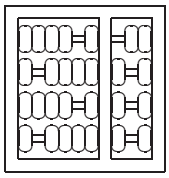


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Searching in High-Dimensional Metric Spaces using BP-Trees

Jurandy Almeida *Neucimar J. Leite*
Ricardo da S. Torres

Technical Report - IC-11-08 - Relatório Técnico

March - 2011 - Março

Searching in High-Dimensional Metric Spaces using BP-Trees

Jurandy Almeida Neucimar J. Leite Ricardo da S. Torres

Institute of Computing, University of Campinas – UNICAMP
13083-852, Campinas, SP – Brazil

{jurandy.almeida,neucimar,rtorres}@ic.unicamp.br

March 24, 2011

Abstract

Similarity search in high-dimensional metric spaces is a key operation in many applications, such as multimedia databases, image retrieval, object recognition, and others. The high dimensionality of the data requires special index structures to facilitate the search. A problem regarding the creation of suitable index structures for high-dimensional data is the relationship between the geometry of the data and the organization of an index structure. Most of existing indexes are constructed by partitioning the dataset using distance-based criteria. However, those methods either produce disjoint partitions, but ignore the distribution properties of the data; or produce non-disjoint groups, which greatly affect the search performance. In this paper, we study the performance of a new index structure, called Ball-and-Plane tree (BP-tree), which overcomes the above disadvantages. BP-tree is constructed by recursively dividing the dataset into compact clusters. Different from other techniques, it integrates the advantages of both disjoint and non-disjoint paradigms in order to achieve a structure of tight and low overlapping clusters, yielding significantly improved performance. Results obtained from an extensive experimental evaluation with real-world datasets show that BP-tree consistently outperforms the state-of-the-art solutions. In addition, BP-tree scales up well, exhibiting sublinear performance with growing number of objects in the database.

1 Introduction

Similarity search in high-dimensional metric spaces is a subject of interest for many research communities. For over two decades, significant have been spent trying to improve the performance in processing similarity queries. In spite of that, many issues are still considered open problems. A problem regarding indexes for search in high-dimensional spaces is the relationship between the data geometry and index organization.

Most of existing indexes employed to accelerate data retrieval are constructed by partitioning a set of objects using distance-based criteria. In order to keep the balance of the structure, the dataset is divided into even-sized parts, ignoring the inherent grouping of data. In general, those techniques can be divided into two different categories. One type of methods produces disjoint partitions, but ignores the distribution properties of the

data [3, 5, 8, 22, 29, 31]. The other type of methods produces non-disjoint groups, which greatly affect the search performance [6, 10, 28, 30].

In this paper, we study the performance of a new index structure, called Ball-and-Plane tree (BP-tree), which is constructed by dividing the dataset into compact clusters. It combines the advantages of both disjoint and non-disjoint paradigms in order to achieve a structure of tight and low overlapping clusters, yielding significantly improved performance. Those properties of BP-tree are supported by an extensive experimental evaluation performed over real-world datasets. The reported comparative test results demonstrate that our approach consistently outperforms the state-of-the-art solutions. Moreover, BP-tree is scalable, exhibiting sublinear behavior regarding the number of indexed objects.

The major contributions of this paper are:

- First, we propose BP-tree, a new index structure which performs very well for similarity search in high-dimensional metric spaces.
- Second, we study how the relationship between the geometry of the data and the organization of an index structure may greatly affect the search performance.
- Third, we evaluate the performance of BP-tree through a comparison with several state-of-the-art indexes for similarity search in metric spaces.
- Finally, we analyze the scalability of BP-tree using a rigorous experimental design. Our results show that BP-tree presents a sublinear behavior, which makes it well-suited for very large datasets.

A preliminary version of this work was presented as a poster at the ACM International Conference on Information and Knowledge Management (CIKM 2010) [1]. Here, we introduce several innovations. First, we present an in-depth review of the state-of-the-art indexing for metric spaces. Additionally, we also discuss about the interaction between geometric and structural constraints, showing how the proposed method reduces the effects imposed by such a liability. Finally, we significantly extend the experimental evaluation of our technique, including a carefully statistical analysis of its scalability.

The remainder of this paper is organized as follows. Section 2 introduces some basic concepts of the similarity search problems. Section 3 describes related work. Section 4 presents BP-tree and shows how to apply it to similarity search. Section 5 reports the results of our experiments and compares our approach with other methods. Finally, we offer our conclusions and directions for future work in Section 6.

2 Basic Concepts

Traditional database systems [12, 24] are able to efficiently deal with structured records by using the *exact match* paradigm. However, complex data types, such as multimedia data (audio, image, and video), biological data (genomic and protein sequences), among others, cannot be represented effectively as structured records [32].

In those cases, *similarity search* [17] has been established as a fundamental paradigm. Essentially, the problem is to find, in a set of objects, those which are the most similar

to a given query object. The similarity between any pair of objects is computed by some distance function, being understood that low values of distance correspond to high degrees of similarity [32].

The commonest types of similarity queries include (1) *range* queries, where all the objects whose distance to the query does not exceed a threshold are requested; and (2) *k-nearest neighbors* (k-NN) queries, where a specified number k of objects, which are closest to the query are requested [32].

Several index structures have been proposed to speed up similarity queries [1,2,9,13,25]. They can be broadly classified, depending on their field of applicability, as *multi-dimensional* (or spatial) and *metric access methods*, where the former only apply when the feature space is a vector space [32].

Algorithms to search in general metric spaces can be divided into two large areas: pivot-based and clustering-based methods [9]. A pivot-based strategy selects some objects as *pivots* from the collection and then computes and stores the distances between the pivots and the objects of the database. During the search, those distances are used to discard objects without comparing them with the query. Clustering techniques consist in dividing the space into zones as compact as possible, normally in a recursive fashion, and storing a *representative* ("center") for each zone plus a few extra data that allows us to quickly discard the zone at query time. In a search, complete regions are discarded by using the distances from their representatives to the query [9].

Two criteria can be used to delimit a zone in the clustering-based approaches. The first one selects a set of representatives and put each other object inside the zone of its closest representative, thus the areas are limited by *hyperplanes*. The second criterion is the *covering radius*, which is the maximum distance between a representative and any object in its zone [9].

3 Related Work

The problem of supporting nearest neighbor and range queries in metric spaces has recently attracted the attention of researchers. An excellent survey of metric access methods can be found in [9].

The pioneering work of Burkhard and Keller [7] provided two interesting techniques for partitioning a metric dataset in a recursive fashion. Their first approach partitions a dataset by choosing a representative from the set and grouping the objects with respect to their distance to the representative. The second approach divides the original set into a fixed number of groups and chooses a representative from each of the groups.

The metric tree of Uhlmann [29] and the Vantage-point tree (VP-tree) [31] are somewhat similar to the first technique of [7] as they divide the dataset into disjoint partitions according to a representative, called a "vantage point". In order to reduce the number of distance calculations to answer similarity queries using the VP-tree, Baeza-Yates et al. [3] suggested to use the same vantage point in all partitions that belong to the same level. Then, a binary tree degenerates into a simple list of vantage points. Bozkaya and Özsoyogly [5] proposed an extension of the VP-tree called the Multi-Vantage-Point tree (MVP-tree), which

carefully chooses m vantage points for each level of the tree. The Generalized Hyperplane tree (GH-tree) [29] is another method that recursively divides the dataset into disjoint partitions by selecting objects as representatives and assigning the remaining ones to the closest representative.

The Geometric Near-Neighbor Access tree (GNAT) [6] can be viewed as a refinement of the second technique presented in [7]. It partitions the space into so-called Dirichlet domains, which is considered a generalization of the Voronoi-like partitioning. In addition to the representative and the maximum distance, it stores the distances between pairs of representatives. Those distances can be used to prune the search space using the triangle inequality.

The Spatial Approximation tree (SAT) [22] is also based on Voronoi diagrams, but in contrast to GNAT it tries to approximate the structure of the Delaunay graph. The authors proved that the ideal structure is impossible to build on general metric spaces. Therefore, SAT is a simplification which forces some backtracking in the tree.

List of Clusters (LC) [8] partitions the space into clusters according to the proximity for choosing representatives, storing also the covering radius of each cluster. The authors showed that their approach is especially well suited to search in high-dimensional spaces, where it outperformed by far several prominent alternative metric indexes.

The Metric tree (M-tree) [10] is a height-balanced tree also based on the second technique presented in [7], which stores the data in the leaves and builds an appropriate cluster hierarchy on top, allowing for dynamic operations. Traina Jr. et al. [28] proposed an extension of the M-tree, named Slim-tree. They introduced three new features: (1) a node-splitting strategy based on the MST (minimum spanning tree) algorithm, (2) an insertion policy based on the node occupancy, and (3) a post-processing algorithm to reduce the overlapping volumes in the tree, called Slim-down. Vieira et al. [30] suggested to relax the height balance constraint by keeping a trade-off between breadth-searching and depth-searching in order to reduce the overlapping between nodes in high-density regions, improving the search performance in those regions.

The use of multiple representatives was proposed by Santos Filho et al. [26]. Their approach chooses a number of objects from the dataset as global representatives, called "omni-foci". The distances of all other objects to each focus are used for producing a coordinate system. Each coordinate can be indexed using any multi-dimensional method, or even sequential scanning, generating a family of metric access methods named the "Omni-family".

The iDistance [18] is an efficient index for performing k-NN queries in high-dimensional metric spaces. It partitions the space into clusters and selects a representative from each one. Every object is assigned to a one-dimensional key based on the distance to the representative of its cluster. Each key is indexed using a B^+-tree [12, 24], where all subsequent operations are performed. The authors showed that their approach is superior to M-tree and Omni-family. However, the query performance is significantly affected if the choice of partitioning scheme and representative objects is not appropriate.

In general, all the previous works follow two basic paradigms: disjoint or non-disjoint. The former partitions the dataset into disjoint clusters, but ignores the distribution properties of the data [3, 5, 8, 18, 22, 29, 31]. The latter produces non-disjoint groups, which greatly

affect the search performance [6,10,26,28,30]. In order to keep the balance of the structure, they divide the dataset into even-sized parts, ignoring the inherent grouping of data.

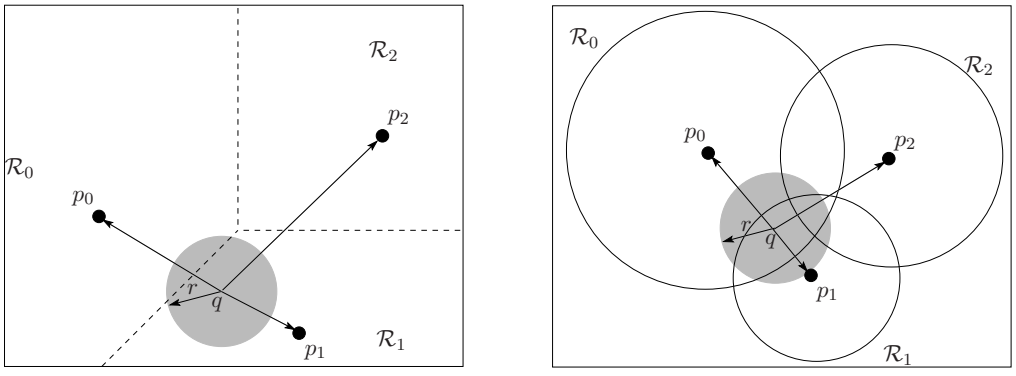
Different from all of the previous techniques, BP-tree does not divide the dataset into disjoint or non-disjoint groups. Instead, it is an index structure that combines the advantages of both strategies. Moreover, BP-tree splits the dataset into compact clusters by taking into account their original distribution.

4 The BP-Tree

Traditional metric access methods divide a dataset into regions and choose objects called representatives to represent each region. This process is usually performed in a recursive fashion, which enables the structure to be organized hierarchically, resulting in a tree. In general, the nodes of that tree contain information about the representatives, the objects in the covered region, and their distances to the representatives. Each node is generally stored in disk using fixed size records. It is essential to warrant data persistence and to allow handling any number of objects. Since disk accesses are slow, it is important to minimize the number of disk accesses (I/O cost) required to answer queries.

When a query is performed, the query object is first compared with the representatives of the root node. The triangle inequality is then used to prune subtrees, avoiding distance calculations between the query object and objects in the pruned subtrees. Distance calculations between complex data types can have a high computational cost. Therefore, for achieving good performance in metric access methods, it is also vital to minimize the number of distance calculations (CPU cost) in query operations.

If the objects in the dataset are uniformly distributed, there exists a high probability of the query region intersects the covered region of several nodes, which greatly affect the search performance. In order to clarify the above situation, look at Figure 1, in which a range query (gray region) using the query object q and a range r is posed. The dashed lines (a) and the solid circles (b) bound three regions $\mathcal{R}_0$, $\mathcal{R}_1$, and $\mathcal{R}_2$. The representatives of each region is denoted by the points p_0, p_1, and p_2, respectively.



(a) Disjoint partitions. (b) Non-disjoint groups.

Figure 1: Effects of the partitioning paradigms in a dataset with an uniform distribution.

Figure 1(a) illustrates the effects of the use of disjoint partitions. In this case, the query region intersects both the regions $\mathcal{R}_0$ and $\mathcal{R}_1$, thus two nodes must be accessed in order to answer the query. Notice that the node covering the region $\mathcal{R}_0$ must be analyzed even if no qualified objects could be found. However, if the query range r is small enough, the whole query can be found in a single node.

We show the effects of the use of non-disjoint groups in Figure 1(b). In this figure, the region of the query response is covered by all the three regions $\mathcal{R}_0$, $\mathcal{R}_1$, and $\mathcal{R}_2$ and, hence, all the three nodes must be accessed. In addition, it is noteworthy that the algorithm must analyze several nodes no matter how small is the radius r.

When the objects in the dataset are sparsely distributed, both disjoint and non-disjoint paradigms may produce tight clusters. Nevertheless, the use of even-sized sets to maintain the balance of those structures may divide inherent grouping of data, as illustrated in Figure 2.

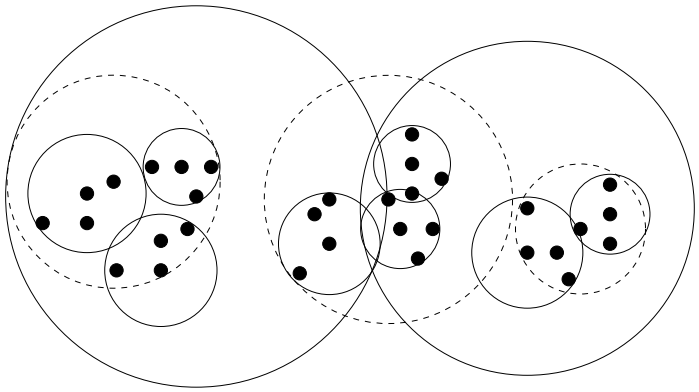


Figure 2: Effects of the use of even-sized sets in a dataset with a sparse distribution.

In the example, there are three natural groups in the data, bounded by the dashed circles. Due to the balance constraints, each group may be divided into smaller clusters, delimited by solid circles. When it happens, the shortcomings previously discussed may arise and, consequently, the search performance may be greatly affected.

The novelty of BP-tree is to combine the advantages of both disjoint and non-disjoint approaches. In BP-tree, the dataset is divided into compact clusters by respecting their distribution. This strategy overcomes the above disadvantages.

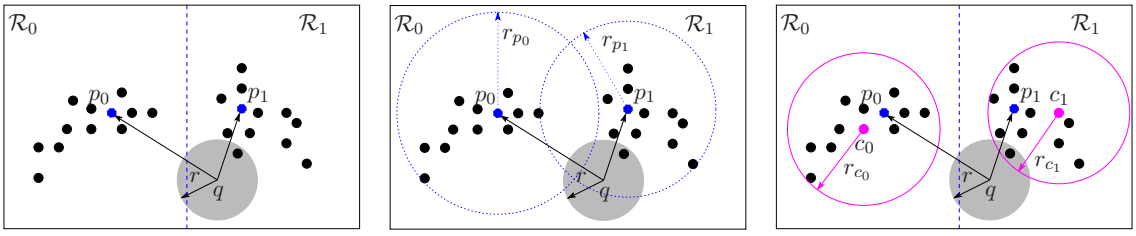
4.1 Overview of BP-tree

Consider an initial set of objects, we first divide the objects into groups based on their global distribution. We can refine this partitioning further by dividing each existing group based on the local distribution of a subset of objects. This process may be repeated by taking a smaller subset at each time until no further improvements are possible. Finally, we have a hierarchical set of groups. This is roughly the basic idea of BP-tree, where such a model is adapted to disk. In other words, given a query object, we can reduce the search space by gradually considering a subset of objects with a more relevant distribution.

At each level, we cluster the data around representatives. Each cluster is a partition in the sense that objects in the same cluster have similar distances to their representative. Thereby, we break the data into subsets by defining hyperplanes between representatives. This procedure captures the data distribution in the sense that far away objects are separated into different clusters. Next, for each subset, we established a bounding region by selecting the reference object that attains the minimum covering radius. The covering radius is the maximum distance between the reference object and any object in its subset. Thereby, we break the data into subsets by defining hyperspheres around reference objects. The introduction of those bounding regions enhances the pruning rate of the search space. After that, we apply this process recursively until each subset of data can no longer be divided. Finally, we have a set of clusters, which is a hierarchical partitioning of the data.

In order to explicitly highlight the novelty of BP-tree, we elaborate further on how BP-tree benefits and combines the advantages of both disjoint and non-disjoint approaches. On the one hand, BP-tree partitions the dataset into clusters based on distances to representatives. There exists a full order on distances to a same representative and, hence, the clusters are disjoint. On the other hand, BP-tree divides the dataset into regions based on distances to the reference object of those clusters. The same object may lie inside intersecting regions, thus the clusters may overlap. This strategy allows us to achieve a structure of tight and low overlapping clusters, improving the performance on similarity queries, especially for high-dimensional spaces.

Figure 3 illustrates how BP-tree and the two types of partitioning paradigms handle the dataset and the query. The disjoint approaches partition the dataset by defining a cut (dashed line) between representatives (Figure 3(a)). In the example, the query region intersects both the partitions $\mathcal{R}_0$ and $\mathcal{R}_1$, thus two nodes must be accessed in order to answer the query. The non-disjoint approaches cluster the dataset around representatives and use a bounding region (dotted circle) to represent each group (Figure 3(b)). Usually, those bounding regions do overlap. In the figure, the region of the query response is covered by two bounding regions and, hence, both the nodes $\mathcal{R}_0$ and $\mathcal{R}_1$ must be accessed. BP-tree partitions the dataset by defining a cut (dashed line) between representatives and, for each partition, it selects the reference object which establishes a bounding region (solid circle) with the minimum covering radius (Figure 3(c)). In this case, triangle inequality is used to prune one subtree ($\mathcal{R}_0$), thus only one node must be accessed ($\mathcal{R}_1$).



(a) Disjoint partitions. (b) Non-disjoint groups. (c) BP-tree.

Figure 3: Example of how BP-tree benefits and combines the advantages of both disjoint and non-disjoint approaches.

At the end, it is noteworthy that BP-tree may be unbalanced. However, for similarity search in high-dimensional spaces, unbalanced trees may provide better performance than balanced trees, as stated by Chávez and Navarro [8]. They have shown that, for similarity search in high-dimensional spaces, the search cost is determined by the pruning rate of the search space, not by the height of the tree. The pruning rate of the search space is directly related to how the dataset is separated. The balanced tree divides the dataset into even-sized parts, ignoring the data distribution. BP-tree divides the dataset by the data distribution, thus it may separate the dataset better than balanced trees. For a more detailed discussion of the benefits of unbalanced trees in similarity search, refer to [8].

4.2 BP-Tree Creation

Overall, BP-tree is an unbalanced tree index generated by the hierarchical partitioning of the dataset. Like other metric trees, the objects of the dataset are stored into fixed size disk pages. Table 1 summarizes the symbols used in this paper.

Table 1: Summary of symbols and definitions

Symbols	Definitions
$d(x, y)$	distance function between objects x and y
C	capacity of a disk page
D	size of a disk page
K	the number of partitions spanned by a set
M	the minimum occupation of a node
N	number of objects in a set
O	a set of objects
P	a set of partitions

BP-tree has two kinds of nodes: leaf nodes and index nodes. Each index node corresponds to a single disk-page and contains a partitioning information. In contrast, each leaf node consists of a list of disk pages and, hence, may have an unlimited capacity. This strategy authorizes the creation of "big leaves" in order to avoid breaking up inherent grouping of data. In this way, we keep a trade-off between sequential and random accesses to disk, allowing for the speedup of the search process.

The objects are stored in both index and leaf nodes. The structure of a leaf node is

$$leafnode\ [\ \text{array of}\ \langle d(o_i, o_{rep}), o_i \rangle\ ],$$

where o_i is an object and $d(o_i, o_{rep})$ is its distance from the representative of this leaf node o_{rep}. The structure of an index node is

$$indexnode\ [\ \text{array of}\ \langle o_i, r(o_i), d(o_i, o_{rep}), r(o_{ref}), d(o_i, o_{ref}), ptr(T(o_i)) \rangle\ ],$$

where o_i keeps the representative of the subtree $T(o_i)$ pointed to by $ptr(T(o_i))$ and $r(o_i)$ is the covering radius of the bounding region defined by o_i. The distance between o_i and

the representative of this node o_{rep} is kept in $d(o_i, o_{rep})$. The distance between the o_i and the reference object o_{ref} that established the bounding region with the minimum covering radius $r(o_{ref})$ is kept in $d(o_i, o_{ref})$. The pointer $ptr(T(o_i))$ points to the root node of the subtree $T(o_i)$ rooted by o_i.

The tree construction is performed in a top-down fashion. In order to clarify this approach, look at Figure 4. At the beginning, the set of objects $O = \{o_1, o_2, \ldots, o_N\}$ is considered to be part of a single partition. Since the page size D is fixed, it holds a predefined maximum number of objects C, such that

$$C = \frac{D}{\arg\max\limits_{o_i \in O} sizeof(o_i)}.$$

So, those objects are first divided into $K \leq C$ disjoint subpartitions $P = \{p_1, p_2, \ldots, p_K\}$ with at least M objects in each. Information about all those subpartitions form the index node of the first level of the tree. For each partition $p_i \in P$, a subset O_{p_i} is created by taking the objects of p_i. To build subsequent levels of the tree, this process is repeated for all of the new subset of objects at each level, creating the hierarchy of index nodes. The process stops when the number of objects in a subset is less than or equals to the page capacity or the number of partitions spanned by a subset is less than 2. Then, the objects in the subset are written to a leaf node on disk.

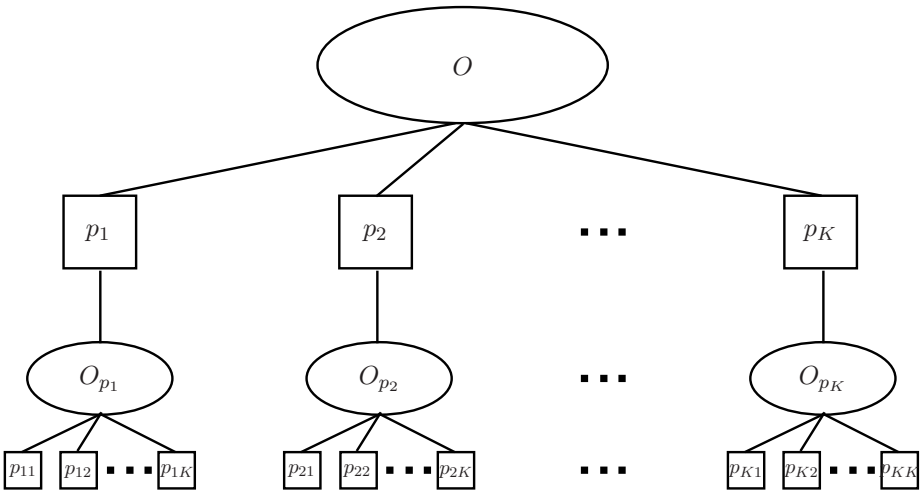


Figure 4: A representation of BP-tree.

Algorithm 1 formalizes the above procedure. It starts by checking the cardinality of the set of objects O (line 3). If it can fit into a disk page, the function CREATE-LEAFNODE is used to create a leaf node (line 4). Otherwise, we call the function SPLIT in order to divide the set into $K \leq C$ partitions with a minimum occupation equals to M (line 6). The function SPLIT can use any partitional clustering method, such as k-medoids [4]. The partitional algorithm is responsible for finding the representatives and the reference objects of each level. The minimum occupation M is set when the structure is created, and this

value must be between one object and at most half of the page capacity C. Next, we check if the set can be divided (line 8). If so, we call the function Initialize-indexnode, which creates an index node using the information about all those partitions (line 11). Thereafter, for each partition, the function Create-subset is used to create a subset of objects (line 13). This function is responsible for selecting the objects of a given partition. After that, we repeat this process for all of the new subset of objects (line 15). Finally, the function Update-indexnode is used to update the information in each entry of the index node (line 16).

Algorithm 1 BP-tree construction.

```
1: function BuildTree(d, D, M, N, O)
2:     C ← D / (arg max_{o_i ∈ O} sizeof(o_i)).
3:     if N ≤ C then
4:         return Create-leafnode(N, O);
5:     else
6:         P ← Split(d, C, M, N, O);
7:         K ← cardinality(P);
8:         if K < 2 then
9:             return Create-leafnode(N, O);
10:        else
11:            Parent ← Initialize-indexnode(K, P);
12:            for all p ∈ P do
13:                O_p ← Create-subset(p, K, P, N, O);
14:                N_p ← cardinality(O_p);
15:                Child ← BuildTree(d, D, M, N_p, O_p);
16:                Update-indexnode(p, Parent, Child);
17:            end for
18:            return Parent
19:        end if
20:    end if
21: end function
```

4.3 Similarity Queries

BP-tree can answer the two commonest types of similarity queries: range query $R(o_q, r(o_q))$, which retrieves all objects found within distance $r(o_q)$ of a query object o_q; and nearest neighbor query $kNN(o_q)$, which retrieves the k nearest neighbors of the query object o_q. In the following subsections, each of those procedures is explained in more detail.

4.3.1 Range Queries

The range search algorithm starts at the root node and traverses the tree in a depth-first

Algorithm 2 Range queries.

```
1: procedure RangeSearch(o_q, r(o_q), d(o_q, o_rep), ptr(T(o_rep)))
2:     if T(o_rep) is not a leaf then
3:         L ← ∅                                   ▷ List of all non-pruned subtrees.
4:         d_min ← ∞
5:         for all ⟨o_i, r(o_i), d(o_i, o_rep), r(o_ref), d(o_i, o_ref), ptr(T(o_i))⟩ ∈ T(o_rep) do
6:             if |d(o_q, o_rep) − d(o_i, o_rep)| − r(o_i) ≤ r(o_q) then
7:                 Calculate d(o_q, o_i)
8:                 if d(o_q, o_i) ≤ r(o_q) then
9:                     Add ⟨d(o_q, o_i), o_i⟩ to the result
10:                end if
11:                if d(o_q, o_i) − r(o_i) ≤ r(o_q) then
12:                    if |d(o_q, o_i) − d(o_i, o_ref)| − r(o_ref) ≤ r(o_q) then
13:                        L ← L ∪ {⟨d(o_q, o_i), ptr(T(o_i))⟩}
14:                        if d_min > d(o_q, o_i) then
15:                            d_min ← d(o_q, o_i)
16:                        end if
17:                    end if
18:                end if
19:            end if
20:        end for
21:        for all ⟨d(o_q, o_i), ptr(T(o_i))⟩ ∈ L do
22:            if (d(o_q, o_i) − d_min)/2 ≤ r(o_q) then
23:                RangeSearch(o_q, r(o_q), d(o_q, o_i), ptr(T(o_i)))
24:            end if
25:        end for
26:    else
27:        for all ⟨d(o_i, o_rep), o_i⟩ ∈ T(o_rep) do
28:            if |d(o_q, o_rep) − d(o_i, o_rep)| ≤ r(o_q) then
29:                Calculate d(o_q, o_i)
30:                if d(o_q, o_i) ≤ r(o_q) then
31:                    Add ⟨d(o_q, o_i), o_i⟩ to the result
32:                end if
33:            end if
34:        end for
35:    end if
36: end procedure
```

manner, visiting all non-discardable subtrees. During the search, all the stored distances are used to prune subtrees.

Algorithm 2 describes the above procedure. If the current node is an index node, we consider all non-empty entries as follows. First, if $|d(o_q, o_{rep}) - d(o_i, o_{rep})| - r(o_i) > r(o_q)$ holds, the subtree pointed to by $ptr(T(o_i))$ is pruned (line 6). Otherwise, we compute the distance $d(o_q, o_i)$ and, if $d(o_q, o_i) \leq r(o_q)$ holds, the object o_i is returned in the query result (line 7 to 10). Afterwards, we prune all subtrees via the criterion $d(o_q, o_i) - r(o_i) > r(o_q)$ (line 11). Next, we use triangle inequality and avoid to access subtrees by checking if $|d(o_q, o_i) - d(o_i, o_{ref})| - r(o_{ref}) > r(o_q)$ (line 12). This pruning criterion is based on the fact that the expression $|d(o_q, o_i) - d(o_i, o_{ref})| - r(o_{ref})$ forms the lower bound on the distances between the query object o_q and any object in the subtree pointed to by $ptr(T(o_i))$ (see Figure 5). All non-pruned subtrees $ptr(T(o_i))$ and their distances $d(o_q, o_i)$ are stored in a list (line 13). Thereafter, we find the entry whose representative o_i is closest to the query object o_q (i.e., $d_{min} \leftarrow \arg\min\ d(o_q, o_i)$) and, then, we remove from the list all entries such that $(d(o_q, o_i) - d_{min})/2 > r(o_q)$ (line 22). Finally, the algorithm proceeds by recursively searching the remaining subtrees in the list (line 23).

Leaf nodes are similarly processed. Each entry is examined using the pruning condition $|d(o_q, o_{rep}) - d(o_i, o_{rep})| > r(o_q)$ (line 28). If it holds, the entry can be safely ignored. Otherwise, the distance $d(o_q, o_i)$ is evaluated and the object o_i is reported if $d(o_q, o_i) \leq r(o_q)$ (line 29 to 32).

The key advantage of BP-tree is to apply the lower bounds designed for both disjoint $((d(o_q, o_i) - d_{min})/2)$ and non-disjoint approaches $(|d(o_q, o_{rep}) - d(o_i, o_{rep})| - r(o_i)$ and $d(o_q, o_i) - r(o_i))$. Note that in all four steps (lines 6, 11, 22, and 28) where pruning criteria hold, we discard some entries without computing distances to the corresponding objects. In this way, the search process is made faster and more efficient. For a formal proof of the pruning rules used above, refer to [32].

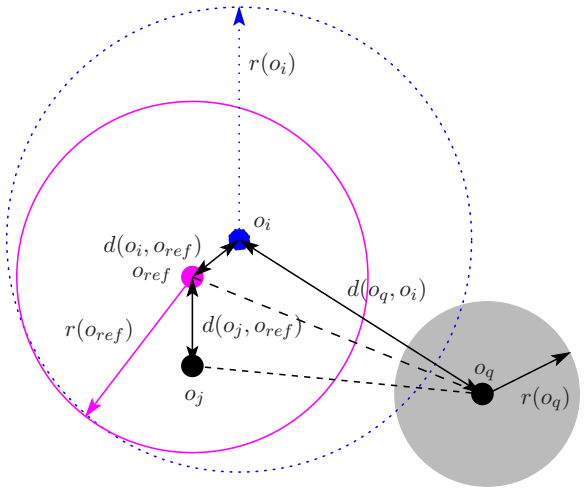


Figure 5: The lower bound on distance $d(o_j, o_q)$ between the query object o_q and any object o_j in subtree $T(o_i)$ rooted by o_i via the reference object o_{ref}.

A different way regarding such a process is to lower bound the distance between o_q and any object o_j in the subtree pointed to by $ptr(T(o_i))$ using the reference object o_{ref}. This situation is demonstrated in Figure 5. By the triangle inequality we have $d(o_j, o_q) \geq |d(o_q, o_{ref}) - d(o_j, o_{ref})|$ and $d(o_q, o_{ref}) \geq |d(o_q, o_i) - d(o_i, o_{ref})|$. Summing up both the inequalities and keeping in mind that $d(o_j, o_{ref}) \leq r(o_{ref})$ (def. of covering radius), we obtain $d(o_j, o_q) \geq |d(o_q, o_i) - d(o_i, o_{ref})| - r(o_{ref})$. Therefore, we compare o_q only against objects that cannot be proved to be far away enough from o_q.

4.3.2 Nearest Neighbor Queries

The algorithm for k nearest neighbors queries is performed by simulating a dynamic range search with the query range r_k being the distance between the query object o_q and the current k-th nearest neighbor. At the beginning, r_k is set to infinity and, during the search process, it is updated (decreased) when a new nearest neighbor is found (∞ if we still have less than k candidates).

In a normal range search with a fixed range r_k, the order in which we backtrack in the tree is unimportant. This is not the case now, as we would like to quickly find objects close to o_q so as to reduce r_k early. A general idea proposed in [29] can be adapted to our index structure. However, its pruning abilities and overall efficiency are considerably improved by BP-tree. This procedure is described in Algorithm 3.

We have a priority queue of nodes, the most promising first. Initially, we insert the root node in the priority queue (line 3). Iteratively, we extract the most promising node (line 5), process it (lines 8 to 27), and insert all non-pruned subtrees in the priority queue (lines 28 to 33). This is repeated until the priority queue becomes empty.

The distance $d(o_q, o_i)$ is used to measure how promising is the subtree $T(o_i)$ and, hence, the subtrees inserted in the priority queue are visited in increasing order of proximity to the query object o_q. Since r_k is reduced along the search, a subtree may seem useful at the moment it is inserted in the priority queue, and becomes useless later when it is extracted from the priority queue to be processed. So, we store the lower bounds $d(o_q, o_i) - r(o_i)$, $|d(o_q, o_i) - d(o_i, o_{ref})| - r(o_{ref})$, and $(d(o_q, o_i) - d_{min})/2$ together the subtree $T(o_i)$ rooted by o_i and its distance $d(o_q, o_i)$. As we extract an entry from the priority queue, we check whether those lower bounds exceed r_k, in which case the extracted node is discarded as it contains no qualified objects.

4.4 Updating the Index

Consider the process of inserting a new object into the BP-tree. The insertion can be done by traversing the hierarchy of index nodes in a depth-first manner. The algorithm follows a path down the tree to locate the most suitable leaf node, descending in each step to the subtree which minimizes the distance from its representative object to the new object. At this point, we insert the new object.

Remember that leaf nodes may have an unlimited capacity. This freedom opens up a number of possibilities that deserve much deeper study, but an immediate consequence is that we can always insert at the leaves of the tree. Hence, the tree is read-only in its top

Algorithm 3 Nearest neighbor queries.

procedure NEIGHBORSEARCH($o_q, k, ptr(T(o_{rep}))$)
  $r_k \leftarrow \infty$
  $Q \leftarrow \{\langle 0, ptr(T(o_{rep})), 0, 0, 0\rangle\}$ ▷ Priority queue of nodes.
  while Q is not empty **do**
    Extract $\langle d(o_q, o_{rep}), ptr(T(o_{rep})), l_1, l_2, l_3\rangle$ from Q
    if the lower bounds l_1, l_2, and l_3 do not exceed r_k **then**
      if $T(o_{rep})$ is not a leaf **then**
        $L \leftarrow \emptyset$ ▷ List of all non-pruned subtrees.
        $d_{min} \leftarrow \infty$
        for all $\langle o_i, r(o_i), d(o_i, o_{rep}), r(o_{ref}), d(o_i, o_{ref}), ptr(T(o_i))\rangle \in T(o_{rep})$ **do**
          if $|d(o_q, o_{rep}) - d(o_i, o_{rep})| - r(o_i) \leq r_k$ **then**
            Calculate $d(o_q, o_i)$
            if $d(o_q, o_i) \leq r_k$ **then**
              Update the response set and r_k by inserting $\langle d(o_q, o_i), o_i\rangle$ and removing the farthest object from o_q
            end if
            $l_1 \leftarrow d(o_q, o_i) - r(o_i)$
            if $l_1 \leq r_k$ **then**
              $l_2 \leftarrow |d(o_q, o_i) - d(o_i, o_{ref})| - r(o_{ref})$
              if $l_2 \leq r_k$ **then**
                $L \leftarrow L \cup \{\langle d(o_q, o_i), ptr(T(o_i)), l_1, l_2\rangle\}$
                if $d_{min} > d(o_q, o_i)$ **then**
                  $d_{min} \leftarrow d(o_q, o_i)$
                end if
              end if
            end if
          end if
        end for
        for all $\langle d(o_q, o_i), ptr(T(o_i)), l_1, l_2\rangle \in L$ **do**
          $l_3 \leftarrow (d(o_q, o_i) - d_{min})/2$
          if $l_3 \leq r_k$ **then**
            Insert $\langle d(o_q, o_i), ptr(T(o_i)), l_1, l_2, l_3\rangle$ into Q
          end if
        end for
      else
        for all $\langle d(o_i, o_{rep}), o_i\rangle \in T(o_{rep})$ **do**
          if $|d(o_q, o_{rep}) - d(o_i, o_{rep})| \leq r_k$ **then**
            Calculate $d(o_q, o_i)$
            if $d(o_q, o_i) \leq r_k$ **then**
              Update the response set and r_k by inserting $\langle d(o_q, o_i), o_i\rangle$ and removing the farthest object from o_q
            end if
          end if
        end for
      end if
    end if
  end while
end procedure

part and changeable only in the bottom.

However, we have to permit the reconstruction of small subtrees in order to avoid them to almost turn into a linked list. Thus, we insert a new object only if the size of the leaf node is small enough. Otherwise, a rebuild of the subtree may be beneficial for the performance. This leads to a trade-off between insertion cost and tree quality.

Deletion can be trivially done except if representative objects are removed. In those cases, the first choice is to keep such objects anyway. Thus, they are just marked as fake objects, without releasing the occupied space.

5 Experimental Evaluation

In this section, we compare the performance and the scalability of our technique with previous work in processing similarity queries.

We implemented BP-tree from scratch in C++ under Linux. In our implementation, the algorithms for partitioning data in the function SPLIT are: **kmedoids**, which uses the well-known PAM (Partitioning Around Medoids) algorithm [4]; and **random**, which selects representatives at random (i.e., the initialization step of the PAM algorithm, which is the most common realization of the k-medoid clustering). The former performs better than the latter, however both of them have a similar behaviour. For the rest of this paper, when we refer to BP-tree, we mean the random procedure, unless otherwise stated. The random strategy forms a baseline on the performance of the proposed method.

BP-tree was compared with MVP-tree [5], SAT [22], List of Clusters [8], M-tree [10], Slim-tree [28], DBM-tree [30], and iDistance [18], which are the most popular approaches for similarity search in metric spaces. For our experimental evaluation, we adopted the implementation of MVP-tree, SAT, and List of Clusters from the Metric Space Library[1] and the implementation of M-tree, Slim-tree, and DBM-tree from the GBDI Arboretum Library[2]. The implementation of iDistance was obtained from the author's home page[3]. In order to guarantee a fair comparison, all the compared methods were configured using their best recommended setup.

The experiments were performed on a machine equipped with a processor Intel Xeon QuadCore X3320 2.5 GHz and 8 GBytes of DDR2-memory. The machine runs Gentoo Linux (2.6.31 kernel) and the ext3 file system.

Our experiments are intended to answer the following questions:

- How does the performance of BP-tree compare to state-of-the-art solutions?
- How good is the scalability of BP-tree compared with previous work?
- How does the construction cost of BP-tree compare to existing indexes?

[1] `http://www.sisap.org/Metric_Space_Library.html`
[2] `http://www.gbdi.icmc.usp.br/arboretum/`
[3] `http://www.cs.mu.oz.au/~rui/code.htm`

5.1 Performance

This section provides experimental results of the performance of our technique using a large assortment of real-world datasets with different properties that directly affect the behavior of an index structure, such as the embedded dimensionality of the data, the size of the dataset, the distribution of the data in the metric space, among others.

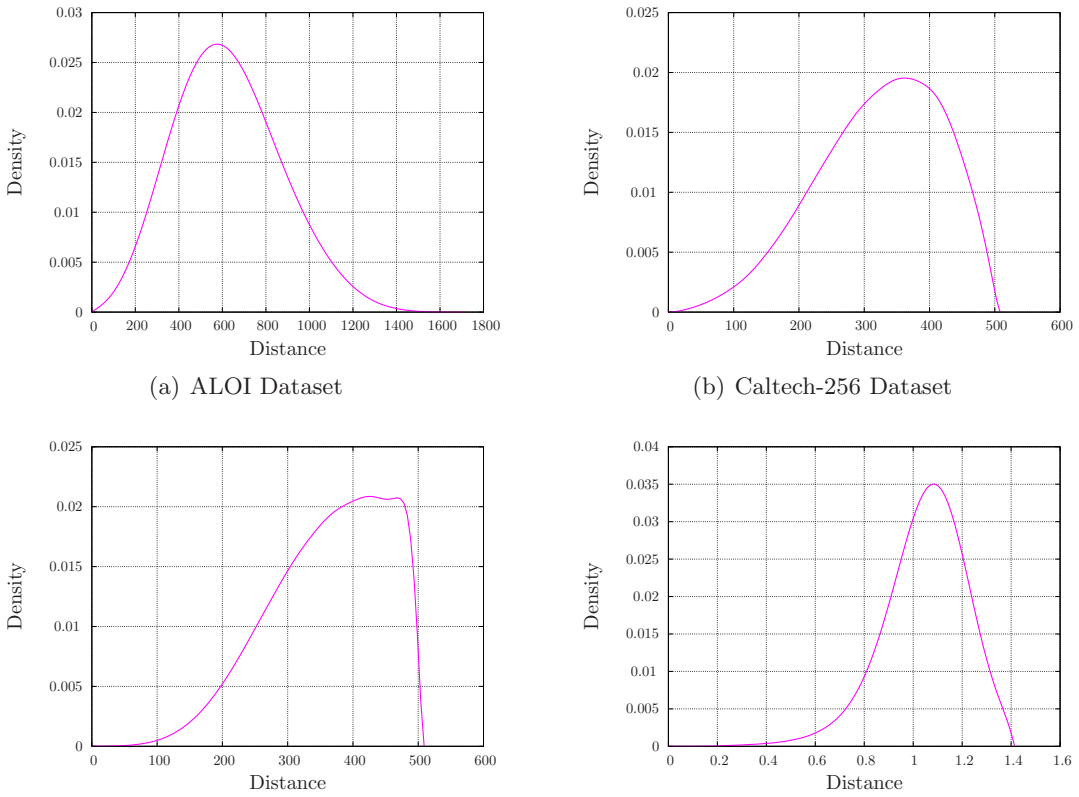
We tested our BP-tree and previous work on four sets of images described in literature and extensively used by the computer vision and image processing communities. Image data has been used for the convenience of obtaining large metric datasets in which a reasonable ground truth can be established. Regardless of that, our indexing scheme itself does not use any property related to the image nature of the data. We are interested in the metric space spanned by feature vectors used to encode image properties. Different feature spaces can be obtained from the same set of images depending on the algorithm employed for generating such vectors and/or the distance function used to measure the dissimilarity between them. Once image databases often produce metric spaces that are very hard to index due to the intrinsic high-dimensionality of the data, they are an excellent test set. Nevertheless, we emphasize that the proposed method can be applied to any type of data, since it is a metric access method and, hence, only the distances are used for indexing. Our image collections are the following:

- ALOI — The Amsterdam Library of Object Images[4] [14] is a collection of 72,000 images. In our experiments, each image is characterized by a 256-dimensional feature vector which represents a Color Correlogram [16]. Each color correlogram is a table indexed by color pairs, where the k-th entry for a pair $< i, j >$ specifies the probability of finding a pixel of color j at a distance k from a pixel of color i in the image. The Manhattan (L_1) distance is used to compare those feature vectors.

- Caltech-256 — The Caltech-256 Object Category Data Set[5] [15] is a set of 30,607 images. We converted each image to a 64-dimensional feature vector by computing a Color Histogram [27]. The color histograms were extracted as follows: the RGB space is divided into 64 subspaces (colors), using four ranges in each color channel. The value for each dimension of a feature vector is the density of each color in the entire image. The distance function used to compare the feature vectors is the Manhattan (L_1) distance.

- Corel — The Corel Image Features[6] [23] is a set of 68,040 image features extracted from the Corel Gallery. In our experiments, each image feature corresponds to a 32-dimensional feature vector which represents a Color Histogram [27]. The color histograms were extracted as follows: the HSV space is divided into 32 subspaces (colors), using eight ranges of H and four range of S. The value for each dimension of a feature vector is the density of each color in the entire image. The Manhattan (L_1) distance is used to compare those feature vectors.

[4] http://staff.science.uva.nl/~aloi/
[5] http://www.vision.caltech.edu/Image_Datasets/Caltech256/
[6] http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html



(a) ALOI Dataset (b) Caltech-256 Dataset

(c) Corel Dataset (d) MNIST Dataset

Figure 6: Distance density functions of the dataset used in the experiments.

- MNIST — The MNIST database[7] [19] contains 60,000 images of handwritten digits. We converted each image to a 128-dimensional feature vector by computing a Histogram of Oriented Gradients [11]. The histogram of oriented gradients was extracted as follows: the image is decomposed into 4×4 equally-sized cells which are individually described by using a orientation histogram of the gradient magnitudes. Each orientation histogram has 8 bins covering 360 degree range of orientations. The feature vector is formed from a vector containing the values of all the orientation histogram entries and normalized to unit length. The distance function used to compare the feature vectors is the Euclidean (L_2) distance.

Table 2 summarizes the most important characteristics of our datasets, which include the total number of objects (N), the embedding dimensionality (E), the distance function used to compare objects ($d(x, y)$), the page size in KBytes (D), and the average capacity of each disk page (C).

Figure 6 shows the distance density functions of each database. Observe the differences in densities of the individual data collections. It is worth noting that these databases are

[7] `http://yann.lecun.com/exdb/mnist/`

characterized by different distance distributions.

Table 2: The most important characteristics of the datasets used in the experiments.

Dataset	N	E	$d(x, y)$	D	C
ALOI	72,000	256	L_1	16	64
Caltech-256	30,607	64	L_1	8	64
Corel	68,040	32	L_1	4	64
MNIST	60,000	128	L_2	32	64

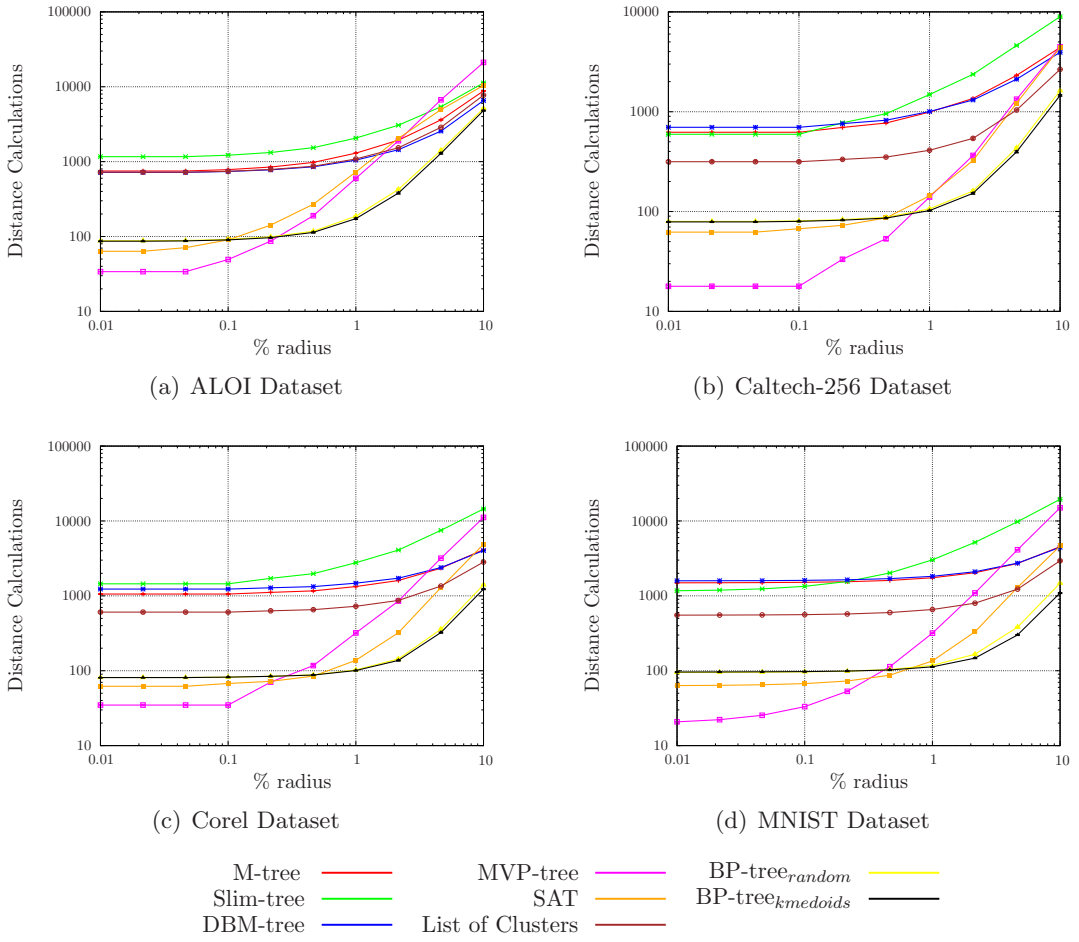


Figure 7: The query performance (given by the average number of distance calculations) for the compared approaches as a function of the query radius, where each of the plots refer to one of our datasets.

For each collection, we randomly selected about five percent of the total number of objects to be used as queries. Five replications were performed for each database in order to ensure statistically sound results. Thereafter, we performed range queries and k-NN

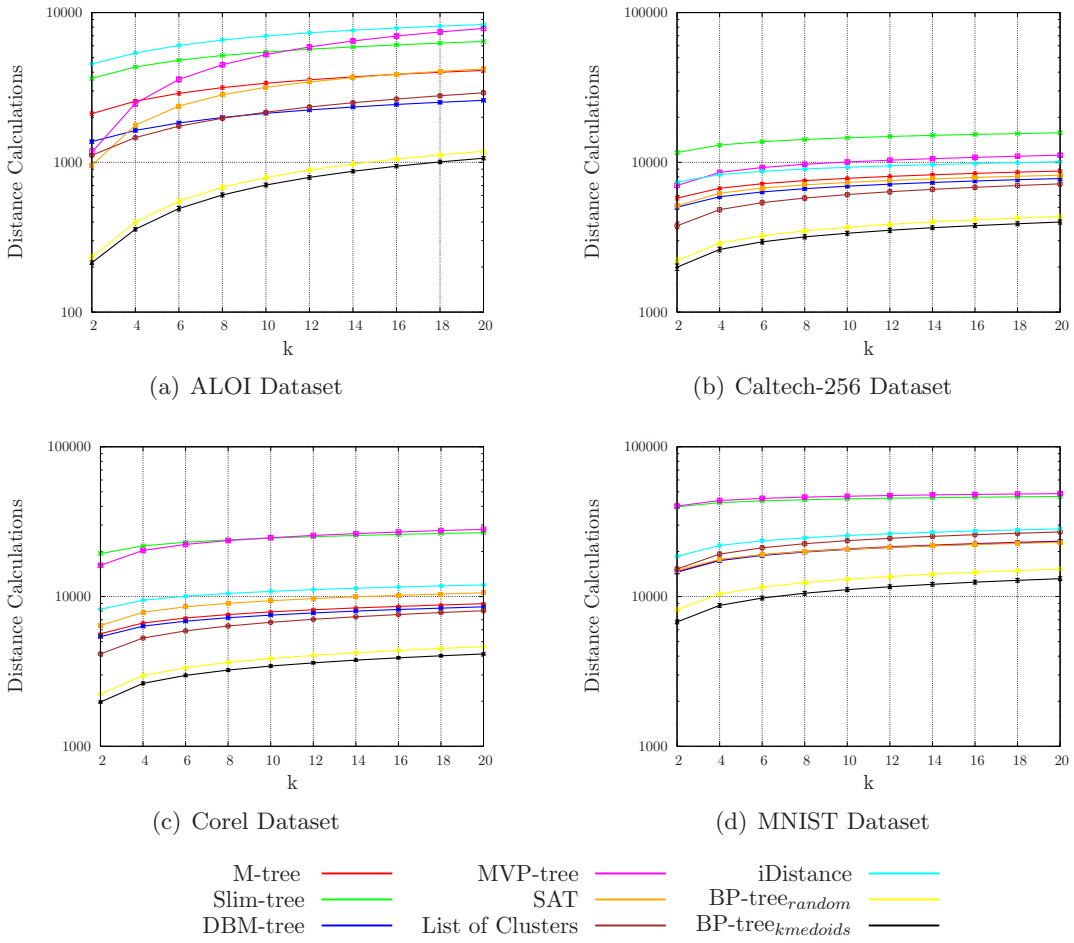


Figure 8: The query performance (given by the average number of distance calculations) for the compared approaches as a function of the number k of nearest neighbors requested for the query, where each of the plots refer to one of our datasets.

queries. The search radius was varied from 0.01% to 10% of the largest distance between pairs of objects in the dataset, because they are the most meaningful range of radii requested when performing similarity queries. The number k of nearest neighbors requested for the query was varied from 2 to 20.

The measurements taken at each experiment were the average number of distance calculations, the average number of disk accesses, and the average time (in milliseconds) required for performing a query. For each dataset, the 99% confidence interval on the mean was computed based on the mean and standard deviation of each replication. In the following graphs, we report the mean and its 99% confidence interval as the function of the query range.

Figures 7 and 8 compare the performance of BP-tree and previous work to answer range queries and k-NN queries, respectively. Those graphs show the average number of distance

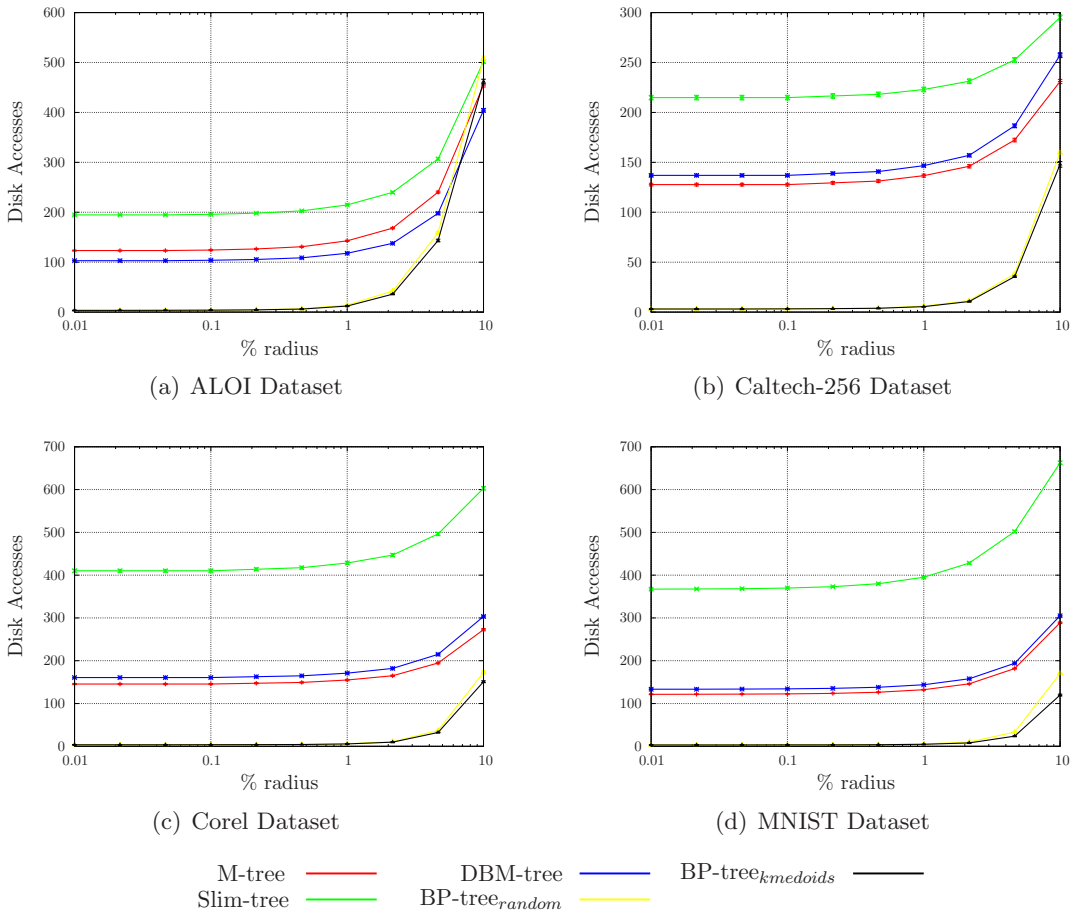


Figure 9: The query performance (given by the average number of disk accesses) for the compared approaches as a function of the query radius, where each of the plots refer to one of our datasets.

calculations for the four datasets. The results are plotted in log scale to minimize the large difference resulting from queries with small and large ranges, which makes the comparison easier.

Clearly, BP-tree consistently outperforms all the compared indexes in the number of distance calculations, with a high statistical significance (confidence higher than 0.99). Note that, by increasing the search radius of range queries, BP-tree performs better than its competitors, reducing up to 70% the average number of distance calculations. For k-NN queries, BP-tree saves, on average, 50% of distance calculations when compared to its best competitor, for any dataset.

In order to evaluate the behaviour of our approach with respect to the number of disk accesses to answer a query, we compared BP-tree with M-tree, Slim-tree, DBM-tree, and iDistance as they are the only ones that consider I/O costs in their analysis. For this purpose, BP-tree was implemented into the GBDI Arboretum Library, with the same code

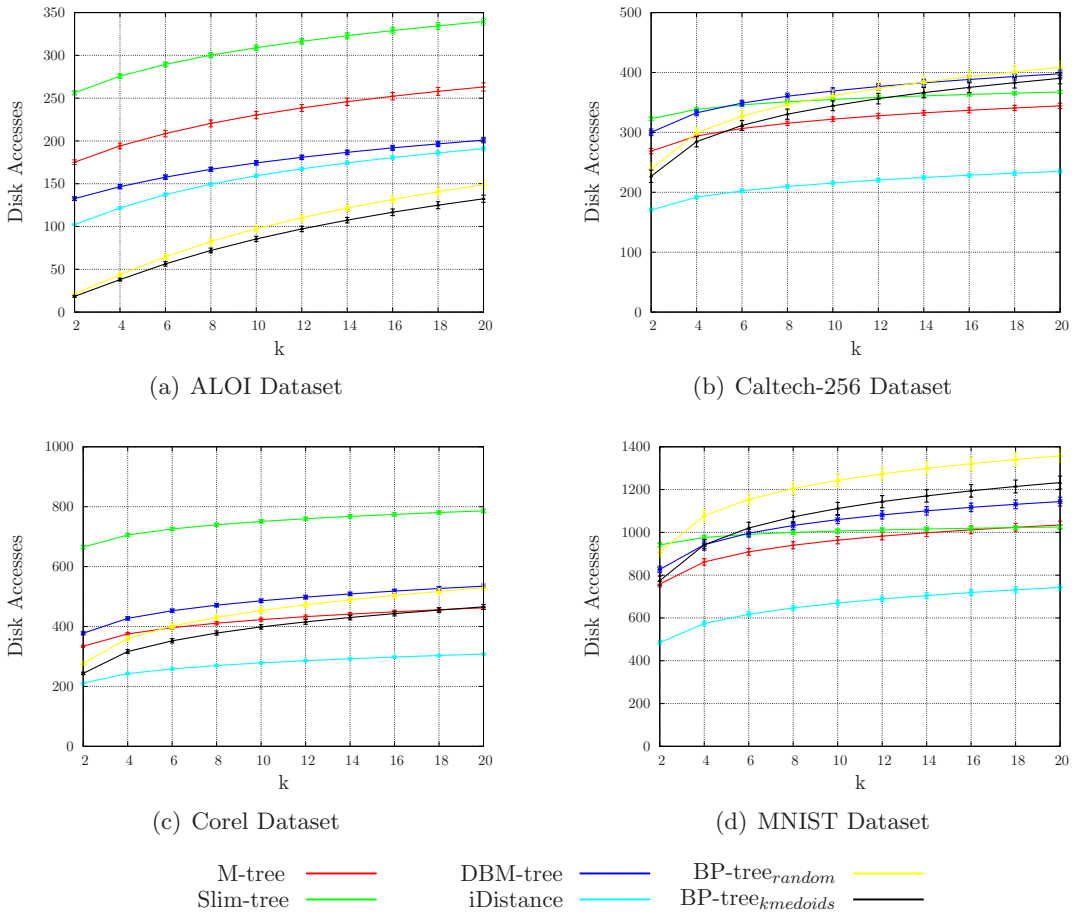


Figure 10: The query performance (given by the average number of disk accesses) for the compared approaches as a function of the number k of nearest neighbors requested for the query, where each of the plots refer to one of our datasets.

optimization, to obtain a fair comparison.

Figures 9 and 10 compare the average number of disk accesses for performing range queries and k-NN queries, respectively. Undoubtedly, BP-tree requires fewer disk accesses than all the above techniques to answer range queries. Notice that, for large query radii, all indexes tend to have the same behavior once almost the entire structure might be accessed. However, it is common to expect a rather small search radius for the majority of queries.

It can be seen from the plots in Figure 10 that BP-tree requires a few more disk accesses than M-tree and Slim-tree for the Caltech-256 and MNIST datasets. For the other collections, BP-tree shows better performance. One of the reasons is that the BP-tree is an unbalanced tree. Note that a similar behaviour is performed by DBM-tree, which is also an unbalanced tree.

Unlike the results achieved in the previous experiment in which iDistance requires a huge number of distance calculations, such an index performs fewer disk accesses to answer k-NN

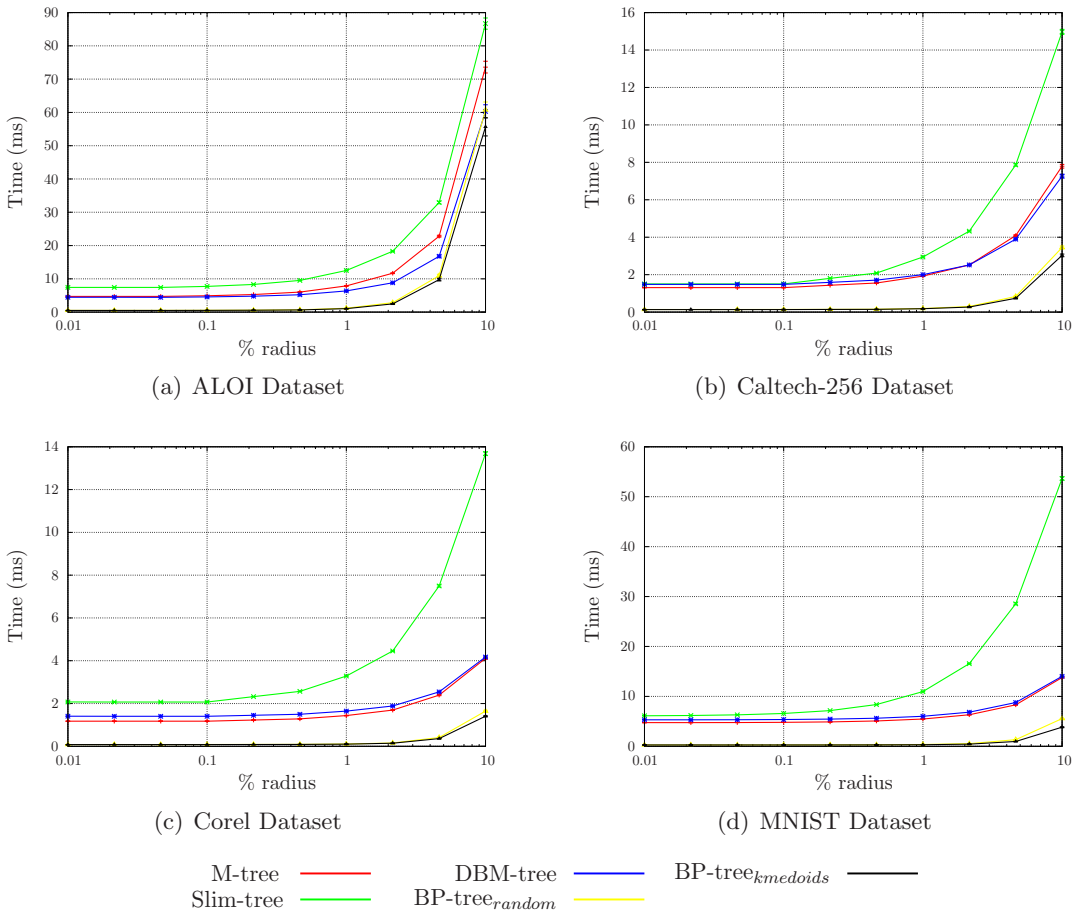


Figure 11: The query performance (given by the average time in milliseconds to answer a query) for the compared approaches as a function of the query radius, where each of the plots refer to one of our datasets.

queries for the most of the datasets. In general, the I/O cost is assumed to be the most important performance measure in large collections. However, some distance functions are so expensive to compute in terms of CPU time that the overall search time is dominated by the total number of distance evaluations performed rather than by the total number of disk pages read, especially for high-dimensional spaces.

This effect can be observed by comparing the query time of the above indexes as it reflects the total complexity of the algorithms besides the number of distance calculations and the number of disk accesses. We present the average time (in milliseconds) required by those indexes for range search and k-NN search in Figures 11 and 12, respectively.

Clearly, BP-tree is significantly faster than all the compared methods for performing similarity queries, with a high statistical significance (confidence higher than 0.99). Moreover, BP-tree performs very well in high-dimensional spaces, even for a poor choice of partitioning scheme and representative objects (i.e., the baseline formed by the random strategy).

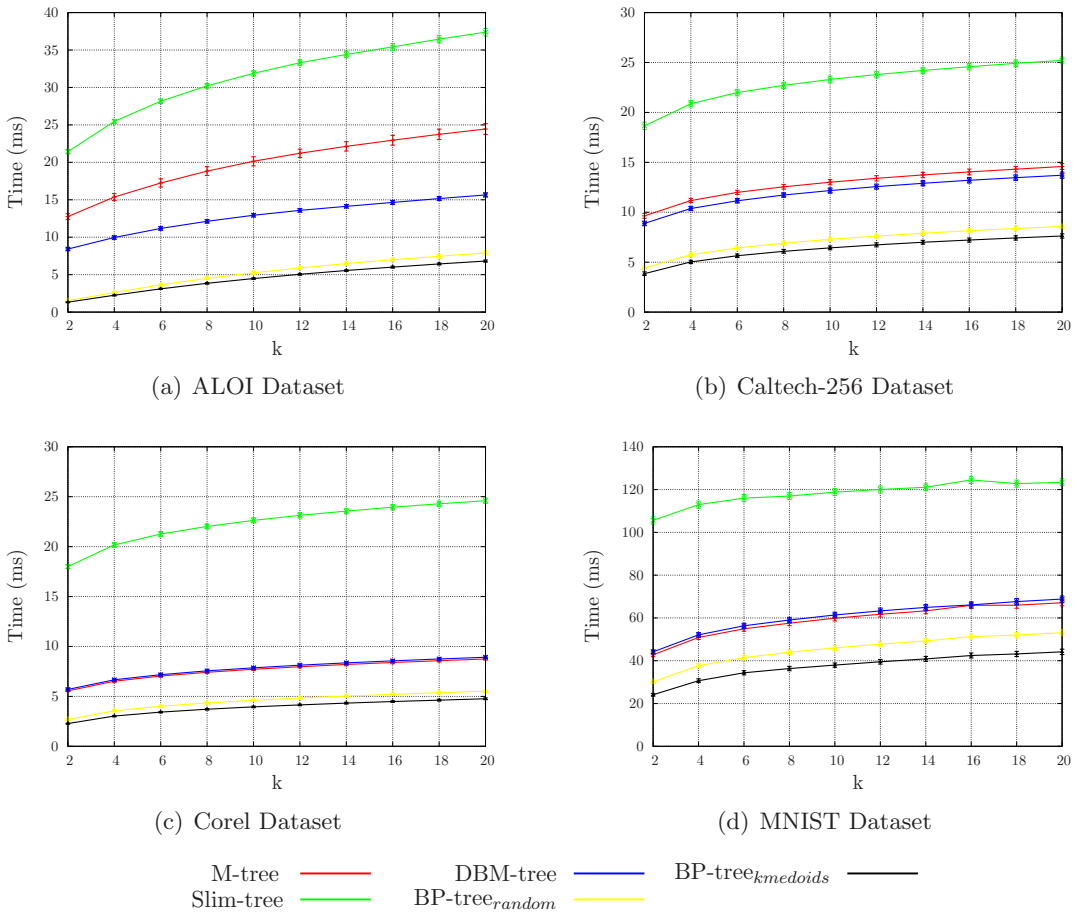


Figure 12: The query performance (given by the average time in milliseconds to answer a query) for the compared approaches as a function of the number k of nearest neighbors requested for the query, where each of the plots refer to one of our datasets.

Notice that BP-tree is at least 20% faster than its best competitor, for any dataset or type of similarity query.

The main reason for those results is the strong interaction between the geometric and data structure constraints. If those are not compatible, the index as a whole suffers. The existing methods usually ignore the distribution properties of the data and, hence, they produce high overlapping areas due to the high dimensionality.

BP-tree aims to fill such a gap by creating an index structure that better fits to the data distribution. This strategy allows to minimize the number of distance calculations and of disk accesses to answer both types of similarity queries. Those benefits are summed up to reduce the query time.

5.2 Scalability

In this section, we examine the behaviour of BP-tree with respect to the number of objects stored in the dataset. The collection used in this experiment was obtained by extracting local features from the ETH-80 Image Set[8] [20], which is a set of 3,280 images.

We used the well-known SIFT method [21] for extracting local features from those images. The resulting collection contains a total of 134,173 SIFT descriptors. Each SIFT descriptor consists of a 128-dimensional feature vector. The distance function used to compare the feature vectors is the Euclidean (L_2) distance. The page size used to build all the compared indexes is 16 KBytes, yielding an average capacity of 64 objects. Figure 13 shows the distance density function of the above database.

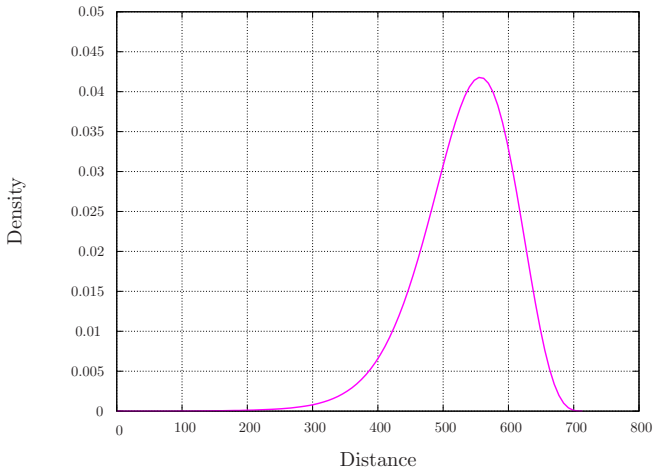


Figure 13: Distance density function of the ETH-80 dataset.

Our first experiment aims at measuring the construction cost of BP-tree. For this purpose, we randomly divided the collection into ten equal-sized parts. We started the experiment by using one of the ten parts as the dataset and gradually added the remaining nine parts one by one until all the parts were included.

Figure 14 shows how the building time (in seconds) of BP-tree grows with the increase of the number of indexed objects. For comparison, we also present the results of M-tree, Slim-tree, and DBM-tree. We use log scale in order to highlight the behaviour of each approach. Like the other methods, BP-tree exhibits a moderate increase with the growing size of the dataset, indicating a good scalability.

The space occupation (in terms of the number of disk pages) of those indexes is shown in Figure 15. Note that the space requirements of BP-tree is similar to that of DBM-tree. The index structures of M-tree and Slim-tree require fewer disk pages. One of the reasons is the better occupation of nodes in balanced trees. In spite of that, the storage utilization by BP-tree scales up very well with respect to the dataset size.

[8] `http://tahiti.mis.informatik.tu-darmstadt.de/oldmis/Research/Projects/categorization/eth80-db.html`

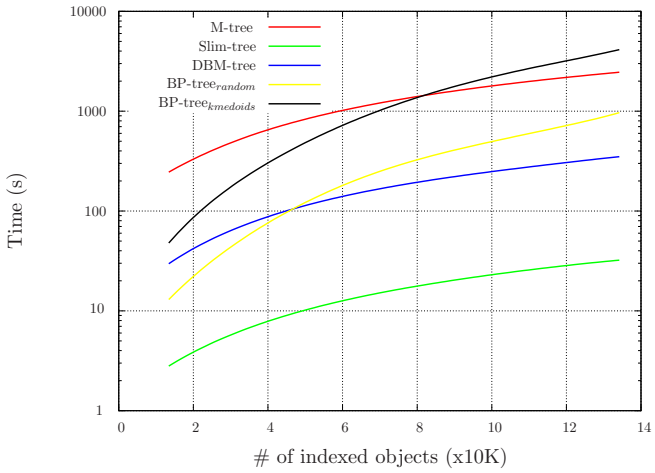


Figure 14: Comparison between the building time (in seconds) of different indexes regarding the size of the dataset.

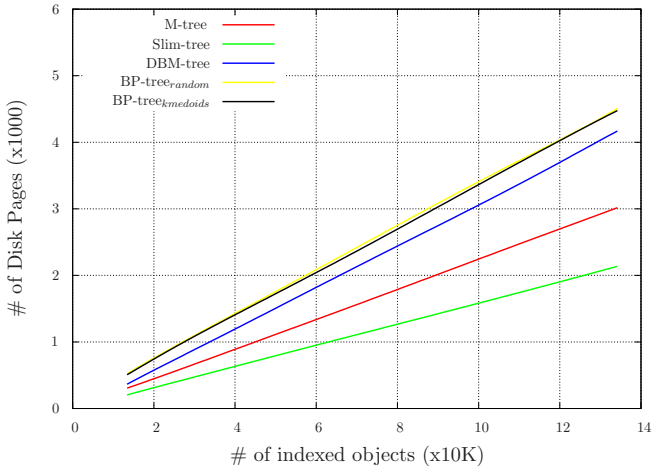


Figure 15: Comparison between the space occupation (in terms of the number disk pages) of different indexes with respect to the dataset size.

We now consider the cost of searching the index. For that, we randomly selected about one percent (1300 local features) of the total number of objects to be used as queries. Five replications were performed in order to ensure statistically sound results. The same set of queries was used at each of the ten steps of the experiment.

The behaviour was equivalent for different values of k and radius. Thus, we report only the results for the maximum values (i.e., the search radius was set to 10% of the largest distance between pairs of objects in the dataset and the number k of nearest neighbors was equal to 20).

Figure 16 indicates the behaviour of BP-tree and previous work regarding the size of

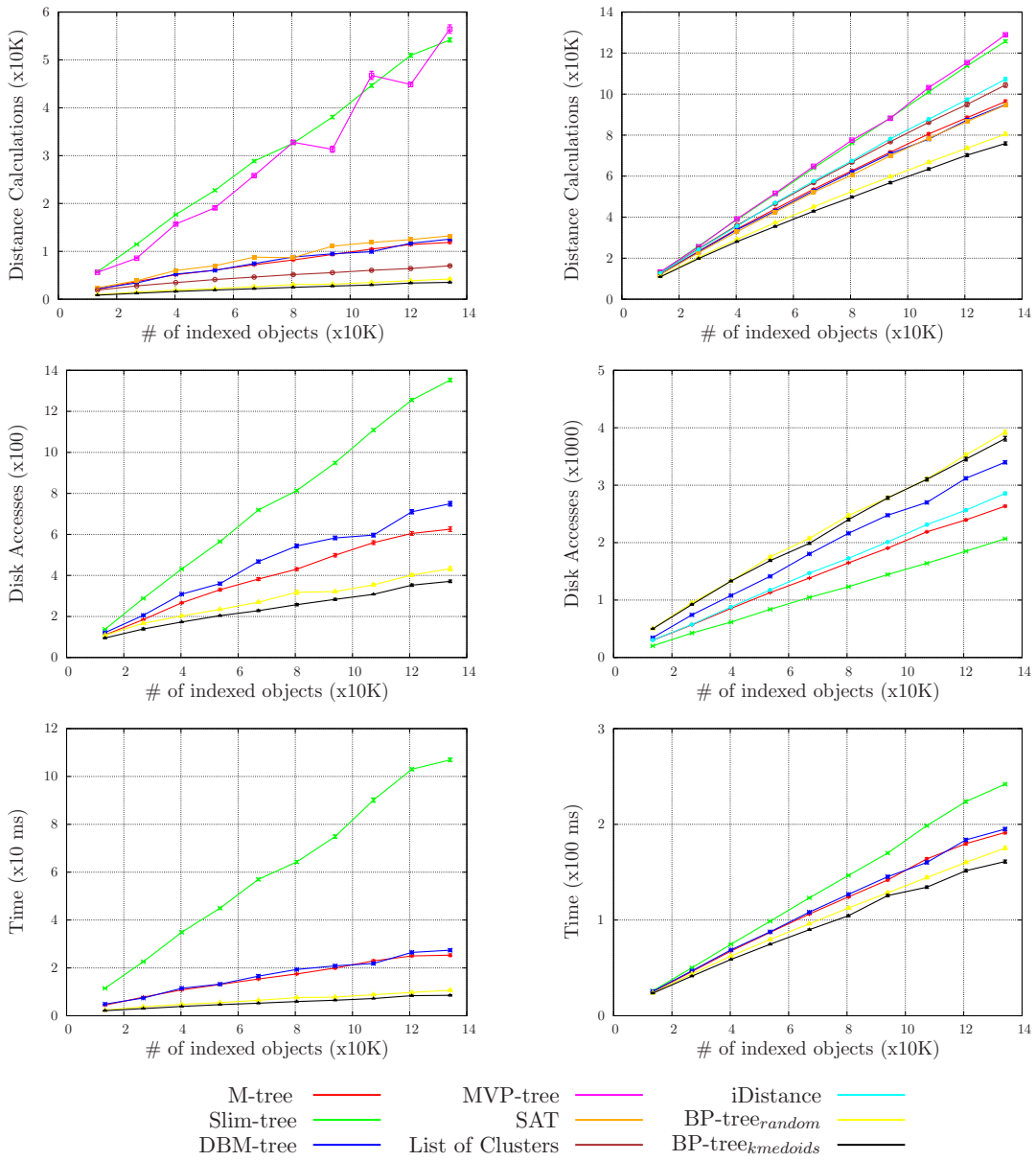


Figure 16: The scalability of the compared approaches with respect to the size of the dataset. We evaluate range queries (left column) and k-NN queries (right column). These graphs present the average number of distance calculations (first row), the average number of disk accesses (second row), and the average time (in milliseconds) to answer a query (third row).

the dataset. The graphs present the average number of distance calculations (first row), the average number of disk accesses (second row), and the average time (in milliseconds) to answer a query (third row). We show the results for both range queries (left column) and

k-NN queries (right column).

Those graphs demonstrate that, with the growth of the dataset, the average number of distance calculations performed by the traditional approaches increases significantly faster than that of BP-tree, for both types of similarity queries.

As it can be seen, by increasing the size of the dataset, BP-tree requires fewer disk accesses than its competitors for range queries. On the other hand, BP-tree performs a few more accesses to disk than the other indexes to answer k-NN queries.

Not surprisingly, BP-tree presents a much slower growing search time than all previous work. Notice that BP-tree exhibits a sublinear behavior with the growing number of indexed objects, which makes it suitable for indexing very large datasets.

6 Conclusions

In this paper, we have presented BP-tree, a new approach for performing similarity search in high-dimensional metric spaces. In BP-tree, the dataset is divided into compact clusters by respecting their distribution. It combines the advantages of both disjoint and non-disjoint strategies in order to achieve a structure of tight and low overlapping clusters, yielding significantly improved performance on similarity queries, especially for high-dimensional spaces.

Our experiments conducted over real-world datasets have shown that BP-tree consistently outperforms the state-of-the-art indexes for similarity search in metric spaces. BP-tree is, on average, 50% faster than traditional approaches for performing similarity queries. We also have shown that BP-tree scales up very well with respect to the number of indexed objects, presenting sublinear behavior, which makes it well-suited to very large datasets.

Future work includes the improvement of the construction procedure of BP-tree, possibly by performing several subtrees in parallel, for they are fully independent from each other. In addition, most of top-down approaches are offline algorithms and they can be sensitive to insertions and deletions. We plan to extend BP-tree to perform regional repartitioning for supporting dynamic operations after the initial creation of the index structure. Moreover, it would be interesting to build approximate or probabilistic algorithms based on this structure, as they have proved to be of great interest in extremely difficult metric spaces using other data structures that typically work well only on easier spaces.

Acknowledgment

This research was supported by Brazilian agencies FAPESP (Grant 07/52015-0, and 08/50837-6), CNPq (Grant 311309/ 2006-2, and 472402/2007-2), and CAPES (Grant 01P-05866/2007).

References

[1] Almeida, J., Torres, R.S., Leite, N.J.: BP-tree: An efficient index for similarity search in high-dimensional metric spaces. In: Proceedings of the ACM International Confer-

ence on Information and Knowledge Management (CIKM'10), pp. 1365–1368. Toronto, ON, Canada (2010)

[2] Almeida, J., Valle, E., Torres, R.S., Leite, N.J.: DAHC-tree: An effective index for approximate search in high-dimensional metric spaces. Journal of Information and Data Management **1**(3), 375–390 (2010)

[3] Baeza-Yates, R.A., Cunto, W., Manber, U., Wu, S.: Proximity matching using fixed-queries trees. In: Proceedings of the Annual Symposium on Combinatorial Pattern Matching (CPM'94), pp. 198–212. Asilomar, CA, USA (1994)

[4] Bishop, C.M.: Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Inc. (2006)

[5] Bozkaya, T., Özsoyoglu, Z.M.: Indexing large metric spaces for similarity search queries. ACM Transactions on Database Systems **24**(3), 361–404 (1999)

[6] Brin, S.: Near neighbor search in large metric spaces. In: Proceedings of the International Conference on Very Large Data Bases (VLDB'95), pp. 574–584. Zurich, Switzerland (1995)

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

[8] Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters **26**(9), 1363–1376 (2005)

[9] Chávez, E., Navarro, G., Baeza-Yates, R.A., Marroquín, J.L.: Searching in metric spaces. ACM Computing Surveys **33**(3), 273–321 (2001)

[10] Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proceedings of the International Conference on Very Large Data Bases (VLDB'97), pp. 426–435. Athens, Greece (1997)

[11] Dalal, N., Triggs, B.: Histograms of oriented gradients for human detection. In: Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR'05), pp. 886–893. San Diego, CA, USA (2005)

[12] Elmasri, R.A., Navathe, S.B.: Fundamentals of Database Systems. Addison-Wesley Longman Publishing Co., Inc. (2005)

[13] Gaede, V., Günther, O.: Multidimensional access methods. ACM Computing Surveys **30**(2), 170–231 (1998)

[14] Geusebroek, J.M., Burghouts, G.J., Smeulders, A.W.M.: The amsterdam library of object images. International Journal of Computer Vision **61**(1), 103–112 (2005)

[15] Griffin, G., Holub, A., Perona, P.: Caltech-256 object category dataset. Tech. Rep. 7694, California Institute of Technology (2007)

[16] Huang, J., Kumar, R., Mitra, M., Zhu, W.J., Zabih, R.: Image indexing using color correlograms. In: Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR'97), pp. 762–768. San Juan, Puerto Rico (1997)

[17] Jagadish, H.V., Mendelzon, A.O., Milo, T.: Similarity-based queries. In: Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'95), pp. 36–45. San Jose, CA, USA (1995)

[18] Jagadish, H.V., Ooi, B.C., Tan, K.L., Yu, C., Zhang, R.: idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. ACM Transactions on Database Systems **30**(2), 364–397 (2005)

[19] Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)

[20] Leibe, B., Schiele, B.: Analyzing appearance and contour based methods for object categorization. In: Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition (CVPR'03), pp. 409–415. Madison, WI, USA (2003)

[21] Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision **60**(2), 91–110 (2004)

[22] Navarro, G.: Searching in metric spaces by spatial approximation. VLDB Journal **11**(1), 28–46 (2002)

[23] Ortega, M., Rui, Y., Chakrabarti, K., Porkaew, K., Mehrotra, S., Huang, T.S.: Supporting ranked boolean similarity queries in MARS. IEEE Transactions on Knowledge and Data Engineering **10**(6), 905–925 (1998)

[24] Ramakrishnan, R., Gehkre, J.: Database Management Systems. McGraw-Hill Co., Inc. (2003)

[25] Rocha, A., Almeida, J., Nascimento, M.A., Torres, R., Goldenstein, S.: Efficient and flexible cluster-and-search approach for cbir. In: Proceedings of the International Conference on Advanced Concepts for Intelligent Vision Systems (ACIVS'08), pp. 77–88. Juan-les-Pins, France (2008)

[26] Santos Filho, R.F., Traina Jr., C., Traina, A.J.M., Vieira, M.R., Faloutsos, C.: The omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. VLDB Journal **16**(4), 483–505 (2007)

[27] Swain, M.J., Ballard, B.H.: Color indexing. International Journal of Computer Vision **7**(1), 11–32 (1991)

[28] Traina Jr., C., Traina, A.J.M., Faloutsos, C., Seeger, B.: Fast indexing and visualization of metric data sets using slim-trees. IEEE Transactions on Knowledge and Data Engineering **14**(2), 244–260 (2002)

[29] Uhlmann, J.K.: Satisfying general proximity/similarity queries with metric trees. Information Processing Letters **40**(4), 175–179 (1991)

[30] Vieira, M.R., Traina Jr., C., Chino, F.J.T., Traina, A.J.M.: DBM-tree: Trading height-balancing for performance in metric access methods. Journal of the Brazilian Computer Society **11**(3), 37–52 (2006)

[31] Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the ACM/SIGACT-SIAM International Symposium on Discrete Algorithms (SODA'93), pp. 311–321. Austin, Texas, USA (1993)

[32] Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach. Springer-Verlag, Inc. (2005)