A New Point Access Method based on Wavelet Trees[⋆]

Nieves R. Brisaboa[1], Miguel R. Luaces[1], Gonzalo Navarro[2], and Diego Seco[1]

[1] Database Laboratory, University of A Coruña
Campus de Elviña, 15071, A Coruña, Spain
{brisaboa,luaces,dseco}@udc.es
[2] Center for Web Research, Department of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract. The development of index structures that allow efficient retrieval of spatial objects has been a topic of interest in the last decades. Most of these structures have been designed for secondary memory. However, in the last years the price of memory has decreased drastically. Nowadays it is feasible to place complete spatial indexes in main memory.
In this paper we focus in a subcategory of spatial indexes named Point Access Methods. These indexes are designed to solve the problem of indexing points. We present a new index structure designed for two dimensions and main memory that keeps a good trade-off between the space needed to store the index and its search efficiency. Our structure is based on a *wavelet tree*, which was originally designed to represent sequences, but has been successfully used as an index in areas like information retrieval or image compression.

Key words: spatial index, point access methods, wavelet tree.

1 Introduction

Recent improvements in hardware have made the implementation of Geographic Information Systems (GIS) affordable for many organizations. An outstanding feature of this kind of systems is that huge amounts of spatial data have to be stored and processed. Therefore, a topic of interest in this research area has been the development of spatial indexing methods, which allow efficient access to these data. Many different spatial index structures have been proposed along the years. These structures can be broadly classified into Point Access Methods (PAMs) and Spatial Access Methods (SAMs) [1]. PAMs are used to improve the access time in collections of spatial points. SAMs are more general and are used

⋆ This work has been partially supported by "Ministerio de Educación y Ciencia" (PGE y FEDER) ref. TIN2006-15071-C03-03, by "Xunta de Galicia" ref. 2006/4 and ref. 08SIN009CT, and by Fondecyt Grant 1-080019, Chile.

to improve the access time in collections of geographic objects (e.g. points, lines, polygons, etc.).

Most of these spatial index methods have been designed for secondary memory. This is mainly due to historical reasons. A few years ago, the main memory was small and very expensive. Thus, the development of spatial index structures for main memory was unimaginable. However, in the last years the price of memory has decreased drastically and nowadays it is feasible to place complete spatial indexes in main memory. Hence, new requirements in the design of spatial access methods must be considered in order to develop structures suitable for main memory. In this paper we present a new PAM for two dimensions that stores both the index and the collection of points in a compact structure. This structure reaches a good trade-off between the space needed and its search efficiency. This makes it suitable for main memory.

In the last years, the idea of storing both the data and the index in a compact form has been widely used in the design of index structures in several research fields. These structures are known as *self-indexes*. In [2], a new approach for document indexing using wavelet trees is presented. A wavelet tree [3] is a self-index organized as a binary tree, originally designed to represent and index a sequence. Here we adapt this structure to the special characteristics of spatial data.

2 Related Work

Many different SAMs and PAMs have been proposed along the years. A good survey of these structures can be found in [1]. The main goal of these structures is to improve the performance in the retrieval of geographic objects that satisfy a search query. A common kind of search query that must be solved by both categories of methods is the *region query*. This operation defines a query window (i.e. a rectangular region in the geographic space) and it returns all the geographic objects that overlap that region.

One of the most popular spatial access methods and a paradigmatic example is the R-tree [4]. The R-tree is a balanced tree derived from the B-tree that splits the space into hierarchically nested, possibly overlapping, MBRs (minimum bounding rectangles). The number of children of each internal node varies between a minimum and a maximum. The tree is kept balanced by splitting overflowing nodes and merging underflowing nodes. MBRs are associated with the leaf nodes, and each internal node stores the MBR that contains all the nodes in its subtree. The decomposition of the space provided by an R-tree is adaptive (dependent on the rectangles stored) and overlapping (nodes in the tree may represent overlapping regions). Several variations of the original R-tree have been proposed to improve its efficiency (e.g. the R+-tree or the R*-tree) and to take into account some specific problems (e.g. the STR R-tree for static data). Most of these proposals have been summarized in [5].

The K-d-tree [6] is a d-dimensional data structure and one of the most prominent PAMs. When this structure is used to index a collection of points, it is also known as Point K-d-tree. The K-d-tree is a binary search tree that represents a recursive subdivision of the space based on the value of just one coordinate at each level of the tree. Many variations of this structure differ in the manner in which they partition the space. In our experiments we use a static approach, proposed in [6], that assumes that all the data points are known a priori. In this variation, the partition lines must pass through the data points and the partition axis changes cyclically in a fixed order.

As we noted before, the wavelet tree [3] is a compact structure used in other fields to store and index data in a compressed way. For instance, in [2] a wavelet tree is used to index and retrieve documents and in [7] it is used to index images. It is known to be efficiently implementable [8]. The basic tool used in the wavelet tree is the bit-vector *rank* operation: given a bit vector $B[1,n]$, the query $rank(B,i) = rank_1(B,i)$ returns the number of bits set to 1 in the prefix $B[1,i]$ of B. Symmetrically, $rank_0(B,i) = i - rank_1(B,i)$. The dual query to $rank_1$ is $select_1(B,j)$. It returns the position of the j-th bit set to 1 in B. The definition of $select_0(B,j)$ is analogous. For example, given a bitmap $B = 1000110$, $rank_1(B,5) = 2$, and $select_0(B,4) = 7$. Both rank and select operations can be implemented in constant time and using little additional space on top of B [9,10,11].

3 Spatial Indexing Using Wavelet Trees

3.1 Index Construction

Given a set of N points $P = P_1 \dots P_N$, each point consisting of two coordinates (e.g. latitude and longitude) that define its position in the geographic space with regard to a spatial reference system, we can assume that these points can be distributed in an $N \times N$ matrix with only one point in each row and column. This is not a strong restriction because if two points have the same coordinate we can order them arbitrarily and assign them consecutive rows or columns in the matrix. It is important to note that the matrix is only used to keep the relative positions of the points. Neither the distances nor the proportions are kept in it. This is a very important characteristic because it allows us to construct the matrix for any set of points, even if there are points with duplicate coordinates in the set. The translation from the geographic space to a matrix is illustrated with an example in the left part of Figure 1.

The wavelet tree is a compact structure that can be used to store this matrix with little storage cost. Given an $N \times N$ representative matrix, a wavelet tree with $\lceil \log_2 N \rceil$ levels and N bits per level can be built to store the permutation from the order of the points in one dimension (e.g. longitude) to their order in the other (e.g. latitude). Let $X = P_{X_1} \dots P_{X_N}$ and $Y = P_{Y_1} \dots P_{Y_N}$ be the permutations where the points are ordered by their longitudes and latitudes, respectively. For example, in Figure 1 we can name the points from left to right (i.e. P_i is the i-th point counting from the left). Therefore, the first permutation can be written as $X = P_1 P_2 \dots P_{16}$ and the second as $Y = P_2 P_{13} P_{11} \dots P_1 P_5$.

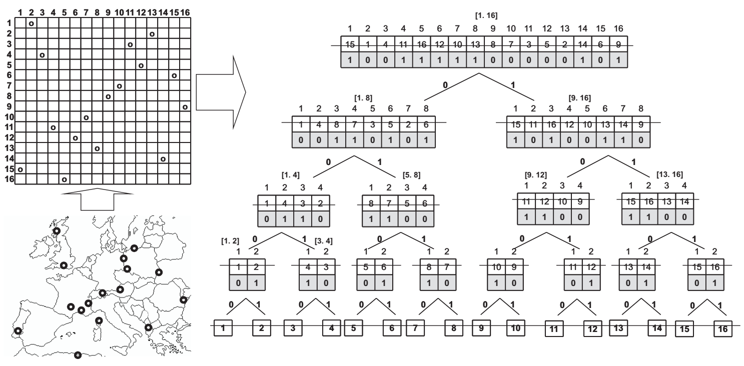

Fig. 1. Wavelet tree construction. Only the greyed data are stored

The point P_1, for instance, is the first one in the order of the longitudes and it is the second to last in the order of the latitudes.

The root of the wavelet tree is a bitmap $B = b_1 \dots b_N$ with the same length of the set of points (i.e. N positions). Each position i represents the i-th point assuming them ordered in the first dimension (e.g. longitude). In the example, $P_{X_1} = P_1$, $P_{X_2} = P_2$, etc. Then, $b_i = 0$ if $P_{Xi} \in P_{Y_1} \dots P_{Y_{N/2}}$, and $b_i = 1$ if $P_{X_i} \in P_{Y_{N/2+1}} \dots P_{Y_N}$. The sequence of the points given a 1 in this vector are processed in the right child of the node, and those marked 0 are processed in the left child of the node. In this way, each node indexes half the symbols indexed by its parent node. This process is repeated recursively in each node until the leaf nodes where the sequence of indexed symbols corresponds to the permutation in the second dimension (e.g. latitude). The right part of Figure 1 shows the wavelet tree that represents the matrix on the left. Each position in each node of the wavelet tree has been annotated with the order of the corresponding point in the second permutation (these orders are crossed out because in fact they are not stored in the wavelet tree).

3.2 Solving Queries

Obtaining the order of a point in a dimension knowing its order in the other dimension is quite simple. If we know the order of a point in the first dimension (in our example, the longitude) we can go down the wavelet tree to obtain its order in the second dimension (in our example, the latitude). The value at a certain position and the *rank* operation are used to go down in the wavelet tree. The bit b_i in the bitmap of a node defines whether the corresponding point is indexed by either the left ($b_i = 0$) or right ($b_i = 1$) branch of this node. In addition, $rank_{b_i}(B, i)$ gives us the position of that point in the bitmap of the child node. This process is repeated until a leaf node is reached, which gives us the position of the point in the other permutation. As an example, in the wavelet tree of Figure 1, the point in column 6 is at row 12. To obtain this result we first retrieve the bit at position 6 of the root node. That bit is set to 1. Then, we obtain $rank_1(B, 6) = 4$. Both results indicate that we have to repeat the operation in position 4 of the right node. If we repeat this process until a leaf node is reached, we obtain the result 12 (i.e. the order in the second dimension of the element at the sixth position in the first dimensions).

On the other hand, if we know the order of a point in the second dimension (in our example, the latitude) we can go up the wavelet tree to obtain its order in the first dimension (in our example, the longitude). The value in the label of the branch that gives access to the node and the *select* operation are used to go up the wavelet tree. As our structure is a perfect binary tree, it is very easy to know at each level of the tree whether the current node is a left or right child of its parent. In the example, the point in row 13 is at column 8. To obtain this result we first calculate $select_0(B, 1) = 1$. We have made a $select_0$ because the position 13 in the leaf level is stored in a left node (i.e. a branch labeled with a 0) and the position 1 because 13 is the first position of its node. In the next level, we have to calculate $select_0(B, 1) = 3$ and then $select_1(B, 3) = 6$ (1 because it is a right child and 3 because this is the position computed in the previous step). Finally we reach the root, where we obtain the result $select_1(B, 6) = 8$.

We can also use the wavelet tree to solve *region query* operations. However, for this purpose we need three auxiliary structures: two arrays with the coordinates ordered in each dimension and the point identifiers ordered in the same order as one of the other arrays. The arrays of ordered coordinates are used to translate spatial queries to ranges of valid rows and columns in the wavelet tree. Once the query has been translated, the range of columns (longitudes) is the range of valid positions in the root node of the wavelet tree. We can go down through the structure using the algorithm that we have sketched before. Nevertheless, the performance of that algorithm can be easily improved taking into account that consecutive points in a parent node remain consecutive in the corresponding child node. Hence, only two *rank* operations (one for the first position of the range and one for the last one) have to be calculated. Furthermore, the range of valid rows (latitudes) obtained in the translation of the query can be used to prune the search tree. Each node in the wavelet tree contains points in a certain range. If this range does not intersect with the range of rows, the algorithm does not continue in that branch.

Figure 2 shows the wavelet tree of the example with the auxiliary structures. In the figure, two arrays with the point identifiers are shown (IDs(X) and IDs(Y)). The structure needs only one of them, and the decision of which one to employ has pros and cons, as we see later. The figure shows an example of a region query q = <(27.53, 15.75), (30.71, 19)>. The translation of this query to the representative matrix defines the range of valid columns [6, 10] and the range of valid rows [9, 14]. The algorithm to solve the query begins with the traversal of the wavelet tree. As we noted before, only the first and the last positions of a chunk (i.e. several consecutive positions) are relevant to decide the chunks of interest in the next level. Therefore, in the first step the algorithm has to calculate $rank_0(B, 6-1)+1 = 3$, $rank_0(B, 10) = 4$, $rank_1(B, 6-1)+1 = 4$

and $rank_1(B, 10) = 6$. Actually, only two of them have to be computed because $rank_0(B, i) + rank_1(B, i) = i$. Thus, the valid chunks in the second level are [3, 4] in the left node and [4, 6] in the right one. However, solutions to the query cannot be in the left node because it covers the range of rows [1, 8], which does not intersect with the range of rows in the query ([9, 14]). Hence, this branch is discarded. The algorithm repeats this process until the leaf level is reached.

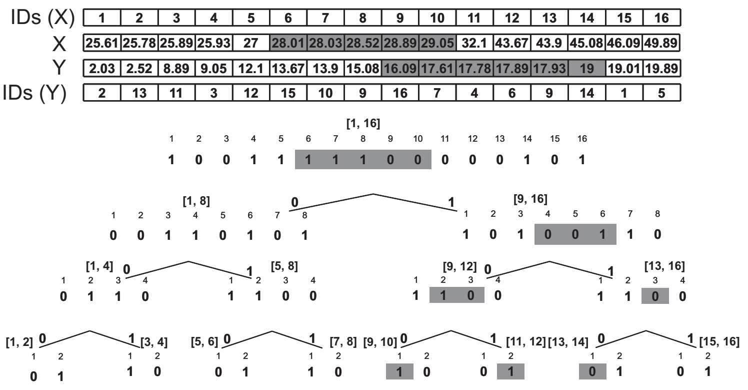
IDs (X) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
X 25.61 25.78 25.89 25.93 27 28.01 28.03 28.52 28.89 29.05 32.1 43.67 43.9 45.08 46.09 49.89
Y 2.03 2.52 8.89 9.05 12.1 13.67 13.9 15.08 16.09 17.61 17.78 17.89 17.93 19 19.01 19.89
IDs (Y) 2 13 11 3 12 15 10 9 16 7 4 6 9 14 1 5
[1, 16]
1 0 0 1 1 1 1 1 0 0 0 0 0 1 0 1
[1, 8] 0 0 1 1 0 1 0 1
[9, 16] 1 0 1 0 0 1 1 0
[1, 4] 0 1 1 0
[5, 8] 1 1 0 0
[9, 12] 1 1 0 0
[13, 16] 1 1 0 0
[1, 2] 0 1
[3, 4] 1 0
[5, 6] 0 1
[7, 8] 1 0
[9, 10] 1 0
[11, 12] 0 1
[13, 14] 0 1
[15, 16] 0 1

Fig. 2. Query solution using the wavelet tree

Once the leaf nodes are reached, the way the algorithm continues depends on the order selected for the point identifiers. If the point identifiers are ordered in the same way of the second array of coordinates (in our example, latitudes), the positions of the leaf nodes can be directly translated to the positions in the array of identifiers. Hence, this version of the algorithm is simpler and, as we will see in the next section, it is more efficient too. On the other hand, if the point identifiers are ordered in the same way of the root node (in our example, longitudes), the algorithm is more complex because when the algorithm discovers that the latitude of a point is valid for the query, the algorithm has to go up the wavelet tree again to obtain its identifier. Since the validity of a latitude can be discovered at any level of the tree, and therefore the algorithm does not always have to reach the leaf nodes, this ordering of the identifiers could improve the performance of the solution. However, as we will see in the next section, its performance is worse than that of the previous version.

4 Experiments

We compare the efficiency of our structure with respect to other spatial index structures, considering first the space requirements and then their efficiency to solve region queries. The results show that our structure achieves a good trade-off between the required space and its time efficiency.

We compare four spatial index structures that run in main memory. The first two are the variants of our index structure presented in Section 3. In the first one, called DPW-tree (*down point wavelet tree)*, the identifiers of the points are stored ordered following the permutation of the leaf nodes and it is only necessary to descend the wavelet tree to obtain the identifiers of the points that fulfil the query. In the second one, called UPW-tree (*up point wavelet tree*), the identifiers of the points are stored ordered following the permutation of the root node and once that a point is known to belong to the query result it is necessary to ascend the tree to retrieve its identifier. The third index structure is a classical R-tree adapted to run in main memory [12]. Although this index structure is not a point access method but a spatial access method, and therefore it is not optimized for point indexing, it is nonetheless the most used index structure in the geographic information systems that are developed nowadays. We use two variations of the original structure: the R*-tree [13] and the static construction of the STR R-tree [14]. We count its space assuming a contiguous layout in memory. Finally, the fourth index structure is a K-d-tree that represents the point access methods. The K-d-tree variant that we have selected [15] is probably the most efficient one because it is optimized for scenarios where the set of points to be indexed is known *a priori*.

To build the query sets, we implemented an algorithm that generates query windows of a given size. This algorithm is based on the one used on the evaluation of the R*-tree in [13]. The query windows generated by the algorithm are distributed uniformly in the space. Furthermore, the size of the window sides is adjusted so that the ratio between the horizontal and vertical extensions varies uniformly between 0.25 and 2.25.

4.1 Space Comparison

Both variants of our structure need to store the coordinates of the N points (two arrays of N 8-byte floating-point numbers), the identifiers (an array of N 4-byte integer numbers) and the wavelet tree. The wavelet tree is a very compact structure that needs only $N \times \lceil \log_2 N \rceil$ bits (1 bit per point per level, that is, N bits per level, and there are $\lceil \log_2 N \rceil$ levels). Moreover, in order to perform *rank* and *select* operations in constant time, some auxiliary structures are needed that use an additional space of around 37.5% of the wavelet tree size [10]. Therefore, the complete structure requires $20 \times N + (N \times \lceil \log_2 N \rceil \times 1.375)/8$ bytes.

The space needed by an R-tree over a collection of N points can be estimated considering an average arity (M). The leaves store the point identifiers. A static structure can store the leaves contiguously without spare space. Thus the leaves amount to $4 \times N$ bytes, and with the table storing the coordinates of the points, we add up to $20 \times N$ bytes. Each leaf costs a MBR and a pointer at its parent, which requires 36 bytes. Over all the levels, there are $N/(M-1)$ nodes, so the total R-tree space is $20 \times N + 36 \times N/(M-1)$. The best performance of the STR R-tree is achieved with an effective M value of 30.

Finally, a K-d-tree that indexes N points has height $h = \lceil \log_2 N \rceil$ and $2^h - 1 + (N \bmod 2^{\lfloor \log_2 N \rfloor})$ nodes, where each node needs 16 bytes (a floating point number and two pointers). Just like the R-tree, we must also consider the $20 \times N$ bytes of the table of points.

To finalize the space comparison, we show the space per point necessary in each spatial index structure. First, both variants of our structure need the same space: 23.69 *bytes/point*. In the same way, both variants of the R-tree need 21.24 *bytes/point*. Finally, the K-d-tree needs 36.00 *bytes/point*. The main conclusion that we can extract from these results is that our structure needs less space than the K-d-tree and more than the R-tree.

4.2 Time Comparison

To perform the time comparison we take into account the two variables that can affect the tests: the selectivity of the queries and the size of the test collections. The query selectivity depends directly on the size of the query windows used. In our tests, we created windows of four different sizes that represent 0.01%, 0.1%, 1% and 10% of the area of the space where the points are represented. We use four synthetic collections with 2^{19}, 2^{20}, 2^{23} and 2^{24} points uniformly distributed in the space. Figure 4 shows four graphs where one can appreciate the influence of these variables in the time needed to solve the queries.

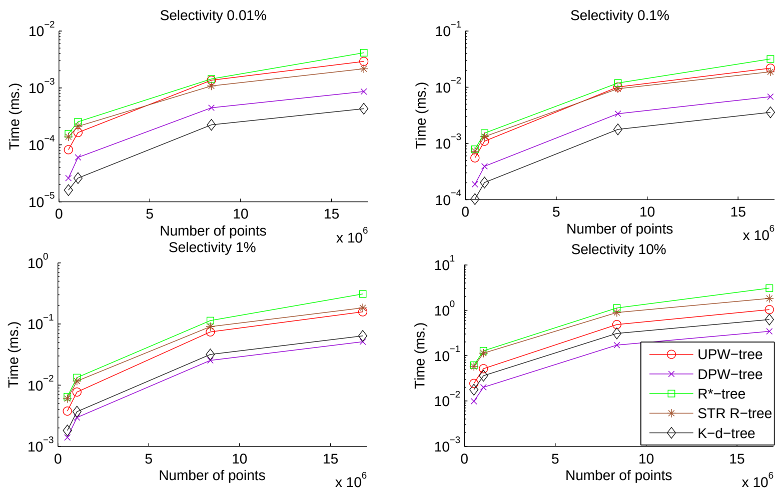

Fig. 3. Time comparison. Note the logscale

We have also experimented with non-uniform spaces. Figure 4 shows the results with two synthetic collections and two real collections. Both synthetic collections have one million points each, the first one with a Zipf distribution (world size $= 1000 \times 1000$, $\rho = 1$) and the second one with a Gauss distribution (world size = 1000 × 1000, mean = 500, sigma = 200). The two real collections have 123,593 postal addresses from New York, Philadelphia and Boston (NE dataset available at http://www.rtreeportal.org) and 2,693,569 populated places distributed all over the world (available at http://www.geonames.org).

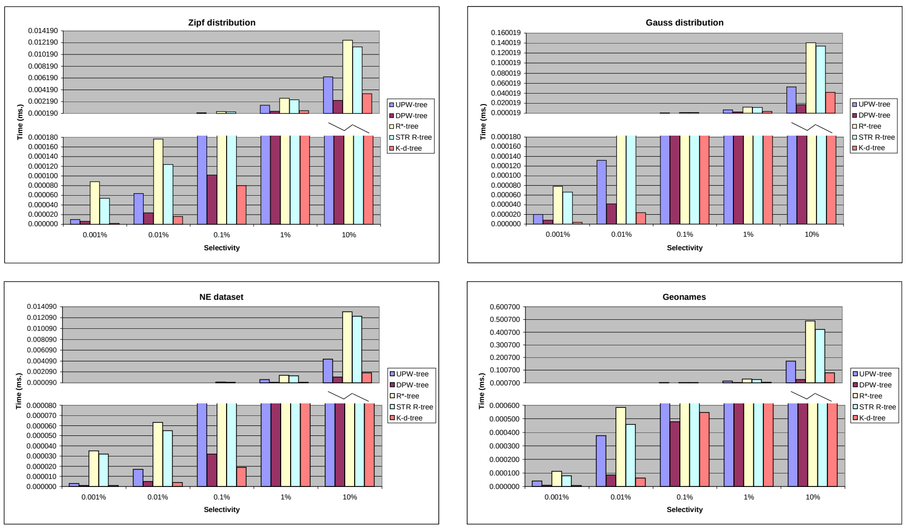

Fig. 4. Time comparison (other collections)

The main conclusion that can be extracted from these results is that our structure is competitive with respect to query time efficiency. The K-d-tree is generally the most efficient structure, but the DPW-tree is always close, and becomes better for low selectivities. On the other hand, the K-d-tree requires significantly more space. Both the R*-tree and the STR R-tree uses less space than the DPW-tree but they are not competitive in time as a point access method. We have included them in the experiments because the R-tree is the paradigmatic example of spatial index structures, and it must be taken into account because it is widely used nowadays. Finally, regarding the two variants of our structure, the DPW-tree (the version that only needs to descend the wavelet tree) is more efficient than the UPW-tree (the version that requires ascending in the wavelet tree).

5 Conclusions and Future Work

We have presented a new point access method based on the *wavelet tree*, a compact structure widely used in other areas such as information retrieval. Our spatial index structure is designed for two dimensions and for main memory, and keeps a good trade-off between the space needed to store the index and its search efficiency. This is an important advantage, as main-memory spatial indexes are becoming popular.

We are currently working on several research lines. First, we are working on allowing the insertion or removal of points once the structure has been constructed. Second, we plan to design algorithms to solve other kinds of queries such as k-nearest neighbor queries or spatial joins. We are also integrating this structure in real geographic information systems in order to check how their performance is improved by our structure. Finally, we are developing a new index structure based on the wavelet tree to index any type of geographic object by means of their MBRs. Alternatively, it could be interesting to see how is the time performance of a static K-d-tree if we reduce its space by replacing the pointer-based structure by a balanced extending representation (see [16]).

References

1. Gaede, V., Günther, O.: Multidimensional access methods. ACM Comput. Surv. **30**(2) (1998) 170–231
2. Brisaboa, N.R., Cillero, Y., Fariña, A., Ladra, S., Pedreira, O.: A new approach for document indexing using wavelet trees. In: Proc. of DEXA'07. (2007) 69–73
3. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. of ACM-SIAM SODA'03. (2003) 841–850
4. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: Proc. of SIGMOD'84, ACM Press (1984) 47–57
5. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-Trees: Theory and Applications. Springer-Verlag New York, Inc. (2005)
6. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9) (1975) 509–517
7. Mäkinen, V., Navarro, G.: On self-indexing images - image compression with added value. In: Proc. of DCC'08, IEEE Computer Society (2008) 422–431
8. Claude, F., Navarro, G.: Practical rank/select queries over arbitrary sequences. In: Proc. of SPIRE'08. (2008) 176–187
9. Munro, I.: Tables. In: Proc. of 16th FSTTCS. (1996) 37–42
10. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Proc. of 4th WEA (Poster). (2005) 27–38
11. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. of 9th ALENEX. (2007)
12. Hadjieleftheriou, M.: Spatial index library. Retrieved March 2009 from http://research.att.com/ marioh/spatialindex/.
13. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. SIGMOD Rec. **19**(2) (1990) 322–331
14. Leutenegger, S., Lopez, M., Edgington, J.: Str: A simple and efficient algorithm for r-tree packing. In: Proc. of ICDE'97. (1997) 497–506
15. Tagliasacchi, A.: Kd-tree for matlab. Retrieved March 2009 from http://www.mathworks.com/matlabcentral/fileexchange/21512.
16. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. **39**(1) (2007)