nor to discard traversed elements. They are used just to save (in some cases) distance evaluations when the stopping criterion of *dsa-trees* determines that a given branch of the tree must be pruned. Every traversed node is inevitably compared against the query, even though it is not contained within the query radius. This obviously increases the query cost. Our research question is: What are the consequences for the spatial approximation approach, if we use pivot information to avoid comparing traversed elements?

In this paper we extend previous work [1] on *dsa-trees* with pivots, and show how the pivot information can be used to a full extent to improve the search performance. Basically, we adapt the search approach [1] such that pivots are used to avoid distance evaluation on traversed nodes. The resulting algorithms allows one to traverse a dynamic *sa-tree* without necessarily comparing all traversed nodes against the query. Avoiding such distance evaluations does not necessarily represents an improvement of query performance: we have less information for the spatial approximation, hence probably more tree branches would be visited. Ours is a compromise which probably traverses more tree branches, yet using pivots to avoid distance evaluations. We will show experimental results indicating that our approach uses the available memory more efficiently than previous work [1]: our search algorithm makes a better use of the available space, yielding a saving of distance computations of about 10% to 70%.

2 Preliminary Concepts on Metric Space Indexing

Indexing metric spaces is key for achieving efficient search performance in similarity search applications. We review in this section the most important indexing approaches needed to understand our work.

2.1 Pivoting Algorithms

Pivoting algorithms choose a set $\mathcal{P} = \{p_1, \ldots, p_k\}$ of *pivots* from the database S. They precompute and store all distances $d(a, p_1), \ldots, d(a, p_k)$ for all $a \in S$. Given a query $q \in \mathcal{U}$, pivoting algorithms compute the distances $d(q, p_1), \ldots, d(q, p_k)$ against the pivots. By using the information stored for every database object and the distances between the pivots and the query, we define:

Definition 1. *Given a query element $q \in \mathcal{U}$, the* pivot distance *between $a \in S$ and q gets defined as:*

$$\mathcal{D}(a, q) = \max_{p_i \in \mathcal{P}} |d(a, p_i) - d(q, p_i)|.$$

It can be proven that $\mathcal{D}(a, q) \leqslant d(a, q)$ for any $a \in S$, $q \in \mathcal{U}$. The pivot distance $\mathcal{D}$ is an estimation of the actual distance d, which is used to save distance evaluations: each a such that $\mathcal{D}(a, q) > r$ can be discarded because we deduce $d(a, q) > r$. All the elements that cannot be discarded in this way are directly compared against q.

Pivoting schemes perform better as more pivots are used, this way beating any other index. They are, however, better suited to "easy" metric spaces [2]. In hard spaces they need too many pivots to beat other algorithms.

2.2 Dynamic Spatial Approximation Trees

We briefly outline in this subsection how dynamic spatial approximation trees (*dsa-trees*) work, as we build on this data structure. See [3] for further details and proofs of correctness of the algorithms.

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

Range Search Algorithm. The idea is to replicate the insertion process of the elements to be retrieved. Given a query q and a radius r, we act as if we wanted to insert q but keep in mind that relevant elements may be at distance up to r from q, so in each decision for simulating the insertion of q we permit a tolerance of $\pm r$. So it may be that relevant elements were inserted in different children of the current node, and backtracking is necessary.

Note that, at the time an element x was inserted, a node a may not have been chosen as its parent because its arity was already maximal. So, at query time, we must choose the minimum distance to x only among $N(a)$. Note also that, when x was inserted, elements with higher timestamp were not yet present in the tree, so x could choose its closest neighbor only among older elements. Hence, we consider the neighbors $\{b_1, \ldots, b_k\}$ of a from oldest to newest, disregarding a, and perform the minimization as we traverse the list. That is, we enter into subtree b_i if $d(q, b_i) \leqslant \min\{d(q, b_1), \ldots, d(q, b_{i-1})\} + 2r$.

We use timestamps to reduce the work inside older neighbors. Say that $d(q, b_i) > d(q, b_{i+j}) + 2r$. We have to enter subtree b_i anyway because b_i is older. However, only the elements with timestamp smaller than $time(b_{i+j})$ should be considered when searching inside b_i; younger elements have seen b_{i+j} and they cannot be interesting for the search if they are inside b_i. As parent nodes are older than their descendants, as soon as we find a node inside subtree b_i with timestamp larger than $time(b_{i+j})$ we can stop the search in that branch.

Algorithm 1 performs range searching on a *dsa-tree*. Note that, except in the first invocation, $d(a, q)$ (lines 1 and 2) is already known from the invoking process, so it must no be recomputed in a real implementation.

Algorithm 1. *dsat* Search(Node a, Query q, Radius r, Timestamp t).

```
 1: if time(a) < t ∧ d(a,q) ⩽ R(a) + r then
 2:    if d(a,q) ⩽ r then
 3:       report a
 4:    end if
 5:    d_min ← +∞
 6:    for b_i ∈ N(a) in increasing timestamp order do
 7:       if d(b_i,q) ⩽ d_min + 2r then
 8:          k ← min {j > i,  d(b_i,q) > d(b_j,q) + 2r}
 9:          dsat Search(b_i, q, r, time(b_k))
10:       end if
11:       d_min ← min {d_min, d(b_i,q)}
12:    end for
13: end if
```

2.3 DSA-Trees with Pivots

Previous work [1] showed how to use the available memory to improve the search performance of *dsa-trees*. We associate a set of pivots to every tree node. At insertion time, in order to decide that a new element x must be added as a

children (or neighbor) of an already existing node a, note that x has been already compared against the set $\mathcal{A}(x)$ of ancestors of x, and also against the siblings of the ancestors. Some of these distances are used as pivot information, without introducing extra distance computations. See the original work [1], which shows how these pivots are computed at insertion time. From now on, we assume that each node x of a *dsa-tree* has a set $\mathcal{P}(x)$ of pivots. The resulting data structure is called *hybrid dsa-tree* (H-DSAT for short).

Range Search Algorithm. dsa-tree Algorithm 1 is modified to use the set $\mathcal{P}(x)$ stored at each tree node x. Recall that, given a set of pivots, $\mathcal{D}(a, q)$ is a lower bound for $d(a, q)$. Consider again Algorithm 1. If at line 1 it holds that $\mathcal{D}(a, q) > R(a) + r$, then surely $d(a, q) > R(a) + r$, and hence we can stop the search at node a without actually evaluating $d(a, q)$. This leads to the following definition.

Definition 2. *An element a in S is said to be* covering radius feasible *(*cr-feasible *for short) for query q if $\mathcal{D}(a, q) \leqslant R(a) + r$. The set of* cr-feasible neighbors *of a node a is a subset of $N(a)$, and will be denoted by* cr-$F(a)$.

Also, we use $\mathcal{D}$ along with the *hyperplane criterion* to save distance computations at search time: for any cr-feasible element b_i such that $\mathcal{D}(b_i, q) > d_{min} + 2r$, it holds that $d(b_i, q) > d_{min} + 2r$. Hence, we can stop the search in the cr-feasible node b_i without evaluating $d(b_i, q)$ (at line 5 of Algorithm 1).

Definition 3. *Let* cr-$F(a)$ *be the set $\{b_1, \ldots, b_k\}$, in increasing order of timestamp. An element $b_i \in$* cr-$F(a)$ *is said to be* hyperplane feasible *(*h-feasible *for short) for query q if $\mathcal{D}(b_i, q) \leqslant d_{min} + 2r$, where d_{min} is minimized using only the distances $d(b_1, q), \ldots, d(b_{i-1}, q)$ that have been computed in the current query.*

Definition 4. *The* feasible neighbors *of node a, denoted $F(a)$, are the cr-feasible plus the h-feasible neighbors $b \in N(a)$. The other neighbors of a are said to be* infeasibles.

Note that only feasible neighbors of a node a must be taken into account when processing a query. The remaining subtrees can be discarded completely using $\mathcal{D}$ rather than d. However, it does not immediately follow that we obtain for sure an improvement in search performance. The reason is that infeasible nodes still serve to reduce d_{min} in Algorithm 1, which in turn may save us entering into younger siblings. Hence, by saving computations against infeasible nodes, we may have to enter into new siblings later. This is an intrinsic price of our method. At search time, $\mathcal{D}(a, q)$ can be computed without additional evaluations of d for any a in the data structure. A query stack is used to maintain the distances between the query object and the pivots as we backtrack the tree (see [1] for details). Algorithm 2 shows the first basic approach for range search on a H-DSAT.

However, in order to use timestamp information as much as possible in line 8, we run into the risk of comparing infeasible elements against q. this reduces the benefits of pivots in the data structure. Some improvements to this weakness were presented [1], being the best one as follows.

Algorithm 2. H-DSAT Search(Node a, Query q, Radius r, Timestamp t)

```
1: if time(a) < t ∧ d(a,q) ⩽ R(a) + r then
2:    if d(a,q) ⩽ r then
3:       report a
4:    end if
5:    d_min ← +∞
6:    cr-F(a) ← {b ∈ N(a), D(b,q) ⩽ R(b) + r}
7:    for b_i ∈ N(a) in increasing timestamp order do
8:       if b_i ∈ cr-F(a) ∧ D(b_i,q) ⩽ d_min + 2r then
9:          if d(b_i,q) ⩽ d_min + 2r then
10:            k ← min {j > i,  d(b_i,q) > d(b_j,q) + 2r}
11:            H-DSAT Search(b_i, q, r, time(b_k))
12:         end if
13:      end if
14:      if d(b_i,q) has already been computed then
15:         d_min ← min {d_min,  d(b_i,q)}
16:      end if
17:   end for
18: end if
```

Using Timestamps of Feasible Neighbors. The use of timestamps is not essential for the correctness of the algorithms. Any larger value would work, although the optimal choice is to use the smallest correct timestamp. Another alternative is to compute a safe approximation to the correct timestamp, but ensuring that no infeasible elements are ever compared against q. Note that every feasible neighbor of a node will be compared against q inevitably. If for $b_i \in F(a)$ it holds that $d(b_i, q) \leqslant d_{min} + 2r$, then we compute the oldest timestamp t among the reduced set $\{b_{i+j} \in F(a), \quad d(b_i, q) > d(b_{i+j}, q) + 2r\}$, and stop the search inside b_i at nodes whose timestamp is newer than t. This ensures that only feasible elements are compared against q, and under that condition it uses as much timestamping information as possible. This alternative is called H-DSATF.

3 Reducing the Cost of Traversing an H-DSAT

H-DSATs [1] use the available memory space to improve the search performance of *dsa-trees*. However, their search algorithms use pivots only to check the spatial approximation stopping criteria (that is, the covering-radius and hyperplane feasibility, see line 8 of Algorithm 2). This means that all traversed nodes are inevitably compared against q, even though for some element x in the search path it holds that $\mathcal{D}(x, q) > r$. The question is, therefore, whether we can improve the search cost if we avoid comparing elements in the search path of a *dsa-tree*. However, and as we will see, saving distances in this way is not for free. When the distance among the query and a traversed node is not computed, many search criteria would need to be relaxed, as we will see. Hence, it is not clear whether we will obtain an improvement or not.

To answer this question, we define a new search alternative for H-DSAT that avoids computing $d(x,q)$ whenever $\mathcal{D}(x,q) > r$ holds. Assume that, at search time, we reach the node a of the tree. For each $b_i \in N(a)$ in increasing order of timestamp, we perform the following steps:

Step 1: If $\mathcal{D}(b_i,q) > R(b_i) + r$ or $\mathcal{D}(b_i,q) > d_{min} + 2r$, prune the search at b_i since it is infeasible; otherwise, go to the next step.

Step 2: If $\mathcal{D}(b_i,q) > r$, b_i is not within the query radius. Therefore, we search inside the subtree of b_i *without* evaluating $d(b_i,q)$. Thus, all the descendants of b_i cannot use it as a pivot in the current query. We mark this fact by pushing an invalid distance into the query stack [1]. As $d(b_i,q)$ has not been computed, we cannot check whether $d(b_i,q) > d(b_{i+j},q) + 2r$ holds. Therefore we cannot search for the timestamp of a younger sibling of b_i to search inside the subtree of b_i (step 10 of Algorithm 2). In order to use timestamp information even in this case, if we reach b_i searching for elements with timestamp older than t, then we also use t to search inside the subtree of b_i. This is a correct (although not optimal) timestamp to search inside b_i.

Step 3: On the other hand, if $\mathcal{D}(b_i,q) \leqslant r$, we compute $d(b_i,q)$, and we report b_i if it lies within the search radius. Also, we try to prune the search using the covering radius and hyperplane criterions: if $d(b_i,q) > R(b_i) + r$ or $d(b_i,q) > d_{min} + 2r$, the search can be pruned at b_i. If the search cannot be prunned at b_i, we compute the oldest timestamp t among the set $\{b_{i+j} \in F(a), \mathcal{D}(b_{i+j},q) \leqslant r \wedge d(b_i,q) > d(b_{i+j},q) + 2r\}$, and stop the search inside b_i at nodes whose timestamp is newer than t.

We call H-DSATP this search alternative, which is formalized in Algorithm 3. Notice that we have added an extra parameter *dist*, which is the value $d(a,q)$ in case a has not been discarded using pivots, otherwise $dist = 0$ holds (see line 10). Let us take a look also at the condition in line 1: every time $\mathcal{D}(a,q) > r$ holds, it also holds that $dist = 0$, hence condition $dist \leqslant R(a) + r$ is true in these cases. Thus, only the timestamp condition $time(a) < t$ can be used to prune the search when $\mathcal{D}(a,q) > r$ holds.

Notice that our algorithm can be regarded as a pivoting scheme that uses the spatial approximation approach to prune the search space. This has the additional advantage of reducing the overhead incurred when computing $\mathcal{D}$ (which uses to be high for pure pivoting algorithms [2]).

When an element b_i is not compared against the query q (lines 9 and 10), the descendants of b_i cannot use it as a pivot. As a result, the value of $\mathcal{D}$ for these descendants can become underestimated, which is obviously a drawback. However, this gives the data structure the potential to adapt itself to the difficulty of the metric space and "decide" the number of pivots used for each element.

Algorithm 3. H-DSATP Search(Node a, Query q, Radius r, Timestamp t, distance $dist$)

```
// dist is d(a,q) in case it has been computed, 0 otherwise.

 1: if time(a) < t ∧ dist ⩽ R(a) + r then
 2:    if 𝒟(a,q) ⩽ r ∧ dist ⩽ r then
 3:       report a
 4:    end if
 5:    d_min ← +∞
 6:    cr-F(a) ← {b ∈ N(a), 𝒟(b,q) ⩽ R(b) + r}
 7:    for b_i ∈ N(a) in increasing timestamp order do
 8:       if b_i ∈ cr-F(a) ∧ 𝒟(b_i,q) ⩽ d_min + 2r then
 9:          if 𝒟(b_i,q) > r then
10:             H-DSATP Search(b_i, q, r, t, 0)
11:          else
12:             if d(b_i,q) ⩽ d_min + 2r then
13:                k ← min {j > i, b_j ∈ F(a) ∧ 𝒟(b_j,q) ⩽ r ∧ d(b_i,q) > d(b_j,q) + 2r}
14:                H-DSATP Search(b_i, q, r, time(b_k), d(b_i,q))
15:             end if
16:          end if
17:       end if
18:       if d(b_i,q) has been already computed then
19:          d_min ← min {d_min, d(b_i,q)}
20:       end if
21:    end for
22: end if
```

If for an element $b_j \in F(a)$ it holds that $\mathcal{D}(b_j, q) > r$, hence b_j cannot be used to minimize d_{min} when searching inside the subtree of an element $b_i \in F(a)$ younger than b_j: the condition $d(b_i, q) > d(b_j, q) + 2r$ implies computing $d(b_j, q)$. Since we know that $d(b_j, q) > r$, we will prefer not to use b_j to minimize d_{min}, saving the distance computation. This is a relaxation to the original spatial approximation approach. Line 13 of Algorithm 3 shows this formally. Also, every time $d(b_i, q)$ is computed (Step 3 above), we take full advantage of this evaluation by using the pruning criterion of the original *dsa-trees*, we use $d(b_i, q)$ to minimize d_{min}, and later, the descendants of b_i can use it as a pivot.

H-DSATP might traverse more nodes of the data structure than the original H-DSATs, because if $\mathcal{D}(b_i, q) \leqslant R(b_i) + r$, $\mathcal{D}(b_i, q) \leqslant d_{min} + 2r$, and $\mathcal{D}(b_i, q) > r$, then we have not computed $d(b_i, q)$ and the search must continue in the subtree of b_i. However, it might be that $d(b_i, q) > R(b_i) + r$ or $d(b_i, q) > d_{min} + 2r$, and the search would have stopped at b_i. That is, the cost of traversing a node is, in some cases, less expensive, but we may traverse more nodes than the original H-DSATs. The experiments of the next section will show that, despite the possible drawbacks we have remarked, in general it pays off to use $\mathcal{D}$ to exchange more traversed nodes for a smaller total cost.

4 Experimental Results

For the experiments of this paper we have considered range queries on four widely different metric spaces.

NASA images: a set of 40,700 feature vectors of dimension 20, generated from images downloaded from NASA [1]. The Euclidean distance is used. This is an easy space (sparse histogram of distances). For this space we use radii 0.605740, 0.780000 and 1.009000, which retrieve on average 0.01%, 0.1%, and 1% of the database respectively.

Words: a dictionary of 69,069 English words [2]. We use the *edit* or *Levenshtein* distance, that is, the minimum number of character insertions, deletions and replacements needed to make two strings equal. This distance is useful in text retrieval to cope with spelling, typing and optical character recognition (OCR) errors. The space turns out to be of low to medium difficulty. As the distance is discrete, we use radii 1 to 4, which retrieve on average 0.00003%, 0.00037%, 0.00326% and 0.01757% of the dataset, respectively

Color histograms: a set of 112,682 color histograms (112 dimensional vectors) from an image database [3]. Any cuadratic form can be used as a distance, so we chose Euclidean distance as the simplest meaningful alternative. The resulting space is of medium difficulty. For this space we use radii 0.051768, 0.082514 and 0.131163, which retrieve on average 0.01%, 0.1%, and 1% of the database respectively.

Documents: a set of 1,265 documents under the Cosine similarity, heavily used in Information Retrieval. In this model the space has one coordinate per term and documents are seen as vectors in this high-dimensional space. The distance we use is the angle (arccos of inner product) among the vectors. The documents are the files of the TREC-3 collection [4]. This is a space of medium to high difficulty, and the distance is expensive to compute. For this space we use radii 0.140000, 0.150000 and 0.195000, retrieving on average 1, 2, and 16 documents respectively.

For all these metric spaces, we build the indexes 10 times using 90% of the database elements, leaving the remaining 10% (randomly chosen) as queries. We test with arities 4, 8, 16 and 32 in the tree [3]. Due to lack if space, we show results only for the arity that produced the best results in each case.

We will suffix "1" the versions of H-DSAT that use the ancestors as pivots, and will use "2" for the versions that use the ancestors and their older siblings as pivots [1]. Hence, the alternative proposed in this paper will have two instances, H-DSATP1 and H-DSATP2. We will compare against the best alternative in previous work [1]: H-DSATF (the resulting instances are H-DSATF1 and H-DSATF2).

[1] `http://www.sisap.org/library/metricSpaces/dbs/vectors/nasa.tar`
[2] `http://www.sisap.org/library/metricSpaces/dbs/strings/dictionaries.tar`
[3] `http://www.sisap.org/library/metricSpaces/dbs/vectors/colors.tar`
[4] `http://trec.nist.gov`

Figure 1 shows the experimental query cost for search variants of H-DSAT using just ancestors as pivots. In all metric spaces we tested, H-DSATP1 performs better with arity 4. This is because the pivot information is more heavily used in this alternative, hence having small arity makes the three higher, hence each element has a bigger amount of pivots. H-DSATF1, on the other hand, uses the spatial approximation idea as much as it can. Hence, it performs better using arity 16 (except for the space of documents, where H-DSATF1 has the best performance using arity 32).

In the experiments, H-DSATP1 outperforms H-DSATF1, in many cases considerably. In the space of NASA images, we obtain about 15% (large radius) to 30% (small radius) less distance evaluations at query time. For color histograms, the improvements are from 28% to 40%. For the dictionary of English words, from 11% to 35%. Finally, for documents from 10% to 11%. The best improvements are obtained for small radii —since the problem is easier in these cases, on which pivots are more effective— and easier metric spaces —e.g., color histograms.

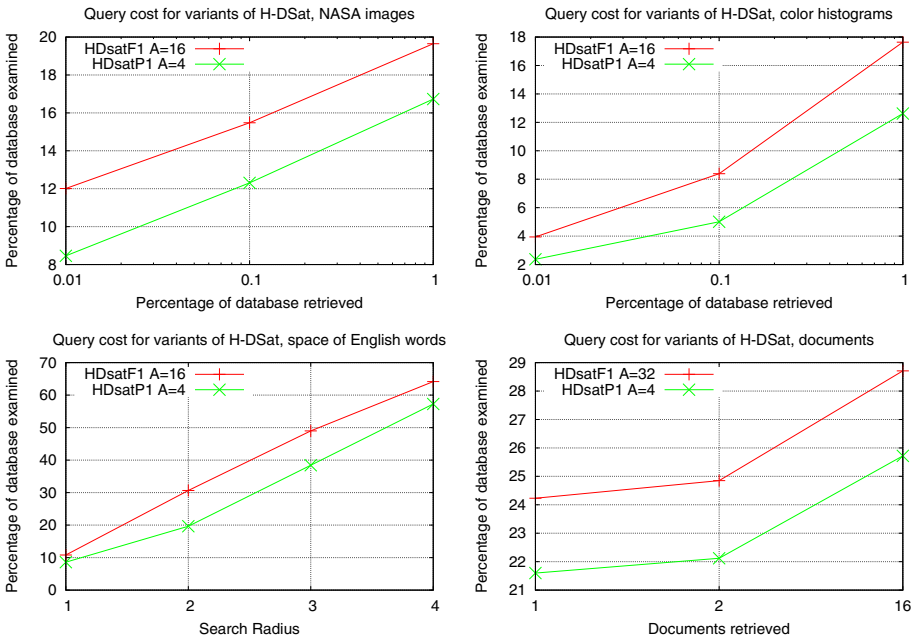

Fig. 1. Experimental query cost for different search alteratives of H-DSAT1

Figure 2 shows the experimental query cost for variants of H-DSAT2. As it can be seen, H-DSATF2 and H-DSATP2 perform better with arity 32 (except in the space of documents, where H-DSATP2 performs better using arity 16). As before, H-DSATP2 outperforms H-DSATF2, obtaining better improvements compared with the former alternatives. In the space of NASA images, we obtain

about 14% (large radius) to 37% (small radius) less distance computations, for color histograms 35% to 68%, for English dictionary 20% to 77%, and for the document database 13% to 17%.

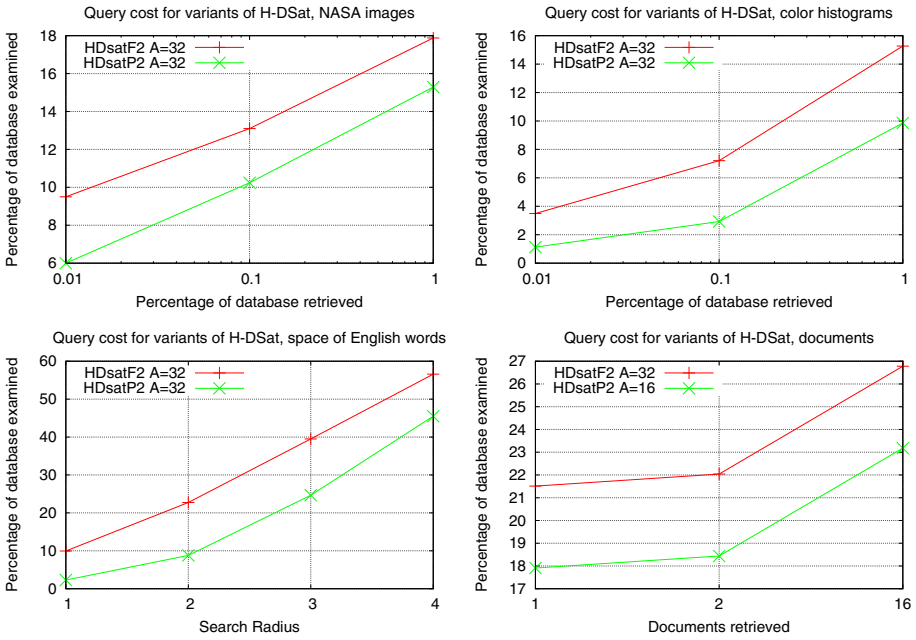

Fig. 2. Experimental query cost for different search alternatives of H-DSAT2

An important result that must be also considered is that comparing the results in Figures 1 and 2, we can conclude that HDSATP1 outperforms HDSATF2 in all cases, even though the former uses less pivots per node than the latter [1]. This reinforces the fact that our algorithm makes a better use of pivots, compared to the algorithms proposed in [1].

Finally, we obtained the following results regarding the number of traversed nodes by our algorithm. In the space of NASA images, H-DSATP1 traverses from about 23% (small radius) to 28% (large radius) of the tree nodes. This is 1.92 and 1.45 times the number of nodes traversed by H-DSATF1. For color histograms, H-DSATP1 traverses from about 17% (small radius) to 28% (large radius) of the tree nodes. This is 4.34 and 1.62 times the number of nodes traversed by H-DSATF1, respectively. For the English dictionary, H-DSATP1 traverses from about 44% ($r = 1$) to 77% ($r = 4$) of the tree nodes. This is 4.04 and 1.19 times the number of nodes traversed by H-DSATF1. For H-DSATP2, the results are similar, yet smaller than those of H-DSATP1.

Note that, even though H-DSATP traverses more nodes of the data structure than the original *dsa-tree* data structures, the total number of traversed nodes is

a relatively small fraction of the whole tree. This is important in cases where one wants to reduce the overhead incurred by traversing the whole database. Tree-based pivoting schemes are specifically good for this matter. However, given a fixed amount of storage, they must encode the tree structure, hence using space that the array-based indexes could use just for pivots (hence, storing a bigger number of pivots, improving the overall search performance). Nowadays, however, trees (even dynamic ones, as in our case) can be encoded using about 2 bits per node [4].

Our results clearly indicate a trend: for small radii and easier spaces (e.g., color histograms), we obtain the best improvements over H-DSATF (in number of distance evaluations), yet the number of traversed nodes by H-DSATP is higher than for H-DSATF. This is because in such cases our algorithm behaves like a pivoting scheme in these cases. For large radii and more difficult spaces, on the other hand, the improvements over H-DSATF are moderate (yet important), and the number of traversed nodes is similar to H-DSATF. This is because in these cases the data structure tends to behave as *dsa-trees*.

5 Conclusions

From our experimental results, we conclude that it is worth to relax some spatial approximation criteria (hence probably traversing more *dsa-tree* nodes) provided pivot information is used at every tree node as we propose. The search algorithm we proposed in this paper makes a better use of the available memory space used by pivots in *dsa-trees*. Compared with previous approaches [1] that use pivots on *dsa-trees*, our range search algorithm carries out from 10% to 70% less distance evaluations at query time. Our best improvements on previous results [1] were obtained in cases of small radii and easier spaces.

Our experimental results seem to indicate that our algorithm is adaptive to the difficulty of the search: on easier cases (i.e., easier metric spaces and small query radii) the data structure tends to behave as a pivoting algorithm; on harder cases (i.e., harder metric spaces and large radii), the data structure behaves like a *dsa-tree*, which are known to be more resistant to hard spaces. This deserves future research.

References

1. Arroyuelo, D., Muñoz, F., Navarro, G., Reyes, N.: Memory-adaptative dynamic spatial approximation trees. In: Nascimento, M.A., de Moura, E.S., Oliveira, A.L. (eds.) SPIRE 2003. LNCS, vol. 2857, pp. 360–368. Springer, Heidelberg (2003)
2. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)
3. Navarro, G., Reyes, N.: Dynamic spatial approximation trees. ACM Journal of Experimental Algorithmics (JEA) 12:article 1.5, 68 pages (2008)
4. Navarro, G., Sadakane, K.: Fully-functional static and dynamic succinct trees. ACM Transactions on Algorithms 10(3):article 16 (2014)