

Antipole Tree Indexing to Support Range Search and K -Nearest Neighbor Search in Metric Spaces

Domenico Cantone, Alfredo Ferro, Alfredo Pulvirenti, Diego Reforgiato Recupero, and Dennis Shasha

Abstract—Range and k -nearest neighbor searching are core problems in pattern recognition. Given a database S of objects in a metric space M and a query object q in M , in a range searching problem the goal is to find the objects of S within some threshold distance to q , whereas in a k -nearest neighbor searching problem, the k elements of S closest to q must be produced. These problems can obviously be solved with a linear number of distance calculations, by comparing the query object against every object in the database. However, the goal is to solve such problems much faster. We combine and extend ideas from the M-Tree, the Multivantage Point structure, and the FQ-Tree to create a new structure in the “bisector tree” class, called the *Antipole Tree*. Bisection is based on the proximity to an “Antipole” pair of elements generated by a suitable linear randomized tournament. The final winners a, b of such a tournament are far enough apart to approximate the diameter of the splitting set. If $\text{dist}(a, b)$ is larger than the chosen cluster diameter threshold, then the cluster is split. The proposed data structure is an indexing scheme suitable for (exact and approximate) best match searching on generic metric spaces. The Antipole Tree outperforms by a factor of approximately two existing structures such as List of Clusters, M-Trees, and others and, in many cases, it achieves better clustering properties.

Index Terms—Indexing methods, similarity measures, information search and retrieval.

1 INTRODUCTION

SEARCHING is a basic problem in metric spaces. Hence, much efforts have been spent both in clustering algorithms, which are often included in the searching process as a preliminary step (see BIRCH [53], DBSCAN [24], CLIQUE [3], BIRCH* [27], WaveClusters [46], CURE [32], and CLARANS [41]), and in the development of new indexing techniques (see, for instance, MVP-Tree [9], M-Tree [22], SLIM-Tree [48], FQ-Tree [4], List of Clusters [16], and SAT [40]; the reader is also referred to [18] for a survey on this subject). For the special case of Euclidean spaces, one can see [2], [29], [8], X-Tree [7], and CHILMA [47].

We combine and extend ideas from the M-Tree, MVP-Tree, and FQ-Tree structures together with randomized techniques coming from the approximate algorithms community [6], to design a simple and efficient indexing scheme called *Antipole Tree*. This data structure is able to support range queries and k -nearest neighbor queries in generic metric spaces.

The Antipole Tree belongs to the class of “bisector trees” [18], [13], [42], which are binary trees whose nodes represent sets of elements to be clustered. Its construction begins by first allocating a root r and then selecting two splitting points c_1, c_2 in the input set, which become the children of r . Subsequently, the points in the input set are partitioned according to their proximity to the points c_1, c_2 .

Then, one recursively constructs the tree rooted in c_i associated with the partition set of the elements closer to c_i , for $i = 1, 2$.

A good choice for the pair (c_1, c_2) of splitting points consists of maximizing their distance. For this purpose, we propose a simple approximate algorithm based on tournaments of the type described in [6]. Our tournament is played as follows: At each round, the winners of the previous round are randomly partitioned into subsets of a fixed size τ and their 1-medians¹ are discarded. Rounds are played until one is left with less than 2τ elements. The farthest pair of points in the final set is our Antipole pair of elements.

The paper is organized as follows: In the next section, we give the basic definitions of range search and k -nearest neighbor queries in general metric spaces and we briefly review relevant previous work, with special emphasis on those structures which have been shown to be the most effective, such as List of Clusters [16], M-Trees [22], and MVP-Trees [9]. The Antipole Tree is described in Section 3. Techniques to compute the approximate 1-Median and the diameter of a subset of a generic metric space are illustrated, respectively, in Sections 3.1 and 3.2. In Section 4, we present a procedure for range searching on the Antipole Tree. Section 5 presents an algorithm for the exact k -nearest neighbor problem. The Antipole Tree is experimentally compared with List of Clusters, M-Tree, and MVP-Tree in Section 6. In particular, cluster diameter threshold tuning is discussed. An approximate k -nearest neighbor algorithm is also introduced in Section 7 and a comparison with the version for approximate search of List of Clusters [12] is given with a precision-recall analysis. In Section 8, we deal with the problem of the curse of

- D. Cantone, A. Ferro, A. Pulvirenti, and D.R. Reforgiato are with the Dipartimento di Matematica e Informatica, Università degli Studi di Catania, Italy, Viale Andrea Doria n. 6 95125 Catania.
E-mail: {cantone, ferro, apulvirenti, diegoref}@dmi.unict.it.
- D. Shasha is with the Computer Science Department, New York University, 251 Mercer Street, New York, NY 10012.
E-mail: shashi@cs.nyu.edu.

Manuscript received 27 Aug. 2003; revised 9 Apr. 2004; accepted 14 Sept. 2004; published online 17 Feb. 2005.

For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0160-0803.

1. We recall that the 1-median of a set of points S in a metric space is an element of S whose average distance from all points of S is minimal.

dimensionality. Indeed, in high dimension, linear scan for uniform data sets may become competitive with the best searching algorithms. However, most of the real-world data sets are nonuniform. We successfully compare our algorithm with linear scan in nonuniform data sets of very high-dimensional Euclidean spaces. We draw our conclusions in Section 9. Finally, the Appendix which can be found on the Computer Society Digital Library at <http://computer.org/tkde/archives.htm> proposes an efficient approximation scheme for the diameter computation in the Euclidean case.

2 BASIC DEFINITIONS AND RELATED WORKS

Let M be a nonempty set of objects and let $dist: (M \times M) \rightarrow \mathbb{R}$ be a function such that the following properties hold:

1. $(\forall x, y \in M) dist(x, y) \geq 0$ (positiveness);
2. $(\forall x, y \in M) dist(x, y) = dist(y, x)$ (symmetry);
3. $(\forall x \in M) dist(x, x) = 0$ (reflexivity) and $(\forall x, y \in M) (x \neq y \rightarrow dist(x, y) > 0)$ (strict positiveness);
4. $(\forall x, y, z \in M) dist(x, y) \leq dist(x, z) + dist(z, y)$ (triangle inequality);

then, the pair $(M, dist)$ is called a *metric space* and $dist$ is called its *metric function*. Well-known metric functions include Manhattan distance, Euclidean distance, string edit distance, or the shortest path distance through a graph. Our goal is to build a low cost data structure for the range search problem and k -nearest neighbor searching in metric spaces.

Definition 2.1 (Range query). *Given a query object q , a database S , and a threshold t , the Range Search Problem is to find all objects $\{o \in S | dist(o, q) \leq t\}$.*

Definition 2.2 (k -Nearest Neighbor query). *Given a query object q and an integer $k > 0$, the k -Nearest Neighbor Problem is to retrieve the k closest elements to q in S .*

Our basic cost measure is the number of distance calculations, since these are often expensive in metric spaces, e.g., when computing the editing distance among strings.

Three main sources of ideas have contributed to our work. The FQ-Tree [4], an example of a structure using pivots (see [18] for an extended survey), organizes the items of a collection ranging over a metric space into the leaves of a tree data structure. Viewed abstractly, FQ-Trees consist of a vector of reference objects r_1, \dots, r_k and a distance vector v_o associated with each object o such that $v_o[i] = dist(o, r_i)$. A query object q computes a distance to each reference object, thus obtaining a v_q . Object o cannot be within a threshold distance t from q if for any i , $v_q[i] > v_o[i] + t$. That is, even if o is closer to q than r_i , q cannot be closer to o than t .

We use a similar idea except that our reference objects are the centroids of clusters.

M-Trees [22], [20] are dynamically balanced trees. Nodes of an M-Tree store several items of the collection provided that they are “close” and “not too numerous.” If one of these conditions is violated, the node is split and a suitable subtree originating in the node is recursively constructed. In the M-Tree, each parent node corresponds to a cluster with a radius and every child of that node corresponds to a subcluster with a smaller radius. If a centroid x has a distance $dist(x, q)$ from the query object and the radius of the cluster is r , then the entire cluster corresponding to x can be discarded if $dist(x, q) > t + r$.

We take the idea that a parent node corresponds to a cluster and its children nodes are subclusters of that parent cluster from the M-Tree. The main differences between our algorithm and the M-Tree are the construction method, that clusters in the M-Tree must have a limited number of elements, and the search strategy as our algorithm produces a binary tree data structure.

VP-Trees ([49], [52]) organize items coming from a metric space into a binary tree. The items are stored both in the leaves and in the internal nodes of the tree. The items stored in the internal nodes are the “vantage points.” To process a query requires the computation of the distance between the query point and some of the vantage points. The construction of a VP-Tree partitions a data set according to the distances that the objects have with respect to a reference point. The median value of these distances is used as a separator to partition objects into two balanced subsets (those as close or closer than the median and those farther than the median). The same procedure can recursively be applied to each of the two subsets.

The Multi-Vantage-Point tree [9] is an intellectual descendant of the vantage point tree and the GNAT [10] structure. The MVP-Tree appears to be superior to the previous methods. The fundamental idea is that, given a point p , one can partition all objects into m partitions based on their distances from p , where the first partition consists of those points within distance d_1 from p , the second consists of those points whose distance is greater than d_1 and less than or equal to d_2 , etc. Given two points, p_a and p_b , the partitions a_1, \dots, a_m based on p_a and the partitions b_1, \dots, b_m based on p_b can be created. One can then intersect all possible a and b -partitions (i.e., a_i intersect b_j for $1 \leq i \leq m$ and $1 \leq j \leq m$) to get m^2 partitions. In an MVP-Tree, each node in the tree corresponds to two objects (vantage points) and m^2 children, where m is a parameter of the construction algorithm and each child corresponds to a partition. When searching for objects within distance t of query point q , the algorithm does the following: Given a parent node having vantage points p_a and p_b , if some partition Z has the property that for every object $z \in Z$, $dist(z, p_a) < d_z$, and $dist(q, p_a) > d_z + t$, then Z can be discarded. There are other reasons for discarding clusters, also based on the triangle inequality. Using multiple vantage points together with precomputed distances reduces the number of distance computations at query time. Like the MVP-Tree, our structure makes aggressive use of the triangle inequality.

Another relevant recent work, due to Cháavez and Navarro [16], proposes a structure called List of Clusters. Such a list is constructed in the following way: Starting from a random point, a cluster with bounded diameter (or limited number of objects) centered in that random point is constructed. Then, such a process is iterated by selecting a new point, for example, the farthest from the previous one, and constructing another cluster around it. The process terminates when no more points are left. Authors experimentally show that their structure outperforms other existing methods when parameters are chosen in a suitable way.

Other sources of inspiration include [11], [23], [26], [30], [45], [44], [48], [40].

3 THE ANTIPOLE TREE

Let $(M, dist)$ be a finite metric space, let S be a subset of M , and suppose that we aim to split it into the minimum possible number of clusters whose radii should not exceed a

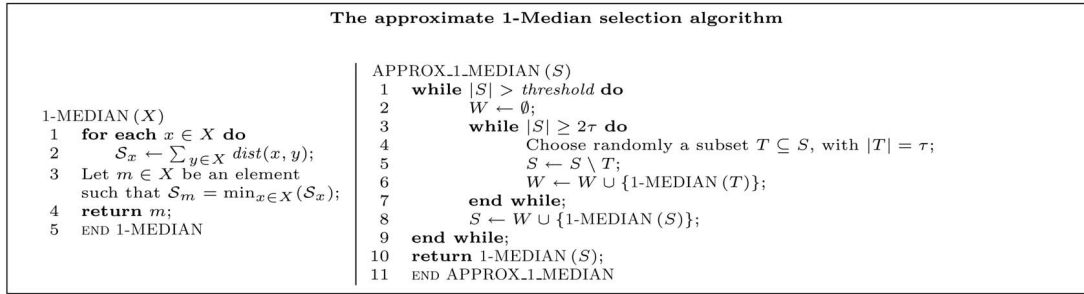


Fig. 1. The 1-Median algorithm.

given threshold σ . This problem has been studied by Hochbaum and Maass [35] for Euclidean spaces. Their approximation algorithm has been improved by Gonzalez in [31]. Similar ideas are used by Feder and Greene [25] (see [43] for an extended survey on clustering methods in Euclidean spaces).

The Antipole clustering of bounded radius σ is performed by a recursive top-down procedure starting from the given finite set of points S and checking at each step if a given splitting condition Φ is satisfied. If this is not the case, then splitting is not performed, the given subset is a cluster, and a centroid having distance approximatively less than σ from every other node in the cluster is computed by the procedure described in Section 3.1.

Otherwise, if Φ is satisfied then a pair of points $\{A, B\}$ of S , called the Antipole pair, is generated by the algorithm described in Section 3.2 and is used to split S into two subsets S_A and S_B obtained by assigning each point p of S to the subset containing the endpoint closest to p of the Antipole $\{A, B\}$. The splitting condition Φ states that $dist(A, B)$ is greater than the cluster diameter threshold corrected by the error coming from the Euclidean case analysis described in the Appendix which can be found on the Computer Society Digital Library at <http://computer.org/tkde/archives.htm>. Indeed, the diameter threshold is based on a statistical analysis of the pairwise distances of the input set (see Section 6.2) which can be used to evaluate the intrinsic dimension [18] of the metric space. The tree obtained by the above procedure is called an Antipole Tree. All nodes are annotated with the Antipole endpoints and the corresponding cluster radius; each leaf contains also the 1-median of the corresponding final cluster. Its implementation is described in Section 3.3.

3.1 1-Median

In this section, we review a randomized algorithm for the approximate 1-median selection [14], an important subroutine in our Antipole Tree construction. It is based on a tournament played among the elements of the input set S . At each round, the elements which passed the preceding turn are randomly partitioned into subsets, say X_1, \dots, X_k . Then, each subset X_i is locally processed by a procedure which computes its exact 1-median x_i . The elements x_1, \dots, x_k move to the next round. The tournament terminates when we are left with a single element \bar{x} , the final winner. The winner approximates the exact 1-median in S . Fig. 1 contains the pseudocode of this algorithm. The local optimization procedure 1-MEDIAN (X) returns the exact 1-median in X . A running time analysis (see [14] for details) shows that the above procedure takes time $\frac{1}{2}n + o(n)$ in the worst-case.

3.2 The Diameter (Antipole) Computation

Let (M, d) be a metric space with distance function $dist : (M \times M) \rightarrow \mathbb{R}$ and let S be a finite subset of M . The *diameter computation problem* or *furthest pair problem* is to find the pair of points A, B in S such that $dist(A, B) \geq dist(x, y), \forall x, y \in S$.

As observed in [36], we can construct a metric space where all distances among objects are set to 1 except for one (randomly chosen) which is set to 2. In this case, any algorithm that tries to give an approximation factor greater than $1/2$ must examine all pairs, so a randomized algorithm will not necessarily find that pair.

Nevertheless, we expect a good outcome in nearly all cases. Here, we introduce a randomized algorithm inspired by the one proposed for the 1-median computation [14] and reviewed in the preceding section. In this case, each subset X_i is locally processed by a procedure LOCAL_WINNER which computes its exact 1-median x_i and then returns the set \bar{X}_i , obtained by removing the element x_i from X_i . The elements in $\bar{X}_1 \cup \bar{X}_2 \dots \cup \bar{X}_k$ are used in the subsequent step. The tournament terminates when we are left with a single set, \bar{X} , from which we extract the final winners A, B , as the furthest points in \bar{X} . The pair A, B is called the *Antipole pair* and their distance represents the approximate diameter of the set S .

The pseudocode of the Antipole algorithm

APPROX_ANTIPOLE,

similar to that of the 1-Median algorithm, is given in Fig. 1.

A faster (but less accurate) variant of

APPROX_ANTIPOLE

can be used. Such variant, called

FAST_APPROX_ANTIPOLE,

consists of taking \bar{X}_i as the farthest pair of X_i . Its pseudocode can therefore be obtained simply by replacing in APPROX_ANTIPOLE each call to LOCAL_WINNER by a call to FIND_ANTIPOLE. In the next section, we will prove that both variants have a linear running time in the number of elements. We will also show that FAST_APPROX_ANTIPOLE is also linear in the tournament size τ , whereas APPROX_ANTIPOLE is quadratic with respect to τ .

For tournaments of size 3, both variants plainly coincide. Thus, since in the rest of the paper only tournaments of size 3 will be considered, by referring to the faster variant we will not lose any accuracy.

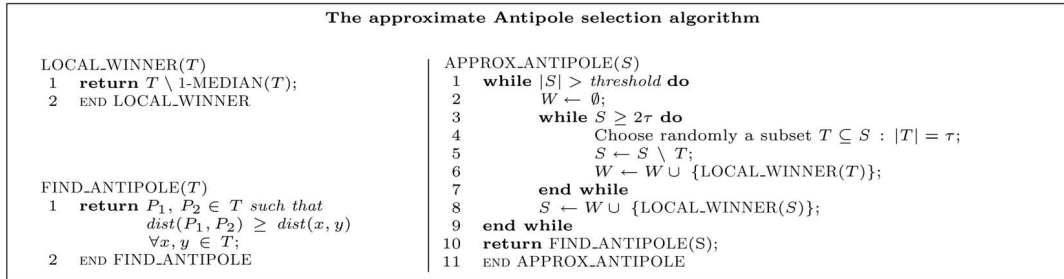


Fig. 2. The Antipole Algorithm.

3.2.1 Running Time Analysis of Antipole Computation

Two fundamental parameters present in the algorithm reported in Fig. 2 (also reported in Fig. 1), namely, the *splitting factor* τ (also referred to as the *tournament size*) and the parameter *threshold*, need to be tuned.

The splitting factor τ is used to set the size of each subset X processed by procedure LOCAL_WINNER, with the only exception of one subset for each round of the tournament (whose size is at most $(2\tau - 1)$), and the argument of the last call to FIND_ANTIPOLE (whose size is at most equal to *threshold*). It is clear that the larger values of τ are, the better the output quality is and the higher the computational costs are. In many cases, a satisfying output quality can be obtained even with small values for τ .

A good trade off between output quality and computational cost is obtained by choosing as value for τ one unit more than the dimension that characterizes the investigated metric space [18]. This suggestion lies on intuitive grounds developed in the case of a Euclidean metric space \mathbb{R}^m and is largely confirmed by the experiments reported in [14]. The parameter *threshold* controls the termination of the tournament. Again, larger values for *threshold* ensure better output quality, though at increasing cost. Observe that the value $(\tau^2 - 1)$ for *threshold* forces the property that the last set of elements, where the final winner is selected, must contain at least τ elements, provided that $|S| \geq \tau$. Moreover, in order to ensure a linear computational complexity of the algorithm, the *threshold* value need to be $\mathcal{O}(\sqrt{|S|})$. Consequently, a good choice is $threshold = \min\{\tau^2 - 1, \sqrt{|S|}\}$.

The algorithm APPROX_ANTIPOLE given in Fig. 2 is characterized by its simplicity and, hence, it is expected to be very efficient from the computational point of view, at least in the case in which the parameters τ and *threshold* are taken small enough. In fact, we will show below that our algorithm has a worst-case complexity of $\frac{\tau(\tau-1)}{2}n + o(n)$ in the input size n , provided that *threshold* is $o(\sqrt{n})$.

Plainly, the complexity of the algorithm APPROX_ANTIPOLE is dominated by the number of distances computed by it within calls to procedure LOCAL_WINNER. We shall estimate below such a number.

Let $W(n, \tau, \vartheta)$ be the number of calls to procedure LOCAL_WINNER made within the **while**-loops by APPROX_ANTIPOLE, with an input of size n and using parameters $\tau \geq 3$ and *threshold* $\vartheta \geq 1$. Plainly, $W(n, \tau, \vartheta) \leq W(n, \tau, 1)$, for any $\vartheta \geq 1$, thus it will suffice to find an upper bound for $W(n, \tau, 1)$. For notational convenience, let us put $W_1(n) = W(n, \tau, 1)$, where τ has

been fixed. It can easily be seen that $W_1(n)$ satisfies the following recurrence relation:

$$W_1(n) = \begin{cases} 0 & \text{if } 0 \leq n \leq 2, \\ 1 & \text{if } 3 \leq n < 2\tau, \\ \lfloor \frac{n}{\tau} \rfloor + W_1((\tau - 1) \cdot \lfloor \frac{n}{\tau} \rfloor) & \text{if } n \geq 2\tau. \end{cases}$$

By induction on n , we can show that $W_1(n) \leq n$. For $n < 2\tau$, our estimate is trivially true. Thus, let $n \geq 2\tau$. Then, by inductive hypothesis, we have

$$\begin{aligned} W_1(n) &= \lfloor \frac{n}{\tau} \rfloor + W_1\left((\tau - 1) \cdot \lfloor \frac{n}{\tau} \rfloor\right) \leq \lfloor \frac{n}{\tau} \rfloor + (\tau - 1) \cdot \lfloor \frac{n}{\tau} \rfloor \\ &= \lfloor \frac{n}{\tau} \rfloor \cdot (1 + (\tau - 1)) = n. \end{aligned}$$

The number of distance computations made by a call LOCAL_WINNER(X) is equal to $\sum_{i=1}^{|X|} (i - 1) = \frac{|X|(|X|-1)}{2}$. At each round of the tournament, all the calls to procedure LOCAL_WINNER have an argument of size τ , with the possible exception of the last call, which can have an argument of size between $(\tau + 1)$ and $(2\tau - 1)$. We notice that the last call to procedure FIND_ANTIPOLE made within the **return** instruction of APPROX_ANTIPOLE has argument of size at most ϑ . Since there are $\lceil \log_{\tau/(\tau-1)} n \rceil$ rounds, it follows that the total number of distances computed by a call of APPROX_ANTIPOLE(S), with $|S| = n$, tournament size τ , and *threshold* ϑ , is majorized by the expression

$$\begin{aligned} W(n, \tau, \vartheta) \cdot \frac{\tau(\tau - 1)}{2} + \lceil \log_{\tau/(\tau-1)} n \rceil \\ \cdot \left[\frac{(2\tau - 1)(2\tau - 2)}{2} - \frac{\tau(\tau - 1)}{2} \right] + \frac{\vartheta(\vartheta - 1)}{2} \\ = \frac{\tau(\tau - 1)}{2}n + \mathcal{O}(\log n + \vartheta^2). \end{aligned}$$

By taking $\vartheta = o(\sqrt{n})$, the above expression is easily seen to be $\frac{\tau(\tau-1)}{2}n + o(n)$.

Summing up, we have:

Theorem 3.1. *Given an input set of size $n \in \mathbb{N}$, a constant tournament size $\tau \geq 3$, and a threshold $\vartheta = o(\sqrt{n})$, the algorithm APPROX_ANTIPOLE performs $\frac{\tau(\tau-1)}{2}n + o(n)$ distance computations.*

Concerning the complexity of the faster variant FAST_APPROX_ANTIPOLE, we have the following recurrence relation $W_1(n) = \lfloor \frac{n}{\tau} \rfloor + W_1(2 \cdot \lfloor \frac{n}{\tau} \rfloor)$, for $n \geq 2\tau$. By

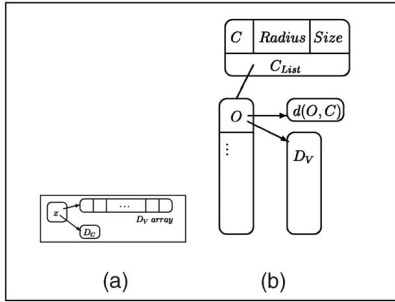


Fig. 3. (a) A generic *object* in the Antipole data structure. (b) A generic cluster in the Antipole data structure.

induction on n , we can show that the number of calls to the subroutine `FIND_ANTIPOLE` is $W_1(n) \leq \lceil \frac{n}{\tau-2} \rceil$. For $n < 2\tau$, our estimate is trivially true. Thus, let $n \geq 2\tau$. Then, by inductive hypothesis, we have

$$\begin{aligned} W_1(n) &= \left\lfloor \frac{n}{\tau} \right\rfloor + W_1\left(2 \cdot \left\lfloor \frac{n}{\tau} \right\rfloor\right) \leq \left\lfloor \frac{n}{\tau} \right\rfloor + \left\lceil \frac{2 \cdot \lfloor \frac{n}{\tau} \rfloor}{\tau-2} \right\rceil \\ &\leq \left\lfloor \frac{n}{\tau} \right\rfloor \cdot \left\lceil 1 + \frac{2}{\tau-2} \right\rceil \leq \left\lceil \frac{n}{\tau-2} \right\rceil. \end{aligned}$$

Finally, much by the same arguments as those preceding Theorem 3.1, we can show that the following holds:

Theorem 3.2. *Given an input set of size $n \in \mathbb{N}$, a constant tournament size $\tau \geq 3$, and a threshold $\vartheta = o(\sqrt{n})$, the algorithm `FAST_APPROX_ANTIPOLE` performs $\frac{\tau(\tau-1)}{2(\tau-2)}n + o(n)$ distance computations.*

3.3 The Antipole Tree Data Structure in General Metric Spaces

The Antipole Tree data structure can be used in a generic metric space $(M, dist)$, where $dist$ is the distance metric. Each element of the metric space along with its related data constitutes a type called *object*. An *object* O (Fig. 3a) in the Antipole data structure contains the following information: an element x , an array D_V storing the distances between x and all its ancestors (the Antipole pairs) in the tree, and a variable D_C containing the distance from the centroid C of x 's cluster. A data set S is a collection of *objects* drawn from M . Each cluster (Fig. 3b) stores the following information:

- *centroid*, C , the element that minimizes the sum of the distances from the other cluster members;
- *radius*, $Radius$, containing the distance from C to the farthest *object*;
- *member list*, C_{List} , storing the catalog of the *objects* contained in the cluster;
- *size* of C_{List} , $Size$, stored in the cluster.

The Antipole data structure has internal nodes and leaf nodes:

- An internal node stores 1) the identities of two Antipole objects A and B , called the Antipole pair of distance at least 2σ apart, 2) the radii Rad_A and Rad_B of the two subsets (S_A, S_B) obtained by splitting S based on their proximity to A and B , respectively), and 3) pointers to the left and right subtrees *left* and *right*.
- A leaf node stores a cluster.

To build such a data structure, the procedure `BUILD` (see Fig. 4) takes as input the data set S , a target cluster radius σ , and a set Q (empty at the beginning). The algorithm starts by checking if Q is empty and, if so, it calls the subroutine `ADAPTED_APPROX_ANTIPOLE`² which returns an Antipole pair. Then, the Antipole pair is inserted into Q .

Next, the algorithm checks if the splitting condition is true. If this is the case, the set S is divided into S_A and S_B , where the objects closer to A are put in S_A and symmetrically for B . Otherwise, a cluster is generated. The other subroutine used in `BUILD` is `CHECK` which checks whether there is an object O in S_A (or S_B) that may become the Antipole of A (or B), by using the distances already computed and cached. If an Antipole is found, it is inserted into Q and then the recursive call in `BUILD` skips the computation of another Antipole pair.

The routine `MAKE_CLUSTER` (Fig. 4) creates a cluster of objects with bounded radius. This procedure computes the cluster centroid C with the randomized algorithm `APPROX_1_MEDIAN` and then computes the distance between each O in the cluster and C .

The data structure resulting from `BUILD` is a binary tree whose leaves contain a set of clusters, each of which has an approximate centroid and the radius, based on that centroid, is less than σ . Fig. 5a shows the evolution of the data set during the construction of the tree. At the first step, the pair A, B is found by the algorithm `ADAPTED_APPROX_ANTIPOLE`, then the input data set is split into the subsets S_A and S_B . The second step proceeds as the first for the subset containing A while, for the subset containing B , it produces a cluster since its diameter is less than 2σ . The third and final step produce the final clusters for the subsets containing A_1 and B_1 . Fig. 5b shows the corresponding Antipole data structure.

3.3.1 Construction Time Analysis

Let us compute the running time of each routine. Building the Antipole Tree takes quadratic time in the worst case. For example, let us consider a metric space in which the distance between any pair of distinct objects is $2\sigma + 1$. In this case, if the subsets S_A and S_B have size 1 and $|S| - i$ respectively, where i is the i th recursive call, then the complexity becomes $O(n^2)$. Notice that `ADAPTED_APPROX_ANTIPOLE` will take constant computational time in this case because all the pairwise distances are supposed to be strictly greater than 2σ .

4 RANGE SEARCH ALGORITHM

The range search algorithm takes as input the Antipole Tree T , the query object q , the threshold t , and returns the result of the range search of the database with threshold t . The search algorithm recursively descends all branches of the tree until either it reaches a leaf representing a cluster to be visited or it detects a subtree that is certainly out of range and, therefore, may be pruned out. Such branches are filtered by applying the triangle inequality. Notice that the triangle inequality is used both for exclusion and inclusion. The use for exclusion establishes that an object can be pruned, thus avoiding the computation of the distance between such an object and the query. The other usage establishes that an object must be inserted because the

2. Notice that this algorithm is a variation of `FIND_ANTIPOLE` that stops when a pair of objects with distance greater than 2σ is found, otherwise, it returns an empty set.

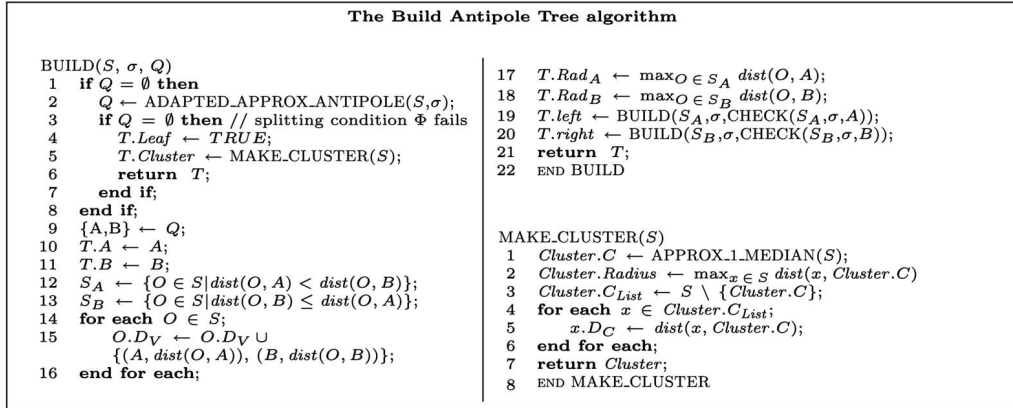


Fig. 4. The algorithm Build Antipole Tree and the routine MakeCluster.

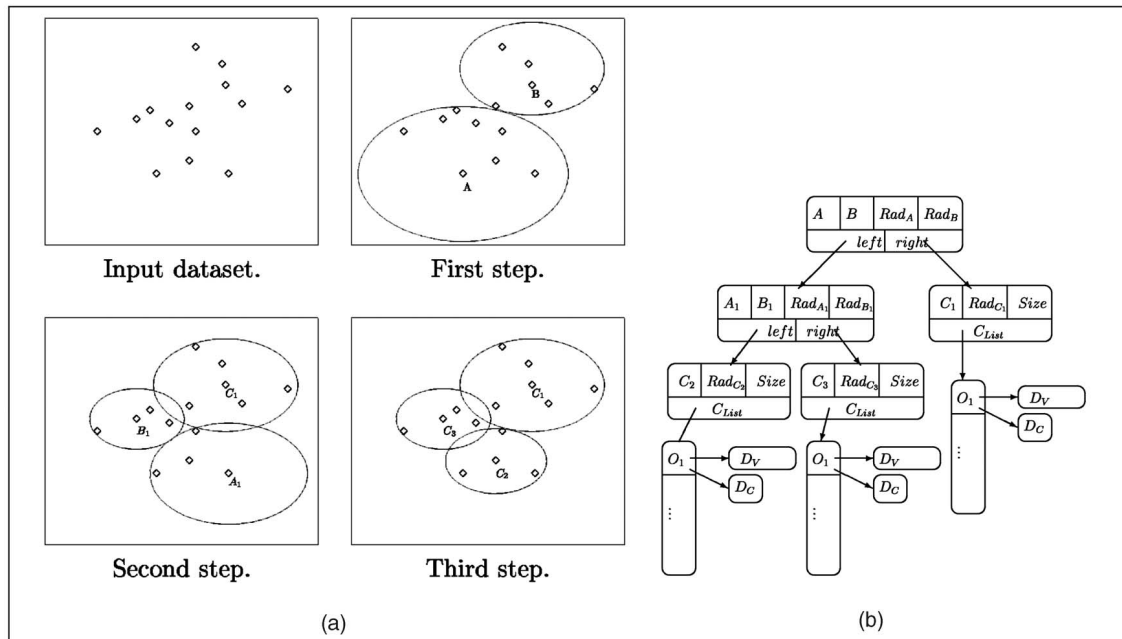


Fig. 5. A clustering example (a) in a generic metric space and (b) the corresponding Antipole data structure.

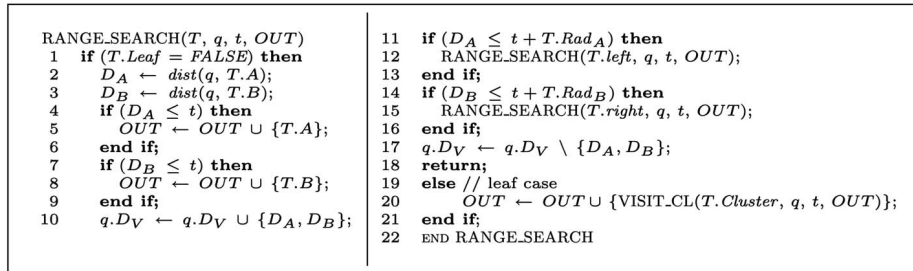


Fig. 6. The Range Search algorithm.

object is close to its cluster's centroid and the centroid is very close to the query object (see Figs. 6 and 7 for the pseudocode).

5 K-NEAREST NEIGHBOR ALGORITHM

The k -nearest neighbor search algorithm takes as input the Antipole Tree T , the query object q , and the k parameter

indicating the number of objects requested. It returns the set of objects in S which are the k -nearest neighbors of q . Hjaltason and Samet in [34] propose a method called Incremental Nearest Neighbor to perform k -nearest neighbor search in spatial databases. Their approach uses a priority queue storing the subtrees that should be visited, ordered by their distance from the query object. The authors claim that their approach can be applied to all hierarchical

```

VISIT_CL(Cluster, q, t, OUT)
1   $q.D_C \leftarrow \text{dist}(q, \text{Cluster}.C)$ ;
2  if ( $q.D_C \leq t$ ) then
3     $OUT \leftarrow OUT \cup \{\text{Cluster}.C\}$ ;
4  end if;
5  if ( $q.D_C \geq t + \text{Cluster}.Radius$ ) then
6    return;
7  end if;
8  if ( $q.D_C \leq t - \text{Cluster}.Radius$ ) then
9     $OUT \leftarrow OUT \cup \text{Cluster}$ ;
10   return OUT;
11 end if;
12 for each  $O \in \text{Cluster}.C_{List}$  do
13   if ( $q.D_C \geq t + O.D_C$ ) then
14     continue;
15   end if;
16   if ( $q.D_C \leq t - O.D_C$ ) then
17      $OUT \leftarrow OUT \cup \{O\}$ ;
18     continue;
19   end if;
20   if ( $\exists (d_q \in q.D_V \text{ and } d_O \in O.D_V) \mid$ 
21      $d_q \geq t + d_O \text{ or } d_q \leq t - d_O$ ) then
22     if ( $\text{dist}(q, O) \leq t$ ) then
23        $OUT \leftarrow OUT \cup \{O\}$ ;
24     end if;
25   else
26     if ( $d_q \leq t - d_O$ ) then
27        $OUT \leftarrow OUT \cup \{O\}$ ;
28     end if;
29   end for each;
30   return OUT;
31 END VISIT_CL

```

Fig. 7. The Visit Cluster algorithm.

data structures. Here, we propose an application of such a method to Antipole Tree.

The algorithm described below uses two different priority queues. The first one stores the subtrees of the Antipole data structure which may be visited during the search (left subtree, right subtree, or leaf); the second one keeps track of the objects that will be returned as output.

The incremental nearest-neighbor algorithm starts by putting the root of the Antipole Tree in the priority queue $pQueue$. Then, it proceeds by extracting the minimum from the priority queue. If the extracted node is a leaf (cluster), it visits it. Otherwise, it decides to visit each of its subtrees on the basis of the subtree's radius, the distance of the Antipole endpoint from the query, and a threshold t by applying the triangle inequality. The threshold t , which is initialized to ∞ , stores the largest distance from the query q to any of the current k -nearest neighbors. Subtrees which need to be visited will be put in the priority queue. All current k -nearest neighbors found are stored in another heap $outQueue$ in order to optimize the dynamic operations (such as insertions, deletions, and updates). Figs. 8 and 9 summarize the pseudocode.

6 EXPERIMENTAL ANALYSIS

In this section, we evaluate the efficiency of constructing and searching through an Antipole Tree. We have implemented the structure using the C programming language under Linux operating system. The experiments use synthetic and real data sets. The synthetic data sets are based on those ones used by [9]:

- uniform 10-dimensional Euclidean space (sets of 100,000, 200,000, ..., 500,000 objects uniformly distributed in $[0, 1]^{10}$);

- clustered 20-dimensional Euclidean space. More precisely, a set of 100,000 objects obtained in the following way: By using uniform distributions, take 100 random spheres and select 1,000 random points in each of them.

The real data sets are, respectively:

- a set of 45,000 strings chosen from the Linux dictionary with the editing distance;
- a set of 42,000 images chosen from the Corel image database with the metric L_2 ;
- high-dimensional Euclidean space sets of points corresponding to textures of VISTEX database [50] with the metric L_2 .

For each experiment, we ran 100 random queries: half of them were chosen in the input set, the remaining ones in the complement.

6.1 Construction Time

We measure construction time in terms of the number of distance computations and CPU time on uniformly distributed objects in $[0, 1]^{10}$, as described above. Fig. 10a illustrates a comparison between the Antipole Tree, the MVP-Tree, and the M-Tree, showing the distances needed during the construction. Data were taken again in $[0, 1]^{10}$ with size from 100,000 to 500,000 elements. The cluster radius σ used was $\sigma = 0.625$, as found by our estimation algorithm described below. We used the parameter settings for MVP-Trees and M-Trees suggested by the authors [9], [20]. Fig. 10a shows also that building the Antipole Tree requires fewer distance computations than the M-Tree but more than the MVP-Tree. The difference is roughly a factor of 1.5. Fig. 11 shows that the difference in construction costs can be compensated by faster range queries on less than

```

K_NEAREST_NEIGHBOR(T, q, t, outQueue, k, pQueue)
1  Enqueue(pQueue, Tree, NULL);
2  while NotEmpty(pQueue) do
3    node = Dequeue(pQueue);
4    if (node.leaf = TRUE) then
5      KNN_VISIT_CLUSTER(node, q, t, outQueue, k);
6    else
7       $D_A \leftarrow \text{KNN\_CHECK}(q, \text{node}.A, t, \text{outQueue})$ ;
8       $D_B \leftarrow \text{KNN\_CHECK}(q, \text{node}.B, t, \text{outQueue})$ ;
9      Enqueue(pQueue, node.left,  $D_A - \text{node}.Rad_A$ );
10     Enqueue(pQueue, node.right,  $D_B - \text{node}.Rad_B$ );
11   end if;
12 end while;
13 END K_NEAREST_NEIGHBOR

```

Fig. 8. The incremental k -nearest neighbor search algorithm.

```

KNN_CHECK(q, O, t, OUT)
1   $D_O \leftarrow \text{dist}(q, O)$ ;
2  if ( $|OUT| < k$ ) then
3    HEAP_INSERT( $O, OUT$ );
4     $t \leftarrow \text{HEAP\_EXTRACT\_MAX}(OUT)$ ;
5  else
6    if ( $D_O < t$ ) then
7      HEAP_INSERT( $O, OUT$ );
8       $t \leftarrow \text{HEAP\_EXTRACT\_MAX}(OUT)$ ;
9    end if;
10 end if;
11 return  $D_O$ ;
13 END KNN_CHECK

```

Fig. 9. A procedure for checking whether the object O should be added to the OUT set.

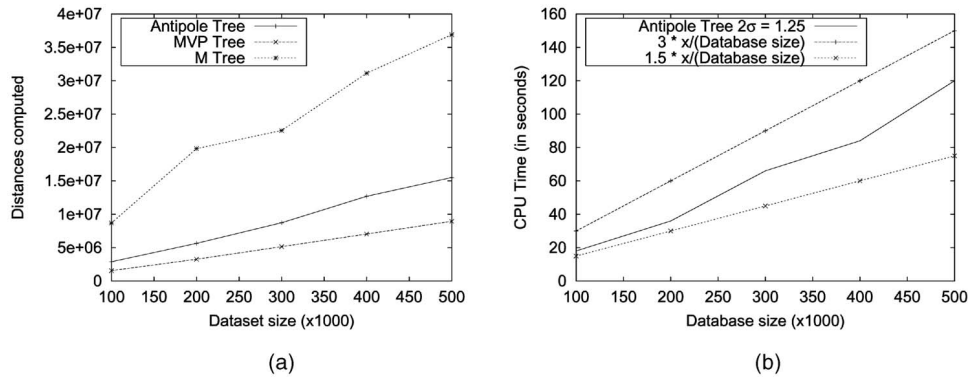


Fig. 10. (a) Construction complexity using uniformly generated data measured by the number of distance computations needed by the Antipole Tree with cluster diameter 1.25 versus M-Tree and MVP-Tree. (b) CPU time in seconds needed to build the Antipole Tree.

0.2 percent of the entire input database. Thus, unless queries are very rare, the Antipole Tree recovers in terms of queries cost what it loses in construction. Experiments proving this fact are reported in Section 6.3.

Fig. 10b shows the CPU time needed to bulk load the proposed data structure; it also shows that the CPU time needed to construct the Antipole Tree grows linearly in many cases. Because the MVP-Tree entails sorting, it requires at least $O(n \log n)$ operations (though not distance calculations) to build the data structure.

6.2 Choosing the Best Cluster Diameter

In this section, we discuss how to tune the Antipole Tree for range queries. We measure the cost by the number of distance calculations among objects of the underlying metric space.

Before the Antipole data structure can be used, it needs to be tuned. To tune the Antipole Tree, we must choose the radius σ of the clusters very carefully by analyzing the data set properties. In what follows, we will show that optimal cluster radius depends on the intrinsic dimensionality of the underlying metric space.

We performed, as described before, our experiments in 10 and 20-dimensional spaces with uniform and clustered distributions having size 100,000. However, the methodology of finding the optimal diameter can be applied to other dimensions and arbitrary data sizes.

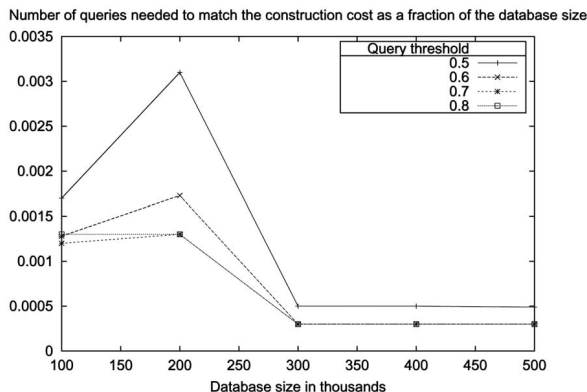


Fig. 11. Number of range queries, as a fraction of the data set size, which are sufficient to recover the higher cost of Antipole Tree construction with respect to MVP-Tree construction.

Figs. 12 (Uniform) and (Clustered) show that across different values of the threshold t of the range search, the best choice of the cluster diameter is 0.625 for the uniform data set and 2.5 for the clustered one.

Experiments with real and synthetic data showed that choosing the cluster diameter 10 percent less than the median pairwise distance value gives, regardless of the range search threshold, a quite surprising result.

6.3 Range Search Analysis and Comparisons

In this section, we present an extensive comparison among the Antipole Tree, the MVP-Tree, the M-Tree, and List of Clusters in terms of the number of distance computations for range queries. The number of distance computations required by each query has been estimated as the average value in a set of 100 queries. In order to perform a fair comparison with the three competing data structures, MVP-Tree, M-Tree, and List of Cluster, we have set their implementation parameters to the best values according to the ones suggested by the authors. For the MVP-Tree, in [9] it is shown that its best performance is achieved by setting the parameters in the following way:

1. Two vantage points in every internal node v_1 and v_2 .
2. $m^2 = 4$ partition classes. Four children for each pair of vantage points.
3. $k = 13$ maximum number of objects in a leaf node.
4. p unbounded, the size of the vector storing the distances between the objects in a leaf and their ancestors in the tree (the vantage points). Such a vector is used during the range search to discard objects without having to compute their distance from the query object. Notice that the higher the dimension is of such a vector the more distances from vantage points can be used to prune candidates and this improves the performance of the MVP-Tree in terms of distance computations. For this reason, we have set this parameter to its maximum value: the height of the MVP-Tree.³

For the M-Tree implementation, we made use of the *BulkLoading*⁴ algorithm [20]. The two parameters needed to tune the data structure in order to obtain better performance are the minimum node utilization and the secondary memory page size. The best performance observed during

3. The authors are grateful to T. Bozkaya and M. Ozsoyoglu for providing them the program to generate the input for the clustered data set.

4. The authors are grateful to P. Ciaccia, M. Patella, and P. Zezula for providing them the source code of the M-Tree.

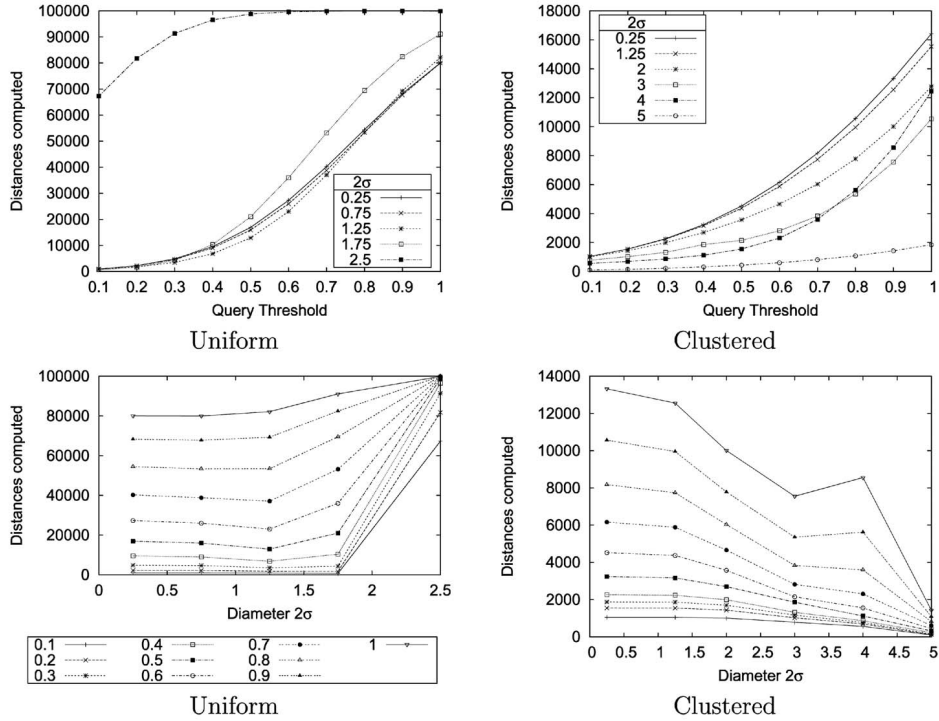


Fig. 12. Diameter tuning using uniformly and clustered generated points in dimensions 10 and 20, respectively.

the search was obtained with minimum node utilization 0.2 and page size 8K.

Concerning List of Clusters, we used fixed bucket size according with heuristics p_3 and p_5 suggested by the authors in [16]. p_3 consists in choosing the center of the i th cluster as the furthest element from the $(i - 1)$ th center, whereas p_5 picks the element which maximizes the sum of distances from previous centers.

In the first experiment (Fig. 13), we compare the four data structures in a uniform data set taken from $[0, 1]^n$ with $n = 10$, varying the query threshold from 0.1 to 0.8, and using a data set of size 300,000. For the Antipole, we used two different cluster radii σ : 0.5 and 0.625, respectively. Antipole Tree performs better than the other three data structures computing less distances during the search.

Notice that using a query threshold from 0.1 to 0.7, we capture in the output data set from 0% to 1% of the elements of the entire data set (0.8 captures the 3% of the entire set). Fig. 14 shows that with query thresholds from 0.4 to 0.6, we save between 10 percent and 70 percent of the distance computations, which, in the figure, is indicated as the gain percentage.

The next set of experiments (see Fig. 15) was designed to compare the four data structures in different metric spaces: the clustered Euclidean space \mathbb{R}^{20} , a string space under an editing distance metric, and an image histogram space with an L_2 distance metric. The corresponding data sets are: 100,000 clustered points, 45,000 strings from the Linux dictionary, and 42,000 image histograms from the Corel image database,⁵ respectively. Results show a 30 percent of savings in distance computations.

Since List of Clusters reportedly works well in high dimension in Fig. 16, we show a comparison in range search in very high dimension Euclidean spaces \mathbb{R}^{147} and \mathbb{R}^{267} ,

with a database size 3,000 obtained from the VISTEX [50] texture database. Notice that by using the query thresholds depicted in Fig. 16, the output set captures from 0 percent to 5 percent of the elements of the entire data set in \mathbb{R}^{147} and from 0 percent to 10 percent of the elements of the entire data set in \mathbb{R}^{267} . Antipole Tree shows a better behavior with regard to List of Clusters tuned with the best fixed bucket size we noticed.

6.4 K-Nearest Neighbor Comparisons

In Fig. 17, we present a set of experiments in which the K_NEAREST_NEIGHBOR algorithm is compared with the M-Tree and the List of Clusters. Notice that we compared the Antipole Tree with just the M-Tree and List of Clusters because the k -nearest neighbor search is not discussed for the MVP-Tree (see [9]). As described in Section 6.3, we choose uniform and clustered data in \mathbb{R}^{10} and \mathbb{R}^{20} . Each data set has size 100,000. We run the K_NEAREST_NEIGHBOR algorithm with $k = 1, 2, 4, 6, 8, 10, 15, 20$ using 100 queries for

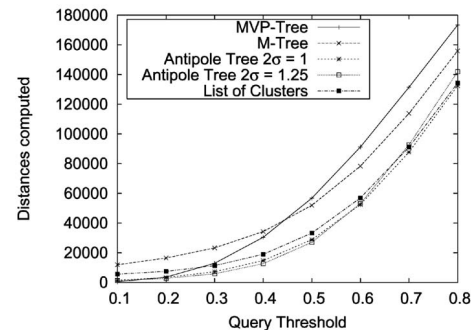


Fig. 13. Comparisons in \mathbb{R}^{10} using 300,000 randomly generated vectors. The query threshold goes from 0.1 to 0.8.

5. Obtained from the UCI Knowledge Discovery in Databases Archive, <http://kdd.ics.uci.edu>.

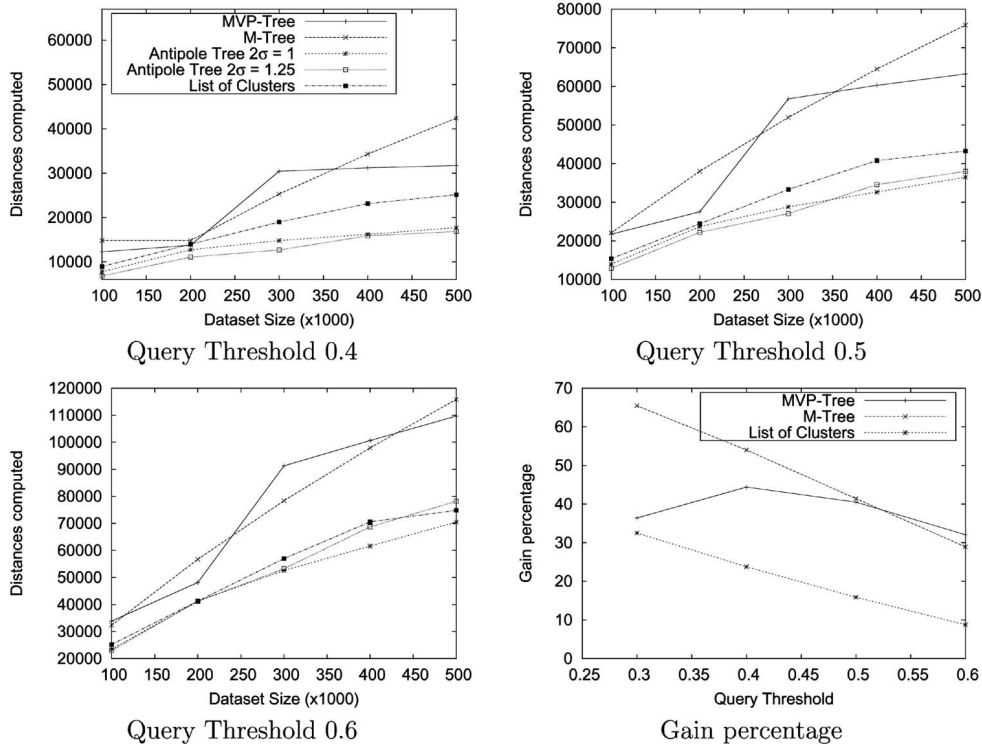


Fig. 14. Each picture shows the number of distances computed by the compared data structures using threshold from 0.4 to 0.6. The respective gain percentage (percentage of distances saved) of the Antipole Tree with regard to the MVP-Tree, the M-Tree, and the List of Clusters is also plotted.

each experiment (half belonging to the data structure and half not). Using the Antipole Tree, we save up to 85 percent of distance computations.

Concerning experiments in very high dimension, in Fig. 18 we show a comparison with List of Clusters using a data set of 3,000 elements in Euclidean \mathbb{R}^{147} and \mathbb{R}^{267} from VISTEX [50]. Antipole Tree clearly outperforms List of Clusters.

7 APPROXIMATE K-NEAREST NEIGHBOR SEARCH VIA ANTIPOLE TREE

When the dimension of the space becomes very high (say ≥ 50), all existing data structures perform poorly on range and k -nearest neighbor searches. This is due to the well-known problem of the *curse of dimensionality* [37]. Lower bounds [19] show that the search complexity exponentially grows with the space dimension. For generic metric spaces, following [17] and [18], we introduce the concept of *intrinsic dimensionality*:

Definition 7.1. Let $(M, dist)$ be a metric space, and let $S \subseteq M$. The intrinsic dimension of S is $\rho = \frac{\mu_S}{2\sigma_S^2}$, where μ_S and σ_S^2 are the mean and the variance of its histogram distances.

A promising approach to alleviate at least the curse of dimensionality is to consider approximate and probabilistic algorithms for k -nearest neighbor search. In some applications, such algorithms give acceptable results. Several interesting algorithms have been proposed in the literature [17], [21], [39], [28]. One of the most successful data structure seems to be the Tree Structure Vector Quantization (TSVQ). Here, we will show how to use the Antipole Tree to design a suitable approximate search algorithm for

the nearest neighbor search. A first simple algorithm, called BEST_PATH_SEARCH, follows the best path in the tree from the root to the leaf, and returns the centroid stored in the leaf node. This algorithm uses the same strategy as the TSVQ to find quickly an approximate nearest neighbor of a query object.

In what follows, we present a set of experiments where TSVQ and Antipole Tree are compared. The experiments refer to uniformly generated objects in spaces whose dimension ranges from 10 to 50. For each input data set, 100 queries were executed. In order to evaluate the quality of the results, we run the exact search first. Then, the error δ is computed in the following way:

$$\delta = \frac{|dist(O_{opt}, q) - dist(O_{TSVQ/Antipole}, q)|}{dist(O_{opt}, q)}$$

In Fig. 19a, the errors introduced by the two approximate algorithms in uniformly generated set of points (upper figures) and clustered set of points (lower figures) are depicted. On the other hand, Figs. 19b and 19d show the number of distances computed by the two algorithms.

The experiments clearly show that the Antipole Tree improves on TSVQ. We think that this is due to the better position of the Antipole pairs.

A more sophisticated approximation algorithm to solve the k -nearest neighbor problem can be obtained by using the K_NEAREST_NEIGHBOR algorithm. The idea is the following: For each cluster reached during the search, the algorithm compares the query object with the cluster centroid without taking into consideration the objects inside it.

This search is slower than the BEST_PATH_SEARCH, but is more precise and can be used to perform k -nearest neighbor search. Fig. 20a shows a set of experiments done in uniform spaces in dimension 30 with radius σ set to 1 and 1.5.

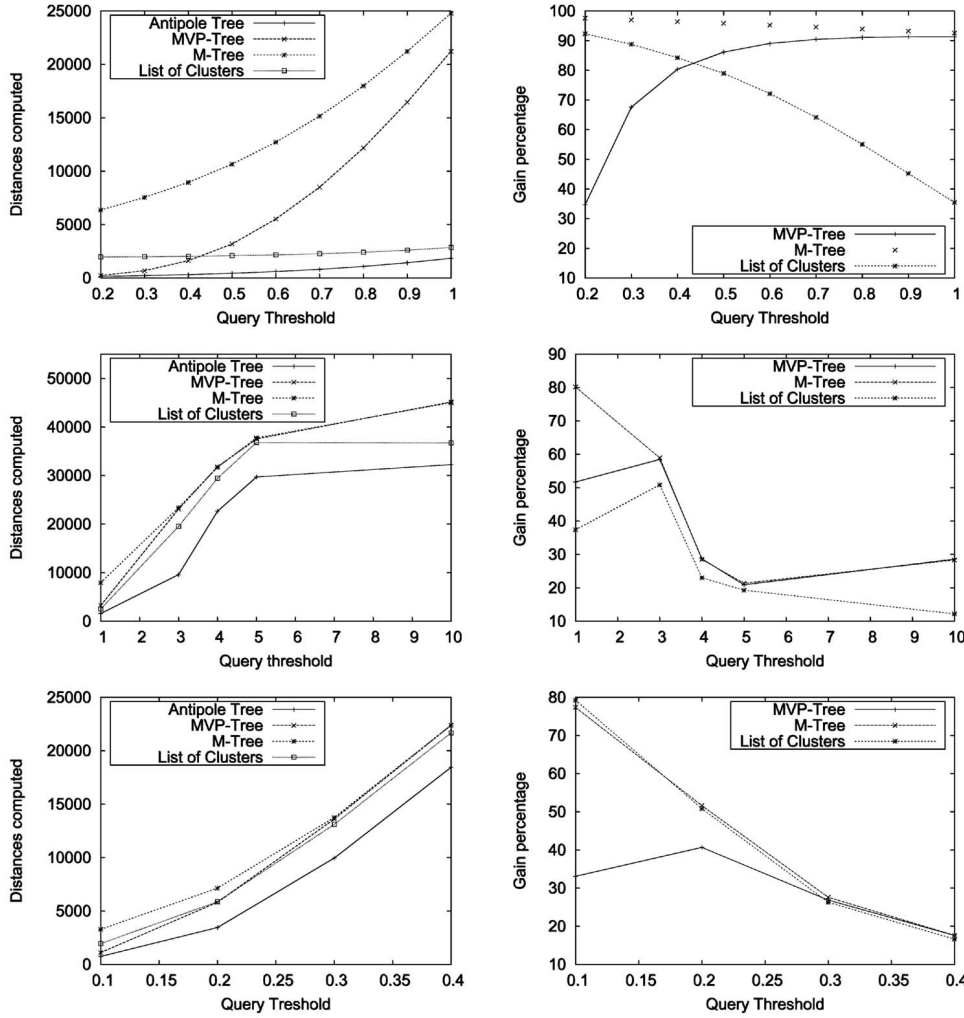


Fig. 15. (top) Comparisons of Antipole Tree versus MVP-Tree, M-Tree, and List of Clusters in a clustered space from \mathbb{R}^{20} varying the query threshold from 0.1 to 1, with cluster radius 2. (middle) Antipole Tree versus MVP-Tree, M-Tree, and List of Clusters using an editing distance metric with cluster radius 5. (bottom) Antipole Tree versus MVP-Tree, M-Tree, and List of Clusters using a set of image histograms with cluster radius 0.4.

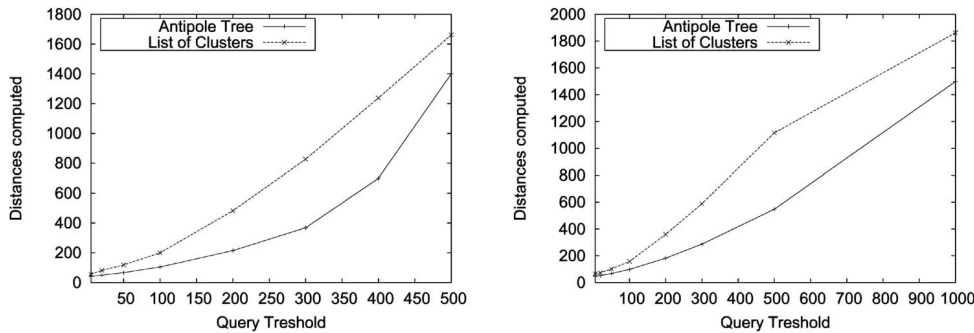


Fig. 16. A comparison between Antipole Tree and List of Clusters using real database in \mathbb{R}^{147} (left) and \mathbb{R}^{267} (right).

In approximate matching, precision and recall [38] are important metrics. Following [38], we call the k -nearest neighbor elements of a query q : the k golden results. Then, the *recall* after quota distances can be defined as the fraction of the k top golden elements retrieved fixing a bound, called quota, in the number of distances that can be computed during the search. The *precision* is the number of golden elements retrieved over the number of distances computed. On the other hand, if the recall R is fixed (i.e. 50 percent),

the *R-precision* (precision after R recalls) gives the number of distances which must be computed to obtain such recall. We performed precision-recall analysis between Antipole Tree and the approximate version of List of Clusters [12]. Experiments in Fig. 22 made use of 100,000 elements of dimension 30. We fixed several quotas and recalls ranging from 7,000 to 42,000 and from 0.5 to 0.9, respectively. Results clearly show that Antipole Tree gives precision-recall factors better than List of Clusters (with fixed bucket

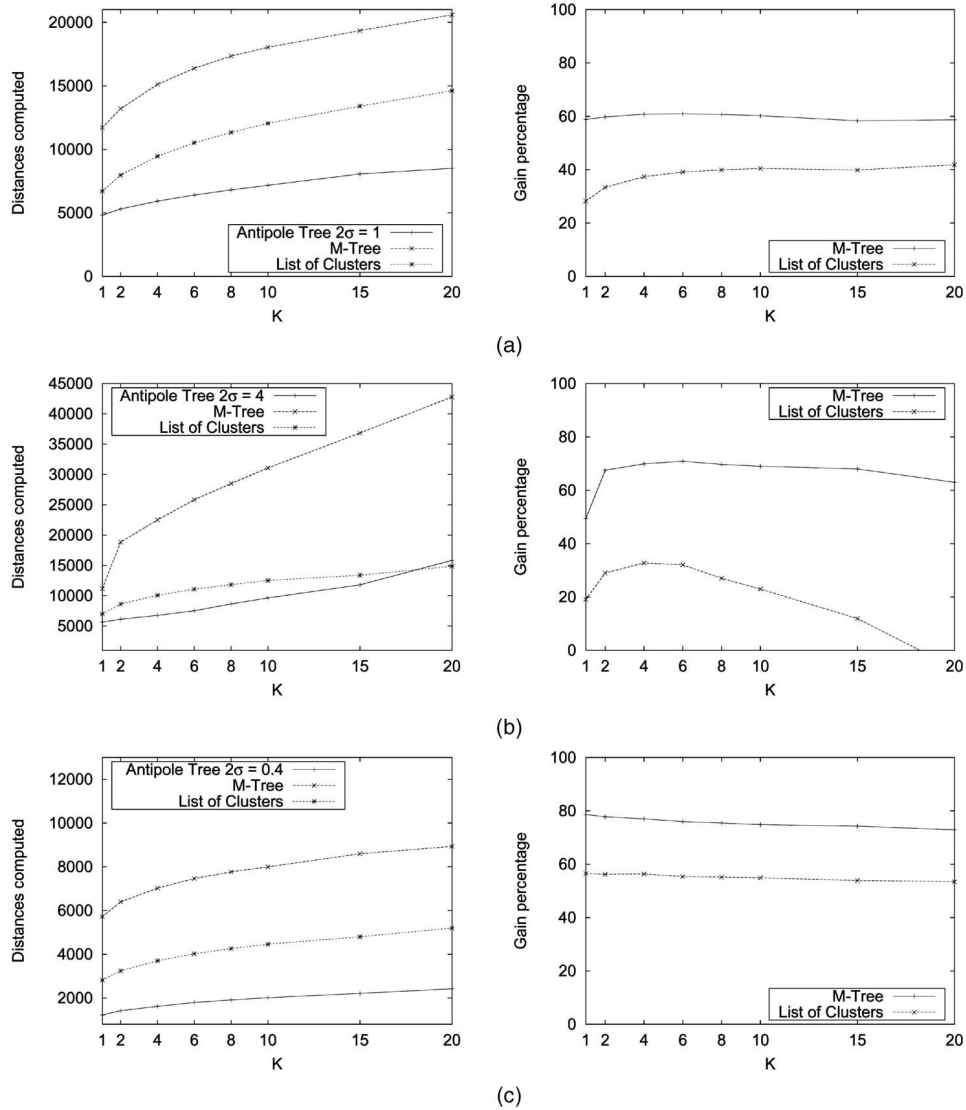


Fig. 17. k -nearest neighbor comparisons. (a) 100,000 uniformly generated points in $[0,1]^{10}$. (b) 100,000 points from \mathbb{R}^{20} generated in clusters. (c) Comparisons using the image histogram database.

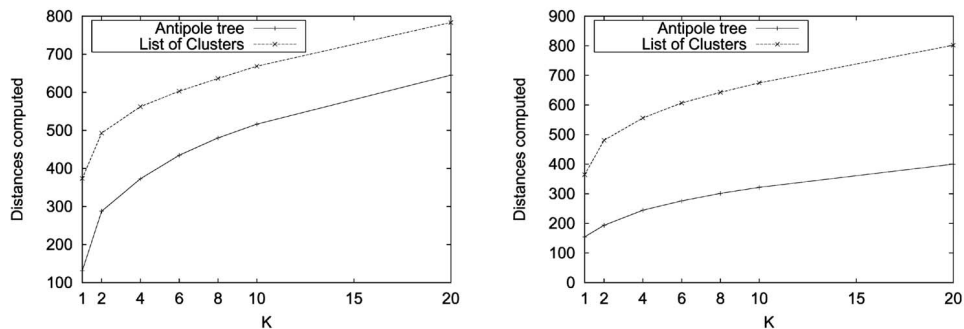


Fig. 18. k -nearest neighbor search using real data from the VISTEX database in dimension \mathbb{R}^{147} and \mathbb{R}^{267} .

size). Fig. 21a makes the same comparison but using Image histogram database, also Fig. 21b illustrates the effect of curse of dimensionality in precision-recall factor analysis for the Antipole Tree using uniformly distributed objects in Euclidean spaces of dimension ranging from 30 to 50.

8 A COMPARISON WITH LINEAR SCAN

In this section, we present a set of experiments in which we compare the proposed data structure with a naive linear scan. We used a set of very high-dimensional Euclidean data sets. Such data sets were obtained from a set of textures taken from the VISTEX database [50]. Starting from

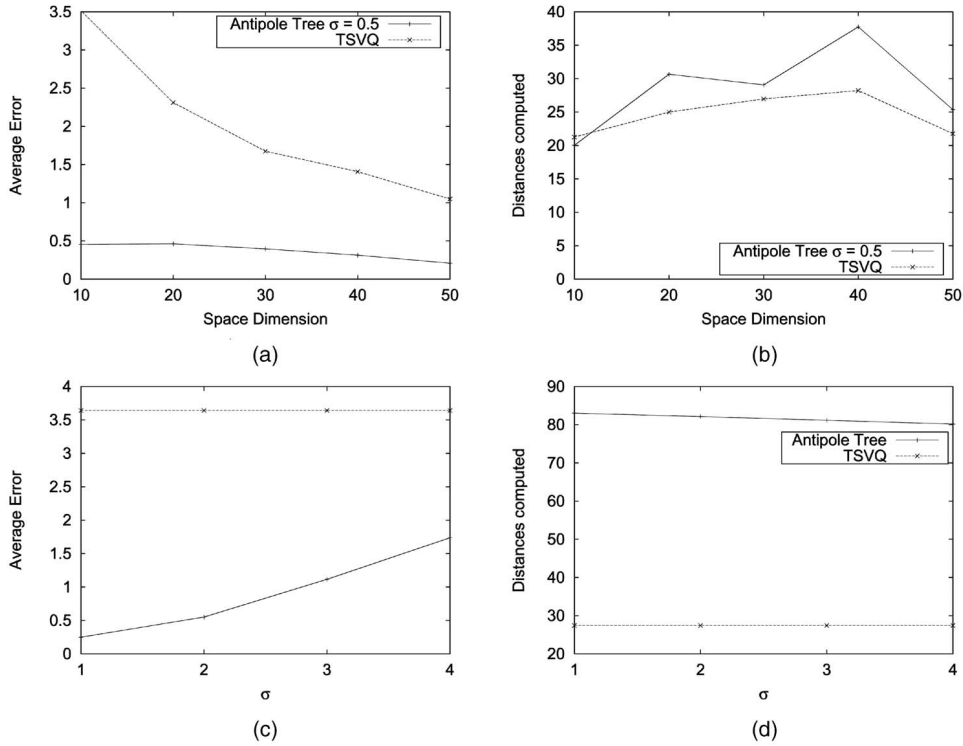


Fig. 19. A comparison between the approximate Antipole search and TSVQ search. (a) Shows the average error introduced by the two algorithms in uniformly generated points with $\sigma = 0.5$ varying the space dimension from 10 to 50. (b) Shows the number of distances computed. (c) Shows the average error introduced using points generated in clusters of space dimension 20 varying the cluster radius σ . (d) Shows the corresponding number of distances needed.

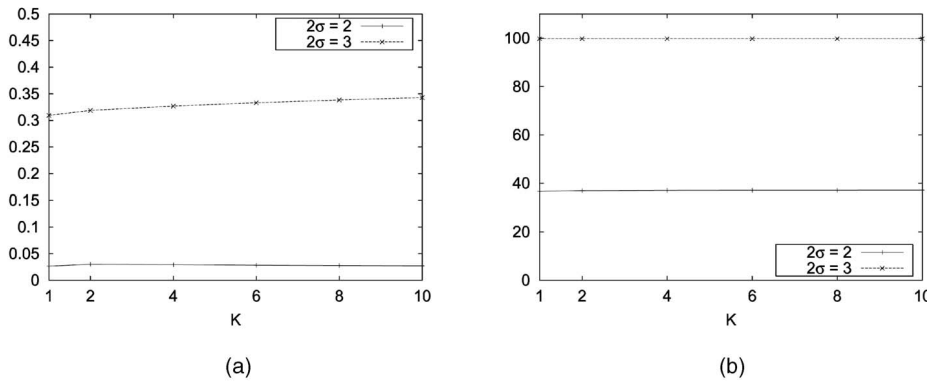


Fig. 20. An experiment with the approximate k -nearest neighbor algorithm in dimension 30. In (a), the average error is showed. (b) Depicts the gain percentage in the number of distance computations.

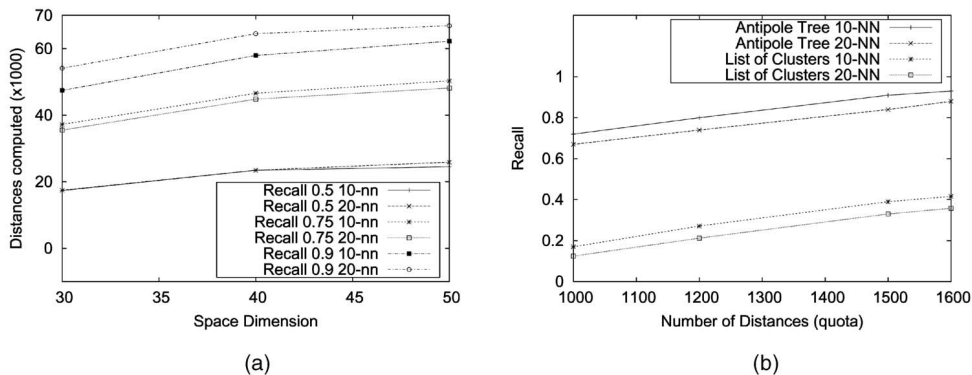


Fig. 21. (a) Analysis of curse of dimensionality using Antipole Tree from dimension 30 to 50. Number of distances needed fixing the recall. (b) Comparisons using the image histogram database between the Antipole Tree and List of Clusters with regard to approximated k -nearest neighbor. The recall varying the quota is depicted.

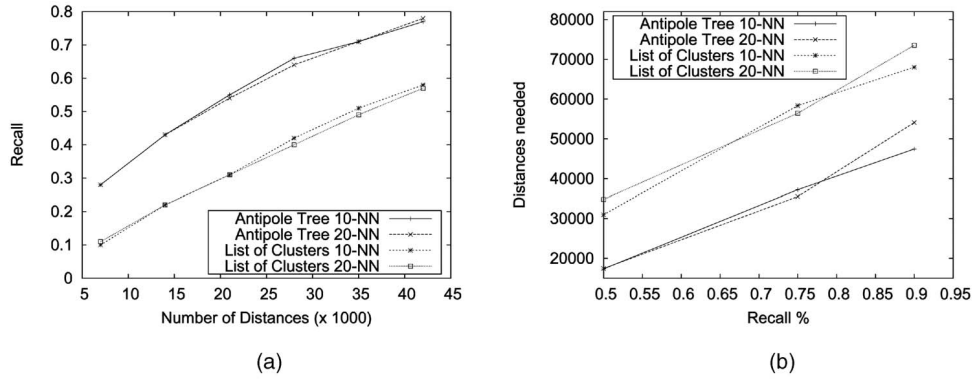


Fig. 22. Comparing Antipole Tree and List of Clusters with regard to approximated k -nearest neighbor. In (a), the Recall varying the quota is depicted. In (b), the number of distance computations with fixed recall are shown.

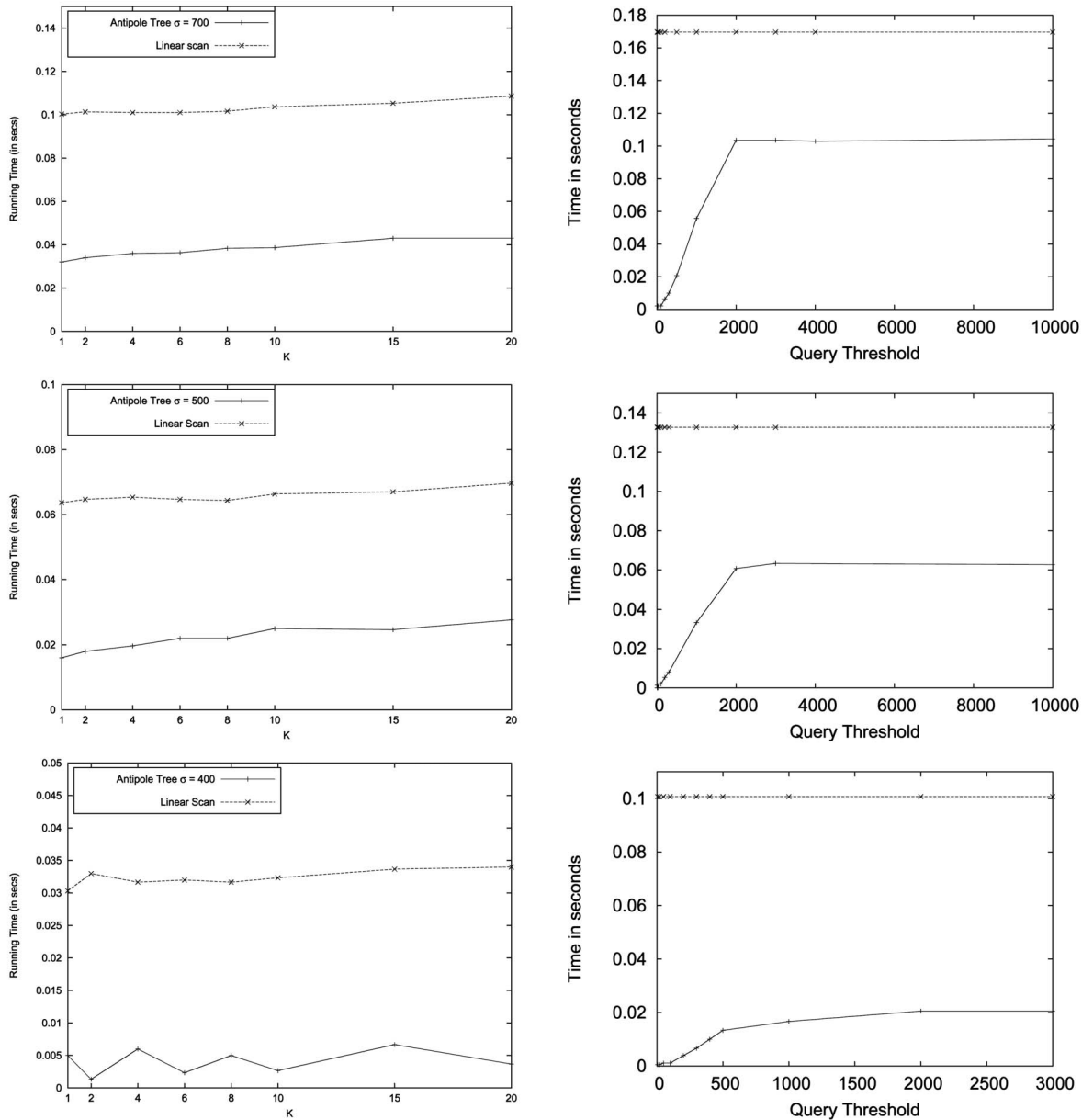


Fig. 23. Comparing Antipole Tree and linear scan with regard to k -nearest neighbor (left side) and range search (right side) in \mathbb{R}^{267} top, \mathbb{R}^{147} middle, and \mathbb{R}^{63} bottom.

a given texture, the data sets of tuples were built in the following way: For each pixel p in the texture, we considered, per color channel, half of its $h \times h$ neighborhood (see [51] for more details). We obtained data sets of dimension ranging from 63 to 267. Results, which are plotted in Fig. 23, show that the proposed data structure outperforms the linear scan in such high-dimensional data sets. We have also noticed that the intrinsic dimension of these spaces goes from 5 to 10.

9 CONCLUSIONS

We extended the ideas of the most successful best match retrieval data structures, such as M-Tree, MPV-Tree, FQ-Tree, and List of Clusters, by using pivots based on the farthest pairs (Antipoles) in data sets. The resulting Antipole Tree is a bisector tree using pivot-based clustering with bounded diameter. Both range and k -nearest neighbor searches are performed by eliminating those clusters which cannot contain the result of the query. Antipoles are found by playing a linear time randomized tournament among the elements of the input set.

Proliferation of clusters is limited by using a suitable diameter threshold, which is determined through a statistical analysis on the set of distances. Moreover, an estimate of the ratio between pseudodiameter (Antipole length) and the real diameter is used to determine when a splitting is needed. Since no guaranteed approximation algorithm for diameter computation in general metric spaces can exist, we used the approximation ratio given by a very efficient algorithm for diameter computation in Euclidean spaces together with the intrinsic dimension of the given metric space (Appendix, which can be found on the Computer Society Digital Library at <http://computer.org/tkde/archives.htm>).

By using the tournament size equal to 3 or $d - 1$, where d is the intrinsic dimension of the metric space, we obtained good experimental results. However, we are currently investigating from a theoretical point of view how to determine an optimal value for the tournament size parameter. Extensive experimentations have been performed on both synthetic and real data sets, with normal and clustered distributions. All the experiments have shown that our proposed structure outperforms the most successful data structures for best match search by a factor ranging between 1.5 and 2.5.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for useful suggestions and comments.

REFERENCES

- [1] P. Agarwal, J. Matousek, and S. Suri, "Farthest Neighbors, Maximum Spanning Trees, and Related Problems in Higher Dimensions," *Computational Geometry: Theory and Applications*, vol. 1, pp. 189-201, 1991.
- [2] C. Aggarwal, J.L. Wolf, P.S. Yu, and M. Epelman, "Using Unbalanced Trees for Indexing Multidimensional Objects," *Knowledge and Information Systems*, vol. 1, no. 3, pp. 157-192, 1999.
- [3] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications," *Proc. ACM SIGMOD*, pp. 94-105, 1998.
- [4] R. Baeza-Yates, W. Cunto, A. Manber, and S. Wu, "Proximity Matching Using Fixed-Queries Trees," *Proc. Combinatorial Pattern Matching, Fifth Ann. Symp.*, pp. 198-212, 1994.
- [5] G. Barequet and S. Har-Peled, "Efficiently Approximating the Minimum-Volume Bounding Box of a Point Set in Three Dimensions," *Proc. 10th Ann. ACM-SIAM Symp. Discrete Algorithms*, pp. 82-91, 1999.
- [6] S. Battiato, D. Cantone, D. Catalano, G. Cincotti, and M. Hofri, "An Efficient Algorithm for the Approximate Median Selection Problem," *Proc. Fourth Italian Conf. Algorithms and Complexity*, pp. 226-238, 2000.
- [7] S. Berchtold, D.A. Keim, and H.-P. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data," *Proc. 22nd Int'l Conf. Very Large Databases*, pp. 28-39, 1996.
- [8] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is Nearest Neighbor Meaningful?" *Proc. Seventh Int'l Conf. Database Theory*, vol. 1540, pp. 217-235, 1999.
- [9] T. Bozkaya and M. Ozsoyoglu, "Indexing Large Metric Spaces for Similarity Search Queries," *ACM Trans. Database Systems*, vol. 24, no. 3, pp. 361-404, 1999.
- [10] S. Brin, "Near Neighbor Search in Large Metric Spaces," *Proc. 21st Int'l Conf. Very Large Data Bases*, pp. 574-584, 1995.
- [11] W.A. Burkhard and R.M. Keller, "Some Approaches to Best-Match File Searching," *Comm. ACM*, vol. 16, no. 4, pp. 230-236, 1973.
- [12] B. Bustos and G. Navarro, "Probabilistic Proximity Searching Algorithms Based on Compact Partitions," *Proc. Symp. String Processing and Information Retrieval*, pp. 284-297, 2002.
- [13] I. Calantari and G. McDonald, "A Data Structure and an Algorithm for the Nearest Point Problem," *IEEE Trans. Software Eng.*, vol. 9, no. 5, pp. 631-634, 1983.
- [14] D. Cantone, G. Cincotti, A. Ferro, and A. Pulvirenti, "An Efficient Algorithm for the 1-Median Problem," *SIAM J. Optimization*, to appear.
- [15] T.M. Chan, "Approximating the Diameter, Width, Smallest Enclosing Cylinder, and Minimum-Width Annulus," *Int'l J. Computational Geometry and Applications*, vol. 12, nos. 1-2, pp. 67-85, 2002.
- [16] E. Chávez and G. Navarro, "An Effective Clustering Algorithm to Index High Dimensional Metric Spaces," *Proc. 11th Ann. ACM-SIAM Symp. Discrete Algorithms*, pp. 75-86, 2000.
- [17] E. Chávez and G. Navarro, "A Probabilistic Spell for the Curse of Dimensionality," *Proc. Third Workshop Algorithm Eng. and Experimentation (ALENEX '01)*, pp. 147-160, 2001.
- [18] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquin, "Searching in Metric Spaces," *ACM Computing Surveys*, vol. 33, no. 3, pp. 273-321, 2001.
- [19] B. Chazelle, "Computational Geometry: A Retrospective," *Proc. 26th Ann. ACM Symp. Theory of Computing*, pp. 75-94, May 1994.
- [20] P. Ciaccia and M. Patella, "Bulk Loading the M-Tree," *Proc. Ninth Australasian Database Conf. (ADC)*, pp. 15-26, 1998.
- [21] P. Ciaccia and M. Patella, "PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-Dimensional and Metric Spaces," *Proc. 16th Int'l Conf. Data Eng.*, pp. 244-255, 2000.
- [22] P. Ciaccia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. 23rd Int'l Conf. Very Large Data Bases*, pp. 426-435, 1997.
- [23] K. Clarkson, "Nearest Neighbor Queries in Metric Spaces," *Proc. 29th Ann. ACM Symp. Theory of Computing*, pp. 609-617, May 1997.
- [24] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," *Proc. Second Int'l Conf. Knowledge Discovery in Databases and Data Mining*, pp. 226-231, 1996.
- [25] T. Feder and D. Greene, "Optimal Algorithms for Approximate Clustering," *Proc. 20th Ann. ACM Symp. Theory of Computing*, pp. 434-444, 1988.
- [26] A.W.-C. Fu, P.M. Chan, Y.-L. Cheung, and Y. Moon, "Dynamic VP-Tree Indexing for n-Nearest Neighbor Search Given Pair-Wise Distances," *The VLDB J.*, vol. 9, no. 2, pp. 154-173, 2000.
- [27] V. Ganti, R. Ramakrishnan, J. Gehrke, A. Powell, and J. French, "Clustering Large Datasets in Arbitrary Metric Spaces," *Proc. IEEE 15th Int'l Conf. Data Eng.*, pp. 502-511, 1999.
- [28] A. Gersho and R. Gray, *Vector Quantization and Signal Compression*. Kluwer Academic, 1992.
- [29] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," *Proc. 25th Int'l Conf. Very Large Data Bases*, pp. 518-529, 1999.
- [30] T. Gonzalez, "Clustering to Minimize the Maximum Intercluster Distance," *Theoretical Computer Science*, vol. 38, pp. 293-306, 1985.
- [31] T. Gonzalez, "Covering a Set of Points in Multidimensional Space," *Information Processing Letters*, vol. 40, pp. 181-188, 1991.

- [32] S. Guha, R. Rastogi, and K. Shim, "Cure: An Efficient Clustering Algorithm for Large Databases," *Proc. ACM SIGMOD*, pp. 73-84, 1998.
- [33] S. Har-Peled, "A Practical Approach for Computing the Diameter of a Point Set," *Proc. 17th Symp. Computational Geometry*, pp. 177-186, 2001.
- [34] G.R. Hjaltason and H. Samet, "Distance Browsing in Spatial Database," *ACM Trans. Information Systems*, vol. 24, no. 2, pp. 265-318, 1999.
- [35] D.S. Hochbaum and W. Maass, "Approximation Schemes for Covering and Packing Problems in Image Processing and VLSI," *J. ACM*, vol. 32, no. 1, pp. 130-136, 1985.
- [36] P. Indyk, "Sublinear Time Algorithms for Metric Space Problems," *Proc. 31st Ann. ACM Symp. Theory of Computing*, pp. 428-434, 1999.
- [37] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," *Proc. 30th Ann. ACM Symp. Theory of Computing*, pp. 604-613, 1998.
- [38] C. Li, E. Chang, and H.G.-M.G. Wiederhold, "Clustering for Approximate Similarity Search in High-Dimensional Spaces," *IEEE Trans. Knowledge and Data Eng.*, vol. 14, no. 4, pp. 792-808, July-Aug. 2002.
- [39] T.M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.
- [40] G. Navarro, "Searching in Metric Spaces by Spatial Approximation," *The VLDB J.*, vol. 11, pp. 28-46, 2002.
- [41] R. Ng and J. Han, "Clarans: A Method for Clustering Objects for Spatial Data Mining," *IEEE Trans. Knowledge and Data Eng.*, vol. 14, no. 5, pp. 1003-1016, Sept./Oct. 2002.
- [42] H. Nolte, K. Verbar, and C. Zirkelbach, "Monotonous Bisection* Trees—A Tool for Efficient Partitioning of Complex Scenes of Geometric Objects," *Data Structure and Efficient Algorithms*, Lecture Notes in Computer Sciences, vol. 594, pp. 186-203, Springer-Verlag, 1992.
- [43] C. Procopiuc, "Geometric Techniques for Clustering Theory and Practice," PhD dissertation, Duke Univ., 2001.
- [44] M. Shapiro, "The Choice of Reference Points in Best-Match File Searching," *Comm. ACM*, vol. 20, no. 5, pp. 339-343, 1997.
- [45] D. Shasha and T.-L. Wang, "New Techniques for Best-Match Retrieval," *ACM Trans. Information Systems*, vol. 8, no. 2, pp. 140-158, 1990.
- [46] G. Sheikholeslami, S. Chatterjee, and A. Zhang, "WaveCluster: A Wavelet Based Clustering Approach for Spatial Data in Very Large Databases," *The VLDB J.*, vol. 8, nos. 3-4, pp. 289-304, 2000.
- [47] S. Sumanasekara and M. Ramakrishna, "CHILMA: An Efficient High Dimensional Indexing Structure for Image Databases," *Proc. First IEEE Pacific-Rim Conf. Multimedia*, pp. 76-79, 2000.
- [48] C. Traina Jr., A. Traina, B. Seeger, and C. Faloutsos, "Slim-Trees: High Performance Metric Trees Minimizing Overlap between Nodes," *Proc. Seventh Int'l Conf. Extending Database Technology*, vol. 1777, pp. 51-65, 2000.
- [49] J. Uhlmann, "Satisfying General Proximity/Similarity Queries with Metric Trees," *Information Processing Letters*, vol. 40, pp. 175-179, 1991.
- [50] VisTex, http://graphics.stanford.edu/projects/texture/demo/synthesis_VisTex_192.html, Texture Synthesis: VisTex Texture, 2004.
- [51] L. Wei and M. Levoy, "Texture Synthesis over Arbitrary Manifold Surfaces," *Proc. ACM-SIGGRAPH '01*, pp. 355-360, 2001.
- [52] P. Yianilos, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces," *Proc. Third Ann. ACM-SIAM Symp. Discrete Algorithms*, pp. 311-321, Jan. 1993.
- [53] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An Efficient Data Clustering Method for Very Large Databases," *Proc. ACM SIGMOD Conf. Management of Data*, pp. 103-114, 1996.



Domenico Cantone received the BS degree in mathematics from the University of Catania, Italy, in 1982. In 1985 and 1987, he received the MS and PhD degrees in computer science from New York University. In 1990, he became a professor of computer science at the University of L'Aquila, Italy. In 1991, he moved to the University of Catania, where since 2002 he has been the director of the graduate studies in computer science. Since 1995, he has also been a member of the Board of Directors of the journal *Le Matematiche*. His main scientific interests include: computable set theory, automated deduction in various mathematical theories, and, more recently, string matching and algorithmic engineering. In the field of computable set theory, he has coauthored two monographs published in 1989 and 2001, and he is currently working on a third monograph which will be published in 2005.



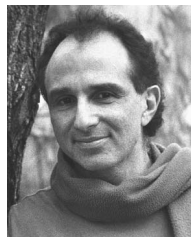
Alfredo Ferro received the BS degree in mathematics from Catania University, Italy, in 1973 and the PhD degree in computer science from New York University (NYU) in 1981 (Jay Krakauer Award for the best dissertation in the field of sciences at NYU). He is currently professor of computer science at Catania University and has been director of graduate studies in computer science for several years. Since 1989, he has been the director of the International School for Computer Science Researchers (Lipari School, <http://lipari.cs.unict.it>). Together with Michele Purrello, he is the director of the International School in BioMedicine and Bioinformatics (<http://lipari.cs.unict.it/bio-info/>). His research interests include bioinformatics, algorithms for large data sets management, data mining, computational logic, and networking.



Alfredo Pulvirenti received the BS degree in computer science from Catania University, Italy, in 1999 and the PhD degree in computer science from Catania University in 2003. He has currently a postdoctoral position in the Department of Computer Science at Catania University. His research interests include bioinformatics, data structure, approximate algorithms, structured databases, information retrieval, graph theory, and networking.



Diego Reforgiato Recupero received the BS degree in computer science in 2001 from the University of Catania, Italy. He is currently a PhD candidate in computer science at the same university. His research interests include database systems and information technologies. In particular, he has been working on clustering data in high-dimensional spaces, graphs clustering, and graph matching problem.



Dennis Shasha is a professor of computer science in the Courant Institute at New York University, where he works with biologists on pattern discovery for microarrays, combinatorial design, and network inference; and with physicists and financial people on algorithms for time series. Other areas of interest include database tuning, tree and graph matching, and cryptographic file systems. In his spare time, he has written three books of puzzles, a biography of

great computer scientists, and technical books about database tuning, biological pattern recognition, and a book on time series. He also writes the puzzle column for *Scientific American* and *Dr. Dobbs Journal*.