Two-Dimensional Block Trees [1]

Nieves R. Brisaboa[a], Travis Gagie[b], Adrián Gómez-Brandón[a,*], Gonzalo Navarro[c]

[a] *Universidade da Coruña, CITIC, Facultade de Informática, 15071 A Coruña, Spain*
[b] *Faculty of Computer Science, Dalhousie University, Halifax, Canada*
[c] *Millennium Institute for Foundational Research on Data, Department of Computer Science, University of Chile, Beauchef 851, Santiago, Chile*

Abstract

The Block Tree is a data structure for representing repetitive sequences in compressed space, which reaches space comparable to that of Lempel-Ziv compression while retaining fast direct access to any position in the sequence. In this paper we generalize Block Trees to two dimensions, in order to exploit repetitive patterns in the representation of images, matrices, and other kinds of bidimensional data. We demonstrate the practicality of the two-dimensional Block Trees (2D-BTs) in representing the adjacency matrices of Web graphs, and raster images in GIS applications. For this purpose, we integrate our 2D-BT with the k^2-tree –an efficient structure that exploits clustering and sparseness to compress adjacency matrices– so that it also exploits repetitive patterns. Our experiments show that this structure uses 60%–80% of the space of the original k^2-tree, while being from 30% faster to 3 times slower when accessing cells.

Keywords: Block Trees, k^2-trees, graph compression, image compression.

1. Introduction

Various kinds of applications require handling large amounts of bidimensional data containing a significant amount of repeated areas. In this article we do not focus on continuous image data where near-repetitiveness arises, like video frames or periodical sky surveys, for which other kinds of representations are used, typically lossy ones. Instead, we focus on scenarios featuring discrete bidimensional data, which we must represent exactly but where we can still exploit repetitiveness. Further, we aim not at mere compression, but at representations that can be accessed and queried directly without decompression. These so-called *compressed data structures* [21] enable applications work directly on the compressed data without ever having to decompress it.

In terms of compression, a technique that exploits bidimensional repetitions on discrete matrices is the two-dimensional Lempel-Ziv compression format [19] (2D-LZ), which

[1] An early partial version of this article appeared in *Proc. Data Compression Conference 2018*.

* Corresponding author. Tel. +34981167000 Fax. +34981167160.
Email addresses: brisaboa@udc.es (Nieves R. Brisaboa), travis.gagie@dal.ca (Travis Gagie), adrian.gbrandon@udc.es (Adrián Gómez-Brandón), gnavarro@dcc.uchile.cl (Gonzalo Navarro)

stores only the first occurrence of each sub-matrix in a dictionary and represents the others as pointers to that reference. Although 2D-LZ can obtain good compression, it cannot support efficient access to any submatrix from a compressed collection of matrices without decompressing all the preceding data. Some proposals [1, 22] trade compression effectiveness for direct access performance by partitioning the matrices into squares of fixed size and compressing them independently.

In this paper we study a more principled approach that builds on a representation for one-dimensional sequences called Block Trees (BTs) [4]. BTs exploit repetitiveness and obtain compression ratios close to those of (one-dimensional) Lempel-Ziv [18], though they retain fast random access to any part of the sequence. Our proposal, *Two-Dimensional Block Trees (2D-BTs)*, generalizes the BT concept to two dimensions, thereby providing a compressed representation for discrete repetitive bidimensional data that retains direct access to any compressed submatrix.

A prominent case where this kind of data arises are the adjacency matrices of *Web graphs*, that is, directed graphs of pages pointing to other pages on the Web. The corresponding matrices contain 1s at the positions (p, q) such that the page p has a link to page q. Such matrices are considerably sparse and clustered, and furthermore contain large identical submatrices (consider mirror sites, for example). Various techniques have been developed to exploit those properties [2, 8, 13, 14, 23]. Their way to exploit repeated areas, however, is usually limited to row-wise (i.e., adjacency list) similarity, not to whole equal submatrices. Further, the main access operation they provide is to extract the *direct neighbors* of a page (i.e., extract an adjacency list, or a matrix row), and do not implement efficient ways to handle other useful operations like determining whether a given page points to another (i.e., accessing a single cell of the matrix) or extracting all the links between two sites (i.e., retrieving a rectangular area of the matrix).

An elegant representation of Web graphs that supports those operations efficiently are the k^2-trees [9]. These exploit the sparseness and clustering of adjacency matrices, but do not exploit similarity of adjacency lists or submatrices. Bille et al. [5] replace identical subtrees in the k^2-tree, converting it into a DAG, which goes in the line of exploiting similarity. It is unlikely, however, that identical submatrices are aligned with subtrees. Our combination of *2D-BT*s with k^2-trees, instead, can capture any repetition of a submatrix, even if it is not aligned to another subtree. We present a construction algorithm that makes it possible to build the combined structure on large Web graphs. In our experimental evaluation, combining k^2-trees with *2D-BT*s reduces the space to 60%–80% of the original k^2-trees, while being from 30% faster to 3 times slower at accessing cells. When accessing larger regions, the *2D-BT* takes from similar time to an order of magnitude more than k^2-trees.

The ability of k^2-trees to retrieve the values of arbitrary submatrices makes them useful for the compressed representation of binary matrices arising in other areas, like GIS applications. For example, a k^2-tree variant called k^2-acc [10] is used to represent raster numeric data. Usually, this kind of data is stored as a matrix of integers where each cell represents an area of the space and stores the value of a measurement (e.g., temperature, altitude) in that location. The k^2-acc represents this data with a binary matrix marking all the cells smaller than each possible threshold value. Those binary matrices are compressed with k^2-trees that not only exploit submatrices full of 0s, but also full of 1s. A combination of *2D-BT*s with the k^2-acc uses in our experiment 2.0–2.7 times less space, at the expense of being 1.2–1.8 times slower to access the data.

2. Background

A basic data structure lying at the core of the compact data structures we review in this section is the *bitmap*, also called *bitvector*. A bitmap $B[1..n]$ is an array of n bits, where each entry has two possible values: 1 or 0, and thus can be represented with only one bit. Bitmaps typically support two operations: *rank* and *select*. Given an integer i and a bit $b \in \{1, 0\}$, $rank_b(B, i)$ returns the number of times value b occurs in B up to position i. Instead, $select_b(B, i)$ computes the position of the i-th occurrence of b in B. Both operations can be supported in $O(1)$ time by using an extra space of $o(n)$ bits [20] on top of the n bits used to represent B itself.

In the rest of the section we use bitmaps as building blocks of various structures that represent Web graphs, numeric matrices, and repetitive sequences.

2.1. Web graphs

Various characteristics of Web graphs have been exploited in order to compress them. Most of the methods that offer direct access focus on extracting the whole adjacency lists of nodes, and thus focus on compression of those lists.

The *WebGraph Framework* [8] is the most famous technique for the compression of Web graphs. It exploits three properties of adjacency lists: *power-law distribution* (of the number of incoming and outcoming edges of nodes), *locality of reference* (most lists reference nearby nodes) and the *copy property* (some lists are very similar to other lists).

Several other methods improved upon WebGraph with different techniques. Apostolico and Drovandi [2] (*AD*) sort the nodes in BFS order, which promotes further locality than the lexicographic URL order used by standard WebGraph, leading to sometimes better compression with comparable extraction performance. The same WebGraph group obtained better results with other node orderings, like Layered Level Propagation (*LLP*) [7]. Another technique called List Merging (*LM*) [13], merges several adjacency lists into a block. The most recent method, *Zuckerli* [23], applies new compression techniques and heuristics on top of *WebGraph*, improving its compression.

2.1.1. The k^2-tree

The previous methods can efficiently compute only the direct neighbors of a node. The k^2-tree [9] is a compact data structure designed to compress adjacency matrices of size n^2, by exploiting the sparseness and clustering of 1s. The k^2-tree, instead, can effciently check the presence of a link between two nodes, and in general can extract all the links in a two-dimensional range.

The data structure is a k^2-ary tree where each node represents a submatrix of the adjacency matrix. For simplicity, the size n is virtually extended to the next power of k. The root then represents the whole matrix and each of its children represent a submatrix of size n^2/k^2. If there is some 1 inside a submatrix, its node is represented with a bit 1; otherwise, it is represented with a 0. These nodes are sorted in row-major order, left to right and top to bottom. Once the first level is built, the procedure continues recursively splitting the submatrices with 1s into smaller k^2 submatrices. Note that a submatrix is not split further when it is full of 0s.

The k^2-tree is stored without pointers, using two bitmaps: $T[1..]$ and $L[1..]$. The values of all the levels except the last one are represented in T. Those bits are appended to T following a level-wise traversal of the tree. The last level is stored in bitmap L and

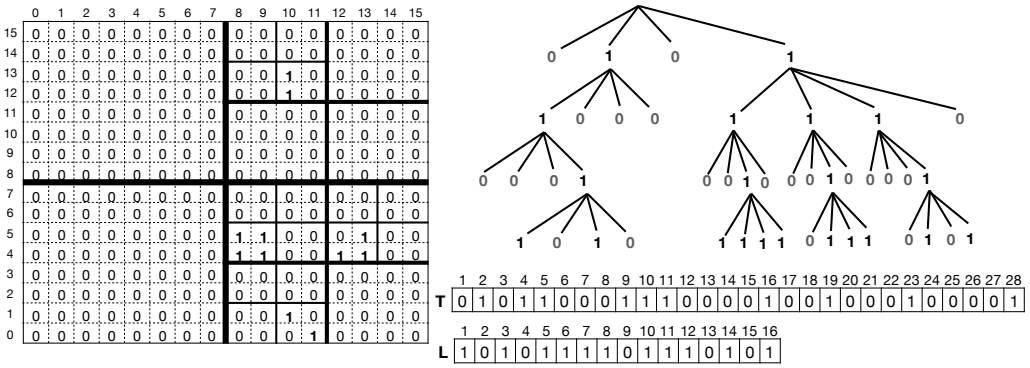

Figure 1: Example of a k^2-tree and its pointerless representation.

each bit is the value of its corresponding cell. Figure 1 shows an example. The values of a cell or region of the matrix can be efficiently obtained by traversing the nonempty subtrees that intersect the query area. Each traversal step can be simulated in constant time using *rank* operations on the bitvector T. An individual cell is accessed in $O(\log_k n)$ time (i.e., proportional to the k^2-tree height), whereas the *occ* points in an area of size $p \times q$ are retrieved in time $O(p + q + (occ + 1)k \log_k n)$.

2.2. Raster data

The ability to compute the value of any matrix cell makes the k^2-tree suitable for other applications. In particular, it can be extended to represent non-binary matrices. These arise for example when storing *raster data*, where each cell of the matrix represents a location in space and stores the value of a feature (e.g, mean temperature or altitude). Two extensions of the k^2-tree for raster data are the k^2-*acc* [10] and the k^2-*raster* [17].

Consider a matrix M over an alphabet $[1..\sigma]$. The k^2-acc represents M by building a binary matrix for each alphabet value, $M_1, \ldots, M_\sigma$. If $M[x, y] = \alpha$, then the cell (x, y) of all the matrices M_i, for $1 \leq i \leq \alpha$, are set to 1. Each binary matrix is then represented with a k^2-tree, with a particular modification: submatrices full of 1s are represented with only one node, instead of being split into smaller full submatrices.

The new kind of nodes are represented in T with a 0. Now in T the 0s can represent areas full of 0s (empty) or full of 1s (full). Those are distinguished by adding another bitmap T' aligned with the 0s of T. Given $T[i] = 0$, the node is full if $T'[rank_0(T, i)] = 1$ and empty otherwise.

This representation permits computing the value of a cell, $M[x, y]$, with a binary search on the matrices, in time $O(\log_k n \log \sigma)$. Further, it allows finding the cells of M within a region R where the values are in a range $[l, u]$. This so-called *window query* is essential for many applications managing raster data. It is solved in the k^2-acc by obtaining the 1s in M_u within R, and then subtracting the 1s within R in M_{l-1}.

The k^2-raster representation splits the matrix exactly as the k^2-tree of the points, but each node also stores the minimum and maximum value in its submatrix. If at some node the minimum and maximum values are equal, then all the values within the submatrix are identical, so the structure switches to a classical k^2-tree. The minimum and maximum

values of the nodes are differentially encoded with respect to their parent. The k^2-raster obtains better compression than the k^2-acc and outperforms it at computing the value of a cell, which is done in $O(\log_k n)$ time. However, the k^2-raster is slower than the k^2-acc on window queries for large regions.

2.3. Block Tree

The Block Tree (BT) [4] is a structure that, given a text $T[1..n]$ over an alphabet $[1..\sigma]$, uses $O(z \lg(n/z) \lg n)$ bits, where z is the number of phrases produced by the Lempel-Ziv [18] parse. This structure builds a tree whose arity is defined by a parameter r. The construction algorithm virtually extends n to the next power of r. It then starts by splitting T into r substrings (blocks) of size n/r. Each block is represented by a node of the first level of the BT. Nodes can be of two kinds: internal or leaves. A node is internal when it contains or overlaps the first occurrence of its substring content; otherwise it is a leaf. Every leaf stores a pointer to the first node that matches or overlaps the first occurrence of its substring, and an offset that indicates where the substring starts in the pointed node. For the next level, each internal node is split into r sub-blocks of size n/r^2. These steps are recursively repeated until storing the pointers and offsets is more expensive than storing the substring. Figure 2 shows an example.

For each BT level d, the types of nodes are represented in a bitmap D_d whose 0s correspond to the leaves and the 1s to the internal nodes. The pointers and offsets of the leaves at that level are stored in arrays P_d and O_d. If the i-th block of level d is a leaf ($D_d[i] = 0$), then its pointer and offset are at $P_d[j]$ and $O_j[j]$, respectively, with $j = rank_0(D_d, i)$. The only nodes that store values are those of the last level.

Any symbol $T[i]$ is accessed in time $O(\log_r n)$, by traversing the BT from the root to the leaves. Let S be the string represented by the current node, at level d (at the root, $S = T$ and $d = 0$). If the node is internal, we move to the child number $\lceil i/(|S|/r) \rceil$ and update $i \leftarrow i \bmod (|S|/r)$. If the node is a leaf, instead, we obtain its pointer and offset, *ptr* and *off*. Then, i is updated to $i + off$ and the traversal continues from node *ptr* if $i \leq |S|$. Otherwise, the traversal continues by the next node, $ptr + 1$, with i updated to $i - |S|$. The BT guarantees that the nodes *ptr* and $ptr + 1$ are internal, so we can now descend. Finally, when we reach the last level we directly access the position i of the explicit string stored at the node.

For example, in Figure 2, to obtain the value at position 12, we descend up to $d = 2$, where we fall into a leaf. The desired value is located at position 4 within that leaf. Since the pointer has $off = 1$, the value is located within the pointed node at position 5. However, the maximum number of elements contained at that level is 4. Hence, the value is located at position 1 in the sibling of that node. Then, the algorithm continues from the second node at $d = 2$ until the last level, where the symbol corresponds to the first symbol of the third leaf. Since it has a pointer, the symbol is not stored at that leaf. By following the pointer and applying the offset, we know that the symbol is stored in the second position of the first node from the last level.

3. Two-Dimensional Block Trees (*2D-BT*)

In order to compress binary matrices by exploiting their repetitiveness, we extend the Block Trees to two dimensions. In this section we present this extension, called Two-

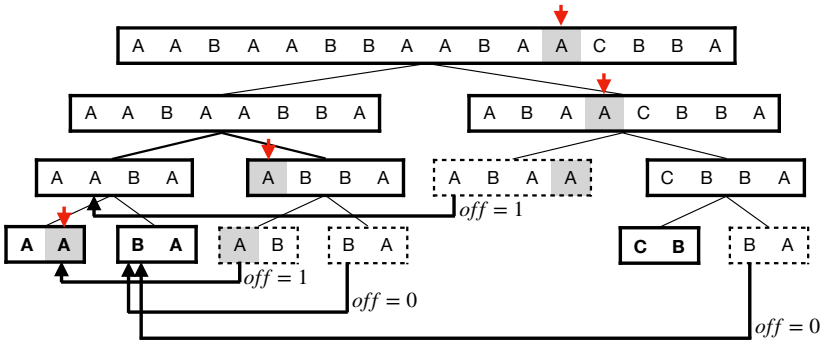

Figure 2: Example of a BT with $r = 2$.

Dimensional Block Tree (*2D-BT*), and combine it with k^2-trees so as to simultaneously exploit sparseness, clustering, and repetitiveness.

3.1. Description

Given a matrix M of size $n \times n$ over alphabet $[1 \dots \sigma]$ and a fixed parameter k, the corresponding Two-Dimensional Block Tree (*2D-BT*) is computed by splitting M into k^2 submatrices of size n^2/k^2 (again, assume n is a power of k for simplicity). Those submatrices are called blocks, and each block represents a node of the *2D-BT*. The nodes are classified into internal nodes and leaves. By considering some submatrix order (we stick to row-major order in this paper), those nodes matching or overlapping the first occurrence of a submatrix with their same content are internal; the others are leaves. The internal nodes are then recursively divided into submatrices. Instead, the submatrix of a leaf node is a copy (called a *target*), whose *source* is the first occurrence of its submatrix. The leaf will then be replaced by a pointer to its source.

Notice that a source can overlap up to four adjacent (internal) nodes. To reference its source, each leaf stores a pointer (*ptr*) and two offsets $\langle O_x, O_y \rangle$ marking the top-left cell of the source inside the block *ptr*. A target can point to a source that precedes it, in the matrix order we have chosen.

We continue this subdivision process recursively until storing the pointers and offsets in the current level is more expensive than representing its submatrices in plain form. At that point, the remaining submatrices are stored in plain form. The *2D-BT* has then a tree-shaped structure of height $h \leq \log_k n$, with some additional inter-level pointers.

Figure 3 shows an example of the *2D-BT*. The fourth node of the first level, which corresponds to the bottom-right submatrix, is a leaf node because it appears earlier, between the submatrices of the first and second node. Thus, it is represented by a pointer to the first node with offset $\langle 2, 0 \rangle$. The other three nodes of the first level are internal and thus split into submatrices. Other three nodes in the second level become leaves. The remaining 9 nodes are split again, reaching the individual cells in this case. Note that the node representing the submatrix $\left(\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}\right)$, starting at $(x, y) = (4, 2)$,[2] is not

[2]Coordinates start at zero.

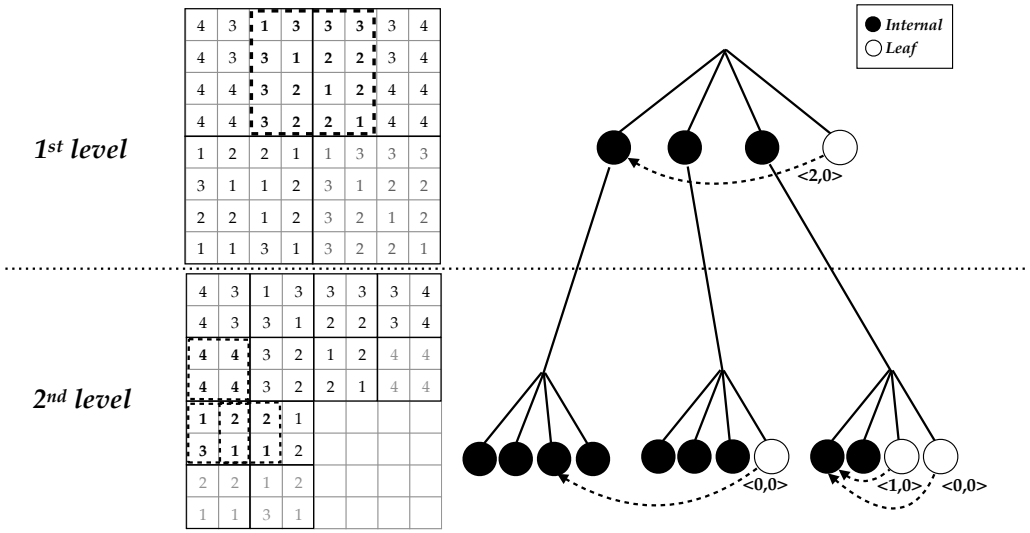

Figure 3: Example of a Two-Dimensional Block Tree with $k = 2$ and $\sigma = 4$. The left part represents the different blocks for each level and the right part shows the corresponding tree nodes.

the first occurrence of its content, which also occurs starting in $(x', y') = (3, 1)$. However, it overlaps that first occurrence, and thus it is not replaced. This is correct, as otherwise the leaf would partly reference itself.

3.2. Combination with k^2-trees

A particular case of the *2D-BT* arises when $\sigma = 2$, that is, the matrix stores bits. In many applications of binary matrices where the *2D-BT* exploits repetitiveness, the 1s also tend to be sparse and clustered. In this section we show how to incorporate the k^2-tree concepts to a *2D-BT* specialized for binary matrices, so as to simultaneously exploit repetitiveness, sparseness, and clustering.

We further exploit sparseness and clustering by adding two new kinds of leaf nodes in the *2D-BT*: *empty nodes*, which represent blocks full of zeroes; and *explicit nodes*, which represent a blocks with a single 1. Both kinds of leaves store their content explicitly, so they can be pointed from other nodes. The resulting *2D-BT* leaves may then be empty nodes, explicit nodes, pointers to sources, or the last level of the tree. For simplicity, the last level of the tree will be that of individual cells, until Section 3.5.

Figure 4 exemplifies a specialized *2D-BT*. The first level is composed of two internal nodes, the top ones. The submatrix starting at $(2, 0)$ is the source of the third block, so the third node points to the first, inside which the source starts, with the appropriate offset. The fourth block of this level is an empty leaf because its submatrix is full of 0s.

The first two nodes of the first level are subdivided, so we have four nodes in the second level. There are several empty leaves (full of 0s) and explicit leaves (with a single 1) in this level. There is also a leaf holding a pointer to the first occurrence of $\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$, and only one internal node. This is split again to form the only four nodes of the third level, which are explicit because it is the last level.

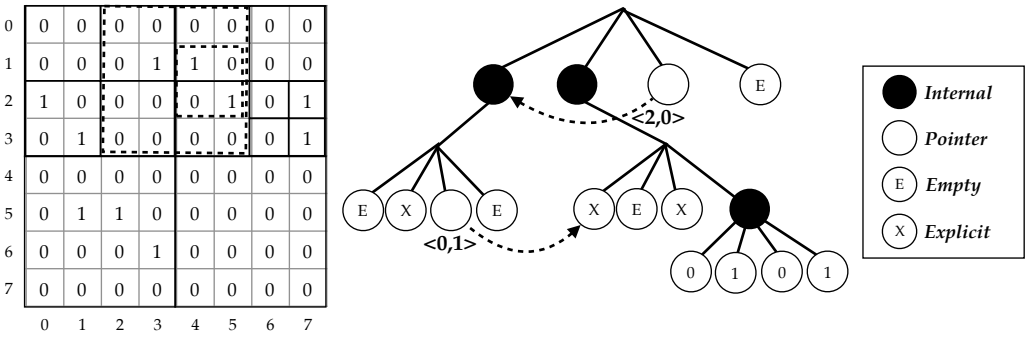

Figure 4: Specialization of a Two-Dimensional Block Tree to Web graphs.

3.3. Concrete representation

In order to represent the specialized *2D-BT* (just *2D-BT* from now on), we use several bitmaps and arrays, which enable efficient navigation of the tree. Figure 5 shows the representation of the *2D-BT* illustrated in Figure 4.

Describing the topology. The first two bitmaps are $T[1..]$ and $L[1..]$. Bitmap T stores a bit for each node in the tree in levelwise order, except for those of the last level. The bit of each node is a 1 if the node is internal and a 0 otherwise. Bitmap L represents the content of the leaves of the last level, left to right. In Figure 5, $T[1], T[2]$ and $T[12]$ contain 1s because they are internal nodes; the other positions contain 0s because they are leaves of various sorts. In L we see the four bits of the submatrix $\binom{0\ 1}{0\ 1}$, the only one reaching the last level.

As in the k^2-tree, every internal node has k^2 children and we can simulate the navigation of the tree in both directions (top-down and bottom-up) by supporting *rank* and *select* operations over T. That is, from an internal node n_i whose bit is located at $T[i]$, we can compute the position of its c-th child ($0 \leq c < k^2$) as $i'+c$, with $i' = rank_1(T, i) \cdot k^2 + 1$ (the children are located consecutively in the range $children(i) = [i' \dots i'+k^2-1]$). Notice that when $i' > |T|$, the k^2 children are in L, after position $i'-|T|$. For example, in Figure 5 the children of the internal node at $T[12]$ would start at $i' = rank_1(T, 12) \times 4 + 1 = 13$, so they are stored from position $13 - 12 = 1$ of L.

Similarly, we can compute the parent position of $T{:}L[i]$ as $parent(i) = select_1(T, \lfloor (i-1)/k^2 \rfloor)$, where $T{:}L$ is the concatenation of bitmaps T and L. Recall that we can support *rank* and *select* operations in $O(1)$ time with an extra space of $o(|T|)$ bits.

Identifying kinds of leaves. In order to identify the kind of leaf nodes, we use two additional bitmaps: one for *non-empty nodes* (N) and another for *explicit nodes* (X). Following the same order of T, for each leaf node $T[i] = 0$, we add a 0 to N if the node is empty; otherwise we add a 1. Therefore, if $T[i] = 0$ and $N[rank_0(T, i)] = 0$, the node is empty. In case $N[rank_0(T, i)] = 1$, the node has a pointer or is explicit. In Figure 5, the first node of the second level corresponds to $T[5] = 0$. Since $N[rank_0(T, 5)] = N[3] = 0$, we know it is empty. Instead, the second node of the second level is non-empty because $N[rank_0(T, 6)] = N[4] = 1$.

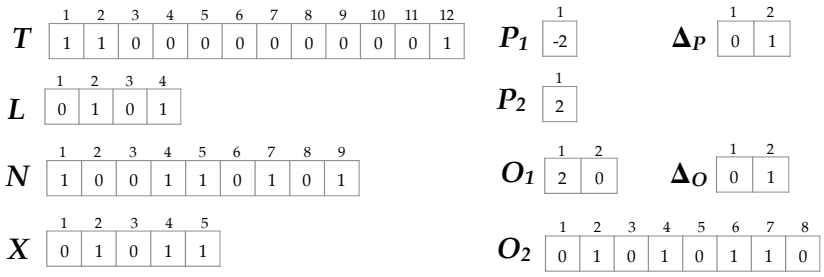

Figure 5: Compact representation of the specialized *2D-BT*.

Bitmap X distinguishes between explicit and pointer nodes. For each position j where $N[j] = 1$, we append a 0 or a 1 to X when the node is a pointer or is explicit, respectively. We then know that $N[j] = 1$ has a pointer if $X[rank_1(N, j)] = 0$; otherwise it is explicit. Coming back to the second node of the second level ($T[6] = 0$ and $N[4] = 1$) in Figure 5, we know that it is explicit because $X[rank_1(N, 4)] = X[2] = 1$. To solve that *rank* operation in $O(1)$ time, we need $o(|N|)$ additional bits.

Storing additional information on leaves. Explicit leaves must store the offset of their only 1, whereas pointer leaves must store the pointed node and the offset of the source inside it. Note that all the offsets are relative coordinates within a submatrix and all the submatrices at the same depth d have the same size, n/k^d. Hence, the offsets from both explicit and pointer nodes at the same depth can be stored together, in an array $O_d[1..]$ with cells of $\lceil \log_k(n/k^d) \rceil = \log_k n - d$ bits. Along a levelwise traversal of the tree, the x- and y- coordinates of explicit and pointer leaves are appended.

In principle, the offset entries in arrays O_d correspond to positions in X. Notice, however, that we have several arrays O_d, separated by depth. We then store a small array Δ_O, which at position d contains the number of nodes with depth less than d having offsets. Then, the coordinates of the node of depth d represented by $X[p]$ are stored at $O_d[e-1, e]$, with $e = 2 \cdot (p - \Delta_O[d])$.

In Figure 5, to obtain the offset from the second node of the second level ($T[6] = 0$, $N[4] = 1$, $X[2] = 1$), we observe that there is one node with offsets preceding the current level ($\Delta_O[2] = 1$). We then compute $e = 2 \cdot (2 - 1) = 2$ and find the offsets in $O_2[1, 2] = \langle 0, 1 \rangle$. As the submatrix starts at position $(2, 0)$, the 1-bit is at position $(2, 1)$. In the case of the third node at depth $d = 2$ ($T[7] = 0$, $N[5] = 1$, $X[3] = 0$), the offsets are stored at $O_2[3, 4] = \langle 0, 1 \rangle$.

Finally, we must represent the pointers. For each depth d we have an array P_d of signed integers storing the distance $i_s - i_t$ between the source and target nodes, at positions i_s and i_t in T, respectively. Unlike the one-dimensional Block Tree, we admit backward and forward pointers, thus $i_s - i_t$ can be negative or positive. The reason is that, at construction time, the order in which we detect possible sources is, for efficiency, different from the order in which the *2D-BT* arranges the children of the nodes.

We then obtain the pointer of the node at position $X[p] = 0$, with depth d, as $P_d[rank_0(X, p) - \Delta_P[d]]$. In Figure 5, to complete the information on the third node at the second depth ($T[7] = 0$, $N[5] = 1$, $X[3] = 0$), we obtain its pointer. This is stored

at position $rank_0(X, 3) - \Delta_p[2] = 2 - 1 = 1$ of P_2. Since $P_2[1] = 2$ and $i_t = 7$, we know that the bit of the source in T is $i_s = 7 + 2 = 9$. That is, the pointer from the 7th node refers to the 9th node of the tree and has offset $\langle 0, 1\rangle$.

3.4. Access to a region

Our representation allows us navigate the tree, determine the kind of each node, and retrieve the information of pointers and offsets, all in constant time. Those primitives support operations like $access(R, 0, 0, 0, 0, 0)$ in Algorithm 1, which given a region $R = [(x_{min}, y_{min}), (x_{max}, y_{max})]$ outputs the values of the region in an initially zeroed matrix *result*. The algorithm traverses the *2D-BT* towards the nodes whose submatrix intersects R, until finding all the 1-bits at the tree leaves. Algorithm 1 uses the following notation:

- $children(i)$ retrieves the range of children of the node at position i in T.
- $parent(i)$ computes the parent position in T from the node at position i in T.
- $offsets(i)$ retrieves the offsets $\langle O_x, O_y\rangle$ for an explicit node at position i in T.
- $pointer(i)$ obtains, for the pointer node at position i in T, the triple $\langle i_s, O_x, O_y\rangle$ with the corresponding pointer i_s and offset $\langle O_x, O_y\rangle$.
- $region(i)$ computes the region Q covered by the node at position i in T.

The first four functions were explained in Section 3.3. For $region(i)$, we note that every time we compute the region of a node, we have previously traversed its ancestors. This enables constant-time computation. Assume node i has depth d (and thus submatrix size $s \times s$, with $s = n/k^d$) and its top-left corner is at position (x, y). The c-th child of i, for $0 \leq c < k^2$, starts at x-offset $x + (c \bmod k) \cdot (s/k)$ and y-offset $y + \lfloor c/k \rfloor \cdot (s/k)$.

The only case where we cannot obtain $region(i)$ from the parent node is for $region(i_s)$ in the first line of Algorithm 2, when a pointer is taken. To avoid this problem, instead of computing $region(i)$, we always use R relative to the current $region(i)$. Thus, when we move from region R relative to i to region R' relative to i_s in the first line of Algorithm 2, all we have to do is $R' \leftarrow R + (O_x, O_y)$. From there, we might have to compute *region* for the parent of node i, knowing $region(i)$. We know that i is the c-th child of its parent, with $c = i - i'$ and $children(parent(i)) = [i'..i' + k^2 - 1]$. If i starts at (x, y) and is of size $s \times s$, its parent is of size $sk \times sk$ and its top-left corner has x-coordinate $x - (c \bmod k) \cdot s$ and y-coordinate $y - \lceil c/k \rceil \cdot s$.

Algorithm 1 shows the procedure to obtain the values within a region R. It receives the region R to extract, the starting node i, its depth d, a previous depth d_{ptr} (to be explained later), and the current offset (x, y) of node i within R. The algorithm performs a top-down tree traversal. In lines 1–2, it checks if the depth corresponds to the last level of the tree, in which case it copies the value of the specific node, stored at $L[i - |T|]$, to *result* at position (x, y).

Lines 3–7 handle internal nodes. We traverse the children i' of i, computing their region and recursively entering those that intersect R. Lines 8–10 handle explicit nodes. We obtain the offset of the only 1 in the region and write it in *results*. Note that, since *results* is zero-initialized, we do nothing for empty leaves.

The most complex situation occurs when the current node contains a pointer (lines 11–14), whose information we obtain in line 11. In principle, the procedure should be

Algorithm 1: access(R, i, d, d_{ptr}, x, y)

```
1  if d = height then
2  |  result[x, y] ← L[i − |T|]
3  else if i is internal then
4  |  for i' in children(i) do
5  |  |  Q ← region(i')
6  |  |  if Q ∩ R ≠ ∅ then
7  |  |  |  access(Q ∩ R, i', d + 1, d_ptr, Q.x_min, Q.y_min)
8  else if d ≥ d_ptr and i is explicit then
9  |  ⟨O_x, O_y⟩ ←offsets(i)
10 |  result[x + O_x, y + O_y] ← 1
11 else if d ≥ d_ptr and i is a pointer then
12 |  ⟨i_s, O_x, O_y⟩ ←pointer(i)
13 |  ⟨R', i', d'⟩ ← back(i_s, O_x, O_y, d)
14 |  access(R', i', d', d, x, y)
```

Algorithm 2: back(i_s,O_x,O_y,d)

```
1  R' ← region(i_s) + (O_x, O_y)
2  while R' ⊈ region(i_s) do
3  |  i_s ← parent(i_s)
4  |  d ← d − 1
5  return ⟨R', i_s, d⟩
```

resumed from the node i_s, considering the offsets $\langle O_x, O_y \rangle$. Indeed, node i_s contains the top-left of the region to extract, but unless $O_x = O_y = 0$, the region contents are spread across up to 4 nodes. For example, in Figure 4, when following the pointer of the second level (with $O_y > 0$), the target contents are distributed across the pointed node (first child of its parent) and the third child of that parent. In order to find those nodes, we move recursively to the ancestor i' at depth d' that completely contains the source. In our example this is the parent node, but it could be a farther ancestor. That search is done with *back* (line 13), shown in Algorithm 2.

The top-down traversal then resumes from the ancestor (line 16). In our example, to obtain the information of the target, the algorithm will resume from the parent of node i_s and return to its first and third children. Note that, although we go up in the *2D-BT* until level d', such recursive call will eventually go back to level d only for the (up to) 4 desired nodes, entering into (up to) 4 children at level d' and then into only one child in levels $d' + 1$ to $d - 1$.

This is the only case that updates d_{ptr}, the depth where the pointer was taken. Recall that, if in a previous level a specific node corresponds with a leaf, the following levels of the tree do not consider the submatrix of that node, that is, we can assume it is empty. Therefore, when we come back to the sources of the pointer from the ancestor, if in that path there is any leaf whose depth d is less than d_{ptr}, the corresponding submatrices are empty at d_{ptr}. For this reason lines 8 and 11 of Algorithm 1 ignore those leaves.

3.5. Compression of leaves

In general, we can stop splitting internal nodes at some size where storing the contents directly is more convenient than using a pointer. The base k^2-tree [9] has a similar mechanism, coupled with statistical compression of those contents. We adopt that mechanism in our specialized *2D-BT* as well.

Let us define $n_L \times n_L$ as the size of the last-level submatrices. To each node of size $n_L \times n_L$ that remains in the *2D-BT*, we assign a binary codeword $cw[1..n_L^2]$ with the submatrix contents. The different codewords are then sorted from most to least frequent. Instead of storing the explicit codewords cw in the *2D-BT* leaves, we store the index of cw in the sorted list. This arrangement makes smaller indexes more frequent, so we store the indexes with an encoding like Directly Addressable Codes (DACs) [11], which obtains good compression of small numbers and supports efficient random access to them. Notice that the compression of leaves also reduces the number of levels to traverse, improving the performance in some queries.

4. Construction

The key task in the construction of a *2D-BT* is to detect the occurrences of a block, so as to determine which nodes are internal and which are leaves. Although the concept is simple, efficiently looking for those repetitions is not an easy task. We use an algorithm based on the technique of Karp and Rabin [16] to compute the *fingerprint* of a string. Given a string S, the Karp-Rabin algorithm computes the fingerprint of any substring, $H_{KR}(S[i \dots j])$, in time $O(j - i + 1)$. However, given $H_{KR}(S[i \dots j])$, the fingerprint of the next substring of the same length, $H_{KR}(S[i + 1 \dots j + 1])$, can be computed in $O(1)$ time. We extend this scheme to two dimensions, in the style of Bird [6] and Baker [3].

4.1. The basic idea

In order to construct a level whose block size is $s \times s$, a naive approach would store the whole matrix M in main memory and detect the kinds of nodes in two stages:

1. *Fingerprints of blocks.* The algorithm computes the fingerprints of the blocks of size $s \times s$ with top-left corners at $(i \cdot s,\ j \cdot s)$, for every $0 \leq i, j < n/s$. Those fingerprints, with their corresponding top-left corner positions, are stored in a hash table HT. If, during the insertion of a block b, there is a collision with an already inserted block b_{HT}, we compare the corresponding submatrices. If they are identical, the node of b becomes a leaf pointing to the node of b_{HT}. At the end of this stage, the blocks b_{HT} receiving pointers are removed from HT, to avoid them becoming in turn pointer leaves in during the next stage.
2. *Fingerprints of submatrices.* The first stage detects only those leaves whose source covers exactly one block, so we still have to find the leaves whose source overlaps several blocks. We then compute the fingerprint of *all* the submatrices S of size $s \times s$ with top-left corner $(i,\ j)$, for every $0 \leq i, j < n - s$, such that the (up to) four blocks covered by S exist in this level and are not already pointers. Every time a fingerprint of a submatrix S is computed, the algorithm checks if there is a block b_{HT} with the same fingerprint in HT. If so, and the corresponding submatrices match, the node of b_{HT} becomes a leaf that points to the top-left node that covers S. The (up to) four blocks covered by S are removed from HT (to avoid replacing

them by pointers later) and the block b_{HT} is also removed (since we have already replaced it by a pointer).

Note that storing the whole binary matrix in main memory requires n^2 bits of space and just computing all the fingerprints takes time $\Theta(n^2 s)$, which becomes $\Theta(n^3)$ when added over all the tree levels. Both space and time are unaffordable when compressing large and very sparse matrices, like Web graphs.

Our actual construction algorithm exploits the sparseness of the matrices, so as to simulate the naive algorithm in space and time proportional to the number m of 1s they have. In particular, it will run over a sparse matrix representation: an increasing list L_r of integers for each row r stores the columns c where there are 1s in row r.

A sparse Karp-Rabin function. To compute the fingerprint of a block $S[0..s-1, 0..s-1]$ in time proportional to the number of 1s in it, we use the following Karp-Rabin-like function:

$$H_{KR}(S) \quad = \quad \left(\sum_{0 \le r,c < s} S[r,c] \cdot b^{cs+r} \right) \mod p,$$

where $2 \le b < p$ and p is a large prime. When the matrix is binary, we can compute H_{KR} by just adding b^{cs+r} for the positions $\langle r, c \rangle$ of the 1s in S. We implement the following functions, which are needed for the construction:

- $add(H_{KR}, \langle r, c \rangle)$ adds a 1 at $\langle r, c \rangle$: $H_{KR} \leftarrow (H_{KR} + b^{cs+r}) \bmod p$.
- $remove(H_{KR}, \langle r, c \rangle)$ removes the 1 at $\langle r, c \rangle$: $H_{KR} \leftarrow (H_{KR} - b^{cs+r}) \bmod p$.
- $shift(H_{KR})$ shifts all the points in the block one cell to the left, assuming the first column is empty: $H_{KR} \leftarrow (b^{-s} \cdot H_{KR}) \bmod p$, where b^{-s} is the multiplicative inverse of b^s modulo p.

4.2. Computing fingerprints of blocks

The first stage starts by building a min-heap I_{KR} with the values from the lists $L_0, \ldots, L_{s-1}$. I_{KR} stores the pairs $\langle r, c \rangle$ such that $c \in L_r$, sorted by c. Thus, the pairs with the minimum value of c are on top of I_{KR}. To compute the fingerprint of the first block, $[(0,0), (s-1, s-1)]$, we extract successive pairs from I_{KR} until we obtain a pair $\langle r', c' \rangle$ that is out of the block (i.e., $c' \ge s$). The block fingerprint is then computed from all the extracted 1s.

The least extracted value c' out of the block indicates which is the next nonempty block to the right (i.e., $\lfloor c'/s \rfloor$). We then restart the process from the $\lfloor c'/s \rfloor$-th block, and continue until processing the first row. All the rows of blocks are processed similarly.

Note that we do not need to insert the whole rows L_r in I_{KR}: we can maintain only the next value c from each L_r, and insert the next one as it is consumed. The heap then requires $O(s)$ space, and each of the m 1s is processed in $O(\log s)$ time.

Algorithm 3 gives the pseudocode. It returns the fingerprints of all the nonempty blocks, ignoring the others. It also returns the number of points in the blocks and one of those points, so that we can later identify and process the explicit leaves. Its overall time, for a given block size s, is $O(m \log s)$ because every 1 in the matrix is inserted and removed exactly once.

Algorithm 3: KRBlocks$([L_0, \ldots, L_{n-1}],\ n,\ s)$

```
1  b ← 0;
2  while b < n do
3  |  e ← min(b + s − 1, n − 1);
4  |  I_KR ← ∅; H_KR ← 0;
5  |  cc ← 0;
6  |  for r ∈ b, ..., e do
7  |  |  I_KR.insert(⟨r, L_r.first()⟩); // first() gives n on empty lists
8  |  while cc < n do
9  |  |  ⟨r, c⟩ ← I_KR.min();
10 |  |  p ← 0; ⟨ro, co⟩ ← ⟨r, c⟩
11 |  |  while c < min(cc + s, n) do
12 |  |  |  add(H_KR, ⟨r − b, c − cc⟩);
13 |  |  |  p ← p + 1;
14 |  |  |  I_KR.deleteMin();
15 |  |  |  I_KR.insert(⟨r, L_r.next()⟩); // next() gives n after its last element
16 |  |  |  ⟨r, c⟩ ← I_KR.min();
17 |  |  if p > 0 then output ⟨H_KR, ro, co, p⟩;
18 |  |  cc ← s · ⌊c/s⌋;
19 |  b ← b + s;
```

4.3. Computing fingerprints of submatrices

The second stage is more complex because we need to maintain the fingerprint of a sliding window that we move across the matrix. To update the fingerprint, we need to know not only which 1s enter the window as we slide it, but also which 1s disappear from it. We then use an additional min-heap, O_{KR}, that contains the 1s that are currently inside the window, using the same order of I_{KR}. Since there are at most s^2 elements in O_{KR}, its operations also take time $O(\log s^2) = O(\log s)$.

Algorithm 4 gives the pseudocode. It explores every matrix position, not only the block-aligned ones as in the previous section. In general, there are points in the current window, so we must slide it by one position to the right. This is done by (1) removing the points in the current leftmost column (lines 9–13), (2) shifting the window fingerprint (lines 14–16), and (3) adding the points in the new rightmost column (lines 17–22). A special case occurs when the window becomes empty of points, so we must slide it until the next point from I_{KR} appears on the right column. This is also handled in lines 14–16.

The algorithm acts as an iterator that outputs all the fingerprints of the nonempty submatrices, together with their position. Its complexity is dominated by the operations on the heaps. For every band of the matrix, every point in the band enters and exits both heaps exactly once. However, every point belongs to s bands, which yields a total time complexity of $O(ms \log s)$.

4.4. Determining the kinds of nodes

In order to build a level of the *2D-BT*, we first initialize all the nodes as *empty*, so that we can ignore the empty blocks that Algorithm 3 skips. We initialize the hash table

Algorithm 4: KRSubmatrices$([L_0, \ldots, L_{n-1}],\ n,\ s)$

```
1  b ← 0;
2  while b ≤ n − s do
3  |  e ← min(b + s − 1, n − 1);
4  |  I_KR ← ∅; O_KR ← ∅; H_KR ← 0;
5  |  cc ← −s;
6  |  for r ∈ b, ..., e do
7  |  |  I_KR.insert(⟨r, L_r.first()⟩); // first() gives n on empty lists
8  |  while cc < n do
9  |  |  ⟨ro, co⟩ ← O_KR.min(); // min() gives n on empty heaps
10 |  |  while co = cc do
11 |  |  |  O_KR.deleteMin();
12 |  |  |  remove(H_KR, ⟨ro − b, 0⟩);
13 |  |  |  ⟨ro, co⟩ ← O_KR.min();
14 |  |  ⟨r, c⟩ ← I_KR.min();
15 |  |  if O_KR ≠ ∅ then shift(H_KR) ;
16 |  |  else cc ← c − s ;
17 |  |  while c = cc + s do
18 |  |  |  I_KR.deleteMin();
19 |  |  |  I_KR.insert(⟨r, L_r.next()⟩) // next() gives n after its last element;
20 |  |  |  O_KR.insert(⟨r, c⟩);
21 |  |  |  add(H_KR, ⟨r − b, s − 1⟩);
22 |  |  |  ⟨r, c⟩ ← I_KR.min();
23 |  |  cc ← cc + 1;
24 |  |  output ⟨H_KR, b, cc⟩;
25 |  b ← b + 1;
```

HT, whose keys are the fingerprints of the blocks. Then, for every tuple $\langle H_{KR}, r, c, p\rangle$ reported by Algorithm 3, we first check if $p = 1$, in which case we mark the block as *explicit* and record its only point, (c, r). Otherwise, we insert the block fingerprint H_{KR} in HT, with block row $\lfloor r/s \rfloor$ and column $\lfloor c/s \rfloor$, and mark it as *internal* for now. To find the tree node corresponding to a block row and column, we descend by the partial *2D-BT* we have already built for the previous levels (these searches add, at worst, a negligible $O(m \log_k n)$ time per level).

If there is a collision between the fingerprint H_{KR} we are inserting and that of a block b_{HT} already in HT, we check if both blocks are identical. If they are not, this is treated as a classic hash collision. If the blocks match, however, as explained in Section 4.1, then the new block becomes a *pointer* leaf, pointing to b_{HT} with offset zero.

After completing the first step, we remove from HT all the blocks b_{HT} that receive pointers, as explained in Section 4.1. We now process the tuples $\langle H_{KR}, r, c\rangle$ delivered by Algorithm 4, searching HT for each window fingerprint H_{KR}. Let S be a submatrix whose fingerprint H_{KR} matches that of a block b_{HT} in HT. If their content also match, we convert b_{HT} into a *pointer* leaf to the block that covers the point (c, r) (the block is also found using the previous *2D-BT* levels). Finally, as explained in Section 4.1, we

remove from HT the (up to) four blocks covered by S, as well as b_{HT}.

Accessing submatrix contents. We do not have direct access to the matrix values to easily check whether a submatrix S matches a block b_{HT}. Right after Algorithms 3 and 4 output a tuple, the cursors in the corresponding lists $L_b, \ldots, L_e$ are next to the rightmost values in each row. We can then traverse backwards from those pointers in order to obtain the points in S. For b_{HT}, however, we only know the range of lists it spans. We then binary search the lists (which are in practice arrays) to find the corresponding cursor positions. Each fingerprint collision then takes at most $O(s \log m + \min(m, s^2))$ time.

4.5. Preparing for the next level

A delicate issue is that, when scanning the windows of a level, we must avoid the areas that have become *2D-BT* leaves in previous levels. We enforce that by removing the 1s of the matrix covered by every block that becomes a leaf. Since our scanning in the next level skips over the empty areas, it will automatically ignore higher-level leaves.

Whenever we convert a block to a pointer leaf, we binary search its lists L_r as done for matching submatrices, and mark the 1s that are covered by the block. After the level is processed, we pass over all the lists and remove all the marked 1s, in $O(m)$ additional time per level. Note that in some cases we have just done the binary searches to compare the block contents.

Note that the submatrices S we find after removing those 1s may still overlap the areas of higher-level leaves. However, at least one of the blocks covered by S must exist in the current level, otherwise S would be empty. We expand our pointer mechanism so that a pointer leaf can point to any of the four corners of S (i.e., if an offset is $\geq s$, then it refers to the final, not the initial, submatrix position). We can then always point to a node in the same level. The mechanism we described in Section 3.4 for finding the tree ancestor that covers the submatrix works without modifications. This mechanism actually gives more opportunities for matching submatrices than the basic mechanism.

4.6. Time and space complexity

As seen, the time per level with submatrices of $s \times s$ is $O(ms \log s + pm(s \log m + \min(m, s^2)))$, where p is the probability of two fingerprints matching (either because of equality of submatrices of because of a collision). Adding over all the $\log_k n$ levels, this yields $O(mn \log n + pm(n \log m + m \log_k(n^2/m))) = O(mn \log n + pm^2 \log_{k^2}(n^2/m))$. In practice p is sufficiently low to let the first term dominate. The naive algorithm, under the same assumptions, would require time $O(n^3 + pn^4)$. The construction cost is then significantly smaller than that of the naive algorithm already on mildly sparse matrices, $m = o(n^2/\log n)$.

5. Experimental Evaluation

The Two-Dimensional Block Trees (*2D-BT*) were coded in C++, using several structures from the SDSL library [12]. The bitmaps of the *2D-BT* were implemented as plain bitmaps with the `bit_vector` class. To support *select*, we use `select_support_mcl`, which takes constant time by using 20% of extra space. The *rank* structures were implemented with `rank_support_v5`, which requires 6.25% of additional space and takes

name	nodes	edges
arabic-2005	27,744,080	639,999,458
eu-2005	862,664	19,235,140
indochina-2004	7,414,866	194,109,311
uk-2002	18,520,486	298,113,762

Table 1: Datasets from the WebGraph framework.

constant time. Arrays P_d and O_d are `int_vectors` where each entry uses the number of bits required by the largest integer.

The *2D-BT* can use two settings, with or without compression of leaves. The configuration with compression of leaves is called `2D-BT-CL`, whereas the original one is named `2D-BT-O`. In `2D-BT-CL`, we use the DAC implementation where the bits of the entries of each level are chosen optimally via dynamic programming [11].

The construction time we obtained, essentially $O(mn \log n)$, is still too large for massive matrices. To speed up the construction, we defined a maximum block size β from where the construction starts. Higher levels are implemented as basic k^2-trees, that is, allowing only internal and empty nodes. The construction time is then $O(m\beta \log \beta)$.

We measure and compare the space and the time for accessing the matrix of the *2D-BT* in two scenarios:

1. *Web graphs.* The *2D-BT* is used to compress Web graphs, and compared with state-of-the-art techniques.
2. *Raster data.* We analyze the effect of using *2D-BT*s instead of k^2-trees in the k^2-acc representation [10] for raster data, comparing it also with the k^2-raster [17].

The experiments were conducted on an Intel® Core™ i7-3820 CPU @ 3.60GHz (4 cores) with 10MB of cache and 256 GB of RAM, running Debian GNU/Linux 9 with kernel 4.9.0-8 (64 bits), gcc version 6.3.0 with `-O9` optimization.

5.1. Web graphs

We used different real datasets from *WebGraph framework* [8], available from `http://webgraph.dsi.unimi.it`. Their specifications are given in Table 1. The nodes of the datasets are sorted according to the lexicographical order of their URL.

For each dataset we build structures `2D-BT-O` and `2D-BT-CL` with $k = 2$. In `2D-BT-CL`, we set $n_L = 4$. We use $\beta = 1{,}024$ for all datasets except in *enwiki-2018*, *hollywood-2009*, *ljournal-2008* and *twitter-2010*, where we used $\beta = 256$ for the latter and $\beta = 512$ for the others in order to build the *2D-BT*s in reasonable time. We compare the *2D-BT* with all the techniques presented in Section 2.1. We use the implementation of the `WebGraph` algorithm [8] available at the *WebGraph framework* site, configured with six different values of $\langle w, m \rangle$: $\langle 3, 3 \rangle$, $\langle 3, 50 \rangle$, $\langle 30, 50 \rangle$, $\langle 30, 300 \rangle$, $\langle 70, 300 \rangle$, and $\langle 70, 1000 \rangle$. We use the implementation of algorithm `AD` available at `https://github.com/drovandi/GraphCompressionByBFS`, configured with different values of ℓ: 16, 64, 100, 200, 500, and 1000. For the algorithm `LM` [13] we use the implementation of the original authors and the chunk size set to 8, 16, 64, 128, 256, and 512. For `Zuckerli` [23] we use the implementation of `https://github.com/google/zuckerli` with its default settings. We use the code available at `https://lbd.udc.es/research/k2tree/` for the `k2-tree`. To obtain a fair comparison between the *2D-BT* and the `k2-tree`, we set $k = 2$ and $n_L = 4$.

Although the Web graphs can be used for different proposes, we focus here on the operations the *2D-BT* is designed for: access to individual cells (i.e., determining if there is a link between two given pages) and extraction of submatrices (e.g., extracting all the links between two domains) of sizes 100×100 and $10{,}000 \times 10{,}000$. To measure time, we average user times over 100,000 queries at random positions.

Figure 6 compares the space and query time on the different Web graphs. In most cases, `2D-BT-CL` is slightly smaller and slower than `2D-BT-O`, though in some cases it is both smaller and faster. On Web graphs, this variant uses 60%–80% of the space of `k`^{2}`-tree`, and it is from 30% faster to 3 times slower than it for accessing individual cells. When extracting larger regions, `2D-BT-CL` takes from about the same time to an order of magnitude more time than `k`^{2}`-tree`. This is expected, since the *2D-BT* must jump from targets to sources when extracting cells, and this is more frequent when extracting larger areas. In general, the *2D-BT* offers an interesting space reduction from the k^2-tree, at a generally moderate price in time.

When comparing with other techniques, `2D-BT-CL` is outperformed in space only by `LM`, and only slightly and occasionally by `AD`. In its smallest configuration, `LM` uses 70%–90% of the space of `2D-BT-CL`, but it is 2–4 times slower for accessing individual cells. `LM` is then an interesting space/time tradeoff as well. Since `LM` is paremeterizable, it can become faster by using more space. When it reaches the same space of `2D-BT-CL`, however, it is still 30% to 4 times slower than it for accessing individual cells. `LM` ceases to be a relevant alternative when accessing regions, however: the difference in time with `2D-BT-CL` raises to 3–7 orders of magnitude. `AD` is worse than `LM` for accessing individual cells. For regions, `AD` can use as little as 80% of the space of `2D-BT-CL`, but it is 2–3 orders of magnitude slower.

In conclusion, the *2D-BT* offers a relevant space/time tradeoff with respect to k^2-trees when representing Web graphs. Some of the classic compressed graph representations offer another interesting space/time tradeoff for accessing individual cells, but they fail when extracting a larger submatrix.

5.1.1. Construction

The left part of Figure 7 shows the memory consumption of the *2D-BT* during its construction over the dataset *eu-2005* with $\beta = 1{,}024$. The vertical dashed lines delimit the different stages of the construction. The first stage is the construction of the levels with block size over β. This point reaches the peak of memory because we need to keep an array with the z-order values of the adjacency lists to obtain a faster construction. The space of this array is equal to that of the adjacency lists (77 MB), and our peak of memory is 437 MB. After building those levels, that space is freed.

The following stages need to compute the fingerprint of each block and submatrix. The label on the vertical lines specifies the block size of the processed level. We can observe that building the level, of block size 1,024, takes around 4,000 seconds and its memory usage is 287 MB. That space stays constant during the construction until the block size is 128. From that point on, every time we descend one level, the number of nodes (blocks) to check increases considerably and affects the memory usage.

The right part of Figure 7 shows that the *2D-BT* is the third structure with the smallest peak of memory, which makes it competitive with previous work in this aspect.

Instead, the *2D-BT* is considerably slow to build. Its construction on *eu-2005* is two orders of magnitude slower than the baselines, 7,200 versus 67 seconds. The most

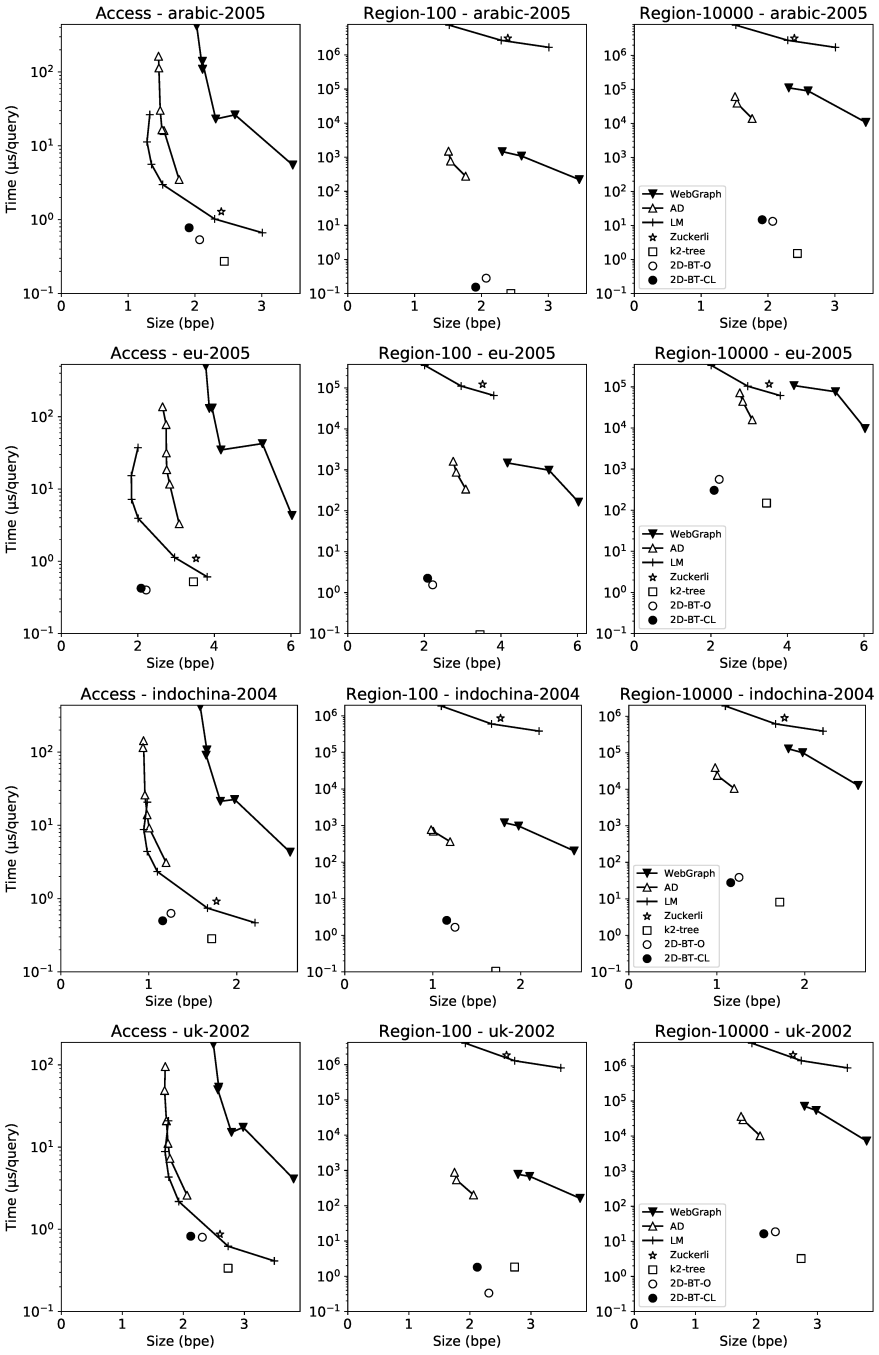

Figure 6: Time performance and space in four Web graphs.

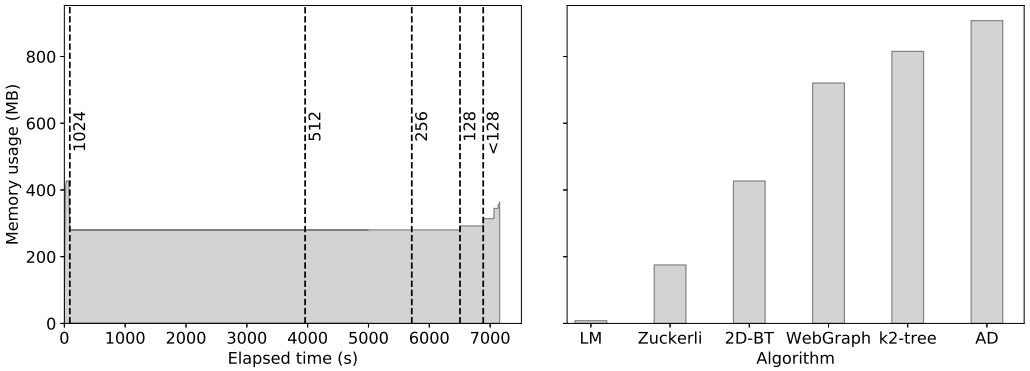

Figure 7: Memory usage of *2D-BT* (left) and comparison with the other techniques (right).

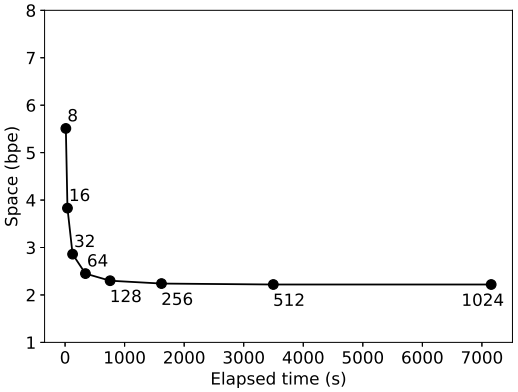

Figure 8: Space of *2D-BT* over *eu-2005* with different values of β.

consuming part during the construction of a *2D-BT* is the computation of the fingerprints of the submatrices. Those costs are greater when the block size is larger, so we can reduce the value of β to speed up construction at the price of worse compression. Figure 8 shows that the resulting space/time tradeoff is very good, however: we already obtain almost the final space with a maximum block size of $\beta = 128$, where construction time is around 800 seconds, an order of magnitude faster than for $\beta = 1{,}024$.

5.1.2. A simple compression baseline

As a sanity check, we considered the basic alternative of using a standard lossless image compressor on the binary matrices. In order to provide reasonable extraction times, we divide the image into squares of $\beta \times \beta$ and compress each square separately. When it comes to extract a region, we only decompress the squares that intersect it.

We used a PNG compression API called LodePNG[3]. For the empty squares we only spend an integer indicating so; we do not use the compressor. We tried different powers

[3] `https://lodev.org/lodepng`, seeking to maximize compression with the flags `info_png.color.colortype` = `LCT_GREY`, `info_png.color.bitdepth` = 1, `encoder.filter_palette_zero` = 0, `encoder.add_id` = `false`, `encoder.text_compression` = 1, `encoder.zlibsettings.nicematch` = 258,

	σ	empty	2DBT-acc	k^2-raster	k^2-acc
eua00	322	45.56%	1.96	**1.95**	4.70
eua01	396	83.39%	1.77	**1.17**	5.31
eua02	255	77.90%	2.88	**1.79**	4.90
eua03	244	70.71%	**1.59**	1.61	3.30
eua04	159	59.75%	**1.39**	1.68	2.18
eua10	214	97.00%	0.26	**0.24**	1.78
eua11	245	82.69%	2.40	**1.37**	3.46
eua12	304	37.09%	6.50	**4.13**	8.45
eua13	399	9.68%	7.80	**5.12**	7.56
eua14	498	0.00%	14.09	**6.04**	12.96

Table 2: Datasets of raster data and space (MB) after applying each technique.

of 2 for β on *eu-2005*. The minimum space is reached with $\beta = 1024$, but this is still very high: 11.63 bpe, nearly 6 times over the 2.08 bpe obtained with our *2D-BT*. The extraction times are also much higher: 800 times slower than *2D-BT* for random regions of size 100×100, and 130 times slower for size 10000×10000. Experiments with other Web graphs yield even larger gaps (starting from 20 times larger encodings).

5.2. Raster data

Section 2 describes the structure k^2*-acc* [10], which builds σ k^2-trees to represent a matrix with values in $[1..\sigma]$. Those k^2-trees are modified to compress submatrices full of 1-bits with only one node. This structure requires more space than the k^2*-raster* [17], but the k^2*-acc* outperforms the k^2*-raster* in large window queries. We now consider replacing each k^2-tree of k^2*-acc* with a *2D-BT*, in order to improve the compression while maintaining the window query functionality.

For this purpose, we modify the *2D-BT* so as to support the new k^2-tree functionality of compressing areas full of 1s. We change the meaning of *explicit* nodes so that, instead of representing areas with a single 1, they represent submatrices full of 1s. As a consequence, *explicit* nodes now store no coordinate information. We call `2DBT-acc` the structure that replaces k^2-trees by these *2D-BT* variants in the k^2*-acc*.

We test the resulting structure on the WorldClim datasets [15], which provide a set of layers with global climate information. We chose 10 datasets of 50 MB each, containing mean temperature values measured in Celsius degrees with one decimal. Those values are represented using integers by multiplying them by 10. On those data, we build the `2DBT-acc` with $\beta = 256$ and compare it with the k^2*-raster* and the k^2*-acc*.

The first two columns of Table 2 show the highest value of the dataset and the percentage of cells without values. The last columns show the space required by `2DBT-acc`, `k`^{2}`-raster`, and `k`^{2}`-acc` over each dataset. We observe that `2DBT-acc` is 0.8–2.3 times smaller than `k`^{2}`-acc`, the original structure it modifies. Still, `k`^{2}`-raster` is almost always the smallest structure, using 40%–120% of the space of `2DBT-acc`.

`encoder.zlibsettings.lazymatching` = 1, `encoder.zlibsettings.windowsize` = 32768, `encoder.filter_strategy` = `LFS_ZERO`, and `encoder.zlibsettings.minmatch` = 6.

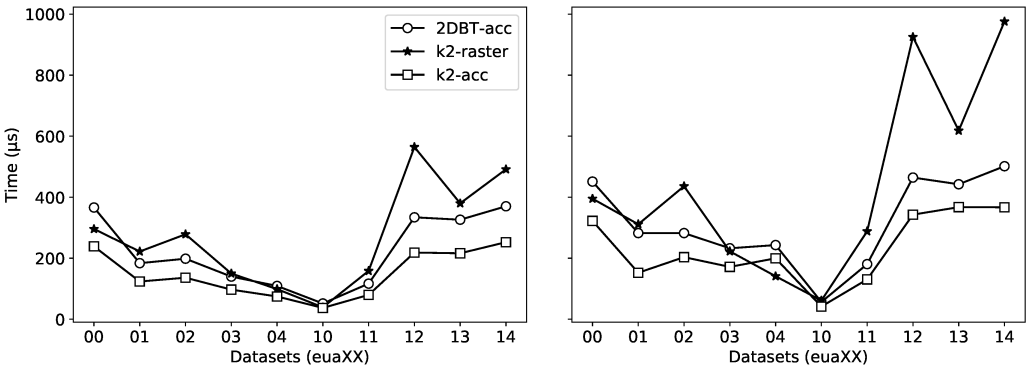

Figure 9: Response time in window queries whose range of values have lengths 8 (left) and 80 (right).

In exchange for the improvement in space, the time performance of `2DBT-acc` for window queries worsens with respect to `k2-acc`, but it is still faster than `k2-raster`. Figure 9 shows the performance on all the datasets averaging 100,000 queries of size 500×500 and two different spans between l and u: 8 and 80. We can observe that `k2-acc` is the fastest structure in almost all cases; `2DBT-acc` is 1.4–1.5 times slower than `k2-acc` for the small range of values and 1.2–1.8 for the large one. In almost all cases, `k2-raster` is slower than `2DBT-acc`, 1.2 times slower on average. Overall, `2DBT-acc` offers a relevant space/time tradeoff between `k2-acc` and `k2-raster`.

5.2.1. The compression baseline

We evaluated again the compression baseline, this time using 256 gray levels for the PNG format. When the range of values exceeded 255, we mapped it to the allowed interval and discretized to integers, which slightly helps the PNG compression compared to the *2D-BT* lossless compression. We considered blocks of side $\beta = 32$ to 1024. This time, the PNG format obtains better compression than the *2D-BT*, by a factor ranging from 1.08 to 4.58 if we use the best β value for each collection. Yet, it is still considerably slower than *2D-BT* to extract 500×500 submatrices, by a factor from 2.26 to 7.61 if we use the best value of β per collection.

6. Conclusions

We have proposed a new structure called Two Dimensional Block-Tree (*2D-BT*), which extends the Block Tree [4] to two dimensions. We first describe the structure in general, which captures repetitiveness in two-dimensional matrices, and then combine it with variants of the k^2-tree [9] to fit different application scenarios like adjacency matrices of Web graphs and two-dimensional raster data. This adaptation lets the *2D-BT* exploit not only repetitiveness, but also clustering and sparseness of binary matrices. Our experiments on various real-life datasets show that the *2D-BT* offers relevant space/time tradeoffs compared to the same structures built on k^2-trees, obtaining considerable space reductions at the price of a reasonable increase in the time for accessing matrix cells or extracting complete submatrices. This tradeoff is especially attractive when the reduced space allows fitting the whole matrix in a higher level of the memory hierarchy (e.g., RAM

vs disk). We believe that the *2D-BT* can also be useful in other application scenarios where the data features some spatial repetitiveness.

The most serious challenge posed by the *2D-BT* is its construction. Although we optimized it considerably for sparse matrices, from essentially $O(n^3)$ to $O(mn \log n)$ average time complexity on an $n \times n$ matrix with m 1s, the construction is still in practice an order of magnitude higher than the original k^2-tree. An interesting line of future work is to further reduce this construction time.

Acknowledgements

For the A Coruña team: This work was supported by CITIC, as Research Center accredited by Galician University System, is funded by "Consellería de Cultura, Educación e Universidade from Xunta de Galicia", supported in an 80% through ERDF Funds, ERDF Operational Programme Galicia 2014-2020, and the remaining 20% by "Secretaría Xeral de Universidades" (Grant ED431G 2019/01), Xunta de Galicia/FEDER-UE under Grants [ED431C 2021/53; IG240.2020.1.185]; Ministerio de Ciencia e Innovación under Grants [PID2020-114635RB-I00; PDC2021-120917-C21]. Gonzalo Navarro was funded by ANID – Millennium Science Initiative Program – Code ICN17_002 and by Fondecyt grant 1-200038. Travis Gagie was funded by Fondecyt grant 1171058 and NSERC Discovery Grant RGPIN-07185-2020.

References

[1] Ageenko, E. and Fränti, P., 2000. Lossless compression of large binary images in digital spatial libraries. *Computers & Graphics*, 24 (1), 91–98.

[2] Apostolico, A. and Drovandi, G., 2009. Graph compression by bfs. *Algorithms*, 2 (3), 1031–1044.

[3] Baker, T.P., 1978. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7 (4), 533–541.

[4] Belazzougui, D., *et al.*, 2021. Block trees. *Journal of Computer and System Sciences*, 117, 1–22.

[5] Bille, P., Gørtz, I.L., and Vind, S., 2015. Compressed data structures for range searching. *In*: *Proc. 42nd International Conference on Language and Automata Theory and Applications (ICALP)*. Springer, 577–586.

[6] Bird, R.S., 1977. Two dimensional pattern matching. *Information Processing Letters*, 6 (5), 168–170.

[7] Boldi, P., *et al.*, 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. *In*: *Proc. 20th International Conference on World Wide Web (WWW)*. 587–596.

[8] Boldi, P. and Vigna, S., 2004. The WebGraph framework I: Compression techniques. *In*: *Proc. 13th International Conference on World Wide Web (WWW)*. 595–602.

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

[10] Brisaboa, N.R., *et al.*, 2020. Extending general compact querieable representations to gis applications. *Information Sciences*, 506, 196–216.

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

[12] Gog, S., *et al.*, 2014. From theory to practice: Plug and play with succinct data structures. *In*: *Proc. 13th International Symposium on Experimental Algorithms (SEA)*. 326–337.

[13] Grabowski, S. and Bieniecki, W., 2011. Merging adjacency lists for efficient Web graph compression. *In*: *Man-machine interactions 2*. Springer, 385–392.

[14] Hernández, C. and Navarro, G., 2014. Compressed representations for Web and social graphs. *Knowledge and Information Systems*, 40 (2), 279–313.

[15] Hijmans, R.J., *et al.*, 2005. Very high resolution interpolated climate surfaces for global land areas. *International Journal of Climatology: A Journal of the Royal Meteorological Society*, 25 (15), 1965–1978.
[16] Karp, R.M. and Rabin, M.O., 1987. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31 (2), 249–260.
[17] Ladra, S., Paramá, J.R., and Silva-Coira, F., 2016. Compact and queryable representation of raster datasets. *In*: *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*. 1–12.
[18] Lempel, A. and Ziv, J., 1976. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22 (1), 75–81.
[19] Lempel, A. and Ziv, J., 1986. Compression of two-dimensional data. *IEEE Transactions on Information Theory*, 32 (1), 2–8.
[20] Munro, J.I., 1996. Tables. *In*: *Proc. 16th Conference Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*. 37–42.
[21] Navarro, G., 2016. *Compact data structures – a practical approach*. Cambridge University Press.
[22] Pajarola, R. and Widmayer, P., 1996. Spatial indexing into compressed raster images: how to answer range queries without decompression. *In*: *Proc. International Workshop on Multimedia Database Management Systems*. 94–100.
[23] Versari, L., *et al.*, 2020. Zuckerli: A new compressed representation for graphs. *IEEE Access*, 8, 219233–219243.