remote sensing

Article

Analysis of Variable-Length Codes for Integer Encoding in Hyperspectral Data Compression with the k^2-Raster Compact Data Structure

Kevin Chow *, Dion Eustathios Olivier Tzamarias, Miguel Hernández-Cabronero, Ian Blanes and Joan Serra-Sagristà

Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, 08193 Cerdanyola del Vallès, Barcelona, Spain; dion.tzamarias@uab.cat (D.E.O.T.); miguel.hernandez@uab.cat (M.H.-C.); ian.blanes@uab.cat (I.B.); joan.serra@uab.cat (J.S.-S.)

* Correspondence: kevin.chow@uab.cat

Received: 19 May 2020; Accepted: 18 June 2020; Published: 20 June 2020

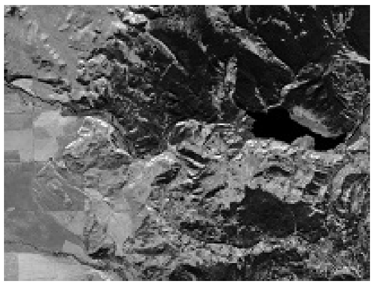
Abstract: This paper examines the various variable-length encoders that provide integer encoding to hyperspectral scene data within a k^2-raster compact data structure. This compact data structure leads to a compression ratio similar to that produced by some of the classical compression techniques. This compact data structure also provides direct access for query to its data elements without requiring any decompression. The selection of the integer encoder is critical for obtaining a competitive performance considering both the compression ratio and access time. In this research, we show experimental results of different integer encoders such as Rice, Simple9, Simple16, PForDelta codes, and DACs. Further, a method to determine an appropriate k value for building a k^2-raster compact data structure with competitive performance is discussed.

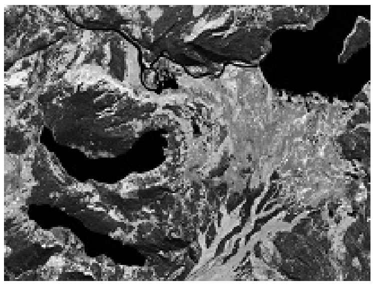
Keywords: compact data structure; k^2-raster; DACs; Elias codes; Simple9; Simple16; PForDelta; Rice codes; hyperspectral scenes

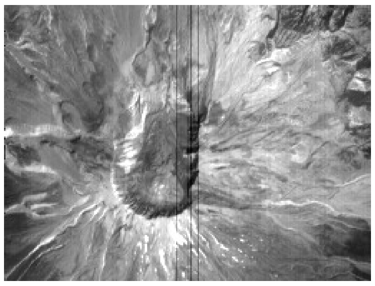
1. Introduction

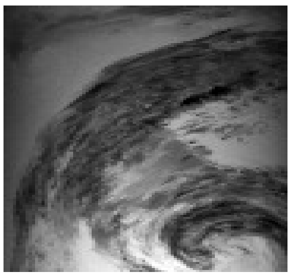
Hyperspectral scenes [1–10] are data taken from the air by sensors such as AVIRIS (Airborne Visible/Infrared Imaging Spectrometer) or by satellite instruments such as Hyperion and IASI (Infrared Atmospheric Sounding Interferometer). These scenes are made up of multiple bands from across the electromagnetic spectrum, and data extracted from certain bands are helpful in finding objects such as oil fields [11] or minerals [12]. Other applications include weather prediction [13] and wildfire soil studies [14], to name a few. Due to their sizes, hyperspectral scenes are usually compressed to facilitate their transmission and reduce storage size.

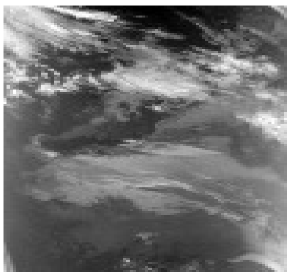
Compact data structures [15] are a type of data structure where data are stored efficiently while at the same time providing real-time processing and compression of the data. They can be loaded into main memory and accessed directly by means of the rank and select functions [16] in the structures. Compressed data provide reduced space usage and query time, i.e., they allow more efficient transmission through limited communication channels, as well as faster data access. There is no need to decompress a large portion of the structure to access and query individual data as is the case with data compressed by classical compression algorithms such as gzip or bzip2 and by specialized algorithms such as CCSDS123.0-B-1 [17] or KLT+JPEG 2000 [18,19]. In this paper, we are interested in lossless compression of hyperspectral scenes through compact data structures. Therefore, reconstructed scenes should be identical to the originals before compression. Any deterministic analysis process will necessarily yield the same results. Figure 1 shows several images from our datasets.

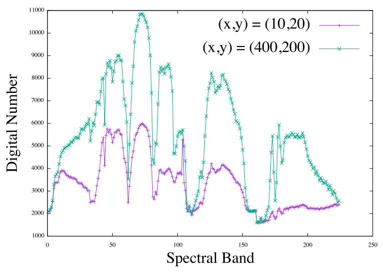

(**a**) Yellowstone 00 (uncal.) (Band 128)


(**b**) Yellowstone 03 (cal.) (Band 128)


(**c**) Hyperion Mt. St. Helens (Band 21)


(**d**) AIRS Gran 6 (Band 1256)


(**e**) AIRS Gran 120 (Band 256)



(**f**) Yellowstone 00 (uncal.)

Figure 1. Several hyperspectral scenes used in this paper. The original and the decompressed scenes discussed in this paper are numerically identical. Depicted also are two spectral signatures for AVIRIS Yellowstone 00 (uncal.). The AVIRIS images in this figure are courtesy of NASA/JPL-Caltech.

The compact data structure used in this paper is called k^2-raster. It is a tree structure developed from another compact data structure called k^2-tree. k^2-raster is built from a raster matrix with its pixel cells filled with integer values, while k^2-tree is from a bitmap matrix with zero and one values. During the construction of the k^2-raster tree, if the neighboring pixels have equal values such as clusters (spatial correlation), the number of nodes in the tree that need to be saved is reduced. If the values are similar, as discussed later in this paper, the values will be made even smaller. They are then compressed or packed in a more compact form by the integer encoders, and with these small integers, the compression results are even better. Moreover, when it comes to querying cells, a tree structure speeds up the search, saving access time. Another added advantage of some of the integer encoders is that they provide direct random access to the cells without any need for full decompression.

Currently, huge amounts of remote sensing data have been produced, transmitted, and archived, and we can foresee that in the future, the amount of larger datasets is expected to keep growing at a fast rate. The need for their compression is becoming more pressing and critical. In view of this trend, we take on the task of remote sensing compression and make it as one of our main objectives. In this research work, we reduce hyperspectral data sizes by using compact data structures to produce lossless compression. Early on, we began by examining the possibility of taking advantage of the spatial correlation and spectral correlation in the data. In our previous paper [20], we presented a predictive method and a differential method that made use of these correlations in hyperspectral data with favorable results. However, in this paper, we would like to focus on selecting a suitable integer encoder that is employed in the k^2-raster compact data structure, as that is also a major factor in providing competitive compression ratios.

Compression of integer data in the most effective and efficient way, in relation to compact data structures, has been the focus of many studies over the past several decades. Some include Elias [21–23], Rice [24–26], PForDelta [27–29], and Directly Addressable Codes (DACs) [30–32]. In our case, we need to store non-negative, typically small integers in the k^2-raster structure. This structure is a tree built in such a way that the nodes are not connected by pointers, but can still be reached with the use of a compact data structure linear rank function. When the data are saved, no pointers need to be stored, thus keeping the size of the structure small. Additionally, we use a fixed code ([15], §2.7) to help us save even more space. In what follows, we investigate the effectiveness of some of these integer encoders.

The rest of the paper is organized as follows: In Section 2, we describe the k^2-raster structure, followed by the various variable-length integer encoders such as Elias, Rice, PForDelta, and DACs. Section 3 presents experimental results for finding the best and optimal values for k and exploring the different integer encoders for k^2-raster. This is done in comparison with classical compression techniques. Lastly, some conclusions and final thoughts on future work are put forth in Section 4.

2. Materials and Methods

In this section, we describe k^2-raster [33] and the integer encoders Elias, Rice, Simple9, Simple16, PForDelta codes, and DACs. This is followed by a discussion on how to obtain the best value of k and two related works on raster compression: heuristic k^2-raster [33] and 3D-2D mapping [34].

2.1. K^2-Raster

The k^2-tree structure was originally proposed by Ladra et al. [35] as a compact representation of the adjacency matrix of a directed graph. Its applications include web graphs and social networks. Based on k^2-tree, the same authors also proposed k^2-raster [33], which is specifically designed for raster data including images. A k^2-raster is built from a matrix of width w and height h. If the matrix can be partitioned into k^2 square subquadrants of equal size, it can be used directly. Otherwise, it is necessary to enlarge the matrix to size $s \times s$, where s is computed as:

$$s = k^{\lceil \log_k max(w,\, h) \rceil}, \tag{1}$$

setting the new elements to 0. This extended matrix is then recursively partitioned into k^2 submatrices of identical size, referred to as quadrants. This process is repeated until all cells in a quadrant have the same value, or until the submatrix has size 1×1 and cannot be further subdivided. This partitioning induces a tree topology, which is represented in a bitmap T. Elements can then be accessed via a rank function. At each tree level, the maximum and minimum values of each quadrant are computed. These are then compared with the corresponding maximum and minimum values of the parent, and the differences are stored in the $Vmax$ and $Vmin$ arrays of each level. Saving the differences instead of the original values results in lower values for each node, which in turn allows a better compression with DACs or other integer encoders such as Simple9, PForDelta, etc. An example of a simple 8×8 matrix is given in Figure 2 to illustrate this process. A k^2-raster is constructed from this matrix with maximum and minimum values as given in Figure 3. Differences from the parents' extrema are then computed as explained above, resulting in the structure shown in Figure 4. Next, with the exception of the root node at the top level, the $Vmax$ and $Vmin$ arrays at all levels are concatenated to form $Lmax$ and $Lmin$, respectively. Both arrays are then compressed by an integer encoder such as DACs. The root's maximum ($rMax$) and minimum ($rMin$) values remain uncompressed. The resulting elements, which fully describe this k^2-raster structure, are given in Table 1.

2.2. Unary Codes and Notation

We denote x as a non-negative integer. The expression $|x|$ gives the minimum bit length needed to express x, i.e., $|x| = \lfloor \log_2 x \rfloor + 1$.

Unary codes are generally used for small integers. Unary codes have the following form:

$$u(x) = 0^x\, 1\,, \tag{2}$$

where the superscript x indicates the number of consecutive 0 bits in the code. For example, $u(1_d)$ = 0^1 1 = 01_b, $u(6_d)$ = 0^6 1 = 0000001_b, $u(9_d)$ = 0^9 1 = 0000000001_b. Here, bits are denoted by a subscript b and decimal numbers by a subscript d. Furthermore, when codes are composed of two parts, they are spaced apart for readability purposes. In general, the notation used in [15] is adopted in this paper.

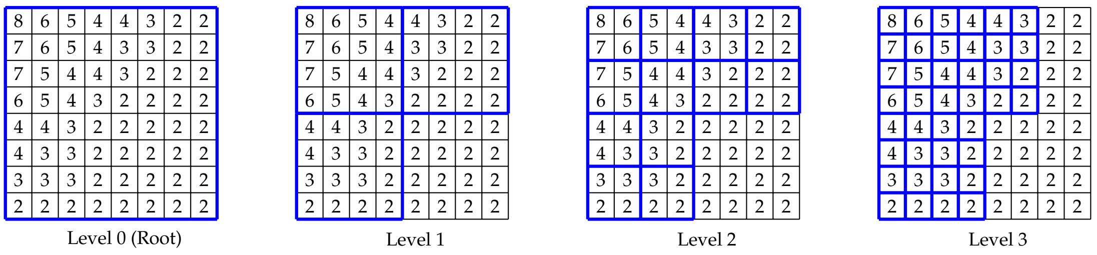


Figure 2. Subdivision of an example 8 × 8 matrix for k^2-raster ($k = 2$).

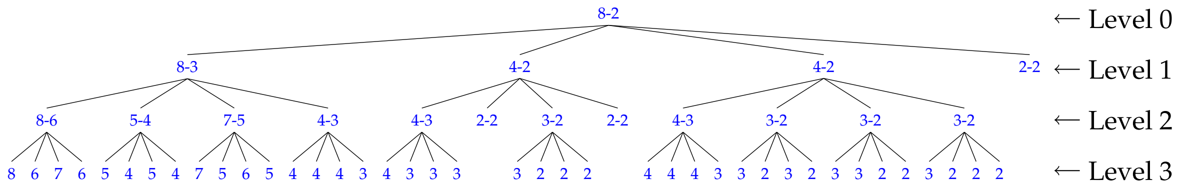


Figure 3. A k^2-raster ($k = 2$) tree storing the maximum and minimum values for each quadrant of every recursive subdivision of the matrix in Figure 2. Every node contains the maximum and minimum values of the subquadrant, separated by a dash. On the last level, only one value is shown as each subquadrant contains only one cell.

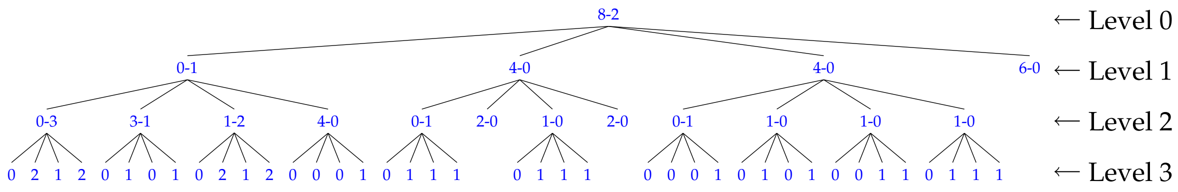


Figure 4. Based on the tree in Figure 3, the maximum value of each node is subtracted from that of its parent while the minimum value of the parent is subtracted from the node's minimum value. These differences then replace their corresponding values in the node. The maximum and minimum values of the root remain the same.

Table 1. An example of the elements of a k^2-raster based on Figures 2–4.

T Bitmap	binary		1110 1111 1010 1111
Lmax	decimal	Level 1	0446
		Level 2	0314 0212 0111
		Level 3	0212 0101 0212 0001 0111 0111 0001 0101 0011 0111
Lmin	decimal	Level 1	100
		Level 2	3120 10 1000
rMax	decimal		8
rMin	decimal		2

2.3. Elias Codes

Elias codes include Gamma (γ) codes and Delta (δ) codes. They were developed by Peter Elias [21] to encode natural numbers, and in general, they work well with sequences of small numbers.

Gamma codes have the following form:

$$\gamma(x) = 0^{|x|-1}\,[x]_{|x|} = u(|x|-1)\,[x]_{|x|-1}\,, \tag{3}$$

where $[x]_l$ represents the l least significant bits of x. For example, $\gamma(1_d) = \gamma(1_b) = 1_b$, $\gamma(4_d) = \gamma(100_b)$ = $001\ 00_b$, $\gamma(6_d) = \gamma(110_b) = 001\ 10_b$, $\gamma(9_d) = \gamma(1001_b) = 0001\ 001_b$, $\gamma(14_d) = \gamma(1110_b) = 0001\ 110_b$.

Delta codes have the following form:

$$\delta(x) = \gamma(\mid x \mid)\ [x]_{|x|-1}\ . \tag{4}$$

For values that are larger than 31, Delta codes produce shorter codewords than Gamma codes. This is due to the use of Gamma codes in forming the first part of their codes, which provides a shorter code length for Delta codes as the number becomes larger. Some examples are: $\delta(1_d) = \delta(1_b) = 1_b$, $\delta(6_d) = \delta(110_b) = 011\ 10_b$, $\delta(9_d) = \delta(1001_b) = 00100\ 001_b$, $\delta(14_d) = \delta(1110_b) = 00100\ 110_b$.

2.4. Rice Codes

Rice codes [25] are a special case of Golomb codes. Let x be an integer value in the sequence, and let $y = \lfloor x/2^l \rfloor$, where l is a non-negative integer parameter. The Rice codes for this parameter are defined as:

$$R_l(x) = u(y+1)\ [x]_l\ . \tag{5}$$

Some examples are shown for different values of l in Table 2.

Table 2. Some examples of Rice codes.

Value v		**Rice Code $R_l(v)$**			
Decimal	**Binary**	**$l = 1$**	**$l = 2$**	**$l = 3$**	**$l = 4$**
1_d	1_b	1_b	1_b	1_b	1_b
6_d	110_b	$0001\ 0_b$	$01\ 10_b$	110_b	110_b
9_d	1001_b	$00001\ 1_b$	$001\ 01_b$	$01\ 001_b$	1001_b
14_d	1110_b	$00000001\ 0_b$	$0001\ 10_b$	$01\ 110_b$	1110_b

To obtain optimal performance among Rice codes, l should be selected to be close to the expected value of the input integers. In general, Rice codes give better compression performance than Elias γ and δ codes.

2.5. Simple9, Simple16, and PForDelta

Apart from Elias codes and Rice codes, the codes in this section store the integers in single or multiple word-sized elements to achieve data compression. They have been shown to have good compression ratios [30].

Simple9 [36] assigns a maximum possible number of a certain bit length to a 28-bit segment or packing space of a 32-bit word. The other 4 bits contain a selector that has a value ranging from 0 to 8. Each selector has information that indicates how the integers are stored, and that includes the number of these integers and the maximum number of bits that each integer is allowed in this packing space. For example, Selector 0 tests to see if the first 28 integers in the data have a value of 0 or 1, i.e., a bit length of 1. If they do, then they are stored in this 28-bit segment. Otherwise, Selector 1 tests to see if it can pack 14 integers into the segment with a maximum bit length of 2 bits for each. If this still does not work, Selector 2 tests to see if 9 integers can each be packed into a maximum bit length of 3 bits. This testing goes on until the right number of data are found that can be stored in these 28 bits. Table 3 shows the 9 different ways of using 28 bits in a word of 32 bits in Simple9.

Simple16 [37] is a variant of Simple9 and uses all 16 combinations in the selector bits. Their values range from 0 to 15. Table 4 shows the 16 different ways of packing integers into the 28-bit segment in Simple16.

PForDelta [27] is also similar to both Simple9 and Simple16, but encodes a fixed group of numbers at a time. To do so, 128- or 256-bit words are used.

Due to its relative simplicity, Simple9 is used here as an example to illustrate how an integer sequence is stored in the encoders described in this section. This sequence <3591 25 13 12 15 12 11 26 20 8 13 8 9 7 13 10 12 0 10>$_d$ is taken from the *Lmax* array of one of our data scenes AG9, and the bit-packing is shown in Table 5. There are 19 integers in the sequence. Assuming the integer is 16 bits each, the sequence has a total size of 38 bytes. After packing into the array, the sequence occupies only 16 bytes.

Table 3. Nine different ways of encoding numbers in the 28-bit packing space in Simple9.

Selector	Number of Integers n	Width of Integers $\lfloor 28/n \rfloor$ (Bits)	Wasted Bits
0	28	1	0
1	14	2	0
2	9	3	1
3	7	4	0
4	5	5	3
5	4	7	0
6	3	9	1
7	2	14	0
8	1	28	0

Table 4. Sixteen different ways of encoding numbers in the 28-bit packing space in Simple16. There are no wasted bits in any of the selectors.

Selector	Number of Integers	Width of Integers (Bits)
0	28	28 × 1 bit
1	21	7 × 2 bits, 14 × 1 bit
2	21	7 × 1 bit, 7 × 2 bits, 7 × 1 bit
3	21	14 × 1 bit, 7 × 2 bits
4	14	14 × 2 bits
5	9	1 × 4 bits, 8 × 3 bits
6	8	1 × 3 bits, 4 × 4 bits, 3 × 3 bits
7	7	7 × 4 bits
8	6	4 × 5 bits, 2 × 4 bits
9	6	2 × 4 bits, 4 × 5 bits
10	5	3 × 6 bits, 2 × 5 bits
11	5	2 × 5 bits, 3 × 6 bits
12	4	4 × 7 bits
13	3	1 × 10 bits, 2 × 9 bits
14	2	2 × 14 bits
15	1	1 × 28 bits

Table 5. Example to show how the integer sequence <3591 25 13 12 15 12 11 26 20 8 13 8 9 7 13 10 12 0 10>$_d$ is stored with Simple9.

Element	Selector	Number of Integers	Integers Stored (Decimal)	Integers Stored (Binary)
0	7_d (0111_b)	2	3591 25	00111000000111 00000000011001
1	4_d (0100_b)	5	13 12 15 12 11	01101 01100 01111 01100 01011
2	4_d (0100_b)	5	26 20 8 13 8	11010 10100 01000 01101 01000
3	3_d (0011_b)	7	9 7 13 10 12 0 10	1001 0111 1101 1010 1100 0000 1010

2.6. Directly Addressable Codes

Directly Addressable Codes (DACs) can be used to compress k^2-raster and provide access to variable-length codes. Based on the concept of compact data structures, DACs were proposed in the papers published by Brisaboa et al. in 2009 [30] and 2013 [31]. This structure is proven to yield good compression ratios for variable-length integer sequences. By means of the rank function, it gains fast direct access to any position of the sequence in a very compact space. The original authors also asserted that it was best suited for a sequence of integers with a skewed frequency distribution toward smaller integer values.

Different types of encoding are used for DACs, and the one that we are interested in for k^2-raster is called VBytecoding. Consider a sequence of integers x. Each integer x_i, which is represented by $\lfloor \log_2 x_i \rfloor + 1$ bits, is broken into chunks of bits of size C_S. Each chunk is stored in a block of size $C_S + 1$ with the additional bit used as a control bit. The chunk occupies the lower bits in the block and the control bit the highest bit. The block that holds the most significant bits of the integer has its control bit set to 0, while the others have it set to 1. For example, if we have an integer 41_d (101001_b), which is 6 bits long, and if the chunk size is C_S = 3, then we have 2 blocks: $\underline{0}101\ \underline{1}001_b$. The control bit in each block is shown underlined. To show how the blocks are organized and stored, we again illustrate it with an example. Given five integers of variable length: 7_d (111_b), 41_d (101001_b), 100_d (1100100_b), 63_d (111111_b), 427_d (110101011_b), and a chunk size of 3 (the block size is 4), their representations are listed in Table 6.

Table 6. Example of an integer sequence and the corresponding DACs blocks of the integers.

Decimal	Binary	DACs Blocks
7_d	$\underline{0}111_b$	($B_{1,1}A_{1,1}$)
41_d	$\underline{0}101\ \underline{1}001_b$	($B_{2,2}A_{2,2}$ $B_{2,1}A_{2,1}$)
100_d	$\underline{0}001\ \underline{1}100\ \underline{1}100_b$	($B_{3,3}A_{3,3}$ $B_{3,2}A_{3,2}$ $B_{3,1}A_{3,1}$)
63_d	$\underline{0}111\ \underline{1}111_b$	($B_{4,2}A_{4,2}$ $B_{4,1}A_{4,1}$)
427_d	$\underline{0}110\ \underline{1}101\ \underline{1}011_b$	($B_{5,3}A_{5,3}$ $B_{5,2}A_{5,2}$ $B_{5,1}A_{5,1}$)

We store them in three blocks of arrays A and control bitmaps B. This is depicted in Figure 5. To retrieve the values in the arrays A, we make use of the corresponding bitmaps B with the rank function. This function returns the number of bits, which are set to 1 from the beginning position to the one being queried in the control bitmap B_i. An example of how the function is used follows: If we want to access the third integer (100_d) in the sequence in Figure 5, we start looking for the third element in the array A_1 in Block$_1$ and find $A_{3,1}$ with its corresponding control bitmap $B_{3,1}$. The function rank($B_{3,1}$) then gives a result of 2, which means that the second element $A_{3,2}$ in the array A_2 in Block$_2$ contains the next block. With the control bit in $B_{3,2}$, we compute the function rank($B_{3,2}$) and obtain a result of 1. This means the next block in Block$_3$ can be found in the first element $A_{3,3}$. Since its corresponding control bitmap $B_{3,3}$ is set to 0, the search ends here. All the blocks found are finally concatenated to form the third integer in the sequence.

More information on DACs and the software code can be found in the papers [30,31] by Ladra et al.

Block$_1$	A_1	111 ($A_{1,1}$)	001 ($A_{2,1}$)	100 ($A_{3,1}$)	111 ($A_{4,1}$)	011 ($A_{5,1}$)
	B_1	0 ($B_{1,1}$)	1 ($B_{2,1}$)	1 ($B_{3,1}$)	1 ($B_{4,1}$)	1 ($B_{5,1}$)
Block$_2$	A_2	101 ($A_{2,2}$)	100 ($A_{3,2}$)	111 ($A_{4,2}$)	101 ($A_{5,2}$)	
	B_2	0 ($B_{2,2}$)	1 ($B_{3,2}$)	0 ($B_{4,2}$)	1 ($B_{5,2}$)	
Block$_3$	A_3	001 ($A_{3,3}$)	110 ($A_{5,3}$)			
	B_3	0 ($B_{3,3}$)	0 ($B_{5,3}$)			

Figure 5. Organization of 3 DACs blocks.

2.7. Selection of the k Value

Following the description of Subsection 2.1, using different k values leads to the creation of $Lmax$ and $Lmin$ arrays of different lengths. This, in turn, affects the final results of the size of k^2-raster. With this in mind, we present a heuristic approach that can be used to determine the best k value for obtaining the smallest storage size. First, we compute the sizes of the extended matrix for different values of k within a suitable range using Equation (1). Then, we find the k value that corresponds to the matrix with the smallest size, and the result can be considered as the best k value. Before the start of the k^2-raster building process, the program can find the best k value and use it as the default.

2.8. Heuristic k^2-Raster

In the k^2-raster paper by Ladra et al. [33], a variant of this structure was also proposed whereby the elements at the last level of the tree structure are stored by using an entropy-based heuristic approach. This is denoted by k^2_H-raster. For example, for $k = 2$, each set of the 4 nodes that are from the same parent forms a codeword. It is possible that at this same level of the tree, these codewords may be repeated, and their frequencies of occurrences can be computed. These sets of codewords and their frequencies are then compressed and saved. In effect, the more these codewords are repeated, the less storage space they take up. An example of codeword frequency based on the k^2-raster discussed in Section 2.1 is shown in Table 7. According to experiments conducted by the authors of [33], it saves space in the final representation.

Table 7. Codeword frequency in Level 3 of the *Lmax* bitmap in the k^2-raster structure in Figure 1.

Codeword	Frequency
0111	3
0212	2
0101	2
0001	2
0011	1

2.9. 3D-2D Mapping

A study on compact representation of raster images in a time-series was proposed by Cruces et al. in [34]. This method is based on the 3D to 2D mapping of a raster where 3D tuples $<x, y, z>$ are mapped into a 2D binary grid. That is, a raster of size $w \times h$ with values in a certain range, between 0 and v inclusive, has a binary matrix of $w \times h$ columns and v+1 rows. All the rasters are then concatenated into a 3D matrix and stored as a 3D-k^2-tree.

3. Experimental Results

In this section, we present an exhaustive comparison of the different integer encoders for use with k^2-raster. First, though, we report results from experiments for finding the best k value. Reported also are the experimental results to find out if the heuristic k^2-raster and 3D-2D mapping would give better storage sizes. All storage sizes in this section are expressed as bits per pixel per band (bpppb).

The hyperspectral scenes were captured by different sensors: Atmospheric Infrared Sounder (AIRS), AVIRIS, Compact Reconnaissance Imaging Spectrometer for Mars (CRISM), Hyperion, and IASI. Except for IASI, all of them are publicly available for download (http://cwe.ccsds.org/sls/docs/sls-dc/123.0-B-Info/TestData). The hyperspectral scenes used are listed in Table 8.

The implementations for k^2-raster and k^2_H-raster were based on the algorithms presented in the paper by Ladra et al. [33]. The sdsl-lite implementation of k^2-tree by Simon Gog [38] (https://github.com/simongog/sdsl-lite/blob/master/include/sdsl/k2_tree.hpp) was used for testing 3D-2D mapping described in the paper by Cruces et al. [34]. The DACs software was downloaded from a package called "DACs, optimization with no further restrictions" at the Universidade da Coruña's Database Laboratory website (http://lbd.udc.es/research/DACS/). The programming code for the Rice, PForDelta, Simple9, and Simple16 codes was written by the programmers Diego Caro, Michael Dipperstein, and Christopher Hoobin and was downloaded from these authors' GitHub web pages. Slight modifications to the code were made to meet our requirements to perform the experiments. All programs for this paper were written in C and C++ and compiled with gnu g++ 5.4.0 20160609 with -Ofast optimization. The experiments were carried out on an Intel Core 2 Duo

CPU E7400 @2.80GHz with 3072KB of cache and 3GB of RAM. The operating system was Ubuntu 16.04.5 LTS with kernel 4.15.0-47-generic (64 bits). The software code is available at http://gici.uab.cat/GiciWebPage/downloads.php.

Table 8. Hyperspectral scenes used in our experiments. Also shown are the bit rate and bit rate reduction using k^2-raster. x is the scene width, y the scene height, and z the number of spectral bands. bpppb, bits per pixel per band; CRISM, Compact Reconnaissance Imaging Spectrometer for Mars; IASI, Infrared Atmospheric Sounding Interferometer.

Sensor	Name	C/U *	Acronym	Original Dimensions ($x \times y \times z$)	Bit Depth (bpppb)	Best k Value	k^2-raster Bit Rate (bpppb)	k^2-raster Bit-Rate Reduction (%)
AIRS	9	U	AG9	90 × 135 × 1501	12	6	9.49	21%
	16	U	AG16	90 × 135 × 1501	12	6	9.12	24%
	60	U	AG60	90 × 135 × 1501	12	15	9.72	19%
	126	U	AG126	90 × 135 × 1501	12	6	9.61	20%
	129	U	AG129	90 × 135 × 1501	12	6	8.65	28%
	151	U	AG151	90 × 135 × 1501	12	6	9.53	21%
	182	U	AG182	90 × 135 × 1501	12	6	9.68	19%
	193	U	AG193	90 × 135 × 1501	12	15	9.30	23%
AVIRIS	Yellowstone sc. 00	C	ACY00	677 × 512 × 224	16	6	9.61	40%
	Yellowstone sc. 03	C	ACY03	677 × 512 × 224	16	6	9.42	41%
	Yellowstone sc. 10	C	ACY10	677 × 512 × 224	16	6	7.62	52%
	Yellowstone sc. 11	C	ACY11	677 × 512 × 224	16	6	8.81	45%
	Yellowstone sc. 18	C	ACY18	677 × 512 × 224	16	6	9.78	39%
	Yellowstone sc. 00	U	AUY00	680 × 512 × 224	16	9	11.92	25%
	Yellowstone sc. 03	U	AUY03	680 × 512 × 224	16	9	11.74	27%
	Yellowstone sc. 10	U	AUY10	680 × 512 × 224	16	9	9.99	38%
	Yellowstone sc. 11	U	AUY11	680 × 512 × 224	16	9	11.27	30%
	Yellowstone sc. 18	U	AUY18	680 × 512 × 224	16	9	12.15	24%
CRISM	frt000065e6_07_sc164	U	C164	640 × 420 × 545	12	6	10.08	16%
	frt00008849_07_sc165	U	C165	640 × 450 × 545	12	6	10.37	14%
	frt0001077d_07_sc166	U	C166	640 × 480 × 545	12	6	11.05	8%
	hrl00004f38_07_sc181	U	C181	320 × 420 × 545	12	5	9.97	17%
	hrl0000648f_07_sc182	U	C182	320 × 450 × 545	12	5	10.11	16%
	hrl0000ba9c_07_sc183	U	C183	320 × 480 × 545	12	5	10.65	11%
Hyperion	Agricultural	C	HCA	256 × 3129 × 242	12	16	8.52	29%
	Coral Reef	C	HCC	256 × 3127 × 242	12	8	7.62	36%
	Urban	C	HCU	256 × 2905 × 242	12	16	8.85	26%
	Erta Ale	U	HUEA	256 × 3187 × 242	12	8	7.76	35%
	Lake Monona	U	HULM	256 × 3176 × 242	12	8	7.82	35%
	Mt. St. Helena	U	HUMS	256 × 3242 × 242	12	8	7.91	34%
IASI	Level 0 1	U	I01	60 × 1528 × 8359	12	12	6.32	47%
	Level 0 2	U	I02	60 × 1528 × 8359	12	12	6.38	47%
	Level 0 3	U	I03	60 × 1528 × 8359	12	12	6.31	47%
	Level 0 4	U	I04	60 × 1528 × 8359	12	12	6.43	46%

*: Calibrated (C) or Uncalibrated (U).

3.1. Best k Value Selection

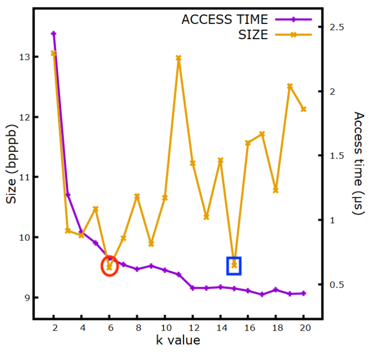
From our previous research [20], the selection of the k value when building a k^2-raster was shown to have a great effect on the resulting size of the structure, as well as the access time to query its elements. In order to further investigate this idea, we extended our research to finding ways of choosing the best k value. One way was to build the k^2-raster structure with different k values for scene data from each sensor to see how the matrix size affected the choice of the k value. Additionally, we measured the time it took to build the k^2-raster and the size of the structure. The results are shown in Table 9. For most tested data, the k value leading to the smallest extended matrix size (attribute S in the table) usually provided the fastest build time and the smallest storage size. With these results, we could say that, in general, when $k = 2$, the compressed data size was large, sometimes even larger than the size of the original scene. As the value of k became larger, beginning with $k = 3$, the compressed data size was reduced. As far as the compressed size was concerned, the best value was in the range from three to 10 for matrices with a small raster size (i.e., if both the original width and original height were less than 1000) such as the ones for the AIRS Granule or AVIRIS Yellowstone scenes. If at least one dimension was larger than 1000 such as Hyperion calibrated or uncalibrated scenes, a larger range, typically between three and 20, needed to be considered.

Table 9. Results for different k values using the scene data from each sensor for the following attributes: (S) the extended matrix Size (pixels), (C) the k^2-raster Compressed storage data rate (bpppb), and (B) the time to Build the k^2-raster (seconds). The original scene width and height are shown in the first column. The best results are highlighted in blue.

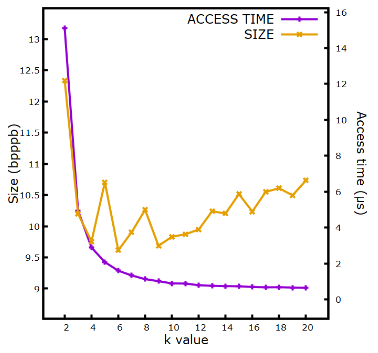
Scene Data (w × h) *		***k*=2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
AG9 (90 × 135)	S	256	243	256	625	216	343	512	729	1000	1331	144	169	196	225	256	289	324	361	400
	C	13.06	10.11	10.03	10.47	9.49	9.98	10.68	9.89	10.65	12.98	11.23	10.33	11.29	9.53	11.57	11.72	10.78	12.52	12.13
	B	5.3	3.2	4.1	10.9	4.2	10.9	12.6	10.7	17.5	29.6	2.9	4.1	3.0	4.3	4.6	6.6	6.8	4.9	7.3
ACY00 (677 × 512)	S	1024	729	1024	3125	1296	2401	4096	729	1000	1331	1728	2197	2744	3375	4096	4913	5832	6859	8000
	C	12.34	10.20	9.76	10.70	9.61	9.91	10.26	9.69	9.83	9.87	9.95	10.24	10.20	10.51	10.24	10.55	10.61	10.49	10.73
	B	19.5	10.7	10.8	30.7	10.5	19.3	42.0	8.8	9.3	11.5	13.2	17.1	23.2	29.1	45.5	55.8	72.6	101.1	131.0
AUY00 (680 × 512)	S	1024	729	1024	3125	1296	2401	4096	729	1000	1331	1728	2197	2744	3375	4096	4913	5832	6859	8000
	C	15.31	12.93	12.20	13.06	12.08	12.35	12.47	11.92	12.11	12.13	12.17	12.52	12.43	12.84	12.44	12.83	12.87	12.69	12.96
	B	18.4	10.7	10.1	30.7	11.4	20.9	41.4	7.7	8.5	10.9	12.7	17.1	22.9	29.3	44.3	55.8	73.0	101.0	130.6
C164 (640 × 420)	S	1024	729	1024	3125	1296	2401	4096	729	1000	1331	1728	2197	2744	3375	4096	4913	5832	6859	8000
	C	12.60	10.42	10.17	11.35	10.08	10.46	11.12	10.34	10.20	10.76	10.48	10.96	10.66	10.77	11.19	11.18	11.55	11.80	11.30
	B	47.1	28.8	27.9	74.3	27.7	47.3	98.6	19.3	21.1	24.8	29.7	38.7	49.4	69.6	96.2	133.3	179.3	231.1	314.9
HCA (256 × 3129)	S	4096	6561	4096	15625	7776	16807	4096	6561	10000	14641	20736	28561	38416	3375	4096	4913	5832	6859	8000
	C	17.2	15.64	9.79	-	10.47	-	8.54	9.13	9.7	-	-	-	-	8.65	8.52	8.75	9.16	9.07	8.92
	B	121.9	183.6	68.7	-	186.6	-	55.2	115.6	238.9	-	-	-	-	44.3	56.7	70.6	91.9	121.8	156.6
HUEA (256 × 3187)	S	4096	6561	4096	15625	7776	16807	4096	6561	10000	14641	20736	28561	38416	3375	4096	4913	5832	6859	8000
	C	16.02	14.63	8.89	-	9.68	-	7.76	8.46	9.00	-	-	-	-	8.50	7.80	8.69	8.68	8.46	8.27
	B	131.1	189.0	74.4	-	172.3	-	60.3	120.8	245.3	-	-	-	-	49.5	60.4	75.7	95.3	123.3	159.4
I01 (60 × 1528)	S	2048	2187	4096	3125	7776	2401	4096	6561	10000	14641	1728	2197	2744	3375	4096	4913	5832	6859	8000
	C	21.99	12.59	17.98	10.25	24.60	7.71	9.33	12.23	16.97	26.49	6.32	7.28	8.28	6.80	7.64	8.54	9.44	10.43	8.25
	B	1021.1	780.5	1728.7	986.5	5167.5	635.6	1426.7	3938.6	7870.7	17973.9	339.7	474.3	658.9	944.0	1498.1	1826.6	2789.8	3543.2	4810.3

*: w = scene width; h = scene height.

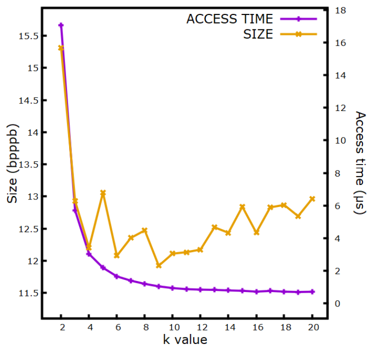
The above experiments were repeated to compare the access time for the different k values. For each scene, the average time over 100,000 consecutive queries is reported. Results are shown in Table 10, and Figure 6 shows how the access time and the size varied depending on the k value. As can be observed, access time became smaller and smaller as the value of k became larger. The plotted data suggested that there was a trade-off between access time and size with respect to the k value. We considered the optimal k value to be the one that created a relatively small size with a minimal access time. For example in AG9, when comparing the results between $k = 6$ and $k = 15$, the difference in bits per pixel per band for storage size was not very significant, but the reduction in access time was. Therefore, for this scene, $k = 15$ was considered an optimal value.



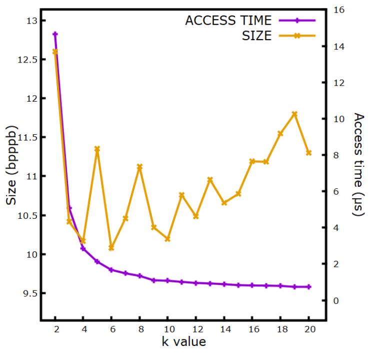
(**a**) AIRS Granule 9



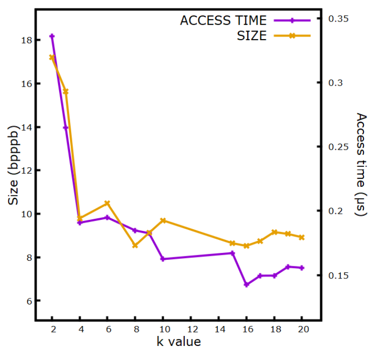
(**b**) Yellowstone 00 (cal.)



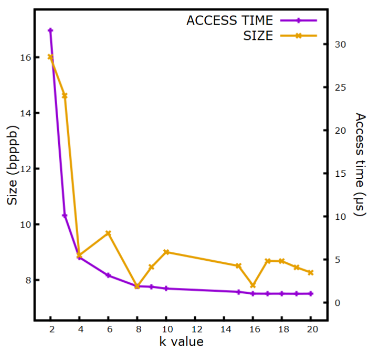
(**c**) Yellowstone 00 (uncal.)



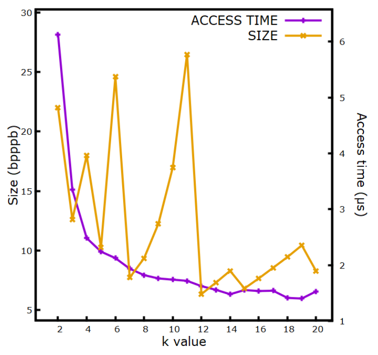
(**d**) CRISM 164



(**e**) Hyperion Agricultural



(**f**) Hyperion Erta Ale



(**g**) IASI Level 0 1

Figure 6. A comparison of the storage size (bpppb) and access time (µs) for different k values of k^2-raster built from scenes in our datasets. Access time is the average time of 100,000 consecutive queries. For AIRS Granule 9, the best value is marked with a red circle, and the optimal value is marked with a blue square.

Table 10. Access time (μs) for a random cell query with different kvalues. Each result is the average time over 100,000 consecutive queries. The best results are highlighted in blue.

Scene Data (w × h) *	***k* = 2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
AG9 (90 × 135)	2.45	1.20	0.91	0.82	0.70	0.65	0.61	0.64	0.61	0.57	0.47	0.47	0.48	0.47	0.45	0.42	0.46	0.43	0.43
ACY00 (677 × 512)	15.10	4.95	2.89	2.07	1.60	1.33	1.13	1.01	0.89	0.88	0.79	0.76	0.74	0.73	0.69	0.67	0.68	0.66	0.64
AUY00 (680 × 512)	17.07	5.70	3.03	2.19	1.66	1.40	1.20	1.05	0.94	0.88	0.85	0.83	0.80	0.77	0.72	0.77	0.72	0.70	0.72
C164 (640 × 420)	14.66	5.09	2.84	2.12	1.67	1.48	1.34	1.10	1.08	1.01	0.95	0.93	0.88	0.83	0.81	0.81	0.79	0.74	0.74
HCA (256 × 3129)	0.34	0.26	0.19	-	0.19	-	0.18	0.18	0.16	-	-	-	-	0.17	0.14	0.15	0.15	0.16	0.16
HUEA (256 × 3187)	31.59	10.09	5.24	-	3.11	-	1.87	1.81	1.60	-	-	-	-	1.22	1.02	1.01	1.01	1.00	1.01
I01 (60 × 1528)	6.13	3.35	2.48	2.24	2.13	1.94	1.82	1.76	1.74	1.72	1.62	1.56	1.48	1.55	1.54	1.54	1.41	1.40	1.53

*: w = scene width; h = scene height.

3.2. Heuristic k^2-Raster

In this section, we present the results of the experiments using the heuristic k^2-raster proposed by Ladra et al. [33] on some of our datasets. Table 11 reports results for two hyperspectral scene datasets: AIRS Granule and AVIRIS Uncalibrated Yellowstone. In the experiments, we found that only when $k = 2$ would there be enough repeated sets of codewords in the last level of nodes to help us save space. When $k \geq 3$, there were no repeated sets of codewords. From the table, it can be seen that there was not much size reduction with k_H^2-raster in most cases. However, if we built a k^2-raster using the best or optimal k value, the size was considerably smaller. Therefore, we can see that k_H^2-raster structure did not produce a better size.

Table 11. Comparison of the structure size (bpppb) built from k^2-raster and k_H^2-raster where $k = 2$. The sizes for k^2-raster using the best k value and the optimal k value are also shown. The best results are highlighted in blue.

AIRS Granule	**k^2-Raster ($k = 6$) (Best)**	**k^2-Raster ($k = 15$) (Optimal)**	**k^2-Raster ($k = 2$)**	**k_H^2-Raster ($k = 2$)**
AG9	9.49	9.53	13.06	13.22
AG16	9.12	9.17	12.72	12.85
AG60	9.81	9.72	13.65	13.86
AG126	9.61	9.72	13.42	13.59
AG129	8.65	8.72	11.98	11.95
AG151	9.53	9.56	13.19	13.35
AG182	9.68	9.71	13.32	13.47
AG193	9.44	9.30	13.29	13.43
AVIRIS Uncalibrated	**k^2-Raster ($k = 9$) (Best)**	**k^2-Raster ($k = 9$) (Optimal)**	**k^2-Raster ($k = 2$)**	**k_H^2-Raster ($k = 2$)**
AUY00	11.92	11.92	15.31	15.19
AUY03	11.74	11.74	15.03	14.74
AUY10	9.99	9.99	12.85	11.86
AUY11	11.27	11.27	14.27	14.08
AUY18	12.15	12.15	15.36	15.25

3.3. 3D-2D Mapping

As discussed earlier, Cruces et al. [34] proposed a 3D to 2D mapping of raster images using k^2-tree as an alternative to achieve a better compression ratio. We used the k^2-tree implementation in sdsl-lite software to obtain the sizes for one of our datasets (AG9) from $k = 2$ to $k = 4$. Note that similar to k^2-raster, if the 2D binary matrix cannot be partitioned into square subquadrants of equal size, it needs to be expanded using Equation (1), and the extra elements are set to zero. The results are presented in Table 12. The sizes for a range of bands from 1481 to 1500 of the scene are also given for comparison.

From the results for AG9, we can see that the 3D-2D mapping did not make the size smaller. Instead, it became larger when the k value increased, and therefore, the method did not produce competitive results.

3.4. Comparison of Integer Encoders for k^2-Raster

Experiments were conducted to determine whether other variable-length encoders of integers might serve as a better substitute for DACs, which were the original choice in the k^2-raster structure initially proposed by Ladra et al. [33]. The performance of DACs was compared to that of other encoders such as Rice, Simple9, PForDelta, Simple16 codes, and gzip. In these experiments, the *Lmax* and *Lmin* arrays were encoded using these codes, and the results are shown in Table 13. For Rice codes, the l value, as explained in Section 2.4, produced different results depending on the mean of the raster's elements, and only the ones with the best l value are shown.

Table 12. The sizes of AIRS Granule (AG9) produced by 3D to 2D mapping from $k = 2$ to $k = 4$. The individual band sizes ranging from 1481 to 1500 are also shown. Sizes for individual bands are in bits per pixel (bpp), while the ones for all bands are in bits per pixel per band (bpppb).

Band	**Original Size**	**k^2-Tree $k = 2$**	**k^2-Tree $k = 3$**	**k^2-Tree $k = 4$**
All bands	16	16.53	20.57	26.57
1481	16	17.56	22.00	28.45
1482	16	17.27	21.54	27.84
1483	16	17.19	21.47	27.67
1484	16	17.45	21.81	28.18
1485	16	16.93	21.10	27.29
1486	16	17.09	21.27	27.50
1487	16	16.82	21.06	27.02
1488	16	17.01	21.21	27.34
1489	16	17.23	21.51	27.78
1490	16	16.94	21.10	27.20
1491	16	16.80	20.86	26.96
1492	16	16.56	20.64	26.51
1493	16	16.80	20.91	26.89
1494	16	16.84	20.93	26.98
1495	16	16.69	20.88	26.72
1496	16	16.66	20.75	26.66
1497	16	16.70	20.87	26.73
1498	16	16.61	20.70	26.58
1499	16	16.67	20.73	26.78
1500	16	16.39	20.40	26.18

Table 13. A comparison of the storage size (in bpppb) using different integer encoders on *Lmax* and *Lmin* from the k^2-raster built from our datasets. The combined entropies for *Lmax* and *Lmin* are listed as a reference. The *l* value that was used in Rice codes is enclosed in brackets. The best and optimal *k* values for DACs are also enclosed in brackets. Except for the entropy, the best rates for each scene's data are highlighted in blue.

Hyperspectral Scene	**Entropy ($Lmax + Lmin$)**	**Rice (l Value)**	**Simple9**	**PForDelta**	**Simple16**	**DACs (Best k)**	**DACs (Optimal k)**	**gzip**
AG9	8.29	10.10 (7)	10.06	9.88	9.69	9.49 (6)	9.53 (15)	12.45
AG16	7.92	9.88 (7)	9.64	9.55	9.30	9.12 (6)	9.17 (15)	11.96
AG60	8.58	10.31 (7)	10.50	10.19	10.12	9.72 (15)	9.81 (6)	12.79
AG126	8.42	10.34 (7)	10.25	9.98	9.81	9.61 (6)	9.72 (15)	12.55
AG129	7.47	9.66 (7)	9.01	9.01	8.61	8.65 (6)	8.72 (15)	11.21
AG151	8.36	10.39 (7)	9.99	9.79	9.54	9.53 (6)	9.56 (15)	12.39
AG182	8.44	10.58 (7)	10.44	10.09	10.01	9.68 (6)	9.71 (15)	12.71
AG193	8.25	10.26 (7)	10.06	9.93	9.65	9.30 (15)	9.44 (6)	12.33
ACY00	8.81	9.89 (7)	10.37	9.80	10.11	9.61 (6)	9.69 (9)	12.56
ACY03	8.48	9.70 (7)	9.80	9.40	9.57	9.42 (6)	9.50 (9)	11.98
ACY10	6.88	9.18 (7)	7.34	7.43	7.18	7.62 (6)	7.74 (9)	9.32
ACY11	8.12	9.45 (7)	9.32	9.02	9.09	8.81 (6)	9.00 (9)	11.61
ACY18	8.96	10.58 (7)	10.52	9.84	10.28	9.78 (6)	9.88 (9)	12.66
AUY00	11.16	17.59 (7)	14.01	11.93	13.79	11.92 (9)	11.92 (9)	15.13
AUY03	10.83	16.59 (7)	13.54	11.56	13.29	11.74 (9)	11.74 (9)	14.59
AUY10	9.26	12.87 (7)	10.90	9.61	10.54	9.99 (9)	9.99 (9)	12.29
AUY11	10.60	15.16 (7)	13.12	11.24	12.89	11.27 (9)	11.27 (9)	14.47
AUY18	11.38	20.70 (7)	14.19	12.10	14.01	12.15 (9)	12.15 (9)	15.53
C164	9.18	10.33 (7)	11.35	10.44	11.14	10.08 (6)	10.08 (6)	12.85
C165	9.48	10.91 (7)	11.78	10.69	11.57	10.37 (6)	10.37 (6)	13.17
C166	10.02	12.83 (7)	12.99	11.41	12.74	11.05 (6)	11.05 (6)	13.61
C181	9.16	9.96 (7)	10.93	10.53	10.72	9.97 (5)	9.97 (5)	13.37
C182	9.27	10.17 (7)	11.24	10.67	10.99	10.11 (5)	10.11 (5)	13.26
C183	9.60	11.15 (7)	12.33	11.21	12.05	10.65 (5)	10.65 (5)	13.32

Table 13. *Cont.*

Hyperspectral Scene	Entropy ($Lmax + Lmin$)	Rice (l Value)	Simple9	PForDelta	Simple16	DACs (Best k)	DACs (Optimal k)	gzip
HCA	7.59	8.94 (7)	9.79	8.80	9.56	8.52 (16)	8.54 (8)	11.20
HCC	6.75	8.20 (7)	8.28	7.60	7.93	7.62 (8)	7.71 (16)	9.51
HCU	7.87	9.78 (7)	10.30	8.91	10.04	8.85 (16)	8.86 (8)	11.35
HUEA	6.66	7.67 (5)	8.30	7.99	8.00	7.76 (8)	7.80 (16)	9.85
HULM	6.71	7.66 (5)	8.38	8.11	8.10	7.82 (8)	7.88 (16)	10.13
HUMS	6.77	7.90 (5)	8.48	8.14	8.20	7.91 (8)	7.94 (16)	10.12
I01	5.39	6.51 (4)	6.26	6.54	5.94	6.32 (12)	6.80 (15)	7.46
I02	5.46	6.56 (4)	6.27	6.55	5.96	6.38 (12)	6.84 (15)	7.51
I03	5.42	6.51 (4)	6.19	6.48	5.89	6.31 (12)	6.79 (15)	7.39
I04	5.51	6.62 (4)	6.37	6.65	6.04	6.43 (12)	6.90 (15)	7.63

The results showed that, in most cases, DACs still provided the best storage size compared to other encoders for our datasets. They also had the added advantage of direct random access to individual elements of the matrix whilst the other encoders would need to decompress each raster in order to retrieve the element, thus requiring much longer access time. When DACs did not yield the best performance, DACs results were usually only less than 0.1 bpppb worse. In the worst cases, DACs results lagged behind by, at most, 0.4 bpppb.

4. Conclusions

In this research, we examined the possibility of using different integer coding methods for k^2-raster and concluded that this compact data structure worked best when it was used in tandem with DACs encoding. The other variable-length encoders, though having competitive compression ratios, lacked the ability to provide users with direct access to the data. We also studied a method whereby we could obtain a k value that gave a competitive storage size and, in most cases, also a suitable access time.

For future work, we are interested in investigating the feasibility of modifying elements in a k^2-raster structure, facilitating data replacements without having to go through cycles of decompression and compression for the entire compact data structure.

Author Contributions: Conceptualization, K.C., D.E.O.T., M.H.-C., I.B., and J.S.-S.; methodology, K.C., D.E.O.T., M.H.-C., I.B., and J.S.-S.; software, K.C.; validation, K.C., I.B., and J.S.-S.; formal analysis, K.C., D.E.O.T., M.H., I.B., and J.S.-S.; investigation, K.C., D.E.O.T., M.H.-C., I.B., and J.S.-S.; resources, K.C., D.E.O.T., M.H.-C., I.B., and J.S.-S.; data curation, K.C., I.B., and J.S.-S.; writing, original draft preparation, K.C., I.B., and J.S.-S.; writing, review and editing, K.C., M.H.-C., I.B., and J.S.-S.; visualization, K.C., I.B., and J.S.-S.; supervision, I.B. and J.S.-S.; project administration, I.B. and J.S.-S.; funding acquisition, M.H.-C., I.B., and J.S.-S. All authors read and agreed to the published version of the manuscript.

Funding: This research was funded by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under Grants RTI2018-095287-B-I00 and TIN2015-71126-R (MINECO/FEDER, UE) and BES-2016-078369 (Programa Formación de Personal Investigador), by the Catalan Government under Grant 2017SGR-463, by the postdoctoral fellowship program Beatriu de Pinós, Reference 2018-BP-00008, funded by the Secretary of Universities and Research (Government of Catalonia), and by the Horizon 2020 program of research and innovation of the European Union under the Marie Skłodowska-Curie Grant Agreement #801370.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Clark, R.N.; Roush, T.L. Reflectance spectroscopy: Quantitative analysis techniques for remote sensing applications. *J. Geophys. Res. Solid Earth* **1984**, *89*, 6329–6340. [CrossRef]
2. Goetz, A.F.; Vane, G.; Solomon, J.E.; Rock, B.N. Imaging spectrometry for earth remote sensing. *Science* **1985**, *228*, 1147–1153. [CrossRef] [PubMed]
3. Kruse, F.A.; Lefkoff, A.; Boardman, J.; Heidebrecht, K.; Shapiro, A.; Barloon, P.; Goetz, A. The spectral image processing system (SIPS)-interactive visualization and analysis of imaging spectrometer data. *AIP Conf. Proc.* **1993**, *283*, 192–201.

4. Joseph, W. Automated spectral analysis: A geologic example using AVIRIS data, north Grapevine Mountains, Nevada. In Proceedings of the Tenth Thematic Conference on Geologic Remote Sensing, Environmental Research Institute of Michigan, San Antonio, TX, USA, 9–12 May 1994; pp. 1407–1418.
5. Asner, G.P. Biophysical and biochemical sources of variability in canopy reflectance. *Remote Sens. Environ.* **1998**, *64*, 234–253. [CrossRef]
6. Parente, M.; Kerekes, J.; Heylen, R. A Special Issue on Hyperspectral Imaging [From the Guest Editors]. *IEEE Geosci. Remote Sens. Mag.* **2019**, *7*, 6–7. [CrossRef]
7. Ientilucci, E.J.; Adler-Golden, S. Atmospheric Compensation of Hyperspectral Data: An Overview and Review of In-Scene and Physics-Based Approaches. *IEEE Geosci. Remote Sens. Mag.* **2019**, *7*, 31–50. [CrossRef]
8. Khan, M.J.; Khan, H.S.; Yousaf, A.; Khurshid, K.; Abbas, A. Modern trends in hyperspectral image analysis: A review. *IEEE Access* **2018**, *6*, 14118–14129. [CrossRef]
9. Theiler, J.; Ziemann, A.; Matteoli, S.; Diani, M. Spectral Variability of Remotely Sensed Target Materials: Causes, Models, and Strategies for Mitigation and Robust Exploitation. *IEEE Geosci. Remote Sens. Mag.* **2019**, *7*, 8–30. [CrossRef]
10. Sun, W.; Du, Q. Hyperspectral band selection: A review. *IEEE Geosci. Remote Sens. Mag.* **2019**, *7*, 118–139. [CrossRef]
11. Scafutto, R.D.M.; de Souza Filho, C.R.; de Oliveira, W.J. Hyperspectral remote sensing detection of petroleum hydrocarbons in mixtures with mineral substrates: Implications for onshore exploration and monitoring. *ISPRS J. Photogramm. Remote Sens.* **2017**, *128*, 146–157. [CrossRef]
12. Bishop, C.A.; Liu, J.G.; Mason, P.J. Hyperspectral remote sensing for mineral exploration in Pulang, Yunnan Province, China. *Int. J. Remote Sens.* **2011**, *32*, 2409–2426. [CrossRef]
13. Le Marshall, J.; Jung, J.; Zapotocny, T.; Derber, J.; Treadon, R.; Lord, S.; Goldberg, M.; Wolf, W. The application of AIRS radiances in numerical weather prediction. *Aust. Meteorol. Mag.* **2006**, *55*, 213–217.
14. Robichaud, P.R.; Lewis, S.A.; Laes, D.Y.; Hudak, A.T.; Kokaly, R.F.; Zamudio, J.A. Postfire soil burn severity mapping with hyperspectral image unmixing. *Remote Sens. Environ.* **2007**, *108*, 467–480. [CrossRef]
15. Navarro, G. *Compact Data Structures: A Practical Approach*; Cambridge University Press: Cambridge, UK, 2016.
16. Jacobson, G. Space-efficient static trees and graphs. In Proceedings of the 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, NC, USA, 30 October–1 November 1989; pp. 549–554.
17. Consultative Committee for Space Data Systems (CCSDS). *Image Data Compression CCSDS 123.0-B-1*; Blue Book; CCSDS: Washington, DC, USA, 2012.
18. Jolliffe, I.T. *Principal Component Analysis*; Springer: Berlin, Germany, 2002; p. 487.
19. Taubman, D.S.; Marcellin, M.W. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*; Kluwer Academic Publishers: Boston, MA, USA, 2001.
20. Chow, K.; Tzamarias, D.E.O.; Blanes, I.; Serra-Sagristà, J. Using Predictive and Differential Methods with K2-Raster Compact Data Structure for Hyperspectral Image Lossless Compression. *Remote Sens.* **2019**, *11*, 2461. [CrossRef]
21. Elias, P. Efficient storage and retrieval by content and address of static files. *J. ACM (JACM)* **1974**, *21*, 246–260. [CrossRef]
22. Ottaviano, G.; Venturini, R. Partitioned Elias-Fano indexes. In Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval, Gold Coast, Australia, 6–11 July 2014; pp. 273–282.
23. Pibiri, G.E. Dynamic Elias-Fano Encoding. Master's Thesis, University of Pisa, Pisa, Italy, 2014.
24. Sugiura, R.; Kamamoto, Y.; Harada, N.; Moriya, T. Optimal Golomb-Rice Code Extension for Lossless Coding of Low-Entropy Exponentially Distributed Sources. *IEEE Trans. Inf. Theory* **2018**, *64*, 3153–3161. [CrossRef]
25. Rice, R.; Plaunt, J. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Trans. Commun. Technol.* **1971**, *19*, 889–897. [CrossRef]
26. Rojals, J.S.; Karczewicz, M.; Joshi, R.L. Rice Parameter Update for Coefficient Level Coding in Video Coding Process. U.S. Patent 9,936,200, 3 April 2018.
27. Zukowski, M.; Heman, S.; Nes, N.; Boncz, P. Super-scalar RAM-CPU cache compression. In Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), Atlanta, GA, USA, 3–7 April 2006; pp. 59–59.
28. Silva-Coira, F. Compact Data Structures for Large and Complex Datasets. Ph.D. Thesis, Universidade da Coruña, A Coruña, Spain, 2017.

29. Al Hasib, A.; Cebrian, J.M.; Natvig, L. V-PFORDelta: Data compression for energy efficient computation of time series. In Proceedings of the 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), Bengaluru, India, 16–19 December 2015; pp. 416–425.
30. Brisaboa, N.R.; Ladra, S.; Navarro, G. DACs: Bringing direct access to variable-length codes. *Inf. Process. Manag.* **2013**, *49*, 392–404. [CrossRef]
31. Brisaboa, N.R.; Ladra, S.; Navarro, G. Directly addressable variable-length codes. In Proceedings of the International Symposium on String Processing and Information Retrieval, Saariselkä, Finland, 25–27 August 2009; pp. 122–130.
32. Baruch, G.; Klein, S.T.; Shapira, D. A space efficient direct access data structure. *J. Discret. Algorithms* **2017**, *43*, 26–37. [CrossRef]
33. Ladra, S.; Paramá, J.R.; Silva-Coira, F. Scalable and queryable compressed storage structure for raster data. *Inf. Syst.* **2017**, *72*, 179–204. [CrossRef]
34. Cruces, N.; Seco, D.; Gutiérrez, G. A compact representation of raster time series. In Proceedings of the Data Compression Conference (DCC), Snowbird, UT, USA, 26–29 March 2019; pp. 103–111.
35. Brisaboa, N.R.; Ladra, S.; Navarro, G. k 2-trees for compact web graph representation. In Proceedings of the International Symposium on String Processing and Information Retrieval, Saariselkä, Finland, 25–27 August 2009; pp. 18–30.
36. Anh, V.N.; Moffat, A. Inverted index compression using word-aligned binary codes. *Inf. Retr.* **2005**, *8*, 151–166. [CrossRef]
37. Zhang, J.; Long, X.; Suel, T. Performance of compressed inverted list caching in search engines. In Proceedings of the 17th International Conference on World Wide Web, Beijing, China, 21–25 April 2008; pp. 387–396.
38. Gog, S.; Beller, T.; Moffat, A.; Petri, M. From theory to practice: Plug and play with succinct data structures. In Proceedings of the International Symposium on Experimental Algorithms, Copenhagen, Denmark, 29 June–1 July 2014; pp. 326–337.