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

D. Cantone, A. Ferro, A. Pulvirenti, D. Reforgiato

Dipartimento di Matematica e Informatica,

Università degli Studi di Catania, Italy

e-mail: {cantone, ferro, apulvirenti, diegoref}@dmi.unict.it

D. Shasha

Computer Science Department, New York University, USA

e-mail: shasha@cs.nyu.edu

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 target 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 Multi-Vantage 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 $dist(a, b)$ 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 compares very well with existing structures such as List of Clusters, M-Trees and others, and in many cases it achieves better results.

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], 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 in 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-medians[1] 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 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 non-uniform. We successfully compare our algorithm with linear scan in non-uniform data sets of very high-dimensional Euclidean spaces. We draw our conclusions in Section 9. Finally, Appendix A proposes an efficient approximation scheme for the diameter computation in the Euclidean case.

2 Basic Definitions and Related Works

Let M be a non-empty set of objects and let $dist : (M \times M) \longrightarrow \mathbb{R}$ 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);

[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.

3. $(\forall x \in M)\, dist(x, x) = 0$ (reflexivity),

   $(\forall x, y \in M)\, (x \neq y \rightarrow dist(x, y) > 0)$ (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 $r_1, \cdots, r_k$ 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 sub-tree 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 $a_1, \cdots, a_m$ based on p_a and the partitions $b_1, \cdots, b_m$ 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 pre-computed 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 Chávez et al. [16], proposes a structure called List of Clusters. Such 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 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 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 Appendix A. 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 $X_1, \ldots, X_k$. Then, each subset X_i is locally processed by a procedure which computes its exact 1-median x_i. The elements $x_1, \ldots, x_k$ move to the next round. The tournament terminates when we are left with a single element $\overline{x}$, 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 above procedure takes time $\frac{t}{2}n + o(n)$ in the worst-case.

The approximate 1-Median selection algorithm

```
1-MEDIAN ($X$)
    for each $x \in X$ do
        $\mathcal{S}_x \leftarrow \sum_{y \in X} dist(x, y)$;
    Let $m \in X$ be an element
    such that $\mathcal{S}_m = \min_{x \in X}(\mathcal{S}_x)$;
    return $m$
```

```
APPROX_1_MEDIAN ($S$)
    while $|S| > threshold$ do
        $W \leftarrow \emptyset$;
        while $|S| \geq 2\tau$ do
            Choose randomly a subset $T \subseteq S$, with $|T| = \tau$;
            $S \leftarrow S \setminus T$;
            $W \leftarrow W \cup \{$1-MEDIAN $(T)\}$;
        end while;
        $S \leftarrow W \cup \{$1-MEDIAN $(S)\}$;
    end while;
    return 1-MEDIAN $(S)$;
```

Figure 1: The 1-Median algorithm.

3.2 The Diameter (Antipole) computation

Let (M, d) be a metric space with distance function $dist : (M \times M) \longmapsto \mathbb{R}$ 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 $\overline{X}_i$, obtained by removing the element x_i from X_i. The elements in $\overline{X}_1 \cup \overline{X}_2 \ldots \cup \overline{X}_k$ are used in the subsequent step. The tournament terminates when we are left with a single set, $\overline{X}$, from which we extract the final winners A, B, as the furthest points in $\overline{X}$. The pair A, B is called the *Antipole pair* and their distance represents the approximate diameter of the set S.

The approximate Antipole selection algorithm

```
LOCAL_WINNER($T$)
    return $T \setminus$ 1-MEDIAN($T$);
    END LOCAL_WINNER

FIND_ANTIPOLE($T$)
    return $P_1, P_2 \in T$ such that
            $dist(P_1, P_2) \geq dist(x, y)$
            $\forall x, y \in T$;
    END FIND_ANTIPOLE
```

```
APPROX_ANTIPOLE($S$)
    while $|S| > threshold$ do
        $W \leftarrow \emptyset$;
        while $S \geq 2\tau$ do
            Choose randomly a subset $T \subseteq S : |T| = \tau$;
            $S \leftarrow S \setminus T$;
            $W \leftarrow W \cup$ {LOCAL_WINNER($T$)};
        end while
        $S \leftarrow W \cup$ {LOCAL_WINNER($S$)};
    end while
    return FIND_ANTIPOLE(S);
    END APPROX_ANTIPOLE
```

Figure 2: The pseudocode of Antipole Algorithm.

The pseudocode of the Antipole algorithm APPROX_ANTIPOLE, similar to that of the 1-Median algorithm, is given in Fig. 2.

A faster (but less accurate) variant of APPROX_ANTIPOLE can be used. Such variant, called FAST_APPROX_ANTIPOLE, consists of taking $\overline{X}_i$ 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 loose any accuracy.

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 $\textit{threshold} = \min\left\{\tau^2 - 1, \sqrt{|S|}\right\}$.

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:

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

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

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

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 $\left\lceil \log_{\tau/(\tau-1)} n \right\rceil$ 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

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

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 $W_1(n) = \lfloor \frac{n}{\tau} \rfloor + W_1\left(2 \cdot \lfloor \frac{n}{\tau} \rfloor\right)$, for $n \geq 2\tau$. By induction on n, we can show that the number of calls to the subroutine FIND_ANTIPOLE is $W_1(n) \leq \left\lceil \frac{n}{\tau-2} \right\rceil$. 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. 3 (a)) 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. 3 (b)) stores the following information:

- *centroid*, C, the element that minimizes the sum of the distances from the other cluster members;

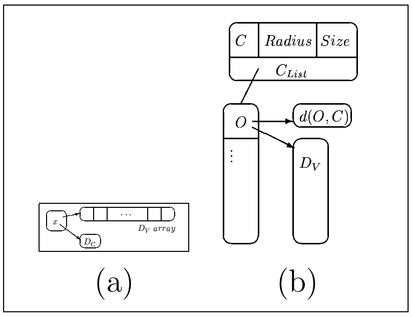


Figure 3: (a) A generic *object* in the Antipole data structure. (b) A generic cluster in the Antipole data structure.

- *radius*, *Radius*, containing the distance from C to the farthest *object*;
- *member list*, C_{List}, storing the catalogue 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 (i) the identities of two Antipole objects A and B, called the Antipole pair of distance at least 2σ apart, (ii) the radii Rad_A and Rad_B of the two sub-sets (S_A, S_B obtained by splitting S based on their proximity to A and B respectively), and (iii) pointers to the left and right sub-trees *left* and *right*;
- A leaf node stores a cluster.

The Build Antipole Tree algorithm

```
BUILD($S$, $\sigma$, $Q$)
  if $Q = \emptyset$ then
     $Q \leftarrow$ ADAPTED_APPROX_ANTIPOLE($S$,$\sigma$);
     if $Q = \emptyset$ then // splitting condition $\Phi$ fails
        $T.Leaf \leftarrow TRUE$;
        $T.Cluster \leftarrow$ MAKE_CLUSTER($S$);
        return $T$;
     end if;
  end if;
  {A,B} $\leftarrow Q$;
  $T.A \leftarrow A$;
  $T.B \leftarrow B$;
  $S_A \leftarrow \{O \in S | dist(O, A) < dist(O, B)\}$;
  $S_B \leftarrow \{O \in S | dist(O, B) \leq dist(O, A)\}$;
  for each $O \in S$;
       $O.D_V \leftarrow O.D_V \cup$
       $\{(A, dist(O, A)), (B, dist(O, B))\}$;
  end for each;
  $T.Rad_A \leftarrow \max_{O \in S_A} dist(O, A)$;
  $T.Rad_B \leftarrow \max_{O \in S_B} dist(O, B)$;
  $T.left \leftarrow$ BUILD($S_A$,$\sigma$,CHECK($S_A$,$\sigma$,$A$));
  $T.right \leftarrow$ BUILD($S_B$,$\sigma$,CHECK($S_B$,$\sigma$,$B$));
  return $T$;
  END BUILD.

MAKE_CLUSTER($S$)
  $Cluster.C \leftarrow$ APPROX_1_MEDIAN($S$);
  $Cluster.Radius \leftarrow \max_{x \in S} dist(x, Cluster.C)$
  $Cluster.C_{List} \leftarrow S \setminus \{Cluster.C\}$;
  for each $x \in Cluster.C_{List}$;
       $x.D_C \leftarrow dist(x, Cluster.C)$;
  end for each;
  return $Cluster$;
  END MAKE_CLUSTER
```

Figure 4: The algorithm Build Antipole Tree and the routine MakeCluster.

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 sub-routine

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.

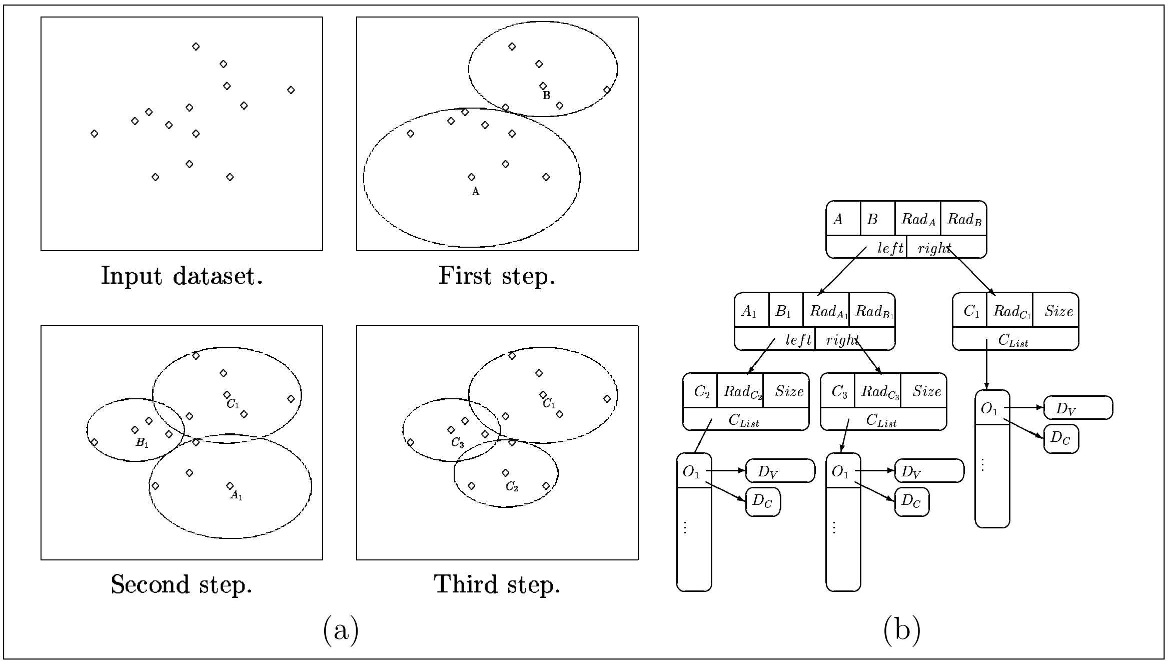
The routine MAKE_CLUSTER (Fig. 4) creates a cluster of objects with bounded radius. This procedure computes the cluster centroid C with the randomized algorithm APPROX_1_MEDIAN and then computes the distance between each O in the cluster and C.

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. 5 (a) 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. 5 (b) 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.

[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.


Input dataset.

First step.

Second step.

Third step.

(a)

(b)

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

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 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

```
RANGE_SEARCH($T$, $q$, $t$, $OUT$)
if ($T.Leaf = FALSE$) then
    $D_A \leftarrow dist(q, T.A)$;
    $D_B \leftarrow dist(q, T.B)$;
    if ($D_A \leq t$) then
        $OUT \leftarrow OUT \cup \{T.A\}$;
    end if;
    if ($D_B \leq t$) then
        $OUT \leftarrow OUT \cup \{T.B\}$;
    end if;
    $q.D_V \leftarrow q.D_V \cup \{D_A, D_B\}$;
    if ($D_A \leq t + T.Rad_A$) then
        RANGE_SEARCH($T.left$, $q$, $t$, $OUT$);
    end if;
    if ($D_B \leq t + T.Rad_B$) then
        RANGE_SEARCH($T.right$, $q$, $t$, $OUT$);
    end if;
    $q.D_V \leftarrow q.D_V \setminus \{D_A, D_B\}$;
    return;
else // leaf case
    $OUT \leftarrow OUT \cup$ {VISIT_CL($T.Cluster$, $q$, $t$, $OUT$)};
end if;
END RANGE_SEARCH.
```

Figure 6: The Range Search algorithm.

```
VISIT_CL($Cluster$, $q$, $t$, $OUT$)
$q.D_C \leftarrow dist(q, Cluster.C)$;
if ($q.D_C \leq t$) then
    $OUT \leftarrow OUT \cup \{Cluster.C\}$;
end if;
if ($q.D_C \geq t + Cluster.Radius$) then
    return;
end if;
if ($q.D_C \leq t - Cluster.Radius$) then
    $OUT \leftarrow OUT \cup Cluster$;
    return $OUT$;
end if;
for each $O \in Cluster.C_{List}$ do
        if ($q.D_C \geq t + O.D_C$) then
            continue;
        end if;
    if ($q.D_C \leq t - O.D_C$) then
        $OUT \leftarrow OUT \cup \{O\}$;
        continue;
    end if;
    if ($\nexists(d_q \in q.D_V$ and $d_O \in O.D_V) \mid$
        $d_q \geq t + d_O$ or $d_q \leq t - d_O$) then
        if ($dist(q, O) \leq t$) then
            $OUT \leftarrow OUT \cup \{O\}$;
        end if;
    else
        if ($d_q \leq t - d_O$) then
            $OUT \leftarrow OUT \cup \{O\}$;
        end if;
    end if;
end for each;
return $OUT$;
END VISIT_CL.
```

Figure 7: The Visit Cluster algorithm.

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 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 sub-tree, right sub-tree 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.

```
K_NEAREST_NEIGHBOR($T$, $q$, $t$, $outQueue$, $k$, $pQueue$)
Enqueue($pQueue, Tree, NULL$);
while NotEmpty($pQueue$) do
        $node$ = Dequeue($pQueue$);
        if ($node.leaf = TRUE$) then
           KNN_VISIT_CLUSTER($node$, $q$, $t$, $outQueue$,$k$);
        else
              $D_A \leftarrow$ KNN_CHECK($q$, $node.A$, $t$, $outQueue$);
              $D_B \leftarrow$ KNN_CHECK($q$, $node.B$, $t$, $outQueue$);
              Enqueue($pQueue$, $node.left$,$D_A$-$node.Rad_A$);
              Enqueue($pQueue$, $node.right$,$D_B$-$node.Rad_B$);
        end if;
end while;
END K_NEAREST_NEIGHBOR.
```

Figure 8: The incremental k-nearest neighbor search algorithm.

```
KNN_CHECK($q$, $O$, $t$, $OUT$)
  $D_O \leftarrow dist(q, O)$;
  if ($|OUT| < k$) then
     HEAP_INSERT($O$, $OUT$);
     $t \leftarrow$ HEAP_EXTRACT_MAX($OUT$);
  else
       if ($D_O < t$) then
          HEAP_INSERT($O$, $OUT$);
          $t \leftarrow$ HEAP_EXTRACT_MAX($OUT$);
       end if;
  end if;
  return $D_O$;
  END KNN_CHECK.
```

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

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 100000, 200000, $\ldots$, 500000 objects uniformly distributed in $[0, 1]^{10}$);
- clustered 20-dimensional Euclidean space. More precisely a set of 100000 objects obtained in the following way: by using uniform distributions, take 100 random spheres and select 1000 random points in each of them.

The real data sets are respectively:

- a set of 45000 strings chosen from the Linux dictionary with the editing distance;
- a set of 42000 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. 10 (a) 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 100000 to 500000 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. 10 (a) 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 0.2% 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. 10 (b) 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 $\mathrm{O}(n \log n)$ 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

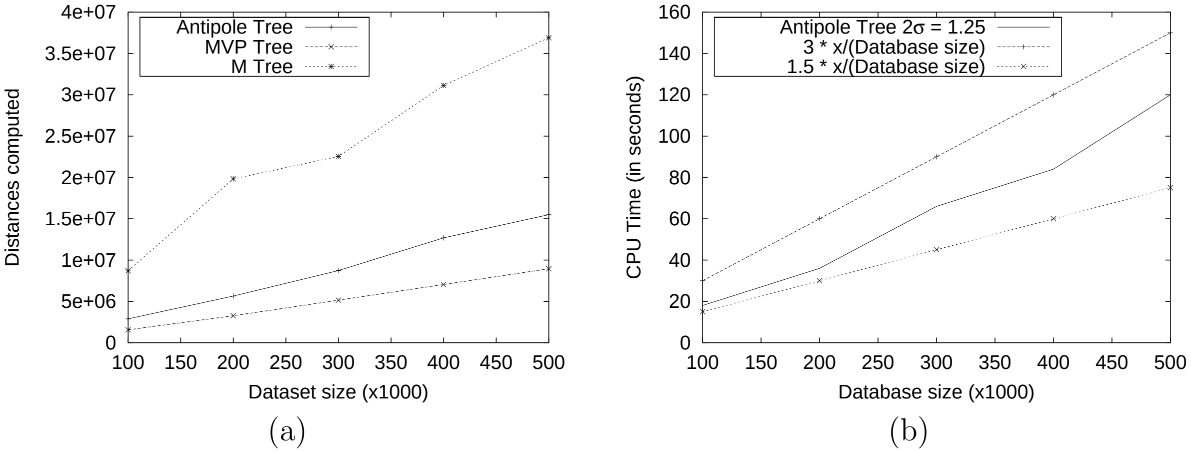


Figure 10: Construction complexity using uniformly generated data. (a) It is measured by the number of distance computations needed by the Antipole Tree with cluster diameter 1.25 vs M-Tree and MVP-Tree. (b) CPU time in seconds needed to build the Antipole Tree.

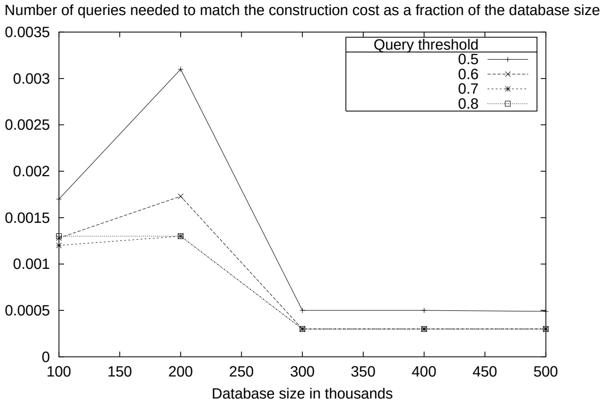


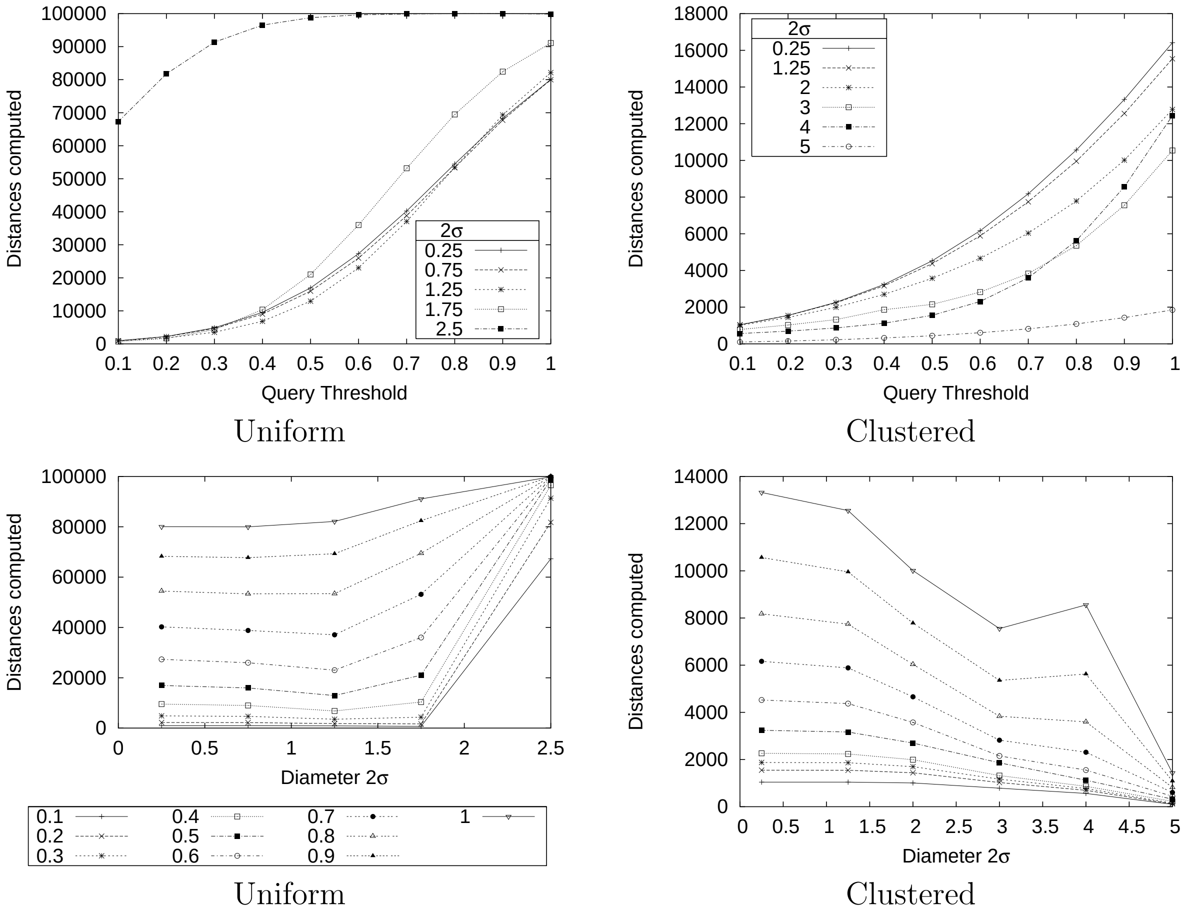
Figure 11: Number of range queries, as a fraction of the data set size, which are sufficient to recover the higher cost of Antipole Tree construction with respect to MVP-Tree construction.

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 100000. However, the methodology of finding the optimal diameter can be applied to other dimensions and arbitrary data sizes.

Figs. 12 (Uniform) and (Clustered) show that across different values of the threshold



Uniform

Clustered

Uniform

Clustered

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

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% 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 is the dimension 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 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 $p3$ and $p5$ suggested by the authors in [16]. $p3$ consists of choosing the center of the i-th cluster as the furthest element from the $(i-1)$-th center, whereas $p5$ 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 300000. 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.

[3] We are grateful to T. Bozkaya and M. Ozsoyoglu for providing us the program to generate the input for the clustered data set.

[4] We are grateful to P. Ciaccia, M. Patella, and P. Zezula for providing us the source code of the M-Tree.

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% and 70% of the distance computations, which in the figure is indicated as the gain percentage.

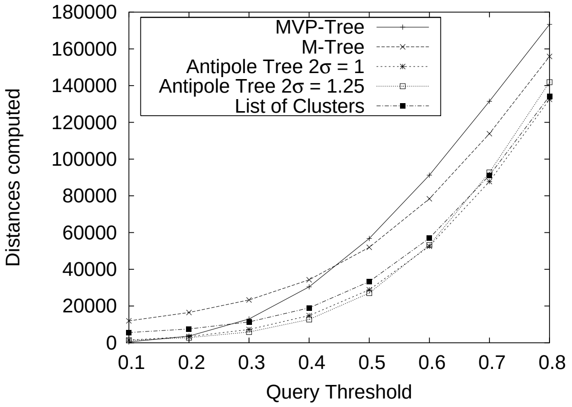
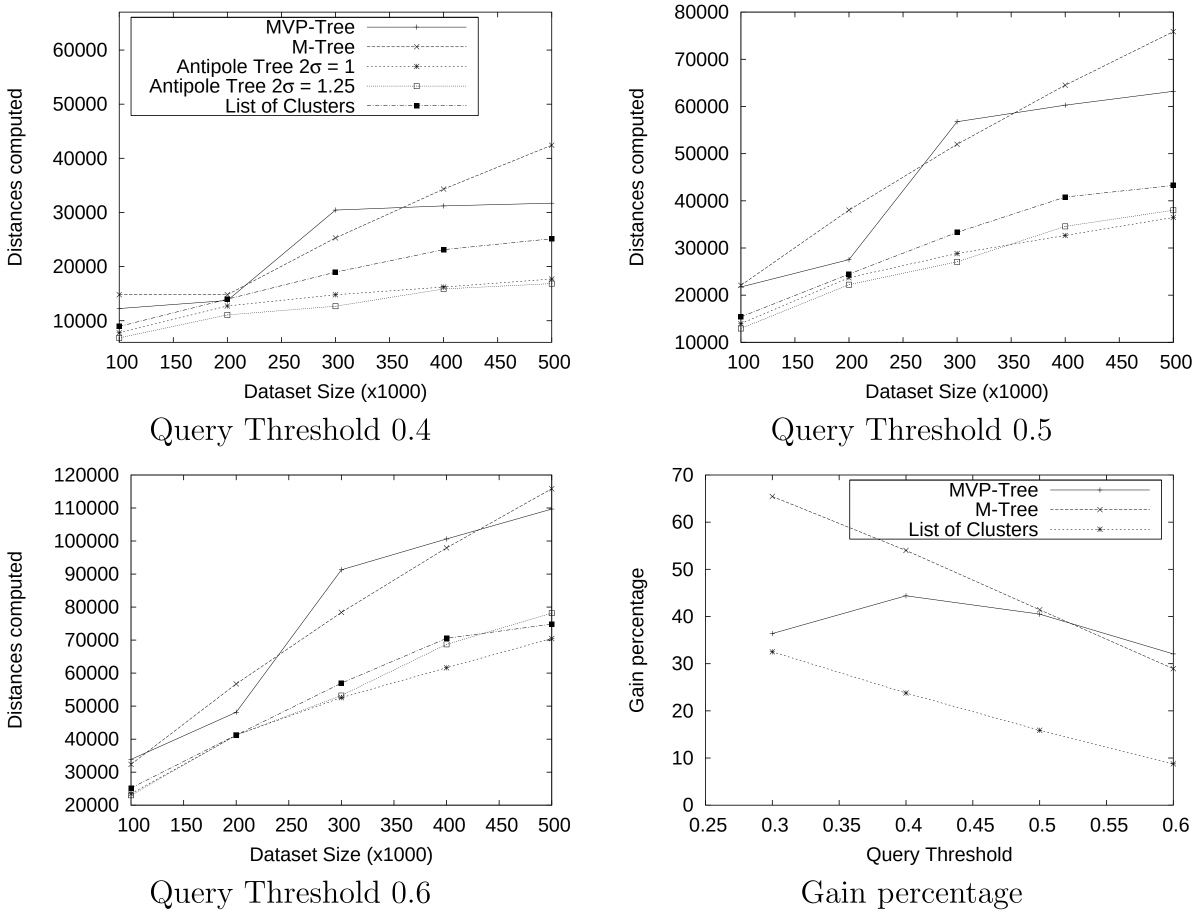


Figure 13: Comparisons in $\mathbb{R}^{10}$ using 300000 randomly generated vectors. The query threshold goes from 0.1 to 0.8.

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: 100000 clustered points, 45000 strings from the Linux dictionary, and 42000 image histograms from the Corel image database,[5] respectively. Results show a 30% 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 Space $\mathbb{R}^{147}$ and $\mathbb{R}^{267}$, with a database size 3000 obtained from the VISTEX [50] texture database. Notice that by using the query thresholds depicted in Fig. 16 the output set captures from 0% to 5% of the elements of the entire data set in $\mathbb{R}^{147}$ and from 0% to 10% of the elements of the entire data set in $\mathbb{R}^{267}$. Antipole Tree shows a better behavior w.r.t. List of Clusters tuned with the best fixed bucket size we noticed.

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



Query Threshold 0.4

Query Threshold 0.5

Query Threshold 0.6

Gain percentage

Figure 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 w.r.t. the MVP-Tree, the M-Tree, and the List of Clusters is also plotted.

6.4 K-Nearest Neighbor comparisons

In the 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 100000. We run the K_NEAREST_NEIGHBOR algorithm with $k = 1, 2, 4, 6, 8, 10, 15, 20$ using one hundred queries for each experiment (half belonging to the data structure and half not). Using the Antipole Tree we save up to 85% 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 3000 elements in Euclidean $\mathbb{R}^{147}$ and $\mathbb{R}^{267}$ from

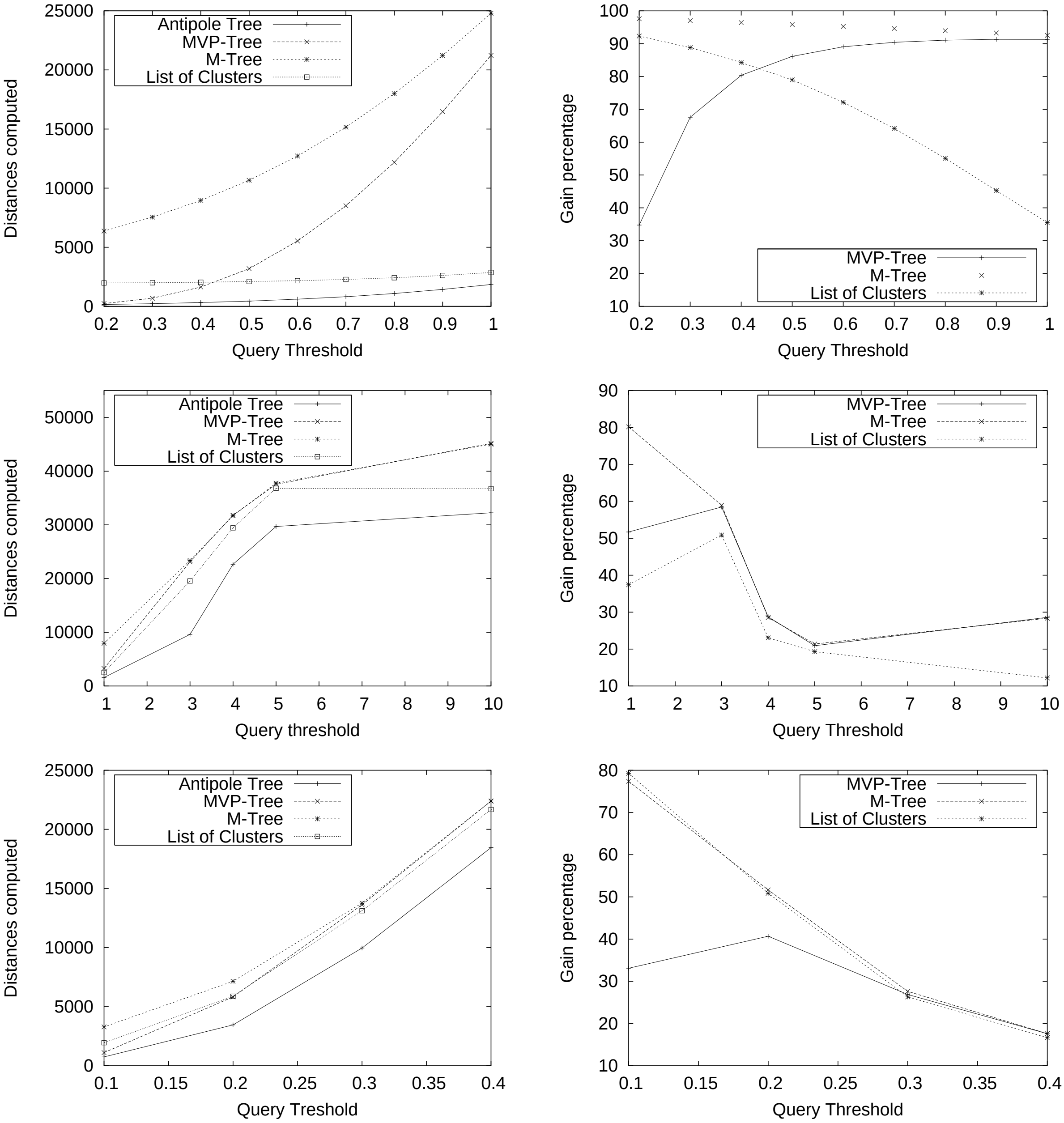


Figure 15: (top) Comparisons of Antipole Tree vs. MVP-Tree, M-Tree, and List of Clusters in a clustered space from $\mathbb{R}^{20}$ varying the query threshold from 0.1 to 1, with cluster radius 2. (middle) Antipole Tree vs. MVP-Tree, M-Tree, and List of Clusters using an editing distance metric with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

VISTEX [50]. Antipole Tree clearly outperforms List of Clusters.

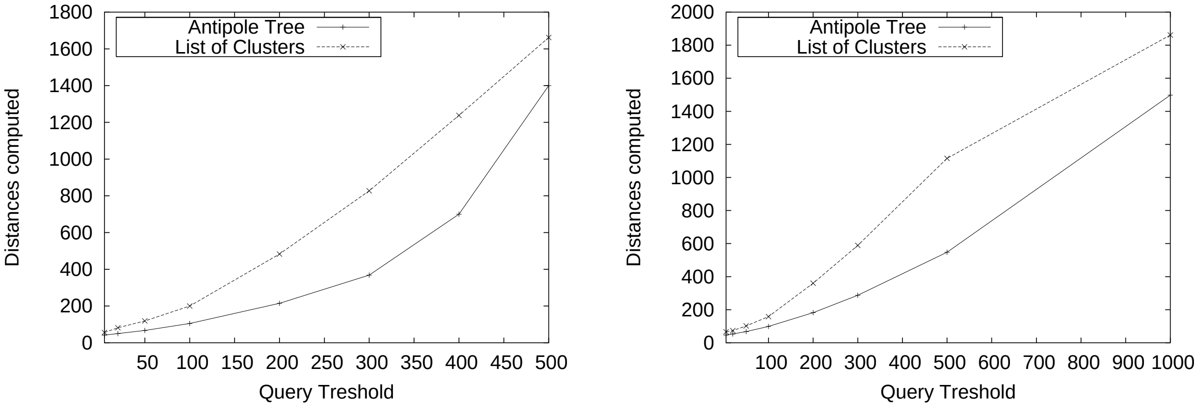


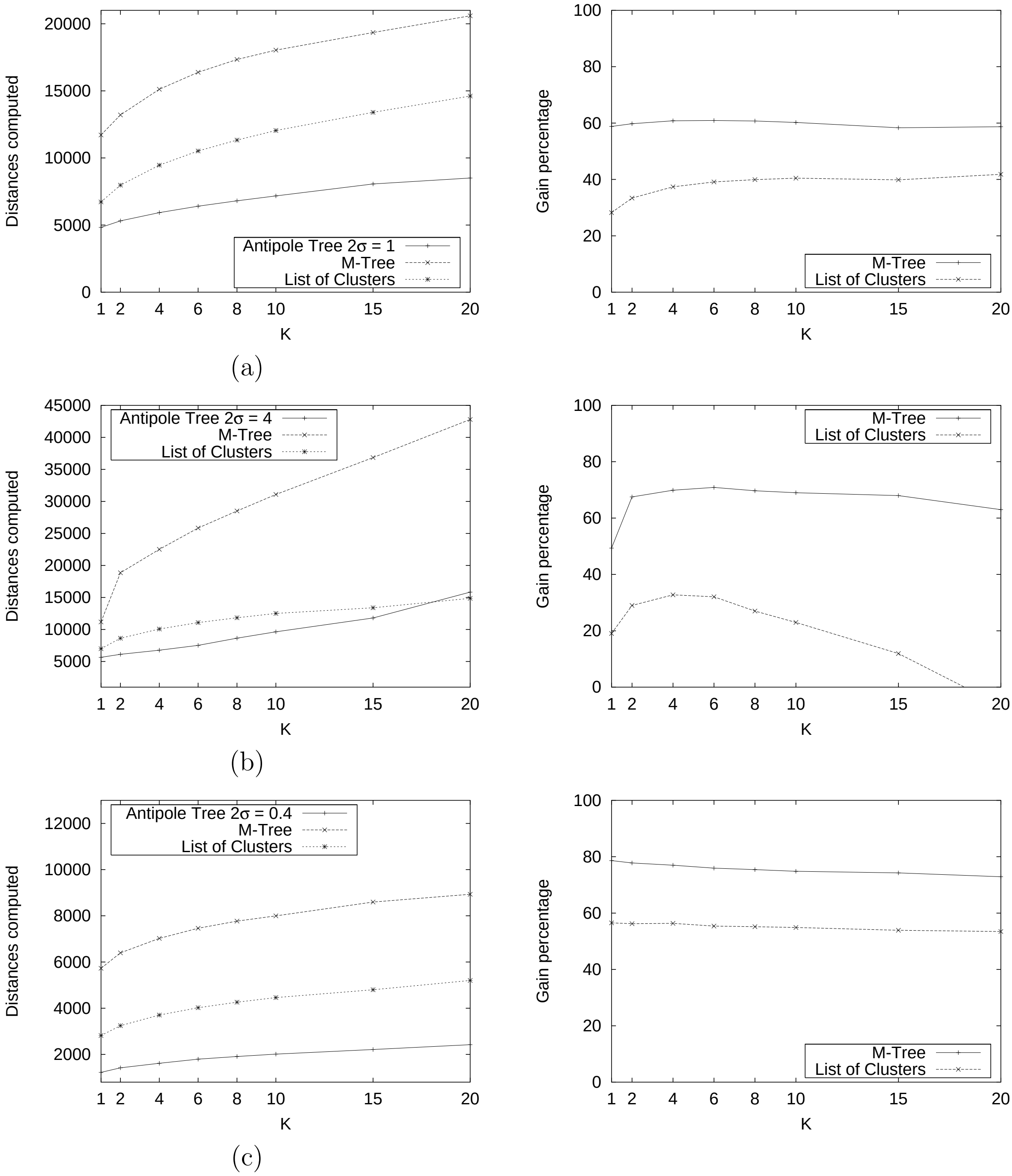
Figure 16: A comparison between Antipole Tree and List of Clusters using real database in $\mathbb{R}^{147}$ (left) and $\mathbb{R}^{267}$ (right).

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, 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 $\rho = \frac{\mu_S^2}{2\sigma_S^2}$, 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 of the TSVQ to find quickly an approximate nearest neighbor of a query object.



(a)

(b)

(c)

Figure 17: k-nearest neighbor comparisons. (a) 100000 uniformly generated points in $[0,1]^{10}$. (b) 100000 points from $\mathbb{R}^{20}$ generated in clusters. (c) Comparisons using the image histogram database.

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 one hundred queries were executed. In order to evaluate the quality of the results, we run the exact search first.

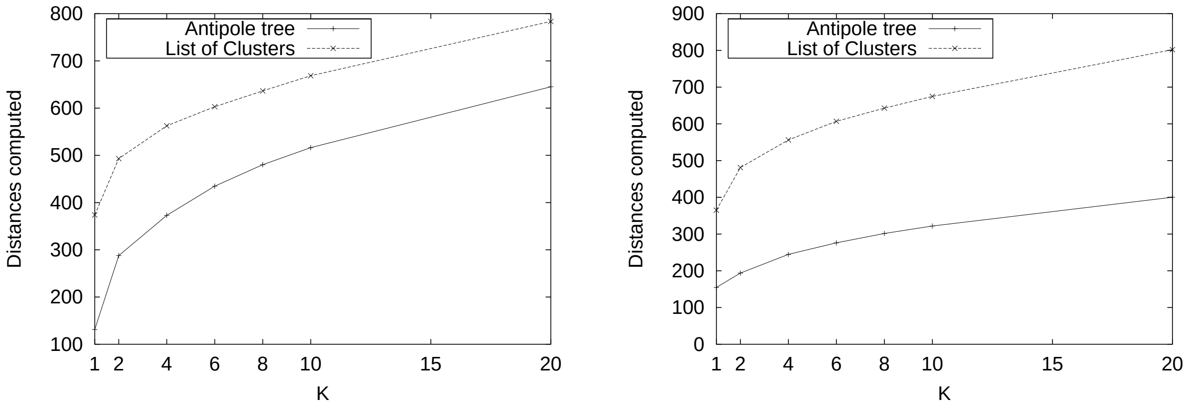


Figure 18: k-nearest neighbor search using real data from the VISTEX database in dimension $\mathbb{R}^{147}$ and $\mathbb{R}^{267}$.

Then the error δ is computed in the following way:

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

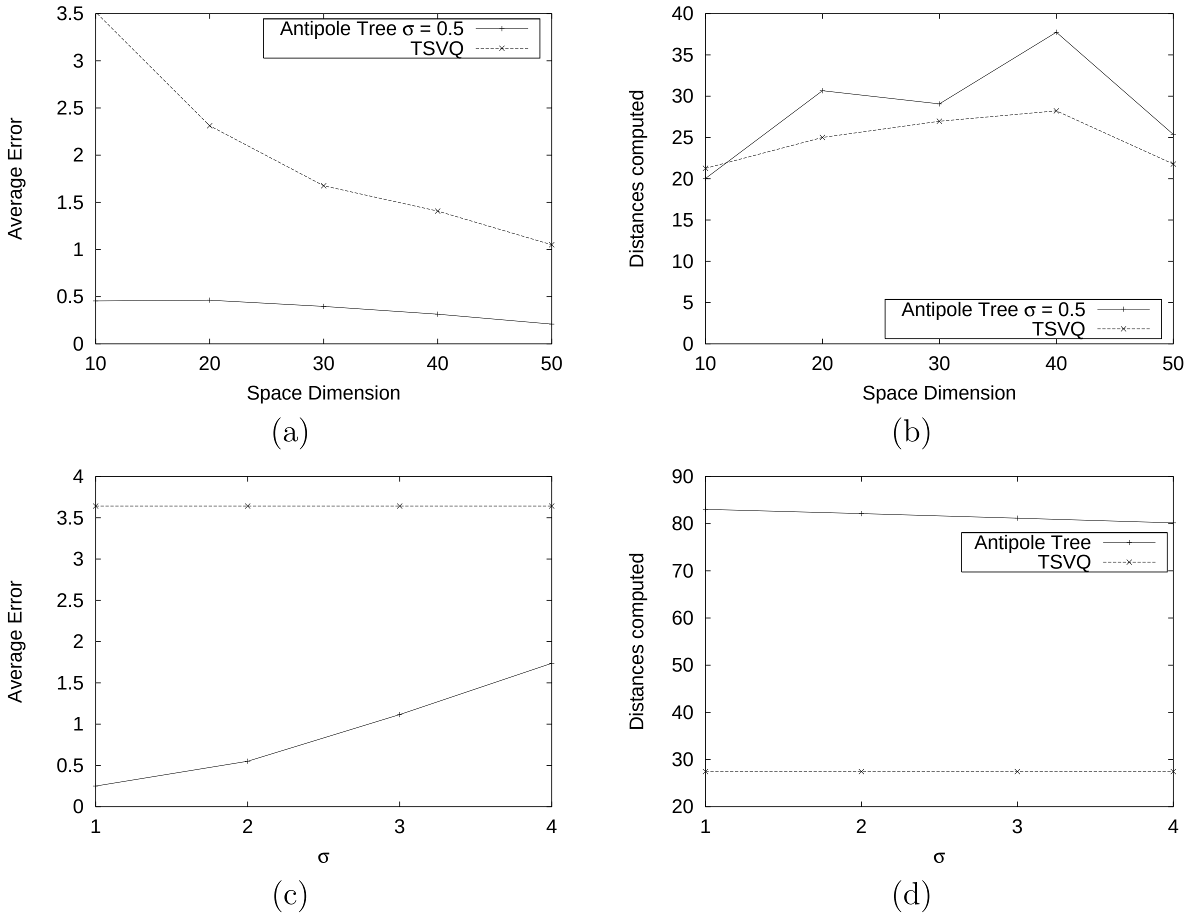
In Fig. 19 (a) the errors introduced by the two approximate algorithms in uniformly generated set of points (upper figures) and clustered set of points (lower figures) are depicted. On the other hand, Figs. 19 (b), (d) show the number of distances computed by the two algorithms.

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. 20 (a) shows a set of experiments done in uniform spaces in dimension 30 with radius σ set to 1 and 1.5.

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



(a) (b) (c) (d)

Figure 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.

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%), the *R-precision* (precision after R recall) 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 100000 elements of dimension 30. We fixed several quotas and recalls ranging from 7000 to 42000 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 size). Fig 21 (left) makes the same comparison but using Image histogram

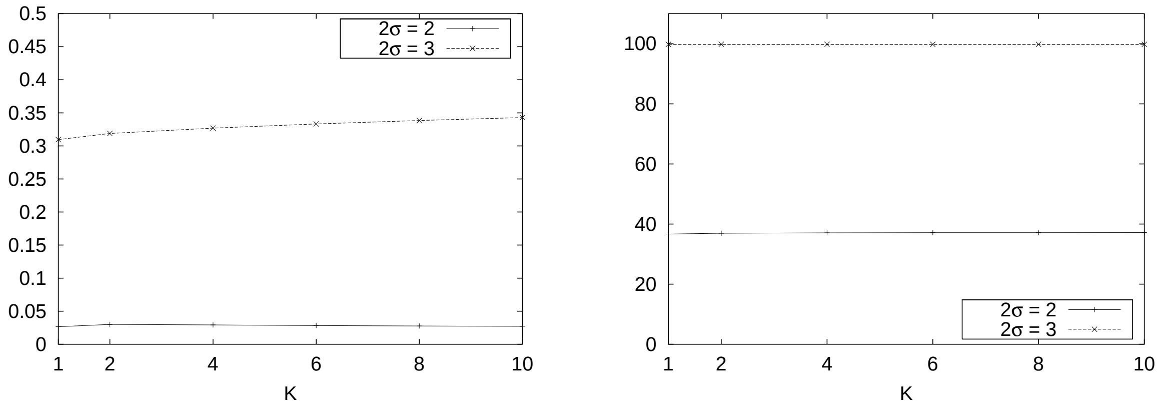


Figure 20: An experiment with the approximate k-nearest neighbor algorithm in dimension 30. In (a) the average error is showed. (b) depicts the gain percentage in the number of distance computations.

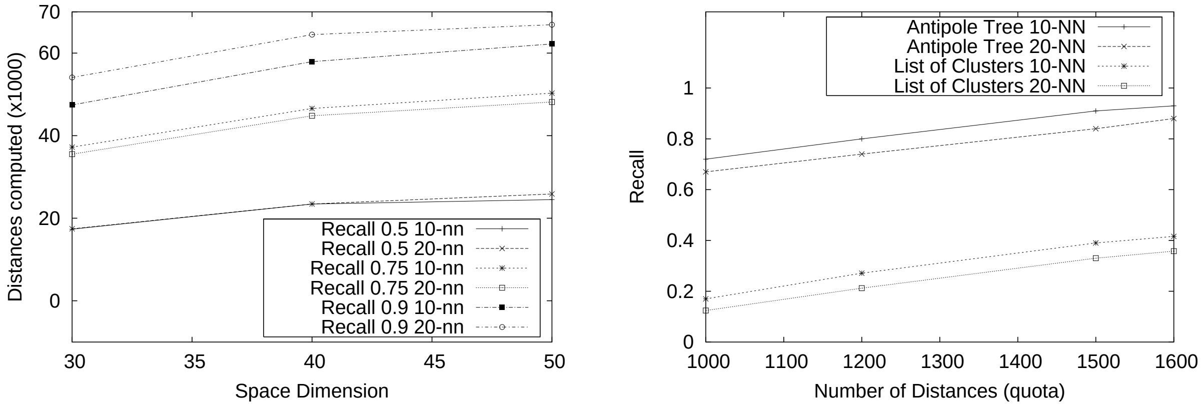


Figure 21: (left) Analysis of curse of dimensionality using Antipole Tree from dimension 30 to 50. Number of distances needed fixing the recall. (right) Comparisons using the image histogram database between the Antipole Tree and List of Clusters w.r.t. approximated k-nearest neighbor. The recall varying the quota is depicted.

database, also it illustrates (right) 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

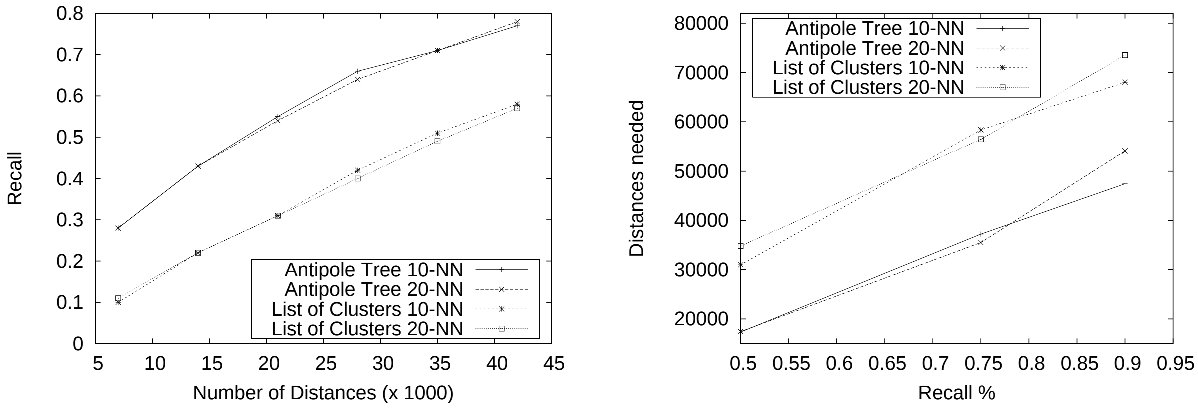


Figure 22: Comparing Antipole Tree and List of Clusters w.r.t. approximated k-nearest neighbor. In (a) the Recall varying the quota is depicted. In (b) the number of distance computations with fixed recall are showed.

database [50]. Starting from 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 pseudo-diameter (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

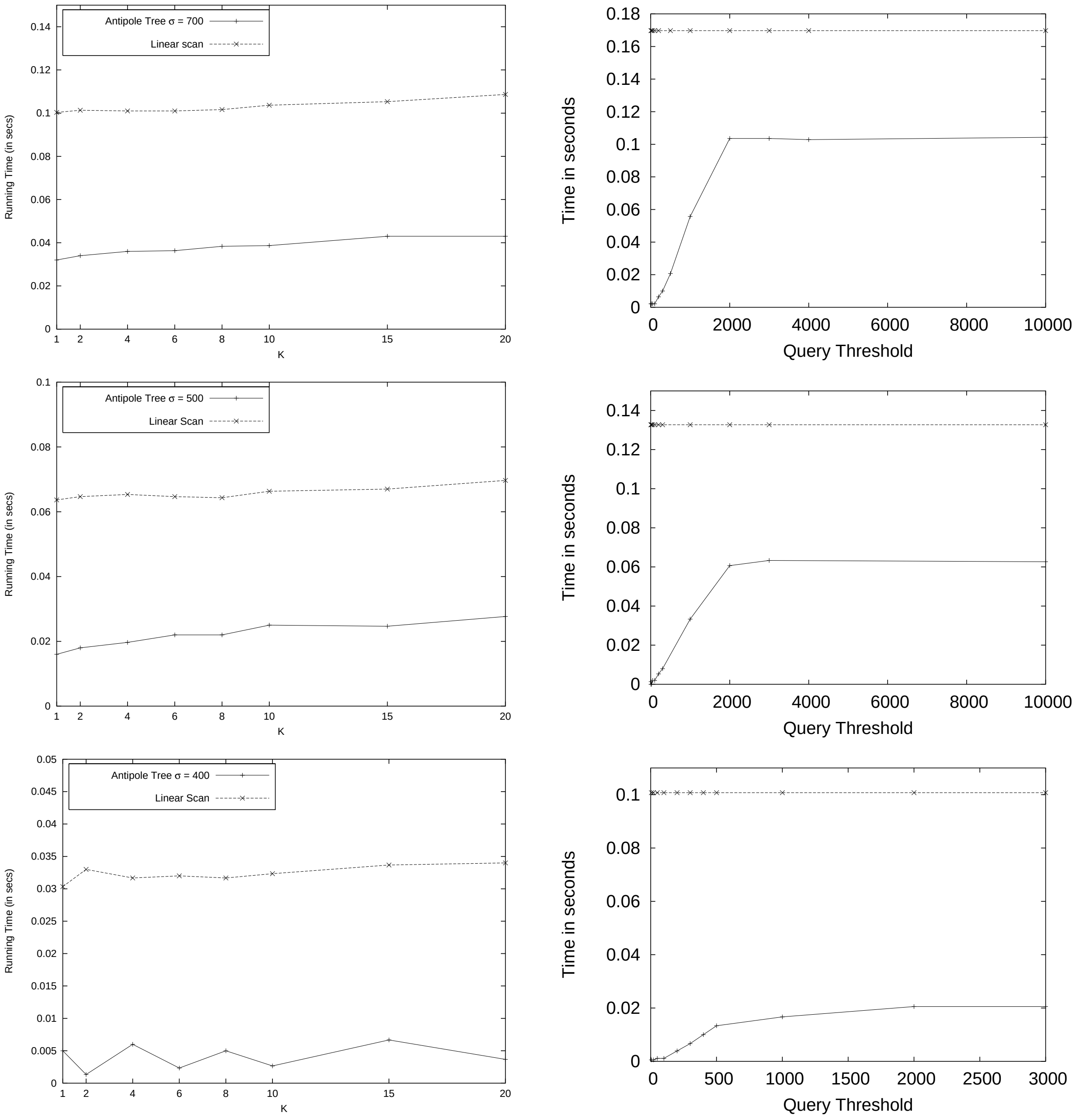


Figure 23: Comparing Antipole Tree and linear scan w.r.t. k-nearest neighbor (left side) and range search (right side) in $\mathbb{R}^{267}$ top, $\mathbb{R}^{147}$ middle, and $\mathbb{R}^{63}$ bottom.

ratio given by a very efficient algorithm for diameter computation in Euclidean spaces together with the intrinsic dimension of the given metric space (Appendix A).

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.

Acknowledgements

We 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," *Proceedings of ACM SIGMOD*, pp. 94–105, 1998.

[4] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu, "Proximity matching using fixed-queries trees," in *Proceedings of Combinatorial Pattern Matching, 5th Annual Symposium*, ser. Lecture Notes in Computer Sciences, vol. 807. Springer-Verlag, 1994, pp. 198–212.

[5] G. Barequet and S. Har-Peled, "Efficiently approximating the minimum-volume bounding box of a point set in three dimensions," *Proceedings of the 10th Annual ACM-SIAM Symposium on 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," in *Proceedings of the 4th Italian Conference on Algorithms and Complexity*, ser. Lecture Notes in Computer Sciences, vol. 1767. Springer-Verlag, 2000, pp. 226–238.

[7] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree: An index structure for high-dimensional data," *Proceedings of the 22Nd International Conference on Very Large Databases*, pp. 28–39, 1996.

[8] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is Nearest Neighbor meaningful?" *Proceedings of the 7Th Intl. Conference on Database Theory*, vol. 1540, pp. 217–235, 1999.

[9] T. Bozkaya and M. Ozsoyoglu, "Indexing large metric spaces for similarity search queries," *ACM Transaction on Database Systems*, vol. 24, no. 3, pp. 361–404, 1999.

[10] S. Brin, "Near neighbor search in large metric spaces," *Proceedings of the 21st International Conference on Very Large Data Bases*, pp. 574–584, 1995.

[11] W. A. Burkhard and R. M. Keller, “Some approaches to best-match file searching,” *Communications of the ACM*, vol. 16, no. 4, pp. 230–236, 1973.

[12] B. Bustos and G. Navarro, “Probabilistic proximity searching algorithms based on compact partitions,” *Proceedings of SPIRE*, pp. 284–297, 2002.

[13] I. Calantari and G. McDonald, “A data structure and an algorithm for the nearest point problem,” *IEEE Transaction on Software Engineering*, 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 Journal on Optimization*, 2004, to appear.

[15] T. M. Chan, “Approximating the diameter, width, smallest enclosing cylinder, and minimum-width annulus,” *International Journal of Computational Geometry and Applications*, vol. 12, no. 1-2, pp. 67–85, 2002.

[16] E. Chávez and G. Navarro, “An effective clustering algorithm to index high dimensional metric spaces,” *Proceedings of SPIRE*, pp. 75–86, 2000.

[17] E. Chávez and G. Navarro, “A probabilistic spell for the curse of dimensionality,” in *Proceedings of 3rd Workshop on Algorithm Engineering and Experimentation (ALENEX’01)*, ser. Lecture Notes in Computer Sciences, vol. 2153. Springer-Verlag, 2001, pp. 147–160.

[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,” *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pp. 75–94, May 1994.

[20] P. Ciaccia and M. Patella, “Bulk loading the M-tree,” *Proceedings of the 9Th Australasian Database Conference (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,” *Proceedings of the IEEE 16th International Conference on Data Engineering*, pp. 244–255, 2000.

[22] P. Ciaccia, M. Patella, and P. Zezula, “M-tree: An efficient access method for similarity search in metric spaces,” *Proceedings of the 23th International Conference on Very Large Data Bases*, pp. 426–435, 1997.

[23] K. Clarkson, “Nearest neighbor queries in metric spaces,” *Proceedings of the 29th Annual ACM Symposium on 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,” *Proceedings of the 2Nd International Conference on Knowledge Discovery in Databases and Data Mining*, pp. 226–231, 1996.

[25] T. Feder and D. Greene, “Optimal algorithms for approximate clustering,” *Proceedings of the 20th Annual ACM Symposium on 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 Journal*, 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,” *Proceedings of the IEEE 15th International Conference on Data Engineering*, 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," *Proceedings of the 25th International Conference on 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," *Proceedings of ACM SIGMOD*, pp. 73–84, 1998.

[33] S. Har-Peled, "A practical approach for computing the diameter of a point set," *Proceedings of the 17th Symposium on Computational Geometry*, pp. 177–186, 2001.

[34] G. R. Hjaltason and H. Samet, "Distance browsing in spatial database," *ACM Transaction on 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," *Journal of the ACM*, vol. 32, no. 1, pp. 130–136, 1985.

[36] P. Indyk, "Sublinear time algorithms for metric space problems," *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pp. 428–434, 1999.

[37] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," *Proceedings of the 30th Annual ACM Symposium on 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 Transactions on Knowledge and Data Engineering*, vol. 14, no. 4, pp. 792–808, July-August 2002.

[39] T. M. Mitchell, *Machine Learning.* McGraw-Hill, 1997.

[40] G. Navarro, "Searching in metric spaces by spatial approximation," *The VLDB Journal*, vol. 11, pp. 28–46, 2002.

[41] R. Ng and J. Han, "Clarans: a method for clustering objects for spatial data mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1003–1016, 2002.

[42] H. Noltemeier, K. Verbarg, and C. Zirkelbach, "Monotonous bisector* trees - a tool for efficient partitioning of complex scenes of geometric objects," in *Data Structure and Efficient Algorithms*, ser. Lecture Notes in Computer Sciences, vol. 594. Springer-Verlag, 1992, pp. 186–203.

[43] C. Procopiuc, "Geometric techniques for clustering theory and practice," Ph.D. dissertation, Duke University, 2001.

[44] M. Shapiro, "The choice of reference points in best-match file searching," *commACM*, vol. 20, no. 5, pp. 339–343, 1997.

[45] D. Shasha and T.-L. Wang, "New techniques for best-match retrieval," *ACM Transaction on 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 Journal*, vol. 8, no. 3-4, pp. 289–304, 2000.

[47] S. Sumanasekara and M. Ramakrishna, “CHILMA: An efficient high dimensional indexing structure for image databases,” *Proceedings of the First IEEE Pacific-Rim Conference on 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,” *Proceedings of the 7th International Conference on 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.

[51] L. Wei and M. Levoy, “Texture synthesis over arbitrary manifold surfaces,” *Proceedings of ACM-SIGGRAPH 2001*, pp. 355–360, 2001.

[52] P. Yianilos, “Data structures and algorithms for nearest neighbor search in general metric spaces,” *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 311–321, January 1993.

[53] T. Zhang, R. Ramakrishnan, and M. Livny, “BIRCH: An efficient data clustering method for very large databases,” *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 103–114, 1996.

APPENDIX

A Diameter computation on Euclidean spaces

In this appendix we describe an approximation algorithm for the diameter computation on the Euclidean plane. Several studies in the literature [1, 5, 33, 15] have provided efficient algorithms for the approximate diameter computation in multidimensional Euclidean Spaces. Our approach can be regarded as the binary search version of [1]. For the sake of simplicity, we will start with a finite set of points in the plane S. We perform the Antipole search as follows. Let (P_{X_m}, P_{X_M}), (P_{Y_m}, P_{Y_M}) be four points of S having minimum and maximum Cartesian coordinates: the so called minimum area bounding box *BBox.* Notice that such four points belong to the convex hull of the set S and all of S is included in the rectangle bounded by $(P_{X_M}.x - P_{X_m}.x)$ and $(P_{Y_M}.y - P_{Y_m}.y)$ The two endpoints (A, B) of the diameter of such four points constitute our Antipole pair. The Antipole distance (the pseudo-diameter) is not less than $Diagonal/\sqrt{2}$. This yields $\frac{Antipole}{Diameter} \geq \frac{1}{\sqrt{2}}$ proving that our approximation ratio in the plane is $1 - 1/\sqrt{2}$.

Now we describe a generalization of this method giving us an approximation algorithm able to obtain an exponentially arbitrary low approximation factor δ for the

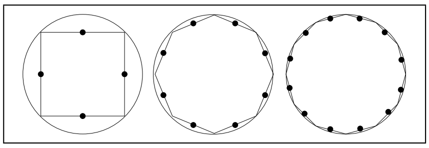


Figure 24: These pictures show the worst cases in the first three iterations of the algorithm.

```
DIAGONAL($S, \delta$)
Let $BBox = \{P_{X_m}, P_{X_M}, P_{Y_m}, P_{Y_M}\}$ be the minimum bounding box of $S$;
$V \leftarrow \{\{S\}\}$;
for $i = 1$ to $\lceil \frac{\pi}{4 \times \arccos(1-\delta)} - 1 \rceil$ do
    $V' = \text{ROTATE\_SET}\left(V, \frac{\pi}{2^{i+1}}\right)$;
    Let $BBox_{\frac{\pi}{2^{i+1}}} = \{P_{X_m}, P_{X_M}, P_{Y_m}, P_{Y_M}\}$
    be the minimum bounding box of the rotated sets in $V'$;
    $V =$ Set catalog of $V'$;
    $BBox = BBox \cup BBox_i$;
end for
return FIND_ANTIPOLE($BBox$);
```

Figure 25: Algorithm for the Pseudo-Diameter Computation.

real diameter. We perform a $\pi/4$ rotation of our Cartesian coordinates, which implies a bisection of the axes, and compute the maximum and minimum coordinate points for such two new axes. We obtain 8 points. Let A, B be the diameter of this set. It is easy to see (middle picture in Fig. 24) that $dist(A,B)/\cos\frac{\pi}{8} > diameter(S)$. By iterating the bisecting process d times, we get $dist(A,B)/\cos\frac{\pi}{2^{d+2}} > diameter(S)$ (see the pseudocode in Fig. 25). Therefore the error introduced by the algorithm is:

$$\delta = \frac{|Diameter - Pseudo\_Diameter|}{Diameter} \leq \left|1 - \cos\frac{\pi}{2^{d+2}}\right| .$$

Hence, we can conclude that:

Theorem A.1 *Let S be a set of points in the plane and let $0 < \delta \leq 1 - \sqrt{2}/2$. Then a call to* DIAGONAL(S, δ) *returns an Antipole pair A, B (say Pseudo-Diameter) which approximates the diameter with an error bounded by δ.* ■

Experiments in n-dimensional Euclidean spaces show that the Antipole pair distance is fully comparable with the first iteration of the above Pseudo_Diameter algorithm, for tournaments with subset size $t = n + 1$. This suggests that one could calculate the intrinsic dimension n of a metric space S and then use tournaments of size $t = n + 1$.