Compact and queryable representation of raster datasets

Susana Ladra
Universidade da Coruña
Facultade de Informática,
Campus de Elviña, s/n,
15071, A Coruña, Spain
susana.ladra@udc.es

José R. Paramá
Universidade da Coruña
Facultade de Informática,
Campus de Elviña, s/n,
15071, A Coruña, Spain
jose.parama@udc.es

Fernado Silva-Coira
Universidade da Coruña
Facultade de Informática,
Campus de Elviña, s/n,
15071, A Coruña, Spain
fernando.silva@udc.es

ABSTRACT

Compact data structures combine in a unique data structure a compressed representation of the data and the structures to access such data. The target is to be able to manage data directly in compressed form, and in this way, to keep data always compressed, even in main memory. With this, we obtain two benefits: we can manage larger datasets in main memory and we take advantage of a better usage of the memory hierarchy.

In this work, we present a compact data structure to represent raster data, which is commonly used in geographical information systems to represent attributes of the space (i.e., temperatures, elevation measures, etc.). The proposed data structure is not only able to represent a dataset in compressed form and access to an individual datum without decompressing the dataset from the beginning, but it also indexes its content, and thus it is capable of speeding up queries.

There have been previous attempts to represent raster data using compact data structures, which work well when the raster dataset has few different values. However, when the range of possible values increases, performance in both space and time degrades. Our new method competes with previous approaches in that first scenario, but scales much better when the number of different values and the size of the dataset increase, which is critical when applying over real datasets.

CCS Concepts

•Information systems → Geographic information systems; •Theory of computation → Data compression; *Sorting and searching;* **•Applied computing →** *Physical sciences and engineering;*

Keywords

Geographic information systems; raster datasets; data compression; query processing

SSDBM '16, July 18 - 20, 2016, Budapest, Hungary

ISBN 978-1-4503-4215-5/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2949689.2949710

1. INTRODUCTION

Geographical information systems can manage spatial information using different data models. At the conceptual level, we have two alternatives: *object-based models* and the *field-based models* [22]. An object-based model considers that the space contains discrete and identifiable entities, each with a geospatial position. Whereas, a field-based model can be seen as a mathematical function that for each position of the space gives a value. Examples of object-based models are those containing buildings, roads, and other man-made objects. Field-based examples are usually more related to physical properties like land elevation, temperature, atmospheric pressure, etc. Considering the logical level, we can find another two models: *vector models*, which represent the geographic information using points and line segments, and *raster models*, which represent geographic information by partitioning the space as a tessellation, in most cases, of squares of the same size that are usually called cells. In that case, the space is represented as a grid where each cell has a value [15]. Although any logical spatial model can be used to represent any conceptual spatial model, vector models are commonly used to represent object-based models and raster models are often used for field-based models.

In this work, we deal with raster models. Given that any image can bee seen as a raster, the use of this data model is massive. Other examples of application of raster datasets could be pollution control, weather forecast, satellite imagery, remote sensing capture, 3D modeling, engineering, etc. All these scenarios have something in common, they deal with big raster datasets.

Data compression has been traditionally applied in these contexts with the goals of saving space and bandwidth [21, 14]. In most cases, it is not possible to process or query the original data in its compressed shape, requiring a decompression phase that is generally time-consuming. However, in the last recent decades, a new family of data structures, called *compact data structures*, has changed this classical way of using compression. Compact data structures combine in a unique data structure a compressed representation of a dataset and the access methods that allow us to retrieve any given datum, without the need of decompressing the dataset from the beginning [12]. The idea is to keep data always compressed, even in main memory. The benefits are obvious, in addition to the typical savings in disk space and bandwidth, we can process larger datasets, and moreover, this processing can be more efficient thanks to a better usage of the memory hierarchy. In many cases, compact data

structures, in addition to a compressed representation, provide some sort of self-indexation, which allow us to answer queries even faster than performing that query over the plain representation and within the same compressed space [16, 18].

Moreover, due to the size of many of those datasets, it is common to process them by pieces using some sort of parallel computing [10, 17, 6]. In that scenario, data interchanges between nodes can slow down the process due to bottlenecks in the network. Compression has been massively used to reduce bandwidth consumption [5] in that scenario. Traditionally, compression methods applied for this context have been designed to perform the compression and decompression processes very fast, since, as explained, data have to be decompressed prior to any process, and thus data should be compressed before any data exchange and decompressed at the destination node. With a compact data structure approach, data can be interchanged between nodes in compressed form, and also processed in compressed form at the destination node, without decompressing it, saving space and time.

There exists vast research focused on compressing raster datasets, and also for creating additional indexes on top of the raster data to improve query and processing performance[8, 23]. However, to the authors' knowledge, only two types of compact data structures have been proposed specifically designed to deal with this type of information [7]. They have been proved to work well when the number of different possible values in the raster is low, yet when that number grows, both the space and the time needed to answer queries increase dramatically. When dealing with field-based models, it is likely to have real values or large integers as values for each cell, and thus, the range of possible values can be high. Therefore, these existing compact representations become impractical for real datasets.

In this work, we present a new compact data structure to represent raster datasets, called k^2-*raster*, which is based on the k^2-tree [4], a compact representation for representing binary matrices in little space that follows a Quadtree strategy [20] and supports efficient random access to the data. The proposed data structure has a competitive performance with that of the previous approaches when the number of possible values in the raster dataset is low, but scales much better when the number of possible values increases, being much better in that scenario. In our experiments, k^2-*raster* occupies between 4% and 58% of the space occupied by the plain representation, which is up to 6 times less that the previous approaches. In addition, k^2-*raster* was able to answer queries up to 34 times faster than those previous data structures.

The rest of the paper is structured as follows. Section 2 presents some previous concepts that are relevant for describing our proposal and the related work, which is presented in Section 3. Section 4 shows our structure to represent and query raster datasets. Section 5 details our experimental study. Finally, Section 6 presents the conclusions and future work.

2. PREVIOUS CONCEPTS

2.1 Rank and select operations over bitmaps

Let $B[1, n]$ be a bitmap, that is, a sequence of bits. We define operation $rank_b(B, i)$ as the number of occurrences of bit $b \in \{0, 1\}$ in $B[1, i]$. When omitting b, *rank* operation returns the number of 1s up to a given position, that is, $rank(B, i) = rank_1(B, i)$.

This operation can be answered in constant time using just $o(n)$ extra bits on top of B [11]. It is the basis of most of the compressed data structures of the literature, and it will also be used in our proposal.

2.2 DACs

Directly Addressable Codes (DACs) [3] is a variable-length encoding scheme for sequences of integers that supports fast direct access to any position of the sequence in a very compact space. It is oriented for sequences of integers with skew frequency distributions, where the number of occurrences of smaller integer values is higher than the number of occurrences of larger integer values.

Given a sequence of integers $X = x_1, x_2, \ldots, x_n$, DACs rearrange the binary representation of those integers into a level-shaped structure as follows: the first level B_1 contains the first n_1 bits of each integer. In addition, there is a bitmap C_1 indicating for each integer, whether its binary representation requires more than n_1 bits or not. If it does not require more than n_1 bits, bitmap C_1 stores a 0. Instead, if it requires more than n_1 bits, it stores a 1 and includes the following n_2 bits of its binary representation in a second level B_2. This scheme is repeated using as many levels as needed. The number of levels L and the number of bits n_l at each level l, with $1 \leq l \leq N$, can be computed to achieve the minimum space.

The fast direct access is provided thanks to the support of *rank* operations over the bitmaps C_l, as they allow us to obtain the complete binary representation of the integer in a given position, by a top-down traversal of the level-shaped structure. Thus, the worst case time for extracting a random codeword is $O(L)$, being L the number of levels used.

Optimizing DACs to obtain the minimum space possible may lead to an inefficient representation in terms of access time, if it requires a considerable large number of levels. DACs can also be configured to obtain the minimum space as possible but limiting the number of levels L to use, which is given as a parameter. This will be the approach followed in our proposal.

2.3 k^2-tree

The k^2-tree is a compact data structure that was originally proposed for representing Web graphs in little space while providing navigational capabilities [4]. It represents the binary adjacency matrix of a graph using a succinct representation for trees, more concretely, the LOUDS (level-ordered unary degree sequence) tree representation [11].

The k^2-tree builds a non-balanced k^2-ary tree from the graph by recursively subdividing the adjacency matrix into k^2 submatrices of the same size. It starts by subdividing the original matrix into k rows and k columns of submatrices of size n^2/k^2. Each of the resulting k^2 submatrices will generate a child of the root node whose value is 1 iff there is at least one 1 in the cells of that submatrix. A 0 child means that the submatrix has all 0s and hence the tree decomposition ends there. The method proceeds recursively for each

child with value 1, until it reaches a submatrix full of 0s, or it reaches the cells of the original matrix (i.e., submatrices of size 1×1). Figure 1 shows an example of this subdivision (left) and the resulting k^2-ary tree (right) for $k = 2$.

The tree is compactly represented by just storing in two bitmaps T and L all the bit values of the tree following a level-wise order. T stores all the bits of the k^2-tree except those at the last level of the tree, and L stores the last level of the tree, thus containing the binary value of (some) original cells of the adjacency matrix. It is possible to navigate this space-efficient representation by just supporting *rank* operations over bitmap T, imposing little extra space on top of it [9]. In particular, it is possible to obtain any cell, row, column or region of the matrix in a very efficient time. The top-down traversals in the conceptual tree are simulated by *rank* operations in its bitmap representation.

The k^2-tree representation obtains excellent performance when the binary matrix is sparse, has large zones of 0s and the 1s are clustered. There exists a variant of the original method that behaves better when the number of 1s grows [7]. This variant also compresses large areas of 1s. It represents uniform areas (either *black* zones of 1s, or *white* zones of 0s) with a 0, adding a way to distinguish black and white areas; and areas with mixed 1s and 0s (*grey* areas) with a 1, requiring further subdivisions in the tree. This representation is more suitable for representing other types of datasets different from Web graphs, such as binary images.

The k^2-tree representation was also adapted to support the compact representation of RDF datasets [2], moving objects [19], general graphs [1], and raster data [7], among others. The approaches used to represent raster data using the k^2-tree as basis will be explained in the next section.

3. RELATED WORK

In this section, we describe the previous proposals for efficiently representing and querying raster datasets as compact data structures in main memory: accumulated k^2-trees, also known as k^2-*acc*, and k^3-*trees*. They are both variations of the k^2-tree to tackle the scenario where, instead of binary matrices, we want to efficiently represent general raster data, where each cell has an integer value.

These approaches have been proved to perform better than other representations of raster datasets, such those based on GeoTIFF images [7]. This behavior was indeed expected, as the k^2-tree uses a MX-Quadtree strategy [20], which has been extensively used for image data.

3.1 k^3-trees

The first of the strategies consists in generalizing the k^2-tree to deal with a three-dimensional binary cube instead of a two-dimensional binary matrix. This can be trivially done by extending the space partitioning while maintaining the representation techniques used for k^2-trees.

Thus, the k^3-tree stores tuples $\langle x, y, z \rangle$ such that the raster value at position (x, y) is z. This representation can obtain the value of a given cell or all the cells with a given value or range of values by just traversing the k^3-tree fixing some of their dimensions (x and y for the first query and z for the lasts queries).

3.2 k^2-acc

The second strategy proposed for representing raster datasets in compact space consists in creating a collection of k^2-trees, as many as different existing values of the raster. Assuming there are t different values in the raster: $v_1 < v_2 < \cdots < v_t$, it creates t k^2-trees, namely $K_1, K_2, \ldots, K_t$, where each K_i has a value 1 set up in those cells whose value is $v \leq v_i$. Thus, this approach is denoted accumulated k^2-trees or k^2-acc. These k^2-trees use the variant that also compresses large zones of 1s, as it can better exploit the regularities of the spatial attributes.

This representation can return the value at a given cell by binary searching the collection of k^2-trees, and can return very efficiently those cells within a given range of values, as it only needs to access two different k^2-trees. Obtaining the cells that contain a specific value also requires the traversal of two k^2-trees.

Compared with the previous approach, k^2-acc obtains in general better time performance for retrieving cells containing a specific value or range of values, whereas k^3-tree obtains better space results and better time results when retrieving the value in a given position.

4. OUR PROPOSAL: k^2-*raster*

In this section we describe the k^2-*raster*. It is a new technique for representing raster datasets that uses compressed space and offers indexing capabilities, thus, improving query times over the raster data.

Let M be a raster matrix of size $n \times n$, being n a power of k, containing values $v \geq 0$ for each cell M_{ij}.[1] The k^2-*raster* recursively partitions the matrix M into k^2 submatrices, analogous to the original k^2-tree, and builds a tree representing this recursive subdivision. In addition, the representation is coupled with an efficient representation of the maximum and minimum values of each submatrix, which are needed for the representation of the raster data, but also provide the indexing functionality. Using the min/max values for indexing and improving query performance has already been used in the past [13, 8, 23], but our proposal efficiently represents this information, obtaining compressed spaces and avoiding the need of representing the original raster matrix, making our approach a self-indexed representation.

4.1 Construction and data structures

The process of construction the k^2-*raster* from the raster matrix is as follows. The root node of the tree stores the minimum and maximum values $(rMin, rMax)$ of the complete matrix, which is then divided into k^2 submatrices that are added as child nodes to the parent, in this case to the root node. For each node, the algorithm proceeds differently depending on the maximum and minimum values of its corresponding submatrix. If these two values are equal, i.e., all

[1] In case that the input matrix is of size $n \times m$, being n and m any integer, we conceptually extend the input matrix to the right and to the bottom with 0s, making it of width and height $n' = k^h = k^{\lceil \log_k \max\{n,m\} \rceil}$, that is, we round n and m up to the next power of k of their maximum value. This does not cause a significant overhead because our technique is efficient to handle large areas of equal values.

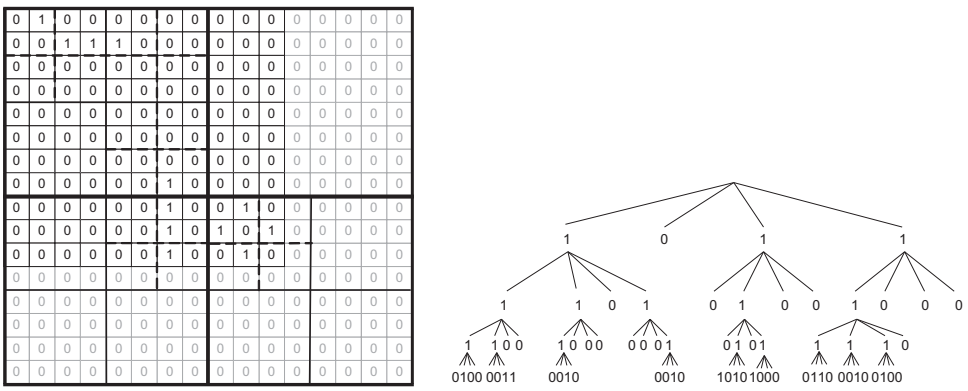


Figure 1: Example of binary matrix(left) and resulting k^2-tree representation (right), with $k = 2$.

cells in the submatrix have the same value, only the maximum value is stored and the subdivision of that branch ends; otherwise, both values (maximum and minimum) are stored and the submatrix is recursively decomposed into k^2 submatrices. The process goes on until the last level is reached, which corresponds to the case where a submatrix is composed of just a single value, or when the maximum and minimum values of a node are equal.

Let illustrate our representation with an example. Under the label "Step 1" of Figure 2, we can see an 8×8 raster matrix, and below it, the corresponding k^2-*raster* using $k = 2$. The root node is built with the maximum and minimum values of the complete matrix. Next, the complete raster is subdivided in 4 regions of 4×4 cells (given that the original raster is 8×8). Under the label "Step 2", we can see those subdivisions and their maximum (marked in dark grey) and the minimum (marked in light grey) values. For each subdivision, we add one child node to the root node of the tree, storing those values. Observe that in the case of the bottom right subdivision, the maximum and the minimum values are the same number (2), therefore such node becomes a leaf node and it is not further subdivided. The three submatrices corresponding to the first three nodes are then subdivided into another 4 submatrices, shown under the label "Step 3", each with its maximum and minimum values highlighted. At level 2 of the tree, we can see the corresponding nodes to each subdivision: those having the maximum value equal to the minimum value are leaf nodes, and the rest are further subdivided. However, at level 3 we reach the cell level of the raster, and then we store the values of the raster. That is, for each subdivision that reaches this level, we add a child node for each of its cells, storing the value in the cell.

The previous description corresponds to the conceptual representation of the k^2-*raster*, but we use succinct data structures to obtain compression. More specifically, we represent the tree and the maximum and minimum values as follows:

- The **tree** representation is similar to that of a k^2-tree. In contrast to the original k^2-tree, a 0 in the tree of a k^2-*raster* means that all values in the corresponding submatrix are equal, and a 1 means that there are two or more different values. Furthermore, the original k^2-tree is divided into two bitmaps T and L, where L represents nodes of last level and T the rest of nodes of the tree. However, in our case, the bitmap L is not necessary because it would always be 0, since the maximum and minimum value of a leaf node will always be equal and be represented by a 0.

- The nodes of the tree must store the **maximum values** of their corresponding submatrix. These values are encoded as the difference with the maximum value of their parent nodes. Notice that we will never get a negative value because the maximum value of a parent node is always equal or greater than the maximum value of its children. The final result is a list of integer sequences, one for each level of the tree, and all but the maximum value of the root level are encoded using DACs. Since the maximum values are encoded as differences, which tend to be small, DACs provide efficient random access to any position of the sequence in little space. The maximum value of the root ($rMax$) is stored as an integer in plain form. Assuming that the tree has l levels, we obtain l sequences of maximum values. We can concatenate all the lists in a unique sequence denoted $Lmax$.

- The construction of the structure for the **minimum values** uses the same technique as for the maximum values, except that the encoding to each value is the difference between its value and the minimum value of the parent node. Again, it will not have negative values because the minimum value of the node is always equal or greater than the minimum value of the parent node.[2] The minimum value of the root ($rMin$) is stored as an integer in plain form. Since we will not store minimum values for the leaf nodes, as the maximum value is enough to represent them, for a tree of l levels, we obtain $l - 1$ sequences of minimum values. We can concatenate all the lists in a unique sequence denoted $Lmin$.

[2]The minimum value of a node could also be represented as a difference with respect to the maximum value of that node. In fact, since only difference greater than zero are represented, we could subtract 1 to this difference value. This variant has also been proved experimentally and it obtained comparable results.

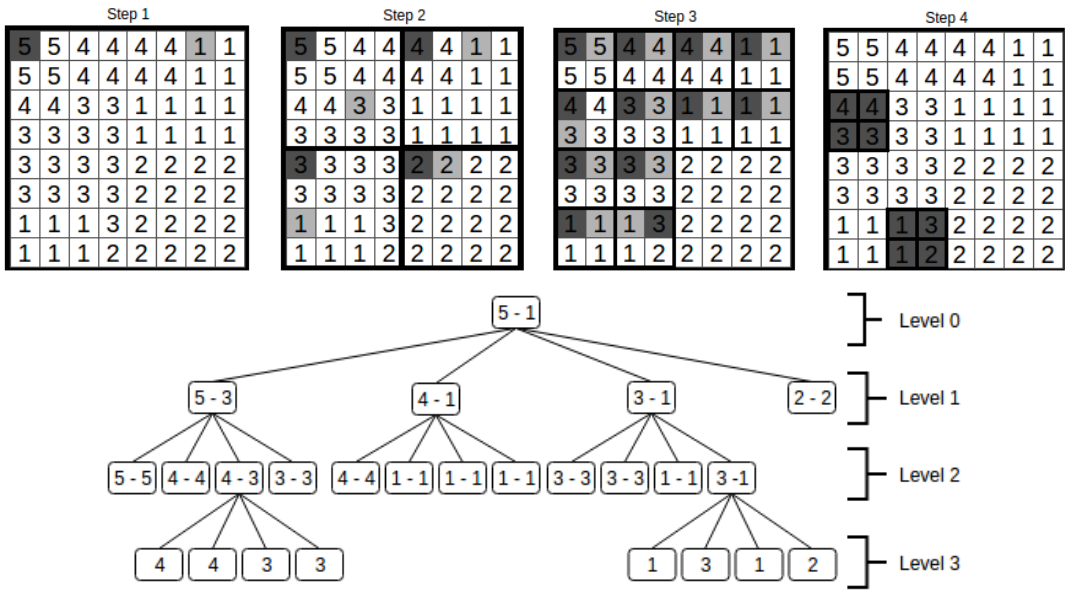


Figure 2: Example of raster matrix (top). We indicate the minimum (light grey) and maximum (dark grey) value of each submatrix for the four steps of the recursive subdivision of the construction algorithm. Conceptual tree representation obtained from the construction of the k^2-*raster* (bottom). Numbers at each node indicate the maximum and minimum value of its corresponding submatrix. In the last level, only the maximum is shown.

Notice that $Lmax$ will have as many elements as T, as each node of the tree stores its maximum value, plus all the values required for the last level of the tree (which is represented in $Lmax$ but not in T). $Lmin$ only contains values for those internal nodes z with $T[z] = 1$, as those nodes with $T[z] = 0$ has a minimum value equal to its maximum value, and it is already stored in $Lmax$. It is straightforward to obtain the position of the value in $Lmax$ corresponding to a node z: it is at position z of $Lmax$. Obtaining the position of its corresponding value in $Lmin$ is also possible by using $rank$ operation over T: it is at position $rank(T, z)$.

Figure 3 shows the final representation of the k^2-*raster* corresponding to the conceptual tree shown in Figure 2. The maximum and minimum values stored at each node are now encoded using differences with the values of its parent. In addition, when the maximum and minimum value are equal, only the maximum value is stored. Using differences instead of the actual values causes that the final sequence of integers to encode is mostly composed of small numbers (assuming some uniformity among the values of the input raster matrix), and this will be exploited by DACs encoding.

Construction

The construction of our representation can be easily done using a bottom-up recursive procedure.

The algorithm we present here consists in a depth-first traversal of the tree that outputs the bit array of the tree representation T_ℓ and the lists of maximum and minimum values for each node, which we will call $Vmax_\ell$ and $Vmin_\ell$, separately for each level ℓ of the tree. The total time of the algorithm is linear in the number of cells in the matrix, that is, $O(nm)$. In fact, it is optimal, as it processes the whole raster accessing each cell only once.

The algorithm proceeds as follows. When it reaches the last level, it reads the k^2 corresponding matrix cells. If they are all equal, it just return that value as maximum and minimum values; otherwise, it appends their values to $Vmax_\ell$, compute their maximum and minimum values and return them as result of the call.

In case of reaching an internal node, the algorithm proceeds with k^2 recursive calls, one for each children. After each child returns its maximum and minimum values, if these values are different, it appends these values to $Vmax_\ell$ and $Vmin_\ell$ lists and sets up a 1 in the T_ℓ of that level. If the maximum and minimum values are equal, it appends the value to $Vmax_\ell$ and sets up a 0 in T_ℓ. The overall maximum and minimum for all the children is also computed. If they are equal, it means that all the children contain the same value, thus, we must undo the last operations we have done, as these nodes will not have a representation in our structure. This can be easily done by removing the last k^2 positions of T_ℓ and $Vmax_\ell$, or just moving the pointer that indicates their last written position, k^2 positions backwards. Finally, the algorithm returns the maximum and minimum values to its parent.

Algorithm 1 shows the pseudocode of the construction process. It is invoked as **Build**$(n, 1, 0, 0)$, where the first parameter is the (possibly extended) raster matrix size, the second is the current level, the third is the row offset of the current submatrix, and the fourth is its column offset. It assumes that k, T_ℓ, $Vmax_\ell$, and $Vmin_\ell$ are global variables, and that T_ℓ, $Vmax_\ell$, and $Vmin_\ell$ have been initialized as empty sequences. In addition, there exist global variables $pmax_\ell$ and $pmin_\ell$ initialized to zero that are used to know the last written position of $Vmax_\ell$ and $Vmin_\ell$ re-

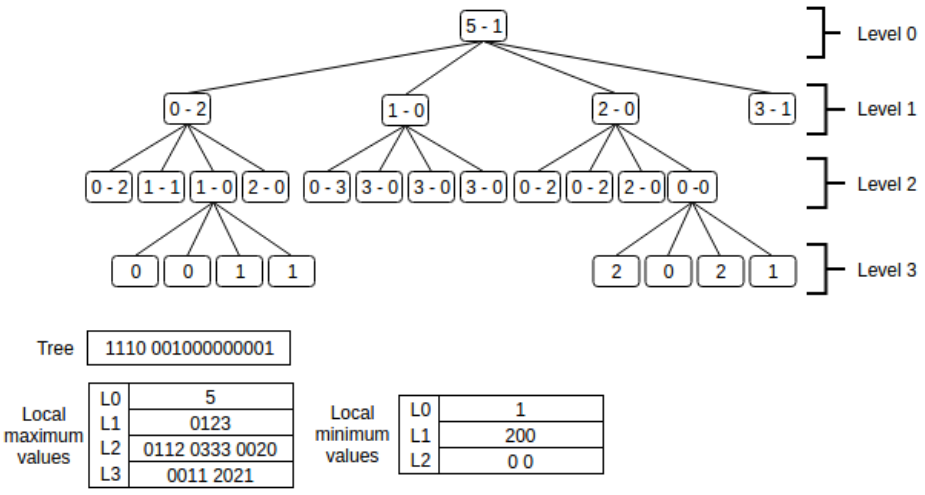


Figure 3: Compact representation of the conceptual k^2-*raster* using differences for the maximum and minimum values (top). Data structures T, $Lmax$ and $Lmin$ used for representing compactly the k^2-*raster* (bottom).

spectively. After running this algorithm, we must append all T_ℓ, $Vmax_\ell$, and $Vmin_\ell$, and convert $Vmax$ and $Vmin$ to $Lmax$ and $Lmin$ by computing the differences and encoding with DACs. Notice that the algorithm returns the maximum and minimum values of the input matrix, that is, $rMax$ and $rMin$, that must be represented in plain form.

4.2 Query algorithms

Obtaining a cell value

To access the value of a given position, the k^2-*raster* performs a top-down traversal of the tree, guided by the quadrant where the cell is located at each level. While going down in the tree, we must decode the maximum values stored for the traversed nodes and subtracting them from the maximum value stored at the root node.

Algorithm 2 shows the pseudocode for this query. To obtain the value at position (x, y) of the raster matrix, that is M_{xy}, it is invoked as **getCell**$(n, x, y, -1, rMax)$. It consists in a recursive procedure whose parameters are: current submatrix size, row of interest in the current submatrix, column of interest in the current submatrix, and the position in T of the node to process (the initial -1 is an artifact because T does not represent the root node). Values T, $Lmax$, and k are global. It is assumed that $rank(T, -1) = 0$ and $rMax$ is the maximum value of the raster matrix, which is not represented in $Lmax$.

The worst-case navigation time to get a value of a cell is $O(\log_k n \cdot L)$, that is, a full traversal from the root node to the last level of the k^2-*raster* requiring to decode a value from $Lmax$ at each level. We use L to denote the number of levels used in DACs for representing $Lmax$, which depends on the largest number encoded in the sequence. The time will be lower when the queried cell is surrounded by cells with the same value.

To illustrate this query, we will obtain the value at position $(6, 3)$. In this case, we invoke the algorithm with the parameters **getCell**$(8, 6, 3, -1, 5)$, we compute the node corresponding with the submatrix that contains this cell at level 1 of the tree. This is $z \leftarrow rank(T, -1) \cdot 4 + 6/4 \cdot 2 + 3/4 = 2$. We obtain $val \leftarrow accessDACs(Lmax, 2) = 2$, $maxval \leftarrow 5 - 2 = 3$, and, since $z = 2 < |T| = 16$, we recursively invoke **getCell**$(8/2, 6 \bmod 4, 3 \bmod 4, 2, 3) =$ **getCell**$(4, 2, 3, 2, 3)$. We repeat the procedure in the next level: $z \leftarrow rank(T, 2) \cdot 4 + 2/2 \cdot 2 + 3/2 = 12 + 2 + 1 = 15$, $val \leftarrow accessDACs(Lmax, 15) = 0$, $maxval \leftarrow 3 - 0 = 3$, and, since $z = 15 < |T|$, we recursively invoke **getCell**$(4/2, 2 \bmod 2, 3 \bmod 2, 15, 3) =$ **getCell**$(2, 0, 1, 15, 3)$. Finally, we obtain $z \leftarrow rank(T, 15) \cdot 4 + 0/1 \cdot 2 + 1/1 = 21$, $val \leftarrow accessDACs(Lmax, 21) = 0$, $maxval \leftarrow 3 - 0 = 3$, and, since $z = 21 \geq |T| = 16$ we return 3, which is the content of cell $(6, 3)$.

If instead we want to retrieve the value of position $(7, 7)$, we invoke the algorithm as **getCell**$(8, 7, 7, -1, 5)$. We compute $z \leftarrow rank(T, -1) \cdot 4 + 7/4 \cdot 2 + 7/4 = 3$, we decode $val \leftarrow accessDACs(Lmax, 3) = 3$, $maxval \leftarrow 5 - 3 = 2$, and, since $T[3] = 0$ we just return 2, which is the content of cell $(7, 7)$.

Retrieving cells with a given value or range of values

The k^2-*raster* can also retrieve the cells of the raster matrix that contain a given value or whose value is within a given range by efficiently traversing the tree from the root node down to the last level. In addition, this query can be restricted to a particular window from the raster matrix, that is, instead of querying the whole matrix, we can search only inside a specific rectangle $[x_1, x_2] \times [y_1, y_2]$.

Lets describe how to obtain all cells within the region $[x_1, x_2] \times [y_1, y_2]$ that contain values in the range $[v_b, v_e]$. Notice that if we are interested in just a particular value v, this is a particular case where the given range is $[v, v]$.

To know whether a cell or region of cells contains a value within a specific range we combine the functionality of the original k^2-trees to solve range queries, that is, to obtain cells with 1s within a given rectangle, with the indexing capabilities offered by our k^2-*rasters*, thanks to the storage of the maximum and minimum values at the nodes of the tree. Thus, we perform a top-down traversal in the tree through the branches corresponding with submatrices over-

Algorithm 1: Build(n, ℓ, x, y) computes T, $Vmax$ and $Vmin$ of the k^2-*raster* representation from matrix M and returns $(rMax, rMin)$

```
minval ← ∞
maxval ← 0
for i ← 0...k−1 do
    for j ← 0...k−1 do
        if ℓ = ⌈log_k n⌉ then /* last level */
            if minval > M_{x+i,y+j} then
                minval ← M_{x+i,y+j}
            end
            if maxval < M_{x+i,y+j} then
                maxval ← M_{x+i,y+j}
            end
            Vmax_ℓ[pmax_ℓ] ← M_{x+i,y+j}
            pmax_ℓ ← pmax_ℓ + 1
        else /* internal node */
            (childmax, childmin) ←
            Build(n/k, ℓ+1, x+i·(n/k), y+j·(n/k))
            Vmax_ℓ[pmax_ℓ] ← childmax
            if maxval <> minval then
                Vmin_ℓ[pmin_ℓ] ← childmin
                pmin_ℓ ← pmin_ℓ + 1
                T_ℓ[pmax_ℓ] ← 1
            end
            pmax_ℓ ← pmax_ℓ + 1
            if minval > childmin then
                minval ← childmin
            end
            if maxval < childmax then
                maxval ← childmax
            end
        end
    end
end
if minval = maxval then
    pmax_ℓ ← pmax_ℓ − k^2
end
return (maxval, minval)
```

lapping with the window, disregarding those that fall outside the range of values sought and returning those regions completely contained inside the region and whose min/max values are completely contained within the given range.

Algorithm 3 shows the pseudocode for this query. It is again a recursive procedure and, in case we want to obtain all the cells containing values in the range $[v_b, v_e]$ inside the window $[x_1, x_2] \times [y_1, y_2]$, the algorithm invoked as **searchValuesInWindow**$(x_1, x_2, y_1, y_2, rMax, rMin, -1)$. For this algorithm, k, T, $Lmax$, $Lmin$, v_b, and v_e are considered global variables.

The k^2-*raster* also allows for other efficient queries, such as retrieving all the values of a region of the raster, obtaining the maximum value or the minimum values of region, determining the existence of cells with a value in a region of the raster (which can be answered faster than searchValueInWindow as it does not require obtaining the actual position of that cell), etc.

4.3 Hybrid variant

Algorithm 2: getCell$(n, x, y, z, maxval)$ returns the value at cell (x, y)

```
z ← rank(T, z) · k^2
z ← z + ⌊x/(n/k)⌋ · k + ⌊y/(n/k)⌋
val ← accessDACs(Lmax, z)
maxval ← maxval − val
if z ≥ |T| or T[z] = 0 then /* leaf */
    return maxval
else /* internal node */
    return
    getCell(n/k, x mod (n/k), y mod (n/k), z, maxval)
end
```

The main goal of our compact data structure is to improve both space and query times. Focusing on efficiency, we implement an optimization that allows us to significantly reduce the time of some queries with the cost of slightly increasing the space requirements of the structure.

In some queries, like getting an individual datum, our method needs to descend the tree until reaching the requested data, therefore, query time increases with the height of the tree. Therefore, we propose a new design that allows us to modify how the matrix is partitioned during the first levels of tree. That is, instead of using a unique k value for partitioning all levels, we use two different values k_1 and k_2; k_1 for the first levels, and k_2 for the rest. The goal is to reduce the total number of levels by performing more subdivisions in the first levels.

More precisely, given the parameters k_1, k_2 and the number of levels n_1, the construction process is as follows: for the first n_1 levels, each submatrix is partitioned into k_1^2 submatrices and for levels $n_1 + 1$ until the leaf nodes is divided into k_2^2 submatrices. We call this version hybrid k^2-$raster$, k^2-$raster_H$ for short.

Figure 4 shows an example of a k^2-$raster_H$ built with $k_1 = 4$, $k_2 = 2$, and $n_1 = 1$. The process is exactly the same, the difference is that in the first level (given that $n_1 = 1$) the matrix is divided into $k_1^2 = 16$ submatrices. As in the normal version, each submatrix adds a child node to the root node informing about the maximum and minimum value in that submatrix. Therefore, we can see in Figure 4 that the root node has 16 children. The second level uses $k = 2$, and thus each submatrix is divided into 4 submatrices again, that in this case are individual cells. The effect of increasing k is that the tree becomes wider and smaller, and thus, it causes faster top-down traversals.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Framework

To test the efficiency of our structure, we run a set of experiments comparing k^2-*raster* against the two previous compact data structures for raster datasets: k^2-acc and k^3-tree. We check the space consumption and the time to answer the two queries shown in Section 4.2.

getCell illustrates the impact on the time to access and recover the original information when we represent the raster with each of the techniques, since they keep the information compressed. *searchValuesInWindow* illustrates the indexing capabilities of each representation.

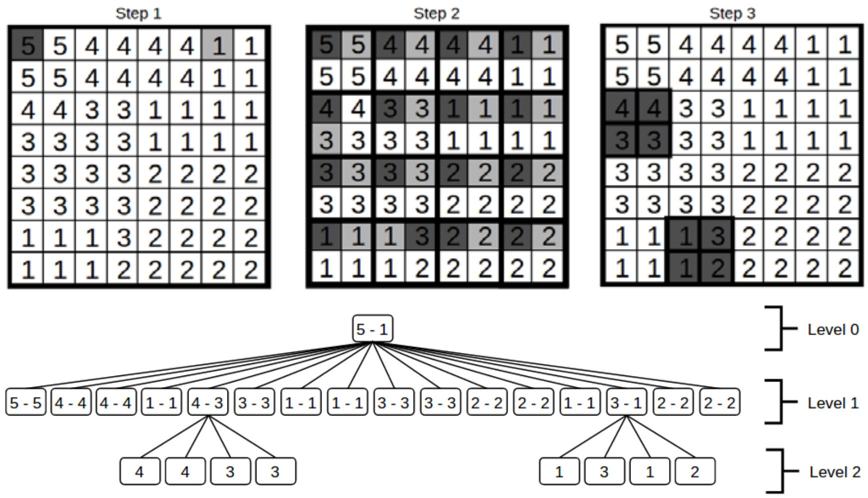


Figure 4: Example of using different k values. We indicate the minimum (light grey) and maximum (dark grey) value of each submatrix for the three steps of the recursive subdivision of the construction algorithm (top). Conceptual tree representation obtained from the construction of the k^2-*raster* with $k_1 = 4$, $k_2 = 2$ and $n_1 = 1$ (bottom).

In the case of getCell queries, we measure the average time result for 100,000 queries, choosing random positions of the input matrix. For searchValuesInWindow queries, we measure the average time result of 10,000 queries, choosing random input rectangles contained in the input matrix and random values for the range boundaries, limited by the number of different values of the matrix.

All the experiments were run on an isolated Intel®Xeon®-E5520@2.26GHz with 72 GB DDR3@800 MHz RAM. It ran Ubuntu 9.10 (kernel 2.6.31-19-server), using gcc version 4.4.1 with `-O9` options. Time results refer to CPU user time.

We use two different configurations for k^2-*raster*: one uses $k = 2$ for the whole tree, whereas the other uses the hybrid approach, with $k_1 = 4, k_2 = 2, n_1 = 4$. This last version will be denoted as k^2-$raster_H$. Both of them use an implementation for supporting *rank* operations that uses 5% of extra space on top of the bit sequence T and provides fast queries [9][3]. In addition, $Lmax$ and $Lmin$ are encoded using the version of DACs that optimizes the space usage while restricting the maximum number of levels. More concretely, we have limited the number of levels to 3. We will compare both variants of our proposal with k^2-acc and k^3-tree with the same configurations as in the original paper [7].

In the experimental evaluation we use real data obtained from the Spanish Geographic Institute[4] (SGI). More specifically, we use several DTM (Digital Terrain Model) data files that contain the spatial elevation data of the terrain from some regions of the country, stored as rectangular equal-spaced grids, called tiles (with 5 meters resolution). Each cell of a tile has a real number of at most three decimal digits. In the experiments we use different DTM files as input raster matrices with different sizes (by considering adjacent tiles, that is, tiles that represent adjacent regions of the terrain) and varying the number of different unique values (by considering different precisions of the spatial elevation values, that is, using different number of decimal digits).

5.2 Analyzing the behavior of k^2-*raster*

Table 1 shows the main properties of the dataset used in this section. We considered several raster matrices of different sizes by joining several tiles. Specifically, 1×1 is built from 1 tile, 2×2 is a raster made up using 2×2 adjacent tiles, and so on. The subscript indicates how many digits of the decimal digits were considered. By considering more or less, we increase or decrease, respectively, the number of different values existing in the raster matrix. Experiments over these datasets were run only with k^2-*raster* since in most cases k^2-acc and k^3-tree did not run.[5] Thus, we devote this section to analyze the behavior of our approach when varying the size and the number of different values of the input matrix.

The results of the experiment are shown in Figure 5. In the x-axis we show the size of the raster, and the curves in the plot correspond to four different discretized versions of the raster, i.e., we generate different input matrices by taking into consideration 0 to 3 decimal digits of the original raster values. We only display the values of the normal k^2-*raster* since the hybrid version obtain similar results. As explained, this experiment could not be run with the rest of techniques.

As expected, when considering more decimal digits, there are more different values, and thus the compression is worse.[6]

[3]If more space and less time is desired, one could replace the bitmap by another that that uses 37.5% extra space and is much faster.

[4]http://www.ign.es

[5]The method k^2-acc can complete the construction of large datasets but it requires a unaffordable time consumption, such as days, and the representation obtained is larger than the original matrix in plain form

[6]Compression refers to the size of the compressed dataset as

Algorithm 3: searchValuesInWindow$(n, x_1, x_2, y_1, y_2, maxval, minval, z)$ returns all cells from region $[x_1, x_2]$ to $[y_1, y_2]$ containing values within $[v_b, v_e]$

```
z ← rank(T, z) · k^2
for i ← ⌊x1/(n/k)⌋ ... ⌊x2/(n/k)⌋ do
    if i ← ⌊x1/(n/k)⌋ then x'1 ← x1 mod (n/k)
    else x'1 ← 0
    if i ← ⌊x2/(n/k)⌋ then x'2 ← x2 mod (n/k)
    else x'2 ← (n/k) − 1
    for j ← ⌊y1/(n/k)⌋ ... ⌊y2/(n/k)⌋ do
        if j = ⌊y1/(n/k)⌋ then y'1 ← y1 mod (n/k)
        else y'1 ← 0
        if j = ⌊y2/(n/k)⌋ then y'2 ← y2 mod (n/k)
        else y'2 ← (n/k) − 1
        z ← z + k · i + j
        maxval ← maxval − accessDACs(Lmax, z)
        if z ≥ |T| or T[z] = 0 then /* leaf */
            minval ← maxval
            if minval ≥ vb and maxval ≤ ve then
                output corresponding region of cells
                return /* all cells meet the
                condition in this branch */
            end
        else /* internal node */
            minval ←
            minval + accessDACs(Lmin, rank(T, z))
            if minval ≥ vb and maxval ≤ ve then
                output corresponding region of cells
                return /* all cells meet the
                condition in this branch */
            end
            if minval > ve or maxval < vb then
                return /* no cells meet the
                condition in this branch */
            end
            if minval < vb or maxval > ve then
                searchValuesInWindow(n/k, x'1, x'2,
                y'1, y'2, maxval, minval, z)
            end
        end
    end
end
```

What is most significant is that if we fix the number of different values, the technique obtains similar results independently of the size of the matrix. This property is of interest for applying over real datasets, which tend to be large.

Figure 6 shows the times obtained for getCell and searchValuesInWindow queries when varying the size and number of different values of the input raster. Figure 6(a) shows that times increase for cell queries as the size of the collection grows, since obtaining the value of a cell from a larger raster requires traversing a higher number of levels. In addition, times are also slower when the number of different values in the raster is larger.

Figure 6(b) shows the times obtained for searchValuesInWindow queries. In this case, we limited the size of the rectangles to 500×500 cells. We have done this because if we used completely free random limits for the query rectangles, as the size of the dataset grows, the size of the query rectangles would increase in the same level. Thus, with larger query rectangles, computing the answer would be more costly, but not due to the size of the raster, but due to the size of the query rectangle.

Results shows that window queries are faster when the dataset is larger. This can be caused by the fact that as we have a higher number of different values, the possibility that a submatrix of the raster contains values from the range sought is lower, and this can be detected in an upper level of the tree thanks to the max/min values stored at the nodes. Thus, the final time is lower than for smaller datasets, which output a higher number of cells meeting the criteria.

a percentage of the original dataset.

Table 1: Datasets used for analyzing the performance of k^2-*raster* when varying the size and number of different values of the input matrix. The rasters have been obtained by joining adjacent tiles.

Name	#rows	#cols	# different values
`raster1`×1_0	4,201	5,841	714
`raster2`×2_0	8,402	11,682	1,015
`raster3`×3_0	12,603	17,523	1,180
`raster1`×1_1	4,201	5,841	7,134
`raster2`×2_1	8,402	11,682	10,149
`raster3`×3_1	12,603	17,523	11,800
`raster1`×1_2	4,201	5,841	69,868
`raster2`×2_2	8,402	11,682	101,239
`raster3`×3_2	12,603	17,523	117,659
`raster1`×1_3	4,201	5,841	632,493
`raster2`×2_3	8,402	11,682	989,433
`raster3`×3_3	12,603	17,523	1,115,408

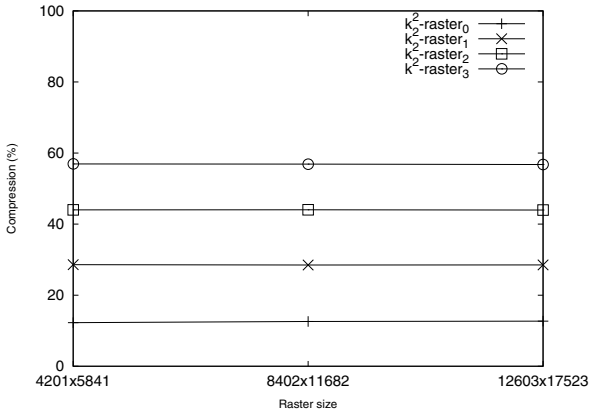
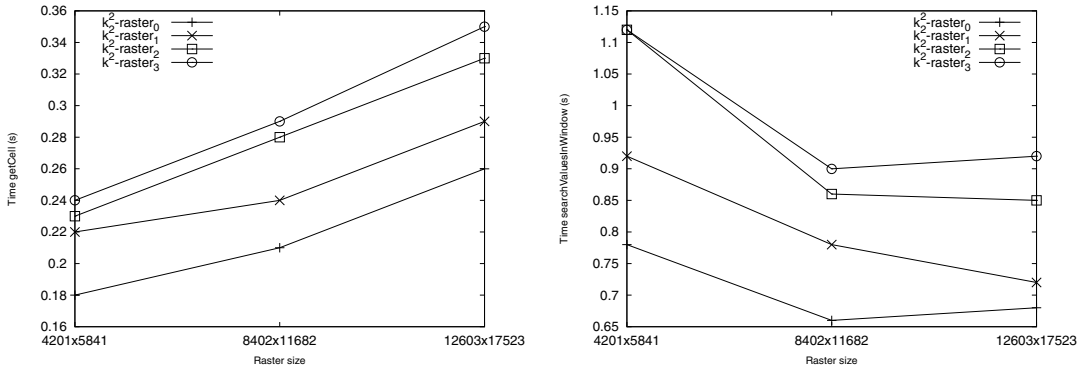


Figure 5: Compression obtained by k^2-*raster* when varying the size and the number of different values of the input rasters. The curves represents the values for different discretization levels.

5.3 Comparison with related work

For comparing k^2-*raster* with k^2-acc and k^3-tree we have generated a collection of raster matrices from just one tile, namely that denoted as `MDT05-0533-H30-LIDAR`. More concretely, we have first truncated the original values by taking



(a) getCell. (b) searchValuesInWindow.

Figure 6: Time performance of k^2-*raster* for cell (left) and window (right) queries when varying the size and number of different values of the input raster.

Table 2: Datasets used for comparing k^2-*raster* with related work. They have been generated from one specific tile, MDT05-0533-H30-LIDAR, to obtain raster matrices with different number of values.

Name	#rows	#cols	# different values
MDT05-0533-H30-LIDAR$_{\gg 9}$	3,881	5,841	227
MDT05-0533-H30-LIDAR$_{\gg 7}$	3,881	5,841	903
MDT05-0533-H30-LIDAR$_{\gg 5}$	3,881	5,841	3,606
MDT05-0533-H30-LIDAR$_{\gg 3}$	3,881	5,841	14,415
MDT05-0533-H30-LIDAR$_{\gg 1}$	3,881	5,841	57,586
MDT05-0533-H30-LIDAR	3,881	5,841	114,966

only the two most significant decimal digits. Then we have created other 5 raster matrices MDT05-0533-H30-LIDAR$_{\gg x}$ by shifting x bits of the value of each cell, for $x = 1, 3, 5, 7, 9$. By doing this, we have generated a collection of matrices with the same size and different number of different values.[7] We have not used the original values with all their precision due to the problems of k^2-acc and k^3-tree for running over datasets with a high number of different values. Table 2 shows the properties of these datasets.

Figure 7 shows the compression with the six datasets of Table 2. We can only create the representations of k^2-acc and k^3-tree for the first four datasets of the table (from top to bottom).

The x-axis shows the number of different values in the dataset. With 227 different values, compression ratios are around 3% of the original collection, obtaining all techniques a similar result. When the dataset reaches around 1,000 different values, k^3-tree and the k^2-*raster* obtain compression ratios around 9%, whilst k^2-acc starts to yield worse values, around 11%. With the third dataset, which has 3603 different values, k^2-acc already obtains a much worse compression ratio, around 44%. k^3-tree obtains a slightly worse compression than k^2-*raster*, namely a 23% versus 19%. This tendency continues with the fourth dataset, but now k^2-acc does not compress at all, occupying more space than the original collection (177%). k^3-tree obtains a 32% of compression and k^2-*raster* continues with a little improvement (29%). As explained, for the last two datasets, we only have values for k^2-*raster*. We can see that the worsening in compression ratio has a very gentle slope, taking into account that the x-axis scale is logarithmic, and as the number of different values grows the distribution is less skewed (given that the size of the raster is always the same), and thus obtaining compression is more difficult.

[7]Notice that by shifting x bit each value is divided by 2^x, thus decreasing the number of different values in the raster.

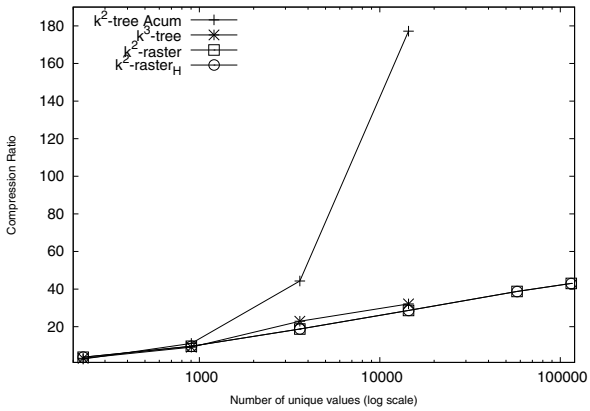
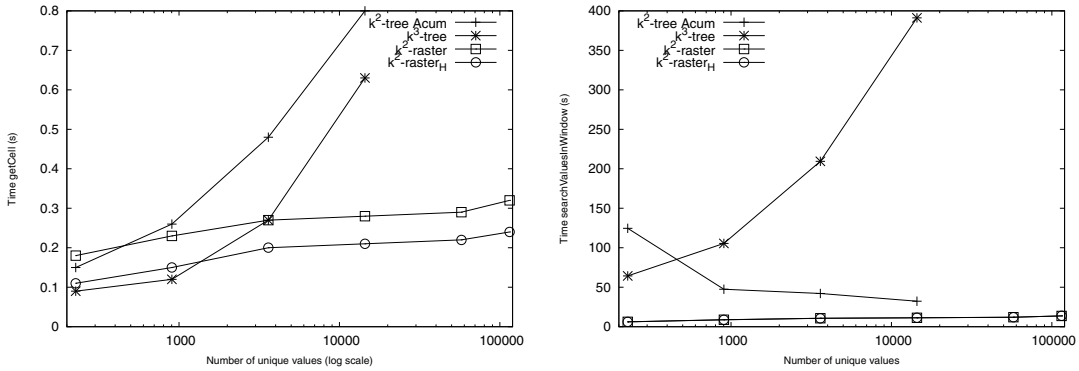


Figure 7: Comparison of the compression obtained (in %) by different methods when varying the number of different values of the input matrix.

Figure 8(a) shows the time to get the value of a given cell using the datasets of Table 2. When the raster has 1,000 or fewer different values, the winner is k^3-tree; but when that number increases, the k^2-*raster* becomes the best option. For this query, the hybrid version of k^2-*raster* is always between 30-60% faster than the normal version. Indeed the normal version is slower than k^2-acc in the first dataset, and it is on a par with k^3-tree in the third dataset. However, in the fourth dataset, with 14,415 different values, the hybrid k^2-*raster* is 3.8 times faster than k^2-acc and 3 times faster than k^3-tree. Again the slope of the curve of the k^2-*raster* is very smooth, even when processing datasets with much more different values (recall that the x axis has a logarithmic scale).



(a) Time to get a cell value.

(b) Window query.

Figure 8: Time as the number of different values grows.

Figure 8(b) shows the time to solve the searchValuesInWindow query. In this query, the k^2-*raster* is the clear winner in all datasets by far. The k^3-tree is faster than k^2-acc only in the first dataset, increasing the times very fast as the number of different values grows. k^2-acc has just the opposite behavior. The reason is the following. To solve this query, k^2-acc accesses two k^2-trees, those representing the extremes of the checked range. The time is affected if one or both boundaries are represented by non-sparse k^2-trees. In that case, the traversal of the tree must continue until reaching the leaves at the deepest level, since few nodes are leaves at a level before the maximum depth. The problem can be attenuated if one of the boundaries is sparse, but if both suffer this problem, the times can be affected. The sparser k^2-trees are those at the beginning of the range of possible values, as the 0s will dominate the raster, and those at the end of the range of possible values, as the 1s will dominate the raster, whereas the values in the middle show a mixture of 0s and 1s that will harm search times. When the range of possible values is small, there are more chances of processing trees that are not sparse, whereas as the number of possible values grows, the amount of trees which are sparse increase. In any case, in this experiment, the k^2-*raster* is between 3 and 20 times faster than k^2-acc and between 10 and 34 times faster than the k^3-tree. Indeed, in this experiment, the scenario of few different values is the best one for the k^2-*raster*. In this query, there are no differences between the two versions of the k^2-*raster*.

5.4 Discussion

In the datasets where we could run experiments for all techniques, we can see that the k^2-acc can consume up to 6 times more space than k^2-*raster*, and indeed, in that case, it occupies more space than the original dataset. The space consumption of the k^3-tree was always close to that of k^2-*raster*, only at most 12% worse.

The time to get a cell value also yields a bad result for the k^2-acc, as it is up to 3.8 times slower than k^2-*raster*. k^3-tree shows a slightly better behavior, but still it is up to 3 times slower than k^2-*raster*. In the searchValuesInWindow query, k^2-acc obtains better results (but at the expense of occupying even more space than the original dataset), though it is between 3 and 20 times slower than k^2-*raster*. This query is disastrous for the k^3-tree, as it is up to 34 times slower than k^2-*raster*.

For our experiments we have used raster matrices extracted from some digital elevation model, which represents the terrain's surface of some part of the planet. We have randomly chosen one tile from one region from Spain to illustrate graphically the space/time results obtained, more concretely the one named `MDT05-0533-H30-LIDAR`, but all the tiles included in the same database showed a similar behavior.

Notice that k^2-*raster* obtains an excellent performance for this kind of raster dataset, that is, when there are large zones of cells containing the same value, and the differences between the values of adjacent cells is generally small. Large zones of cells with the same value are represented by k^2-*raster* with just one bit in an upper level of the tree representation, obtaining very compact spaces and excellent time performance, as the queries can be answered quickly without descending many levels in the tree. Small differences among the cell values of adjacent cells make possible a compact representation of the minimum and maximum values, which also impacts both space and time results. k^2-*raster* is not designed for random raster matrices: it would not obtain any compression, as T would be a complete k-ary tree and we would need to store all the values of the original cells as part of $Lmax$. It would have some indexing capabilities due to the storage of the minimum and maximum values at each node of the tree, but times would also be degraded as we would need to access the last level of the tree for most of the queries.

6. CONCLUSIONS

We have presented k^2-*raster*, a new compact data structure that represents raster data in compressed space with indexing capabilities. Our technique supports, within reduced space, to efficiently return the original value at any cell of the raster and also supports advanced searches, such as retrieving cells containing some specific values restricting the search to any random region of the raster.

We compare our proposal with existing techniques from the literature. Our experiments show that k^2-*raster* clearly outperforms these previous approaches for real datasets. It obtains better space usage and query performance and, which is more importantly, it scales better when the datasets gets

larger or when it contains a higher number of different values. Previous techniques do not support well datasets with these properties, which typically appear when using real datasets.

As future work, we plan to develop more query types over our structure, in order to achieve a fully functional compressed data structure for raster data representation. In addition, we will study its performance over other raster datasets of different nature, as we believe that this proposal can be used in many different application domains, some of them requiring adaptations of the structure. We also explore the possibility of extending the structure to other dimensions, for instance, to be used for spatio-temporal or 3D datasets. In addition, we will also study the adaptation of our data structure to distributed environments.

7. ACKNOWLEDGMENTS

This work was supported by Ministerio de Economía y Competitividad (PGE and FEDER) under grants TIN2013-46238-C4-3-R, TIN2013-46801-C4-3-R, CDTI IDI-20141259, and CDTI ITC-20151247; Xunta de Galicia (co-founded with FEDER) under grant GRC2013/053.

8. REFERENCES

[1] S. Álvarez-García, N. Brisaboa, S. Ladra, and O. Pedreira. A compact representation of graph databases. In *Proceedings of the 8th Workshop on Mining and Learning with Graphs (MLG)*, pages 18–25, 2010.

[2] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, and G. Navarro. Compressed vertical partitioning for efficient RDF management:. *Knowledge and Information Systems*, 44(2):439–474, 2015.

[3] N. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management (IPM)*, 49(1):392–404, 2013.

[4] N. R. Brisaboa, S. Ladra, and G. Navarro. Compact representation of web graphs with extended functionality. *Information Systems*, 39(1):152–174, 2014.

[5] M. Burtscher and P. Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, 2009.

[6] G. Cheng, L. Liu, N. Jing, L. Chen, and W. Xiong. General-purpose optimization methods for parallelization of digital terrain analysis based on cellular automata. *Computers & Geosciences*, 45:57–67, 2012.

[7] G. de Bernardo, S. Álvarez-García, N. R. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 96–108, 2013.

[8] B. Duvenhage. Using an implicit min/max KD-tree for doing efficient terrain line of sight calculations. In *Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visuaisation and Interaction in Africa (AFRIGRAPH)*, pages 81–90, 2009.

[9] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings of the 4th International Workshop on Experimental Algorithms (WEA)*, pages 27–38, 2005.

[10] Q. Guan and K. C. Clarke. A general-purpose parallel raster processing programming library test application using a geographic cellular automata model. *International Journal of Geographical Information Science*, 24(5):695–722, 2010.

[11] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[12] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie-Mellon, Jan. 1989. Tech Rep CMU-CS-89-112.

[13] S. Y. Lee, T. W. Ling, and H. Li. Hierarchical compact cube for range-max queries. In *Proceedings of the 26th Internal Conference on Very Large Databases (VLDB)*, pages 232–241, 2000.

[14] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[15] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind. *Geographic Information Systems and Science*. Wiley, 2005.

[16] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 61 pages, 2007.

[17] C.-Z. Qin, L.-J. Zhan, A.-X. Zhu, and C. Zhou. A strategy for raster-based geocomputation under different parallel computing platforms. *International Journal of Geographical Information Science*, 28(11):2127–2144, 2014.

[18] R. Raman and S. S. Rao. Succinct representations of ordinal trees. In *Space-Efficient Data Structures, Streams, and Algorithms*, LNCS 8066, pages 319–332, 2013.

[19] M. Romero, N. Brisaboa, and M. A. Rodríguez. The SMO-index: a succinct moving object structure for timestamp and interval queries. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, pages 498–501, 2012.

[20] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2006.

[21] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34:31–44, Apr. 1991.

[22] M. F. Worboys and M. Duckham. *GIS: a computing perspective*. CRC press, 2004.

[23] J. Zhang and S. You. Supporting web-based visual exploration of large-scale raster geospatial data using binned min-max quadtree. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 379–396, 2010.