BP-tree: An Efficient Index for Similarity Search in High-Dimensional Metric Spaces[*]

Jurandy Almeida, Ricardo da S. Torres, and Neucimar J. Leite
Institute of Computing, University of Campinas – UNICAMP
13083-852, Campinas, SP – Brazil
{jurandy.almeida,rtorres,neucimar}@ic.unicamp.br

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. Most of existing indexes are constructed by partitioning the data set 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 data set into compact clusters. Distinctive 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 data sets show that BP-tree consistently outperforms state-of-the-art solutions.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*access methods*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms

Algorithms, Performance

Keywords

clustering methods, database indexing, metric access methods, metric spaces, similarity search

[*]The authors thank the financial support of Brazilian agencies FAPESP, CNPq, and CAPES.

1. INTRODUCTION

Similarity search in metric spaces is a subject of interest for many research communities. For over two decades, significant research efforts have been spent trying to improve the performance in processing similarity queries.

Most of existing indexes employed to accelerate data retrieval are constructed by partitioning a set of objects using distance-based criteria [6, 12]. In general, those techniques follow two basic paradigms. One type of methods produces disjoint partitions, but ignores the distribution properties of the data [2, 5, 11, 15, 17]. The other type of methods produces non-disjoint groups, which greatly affect the search performance [3, 7, 14, 16].

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 data set 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. Results from a rigorous empirical comparison on real-world data sets show that our approach consistently outperforms the state-of-the-art solutions.

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

2. RELATED WORK

The problem of supporting similarity queries in metric spaces has long been pursued in the database area. A survey of metric access methods can be found in [6].

Previous works have focused on the static case. For instance, the structures proposed by Burkhard and Keller [4], Uhlmann [15], and also the VP-tree [17], GNAT [3], MVP-tree [2], and List of Clusters [5] do not support further insertions, deletions, and updates after the creation of the index structure. Overcoming this inconvenience, dynamic structures, including M-tree [7], Slim-tree [14], SAT [11], and DBM-tree [16], have been developed.

In general, all those approaches follow two basic paradigms: disjoint or non-disjoint. The former partitions the data set into disjoint clusters, but ignores the distribution properties of the data [2, 5, 11, 15, 17]. The latter produces non-disjoint groups, which greatly affect the search performance [3, 7, 14, 16].

Different from all of the previous techniques, BP-tree does not divide the data set into disjoint or non-disjoint groups. Instead, it is an index structure that combines the advantages of both those strategies.

3. THE BP-TREE

Overall, BP-tree is an unbalanced tree index generated by the hierarchical partitioning of the data set. Like other metric trees, the objects of the data set are stored into fixed size disk pages. Each page holds a predefined maximum number of objects K. 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
k	the number of partitions spanned by a set
K	capacity of a disk page
M	the minimum occupation of a node
N	number of objects in a set
O	a set of objects
C	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. The objects of the data set are stored in both index and leaf nodes.

The structure of a leaf node is

$$leafnode \text{ [array of } < oid(o_i), d(o_i, o_{rep}), o_i > \text{]},$$

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

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

where o_i keeps the representative of the subtree 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 rooted by o_i.

The tree construction is performed in a top-down fashion. In order to clarify this approach, look at Figure 1. At the beginning, the set of objects $O = \{o_1, o_2, \ldots, o_N\}$ is considered to be part of a single partition. Objects in this set are first divided into $k \leq K$ disjoint subpartitions $c_1, c_2, \ldots, c_k$. Information about all those subpartitions form the index node of the first level of the tree. For each partition c_i, a subset O_{c_i} is created by taking the objects of c_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 K 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.

Algorithm 1 formalizes the above procedure. It starts by checking the cardinality of the set of objects O (line 2). If it can fit into a disk page, the function CREATE-LEAFNODE is used to create a leaf node (line 3). Otherwise, we call the function SPLIT in order to divide the set into $k \leq K$ partitions with a minimum occupation equals to M (line 5). The function SPLIT can use any partitional clustering method, such as k-medoids [1]. The partitional algorithm is responsible for finding the representatives and the reference objects of each level. Next, we check if the set can be divided

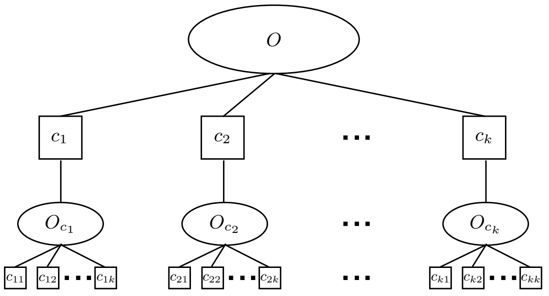


Figure 1: A representation of BP-tree.

(line 7). If so, we call the function INITIALIZE-INDEXNODE, which creates an index node using the information about all those partitions (line 10). Thereafter, for each partition, the function CREATE-SUBSET is used to create a subset of objects (line 12). This function is responsible for taking objects of a given partition. After that, we repeat this process for all of the new subset of objects (line 14). Finally, the function UPDATE-INDEXNODE is used to update the information in each entry of the index node (line 15).

Algorithm 1 BP-tree construction.

```
 1: function BP-TREE($d, K, M, N, O$)
 2:     if $N \leq K$ then
 3:         return CREATE-LEAFNODE($N, O$);
 4:     else
 5:         $C \leftarrow$ SPLIT($d, K, M, N, O$);
 6:         $k \leftarrow cardinality(C)$;
 7:         if $k < 2$ then
 8:             return CREATE-LEAFNODE($N, O$);
 9:         else
10:             $Parent \leftarrow$ INITIALIZE-INDEXNODE($k, C$);
11:             for $c \in C$ do
12:                 $O_c \leftarrow$ CREATE-SUBSET($c, k, C, N, O$);
13:                 $N_c \leftarrow cardinality(O_c)$;
14:                 $Child \leftarrow$ BP-TREE($d, K, M, N_c, O_c$);
15:                 UPDATE-INDEXNODE($c$, $Parent$, $Child$);
16:             end for
17:             return $Parent$
18:         end if
19:     end if
20: end function
```

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.

The range search algorithm starts at the root node and traverses the tree in a depth-first manner, visiting all non-discardable subtrees. During the search, all the stored distances are used to prune subtrees.

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. 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. Afterwards, we prune all subtrees via the criterion $d(o_q, o_i) - r(o_i) > r(o_q)$. Next, we use triangle inequality and avoid to access subtrees by checking if $|d(o_q, o_i) - d(o_i, o_{ref})| - r(o_{ref}) > r(o_q)$. 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 $ptr(T(o_i))$. All non-pruned subtrees $ptr(T(o_i))$ and their distances $d(o_q, o_i)$ are stored in a list. Thereafter, we find the entry whose representative o_i is closest to the query object o_q (i.e., $d_{min} \leftarrow \text{argmin } d(o_q, o_i)$) and, then, we remove from the list all entries such that $(d(o_q, o_i) - d_{min})/2 > r(o_q)$. Finally, the algorithm proceeds by recursively searching the remaining subtrees in the list.

Leaf nodes are similarly processed. Each entry is examined using the pruning condition $|d(o_q, o_{rep}) - d(o_i, o_{rep})| > r(o_q)$. 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 $d(o_q, o_i) \leq r(o_q)$.

The algorithm for k nearest neighbors queries is performed by simulating a dynamic range search with the query range $r(o_q)$ being the distance between the query object o_q and the current k-th nearest neighbor. At the beginning, $r(o_q)$ 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).

For this purpose, we have a priority queue of nodes, the most promising first. Initially, we insert the root node in the priority queue. Iteratively, we extract the most promising node, process it, and insert all non-pruned subtrees in the priority queue. This is repeated until the priority queue becomes empty.

The distance $d(o_q, o_i)$ is used to measure how promising is a subtree $ptr(T(o_i))$ and, hence, the subtrees inserted in the priority queue are visited in increasing order of proximity to the query object o_q. Since $r(o_q)$ 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 with the subtree $ptr(T(o_i))$ and its distance $d(o_q, o_i)$. As we extract an entry from the priority queue, we check whether those lower bounds exceed $r(o_q)$, in which case the extracted node is discarded.

4. EXPERIMENTAL EVALUATION

In this section, we compare the performance of our technique with previous work in processing similarity queries. We implemented BP-tree from scratch in C++ under Linux. In our implementation, we use the k-medoids to partition data. 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 run Gentoo Linux (2.6.31 kernel) and the ext3 file system.

We compared BP-tree against MVP-tree, SAT, List of Clusters, M-tree, Slim-tree, and DBM-tree, 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]. In order to guarantee a fair comparison, all the compared methods were configured using their best recommended setup. The page capacity used to build all the compared indexes was set to 64 objects.

BP-tree was tested using two sets of images described in the literature and extensively used by computer vision and image processing communities. The first set contains 30,607 images from the Caltech-256 Object Category Dataset[3] [9]. We converted each image to a 64-dimensional feature vector by computing a Color Histogram [13]. 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 other data set is a collection of 72,000 images from the Amsterdam Library of Object Images (ALOI)[4] [8]. In this case, each image is characterized by a 256-dimensional feature vector which represents a Color Correlogram [10]. 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 distance function used to compare the feature vectors of both the data sets is the Manhattan (L_1) distance.

For each collection, we randomly selected about five percent of the total number of objects to be used as queries. In order to ensure statistically sound results, five replications were performed for each corpora. Thereafter, we performed k-NN queries and varied the number k of nearest neighbors requested for the query from 2 to 20. For each data set, the 99% confidence interval on the mean was computed based on the mean and standard deviation of each replication.

Figure 2 shows the performance measurements attained for all the compared methods on both the datasets. The first column (Figures 2(a) and (d)) shows the average number of distance calculations. The results are plotted in log scale to minimize the large difference resulting from queries with small and large k, which makes the comparison easier.

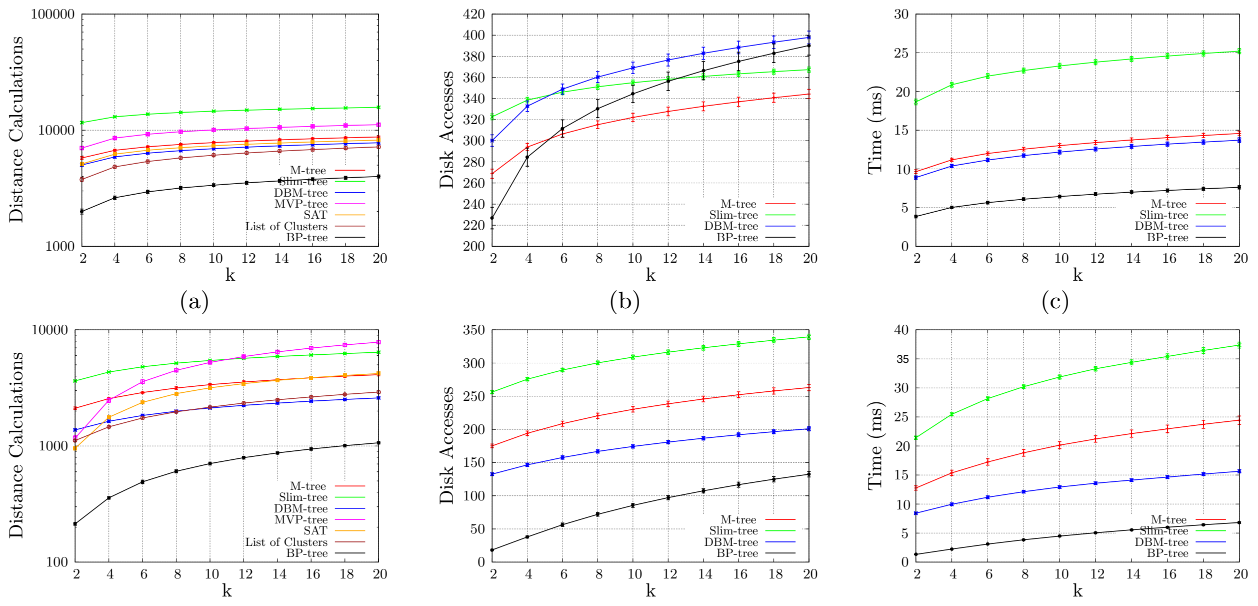
Clearly, BP-tree consistently outperforms all the compared indexes in the number of distance calculations, with a high statistical significance (confidence higher than 0.99). BP-tree saves, on average, 50% of distance calculations for all the data sets when compared to its best competitor.

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, and DBM-tree 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 optimization, to obtain a fair comparison. The second column (Figures 2(b) and (e)) presents the average number of disk accesses required for each of the above techniques.

It can be seen from those plots that BP-tree requires a few more disk accesses than M-tree and Slim-tree for the Caltech-256 dataset (Figure 2(b)). On the other hand, BP-tree shows better performance for the ALOI dataset (Figure 2(e)). One of the reasons is that the BP-tree is an unbalanced tree. Notice that a similar behaviour is performed by DBM-tree, which is also an unbalanced tree.

Nevertheless, the search cost for similarity search in high-dimensional spaces is determined by the pruning rate of the search space, not by the height of the tree, as stated in [5]. It can be seen 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 for those indexes in the third column (Figures 2(c) and (f)). Clearly, BP-tree is significantly faster than the compared indexes for performing similarity queries.

[1]`http://www.sisap.org/Metric_Space_Library.html`
[2]`http://www.gbdi.icmc.usp.br/arboretum/`
[3]`http://www.vision.caltech.edu/Image_Datasets/Caltech256/`
[4]`http://staff.science.uva.nl/~aloi/`



(a) (b) (c)

(d) (e) (f)

Figure 2: Comparison of the average number of distance calculations (first column), the average number of disk accesses (second column), and the average time (in milliseconds) to answer a query (third column) of all the compared methods for the Caltech-256 (top row) and ALOI (bottom row) datasets.

5. 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 data set 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 data sets have shown that BP-tree consistently outperforms the state-of-the-art indexes for similarity search in metric spaces. BP-tree is, on average, up to 50% faster than traditional approaches for performing similarity queries.

Future work includes the investigation of pros and cons of using existing clustering algorithms for indexing high-dimensional data. We also plan to extend BP-tree to perform regional repartitioning for supporting insertions and deletions after the initial creation of the index structure.

6. REFERENCES

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

[2] T. Bozkaya and Z. M. Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Trans. Database Syst.*, 24(3):361–404, 1999.

[3] S. Brin. Near neighbor search in large metric spaces. In *VLDB*, pages 574–584, 1995.

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

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

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

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[8] J.-M. Geusebroek, G. J. Burghouts, and A. W. M. Smeulders. The amsterdam library of object images. *IJCV*, 61(1):103–112, 2005.

[9] G. Griffin, A. Holub, and P. Perona. Caltech-256 object category dataset. Technical Report 7694, California Institute of Technology, 2007.

[10] J. Huang, R. Kumar, M. Mitra, W.-J. Zhu, and R. Zabih. Image indexing using color correlograms. In *CVPR*, pages 762–768, 1997.

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

[12] A. Rocha, J. Almeida, M. A. Nascimento, R. Torres, and S. Goldenstein. Efficient and flexible cluster-and-search approach for cbir. In *Int. Conf. Adv. Concepts Intell. Vision Syst.*, pages 77–88, 2008.

[13] M. J. Swain and B. H. Ballard. Color indexing. *IJCV*, 7(1):11–32, 1991.

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

[15] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[16] M. R. Vieira, C. Traina Jr., F. J. T. Chino, and A. J. M. Traina. DBM-tree: Trading height-balancing for performance in metric access methods. *J. Braz. Comp. Soc.*, 11(3):37–52, 2006.

[17] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993.