Scalable and Queryable Compressed Storage Structure for Raster Data[☆]

Susana Ladra, José R. Paramá, Fernando Silva-Coira[1]

Universidade da Coruña, Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain

Abstract

Compact data structures are storage structures that combine a compressed representation of the data and the access mechanisms for retrieving individual data without the need of decompressing from the beginning. The target is to be able to keep the data always compressed, even in main memory, given that the data can be processed directly in that form. With this approach, we obtain several benefits: we can load larger datasets in main memory, we can make a better usage of the memory hierarchy, and we can obtain bandwidth savings in a distributed computational scenario, without wasting time in compressing and decompressing data during data exchanges.

In this work, we follow a compact data structure approach to design a storage structure for raster data, which is commonly used to represent attributes of the space (temperatures, pressure, elevation measures, etc.) in geographical information systems. As it is common in compact data structures, our new technique is not only able to store and directly access compressed data, but also indexes its content, thereby accelerating the execution of queries.

Previous compact data structures designed to store raster data work well when the raster dataset has few different values. Nevertheless, when the number of different values in the raster increases, their space consumption and search performance degrade. Our experiments show that our storage structure improves previous approaches in all aspects, especially when the number of different values is large, which is critical when applying over real datasets. Compared with classical methods for storing rasters, namely netCDF, our method competes in space and excels in access and query times.

Keywords: Geographic information systems, raster datasets, data compression, indexing, query processing.

1. Introduction

Geographical information systems can use different data models to manage spatial information [2]. At the conceptual level, there are two possibilities: *object-based models*

[☆]A preliminary partial version of this paper was published in [1].

Email addresses: `susana.ladra@udc.es` (Susana Ladra), `jose.parama@udc.es` (José R. Paramá), `fernando.silva@udc.es` (Fernando Silva-Coira)

[1]Corresponding author. Tel. +34981167000 Fax. +34981167160.

and *field-based models* [3]. Object-based models consider a space containing discrete and identifiable entities, each with a geospatial position. In contrast, field-based models can be seen as a continuous mathematical function that for each position of the space returns a value. Typically, object-based models represent spaces containing buildings, roads, and other man-made objects. On the other hand, field-based models usually deal with images and physical properties such as land elevation, temperature, atmospheric pressure, etc. At the logical level, there are also two models: *vector models*, which represent the spatial information using points and line segments, and *raster models*, which consider the space as a regular tessellation of disjoint cells, usually squares, each having a value [4]. Any logical spatial model can be used to represent any conceptual spatial model, however, it is common to use vector models to represent object-based models and raster models for field-based models.

This paper deals with spatial information represented with a raster model. This involves images -including remotely sensed imagery-, engineering, modeling, representations of parameters of the land surface such as pollution levels, atmospheric pressure, rain precipitations, land elevation, vegetation indices, etc. Thanks to the advances in remote sensing and instrumentation, the amount and size of the rasters are increasing rapidly. For example, it has been estimated that each day, remotely sensed imagery is acquired at the rate of several terabytes per day [5], and the archived amount of raster data of this type is slowly approaching the zettabyte scale [6].

In this field, as usual, compression has been used to save space and bandwidth [7, 8, 9]. Long-established compression methods do not allow to process or query compressed data, requiring a previous decompression phase. However, a recent family of storage structures, called *compact data structures*, is changing the way in which compression has been traditionally used. Compact data structures combine in a unique storage structure a compressed representation of a dataset and the mechanisms that allow accessing any given datum without the need of decompressing the data from the beginning [10, 11]. The objective is to keep the data always compressed, even in main memory. In this way, in addition to the classical savings in disk space and bandwidth, we obtain several additional benefits: we can process larger datasets within the same memory, we can make a better usage of the memory hierarchy (including reducing costly disk accesses), and we can improve the performance when using parallel processing. Data interchanges between nodes in that scenario are a big issue since they can produce bottlenecks in the network. Compression has been used to reduce bandwidth consumption [12, 13]. However, data have to be compressed prior any data exchange and decompressed at the destination node, given that data cannot be processed in compressed form. Nevertheless, using compact data structures, we do not need those compression/decompression processes during data exchanges thanks to the ability to process compressed data.

Another advantage is that, in many cases, compact data structures provide some sort of indexation, which allows answering queries even faster than performing that query over the plain representation and within the same compressed space [14, 15, 11]. That is, this indexation is not provided by an auxiliary structure, and the index plus the data, kept in the same storage structure, occupy less space than the original data. This characteristic is usually called *self-indexation*.

There exists vast research focused on compressing raster datasets, proposing both lossless [16, 17, 13, 9] and lossy [7] approaches. In addition, there have been efforts in creating indexes on raster data to improve query and processing performance [18, 19, 20].

However, there is much less work on data structures capable of compressing and indexing data at the same time. The first exponent is the quadtree [21, 22], originally designed as a method to compress images, which allows the manipulation of the compressed image directly in main memory and, in addition, it spatially indexes the values of the raster. However, it does not provide indexation over the values of each cell of the raster. To the best of the authors' knowledge, only two recent compact data structures [23] were designed to represent raster datasets and achieved these three features: a compressed representation, a spatial indexation, and an indexation of the values of the cells. These techniques work well when the number of different values in the raster is low; however, if that number grows, both the space consumption and the query performance degrade dramatically. Our new storage structure scales much better when increasing the size of the input data or when the raster matrix increases its cardinality, that is, when the number of different values grows. Observe that this is an important problem when dealing with rasters, since they are usually obtained from a real continuous phenomenon as temperature, atmospheric pressure, etc.

This work presents a new storage structure designed following a compact data structure approach to represent raster datasets, called k^2-*raster*, and an improved version that we call *heuristic* k^2-*raster* (k^2_H-*raster*). They are based on the k^2-tree [24], a storage structure for representing binary matrices in little space, which can be regarded as a compact version of a quadtree. The basic ideas of k^2-*raster* were already presented in a preliminary work [1]. Here, we described it more in detail, including pseudocodes and examples for more queries. The k^2_H-*raster*, which was not proposed in the original paper, is a variant that significantly improves the spatio-temporal results. We use an entropy-based heuristical approach to compress the last level of the representation, which makes our solution the most space-efficient and scalable compressed and queryable representation up to date.

In this paper, we also enhance the experiments, including more queries. We ran the experiments over new raster matrices extracted from real datasets of different nature (temperatures and elevations). The basic k^2-*raster* overcomes in most parameters to previous approaches, and only when the number of different values is low, it can be on a par in some parameters, whereas the heuristic variant overcomes them in all aspects, even in the scenario of low number of different values.

We also include a comparison with netCDF [25], a classical method to store rasters. NetCDF includes the possibility of compressing the data with Deflate [26], and by using a simple API, transparently accessing the compressed data. k^2-*raster* obtains compression ratios close to those achieved by netCDF, but differences are not significant. However, k^2-*raster* clearly outperforms netCDF in access and query times, even in some cases when using the uncompressed version of netCDF files. Thanks to the indexing capabilities of k^2-*raster*, queries specifying conditions on the values of the raster are solved orders of magnitude faster than over uncompressed netCDF files.

The rest of the paper is structured as follows. Section 2 presents some related work. Section 3 describes the k^2-*raster* in detail, whereas Section 4 presents the k^2_H-*raster*. Section 5 presents our experimental study. Finally, Section 6 shows the conclusions and future work.

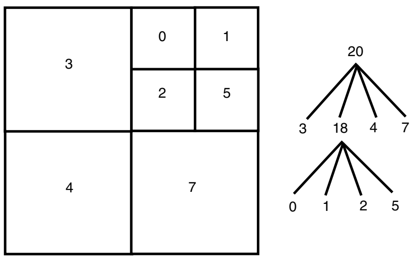


Figure 1: An image (left), where a number inside a square means that all pixels in that square have that value, and the corresponding conceptual quadtree showing the byte representation of each node using the Treecodes strategy.

2. Related work

2.1. Quadtress for raster representation or indexation

As we will see, the k^2-*raster* uses some ideas of the k^2-tree [24], which is a region quadtree for binary matrices built with the latest developments in the field of compact data structures. In particular, the k^2-*raster* uses the partition strategy of the region quadtree data structure. Thus, we present here some notions of quadtrees.

There are many different variants of the quadtree and with different purposes [18, 27], but the compression of images using region quadtrees was one of its original targets [21, 22]. In this scenario, the quadtree was designed as a representation of images not only for storage or transmission purposes but to process them directly in main memory [18]. To fit the structure in main memory, the size is a relevant issue, and thus since it is a tree, pointer-less representations were introduced [28, 29]. These pointerless representations use a *locational code* that for each leaf of the tree gives its position in the space [28] or an implicit ordering [29]. For our work, it is of special interest the latter case, denoted as *Treecodes*. The region quadtree is represented by a sequence of numbers, each representing a node of the conceptual region quadtree. Each of these numbers has 5 bits, the most significant bit indicates whether the corresponding node is a leaf or not, and the remaining 4 bits store a value. In the case of a leaf node, that number is the value corresponding to a pixel of the image; in non-leaf nodes, it is the average value of the pixels contained in the region represented by such a node. This average value is used to give a preview of the image during a slow transmission through a network. The quadtree is stored as a sequence of bytes, each storing a 5-bit number, where the correspondence of each byte with the nodes of the conceptual tree is given by the ordering of the sequence, which is a *breadth-first* traversal of the tree. The representation of the image of Figure 1 is the sequence of bytes: 20, 3, 18, 4, 7, 0, 1, 2, 5. The first 20 corresponds to the root node, which is an inner node signaled with a 1 in the fifth bit, the next 4 bits store the average value of all pixels in the image (4), and thus we have a 10100 (20). The third byte (18) corresponds to the quadrant further divided into subquadrants, therefore it represents an internal node (fifth bit set to one) and the next four bits store the average value (2).

Our work also uses an implicit ordering using a breadth-first traversal, but we separate the topology of the tree (the most significant bit in the 5-bit number) from the content (the remaining 4 bits). Thanks to this split, we can use more appropriate methods to represent these two types of information. The topology (the bit indicating whether a node is a leaf or not) is represented with a k^2-tree, which uses 1 bit per node and it is a very efficient structure to navigate. With respect to the content, the first difference is that, in non-leaf nodes, instead of storing the average value of the corresponding subquadrant (only useful for pre-visualization purposes), we store the min-max values to index the values at the cells of the raster.[2] The second difference is that the cell values (corresponding to leaves) and the min-max values of inner levels are stored using DACs (see Section 2.2.2) and differential encoding, which can achieve good compression and allow fast access times.

As explained, the quadtree has been used with different purposes, although the use of the quadtree to compress rasters (including images) [36, 37, 38, 39, 13] is one of the main research lines. Another use of the quadtree is to index rasters, although much less effort was devoted to this feature [20, 40]. The best known example is storing the leaves of a linear quadtree [41] in a B^+-tree [42, 43].

The region quadtree indexes the space allowing *spatial* searches, however in [20], the inner nodes of a quadtree, called Binned Min-Max Quadtree, are enriched with the min-max values of the region represented by such a node, thus indexing the values of the raster dataset as well. However, there are several differences with respect our work. First, the values stored at the leaf nodes and the corresponding min-max values at inner nodes are not the actual values in the raster. They use a binned or histogram strategy, which consists in assigning a code to ranges of possible values, for example, 0 encodes the values between -50 and -10, 1 the values between -9 and 0, and so on. Then to perform searches, first we have to encode our search value, and then use that encoded value to take decisions at the nodes of the quadtree. This implies that the quadtree is simply a classical index and thus we have to store the original raster separated from the quadtree, using a classical representation, that is, the quadtree is an auxiliary structure of the main data file. In addition, the quadtree with binned codes limits the pruning capacity of the tree to the boundaries of the ranges defined by the binned strategy. Finally, this previous work is mainly focused on search capacity, and thus there are no worries about space, using a naive pointer-based implementation. Later, the same team presented a new data structure, called Cache Conscious Quadtree [44], which is a quadtree where all nodes are placed in a one-dimensional array to avoid non-continuous memory allocations, in order to improve constructions times. Each node has a field indicating the position of its first child in the array and the min-max values. It uses again a binned strategy, and thus, it is just an auxiliary index.

As a summary, we can point out that the main difference of our approach with respect to these works is that while these works are either focused on representing the raster using compression or on designing an auxiliary index of the raster data, our work joins these two worlds. We present a storage structure for representing the raster data in a compressed form, and at the same time, it indexes both the space and the values of cells. This is a common feature in compact data structures, where there are many structures, usually called self-indexes, having this capacity for different data types such

[2]The idea of storing the minimum and maximum values in the internal nodes is known as lightweight indexing, as it is inexpensive to offer [30, 31, 32]. It is used in sparse indexes of well-known databases systems, such as in IBM's ZoneMaps [33], PostgreSQL's Block Range Indexes (BRIN) [34] or Oracle's Storage Indexes [35], as they offer good indexing for different queries with a small footprint.

as text [14] or graphs [15].

2.2. Basic compact data structures

In this section, we present several compact data structures that are used as building blocks of other compact data structures, including the k^2-*raster*.

2.2.1. Rank and select operations over bitmaps

Bitmaps, together with *rank* and *select* operations, are a basic component of most compact data structures [11]. For example, like quadtrees, k^2-*raster* is also conceptually a tree. In order to compactly represent the topology of that tree, k^2-*raster* uses only a bitmap, which can be efficiently navigated by using rank operations.

Let $B[1, n]$ be a bitmap, that is, a sequence of bits. $rank_b(B, i)$ returns 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, $rank(B, i) = rank_1(B, i)$. $select_b(B, i)$ locates the position of the i^{th} occurrence of b in B.

These operations can be answered in constant time using just $o(n)$ extra bits on top of B [45].

2.2.2. DACs

As we will see next, k^2-*raster* will have to store some integer values, thus, we will use a space- and time-efficient encoding for integer sequenes.

In general, compression of sequences (or arrays) of numbers is one of the most old problems in the compression field [46, 47, 48, 49, 50]. In particular, one could also see the problem of representing raster datasets as representing an array of numbers. The basic idea of most techniques is to use fewer bits to represent the most frequent numbers, which are often the smallest ones, and more bits for the least frequent. This approach requires a variable-length encoding, which poses the following problem: in order to use the sequence of numbers directly in compressed form, we must be able to access to the i^{th} number without decompressing the sequence. Therefore, a family of compression methods arises having this property [51, 52, 53, 54, 55, 56, 57].

A member of this family of techniques are the Directly Addressable Codes (DACs) [58], which are of special of interest for this work, since this is the method used to compress the values of the cells of the raster. DACs obtain a very compact representation if the sequence of integers has a skewed frequency distribution, 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 $X = x_1 x_2 \cdots x_n$, DACs take the binary representation of that sequence and rearrange it into a level-shaped structure as follows: the first level A_1 contains the first n_1 bits (least significant) of the binary representation of each integer. A bitmap B_1 is added to indicate, for each integer, whether its binary representation requires more than n_1 bits or not. More precisely, for each integer, there is a bit set to 0 if the binary representation of that integer does not need more than n_1 bits and a 1 otherwise. In the latter case, the second level A_2 stores the next n_2 bits of the integers having a 1 in B_1, and a bitmap B_2 marks the integers needing more than $n_1 + n_2$ bits, and so on. This scheme is repeated as many levels as needed. The number of levels $\mathcal{L}$ and the number of bits n_l at each level l, with $1 \leq l \leq \mathcal{L}$, is calculated in order to achieve the maximum compression.

DACs can efficiently retrieve the integer encoded at any given position by obtaining the n_l bits at each level that form the binary representation of the number. That is, to recover the number, a top-down traversal is needed, and thus, the worst case time for extracting a random codeword is $O(\mathcal{L})$, being $\mathcal{L}$ the number of levels used. The position of the corresponding bits at each level is obtained performing *rank* operations over the bitmaps B_l.

If we adjust the number of levels and the size of the number of bits in each level (n_l) to obtain the maximum possible compression, this may lead to slow access times, if it requires a considerable large number of levels. DACs can be configured to obtain the minimum possible space but limiting the number of levels $\mathcal{L}$. We use this feature in our proposal.

2.2.3. k^2-tree

The topology of the underlying tree of a k^2-*raster* is represented with a bitmap. There are several compact representations of trees using bitmaps [45, 59, 60] that allow efficient navigation. k^2-*raster* uses a simplified and compact representation based on LOUDS (level-ordered unary degree sequence) tree representation [45], together with a region quadtree decomposition [27, Section 2.1.2.4]. This strategy is the basis of the k^2-tree data structure [24], which is a storage structure for binary matrices. It was originally designed to compress Web graphs and, as all compact data structures, allows accessing and querying the data without decompressing it.

From a binary matrix of size $n \times n$, and being k an input parameter, the k^2-tree is built as a non-balanced k^2-ary tree, where each node corresponds to a submatrix resulting from a recursive division of the matrix into k^2 submatrices of the same size. The first partition divides the original matrix into k rows and k columns of submatrices of size n^2/k^2. Each of those submatrices generates a child node of the root having only one bit, 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 then, the tree decomposition ends there. The submatrices having at least one 1 are recursively divided into k^2 submatrices, producing each one a child node of the corresponding parent. This process continues until reaching a submatrix full of 0s or until reaching the cells of the original matrix (i.e., submatrices of size 1×1). Figure 2 shows an example of this subdivision (left) and the resulting k^2-ary tree (right) for $k = 2$.

Instead of using a pointer-based representation, the tree is compactly represented by just using two bitmaps T and L, whose values are the bit values resulting from a breadth-first traversal of the tree. T stores all the bits of the k^2-tree except those at the last level of the tree, whereas 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 accessing bitmaps T and L. In particular, it is possible to retrieve any cell, row, column or region of the matrix in a very efficient time. This navigation is obtained by means of top-down traversals in the conceptual tree, which are simulated with *rank* operations over T.

The k^2-tree has an excellent performance in both space and time when the binary matrix is sparse, with large zones of 0s and where the 1s are clustered. There also exists a variation of the k^2-tree that compresses areas full of 1s [23]. In this variation the subdivision ends when the algorithm finds a submatrix full of 0s (*white* zones) or full of

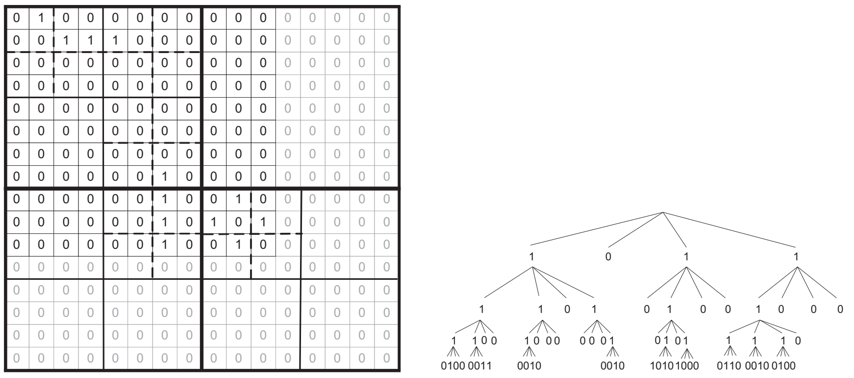


Figure 2: Example of binary matrix(left) and resulting k^2-tree representation (right), with $k = 2$.

1s (*black* zones), adding a method to distinguish black and white areas. Therefore the subdivision only continues when the submatrix has a mixture of 0s and 1s (*gray* zones). This representation is more suitable for representing other types of datasets different from Web graphs, such as binary images.

Apart from its original application, the k^2-tree has been used and adapted for many different purposes, among others, to support the compact representation of RDF datasets [61], moving objects [62, 63], general graphs [64], and raster data [23]. The approaches used to represent raster data using the k^2-tree as basis will be explained in the next section.

2.3. Compact data structures for representing rasters

In this section, we present the k^3-*tree* and the k^2-*acc*, the two previous compact data structures for representing and indexing raster datasets [23]. It has been shown that these approaches are superior in both space and search performance to other ways of representing rasters, such those based on compressing GeoTIFF images. Our work, which was preliminary presented in [1], follows these approaches, obtaining better results both in space and query times.

2.3.1. k^3-tree

The first approach is obtained by simply adding another dimension to the k^2-tree. The k^3-*tree* stores a binary cube using the same partitioning and representation strategies used in the k^2-tree.

The k^3-tree stores points $\langle x, y, z\rangle$, where the first two values represent the position in the 2-D space, and the third component is the value stored in that cell. It is possible to efficiently navigate the k^3-tree, basically using the same procedures used in the k^2-tree, but extended to three dimensions. If we want to obtain the value stored at a given position, we just fix that position in the 2-D space (x and y) and then we check the corresponding z value. To obtain the cells with a given value or a range of values, we fix the value(s) of z, and we search the values of (x, y) having the given value(s).

2.3.2. k^2-acc

Another way to represent a raster having values in the range $v_1 < v_2 < \cdots < v_t$ is to use a k^2-tree for each value. Then, the representation is formed by t k^2-trees $K_1, K_2, \ldots, K_t$, where each K_i has a value 1 in those cells whose value is $v \leq v_i$ in the original raster. Observe that the k^2-trees corresponding to the bigger values (those close to v_t) will have large areas full of 1s, therefore the variant of the k^2-tree that compresses also the areas full of 1s is used. This approach is called *accumulated k^2-trees* or *k^2-acc*.

To obtain the value at a given cell, a binary search over the collection $K_1, K_2, \ldots, K_t$ is needed. This approach is very efficient returning the cells having a value in a given range $[v_b, v_e]$, since it only needs to check K_b and K_e. To obtain the cells having a given value is also solved accessing two k^2-trees.

Comparing k^2-acc and k^3-tree, the first one is better in retrieving cells containing a given value or range of values, whereas the k^3-tree obtains better space consumption and time results when retrieving the value at a given position.

3. Our proposal: *k^2-raster*

In this section, we present the *k^2-raster*, a new storage structure that represents rasters in compressed space, and at the same time, indexes the space and the values stored at cells.

Let M be a raster matrix of size $n \times n$, being n a power of k, where each cell M_{ij} stores a value $v \geq 0$.[3] The *k^2-raster* uses the same partitioning strategy used by the original k^2-tree, that is, it recursively divides M into k^2 submatrices, and builds a tree representing this recursive subdivision. In *k^2-raster*, the recursive division stops when all the cells in a submatrix have the same value. The nodes of the tree store the minimum and the maximum values of the corresponding submatrix in order to index the values at cells. Therefore, the *k^2-raster* puts together the quadtree spatial index, the min/max indexing of rasters, and a compressed representation of the data. As explained, the *k^2-raster* joins in a unique storage structure two desirable properties: indexing capabilities and an efficient representation of the values in the cells and the values at the nodes.

3.1. Construction and data structures

The first step of the construction process is the creation of the root node that stores the minimum and maximum values $(rMin, rMax)$ of the complete matrix. If $rMin$ and $rMax$ are equal, only one value is stored as the maximum, and the process ends here. Otherwise, the two values are stored and the matrix is divided into k^2 submatrices, each adding a child node to the parent, in this case, to the root node. For each generated submatrix, the process is recursively repeated, until the maximum and minimum values become equal, or until the decomposition reaches the last level, that is, when the decomposition of a submatrix obtains submatrices of just one cell. Observe that, being $n' \times n'$ the size of the matrix, the tree has a height of at most $h = \lceil \log_k n' \rceil$ levels.

[3]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, making it of size $n' \times n'$ such that $n' = k^{\lceil \log_k \max\{n,m\} \rceil}$, 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 effectively compresses large areas of equal values.

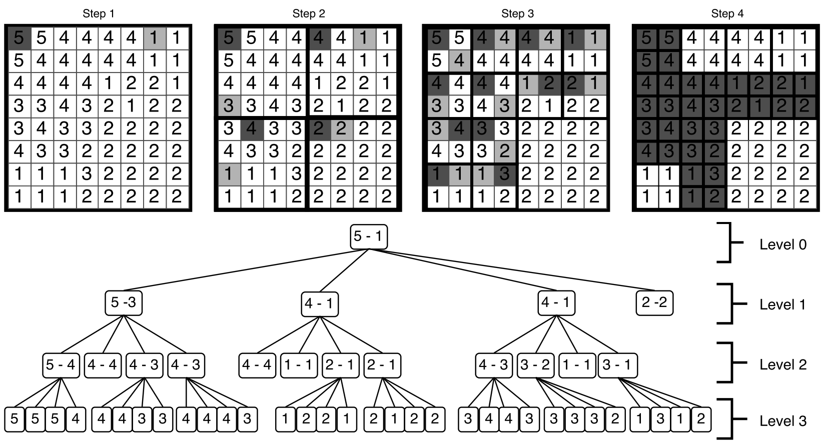


Figure 3: Example of raster matrix (top). We indicate the minimum (light gray) and maximum (dark gray) value of each submatrix for the four steps of the recursive subdivision of the construction algorithm, using $k = 2$. Conceptual tree representation obtained from the construction of the k^2-*raster* (bottom). Numbers at each node indicate the maximum and minimum value of its corresponding submatrix. In the last level, only one value is shown, since the submatrices are formed by only one cell.

Figure 3 shows an example that illustrates the process. Under the label “Step 1”, we can see an 8×8 raster matrix, and below it, the corresponding k^2-*raster* using $k = 2$. The root node stores the maximum and minimum values of the matrix, since these values are different, the 8×8 raster matrix is divided into 4 submatrices of 4×4 cells. Under the label “Step 2”, we can see those submatrices and their maximum (marked in dark gray) and the minimum (marked in light gray) values. For each subdivision, one child node is added to the root node of the tree, storing those values. Observe that in the case of the bottom right submatrix, the maximum and the minimum values are the same (2), therefore such node becomes a leaf node and its corresponding matrix is not further subdivided. The other three submatrices are then subdivided, shown under label “Step 3”, each displaying its maximum and minimum values. Level 2 contains the nodes corresponding to those submatrices, and again, those containing only one value produce a leaf node, and the rest are further subdivided. Finally, at level 3, the process reaches the cell level, and thus, for each subdivision, its node has one child for each of its cells, storing the value of that cell.

The previous description is a high level description of the k^2-*raster*, the actual representation uses several *succinct data structures* strategies to obtain compression. More specifically, we represent the topology of the tree and the maximum and minimum values, which make up the k^2-*raster*, as follows:

- The *topology of the tree* is stored separately from the rest of the information. For this purpose, k^2-*raster* uses a data structure similar to that of a k^2-tree. In contrast to the original k^2-tree, a 0 in a node 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. In addition, while the original k^2-tree is divided into two bitmaps T and L, where L represents nodes of the last level and T the rest of nodes of the tree, k^2-*raster* does not need bitmap L because it would be completely composed of 0s, since the maximum and minimum values of a leaf node will always be equal, as there is just one value at those nodes.

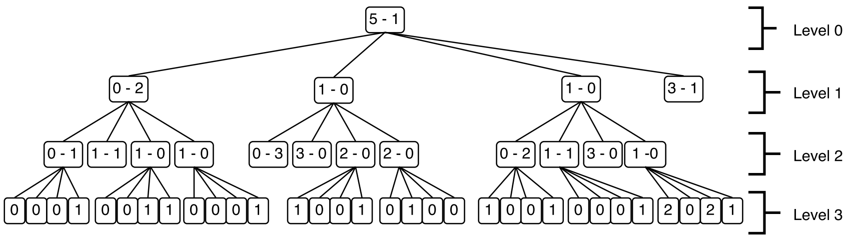
- The *maximum values* of the nodes of the tree are also treated separately. In order to save space, all values, except the value of the root, are encoded as the difference with respect to the maximum value of their parent nodes. Observe that those differences will never be a negative value because the maximum value of a parent node is always equal or greater than the maximum value of its children. We obtain a tree composed of differences, which are stored as a unique array, denoted $Lmax$, where the positions of the values are determined by the breadth-first traversal of that tree. That sequence is composed of differences, which tend to be small, being precisely the situation where DACs can provide good compression and direct access to any given position. The maximum value of the root ($rMax$) is stored separately as an integer in plain form.

- The construction of the structure for the *minimum values* uses the same technique as for the maximum values. The minimum values are again encoded as differences with respect to the minimum value stored at the parent. Again we have always positive values given that the minimum value of a node is always equal or greater than the minimum value of the parent node.[4] The only difference is that we do not need to store any value at the leaf nodes, as the maximum value is enough to represent it. We denote $Lmin$ this array containing the differences for the minimum values, which is also encoded using DACs. The minimum value of the root ($rMin$) is also stored separately as an integer in plain form.

If T has t bits, $Lmax$ has at least t values, and in the first t values of $Lmax$, the i^{th} value corresponds to the maximum value in the submatrix represented by the i^{th} bit of T, as $Lmax$ has one maximum value for each internal node of the tree. That is, they are aligned, since both sequences use the same breadth-first traversal to determine the ordering. However, usually $Lmax$ has more elements, namely the values required for the last level of the tree (which are represented in $Lmax$ but not in T). $Lmin$ only contains values for those internal nodes z with $T[z] = 1$, since the nodes with $T[z] = 0$ has a minimum value equal to its maximum value, and it is already stored in $Lmax$. Since the first t values of T and $Lmax$ are aligned, given a position z of T, its corresponding value in $Lmax$ is the z^{th} number, and we can easily obtain its position in $Lmin$ as $rank(T, z)$.

Figure 4 shows in the upper part a conceptual tree representing a k^2-*raster*. It corresponds to the same raster used in Figure 3. This conceptual tree has an improvement with respect to that in Figure 3, namely the maximum and minimum values stored at each node are now encoded using differences with respect to the values of its parent. The conceptual tree is just shown for illustrative purposes, we only store the data structures shown in the bottom part of the figure. Observe that when the maximum and minimum

[4]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 differences 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.



Tree (T)	1110 1011 0011 1101
rMax	5
rMin	1

Local maximum values (Lmax)	
L1	0113
L2	0111 0322 0131
L3	0001 0011 0001 1001 0100 1001 0001 2021

Local minimum values (Lmin)	
L1	200
L2	100 00 210

Figure 4: 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). Global maximum and minimum values are also stored separately.

values 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 k^2-*raster* can be easily done using a recursive procedure. The algorithm consists in a depth-first traversal of the tree that outputs, separately for each level ℓ of the tree, the bit array of the tree representation T_ℓ and the lists of maximum and minimum values for the nodes of that level ℓ, which we will call $Vmax_\ell$ and $Vmin_\ell$. Then, T can be obtained by concatenating bitmaps T_ℓ for all levels of the tree, and $Lmax$ and $Lmin$ can be obtained from $Vmax_\ell$ and $Vmin_\ell$ respectively, by computing the differences between parents and children, concatenating the sequences of all levels, and encoding the final sequences using DACs. The total time of the algorithm is linear in the number of cells of the matrix, that is, $O(nm)$. In fact, it is optimal, since it processes the raster accessing each cell only once.

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

```
minval ← $\infty$
maxval ← 0
for $i \leftarrow 0 \ldots k-1$ do
    for $j \leftarrow 0 \ldots k-1$ do
        if $\ell = \lceil \log_k n \rceil$ then /* last level */
            if $minval > M_{r+i,c+j}$ then
                $minval \leftarrow M_{r+i,c+j}$
            end
            if $maxval < M_{r+i,c+j}$ then
                $maxval \leftarrow M_{r+i,c+j}$
            end
            $Vmax_\ell[pmax_\ell] \leftarrow M_{r+i,c+j}$
            $pmax_\ell \leftarrow pmax_\ell + 1$
        else /* internal node */
            $(childmax, childmin) \leftarrow$ Build$(n/k, \ell+1, r+i\cdot(n/k), c+j\cdot(n/k))$
            $Vmax_\ell[pmax_\ell] \leftarrow childmax$
            if $maxval <> minval$ then
                $Vmin_\ell[pmin_\ell] \leftarrow childmin$
                $pmin_\ell \leftarrow pmin_\ell + 1$
                $T_\ell[pmax_\ell] \leftarrow 1$
            end
            $pmax_\ell \leftarrow pmax_\ell + 1$
            if $minval > childmin$ then
                $minval \leftarrow childmin$
            end
            if $maxval < childmax$ then
                $maxval \leftarrow childmax$
            end
        end
    end
end
if $minval = maxval$ then
    $pmax_\ell \leftarrow pmax_\ell - k^2$
end
return $(maxval, minval)$
```

The algorithm proceeds as follows: for any level except for the last level of the tree, it performs k^2 recursive calls, each one for the k^2 submatrices resulting from a subdivision. When it reaches the last level of the tree, that call processes k^2 leaf nodes of the tree, which correspond to cells of the original matrix. It checks whether the k^2 cells are all equal. If they are all equal, it just returns that value as maximum and minimum values; otherwise, it appends those k^2 values to $Vmax_\ell$, compute their maximum and minimum values and return them as result of the call.

When returning after a recursive call, the algorithm obtains the maximum and minimum values of its k^2 children. For each child, if these values are different, it appends these values to $Vmax_\ell$ and $Vmin_\ell$ 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. After processing the k^2 children, it checks whether all the maximum and minimum values are equal, which indicates that all the children contain the same value. Thus, the algorithm must undo the last operations, as these nodes will not have a representation in the data 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, the global variables $pmax_\ell$ and $pmin_\ell$ are used to know the last written position of $Vmax_\ell$ and $Vmin_\ell$ respectively. After running the algorithm, all T_ℓ must be joined to make up T, the same must be done with $Vmax_\ell$ and $Vmin_\ell$ to obtain $Vmax$ and $Vmin$, which, in turn, must be converted into $Lmax$ and $Lmin$ by computing the differences and encoding with DACs. Observe that the algorithm returns the maximum and minimum values of the input matrix, that is, $rMax$ and $rMin$, which must be represented in plain form.

3.2. Query algorithms

We describe in this section the algorithms that navigate the k^2-*raster* to solve queries over the raster matrix. We include pseudocodes and examples for some queries to better illustrate the most important procedures.

Obtaining a cell value

To obtain the value of a given cell, the algorithm performs a top-down traversal of the tree. It traverses the node at each level corresponding to the submatrix that contains the queried cell. During the descent through the tree, the algorithm should decode the maximum values stored at the traversed nodes, by subtracting each value from that in the parent. This is needed, since once we reach the queried cell, the stored value is kept as a difference with respect to the maximum value stored at the parent.

Algorithm 2 shows the pseudocode of this query. To obtain the value stored at cell (r, c) of the raster matrix, that is, cell M_{rc} at row r and column c, it is invoked as **getCell**$(n, r, c, -1, rMax)$, where n is the size of the matrix, (r, c) is the position of the queried cell, -1 corresponds to the position in T of the node to process (the initial -1 is an artifact because T does not represent the root node), and $rMax$ is the maximum value in the whole raster. T, $Lmax$, and k are global variables. It is assumed that $rank(T, -1) = 0$.

This query has a worst-case time $O(\log_k n \cdot \mathcal{L})$, which corresponds to 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. $\mathcal{L}$ denotes the number of levels used in DACs for representing $Lmax$,

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

```
$z \leftarrow rank(T, z) \cdot k^2$
$z \leftarrow z + \lfloor r/(n/k) \rfloor \cdot k + \lfloor c/(n/k) \rfloor$
$val \leftarrow accessDACs(Lmax, z)$
$maxval \leftarrow maxval - val$
if $z \geq |T|$ or $T[z] = 0$ then /* leaf */
    return $maxval$
else /* internal node */
    return getCell$(n/k, r \bmod (n/k), c \bmod (n/k), z, maxval)$
end
```

which depends on the largest number encoded in the sequence. This time will be lower when the queried cell is surrounded by cells with the same value.

To illustrate how this query is computed, we will obtain the value at position $(5, 1)$ of the raster shown in Figure 5,which is the cell surrounded with a circle. In the bottom part of the figure we include the corresponding conceptual tree, which is represented using the data structures shown in Figure 4. We invoke the algorithm with **getCell**$(8, 5, 1, -1, 5)$. Having as input the node corresponding to the whole 8×8 matrix, the first step (lines 1–2) is to find the position in T (and thus in $Lmax$) of the node corresponding to the submatrix 4×4 that contains the queried cell, which in our example is the submatrix q_2 of the Figure 5, that is, $z \leftarrow rank(T, -1) \cdot 4 + 5/4 \cdot 2 + 1/4 = 2$. Then, the maximum value of q_2 is obtained (lines 3–4) as follows. First the algorithm obtains the value stored in $Lmax$ as $val \leftarrow accessDACs(Lmax, 2) = 1$, and then it subtracts that value from the maximum value received as a parameter $maxval \leftarrow 5 - 1 = 4$. Next, the condition of line 5 is checked to determine whether we are in an internal node or not. Since $z = 2 < |T| = 16$ and $T[2] = 1$, it recursively invokes **getCell**$(8/2, 5 \bmod 4, 1 \bmod 4, 2, 4) =$ **getCell**$(4, 1, 1, 2, 4)$. Then, the algorithm repeats the same procedure in the next level, this time having as input the node corresponding to submatrix q_2.

Lines 1–2 find the position in T and $Lmax$ of the submatrix of q_2 containing the queried cell as $z \leftarrow rank(T, 2) \cdot 4 + 1/2 \cdot 2 + 1/2 = 12 + 0 + 0 = 12$, which corresponds to the submatrix q_{20} of Figure 5. Then, the algorithm obtains the maximum value of q_{20} as $val \leftarrow accessDACs(Lmax, 12) = 0$, $maxval \leftarrow 4 - 0 = 4$. Since $z = 12 < |T|$ and $T[12] = 1$, the algorithm recursively invokes **getCell**$(4/2, 1 \bmod 2, 1 \bmod 2, 12, 4) =$ **getCell**$(2, 1, 1, 12, 4)$.

Having the node corresponding to submatrix q_{20} as input, the algorithm obtains the position of the submatrix that contains the queried cell (this time is a 1×1 submatrix only containing that cell) as $z \leftarrow rank(T, 12) \cdot 4 + 1/1 \cdot 2 + 1/1 = 39$, and its value as $val \leftarrow accessDACs(Lmax, 39) = 1$, $maxval \leftarrow 4 - 1 = 3$. Finally, since $z = 39 \geq |T| = 16$, a 3 is returned, which is the content of cell $(5, 1)$.

In the conceptual tree of Figure 5, we highlight the nodes affected by this example with ellipses drawn with solid lines.

Obtaining all the values of a region

Obtaining a region of the raster matrix can be done more efficiently than just obtaining its cells individually using *getCell*, since the same top-down traversal of the tree can

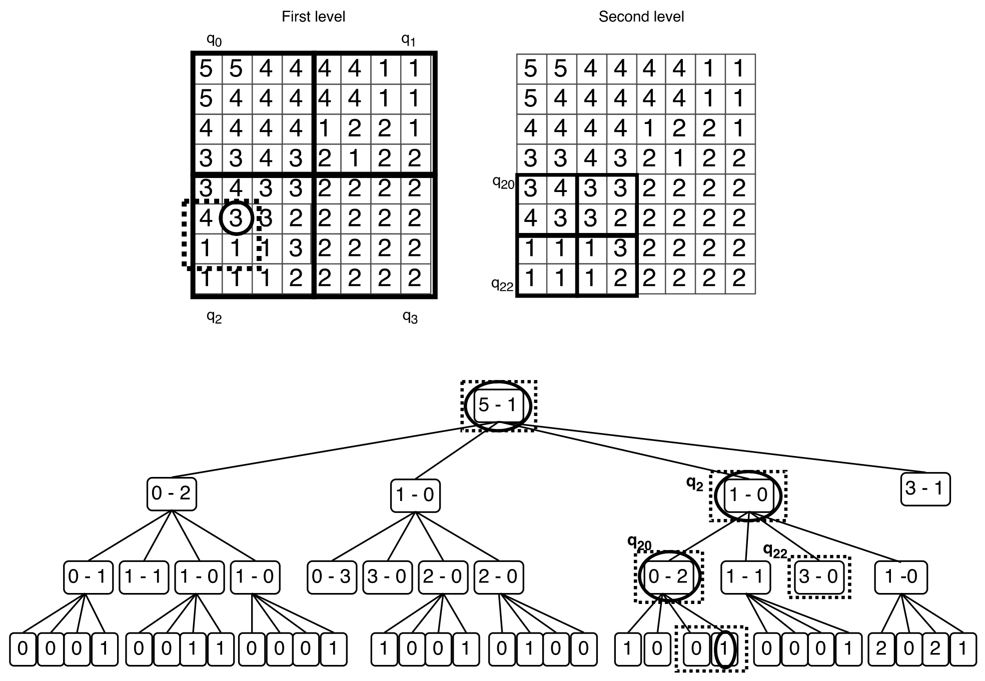


Figure 5: Submatrix subdivision and conceptual tree example to illustrate *getCell* and *getWindow* operations. We highlight the nodes used in the examples.

be used for extracting values from adjacent positions. Thus, decoding maximum values is performed just once per traversed node, instead of once per cell.

Algorithm 3 shows the pseudocode for this query, which is also a recursive procedure. To obtain all the cells contained inside a window $[r_1, r_2] \times [c_1, c_2]$, the algorithm is invoked as $\mathbf{getWindow}(n, r_1, r_2, c_1, c_2, -1, rMax)$. Again, k, T, and $Lmax$ are considered global variables.

Let us illustrate the algorithm with an example using the raster matrix shown in Figure 5. We want to know the cell values in the range $[5, 6] \times [0, 1]$, which is the submatrix surrounded with a square with dotted lines in the figure. In the conceptual tree of Figure 5, we highlight the nodes affected by this example with rectangles drawn with dotted lines.

The algorithm is invoked with $\mathbf{getWindow}(8, 5, 6, 0, 1, -1, 5)$, that is, having as input the size of the whole matrix M, the position of the queried range in M, the position in T of the node representing M (a -1, since the root node is not represented), and the maximum value of M. First, the algorithm computes the position in T and $Lmax$ of the first children of the root node as $z \leftarrow rank(T, -1) = 0$. Then, the algorithm has to determine which submatrices of the first level have to be further inspected to solve the query, that is, which submatrices overlap the queried region. In our case, we only have to inspect the bottom-left submatrix of the current submatrix (corresponding to $i = 1, j = 0$ in line 12), which is the submatrix denoted as q_2. Lines 3–6 and 8–9 give the relative position of the queried range *inside* q_2, in our example, the queried region

Algorithm 3: getWindow$(n, r_1, r_2, c_1, c_2, z, maxval)$ returns all cells from region $[r_1, r_2]$ to $[c_1, c_2]$

```
1  $z \leftarrow rank(T, z) \cdot k^2$
2  for $i \leftarrow \lfloor r_1/(n/k) \rfloor \ldots \lfloor r_2/(n/k) \rfloor$ do
3      if $i = \lfloor r_1/(n/k) \rfloor$ then $r'_1 \leftarrow r_1 \bmod (n/k)$ ;
4      else $r'_1 \leftarrow 0$;
5      if $i = \lfloor r_2/(n/k) \rfloor$ then $r'_2 \leftarrow r_2 \bmod (n/k)$ ;
6      else $r'_2 \leftarrow (n/k) - 1$;
7      for $j \leftarrow \lfloor c_1/(n/k) \rfloor \ldots \lfloor c_2/(n/k) \rfloor$ do
8          if $j = \lfloor c_1/(n/k) \rfloor$ then $c'_1 \leftarrow c_1 \bmod (n/k)$ ;
9          else $c'_1 \leftarrow 0$;
10         if $j = \lfloor c_2/(n/k) \rfloor$ then $c'_2 \leftarrow c_2 \bmod (n/k)$ ;
11         else $c'_2 \leftarrow (n/k) - 1$;
12         $z' \leftarrow z + k \cdot i + j$
13         $maxval' \leftarrow maxval - accessDACs(Lmax, z')$
14         if $z' \geq |T|$ or $T[z] = 0$ then /* leaf */
15             Output $maxval$ $((r'_2 - r'_1) + 1) \cdot ((c'_2 - c'_1) + 1)$ times
16             return
17         else /* internal node */
18             getWindow$(n/k, r'_1, r'_2, c'_1, c'_2, z', maxval')$
19         end
20     end
21 end
```

is the submatrix $[1, 2] \times [0, 1]$ of q_2, that is, it covers rows 1 and 2 and columns 0 and 1 of q_2. Line 12 obtains the position z' in T and $Lmax$ corresponding to the submatrix q_2 as $z' \leftarrow 0 + 2 \cdot 1 + 0 = 2$. Next, the algorithm computes the maximum value of q_2 as $maxval' \leftarrow 5 - accessDACs(Lmax, 2) = 5 - 1 = 4$. Since $T[2] = 1$ and $2 < |T|$, that node is an internal node, and thus the recursive call **getWindow**$(4, 1, 2, 0, 1, 2, 4)$ is launched. That is, to solve our query, it has to return the cells in the region $[1, 2] \times [0, 1]$ of the 4×4 submatrix q_2.

This call starts by computing the position of T and $Lmax$ where the children of q_2 start as $z \leftarrow rank(T, 2) \cdot 4 = 3 \cdot 4 = 12$. The **for** of line 2 iterates i over 0..1 and the **for** of line 7 iterates j only over 0. Therefore, at this call, we have to treat two submatrices of q_2, the top-left and the bottom-left submatrices, denoted q_{20} and q_{22} in Figure 5.

- q_{20}: lines 3–6 and 8–9 give the relative position of the queried range inside q_{20}. Observe that the part of the queried range that overlaps q_{20} is the submatrix $[1, 1] \times [0, 1]$ within q_{20}, which corresponds to submatrix $[5, 5] \times [0, 1]$ in the original matrix. Now, the algorithm obtains the position in T and $Lmax$ of the information corresponding to q_{20} as $z' \leftarrow 12 + 2 \cdot 0 + 0 = 12$ and we obtain the new maximum value as $maxval' \leftarrow 4 - accessDACs(Lmax, 12) = 4 - 0 = 4$. Given that $T[12] = 1$ and $12 < |T|$, that node is an internal node, and thus the recursive call **getWindow**$(2, 1, 1, 0, 1, 12, 4)$ is performed.

  The execution of this call starts by computing the position in $Lmax$ where the children of q_{20} start, $z = rank(T, 12) \cdot 4 = 36$. The **for** of line 2 iterates i only over 1 and the **for** of line 7 iterates j over 0..1. That is, this call has to process the bottom-left and bottom-right submatrices of q_{20}. Those submatrices only contain one cell, that is, they are leaves. For the bottom-left leaf, the algorithm computes its position in $Lmax$ as $z' \leftarrow 36 + 2 \cdot 1 + 0 = 38$, and thus it obtains its value as $maxval' \leftarrow 4 - accessDACs(Lmax, 38) = 4 - 0 = 4$. On the other hand, for the bottom-right leaf, its position in $Lmaxer$ is $z' \leftarrow 36 + 2 \cdot 1 + 1 = 39$, and its value is $maxval' \leftarrow 4 - accessDACs(Lmax, 39) = 4 - 1 = 3$.

- q_{22}: lines 3–6 and 8–9 obtain the relative position of the queried range inside q_{22}. Observe that the part of the queried range that overlaps q_{22} is the relative submatrix $[0, 0] \times [0, 1]$ ($[6, 6] \times [0, 1]$, if we consider the whole matrix).

  Recall that at the start of this call, z was set to 12, then we compute the position in T and $Lmax$ of the information associated with the submatrix q_{22} as $z' \leftarrow 12 + 2 \cdot 1 + 0 = 14$. Thus, we can obtain the maximum value of that submatrix as $maxval' \leftarrow 4 - accessDACs(Lmax, 14) = 4 - 3 = 1$.

  Given that $T[14] = 0$, the node corresponding to q_{22} is a leaf node, therefore line 15 returns the value of $maxval'$ $((0 - 0) + 1) \cdot ((1 - 0) + 1) = 2$ times. That is, since all cells of q_{22} have the same value (1), then it is represented as a leaf node, and the part of q_{22} that overlaps the queried region contains two cells, and then this call returns two 1s.

Retrieving cells with a given value or range of values

We describe now how to obtain the positions of all cells within the region $[r_1, r_2] \times [c_1, c_2]$ that contain values in the range $[v_b, v_e]$. If we want to run the query for the whole matrix, we just adjust $[r_1, r_2] \times [c_1, c_2]$ to the complete matrix, and if we want to search the cells having a particular value v, we adjust the range to $[v, v]$.

The algorithm to solve this query combines the functionality of the original k^2-tree to solve range queries, which is able to efficiently obtain cells with 1s within a given rectangle, with the indexing capabilities offered by the k^2-*raster*, thanks to the storage of the maximum and minimum values at the nodes of the tree. As in previous queries, the search involves a top-down traversal of the tree, but it requires to perform two checks at each level. After obtaining the branches of the tree corresponding to submatrices overlapping the queried region, it has to check whether the maximum and minimum values in those quadrants are compatible with the queried range, discarding those that fall outside the range of values sought.

Algorithm 4 shows the pseudocode for this query. It is again a recursive procedure invoked as **searchValuesInWindow**$(n, r_1, r_2, c_1, c_2, v_b, v_e, rMax, rMin, -1)$, if we want to retrieve the cells inside the window $[r_1, r_2] \times [c_1, c_2]$ having values in the range $[v_b, v_e]$. For this algorithm, k, T, $Lmax$, and $Lmin$ are considered global variables.

Lines 1–14 of Algorithm 4 are exactly the same as those in *getWindow*. If the condition of line 14 is true, we have reached a leaf node that corresponds to a submatrix that overlaps the queried region. In the case of *getWindow*, the algorithm immediately returns the values of the cells in that region, but now the algorithm *searchValuesInWindow* has to perform the second test (line 16) to check whether the values of the cells in that region have values in the range of values $[v_b, v_e]$. Observe that when reaching this point, all cells

Algorithm 4: searchValuesInWindow$(n, r_1, r_2, c_1, c_2, v_b, v_e\ maxval, minval, z)$ returns all cell positions from region $[r_1, r_2] \times [c_1, c_2]$ containing values within $[v_b, v_e]$

```
 1 $z \leftarrow rank(T, z) \cdot k^2$
 2 for $i \leftarrow \lfloor r_1/(n/k) \rfloor \ldots \lfloor r_2/(n/k) \rfloor$ do
 3     if $i = \lfloor r_1/(n/k) \rfloor$ then $r'_1 \leftarrow r_1 \bmod (n/k)$ ;
 4     else $r'_1 \leftarrow 0$;
 5     if $i = \lfloor r_2/(n/k) \rfloor$ then $r'_2 \leftarrow r_2 \bmod (n/k)$ ;
 6     else $r'_2 \leftarrow (n/k) - 1$;
 7     for $j \leftarrow \lfloor c_1/(n/k) \rfloor \ldots \lfloor c_2/(n/k) \rfloor$ do
 8         if $j = \lfloor c_1/(n/k) \rfloor$ then $c'_1 \leftarrow c_1 \bmod (n/k)$ ;
 9         else $c'_1 \leftarrow 0$;
10         if $j = \lfloor c_2/(n/k) \rfloor$ then $c'_2 \leftarrow c_2 \bmod (n/k)$ ;
11         else $c'_2 \leftarrow (n/k) - 1$;
12         $z' \leftarrow z + k \cdot i + j$
13         $maxval' \leftarrow maxval - accessDACs(Lmax, z)$
14         if $z \geq |T|$ or $T[z] = 0$ then /* leaf */
15             $minval' \leftarrow maxval'$
16             if $minval' \geq v_b$ and $maxval' \leq v_e$ then
                   /* all cells meet the condition in this branch */
17                 Output corresponding region of cells
18                 return
19             end
20         else /* internal node */
21             $minval' \leftarrow minval + accessDACs(Lmin, rank(T, z))$
22             if $minval' \geq v_b$ and $maxval' \leq v_e$ then
                   /* all cells meet the condition in this branch */
23                 Output corresponding region of cells
24                 return
25             end
26             if $minval' > v_e$ or $maxval' < v_b$ then
27                 return /* no cells meet the condition in this branch */
28             end
29             if $minval' < v_b$ or $maxval' > v_e$ then
30                 searchValuesInWindow($n/k, r'_1, r'_2,$
                   $c'_1, c'_2, v_b, v_e, maxval', minval', z'$)
31             end
32         end
33     end
34 end
```

in the considered region have the same value, or it is a region with only one cell, and thus the algorithm only has to return the position of the cells of the submatrix that overlap the queried region.

In case of an internal node (lines 20–29), we have to obtain the minimum value of that submatrix, and compare the maximum and minimum values of the submatrix with the queried range:

- If the minimum and maximum values of the submatrix are within the range $[v_b, v_e]$: then all cells meet the condition of the query; thus, all cells inside the queried region must be returned.
- If the minimum value of the submatrix is greater than v_e or the maximum value is smaller than v_b: then no cell in the submatrix meets the criteria; thus nothing is returned.
- If the values in the cells of the considered submatrix partially match $[v_b, v_e]$: then we have to perform a recursive call to further inspect the submatrix.

Note that this query returns the positions of the values that meet the criteria. If it is required to know the exact values of those positions, they could be retrieved with *getCell*, or in a more efficient way by adding calls to *getWindow* when we report that a submatrix has all its elements within the query range.

Checking the existence of a given value or range of values

Given a value or range of values and a region of the raster matrix, the k^2-*raster* can determine if inside that region, there exits at least one cell with a value in the queried range or if all cells have values within the queried range. The first case is known as *weak* semantics, whereas the latter is known as *strong* semantics.

This query can be done more efficiently than retrieving all the values of the region and then checking if they lie within the range of values. This is basically a simplification of Algorithm 4 that, in the case of weak semantics, as soon as it finds that a submatrix of the queried region has values in the range $[v_b, v_e]$ returns *true*. This can be done in a non-leaf node without the necessity of reaching the leaves, with just the minimum and maximum values stored at that node.

In the case of strong semantics, the query is basically the same, but now, as soon as we find that there is, at least, one cell of a submatrix within the queried region that is not within the range, the algorithm stops returning *false*.

The k^2-*raster* also allows for other efficient queries, such as obtaining the maximum value or the minimum values of region, etc.

3.3. Hybrid variant

As seen, most queries require a top-down traversal from the root node to some leaves at the last level of the tree; therefore, the number of levels has an important impact in query times. To reduce the height of the tree, one can use higher values of k. However, the larger k is, the more space the k^2-*raster* requires, as the uniformity of the values decreases when we consider larger areas of the raster matrix. Thus, we present here a modification of the basic k^2-*raster* that significantly reduces the time of some queries with the cost of just a slight increase of the space requirements.

This version allows us to modify how the matrix is partitioned during the first levels of the tree, by allowing the use of two different values of the k parameter, k_1 and k_2; k_1 is used in the subdivision of the first levels, and k_2 for the rest. The target is to obtain

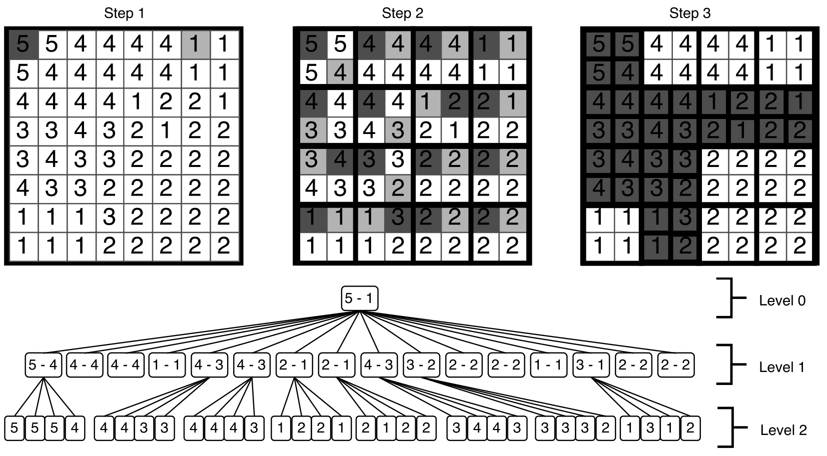


Figure 6: Example of using different k values. We indicate the minimum (light gray) and maximum (dark gray) values of each submatrix for the three steps of the recursive subdivision of the construction algorithm (top). Conceptual tree representation obtained from the construction of the hybrid k^2-*raster* with $k_1 = 4$, $k_2 = 2$ and $n_1 = 1$ (bottom).

a smaller tree, by dividing each quadrant into more submatrices in the first levels, that is, we obtain a wider and smaller tree; but using low values of k for the lower levels of the tree, so that we obtain small submatrices with uniform values that can be compactly represented.

Now, instead of k, the construction algorithm needs the two k_1, k_2 parameters and another new parameter n_1, which is the number of levels where the subdivision is done using k_1. More precisely, when creating 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. From now on, this is the standard version of k^2-*raster*.

In Figure 6, we can see a k^2-*raster* built with $k_1 = 4$, $k_2 = 2$, and $n_1 = 1$. Observe that in the first level (given that $n_1 = 1$), the matrix is divided into $k_1^2 = 16$ submatrices, each producing a child node of the root and storing the maximum and minimum values in that submatrix. Therefore, the root has 16 children. The second level uses $k = 2$, and thus each submatrix is divided into 4 submatrices, which in this case are individual cells. As it can be seen in the figure, the tree is wider and smaller, thus producing faster top-down traversals.

This hybrid variant can be generalized by using a different k value for each level, such that we subdivide level ℓ into k_ℓ^2 submatrices. Using just two values of k, a larger one for the first levels and a smaller one for the last levels of the tree, works well in practice.

3.4. A note on external storage

Although k^2-*raster* was developed following the compact data structure paradigm, which is aimed at storing the whole data structure in main memory, it is possible to store part of it on external storage.

Notice that the k^2-*raster* is composed of several data structures that all together form a tree (at least conceptually). Most indexes are trees given that, among other things, they can be easily split between main memory and disk. The idea is to keep as many complete levels as possible in main memory, starting from the root of the tree. For instance, the typical setup of a B-tree is keeping the first two levels in main memory, and a third level in disk [65, Section 14.2.7]. Observe that this is quite efficient due to two main reasons: i) searches start at the root and continue downwards until reaching the last level; thus, with the aforementioned setup, this implies that searches of a given entry of the index require $O(1)$ disk reads (only for the last level), and ii) the largest part of the tree is precisely the last level of the tree, therefore the upper levels can be easily fitted in main memory.

Hence, with k^2-*raster* we can follow a similar approach, keeping only the last level in disk. This implies to store in main memory the following data structures: *Tree*, *rMax*, *rMin*, *Lmax*, and *Lmin*, except for the part of *Lmax* corresponding to the last level. In fact, when using secondary memory for the last level, we can increase the value of k in the last level of the tree, as we are not that concerned about space requirements on external storage. This allows decreasing the number of levels of the tree, fastening the queries and reducing the space required in main memory for the upper levels.

Let us illustrate this with an example. One raster matrix used in the experimental evaluation occupies 1,488.94 MBs uncompressed and 142.60 MBs as a k^2-*raster*, where *Tree* occupies 3.84 MBs, *rMax* and *rMin* 8 bytes, *Lmax* without the last level 15.38 MBs, and *Lmin* 14.54 MBs. Following the proposed idea, where we keep the last level in secondary memory, our storage structure would require just 33.75 MBs in main memory, which is more than reasonable for current machines. With this setup, searches can be solved in main memory until reaching the last level, where some disk accesses may be required. For instance, to obtain a given cell, only a disk access is enough to obtain that cell.

4. Compressing the last level

4.1. Overview of the heuristic k^2-raster (k^2_H-raster)

The original k^2-tree structure has a variant that uses a compressed representation of the last level of the tree, which is composed of the submatrices of the original adjacency matrix resulting from the last subdivision. This compression allows the use of a large k in the last level, which shortens the tree and improves navigational times, without increasing the space requirements of the structure. In fact, this compression generally causes an improvement on the space results. Thus, following the same strategy used for k^2-trees, we also propose a variant of k^2-*raster* that uses a compressed representation of the last level of *Lmax*, that is, the entries corresponding to the submatrices of size $k_{Lst} \times k_{Lst}$ of the original raster matrix resulting from last subdivision, where *Lst* denotes the last level of the conceptual tree built by the k^2-*raster* recursive subdivision of the raster matrix and k_{Lst} the value of k used for that level.

Therefore, we will compress $Lmax[Lst]$, which denotes the portion of *Lmax* representing the cells in *Lst*, that is, it represents the values of the $k_{Lst} \times k_{Lst}$ non-equal submatrices of the original raster matrix that appear in the last level of the conceptual tree. In the Figure 4, $Lmax[Lst]$ is the part of *Lmax* labeled as $L3$.

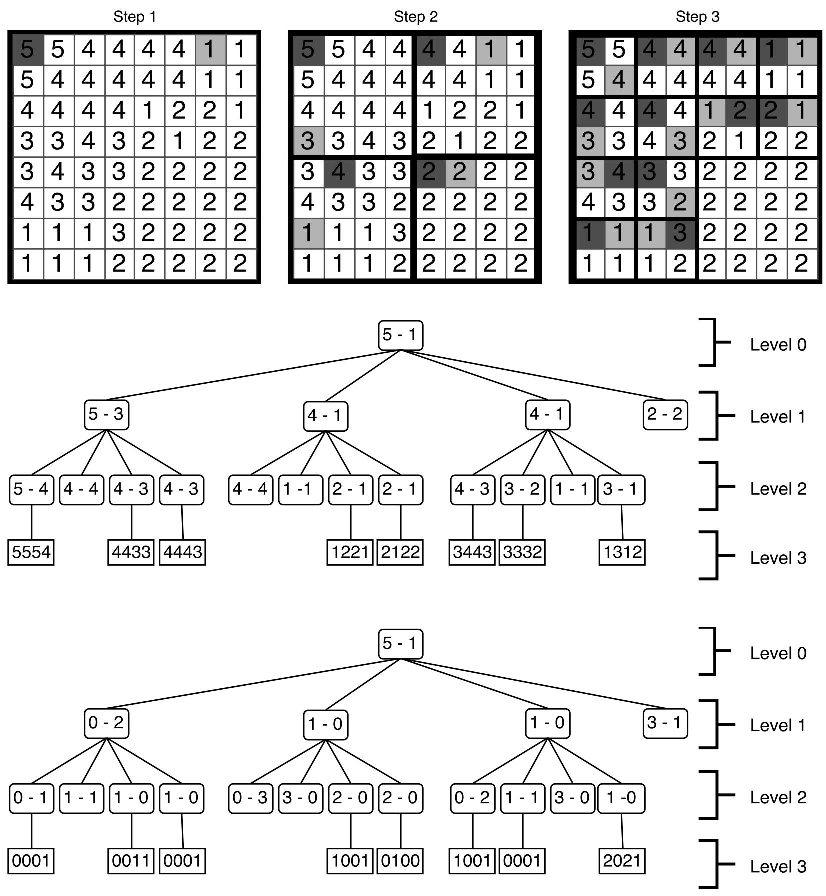


Figure 7: Conceptual tree representation obtained from the construction of the k^2-*raster* (center), and conceptual tree using differential encoding (bottom), where instead of leaf nodes, the last level is represented using $k_{Lst} \times k_{Lst}$ submatrices, being $k_{Lst} = 2$ for this example.

Figure 7 shows the problem we want to address: we maintain the same conceptual representation for the raster matrix except for the last level of the tree, where we want to compactly represent its submatrices of size $k_{Lst} \times k_{Lst}$. We use $k_{Lst} = 2$ for this example, but the idea is to use larger k values for the last level of the tree to shorten the tree and improve time performance.

To compress $Lmax[Lst]$, we will use a statistical compressor, which replaces the most frequent source symbols by shorter codewords, using a semistatic zero-order modeler[5],

[5]Semistatic modelers use a 2-pass algorithm, where the first pass reads the input stream to collect statistics on the data to be compressed, and the second pass does the actual compression using parameters set by the first pass. The statistics (model) are included as part of the compressed data. Zero-order modelers provide the probability of each source symbol without taking into account the surrounding symbols in the source stream. More detailed explanations of these concepts can be found in [66].

where the source symbols of the modeler are $k_{Lst} \times k_{Lst}$ submatrices. One possibility is to follow this strategy for all the submatrices appearing at $Lmax[Lst]$. Thus, we can proceed as follows: we create a vocabulary by extracting all the different $k_{Lst} \times k_{Lst}$ last-level submatrices, sort the vocabulary by frequency, and substitute the $k_{Lst} \times k_{Lst}$ contiguous values in $Lmax$ corresponding to each submatrix by a pointer to its entry in the frequency-sorted vocabulary. However, this strategy, which is the one used for the binary last-level submatrices in the original k^2-tree, is not suitable for k^2-*raster*, since there are many possible different integer submatrices of size $k_{Lst} \times k_{Lst}$, some of them appearing just once, and therefore, the vocabulary becomes very large. Since the compressed representation of $Lmax[Lst]$ consists not only of the pointers but also of the vocabulary, we obtain no compression in case of large vocabularies with many submatrices that are not repeated.

In the example of Figure 7, we would have a vocabulary composed of the five 2×2 different submatrices existing at the last level of the conceptual tree. The vocabulary, sorted by frequency, would be $v = \{\langle 0001 \rangle, \langle 1001 \rangle, \langle 0011 \rangle, \langle 0100 \rangle, \langle 2021 \rangle\}$. If we represented $Lmax[Lst]$ with the vocabulary approach, it would require the list of pointers to each vocabulary entry, that is, $p = \{0, 2, 0, 1, 3, 1, 0, 4\}$, in addition to v. Thus, for those submatrices appearing just once in $Lmax[Lst]$, that is, $\langle 0011 \rangle, \langle 0100 \rangle, \langle 2021 \rangle$, we would require their plain representation in v plus a pointer in $Lmax[Lst]$. The basic k^2-*raster* presented in the previous section would represent these submatrices by simply storing their content, without the overhead of the pointer. Thus, this compression approach would require more space for these submatrices, which may lead to worse space results.

Thus, compressing $Lmax[Lst]$ requires a more refined approach, where we evaluate if including a submatrix in the vocabulary will save space in the final representation. We use an entropy-based heuristical approach to estimate these savings. Thus, we call this improved variant of the technique *heuristic* k^2-*raster* or k^2_H-*raster*.

4.2. Building the k^2_H*-raster*

More specifically, to obtain the k^2_H-*raster* of a given raster matrix, we first build the normal k^2-*raster*, and then perform the following steps:

1. We traverse all $k_{Lst} \times k_{Lst}$ submatrices corresponding to $Lmax[Lst]$, and compute their frequency. Simultaneously, we also compute the frequency for all the individual values that appear in those submatrices.
2. We estimate the average number of bits needed for representing a submatrix using the vocabulary-based approach. Simultaneously, we estimate the average number of bits needed for representing an individual cell value when using DACs to represent them.
3. We sort the vocabulary of submatrices by frequency.
4. For each submatrix of the vocabulary:
   (a) We estimate the cost of representing it as a compressed submatrix using the vocabulary: we multiply its frequency by the average number of bits required for representing a submatrix and we add the space needed to store it in the vocabulary.
   (b) We estimate the cost of representing it as individual values using DACs: we multiply the frequency of the submatrix by its size (i.e., k^2_{Lst} cells) and by the average number of bits required for representing an individual number.

(c) We choose the representation with minimum cost. In case of choosing the vocabulary-based approach, we assign a new correlative codeword to the submatrix, which is a pointer to its position in the vocabulary.

We use the zero-order empirical entropy [67] to estimate the average number of bits needed to encode the submatrices and the individual values (step 2). The zero-order empirical entropy of a sequence S is $H_0(S) = -\sum_{c \in \Sigma} f_c \log_2 f_c$, where Σ is the alphabet of the sequence S and f_c is the relative frequency of symbol c. When we estimate the average number of bits to represent a matrix, the alphabet is the list of different submatrices (the vocabulary of submatrices) in $Lmax[Lst]$. In the case of individual values, the alphabet is formed by the list of different integers appearing in $Lmax[Lst]$.

Then, for a given submatrix s_i having the values $v_1, v_2, \ldots v_{k^2_{Lst}}$ in its cells, we estimate the size (in bits) required to represent that submatrix in step 4a as $E_{s_i} = (f_{s_i} \cdot H_0(S^s)) + (k^2_{Lst} \cdot w)$, where f_{s_i} is the frequency of appearance of s_i in $Lmax[Lst]$, $H_0(S^s)$ is the average number of bits to represent a submatrix using an alphabet of submatrices (computed in step 1), and w is the machine word size. $f_{s_i} \cdot H_0(S^s)$ is an estimation of the size of the pointers that substitute the values of the submatrices in $Lmax[Lst]$. In addition, we also need to store an entry in the vocabulary with the k^2_{Lst} values of the submatrix in plain form.

On the other hand, if we use all the individual values to represent the content of submatrix s_i, in step 4b we estimate the size as $E_{v_i} = f_{s_i} \cdot k^2_{Lst} \cdot H_0(S^v)$, where $H_0(S^v)$ is the average number of bits to represent each individual value (computed in step 2). In step 4c, if $E_{s_i} < E_{v_i}$, we represent the occurrences of s_i in $Lmax[Lst]$ with pointers to the entry of the vocabulary of submatrices corresponding to s_i; otherwise we use the original method, that is, we represent its values individually using DACs.

Notice that we just use the entropy as an estimation of the average bits needed to represent a submatrix, which takes into account all the submatrices appearing at the last level of the tree and their frequencies. This average value is computed just once, at the beginning of the procedure (step 1); thus, it is used as a quick heuristic to determine the convenience of including a submatrix in the vocabulary or not. It would be possible to refine the procedure, recomputing the entropy after including or discarding the processed submatrices, so the number of bits needed to represent a specific submatrix would be more accurate. However, we prioritize efficiency in this procedure, as recomputing would be very costly and would extremely degrade the compression time.

Once we have decided which submatrices will be represented with the vocabulary, we need to create the structures to implement a new $Lmax[Lst]$ where some submatrices are represented as pointers to entries in a vocabulary and others as a list of individual values. For this purpose, we create three additional structures:

- Bitmap *isInVoc*, which indicates whether one submatrix is represented with a pointer to the vocabulary or not.
- Array *encodedValues*, which includes the codewords (pointers) for the submatrices that are represented using the vocabulary.
- Array *plainValues*, which includes the encoding for the individual values of the submatrices that are not represented using the vocabulary.

Then, we traverse again $Lmax[Lst]$ and for each submatrix:

- If it is in the list of submatrices to be represented with the vocabulary, we set to 1 its corresponding bit in *isInVoc* and append its codeword to *encodedValues*.
- Otherwise, we set to 0 its corresponding bit in *isInVoc* and append all its values to *plainValues*.

Algorithm 5 shows the pseudocode of the algorithm that obtains the bitmap $isInVoc$, and the arrays $encodedValues$ and $plainValues$. The inputs of the algorithm are $Lmax$, and the position of $Lmax$ where $Lmax[Lst]$ starts (the parameter P_{Lst}). The value of k for the last level of the tree, that is, k_{Lst}, is a global variable. For the computation of these structures, we create a temporary vocabulary for all the submatrices (s) where we store their values, frequency and codeword. When this procedure ends, we need to add rank support to bitmap *isInVoc* and compact arrays *encodedValues* and *plainValues* using DACs. In addition, we need to create the final vocabulary (Voc) by removing the submatrices that are represented in plain form. The vocabulary is then composed of those submatrices that appear frequently in the last level, according to the heuristic, and they are represented uncompressed in the vocabulary, using $k_{Lst}^2 \cdot w$ bits each, where w is the machine word size.

To better understand this variant, we show in Figure 8 the compact representation (bottom) resulting from a conceptual tree (top). Notice that the codeword for each submatrix at the vocabulary is implicit and it does not consume space in the representation, as it corresponds to its position in the sorted vocabulary. In this example, only $\langle 0001 \rangle$ and $\langle 1001 \rangle$ have been selected for the vocabulary, as the others only have one appearance in the last level and representing their individual values directly saves more space than including them in the vocabulary and using a codeword.

When processing the last level of leaves from left to right, the first leaf is $\langle 0001 \rangle$, which is one of the leaves that should be represented as a pointer to the vocabulary, therefore the first bit of $isInVoc$ is set to 1. In addition, the algorithm adds the codeword that represents $\langle 0001 \rangle$ in the first position of $encodedValues$, that is, it inserts a 0, since that is the position of $\langle 0001 \rangle$ in the frequency-sorted vocabulary. The second submatrix is $\langle 0011 \rangle$, which should be represented in plain form, then the second bit of $isInVoc$ is set to 0 and the four values ($\langle 0011 \rangle$) are stored in the first four position of $plainValues$. Next, we have a $\langle 0001 \rangle$, therefore the third bit of $isInVoc$ is set to 1, and the second entry of $encodedValues$ is filled with a 0, and so on.

4.3. Querying

The navigation over this variant differs from the navigation over the original k^2-*raster* when accessing the last level of $Lmax$. Instead of directly obtaining its values, the k_H^2-*raster* requires accessing the bitmap that indicates whether the submatrix is stored compressed or in plain form, and accessing the corresponding sequence of encoded or plain values. We illustrate how we access the last-level submatrices by showing how the query $getCell$ is done. Algorithm 6 shows the pseudocode of the query. Notice that line 3 from Algorithm 2 has been replaced with lines 3–15. For the sake of simplicity, we use the same k for all levels, but k may have different values at each level of the tree.

Algorithm 5: **Build**$_H$($Lmax$,P_{Lst}) computes *isInVoc*, *encodedValues*, and *plainValues*

```
s ← subMatricesFreq(Lmax,P_Lst,k_Lst) /* Compute the frequency of each
  different k_Lst × k_Lst submatrix in Lmax (step 1) */
v ← valuesFreq(Lmax,P_Lst)/* Compute the frequency of each different
  value in Lmax (step 1) */
H_s ← entropy(s) /* Compute the entropy of the submatrices (step 2) */
H_v ← entropy(v) /* Compute the entropy of the values (step 2) */
sort(s) /* Sort the vocabulary of submatrices by frequency (step 3) */
for i ← 0...|s| − 1/* Determine if we represent the submatrix using a
  codeword or its individual values (step 4) */ do
    if ((H_s · s[i].freq) + (k_Lst^2 · w)) < (H_v · s[i].freq · k_Lst^2) then
        s[i].cdwd ←computeNextCodeword()
    else
        s[i].cdwd ← −1
    end
end
j ← 0
posInEncoded ← 0
posInPlain ← 0
while j < |Lmax| do
    s_i ←searchInS(Lmax[Lst][P_Lst + j ... P_Lst + j + k_Lst^2 − 1]) /* Obtains the
      data in s of the current submatrix */
    if s_i.cdwd = −1 then /* The submatrix is stored in plain form */
        isInVoc[j/k_Lst^2] ← 0
        for t ← 0...k_Lst^2 − 1 do
            plainValues[posInPlain + t] ← s_i.values[t]
        end
        posInPlain ← posInPlain + k_Lst^2
    else /* The submatrix is stored compressed */
        isInVoc[j/k_Lst^2] ← 1
        encodedValues[posInEncoded] ← s_i.cdwd
        posInEncoded ← posInEncoded + 1
    end
    j ← j + k_Lst^2
end
```

To illustrate this with an example, let us obtain the value of the cell at position (5,1) of our running example. The algorithm is invoked as **getCell**$_H$(8,5,1,−1,5). Lines 1–2 obtain the position in T and $Lmax$ of the value corresponding to the 4×4 submatrix that contains the queried cell, $z \leftarrow rank(T, -1) \cdot 4 + 5/4 \cdot 2 + 1/4 = 2$, which corresponds to the bottom-left submatrix. Since $2 < |T|$, we are in a level that is not the last one, and thus the flow reaches line 14. Here, the process is the same as in the case of the normal k^2-*raster*, that is, the algorithm accesses the normal $Lmax$ to obtain the

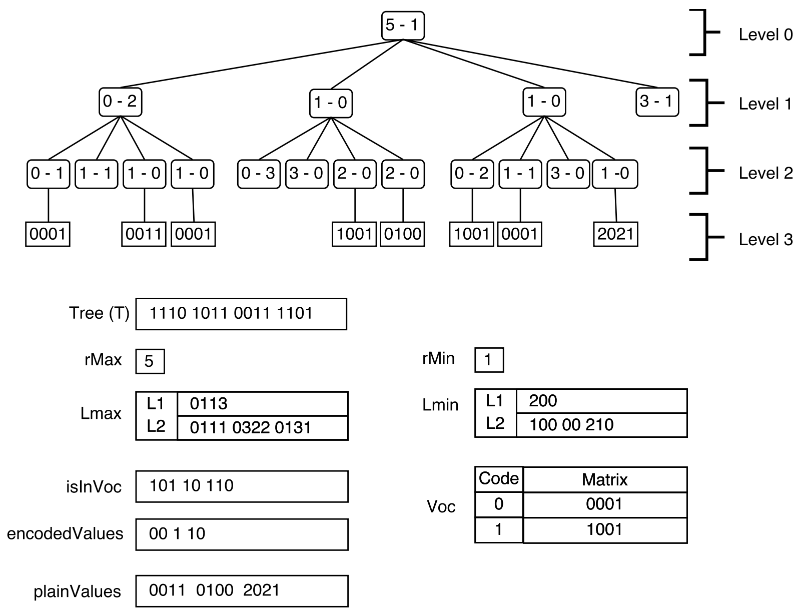


Figure 8: Compact representation of the conceptual k^2_H*-raster* using differences for the maximum and minimum values (top). Data structures T, $Lmax$, $Lmin$, Voc, $isInVoc$, $encodedValues$ and $plainValues$ used for representing compactly the k^2_H*-raster* (bottom).

maximum value of that submatrix: $val \leftarrow accessDACs(Lmax, 2) = 1$, and then it subtracts that value from the maximum value received as a parameter $maxval \leftarrow 5 - 1 = 4$. Then, the condition of line 17 is checked, and since the current node is not a leaf, the flow reaches line 20, performing a recursive call $\mathbf{getCell}_H(8/2, 5 \bmod 4, 1 \bmod 4, 2, 4) = \mathbf{getCell}_H(4, 1, 1, 2, 4)$. In the recursive call, lines 1–2 find the position in T and $Lmax$ corresponding to the node representing the submatrix of the second level that contains the queried cell: $z \leftarrow rank(T, 2) \cdot 4 + 1/2 \cdot 2 + 1/2 = 12 + 0 + 0 = 12$. Again, this node is not in the last level of the tree, then the algorithm obtains the maximum value of that submatrix $val \leftarrow accessDACs(Lmax, 12) = 0$, $maxval \leftarrow 4 - 0 = 4$ and performs a recursive call $\mathbf{getCell}_H(4/2, 1 \bmod 2, 1 \bmod 2, 12, 4) = \mathbf{getCell}_H(2, 1, 1, 12, 4)$. In the next recursive call, $z \leftarrow rank(T, 12) \cdot 4 + 1/1 \cdot 2 + 1/1 = 39$. Now, $z > |T|$, that is, we are accessing a leaf in the last level. Line 4 obtains the corresponding submatrix among those at the last level: $pos \leftarrow \lfloor (39 - |T|)/4 \rfloor = \lfloor (39 - 16)/4 \rfloor = 5$. Line 5 checks if that position is stored as a pointer to the vocabulary or is stored in plain form. Since $isInVoc[5] = 1$, that submatrix is stored as a pointer to the vocabulary. Then the algorithm obtains the position of its codeword in $encodedValues$ as $pos \leftarrow rank_1(isInVoc, 5) - 1 = 3$. Next, the algorithm accesses $encodedValues$ to obtain the codeword of the submatrix: $code \leftarrow accessDACs(encodedValues, 3) = 1$. Thus, the algorithm must obtain the queried cell from the submatrix encoded at position 1 of Voc as $val \leftarrow Voc[1][1 \cdot 2 + 1] = Voc[1][3] = 1$. With that value, line 14 obtains $maxval \leftarrow 4 - 1 = 3$. Finally, since $z \geq |T|$ $(39 \geq 16)$, a 3 is returned, which is the content of cell $(5, 1)$.

Algorithm 6: **getCell**$_H(n, c, r, z, maxval)$ returns the value at cell (c, r)

```
$z \leftarrow rank(T, z) \cdot k^2$
$z \leftarrow z + \lfloor c/(n/k) \rfloor \cdot k + \lfloor r/(n/k) \rfloor$
if $z \geq |T|$ then /* last level */
    $pos \leftarrow \lfloor (z - |T|)/k^2 \rfloor$
    if $isInVoc[pos] = 1$ then /* encoded in Voc */
        $pos \leftarrow rank_1(isInVoc, pos) - 1$
        $code \leftarrow accessDACs(encodedValues, pos)$
        $val \leftarrow Voc[code][c \cdot k + r]$
    else /* plain form */
        $pos \leftarrow rank_0(isInVoc, pos) \cdot k^2 + c \cdot k + r$
        $val \leftarrow accessDACs(plainValues, pos)$
    end
else /* not last level */
    $val \leftarrow accessDACs(Lmax, z)$
end
$maxval \leftarrow maxval - val$
if $z \geq |T|$ or $T[z] = 0$ then /* leaf */
    return $maxval$
else /* internal node */
    return getCell$_H(n/k, c \bmod (n/k), r \bmod (n/k), z, maxval)$
end
```

The rest of query algorithms are easily modified in the same way, that is, only modifying the accesses to the last level of *Lmax* in order to deal with the submatrices in the vocabulary.

5. Experimental evaluation

5.1. Experimental Framework

We measured the space and time results obtained by the two different versions of the proposed data structure, k^2-*raster* (see Section 3), and the heuristic k^2-*raster*, denoted by k^2_H-*raster* (see Section 4).

We performed two different sets of experiments. First, we compared the k^2-*raster* with previous compact data structures for raster datasets: k^2-acc and k^3-tree (see Section 2.3). The second set of experiments compares k^2-*raster* with a classical method to compress rasters, namely Network Common Data Form (NetCDF)[6], which includes a data format to store rasters compressed or uncompressed and the software libraries to access datasets in this format. By using those libraries, it is possible to transparently access compressed netCDF datasets without performing an explicit decompression procedure. However, when accessing a compressed file, the library performs a (hidden for

[6]`http://www.unidata.ucar.edu/software/netcdf/`

the user) decompression procedure, which, since it uses Deflate to compress, must start at the beginning of the dataset.

We separate these two sets of experiments given that compact data structures have a different target with respect to classical approaches. While netCDF is focused on obtaining only compression, as explained, compact data structures obtain compression but also good access times, sometimes even faster than accessing the uncompressed version. Differences between access and query times obtained by netCDF and by compact data structures are so high, that we cannot plot them together as the differences between compact data structures would not be visible.

We measured construction time (only for the first set of experiments), the space consumption, and the navigational time to answer these four types of queries:

- *getCell*: given a position in the raster matrix, this query obtains its cell value. The time was measured by performing 1,000,000 different random queries and we report the average time per query (in microseconds).

- *getWindow*: given a region or window of the raster matrix, this query retrieves all cell values within that window. We measured the time for 100 random queries and report the average time per retrieved cell (in nanoseconds).

- *searchValuesInWindow*: given a range of values and a region of the matrix, this query retrieves all raster positions belonging to the given region whose value lies within that range. We have defined two variants of this query: without any restriction for the range of values and window size, and limiting the range length to 200 and the window size to 500×500. In the first case, we measure the time for 10,000 random queries, and for the second case we measure the time for 100,000 random queries. We report the average time per retrieved cell (in nanoseconds).

- *checkValuesInWindow*: given a region and a range, this query checks if all cell values of the region are within the range of values (we call this variant *strong checkValuesInWindow*) or if there exists at least one cell value in the region whose value lies within the range of values (*weak checkValuesInWindow*). The time was measured by performing 1,000,000 random different queries and obtaining the average time per query (in microseconds for the first set of experiments and in milliseconds for the second set).

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

All the experiments were run on a dedicated Intel® Core™ i7-3820 CPU @ 3.60GHz (4 cores) with 10MB of cache, and 64GB of RAM. It ran Ubuntu 12.04.5 LTS with kernel 3.2.0-115 (64 bits), using gcc version 4.6.4 with `-O9` options. Time results refer to CPU user time. Space consumption was measured in compression percentage, computed as the ratio (in percentage) between the uncompressed size of binary file containing the original raster matrix and the size of the compressed representation.

5.2. Datasets

We used real data in our experiments. More concretely, we used data of different nature from the following two sources:

- WorldClim[7] dataset [68], which provides a set of layers with global climate information. The whole world is divided into equal-spaced tiles, and each cell of a tile is an integer number and has a resolution of about 1 square kilometer. Specifically, we have used the dataset containing the value of the mean temperature, which was measured in degrees Celsius with one decimal, and is represented using integers by multiplying the value by 10.

- Spanish Geographic Institute[8] (SGI), which includes several DTM (Digital Terrain Model) data files that contain the spatial elevation data of the terrain of Spain, stored as rectangular equal-spaced tiles with 5 meters of spatial resolution. Each cell of a tile contains a real number of at most three decimal digits.

In our experiments, we analyzed the scalability and behavior of each technique when varying the size of the input raster matrix and the number of different values included in the raster. Thus, we have created several collections of datasets of different nature with different properties of size and number of different values.

Tables 1 and 2 show the characteristics of collections of datasets with increasing sizes. To obtain them, since in the original datasets all tiles have the same size, we have joined different adjacent tiles to create raster matrices of different sizes. In addition, we have considered different precision by using different number of decimal digits, in the case of the dataset of spatial elevation values, to obtain variability on the number of different values. Tables 1 and 2 show the average values of the main properties (size, number of rows, number of columns and number of different values) for the collection of datasets generated. Specifically, 1×1 matrices were built using just 1 tile, 2×2 matrices were built using 2×2 adjacent tiles, and so on. For each size, we collected a set of different matrices. For example, $\texttt{cat}_0\texttt{-1}\times\texttt{1}$ is composed of 25 1×1 matrices, each corresponding to a different tile of the original dataset. The data shown in the tables represent the mean values obtained by those 25 matrices. This allows us to report the average space and time results obtained in the experiments for each collection, avoiding the dependence on the selection of a unique matrix. The dataset at Table 1 will be denoted as `eua` in the experiments, while the datasets at Table 2 will be denoted $\texttt{cat}_0$ and $\texttt{cat}_3$. The subscript for these datasets 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. Thus, we will report the results when using 0 decimal digits ($\texttt{cat}_0$) and 3 decimal digits ($\texttt{cat}_3$). Notice that $\texttt{cat}_3$ corresponds to raster matrices of the original dataset.

In addition, to analyze the behavior of the methods when only the number of different values varies, but not the matrix size, we generated a collection of matrices from just one random tile (namely, the one denoted as `MDT05-0533-H30-LIDAR`). More concretely, we have first truncated the original values by taking only the two most significant decimal

[7]http://www.worldclim.org/tiles.php
[8]http://www.ign.es

Table 1: Properties of dataset `eua`, obtained from WorldClim datasets. It includes raster matrices of different size and number of different values of the input matrix.

Name	size (MB)	#rows	#cols	# different values
`eua-1×1`	49.44	3,600	3,600	252
`eua-2×2`	197.75	7,200	7,200	413
`eua-3×3`	444.95	10,800	10,800	474
`eua-4×4`	791.02	14,400	14,400	498

Table 2: Properties of datasets $\texttt{cat}_0$ and $\texttt{cat}_3$, obtained from DTM datasets. They include raster matrices of different size and number of different values.

Name	size (MB)	#rows	#cols	# different values
$\texttt{cat}_0$-1×1	91.49	4,100	5,849	868
$\texttt{cat}_0$-2×2	369.03	8,242	11,737	1,201
$\texttt{cat}_0$-3×3	834.76	12,403	17,643	1,503
$\texttt{cat}_0$-4×4	1,488.94	16,564	23,564	1,761
$\texttt{cat}_3$-1×1	91.49	4,100	5,849	779,405
$\texttt{cat}_3$-2×2	369.03	8,242	11,737	1,066,043
$\texttt{cat}_3$-3×3	834.76	1,2403	17,643	1,304,704
$\texttt{cat}_3$-4×4	1,488.94	16,564	23,564	1,545,248

digits. Then we have created other 5 raster matrices $\texttt{MDT05-0533-H30-LIDAR}_{\gg x}$ 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.[9] 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 large number of different values. We denote this dataset as $\texttt{MDT}_x$ in the experiments, and show its properties in Table 3.

5.3. Comparison with compact data structures

We used a hybrid configuration for k^2-*raster* and for k^2_H-*raster*, with $k_1 = 4, k_2 = 2, n_1 = 4$. k^2_H-*raster* used $k_{Lst} = 4$ for the last level of the tree, which made its tree one level shorter. We do not include in the comparison the basic variant that uses the same value of k for all the levels of the tree as it has been proven that the hybrid approach obtains better results [1]. Both variants used an implementation for supporting *rank* operations that adds 5% of extra space on top of the bit sequence T and provides fast queries [69][10]. In addition, *Lmax* and *Lmin* were encoded using the version of DACs that optimizes the space usage while restricting the maximum number of levels. More precisely, we have limited the number of levels to 3. We compared both variants of our

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

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

Table 3: Dataset $\texttt{MDT}_x$, obtained from tile MDT05-0533-H30-LIDAR. It includes raster matrices of the same size, but different number of values.

Name	size (MB)	#rows	#cols	# different values
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 9}$	86.48	3,881	5,841	227
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 7}$	86.48	3,881	5,841	903
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 5}$	86.48	3,881	5,841	3,606
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 3}$	86.48	3,881	5,841	14,415
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 1}$	86.48	3,881	5,841	57,586
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 0}$	86.48	3,881	5,841	114,966

proposal with k^2-acc and k^3-tree using the same hybrid configuration. In addition, we configured parameter $S = 14$ for k^2-acc, which is a parameter used to divide the input raster into 2^S subrasters, each one producing a set of k^2-trees.

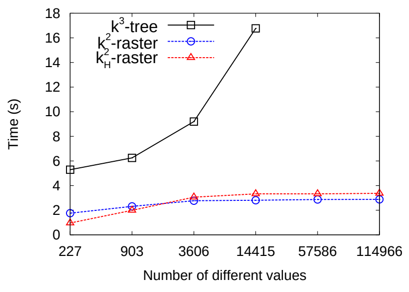
5.3.1. Construction time

Figure 9(left) shows the comparison among all the methods when measuring the construction time. Plots only show the results for k^2-*raster*, k^2_H-*raster*, and k^3-tree, as the times obtained by k^2-acc were more than 2 order of magnitude slower. Moreover, k^3-tree and k^2-acc were not able to create the compressed representation of those raster matrices with a large number of different values, more concretely, they failed when constructing the compressed representation for $\texttt{MDT05-0533-H30-LIDAR}_{\gg 0}$, $\texttt{MDT05-0533-H30-LIDAR}_{\gg 1}$, and all the matrices from dataset $\texttt{cat}_3$.

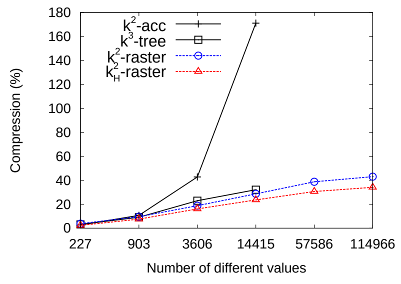
The construction process for k^2-*raster* and k^2_H-*raster* is the same, except for the last level of the representation. k^2-*raster* processes all levels analogously, whereas k^2_H-*raster* has one level less than k^2-*raster*, and it needs to create a vocabulary to compress its last level submatrices. Thus, at that point, the construction time differs between the two structures. To analyze their behavior, we ran two types of experiments, one where we only varied the number of different values, and another where we also varied the size of the input matrices.

For the first experiment we used dataset $\texttt{MDT}_x$, described at Table 3. The results are shown in Figure 9(a). The y-axis shows the time consumption for constructing the compressed representation (in seconds) and the x-axis shows the number of different values for each dataset.

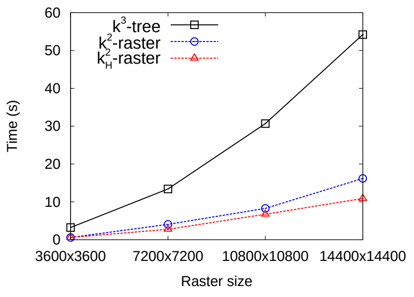
As expected, when increasing the number of different values, the construction time worsens. This happens because it is more likely that a submatrix has more than a single value, which must be divided and processed again. k^2-*raster* and k^2_H-*raster* behave similarly, and clearly outperform k^3-tree, which shows a scalability problem. As previously mentioned, k^2-acc was not included in the plot due to its bad performance. k^2_H-*raster* achieves better results when there is a small number of different values, but it becomes worse than k^2-*raster* for larger numbers of different values. Notice that, in these experiments, k^2_H-*raster* has one level less than k^2-*raster*. In addition, when the number of different values in the matrix is low, the number of different submatrices in level *Lst* is also low, as there is little variability, and the cost of computing the heuristic is not



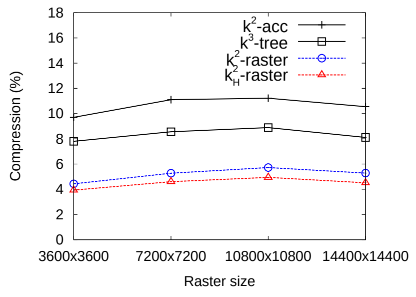
(a) $\texttt{MDT}_x$ – construction time



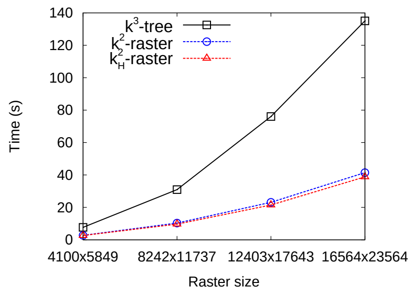
(b) $\texttt{MDT}_x$ – compression



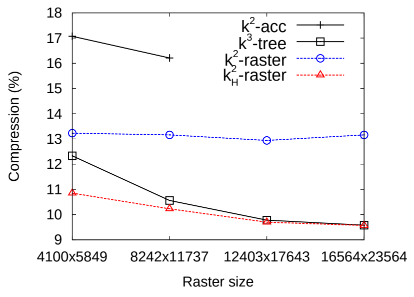
(c) **eua** – construction time



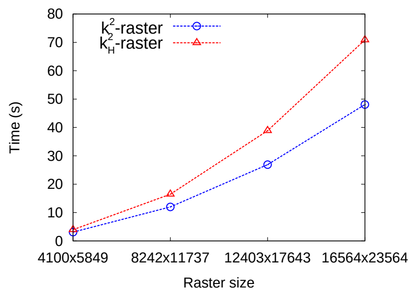
(d) **eua** – compression



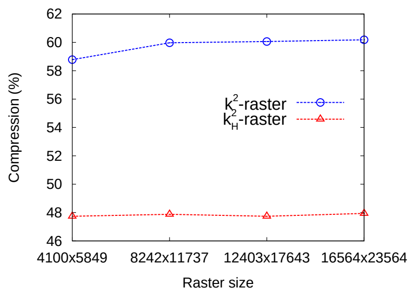
(e) $\texttt{cat}_0$ – construction time



(f) $\texttt{cat}_0$ – compression



(g) $\texttt{cat}_3$ – construction time



(h) $\texttt{cat}_3$ – compression

Figure 9: Construction time (left) and compression percentage (right) for datasets of different nature.

significant. Thus, when the raster matrix has a small number of different values, computing the heuristic is faster than processing one extra level, and thus the k^2_H-*raster* is built faster than k^2-*raster*. When the number of different values increases, the size of the vocabulary grows, and thus, the construction process becomes slower. There are no scalability issues due to this parameter for our proposed structures, as times become almost constant when increasing the number of different values.

We show in Figures 9(c), 9(e), and 9(g) the time consumption to build datasets from collections `eua`, $\texttt{cat}_0$, and $\texttt{cat}_3$ respectively. The y-axis shows the construction time (in seconds) and the x-axis shows the size of each dataset. We can also see that k^2-*raster* and k^2_H-*raster* obtain similar results, being k^2_H-raster faster for those datasets with a lower number of different values. Both of our approaches are faster than the other methods of the state of the art. Neither k^3-tree nor k^2-acc were able to create the compressed representation for collection $\texttt{cat}_3$, which includes raster matrices with a large number of different values. Hence, our proposals show again that are more convenient for real datasets containing a high cardinality.

5.3.2. Space requirements

Figure 9(right) shows the compression obtained by the four methods, k^2-acc, k^3-tree, k^2-*raster* and k^2_H-*raster* over all the datasets. Figure 9(b) shows the results for dataset $\texttt{MDT}_x$, where the number of different values grows while maintaining the size of the raster matrix. With 227 values, the four methods obtained a similar result, around 3% of the original collection size. When the number of different values grows up to 903 values, k^2_H-*raster* begins to obtain better results compared to the other methods. k^2_H-*raster* achieves a compression of 7.5% while for the rest of the structures is around of 9%. With the third raster matrix, which contains 3606 different values, the compression of k^2-acc is significantly worse (43%). Again, k^2_H-*raster* obtains the best compression (16%), followed by the k^2-*raster* (19%) and k^3-tree (32%). This tendency continues with the fourth raster matrix of the dataset. Thus, using a vocabulary-based approach and a selection heuristic improves the compression up to 9% with respect to the standard k^2-*raster* and both structures obtain better results than the techniques of the state of the art. In addition, our techniques were able to create the compressed representation for all datasets, including those with a large number of different values.

We also show the comparison of the compression obtained over the other three collections. As expected, k^2_H-*raster* obtains the best compression for all datasets. Moreover, k^2-acc can only represent the smallest matrices from datasets $\texttt{cat}_0$, and none from $\texttt{cat}_3$, whereas k^3-tree is not able to represent any matrix from dataset $\texttt{cat}_3$. These experiments demonstrate that our solutions can deal with large datasets, rather than the current state of the art. In addition, k^2-*raster* and k^2_H-*raster* maintain good compression ratios even when the number of values of the dataset grows.

5.3.3. Query times

In this section we show the results of the experiments for the queries described in Section 5.1. Again, we used datasets $\texttt{MDT}_x$, `eua`, $\texttt{cat}_0$, and $\texttt{cat}_3$, and compared the results obtained by our two methods, k^2-*raster* and k^2_H-*raster*, to those obtained by the techniques of the state of the art, k^3-tree and k^2-acc.

Time of getCell

Figure 10(left) shows the average time to retrieve the value of a given cell (in microseconds). k^2_H-*raster* outperforms the rest of the techniques for all cases, followed closely by the standard version of k^2-*raster*. Our versions are up to 6 times faster than k^3-tree and 9 times faster than k^2-acc.

The query time of our techniques depends on the height of the tree; in the worst case, it needs to descend down to the last level of the tree, checking one node per level. As we can see in Figure 10(a), query times of k^2_H-*raster*, and also of k^2-*raster*, are almost constant even when the number of different values is high. In addition, k^2_H-*raster* uses a higher value of k for the last level, which decreases the number of tree levels; thus, the time for retrieving an individual cell becomes smaller than using k^2-*raster*, since searching inside the vocabulary is very efficient given that the values are kept in plain form.

From the results, we can observe that k^2-acc and k^3-tree are not suitable for datasets with a large number of different values. To retrieve a value of a cell, k^2-acc performs a binary search among all its k^2-trees, and the number of k^2-trees depends on the number of different values. Thus, for datasets with a large number of different values, obtaining the cell value is slow. In the case of k^3-tree, the z dimension increases accordingly to the number of different values; thus, the searching time by this dimension also grows.

Figure 10(c), Figure 10(e), and Figure 10(g) show the behavior for datasets with different input size. Again k^2_H-*raster* and k^2-*raster* obtain the best results.

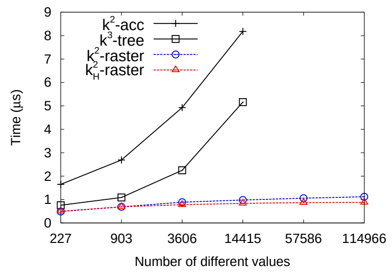
Time of getWindow

Figure 10(right) represents the average time consumption to retrieve all values of a window (measured in nanoseconds per retrieved value). k^2_H-*raster* performs better than the other three methods for all datasets. k^3-tree gets time results similar to those of k^2-*raster* when the number of different values is small, but k^2-*raster* obtains better results with a large number of values, as it is shown in Figure 10(b). The other three plots of Figure 10(right) represent the behavior with different sizes for the input matrix. As expected, k^2_H-*raster* has the best performance. k^2-*raster* and k^3-tree obtain similar result whilst the time of k^2-acc is still the slowest by far. While the other methods know where to find the values of each cell, the k^2-acc needs to search all its k^2-trees until it finds the values corresponding to the cells that is searching, which is a very slow process.

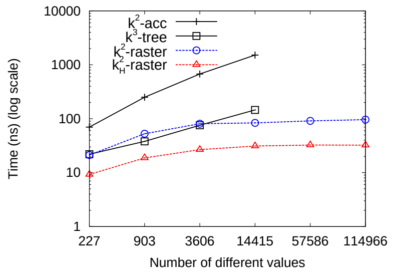
An alternative procedure to obtain all the values of a region is to retrieve each value cell per cell. Comparing the results measured in time per cell retrieved by *getCell* at Figure 10(left) with those obtained by *getWindow* at Figure 10(right), the query *getWindow* takes advantage of the fact that it is possible to obtain adjacent cell values with the same top-down traversal of the tree. Our structures obtain the final value of a cell when reaching a leaf node. If a leaf node, which represents a submatrix with all values equal, belongs to upper levels of the tree, k^2-*raster* and k^2_H-*raster* can complete part of the final result in just one step, without obtaining each value cell per cell.

Time of searchValuesInWindow

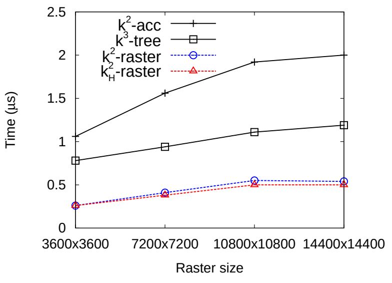
This query retrieves all cells whose values lie within a given range. Figure 11 shows the time consumption per retrieved cell in nanoseconds. We show the results obtained when we do not limit the size of the window nor the range length (left part of the figure),



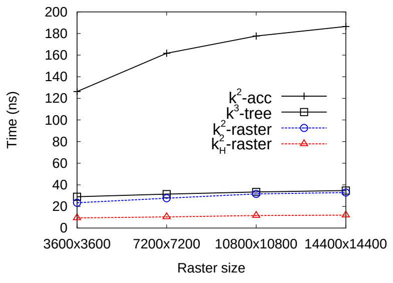
(a) $\mathtt{MDT}_x$ – *getCell*



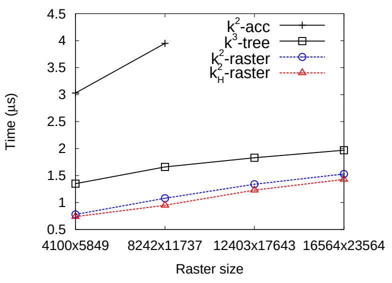
(b) $\mathtt{MDT}_x$ – *getWindow*



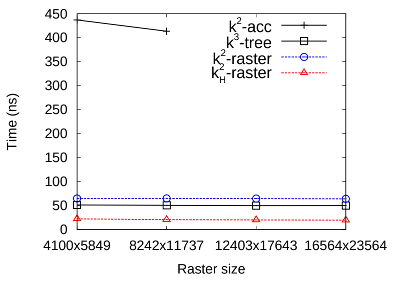
(c) `eua` – *getCell*



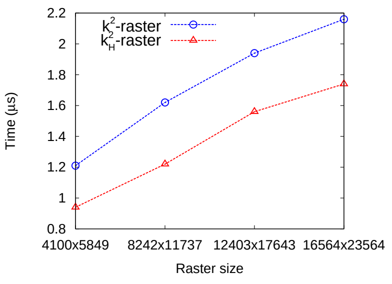
(d) `eua` – *getWindow*



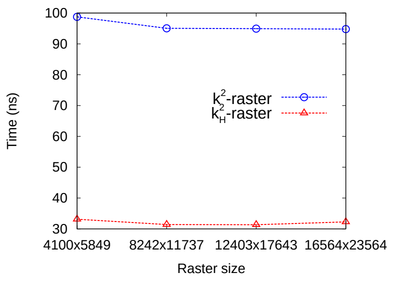
(e) $\mathtt{cat}_0$ – *getCell*



(f) $\mathtt{cat}_0$ – *getWindow*

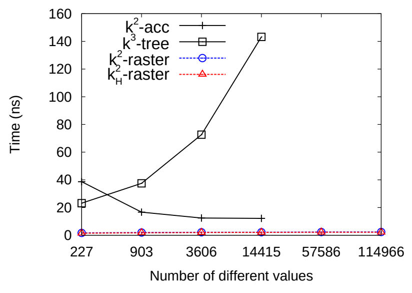


(g) $\mathtt{cat}_3$ – *getCell*

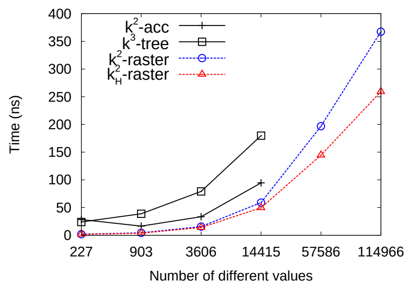


(h) $\mathtt{cat}_3$ – *getWindow*

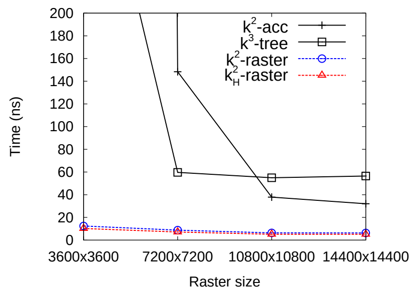
Figure 10: Time results for *getCell* (left) and *getWindow* (right) over datasets with different size and number of different values. We show average time per cell retrieved in microseconds for *getCell* and nanoseconds for *getWindow*.



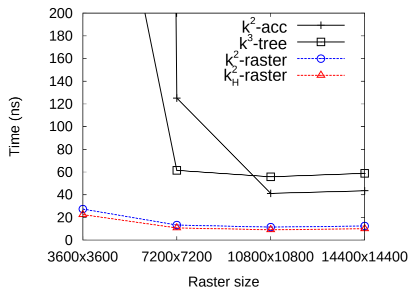
(a) $\texttt{MDT}_x$ – no restriction



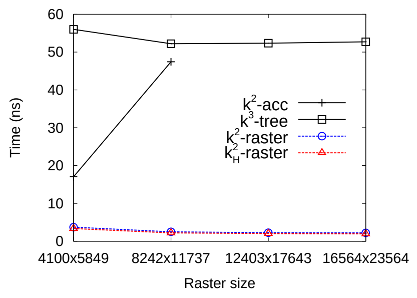
(b) $\texttt{MDT}_x$ – restricted



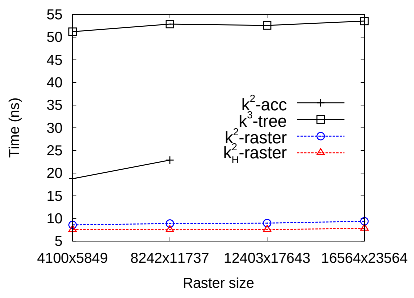
(c) **eua** – no restriction



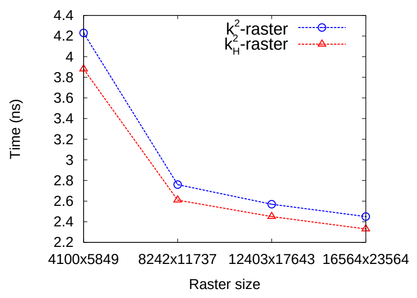
(d) **eua** – restricted



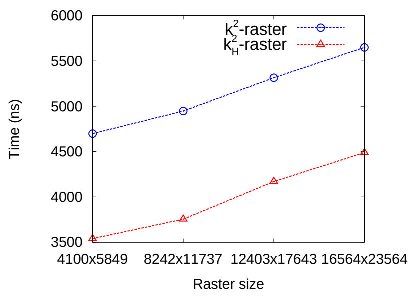
(e) $\texttt{cat}_0$ – no restriction



(f) $\texttt{cat}_0$ – restricted



(g) $\texttt{cat}_3$ – no restriction



(h) $\texttt{cat}_3$ – restricted

Figure 11: Time results for *searchValuesInWindow* using random windows and ranges without any restriction (left) and when restricting the maximum window size to 500×500 and the range length to 200 (right). Time results are measured in nanoseconds per retrieved cell.

and when limiting the range length to 200 and the window size to 500 × 500 (right). We distinguish these two distinct scenarios, as time results show different behaviors. When selecting random ranges without any restriction, these ranges become larger when the number of different values grows. Thus, if the range is large, the query is usually answered in the upper levels of the representation, as there exist a vast amount of valid values that meet the search condition. In addition, the number of retrieved cells is higher, making the time/cell ratio smaller. On the other hand, if we limit the range length to 200, we avoid these two effects; thus, searching times worsen as the number of different values in the raster matrix grows, as the query becomes more selective. Collection `eua` contains very few different values (less than 500 different values); thus, all techniques behave similarly in these two scenarios, as restricting the range length to 200 produces almost no effect.

Our solutions perform better than the state of the art in all cases. With the indexation of the minimum and maximum values in the nodes of the tree, our structures are able to determinate if a region has any valid cell or even if all cells lie within the given range of values by only checking one node; in other case, they skip that node and continue the process with the rest of tree. Comparing the techniques from the state of the art, k^2-acc gets better results than k^3-tree in most datasets, especially when the number of different values is high, as it is shown in Figure 11(a). This is due to the fact that the k^2-acc only needs to check two k^2-trees, that is, the k^2-tree of the minimum value and the k^2-tree of the maximum value of the given range.

Time of checkValuesInWindow

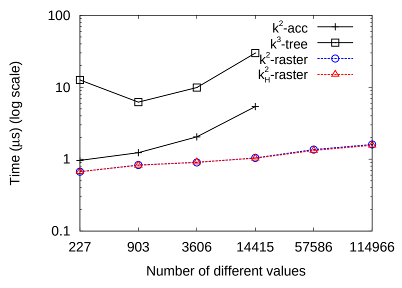
Figure 12 shows the time to check if there exists at least one cell in the region whose value lies within the range of values (left) or if all cells of the region are within the range of values (right).

For the first case, which corresponds to *weak checkValuesInWindow*, k^2-*raster*, k^2_H-*raster*, and k^2-acc obtain very close results. k^2-acc obtains the best time for some datasets containing a small number of different values, more specifically, for datasets from collection `eua`, and some from collection $\texttt{cat}_0$. However k^2-*raster*, and k^2_H-*raster* are able to answer this query efficiently over datasets with a large number of different values or a large size. k^3-tree runs up to 40 times slower. This is the unique query where the standard k^2-*raster* is faster than k^2_H-*raster* in some case.

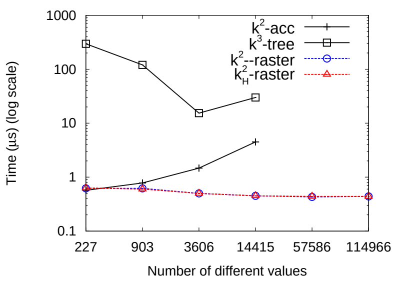
In the case of *strong checkValuesInWindow*, k^2_H-*raster*, k^2-*raster* and k^2-acc obtain similar results, while k^3-tree behaves constantly worse. Our structures use the information on the nodes (the maximum and minimum values) to check if the cells meet the conditions of the query. They are generally able to answer a query in the upper levels of the tree, without the need to descend to the last levels. This is the reason why k^2-*raster* obtains very similar results to those of k^2_H-*raster* for most of the datasets, as they only differ in the last levels of representation, which is rarely accessed in this query.

5.4. Comparison with netCDF

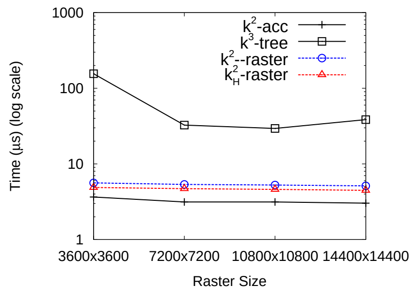
In this section we present a brief comparison of the heuristic k^2-*raster* (k^2_H-*raster*) with netCDF, using only datasets $\texttt{cat}_0$ and $\texttt{cat}_3$. This experiment is intended to give an idea of the differences between a classic compression approach and a compact data structure.



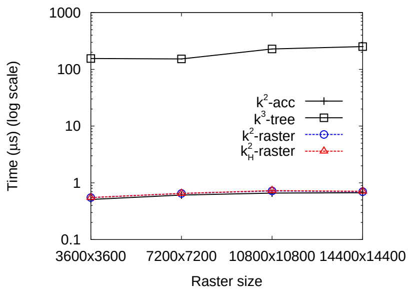
(a) $\texttt{MDT}_x$ – weak *checkValuesInWindow*



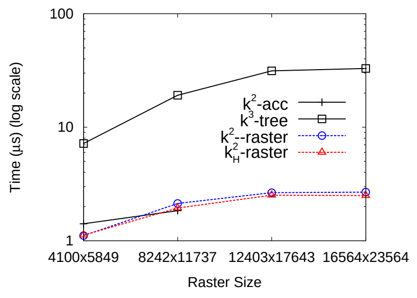
(b) $\texttt{MDT}_x$– strong *checkValuesInWindow*



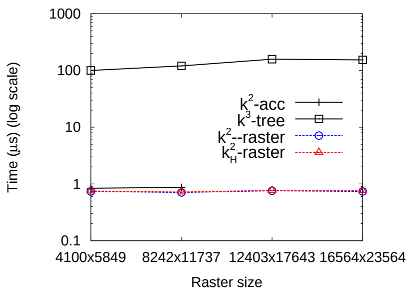
(c) $\texttt{eua}$ – weak *checkValuesInWindow*



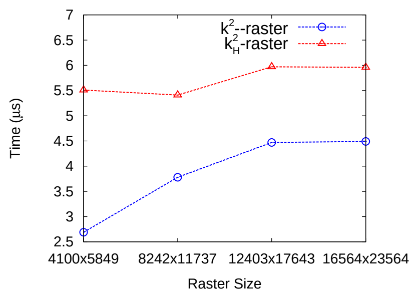
(d) $\texttt{eua}$ – strong *checkValuesInWindow*



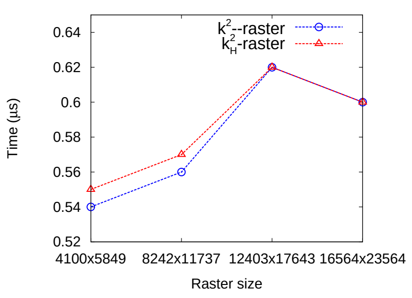
(e) $\texttt{cat}_0$ – weak *checkValuesInWindow*



(f) $\texttt{cat}_0$ – strong *checkValuesInWindow*

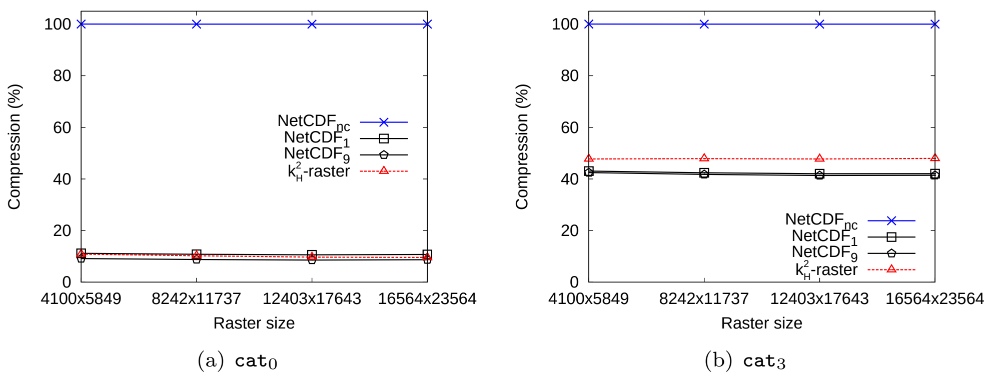


(g) $\texttt{cat}_3$ – weak *checkValuesInWindow*



(h) $\texttt{cat}_3$ – strong *checkValuesInWindow*

Figure 12: Time results for weak (left) and strong (right) *checkValuesInWindow*. Time results are measured in microseconds per query.



(a) $\mathtt{cat}_0$ (b) $\mathtt{cat}_3$

Figure 13: Compression percentage of k^2_H*-raster* and netCDF.

To compress, netCDF uses Deflate, which can be configured in ten compression levels. Level 0 means no compression, and thus faster access times, whereas level 9 obtains the best compression at the cost of slower access times. In any case, to obtain a given datum, Deflate must start the decompression at the beginning of stream of data, which is precisely the situation that compact data structures avoid.

This comparison is a bit difficult to address, since the two techniques were designed with completely different approaches. k^2_H*-raster* was specifically designed to take advantage of the speed of the upper levels of the memory hierarchy, whereas netCDF follows a classic disk-based approach. Therefore, the comparison could be somewhat unfair. In order to try to be as fair as possible, we consider only *user times*, which do not include the times to access disk, therefore the effect is as the netCDF read the data from main memory.

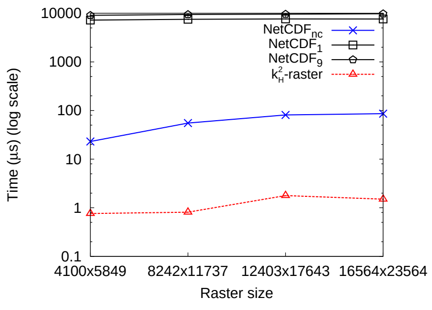
5.4.1. Compression

Figure 13 shows the compression percentage obtained by k^2_H*-raster* and netCDF with different levels of compression, namely: using no compression, level 1, and level 9 (denoted $\text{NetCDF}_{\text{nc}}$, NetCDF_1, and NetCDF_9 respectively). We can observe that in the most compressible file $\mathtt{cat}_0$, k^2_H*-raster* is basically on a par with netCDF, more precisely, it obtains ratios similar to those obtained by netCDF between levels 1 and 3 of Deflate. When compressing $\mathtt{cat}_3$, k^2_H*-raster* is a bit worse than netCDF. While netCDF obtains compression ratios around 41–42%, k^2_H*-raster* obtains compression ratios close to 48%.

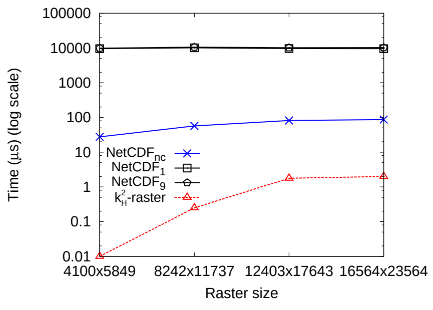
Thus, k^2_H*-raster* achieves compression ratios comparable to those obtained by netCDF, being just around 10–15% worse than the netCDF configured with the maximum level of compression. However, we will see in the following section that k^2_H*-raster* obtains query times that are orders of magnitude better in some cases.

5.4.2. Query times

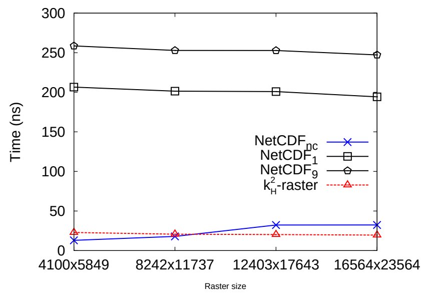
Figure 14 shows time results for performing different queries over the two datasets, using both k^2_H*-raster* and netCDF. It is noticeable that k^2_H-raster obtains better results for almost all the queries, even when compared with the netCDF variant using no compression, which requires much more space.



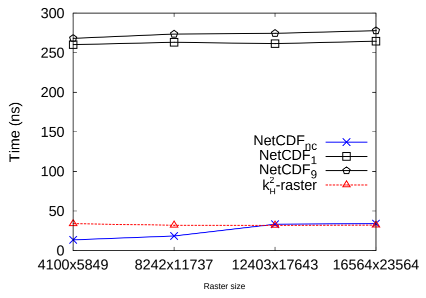
(a) $\mathtt{cat}_0$ – *getCell*



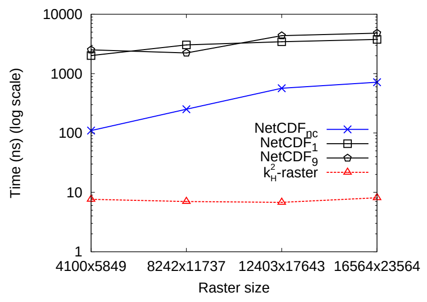
(b) $\mathtt{cat}_3$ – *getCell*



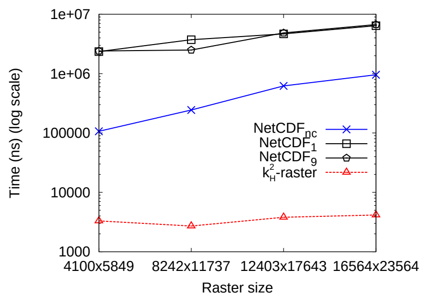
(c) $\mathtt{cat}_0$ – *getWindow*



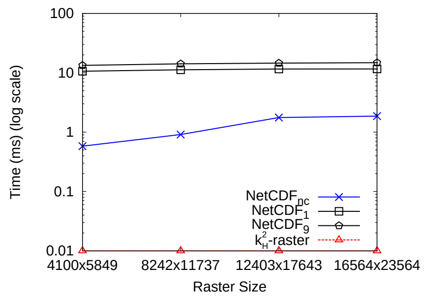
(d) $\mathtt{cat}_3$ – *getWindow*



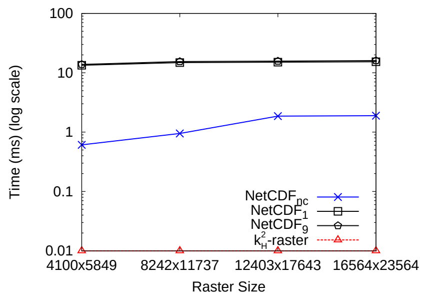
(e) $\mathtt{cat}_0$ – *searchValuesInWindow*



(f) $\mathtt{cat}_3$ – searchValuesInWindow



(g) $\mathtt{cat}_0$ – weak *checkValuesInWindow*



(h) $\mathtt{cat}_3$ – weak *checkValuesInWindow*

Figure 14: Query time results of the comparison of k_H^2*-raster* with netCDF.

In *getCell*, k^2_H-*raster* is between 30 and 67 times faster than the uncompressed version of netCDF. The reason of this surprising result is the compact data structure approach.

Given that obtaining just a cell is so fast that it could not be measured with the *time* linux system call, we have to perform 1,000,000 queries. This also gives a good average value for the *getCell* query on a k^2_H-*raster*, since the time for obtaining cells can vary significantly depending on the level where the downward traversal that solves the query stops. Observe that when retrieving a cell that is within a uniform submatrix (all the cells of the submatrix have the same value), the query can stop potentially in a level quite close to the root, thus requiring a short and fast traversal.

After solving several queries (of the 1,000,000 queries that are executed during the experiment), it is likely that the algorithms controlling the CPU caches keep, at least, the upper levels of the k^2_H-*raster*, since all searches start at the root of the tree. However, netCDF retrieves one disk page per cell request. Given that queries are random, and in netCDF the cells of the raster are placed sequentially in an array, each cell query may require a different disk page. Moreover, after returning from a system call that retrieves a disk page containing the searched cell, the retrieved data is in RAM, which is significantly slower than in the case of CPU cache accesses. Observe that this is not an unfair situation for netCDF. Compact data structures require a complex design in order to keep the data always compressed, precisely to make a better usage of the memory hierarchy by placing the data as close to the CPU as possible.

Moreover, the comparison of k^2_H-*raster* with the compressed versions of netCDF shows overwhelming results, as k^2_H-*raster* is between 3 and 4 orders of magnitude faster. This is due to the fact that netCDF requires a decompression from the beginning of the file; thus, the portion of compressed data preceding the requested cell must be loaded into memory and decompressed.

In the case of *getWindow*, k^2_H-*raster* and the uncompressed netCDF are basically on a par. For small files, netCDF obtains slightly better times, whereas for bigger files, it is the opposite situation. Contrary to what happened when solving *getCell*, *getWindow* outputs several values of cells that are contiguous in the raster matrix; thus, the uncompressed version of netCDF outperforms k^2_H-*raster* in some cases, as these contiguous values are trivially obtained after few disk accesses in the case of netCDF, but require a more complex navigation in the case of k^2_H-raster. This behavior is also noticeable when using netCDF with compression, as it takes advantage of the spatial locality when accessing the compressed file. In any case, k^2_H-*raster* is still around 8 times faster than these compressed representations.

For *searchValuesInWindow* and *checkValuesInWindow*, k^2_H-*raster* is orders of magnitude faster than netCDF, thanks to its indexing capabilities.

6. Conclusions

In this article we propose a new storage structure, denoted k^2-*raster*, which represents raster data in compressed form and offers efficient indexing capabilities. Our technique supports, within reduced space, fast retrieval of single cell values, decompression of regions of cells and also supports advanced searches, such as retrieving cells inside a region containing some specific value or checking the existence of values inside regions of the raster data. We have also presented a variant of the structure, called k^2_H-*raster*, which uses an entropy-based heuristic to create a vocabulary of common patterns in order to obtain further compression.

We have empirically compared our two variants to existing techniques from the literature, showing that both proposals clearly outperform other compact data structures also designed for representing and indexing raster datasets. They not only obtain better space usage and query performance, but they also scale better when increasing the size of the input data or when the raster matrix contains a large number of different values. The scalability property is of extreme importance, as these characteristics appear when using real raster data. When comparing the two proposed variants, k^2_H-*raster* is the clear choice in all scenarios. The simpler proposal, k^2-*raster*, obtains better construction times, but the heuristic version obtains better spatio-temporal results.

We also showed that compared with netCDF, a classical method to store rasters, k^2_H-*raster* obtains comparable, or at least close, compression ratios. In addition, thanks to the ability to access a given datum without decompressing the rest of the data and the indexing capabilities, k^2_H-*raster* obtains much better access and query times, being in some cases orders of magnitude faster.

As future work, we will extend the proposed structure to other dimensions, for instance, to be used for spatio-temporal or 3-D datasets. In addition, we will also study the adaptation of our data structure to distributed or dynamic environments, and also its applicability for storing and querying very large raster matrices on external storage.

Acknowledgments

This work was supported by Ministerio de Economía y Competitividad (PGE and FEDER) under grants [TIN2016-78011-C4-1-R; TIN2016-77158-C4-3-R; TIN2013-46238-C4-3-R; TIN2013-46801-C4-3-R], Centro para el desarrollo Tecnológico e Industrial Programa CIEN 2014 (co-founded with FEDER) [IDI-20141259; ITC-20151247]; Xunta de Galicia (co-founded with FEDER) under grant GRC2013/053.

References

[1] S. Ladra, J. R. Paramá, F. Silva-Coira, Compact and queryable representation of raster datasets, in: Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), 2016, pp. 15:1–15:12. doi:10.1145/2949689.2949710.

[2] H. Couclelis, People manipulate objects (but cultivate fields): Beyond the raster-vector debate in GIS, in: Proceedings of GIS: from space to territory - theories and methods of spatio-temporal reasoning, 1992, pp. 65–77. doi:10.1007/3-540-55966-3_3.

[3] M. Worboys, M. Duckham, GIS: A computing perspective, CRC press, 2004.

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

[5] Y. Li, T. R. Bretschneider, Semantic-Sensitive Satellite Image Retrieval, IEEE Trans. on Geosci. and Remote Sens. 45 (2007) 853–860. doi:10.1109/TGRS.2007.892008.

[6] M. Quartulli, I. G. Olaizola, A review of EO image information mining, ISPRS J. Photogramm. Remote Sens 75 (2013) 11–28. doi:10.1016/j.isprsjprs.2012.09.010.

[7] G. K. Wallace, The JPEG still picture compression standard, Commun. ACM 34 (1991) 30–44. doi:10.1145/103085.103089.

[8] P. Lindstrom, M. Isenburg, Fast and Efficient Compression of Floating-Point Data, IEEE Trans. Vis. Comput. Graph. 12 (2006) 1245–1250. doi:10.1109/TVCG.2006.143.

[9] C. Lee, M. Yang, R. Aydt, Netcdf-4 performance report, Tech. rep., Technical report, HDF Group (2008).

[10] G. Jacobson, Succinct static data structures, Ph.D. thesis, Carnegie-Mellon (1988).

[11] G. Navarro, Compact Data Structures – A practical approach, Cambridge University Press, 2016.

[12] M. Burtscher, P. Ratanaworabhan, FPC: A High-Speed Compressor for Double-Precision Floating-Point Data, IEEE Trans. Comput. 58 (2009) 18–31. doi:10.1109/TC.2008.131.

[13] J. Zhang, S. You, L. Gruenwald, Quadtree-based lightweight data compression for large-scale geospatial rasters on multi-core CPUs, in: Proceedings of the IEEE International Conference on Big Data (IEEE Big Data), 2015, pp. 478–484. doi:10.1109/BigData.2015.7363789.

[14] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (2007) 2. doi:10.1145/1216370.1216372.

[15] R. Raman, S. S. Rao, Succinct Representations of Ordinal Trees, in: Proceedings of the Space-Efficient Data Structures, Streams, and Algorithms, LNCS 8066, 2013, pp. 319–332. doi:10.1007/978-3-642-40273-9_20.

[16] P. Howard, J. Vitter, Fast and efficient lossless image compression, in: Proceedings of the Data Compression Conference (DCC), 2014, pp. 351–360. doi:10.1109/DCC.1993.253114.

[17] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, N. F. Samatova, ISOBAR Preconditioner for Effective and High-throughput Lossless Data Compression, in: Proceedings of the IEEE International Conference on Data Engineering (ICDE), 2012, pp. 138–149. doi:10.1109/ICDE.2012.114.

[18] H. Samet, The Quadtree and Related Hierarchical Data Structures, ACM Comput. Surv. 16 (1984) 187–260. doi:10.1145/356924.356930.

[19] B. Duvenhage, Using an implicit min/max KD-tree for doing efficient terrain line of sight calculations, in: Proceedings of the International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (AFRIGRAPH), Vol. 1, New York, New York, USA, 2009, p. 81. doi:10.1145/1503454.1503469.

[20] J. Zhang, S. You, Supporting Web-Based Visual Exploration of Large-Scale Raster Geospatial Data Using Binned Min-Max Quadtree, in: Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM), 2010, pp. 379–396. doi:10.1007/978-3-642-13818-8_27.

[21] A. Klinger, Pattern and search statistics, Academic Press, 1971. doi:10.1016/B978-0-12-604550-5.50019-5.

[22] A. Klinger, C. R. Dyer, Experiments on picture representation using regular decomposition, Comput. Graph. Image Process. 5 (1976) 68–105. doi:10.1016/S0146-664X(76)80006-8.

[23] G. de Bernardo, S. Álvarez-García, N. R. Brisaboa, G. Navarro, O. Pedreira, Compact Querieable Representations of Raster Data, in: Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE), 2013, pp. 96–108. doi:10.1007/978-3-319-02432-5_14.

[24] N. R. Brisaboa, S. Ladra, G. Navarro, Compact representation of Web graphs with extended functionality, Inf. Syst. 39 (2014) 152–174. doi:10.1016/j.is.2013.08.003.

[25] R. Rew, G. Davis, Netcdf: an interface for scientific data access, IEEE Computer Graphics and Applications 10 (4) (1990) 76–82. doi:10.1109/38.56302.

[26] L. P. Deutsch, RFC 1951: DEFLATE compressed data format specification version 1.3 (May 1996).

[27] H. Samet, Foundations of Multimensional and Metric Data Structures, Morgan Kaufmann, 2006.

[28] H. Samet, Data structures for quadtree approximation and compression, Commun. the ACM 28 (1985) 973–993. doi:10.1145/4284.4290.

[29] M. a. Oliver, Operations on Quadtree Encoded Images, Comput. J. 26 (1983) 83–91. doi:10.1093/comjnl/26.1.83.

[30] M. Seidemann, B. Seeger, ChronicleDB: A high-performance event store, in: Proceedings of the 20th International Conference on Extending Database Technology (EDBT), 2017, pp. 144–155.

[31] G. Moerkotte, Small Materialized Aggregates: A light weight index structure for data warehousing, in: Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB), 1998, pp. 476–487.

[32] M. Athanassoulis, A. Ailamaki, BF-tree: Approximate tree indexing, Proceedings of the VLDB Endowment 7 (14) (2014) 1881–1892. doi:10.14778/2733085.2733094.

[33] P. Francisco, IBM PureData System for Analytics Architecture: A Platform for High Performance Data Warehousing and Analytics, IBM Redbooks.

[34] The PostgreSQL Global Development Group, BRIN indexes, PostgreSQL 9.5.7 Documentation. Chapter 62.

[35] R. Weiss, A technical overview of the oracle exadata database machine and exadata storage server,

Tech. rep., Oracle Corporation (2012).

[36] J. R. Woodwark, Compressed Quad Trees, Comput. J. 27 (1984) 225–229. doi:10.1093/comjnl/27.3.225.

[37] T.-W. Lin, Compressed quadtree representations for storing similar images, Image Vis. Comput. 15 (1997) 833–843. doi:10.1016/S0262-8856(97)00031-0.

[38] Y.-K. Chan, Block image retrieval based on a compressed linear quadtree, Image Vis. Comput. 22 (2004) 391–397. doi:10.1016/j.imavis.2003.12.003.

[39] K.-L. Chung, Y.-W. Liu, W.-M. Yan, A hybrid gray image representation using spatial- and DCT-based approach with application to moment computation, J. Vis. Commun. Image Represent. 17 (2006) 1209–1226. doi:10.1016/j.jvcir.2006.01.002.

[40] J. Zhang, S. You, High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives, Int.l J. Geogr. Inf. Sci. 27 (2013) 2207–2226. doi:10.1080/13658816.2013.828840.

[41] I. Gargantini, An effective way to represent quadtrees, Communications of the ACM 25 (12) (1982) 905–910.

[42] D. Abel, J. Smith, A data structure and algorithm based on a linear key for a rectangle retrieval problem, Computer Vision, Graphics, and Image Processing 24 (1) (1983) 1 – 13. doi:http://dx.doi.org/10.1016/0734-189X(83)90017-8.
URL `http://www.sciencedirect.com/science/article/pii/0734189X83900178`

[43] H. Samet, A. Rosenfeld, C. A. Shaffer, R. E. Webber, A geographic information system using quadtrees, Pattern Recognition 17 (6) (1984) 647 – 656. doi:http://dx.doi.org/10.1016/0031-3203(84)90018-9.
URL `http://www.sciencedirect.com/science/article/pii/0031320384900189`

[44] J. Zhang, S. You, L. Gruenwald, Indexing large-scale raster geospatial data using massively parallel GPGPU computing, in: Proceedings of the International Conference on Advances in Geographic Information Systems (SIGSPATIAL), 2010, p. 450. doi:10.1145/1869790.1869859.

[45] G. Jacobson, Space-efficient static trees and graphs, in: Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS), 1989, pp. 549–554. doi:10.1109/SFCS.1989.63533.

[46] P. Elias, Universal codeword sets and representations of the integers, IEEE Transactions on Information Theory 21 (2) (1975) 194–203.

[47] S. W. Golomb, Run-length encodings, IEEE Trans. Inform. Theory IT-12 (1966) 399–401.

[48] H. E. Williams, Compressing Integers for Fast File Access, The Computer Journal 42 (3) (1999) 193–201. doi:10.1093/comjnl/42.3.193.

[49] V. Ngoc Anh, A. Moffat, Inverted index compression using word-aligned binary codes, Information Retrieval 8 (1) (2005) 151–166.

[50] M. Zukowski, S. Héman, N. Nes, P. A. Boncz, Super-scalar RAM-CPU cache compression, in: Proceedings of the International Conference on Data Engineering, (ICDE'06), IEEE Computer Society, 2006, pp. 59–71.

[51] J. I. Munro, Tables, in: Proceedings of Foundations of Software Technology and Theoretical Computer Science, Vol. 1180, 1996, pp. 37–42.

[52] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proceedings of the Workshop on Algorithm Engineering and Experiments, (ALENEX'07), 2007.

[53] R. Grossi, A. Gupta, J. S. Vitter, High-Order Entropy-Compressed Text Indexes, in: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, Vol. 2068 of SODA '03, 2003, pp. 841–850.

[54] N. R. Brisaboa, A. Faria, S. Ladra, G. Navarro, Reorganizing compressed text, in: Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08), Singapore, 2008, pp. 139–146.

[55] J. Teuhola, Interpolative coding of integer sequences supporting log-time random access, Information Processing Management 47 (5) (2011) 742–761.

[56] M. O. Külekci, Enhanced variable-length codes: Improved compression with efficient random access, in: Proceedings Data Compression Conference, (DCC'14), Snowbird, UT, USA, 26-28 March, 2014, 2014, pp. 362–371.

[57] S. T. Klein, D. Shapira, Random access to fibonacci encoded files, Discrete Applied Mathematics 212 (2016) 115 – 128, stringology Algorithms.

[58] N. R. Brisaboa, S. Ladra, G. Navarro, DACs: Bringing direct access to variable-length codes, Inf. Process. Manag. 49 (2013) 392–404. doi:10.1016/j.ipm.2012.08.003.

[59] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, S. S. Rao, Representing trees of higher degree, Algorithmica 43 (4) (2005) 275–292.

[60] G. Navarro, K. Sadakane, Fully functional static and dynamic succinct trees, ACM Trans. Algorithms 10 (3) (2014) 16:1–16:39.

[61] S. Álvarez-García, N. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, G. Navarro, Compressed vertical partitioning for efficient RDF management, Knowl. Inf. Syst. 44 (2015) 439–474. doi:10.1007/s10115-014-0770-y.

[62] M. Romero, N. Brisaboa, M. A. Rodríguez, The SMO-index: a succinct moving object structure for timestamp and interval queries, in: Proceedings of the International Conference on Advances in Geographic Information Systems (SIGSPATIAL), 2012, p. 498. doi:10.1145/2424321.2424399.

[63] N. R. Brisaboa, A. Gómez-Brandón, G. Navarro, J. R. Paramá, GraCT: A Grammar Based Compressed Representation of Trajectories, in: Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE), 2016, pp. 218–230. doi:10.1007/978-3-319-46049-9_21.

[64] S. Álvarez, N. R. Brisaboa, S. Ladra, Ó. Pedreira, A compact representation of graph databases, in: Proceedings of the Eighth Workshop on Mining and Learning with Graphs (MLG), 2010, pp. 18–25. doi:10.1145/1830252.1830255.

[65] H. Garcia-Molina, J. D. Ullman, J. Widom, Database systems - the complete book (2. ed.), Pearson Education, 2009.

[66] D. Salomon, Data Compression: The Complete Reference, Springer, 2004.

[67] C. E. Shannon, A mathematical theory of communication, Bell System Technical Journal 27 (1948) 370–423,623–656.

[68] R. J. Hijmans, S. E. Cameron, J. L. Parra, P. G. Jones, A. Jarvis, Very high resolution interpolated climate surfaces for global land areas, Int. J. Climatol. 25 (2005) 1965–1978. doi:10.1002/joc.1276.

[69] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical Implementation of Rank and Select Queries, in: Poster Proceedings Volume of the Workshop on Efficient and Experimental Algorithms (WEA), Vol. 0109, 2005, pp. 27–38.