Modern Applied Science; Vol. 14, No. 4; 2020
ISSN 1913-1844 E-ISSN 1913-1852
Published by Canadian Center of Science and Education

# An Efficient Two-level Dictionary-based Technique for Segmentation and Compression Compound Images

Dr. Nidhal Kamel Taha El-Omari[1]

[1]Department of Software Engineering, Faculty of Information Technology, The World Islamic Sciences and Education University (WISE), Amman – Jordan

Correspondence: Dr. Nidhal Kamel Taha El-Omari, Department of Software Engineering, Faculty of Information Technology, The World Islamic Sciences and Education University (WISE), Amman – Jordan.

E-mail: nidhal.omari@wise.edu.jo; omari_nidhal@yahoo.com

Received: February 22, 2019 Accepted: March 22, 2020 Online Published: March 24, 2020

doi:10.5539/mas.v14n4p52 URL: https://doi.org/10.5539/mas.v14n4p52

**Abstract**

Image data compression algorithms are essential for getting storage space reduction and, perhaps more importantly, to increase their transfer rates, in terms of space-time complexity. Considering that there isn't any encoder that gives good results across all image types and contents, this paper proposed an evolvable lossless statistical block-based technique for segmentation and compression compound or mixed documents that have different content types, such as pictures, graphics, and/or texts.

Derived from the number of detected colors and to achieve better compression ratios, a new well-organized representation of the image is created which nonetheless retains the same image components. With the effort of reducing noise or other variations inside the scanned image, some primary operations are implemented. Thereafter, the proposed algorithm breaks down the compound document image into equal-size-square blocks. Next, inspired by the number of colors detected in each block, these blocks are categorized into a set of six-image objects, called classes, where each one contains a set of closely interrelated pixels that share the same common relevant attributes like color gamut and number, color occurrence, grey level, and others. After that, a new arrangement of these coherent classes is formed using the Lookup Dictionary Table (LUD), which is the real essence of this proposed algorithm. In order to form distinguishable labeled regions sharing the same attributes, adjacent blocks of similar color features are consolidated together into a single coherent whole entity, called segments or regions. After each region is encoded by one of the most off-the-shelf applicable compression techniques, these regions are eventually fused together into a single data file which then subjects to another compression stage to ensure better compression ratios. After the proposed algorithm has been applied and tested on a database containing 3151 24-bit-RGB-bitmap document images, the empirically-based results prove that the overall algorithm is efficient in the long run and has superior storage space reduction when compared with other existing algorithms. As for the empirical findings, the proposed algorithm has achieved (71.039%) relative reduction in the data storage space.

**Keywords:** adaptive compression, block-based segmentation, Cloud Computing (CC), Digital Image Processing (DIP), image document compression, image segmentation, Lookup Dictionary Table (LUD), lossless image compression technique

## 1. Introduction

RGB images, referred to as component images, are the most common model of images. Each image may be regarded as a "stack" containing three-equal-size arrays. Working at the level of the pixels which make up images, every image has an MxNx3 array of color pixels. This means that the image contains "M" pixels along the horizontal direction, called image width, and "N" pixels along the vertical direction, called image length. Hence, the total pixel count is "M" multiplied by "N", namely "MxN". Moreover, each pixel is associated with three integers that correspond to the three color information: Red, Green, and Blue. The number of bits that are required to address every integer of these three integers defines the bit depth which is also referred to as "pixel depth", "the number of bits per pixel", or "grey-scale resolution". (Kumar et al. 2019)(Gonzalez, Woods, and L.Eddins 2009)(Gonzalez and Woods 2017)(MathWorks Inc 2019)

As illustrated in Figure 1, Digital Image processing (DIP) aims to solve one or more of the following four overlapped cases that synergistically interact with each other (Petrou and Bosdogianni 2010)(Gonzalez et al. 2009):

- **Image Enhancement**: This is the case when the output image, "**D(M,N)**", is "better" than the input image, "**R(M,N)**". The idea is to manipulate an input image so that the final image is more suitable than the original one for a specific application. Unfortunately, there is no single image processing method that always improves document image quality. The enhancement is a problem-oriented to improve the quality of an image and depends on that specified application. However, depending on the application, such transformations improve some images and degrade others.
- **Image Restoration**: If the output image, "**D(M,N)**", is a recover image of an input image, "**R(M,N)**", this is the problem of image restoration. It is the process of retrieving degraded or destroyed images using some prior knowledge that is associated with the degradation experiences.

  While both of them are referred to as image deformation, the aim of both Image Enhancement and Image Restoration is to improve image quality in some sense in order to reproduce the original image. But, while Image Enhancement is trying to improve the image using subjective criteria, the Image Restoration is trying to reverse specific damage suffered by the image, using objective criteria.
- **Automatic Vision**: By mimicking the behaviours of human beings in solving problems, scientists from different domains try to come up with new goal-oriented operating methods to solve many important real-world problems. And so, automatic vision is that field of DIP which targets toward making computers imitate human vision or human intelligence. In some references, it is also referred to as "Machine Vision", "Machine Intelligence", "Computer Vision" or "Artificial Vision". The problem of preparation an image for Automatic Vision is close-related to the case when the output image, "**D(M,N)**", grasps some certain features of the input image, "**R(M,N)**". Image segmentation is classified as an early stage of this category. It is used to break down an image into smaller segments, called regions.
- **Image Compression**: If the output image, "**D(M,N)**", is viewed by fewer bits than the original input image, "**R(M,N)**", then the former one is called a compressed version of the later one.

As they are intertwined in many ways and with respect to the fact that there are no clear-cut boundaries that are determining where the aforementioned cases start and stop, the last two tackled problems, segmentation and compression, are the scope area of this research and, in turn, the proposed solution has been evolved from the linkage between them.

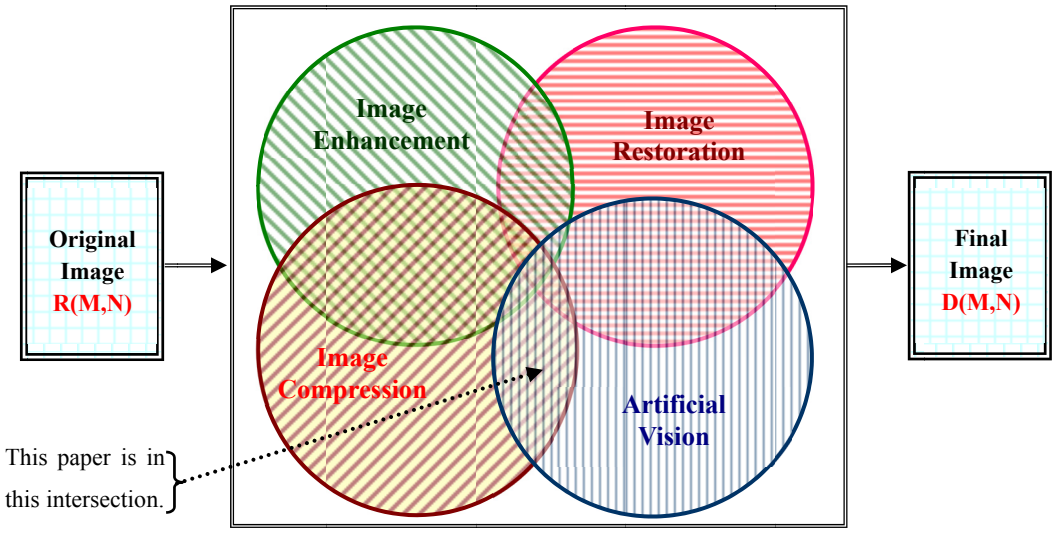


Figure 1. Image Processing Purpose

Most scanned documents are what is called compound or mixed documents, which means that the images consist of a mixture of pictures, graphics (drawings), texts, and backgrounds (El-Omari and Awajan 2009)(El-Omari 2008)(El-Omari et al. 2017). Using these documents without compressing them requires a large mass of data storage capacities and, perhaps more importantly, some "expensive" run-time computations, high-speed network connectivity, and extensive input/output operations to store and transmit data (Kumar et al. 2019)(Rahman and

Hamada 2019). Besides, they may never satisfy the ever-growing information demands of customers or even some of their evident needs (El-Omari 2019)(El-Omari et al. 2012). This is especially true for storing and transferring documents containing a huge amount of images. To come up with this focal point and to diminish the data storage requirements, in terms of storage space complexity, or to increase their transfer rates, in terms of time complexity, there is an essential need for compressing these documents with sophisticated algorithms (El-Omari et al. 2017)(Taha et al. 2012)(El-Omari and Awajan 2009)(El-Omari 2008).

Most data compression techniques generally benefit from the patterns inside the image data to get another equivalent less-space representation. And so, random data are so difficult, if not impossible, to be compressed. However, conventional mechanisms of compression are commonly involved with certain image types that are measured in terms of space-time complexity. Using them with mixed documents may impose many distinctive challenges that have to be adequately addressed. And, thus, the so-called segmentation is evolved out to offer a conceptual way to break down mixed documents into distinct image objects, called segments or regions, where each one of these incorporated parts contains a set of close-related pixels which have "common attributes" like color gamut and number, color occurrence, grey level, and others (Sharma 2019). Rather than using one standard compression technique for the whole document and to achieve a better compression ratio, each segment is extracted alone and then encoded independently (Taha et al. 2012). Thus, this research tackles the problem of segmenting mixed digital documents into six parts. Then at the sender side, each incorporated part is compressed individually apart from others using the most applicable compression technique, thereby ensuring better compression ratios and thence quicker sending data from one machine to another. By this means, the recipient can integrate these various image components to regenerate the original document. However, this arrangement places an emphasis on direct dialogue between the pair of actors, the sender and the recipient. (El-Omari and Awajan 2009)(El-Omari 2008)(El-Omari et al. 2012)

In order to state a truth, this paper is a continuation of the previous works in the area of document image segmentation and compression (El-Omari and Awajan 2009)(El-Omari 2008)(El-Omari et al. 2017)(El-Omari et al. 2012)(Taha et al. 2012). In order to explore further the arguments set out above, this paper is divided into seven sections. After this section provides an introduction to the main theme of the paper, Section 2 surveys the literature to look at the related work and, moreover, reviews some fundamental concepts and terminology that forms the theoretical background. Section 3 is where the real work begins; it presents the current approach developed in this research and then walks through all the different stages which would be required to implement this proposed algorithm. The segments formulation and the mathematical model of this proposed solution are detailed in Section 4. While the conducted experiments and their detailed intensive analysis are discussed in Section 5, Section 6 concludes the project work of this research. Finally, to accomplish the discussion of this paper, the last section, Section 7, highlights an ample research scope and addresses a fairly broad range of possible research opportunities to be further investigated.

## 2. Related Work

Mixed document segmentation, as a well-known research area, aims for dividing a document image into its components: pictures, graphics (drawings), texts, and backgrounds. Data compression, on the other hand, is the process of rearranging the original information with the sole intention of relatively getting fewer numbers of bits which in turn leads to storage space reduction.

While the algorithms that carry out the data compression process are referred to as encoders, the ones that perform the inverse process to reconstruct the original images are referred to as decoders (Kumar et al. 2019). However, this whole process is referred to as encoding. Figure 2 is a schematic diagram that depicts the data compression and decompression processes for an image having "M" pixels in length and "N" pixels in wide. Imagine an input data file, "R(M,N)", is encoded to be "E(M,N)" and transferred through a network from a source computer to a destination one where the file can be decoded back, i.e. decompressed or retrieved back, to be "D(M,N)".

It is a reality that exaggerative numbers of millions of digital images are being generated every single hour. Not only that, but most of these images are rich-mix contents (El-Omari 2019)(El-Omari and Alzaghal 2017). In order to face the reality of this truth, many models of segmentation and/or compression are currently available; each one has its own specifications and essential requirements. And so, the right decision for a particular model selection is no longer an easy duty to be carried out (El-Omari 2019). Besides, the traditional algorithms may no longer enough sufficient to upkeep the new needs and then there is a vital need for new efficient techniques.

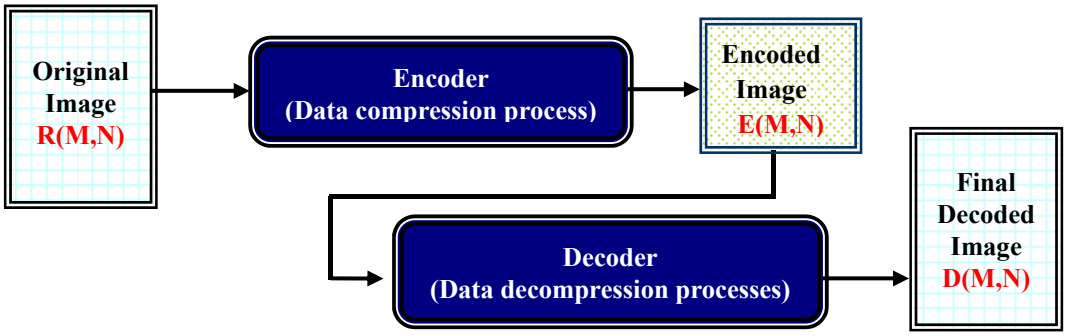


Figure 2. A system block diagram showing the image compression & decompression processes.

To this aim, the goal of this paper is to get the maximum higher transmission rate at which the data can be transferred properly from the source point to the destination. However, this rate is much related to the size of the file which, consecutively, depends on its content types. As such, there is certainly a broad group of techniques proposed in today's growing field of compression which makes it difficult to choose from the most appropriate model selection especially that most of them provide an adequate style to implement. Depending on many relevant aspects, each model has its own specification and, therefore, these techniques can be classified into six overlapped categories that Figure 3 demonstrates:

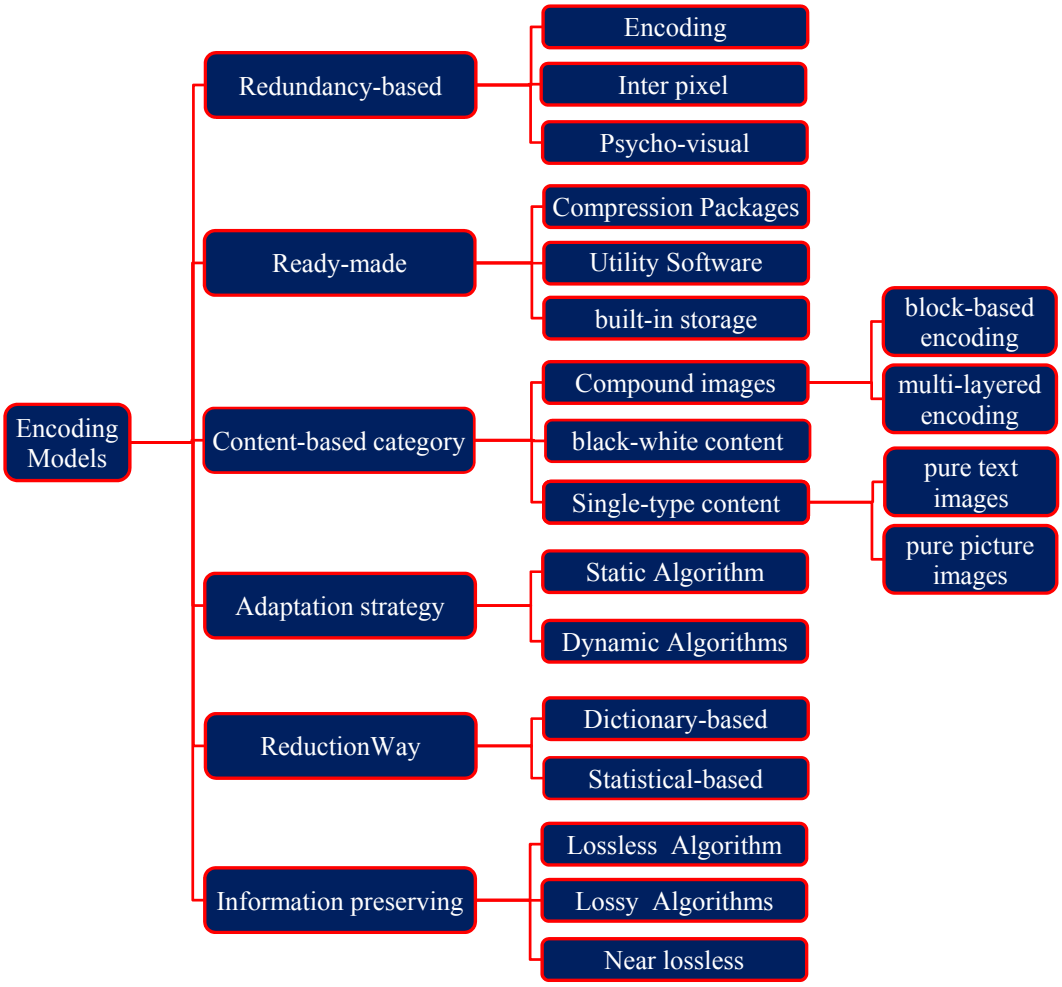


Figure 3. Categories of data compression techniques

- **Redundancy-related categorization:** This category is associated with the way of performing the compression process. This category can be further grouped into three subgroups: Encoding redundancy, Inter-pixel redundancy, and Psycho-visual redundancy. For more information on this category, you can refer to (Taha et al. 2012).

- **Ready-Made categorization**: By considering the strategy of data compression/decompression, a further additional taxonomy can be possible for this group: (Taha et al. 2012)(Gonzalez and Woods 2017)
  - Using off-the-shelf packages that are available on the market for data compression/decompression.
  - Online disk compression: Building the data compression/decompression as a transparent (i.e. real-time) utility inside the operating system. By using this strategy, every data file is directly compressed when it is saved. In contrast, every data file is automatically decompressed when it is retrieved back (i.e. loaded).
  - To speed up operations, the data compression/decompression can be built internally as a special-purpose built-in chip which obviously has its corresponding software driver. Again as stated in the previous subcategory, every file is automatically compressed during the saving process, and vice versa.
- **Adaptation-related categorization**: Upon the adaptation strategy, image compression algorithms are generally categorized into two fields: static and dynamic. Regardless of its content types, the compression process is fixed in the static case and thereby there isn't any attempt to capture any more details about the problem of interest during the process of encoding, hence the term "data-independent". In contrast to the static case, the dynamic encoding processes changes dynamically depending on the extracted data content, hence the term "adaptive compression". (Taha et al. 2012)(Gonzalez and Woods 2017)
- **Information-preserving categorization**: From a classification point of view, "lossless" versus "lossy" is utilized in accordance with the quality requirements. Different than the other one, "lossless" guarantees that the decoded document and the original one are entirely identical and precisely bit-by-bit alike. Therefore, the compression process is reversible. Reasoning from this fact that there may be some image degradation due to the process of discarding away some data forever, "lossy" does not guarantee that they are alike. And in turn, the process is irreversible and the data are an approximated copy of the original.

  From another point of view, "lossless" achieves a lower compression ratio as compared to "lossy". Roughly speaking, the quality and the compression rate run in the opposite direction from each other where lower data quality is much related to a higher compression ratio and versa vice. The degree of quality loss is directly proportional to the compression level being applied to the image. Measured in terms of space-time complexity, the compression reduction level that can be achieved using lossless techniques is lower than that rate of "lossy" techniques and, hence, "lossy" methods typically saves more memory and run-time computations without reporting any distinguishable regression related to the image quality. (Taha et al. 2012)(Gonzalez and Woods 2017)(Boopathiraja, Kalavathi, and Dhanalakshmi 2019)(Kumar et al. 2019)

  As defined for the "lossy" cases and at the expense of getting data storage reduction, some loss of information is reasonable and acceptable by an adequate margin of safety, such as small variation of colors or dropping insignificant detail and inessential characters, whose loss will not be observed or make a big difference. "lossless" compression, by contrast, is the only acceptable mean of data reduction where an exact recovery of an encoded image is vitally essential. Medical images, confidential data, legal and historical documents are the most dominant examples of this norm of compression. (El-Omari 2008)

  In view of Figure 2, if both the reconstructed image, "D(M,N)", and the original one, "R(M,N)", are exactly the same, then the data compression technique is "lossless"; otherwise, it is a "lossy".
- **Content-related categorization**: In the direction of solving the problem of segmenting and compressing compound documents, this group is divided into three subcategories:
  - **Black-white algorithms**: These algorithms, such as "Fax Group 3" and "Fax Group 4", are formally emerged for the purpose of converting the images themselves into black-white color and then encode them through lossless compression algorithms. Even though these algorithms have more storage space reduction, the contrast and the color information are unfortunately lost and, therefore, they are inappropriate for other document types such as medical images, historical documents, or colored magazines. They, on the other hand, are more suitable for some technical and business documents. (El-Omari et al. 2017)(Gonzalez and Woods 2017)
  - **Single-type content**: These algorithms are only designed to encode the documents that have one type of content. Taha et al. (2012) and El-Omari et al. (2017) proposed two special-purpose techniques for compression documents that have only text contents.
  - **Compound images**: Rather than uniformly encoding the entire image as reported in the cases of conventional image compression algorithms, this style of algorithms is used to encode compound images that may contain more than one component, such as pictures and graphics besides texts (Kumar et al. 2019). It is based on

building prior knowledge about the images, then uses this knowledge to divide them into their different content types, and finally encode every type separately aside from the others (Kumar et al. 2019). Two subcategories are forked from these algorithms: Layered encoding and Block-based encoding. Mixed Raster Content (MRC) is one of the most dominant examples of layered encoding. As illustrated in Figure 4, MRC divides the image into three content types: foreground (FG) of 24-bit color, a binary mask of only one bit for each pixel, and background (BG) of 24-bit color. Each identified bit of the binary mask determines that the pixel belongs either to the foreground layer, i.e. has a value of 0, or to the background layer, i.e. has a value of one (Queiroz, Buckley, and Xu 1999)(El-Omari et al. 2017).

Another noteworthy example, El-Omari and Awajan (2009) and El-Omari et al., (2012) utilized the Artificial Neural Network (ANN) to exploit some prior knowledge about the images and then they use this knowledge as a classifier to segment and compress compound images.

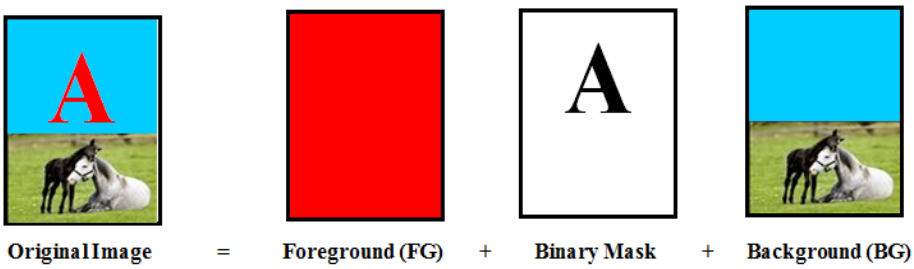


Figure 4. MRC divides images into its relevant components.

Besides the aforesaid categories, there is still much room for improving and investing these existing algorithms or coming up with new effective algorithms and techniques like the one described in this current research.

## 3. The Proposed Framework

The philosophy behind this proposed technique is to store descriptors or pointers that refer to specific references within a special-purpose dictionary rather than storing the actual repeated figures for every pixel, which is its color information. While the repetitive data of these colors are stored only once, this internal dictionary is fabricated specifically for every block/region of the image and it is referred to as Lookup Dictionary Table (LUD). This LUD is organized as key-value pairs: the actual data items being looked up and the reference pointers that point out to where the data are located. So, the LUD reference list should consist of all the reference pointers and the referencing to this dictionary is performed upon coming across any reference pointer

Through the indexing operation, the value of every index pointer should point out to one and only one LUD color item. On the other side, as any reference pointer can point out to exactly one LUD color, any LUD color may be referred by many reference pointers. Because the relevant information is declared and stored in the form of codes, the mapping operation between the values of the index pointers and the corresponding colors of the LUD is guaranteed. Namely, each cited reference has to be cited in advance inside the LUD list and, in turn, there is no reason to include uncited references without they originally exist in the LUD entries. (Azad et al. 2010)(Wikipedia 2016)(El-Omari et al. 2017)

As reflected in Figure 5 and Figure 6, the proposed technique works in a sequence of seven phases. These phases form the roadmap framework of the proposed technique.

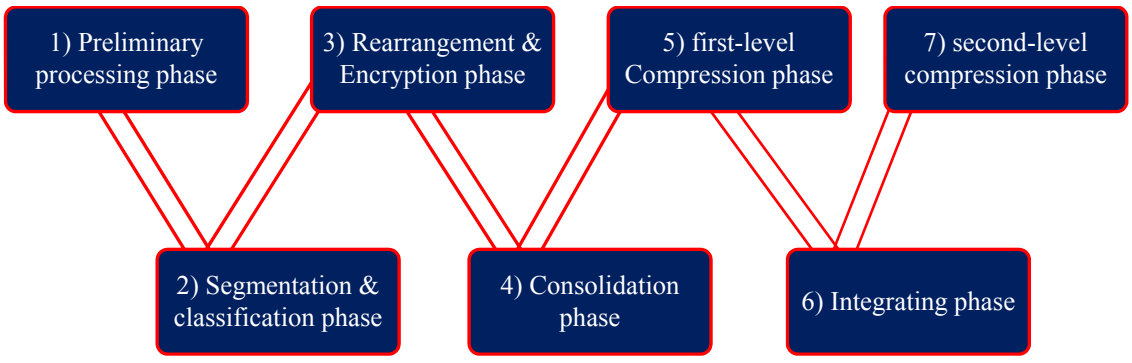


Figure 5. The Roadmap framework of the proposed technique

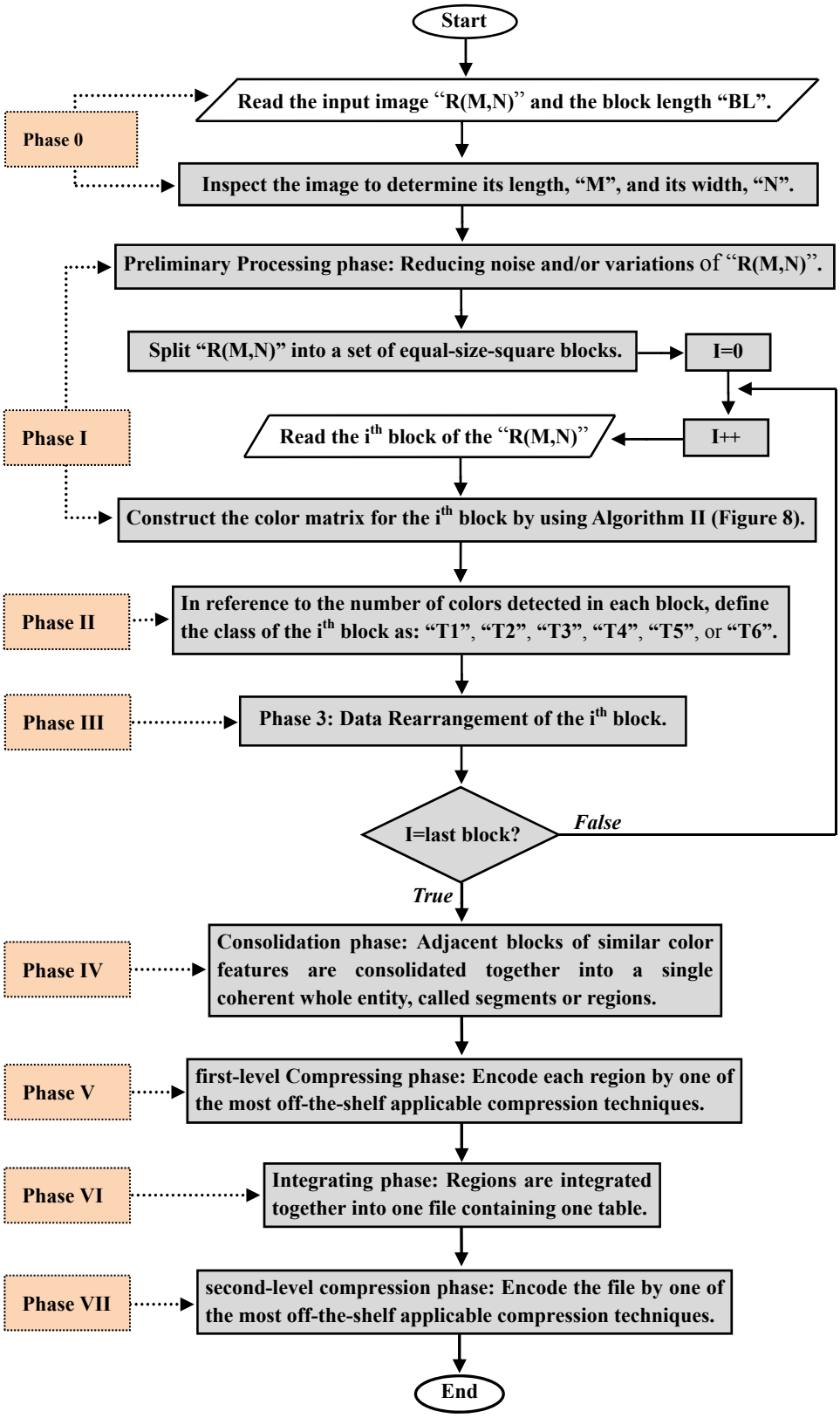


Figure 6. The flowchart of the proposed algorithm

While Algorithm I of Figure 7 represents the main phases included in this proposed Algorithm, Algorithm II that is detailed in Figure 8 is expressed to build the color statistic table (CST).

| **Algorithm I: Color Counts Block-Based Segmentation** | |
|---|---|
| **Description** | This algorithm contains seven phases and six classes in its essence. Through this algorithm, the bitmap table of the original image is divided into six classes based on the number of colors that are detected inside. Then through seven phases, the algorithm builds a new two-level compressed file. |
| **Input** | "R(M,N)" which represents any BMP image of size MxNx3, where "M", "N", and "3" correspond to the image length, width, and the three-RGB-component colors, respectively. |
| **Output** | "E(M,N)" which represents the compressed image file. |

| | **Method** |
|---|---|
| 1. | Initialization: do the followings: |
| 2. | **(i)** Read the input image, "R(M,N)". |
| 3. | **(ii)** Scan the input image, "R(M,N)", to determine the length, "M", and the width, "N". |
| 4. | **(iii)** Read the block length, "**BL**". |
| 5. | Create six empty tables "**T1**", "**T2**", "**T3**", "**T4**", "**T5**", and "**T6**". |
| 6. | Do **preliminary operations**: the required preliminary processing is performed in order to reduce noises and/or variations inside the scanned image. |
| 7. | Divide "R(M,N)" into equal-size-square blocks. |
| 8. | **For** each block, apply the followings: |
| 9. | Using Algorithm II that is detailed in Figure 8, scan the block to build the color statistic table (CST). |
| 10. | Check the color frequencies of the previous table, "CST". Then, colors with low frequency may be considered as noise and eliminated. |
| 11. | Based on identifying the number of colors containing in each block, determine the classes of the blocks as **"T1"**, **"T2"**, **"T3"**, **"T4"**, **"T5"**, or **"T6"**; these classes are outlined in Figure 9. |
| 12. | **If** the block class is of type **"T1"**, insert a new entry in the table **"T1"** that contains the followings: |
| 13. | **(i)** The block length, "**BL**". |
| 14. | **(ii)** Block address: "I" and "J". |
| 15. | **(iii)** The values of the three RGB components of the unique detected color of the block. |
| 16. | **Else If** the block class is of type **"T2"**, i.e. text-based, insert a new entry in the table **"T2"** that contains the following data items: |
| 17. | **(i)** Block address: "I" and "J". |
| 18. | **(ii)** A special-purpose dictionary for the two detected colors. |
| 19. | **(iii)** one-bit-reference-pointer index to designate one of the two colors of the dictionary; using zero for the pixels having the first color and one for the second color. One byte can hold the information of 8 pixels. |
| 20. | **Else If** the block class is of type **"T3"**, insert a new entry in the table **"T3"** with the following data items: |
| 21. | **(i)** Block address: "I" and "J". |
| 22. | **(ii)** A special-purpose 16-color (each color requires three entries) dictionary is built where the detected colors are arranged at first and the remaining unoccupied entries are fulfilled to 16 colors (i.e. 3*16=48 cells) with null values. |
| 23. | **(iii)** four-bit-reference-pointer index to designate a specific color from the sixteen colors of the stored dictionary. Every 2 pixels require one byte to store their indexes. Any reference pointer refers to one of the already detected colors and there isn't any pointer that refers to one of the unoccupied entries (i.e. colors) that are previously fulfilled to complete the number of colors into 16 colors with null values |
| 24. | **Else If** the block class is of type **"T4"**, insert a new entry in the table **"T4"** with the following data: |
| 25. | **(i)** Block address: "I" and "J". |
| 26. | **(ii)** A certain special-purpose 128-color (each color needs three entries) dictionary is built where the detected colors are arranged at first and the remaining unoccupied entries are fulfilled to |

| | |
|---|---|
| | 128 colors (i.e. 3*128=384 cells) with null values. |
| 27. | **(iii)** seven-bit-reference-pointer index to designate a specific color from the 128 colors of the stored dictionary. Every individual pixel requires seven bits to be stored. Any reference pointer refers to one of the previously detected colors and there isn't any pointer that refers to one of the unoccupied entries (i.e. colors) that are already fulfilled to complete the number of colors into 128 colors with null values. |
| 28. | **Else If** the block class is of type **"T5"**, insert a new entry in the table **"T5"** that contains the following data items: |
| 29. | **(i)** Block address: "I" and "J". |
| 30. | **(ii)** For each pixel of the block, store its red color component. Each pixel requires a single byte. |
| 31. | **Else If** the block class is of type **"T6"**, insert a new entry in the table **"T6"** with the following data items: |
| 32. | **(i)** Block address: "I" and "J". |
| 33. | **(ii)** The pixels' data that are detected in that block. Every pixel requires three bytes. |
| 34. | **End If** |
| 35. | **End For-loop** // no more blocks |
| 36. | Invoke **Consolidation**: In order to form higher-level regions, blocks of similar color features are consolidated together into a higher single coherent whole. |
| 37. | Invoke the **first-level compression phase**: Each region is encoded by one of the most off-the-shelf applicable compression techniques. Every region is compressed along with its relevant dictionary. |
| 38. | Invoke **Integration**: integrate all the six tables into one file containing one table. |
| 39. | Invoke the **second-level compression phase**: Again, intending to achieve a better compression ratio, the generated file of the preceding step is going through another stage of compression. |
| 40. | **Return** the generated file "E(M,N)". |

Figure 7. Algorithm I, the proposed technique.

| **Algorithm II: Image Color Statistic Table (CST)** | |
|---|---|
| **Description** | This principal algorithm is designed to build a statistic about the detected colors and their relevant frequencies that are captured inside a given block. This statistic represents the color map or the dictionary of colors. Likewise, this algorithm can be carried out to build a statistic about the detected colors and their frequencies of the whole image. |
| **Input** | Either the whole MxNx3-size BMP image, i.e. "R(M,N)", or one of its blocks. |
| **Output** | A color statistic table "CST" of four columns; three of them correspond to the three basic RGB-color components of each color and the last one corresponds to the frequency of that color. However, every detected color is viewed by one entry. |
| **Method** | |
| 1. | Initialization: construct an empty table "CST" of four columns. |
| 2. | **Read** the input block pixels from left to right and top to bottom. |
| 3. | **For** every pixel of the input the block: |
| 4. | **If** the three basic RGB-color components already exist in "CST" |
| 5. | Add 1 to the frequency that corresponds to that color. |
| 6. | **Else** |
| 7. | Insert this new color in the table "CST" with a frequency equals to one. |
| 8. | **End If** |
| 9. | **End For** |
| 10. | **Return** the color statistic table "CST". |

Figure 8. Algorithm II, generate an Image Color Statistic Table (CST)

*3.1 Phase I: Preliminary Processing Phase*

As data efficiency is crucially important to be improved before using, the original data representation may subject to a set of pre-processing steps that are performed for filtering noise or variations inside the scanned images. What's tricky is that the success of this proposed technique is highly depending upon the thoroughness of this phase. (Kumar et al. 2019)

Going forward, the image with the enhanced quality is then divided into equal-size-square blocks. A color map of each block that represents the detected colors and their frequencies is generated using Algorithm II that is already detailed in Figure 8. This map is referred to as the Color Statistic Table (CST) for these identified colors. Within this context, if the pixels of an input block (I, J) of "BL x BL" in size and its pixels are distributed among "n" three-RGB-component colors, then Table 1 represents the output of this algorithm. Again, like those that are outlined above, colors with low-frequency rates may be considered as noise and excluded from this table.

Table 1. The structure of the Color Statistic Table (CST)

| Red component | Green component | Blue component | Frequency |
|---|---|---|---|
| $R_{001}$ | $G_{001}$ | $B_{001}$ | $F_{001}$ |
| $R_{002}$ | $G_{002}$ | $B_{002}$ | $F_{002}$ |
| $R_{003}$ | $G_{003}$ | $B_{003}$ | $F_{003}$ |
| ⋮⋮⋮ | ⋮⋮⋮ | ⋮⋮⋮ | ⋮⋮⋮ |
| $R_{i-1}$ | $G_{i-1}$ | $B_{i-1}$ | $F_{i-1}$ |
| $R_i$ | $G_i$ | $B_i$ | $F_i$ |
| ⋮⋮⋮ | ⋮⋮⋮ | ⋮⋮⋮ | ⋮⋮⋮ |
| $R_{n-1}$ | $G_{n-1}$ | $B_{n-1}$ | $F_{n-1}$ |
| $R_n$ | $G_n$ | $B_n$ | $F_n$ |
| | | The total frequency | $\mathbf{BL^2}$ |

It is essential to mention that this table is arranged in descending order according to the last column, "Frequency". At the beginning of this algorithm, an empty 4-column table is created. As the image file is read, this table is altered whenever a new color is encountered. If the encountered color already exists in this table, its corresponding frequency is increased by one. Otherwise, a new entry corresponding to this new color is inserted in this table with a frequency equals to one.

Table 2 states an example of this CST where the data is viewed in decimal values. The block of this example is "32x32" pixels in size. The 1024 pixels are distributed among thirteen three-RGB-component colors.

Table 2. An example of the Color Statistic Table (CST)

| | Red component | Green component | Blue component | Frequency |
|---|---|---|---|---|
| 0. | 255 | 244 | 254 | 241 |
| 1. | 000 | 006 | 016 | 218 |
| 2. | 235 | 204 | 122 | 132 |
| 3. | 016 | 008 | 016 | 102 |
| 4. | 007 | 008 | 007 | 95 |
| 5. | 245 | 245 | 225 | 62 |
| 6. | 009 | 026 | 014 | 77 |
| 7. | 240 | 240 | 230 | 20 |
| 8. | 189 | 189 | 189 | 18 |
| 9. | 029 | 001 | 010 | 17 |
| 10. | 218 | 218 | 228 | 16 |
| 11. | 224 | 224 | 224 | 15 |
| 12. | 016 | 014 | 009 | 11 |
| | | | **Total** | **1024** |

*3.2 Phase II: Segmentation and Classification Phase*

After the phase of preliminary operations, each block is assigned a class type based on its CST. All blocks that have the same number of colors are given the same label or class. As reported in Table 3 and shown in the illustration of Figure 9, block types can be categorized into six classes.

For the scanned image, "N", suppose that the numbers of blocks for the classes "**T1**", "**T2**", "**T3**", "**T4**", "**T5**", and "**T6**" are: $N_{T1}$, $N_{T2}$, $N_{T3}$, $N_{T4}$, $N_{T5}$, and $N_{T6}$, respectively. Then, the total number of blocks is defined as shown by Equation 1:

$$N = \sum_{i=1}^{6} N_{Ti} \quad (1)$$

On the other hand, "N" can be entirely decoded back in a reversible way as shown by Equation 2:

$$\sum_{i=1}^{6} N_{Ti} = N \quad (2)$$

Table 3. The description of the six classes of the proposed technique

| Class | No. of Colors | Note |
|---|---|---|
| **"T1"** | **1** | The number of detected colors is one and only one. Generally, this class of blocks represents the background of a document image which is a large expanse of a single color. This color is considered as a background. |
| **"T2"** | **2** | The number of detected colors is exactly two. This class usually represents the text-based data. |
| **"T3"** | **3-16** | The number of detected colors is less than 17 and more than two. This class generally represents the drawing parts of the documents: graphs, charts, and/or curves. |
| **"T4"** | **17-128** | The number of detected colors is less than 129 and more than 16. |
| **"T5"** | **129-256** | The number of detected colors is less than 257 and more than 128. These blocks are mainly the grey part of the image. |
| **"T6"** | **>256** | The blocks of this class generally represent the millions of color pictures found in the images. |

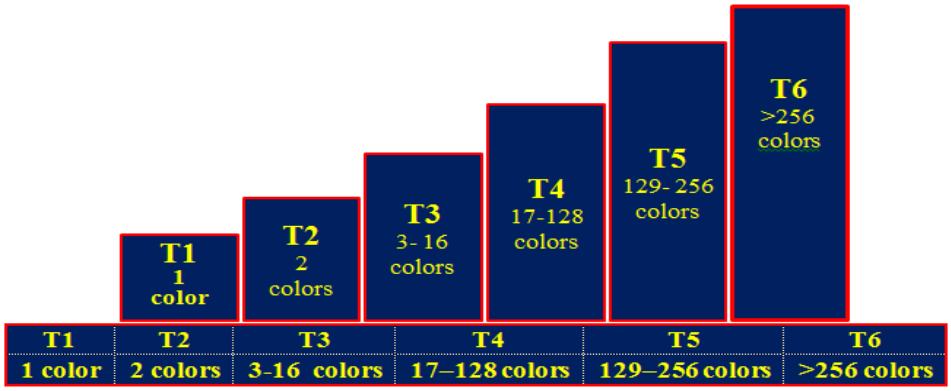


Figure 9. The six classes of the proposed technique

*3.3 Phase III: Rearrangement Phase*

This phase is based on forming newly generated data of each block. The output of this phase is a table where each entry contents vary according to the assigned block class. To explore further details, this phase will be detailed in the next section (specifically, subsections 4. 1 through 4.6).

*3.4 Phase IV: Consolidation Phase*

In order to form higher-level regions (i.e. sub-images), this phase aims at combining together the adjacent neighboring equal-class blocks that have the same dictionary of colors into a larger arrangement of contiguous blocks. It is important to realize that the blocks that have the same class don't essentially have the same colors (i.e. dictionary), but they may have the same number of colors (El-Omari et al. 2017)(Kumar et al. 2019). As

shown in the illustration of Figure 10, adjacent neighbors of a given block can be defined as either four-connectivity, in which the two blocks share a common side, or eight-connectivity, in which the two blocks share either a common side or a common corner.

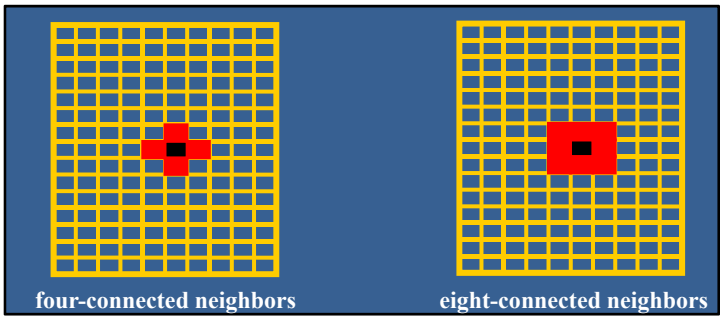


Figure 10. four-Connected & eight-Connected neighbor Blocks (El-Omari et al. 2017)

*3.5 Phase V: First-level Compression Phase*

The data compression process will be carried on two levels. While this phase includes the first level, the second one will be in the last phase, Phase VII. Here in this phase, every current region (i.e. sub-images or blocks of similar features) is compressed separately with the most off-the-shelf appropriate compression technique. It is worth mentioning that every region is compressed along with its corresponding dictionary.

*3.6 Phase VI: Integration Phase*

Although they are developed separately, all the six tables are eventually fused together into one entity using the block address, I and j. This incorporated entity is formed as a single data file containing one table that interlinks between the six close-related classes.

*3.7 Phase VII: Second-level Compression Phase*

This is the final phase; again, like the fifth phase outlined above, this phase is carried out with the intention of achieving a better data compression ratio. Thus, the generated data file of the preceding step is going through another noteworthy level of compression.

**4. Solution Formulation & Mathematical Model**

Related to its critical importance, this section goes through detailing the six data classes that are abstracted in the preceding section, specifically subsection (3.3). Before reporting this section, it is worth mentioning that the first four classes, "**T1**" through "**T4**" are built upon the idea of using special dictionaries and pointers for encoding data, each dictionary, called Lookup Dictionary Table (LUD), is designed for the corresponding class type. However, when the computer at the receiver (i.e. decoder) side and through the inverse decompression (i.e. decoding or retrieved back) process read the encoded compressed file and encounters a pointer, it interprets that pointer by retrieving the corresponding color from its place in the dictionary index; hence the original image part is reconstructed and retrieved up to the last bit.

In order to evaluate the overall performances of the proposed technique, a mathematical measure "Saving Ratio Percentage (SRP)" is calculated to compare the size of the original image with the final encoded image; it can be calculated mathematically as expressed by Equation 3: (El-Omari et al. 2017)(Azad et al. 2010)(Kumar et al. 2019)

$$\mathrm{SRP} = \left[1 - \frac{\text{size of the encoded image (per bytes)}}{\text{size of the original image (per bytes)}}\right] * 100\%$$

$$= \left[1 - \frac{\mathrm{E(M,N)}}{\mathrm{R(M,N)}}\right] * 100\% = \left[1 - \frac{\mathrm{E(M,N)}}{3 * \mathrm{BL}^2}\right] * 100\% \qquad (3)$$

Where "R(M,N)" and "E(M,N)" as already stated in Figure 2.

Being more specific, some references referred to the term "E(M,N)/R(M,N)" as the compression ratio (CR) or the relative data redundancy (Gonzalez et al. 2009)(Kumar et al. 2019). It is important to note that when no data compression is achieved, SRP will be equal to zero. There's no doubt that this measure depends on the image content that leads to the distribution of the original table upon the six classes. Moreover, the proper size of the

block length has some impact on the SRP measure which will be shown at the evaluation of the experimental results, namely Section 5.

*4.1 Class "T1" (One Color)*

This class means that the entire block is a background containing one color. Since class "**T1**" blocks have only one color, the dictionary contains only three cells, one for every basic color component of the single RGB color. Rather than saving the same information for every individual pixel that makes up the background, this approach stores the color data for the background color only once to refer to all pixels of that block. Figure 11 demonstrates how the data can be fabricated for this class by only using a dictionary of one three-RGB-color-component entry.

| | | |
|---|---|---|
| Length Part | The Block length, "**BL**" | Block length (1 byte) |
| Address part | **I value** | Block address (2 bytes) |
| | **J value** | |
| Data Part: just 3 bytes are enough. Neither the special-purpose dictionary (LUD) nor the reference pointers are essential here. | The Red component of the single color | Foreground (FG) color (3 bytes) |
| | The Green component of the single color | |
| | The Blue component of the single color | |

Figure 11. Encoding of a block using a class of type "**T1**"

For this class, each block is represented by its address (I, J), and the three RGB components of its sole color. Since the image has equal-size-square blocks, there is a need to store an additional one-byte cell to represent the block length, "BL". However, this byte is only stored once in this class of blocks to represent all blocks of the image. Moreover, since the blocks of this class have a single color, which is classified as background, there is no need to store more data about the pixels contained in the block. Despite that the special dictionary (i.e. LUD) is essential in this class, the reference pointers are not. Simply, only six bytes are required to store the whole block no matter how much its size. However, this solves one of the drawbacks of layered encoding mentioned in Section 2 which is related to storage space reduction. The SRP per block of this class is modeled mathematically by Equation 4:

$$\text{SRP}(\text{“}\mathbf{T1}\text{”}) = \left[1 - \frac{1+2+3}{3 * \text{BL}^2}\right] * 100\% = \left[1 - \frac{6}{3 * \text{BL}^2}\right] * 100\% \quad (4)$$

The following points analyze the elements of this equation:

- Number "1" of the numerator indicates that only one byte is required to store the "BL".
- Number "2" of the numerator means that two bytes are required to store the address of each block, one byte for the "$I^{th}$" address and one byte for the "$J^{th}$" address.
- Number "3" in the numerator means that there is a necessity for three bytes to store the three basic RGB components of the unique color.
- "BL" stands for the block length and is given in pixels.
- Since the image is divided into equal-size-square blocks, the size of each block is "$BL^2$". Number "3" in the denominator indicates that there are three basic RGB-color components and, therefore, each pixel of "R(M,N)" occupies three bytes. Thus, the denominator stands for the size of the original block before the compression process.
- Based on the aforementioned points, the numerator indicates the size of the compressed block of this type and size.

The following example will clarify how this class is stored. Suppose there is a square block (I, J) = (31, 16) of size (20 x 20=400) pixels and has the following CST (decimal data):

$$(R_{001}, G_{001}, B_{001}, F_{001}) = (254, 019, 028, 400)$$

Since there is only one color in this block, it is identified as a class of type "**T1**". In view of that, this block will be represented neither with reference pointers nor with LUD. Table 4 illustrates the data schematic construction of this example where only six bytes are required. Based on the above-mentioned discussion, the researcher can conclude that:

Rather than using **1200** (i.e. 20 * 20 * 3) bytes to store this block of (20x20) pixels in size, six bytes are enough. To rephrase this outcome:

**6 / (20 * 20 * 3) * 100 % = 0.005%**

This means that the proposed technique needs only 0.005% of the block size. If Equation 4 is recalculated by using BL=20, the same result will be achieved which means that the proposed algorithm is an efficient alternative for this class of blocks.

Table 4. A data structure example for the blocks of type "**T1**"

| Data Type | | Data in Decimal | Byte Sequence |
|---|---|---|---|
| Block length | BL | $(031)_{10}$ | 1 |
| Block address | I | $(016)_{10}$ | 2 |
| | J | $(007)_{10}$ | 3 |
| Dictionary Part (LUD)<br>A dictionary of one three-RGB-color-component entry. The reference pointers are not required here. | $R_{001}$ | $(254)_{10}$ | 4 |
| | $G_{001}$ | $(019)_{10}$ | 5 |
| | $B_{001}$ | $(028)_{10}$ | **6** |

*4.2 Class "T2" (a single pair of colors)*

It is worth remarking that this class depends on storing the detected colors of each block inside a dedicated two-entry dictionary constructed specifically for that block. Then, rather than storing the corresponding color out of the two detected colors for every pixel inside the block, the reference pointer indexes are used instead. For this well-defined reason, a one-bit-reference pointer is used as an indication to determine the corresponding LUD color.

Since class "**T2**" has two colors, the dictionary contains six cells, one for every basic color component of each RGB of the two colors. These blocks are represented by the address (I, J) of each block, the 2-color dictionary, called background and foreground colors, and only one bit for every pixel to indicate whether it can be assigned to the background color and assigned zero or the foreground color and assigned one. Figure 12 shows the data structure representation of this class of blocks.

| | | |
|---|---|---|
| Address part | **I value** | Block address (2 bytes) |
| | **J value** | |
| Dictionary Part (LUD)<br>A special-purpose dictionary of two three-RGB-color-component entries, i.e. 2 * 3 = 6 cells. | The Red component of the 1st color | Foreground (FG) color (3 bytes) |
| | The Green component of the 1st color | |
| | The Blue component of the 1st color | |
| | The Red component of the 2nd color | Background (BG) color (3 bytes) |
| | The Green component of the 2nd color | |
| | The Blue component of the 2nd color | |
| Data Part<br>Using one-bit-reference pointers (i.e. every eight pixels need only one byte to store their references). Each reference pointer should certainly point out to either one of the two colors inside that dictionary. | Pixels 08-01 | Pixels' Data |
| | Pixels 16-09 | |
| | Pixels 24-17 | |
| | Pixels 32-25 | |
| | ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ | |
| | The remaining representation of the pixels (eight pixels per each byte) | |

Figure 12. Encoding of a block using a class of type "**T2**"

For the blocks of this class, "**T2**", the data compression is done by storing the reference pointers that point out to the special dictionary. The SRP per block is mathematically expressed by Equation 5:

$$\mathrm{SRP}(\text{“}\mathbf{T2}\text{”}) = \left[1 - \frac{2 + 2 * 3 + \frac{\mathrm{BL}^2}{8}}{3 * \mathrm{BL}^2}\right] * 100\% \qquad (5)$$

This equation is different than Equation 4 by the following points:

- Address part: The first number "2" of the numerator indicates that two bytes are required to store the address of each block, one byte for the $I^{th}$ address and one byte for the $J^{th}$ address.
- Dictionary part (LUD): Since there are two identified colors and each of them has three basic RGB color components, the expression "2*3=6" of the numerator stands for the number of bytes required to store the LUD.
- Data part: Since there are only two colors in this class type, each bit can hold either zero or one to point out to the foreground (FG) or to the background (BG), respectively. Thus, the number of pixels that can be indicated by a single byte is eight.

For the sake of simplicity, Equation 5 can be redrafted as expressed in Equation 6:

$$\mathrm{SRP}(\text{“}\mathbf{T2}\text{”}) = \left[1 - \frac{8 + \frac{\mathbf{BL^2}}{8}}{3 * \mathrm{BL}^2}\right] * 100\% \qquad (6)$$

As an example of this class type, assume that the square block (4, 8) of (30x30=900) pixels in size has the two-color CST that is viewed in Table 5 which has been attained as an output of Algorithm II:

Table 5. A CST example of two colors (decimal data)

| | Red component | Green component | Blue component | Frequency |
|---|---|---|---|---|
| 0. | 254 | 054 | 253 | 544 |
| 1. | 000 | 002 | 092 | 356 |
| | | | Total | **900** |

Note that the total number of frequencies of this block are ($F_{001} + F_{002} = 544 + 356 = 900$). Then, these two colors are assigned the numbers $(0)_2$, $(1)_2$, respectively. Suppose that the first sixteen pixels of this block contain the following RGB-color components:

| | | | | |
|---|---|---|---|---|
| $(254,054,253)_{10}$ | $(000,002,092)_{10}$ | $(000,002,092)_{10}$ | $(254,054,253)_{10}$ | $(000,002,092)_{10}$ |
| $(254,054,253)_{10}$ | $(254,054,253)_{10}$ | $(000,002,092)_{10}$ | $(000,002,092)_{10}$ | $(000,002,092)_{10}$ |
| $(254,054,253)_{10}$ | $(254,054,253)_{10}$ | $(254,054,253)_{10}$ | $(254,054,253)_{10}$ | $(000,002,092)_{10}$ |
| $(000,002,092)_{10}$ | | | | |

Based on the number of detected colors (i.e. a single pair), the type of the block involved in this example is "**T2**" and, in turn, the corresponding reference pointers for these sixteen pixels are presented in Table 6. On the other side, Table 7 illustrates the data schematic construction of this example where a total of 121 bytes are required for each block of this type and size. For the remaining 884 pixels, other than these sixteen pixels, the same pattern is used.

Table 6. The corresponding reference pointers of the example on the class type "**T2**"

| Pixel no. | Pixel data | one-bit-reference-pointer | Eight references are stored in one byte | The decimal equivalent |
|---|---|---|---|---|
| 01 | $( 254,054,253 )_{10}$ | $( 0 )_2$ | $( 1001\ 0110 )_2$ | $( 150 )_{10}$ |
| 02 | $( 000,002,092 )_{10}$ | $( 1 )_2$ | | |
| 03 | $( 000,002,092 )_{10}$ | $( 1 )_2$ | | |
| 04 | $( 254,054,253 )_{10}$ | $( 0 )_2$ | | |
| 05 | $( 000,002,092 )_{10}$ | $( 1 )_2$ | | |
| 06 | $( 254,054,253 )_{10}$ | $( 0 )_2$ | | |
| 07 | $( 254,054,253 )_{10}$ | $( 0 )_2$ | | |
| 08 | $( 000,002,092 )_{10}$ | $( 1 )_2$ | | |
| 09 | $( 000,002,092 )_{10}$ | $( 1 )_2$ | $( 11000011 )_2$ | $( 195 )_{10}$ |
| 10 | $( 000,002,092 )_{10}$ | $( 1 )_2$ | | |
| 11 | $( 254,054,253 )_{10}$ | $( 0 )_2$ | | |
| 12 | $( 254,054,253 )_{10}$ | $( 0 )_2$ | | |
| 13 | $( 254,054,253 )_{10}$ | $( 0 )_2$ | | |
| 14 | $( 254,054,253 )_{10}$ | $( 0 )_2$ | | |
| 15 | $( 000,002,092 )_{10}$ | $( 1 )_2$ | | |
| 16 | $( 000,002,092 )_{10}$ | $( 1 )_2$ | | |

With regard to the aforementioned discussion, the researcher concludes that**:**

Instead of using 2700 (i.e. 30 * 30 * 3) bytes to store this block of (30x30) pixels in size, only 121 bytes are enough. Otherwise speaking:

**121 / (30 * 30 * 3) * 100% = 4.481%**

This means that the proposed technique has the capability to encode this class of blocks by using only **4.481%** of the block size. If Equation 6 is recalculated by using BL=30 and then the result is compared with this outcome, they are the same which means that the proposed algorithm is an efficient alternative for this class of blocks.

Table 7. A data structure example for the blocks of type "**T2**"

| **Data Type** | | | **Data in Decimal** | **Byte Sequence** |
|---|---|---|---|---|
| Block address | | I | $(016)_{10}$ | 1 |
| | | J | $(008)_{10}$ | 2 |
| Dictionary Part (LUD) A special-purpose dictionary of two three-RGB-color-component entries. | 1st color | $R_{001}$ | $(254)_{10}$ | 3 |
| | | $G_{001}$ | $(054)_{10}$ | 4 |
| | | $B_{001}$ | $(253)_{10}$ | 5 |
| | 2nd color | $R_{002}$ | $(000)_{10}$ | 6 |
| | | $G_{002}$ | $(002)_{10}$ | 7 |
| | | $B_{002}$ | $(092)_{10}$ | 8 |
| Data Part: a one-bit-reference pointer for each pixel of the block. Just the first 16 pixels of the block are shown here. | 1st byte = ( $P_{08}$, $P_{07}$, $P_{06}$, $P_{05}$, $P_{04}$, $P_{03}$, $P_{02}$, $P_{01}$ ) | | $(105)_{10}$ | 9 |
| | 2nd byte = ( $P_{16}$, $P_{15}$, $P_{14}$, $P_{13}$, $P_{12}$, $P_{11}$, $P_{10}$, $P_{09}$ ) | | $(195)_{10}$ | 10 |
| The remaining 884 pixels | ⋮ | | ⋮ | |
| | 121st byte= ( $P_{900}$, $P_{899}$, $P_{898}$, $P_{897}$, $P_{896}$, $P_{895}$, $P_{894}$, $P_{893}$ ) | | …… | **900/8+8=121** |

*4.3 Class "**T3**" (3-16 Colors)*

This class depends on storing the detected colors of each block inside a particular 16-color dictionary dedicated particularly to that block. Then, rather than storing the corresponding color out of the sixteen ones for every pixel inside the block, the reference pointer indexes are used instead. While these reference pointers are typically implemented through using LUD, each four-bit-reference pointer is used as an indication to determine the corresponding LUD color.

Related to this special dictionary and as aforementioned in Algorithm II of Figure 8, this special 16-color dictionary is built where the same detected colors are arranged at first and the remaining unoccupied entries of colors are fulfilled to 16 colors with null values where each color requires three-null values. Clearly, each four-bit-reference pointer should point out to one of the previously detected colors and no reference pointer should point out to one of the null entries (i.e. colors) that are originally unoccupied and fulfilled to sixteen colors with null values.

In line with Figure 13, the data representation of this class, "**T3**", is similar to that of class "**T2**". However, the LUD of this class has 16 * 3 = 48 cells. Each block is represented by the pair (I, J), the 16-color dictionary, and a four-bit reference pointer for every pixel to designate a specific color from the identified sixteen colors of the dictionary. Hence, the value $(0000)_2$ points out to the first color in the dictionary, the value $(0001)_2$ points out to the second color, the value $(0010)_2$ points out to the third color, and so on up to the value $(1111)_2$, which is corresponding to $(15)_{10}$, that points out to the last color.

For the blocks of class **"T3"**, the data compression process is implemented by storing the four-bit-reference pointers that point out to the special-purpose dictionary. Therefore, Equation 7 is proposed in this regard:

$$\text{SRP}(\text{“}\mathbf{T3}\text{”}) = \left[1 - \frac{2 + 3 * 16 + \frac{\text{BL}^2}{2}}{3 * \text{BL}^2}\right] * 100\% = \left[1 - \frac{50 + \frac{\text{BL}^2}{2}}{3 * \text{BL}^2}\right] * 100\% \quad (7)$$

Where the following points clarify this equation:

- Dictionary part (LUD): Since there are sixteen colors and each of them has three basic RGB color components, "3*16=48" stands for the number of bytes requested to store the LUD.
- Data part: As there is a maximum of sixteen RGB colors and each of them required four bits to be coded, the number of pixels that can be stored in a single byte is (8 / 4 = 2). Hence, the expression ($BL^2$/ 2) is used to determine the number of bytes that are required to store the four-bit-reference pointers of each block.
- The rest of this equation is similar to Equation 4.

| Address part | I value<br>J value | Block Address<br>(2 bytes) |
|---|---|---|
| Dictionary Part (LUD)<br>A special-purpose dictionary of 16 three-RGB-color-component entries (i.e. 16 * 3 = 48 cells). | The Red component of the 1st color<br>The Green component of the 1st color<br>The Blue component of the 1st color | 1st color<br>(3 bytes) |
| | The Red component of the 2nd color<br>The Green component of the 2nd color<br>The Blue component of the 2nd color | 2nd color<br>(3 bytes) |
| | ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ | |
| | The Red component of the 16th color<br>The Green component of the 16th color<br>The Blue component of the 16th color | 16th color<br>(3 bytes) |
| Data Part<br>Using four-bit-reference pointers (i.e. every two pixels need only one byte to store their references). Each reference pointer should definitely refer to one and only one of the sixteen related colors (i.e. entries) of the dictionary. | Pixel 002 \| Pixel 001<br>Pixel 004 \| Pixel 003<br>Pixel 006 \| Pixel 005<br>⋮ ⋮ ⋮ ⋮ ⋮ ⋮<br>The remaining representation of the pixels, using four-bit-reference pointers to point out to the sixteen dictionary entries; two pointers are stored per each byte. | Pixels' Data |

Figure 13. Encoding of a block using a class of type **"T3"**

To explain the compression process of this class type in a simple way, the following example clarifies how this class is fabricated. Suppose that the square block (I, J) = (1, 12) of (30 x 30=900) pixels in size has a nine-color CST that is viewed in Table 8. This CST has been achieved as an output of Algorithm II of Figure 8.

So the first color takes the number $(0000)_2$ in the LUD, the second color takes $(0001)_2$, the third color takes $(0010)_2$, and the last color takes $(0011)_2$ . In order to complete the LUD entries, the remaining unoccupied (i.e. unfilled) entries of colors, from the 10th color to the 16th color, are fulfilled with null values and, obviously, there isn't any reference pointer that points out to one of them. Suppose that the first twenty-two pixels of this block contain the following three-RGB-color components:

| | | | | |
|---|---|---|---|---|
| $(000,002,092)_{10}$ | $(000,000,000)_{10}$ | $(254,054,253)_{10}$ | $(000,002,092)_{10}$ | $(000,000,000)_{10}$ |
| $(000,002,092)_{10}$ | $(000,002,092)_{10}$ | $(255,255,254)_{10}$ | $(120,100,199)_{10}$ | $(120,100,199)_{10}$ |
| $(000,002,092)_{10}$ | $(000,002,092)_{10}$ | $(000,002,092)_{10}$ | $(000,002,092)_{10}$ | $(120,104,196)_{10}$ |
| $(120,104,196)_{10}$ | $(120,104,196)_{10}$ | $(118,106,191)_{10}$ | $(118,106,191)_{10}$ | $(118,106,191)_{10}$ |
| $(120,104,196)_{10}$ | $(120,104,196)_{10}$ | | | |

Based on the number of colors detected in this block, this technique treats this block as a class of type "**T3**". Accordingly, the corresponding reference pointers for these twenty-two pixels are described in Table 9. In this regard, Table 10 illustrates the data schematic construction of this example where a total of 500 bytes are required for each block of this type and size. For the remaining 878 pixels, other than these twenty-two pixels, the same pattern is used.

Based on the aforementioned discussion, the researcher concludes that**:**

Rather than consuming 2700 (i.e. 30 * 30 *3) bytes to store this block of (30x30) pixels in size, only 500 bytes are enough. To rephrase this outcome:

**500 / (30 * 30 * 3) * 100% = 18.519%**

Another time, this means that the proposed technique is capable to encode this class of blocks by using only **18.519%** of the block size. If Equation 7 is recalculated by using BL=30, the same result will be achieved which proves that the outcome is consistent with the research findings. And so, this proposed algorithm is an efficient alternative for this class of blocks.

Table 8. A CST example of nine colors (data in decimal)

| | Red component | Green component | Blue component | Frequency |
|---|---|---|---|---|
| 0. | 255 | 255 | 254 | 191 |
| 1. | 254 | 054 | 253 | 190 |
| 2. | 000 | 000 | 000 | 190 |
| 3. | 000 | 002 | 092 | 189 |
| 4. | 120 | 100 | 199 | 050 |
| 5. | 120 | 105 | 193 | 040 |
| 6. | 118 | 106 | 191 | 020 |
| 7. | 119 | 105 | 200 | 017 |
| 8. | 120 | 104 | 196 | 013 |
| | | | **Total** | **900** |

Table 9. The corresponding reference pointers of the example on the class type "**T3**"

| **Pixel no.** | **Pixel data** | **Decimal LUD reference** | **four-bit-reference pointer** | **Eight references are stored in one byte** | **The decimal equivalent** |
|---|---|---|---|---|---|
| 01 | $( 000,002,092 )_{10}$ | $( 03 )_{10}$ | $( 0011 )_2$ | $( 0010\ 0011 )_2$ | $( 035 )_{10}$ |
| 02 | $( 000,000,000 )_{10}$ | $( 02 )_{10}$ | $( 0010 )_2$ | | |
| 03 | $( 254,054,253 )_{10}$ | $( 01 )_{10}$ | $( 0001 )_2$ | $( 0011\ 0001 )_2$ | $( 049 )_{10}$ |
| 04 | $( 000,002,092 )_{10}$ | $( 03 )_{10}$ | $( 0011 )_2$ | | |
| 05 | $( 000,000,000 )_{10}$ | $( 02 )_{10}$ | $( 0010 )_2$ | $( 0011\ 0010 )_2$ | $( 050 )_{10}$ |
| 06 | $( 000,002,092 )_{10}$ | $( 03 )_{10}$ | $( 0011 )_2$ | | |
| 07 | $( 000,002,092 )_{10}$ | $( 03 )_{10}$ | $( 0011 )_2$ | $( 0000\ 0011 )_2$ | $( 003 )_{10}$ |
| 08 | $( 255,255,254 )_{10}$ | $( 00 )_{10}$ | $( 0000 )_2$ | | |
| 09 | $( 120,100,199 )_{10}$ | $( 04 )_{10}$ | $( 0100 )_2$ | $( 0100\ 0100 )_2$ | $( 068 )_{10}$ |
| 10 | $( 120,100,199 )_{10}$ | $( 04 )_{10}$ | $( 0100 )_2$ | | |
| 11 | $( 000,002,092 )_{10}$ | $( 03 )_{10}$ | $( 0011 )_2$ | $( 0011\ 0011 )_2$ | $( 051 )_{10}$ |
| 12 | $( 000,002,092 )_{10}$ | $( 03 )_{10}$ | $( 0011 )_2$ | | |
| 13 | $( 000,002,092 )_{10}$ | $( 03 )_{10}$ | $( 0011 )_2$ | $( 0011\ 0011 )_2$ | $( 051 )_{10}$ |
| 14 | $( 000,002,092 )_{10}$ | $( 03 )_{10}$ | $( 0011 )_2$ | | |
| 15 | $( 120,104,196 )_{10}$ | $( 08 )_{10}$ | $( 1000 )_2$ | $( 1000\ 1000 )_2$ | $( 136 )_{10}$ |
| 16 | $( 120,104,196 )_{10}$ | $( 08 )_{10}$ | $( 1000 )_2$ | | |
| 17 | $( 120,104,196 )_{10}$ | $( 08 )_{10}$ | $( 1000 )_2$ | $( 0110\ 1000 )_2$ | $( 104 )_{10}$ |
| 18 | $( 118,106,191 )_{10}$ | $( 06 )_{10}$ | $( 0110 )_2$ | | |
| 19 | $( 118,106,191 )_{10}$ | $( 06 )_{10}$ | $( 0110 )_2$ | $( 0110\ 0110 )_2$ | $( 102 )_{10}$ |
| 20 | $( 118,106,191 )_{10}$ | $( 06 )_{10}$ | $( 0110 )_2$ | | |
| 21 | $( 120,104,196 )_{10}$ | $( 08 )_{10}$ | $( 1000 )_2$ | $( 1000\ 1000 )_2$ | $( 136 )_{10}$ |
| 22 | $( 120,104,196 )_{10}$ | $( 08 )_{10}$ | $( 1000 )_2$ | | |

Table 10. A data structure example for the blocks of type "**T3**"

| **Data Type** | | | **Data in Decimal** | **Byte Sequence** |
|---|---|---|---|---|
| Block address | | I | $(001)_{10}$ | 01 |
| | | J | $(012)_{10}$ | 02 |
| | 1st color | $R_{001}$ | $(255)_{10}$ | 03 |
| | | $G_{001}$ | $(255)_{10}$ | 04 |
| | | $B_{001}$ | $(254)_{10}$ | 05 |
| | 2nd color | $R_{002}$ | $(254)_{10}$ | 06 |
| | | $G_{002}$ | $(054)_{10}$ | 07 |
| | | $B_{002}$ | $(253)_{10}$ | 08 |
| | 3rd color | $R_{003}$ | $(000)_{10}$ | 09 |
| | | $G_{003}$ | $(000)_{10}$ | 10 |
| | | $B_{003}$ | $(000)_{10}$ | 11 |
| | 4th color | $R_{004}$ | $(000)_{10}$ | 12 |
| | | $G_{004}$ | $(002)_{10}$ | 13 |
| | | $B_{004}$ | $(092)_{10}$ | 14 |
| | 5th color | $R_{005}$ | $(120)_{10}$ | 15 |
| | | $G_{005}$ | $(100)_{10}$ | 16 |
| | | $B_{005}$ | $(199)_{10}$ | 17 |
| | 6th color | $R_{006}$ | $(120)_{10}$ | 18 |
| | | $G_{006}$ | $(105)_{10}$ | 19 |
| | | $B_{006}$ | $(193)_{10}$ | 20 |
| | 7th color | $R_{007}$ | $(118)_{10}$ | 21 |
| | | $G_{007}$ | $(106)_{10}$ | 22 |
| | | $B_{007}$ | $(191)_{10}$ | 23 |
| | 8th color | $R_{008}$ | $(119)_{10}$ | 24 |
| | | $G_{008}$ | $(105)_{10}$ | 25 |
| | | $B_{008}$ | $(200)_{10}$ | 26 |
| | 9th color | $R_{009}$ | $(120)_{10}$ | 27 |
| | | $G_{009}$ | $(104)_{10}$ | 28 |
| | | $B_{009}$ | $(196)_{10}$ | 29 |
| | ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ The colors from 10 to 16 are fulfilled with null values. ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ | | | |
| | 16th color | $R_{016}$ | null | 49 |
| | | $G_{016}$ | null | 50 |
| | | $B_{016}$ | null | 48+2=50 |
| Data Part<br>A four-bit-reference pointer for each pixel of the block. Just the first ten pixels of this block are shown here. | 1st byte = ( $P_2$, $P_1$ ) | | $(035)_{10}$ | 51 |
| | 2nd byte = ( $P_4$, $P_3$ ) | | $(049)_{10}$ | 52 |
| | 3rd byte = ( $P_6$, $P_5$ ) | | $(050)_{10}$ | 53 |
| | 4th byte = ( $P_8$, $P_7$ ) | | $(003)_{10}$ | 54 |
| | 5th byte = ( $P_{10}$, $P_9$ ) | | $(068)_{10}$ | 55 |
| | 6th byte = ( $P_{12}$, $P_{11}$ ) | | $(051)_{10}$ | 56 |
| | 7th byte = ( $P_{14}$, $P_{13}$ ) | | $(051)_{10}$ | 57 |
| | 8th byte = ( $P_{16}$, $P_{15}$ ) | | $(136)_{10}$ | 58 |
| | 9th byte = ( $P_{18}$, $P_{17}$ ) | | $(104)_{10}$ | 59 |
| | 10th byte = ( $P_{20}$, $P_{19}$ ) | | $(102)_{10}$ | 60 |
| | 11th byte = ( $P_{22}$, $P_{21}$ ) | | $(136)_{10}$ | 61 |
| The remaining 878 pixels | ⋮ ⋮ ⋮ | | ⋮ ⋮ ⋮ | ⋮ ⋮ ⋮ |
| | ⋮ ⋮ ⋮ | | ⋮ ⋮ ⋮ | ⋮ ⋮ ⋮ |
| | 450th byte = ( $P_{900}$, $P_{899}$ ) | | **……** | **450+50=500** |

*4.4 Class "**T4**" (17-128 Colors)*

Over again, this class is based on storing the detected colors of each block inside a dedicated 128-color dictionary constructed dedicatedly for each block of that type. Then, instead of storing the corresponding color out of the 128 ones for every pixel inside the block, the reference pointer indexes are used instead. While these reference pointers are typically implemented through using LUD, each seven-bit-reference pointer is used as an indication to determine the corresponding LUD color out of the 128 ones.

Related to this special dictionary and as aforementioned in Algorithm II of Figure 8, this special-purpose 128-color dictionary is built and the detected colors are arranged at first and the remaining unoccupied entries are fulfilled to 128 colors with null values where each color needs three-null values. Any reference pointer points out to one of the already detected colors and, surely, each pointer refers to one of the actual detected colors and there isn't any pointer points out to one of the null entries (i.e. colors) that are originally unoccupied and fulfilled to 128 colors with null values.

Figure 14 illustrates the data structure representation of this class of blocks. Accordingly, the dictionary concept of this class, "**T4**", is similar to that of "**T2**" and "**T3**". Conversely, the dictionary of this class has (128 * 3 = 384) entries (i.e. 384 bytes). Each block is represented by the pair (I, J), the 128-color dictionary, and a seven-bit-reference pointer for each pixel to designate a specific color among the 128 colors of the dictionary. For instance, the value $(000\ 0000)_2$ points out to the first color, the value $(000\ 0001)_2$ points out to the second color, the value $(000\ 0010)_2$ points out to the third color, and so on up to the last value $(111\ 1111)_2$, which is equivalent to $(127)_{10}$, that points out to the last color.

| Address part | I value<br>J value | Block address<br>(2 bytes) |
|---|---|---|
| Dictionary Part (LUD)<br>A special-purpose dictionary of 128 three-RGB-color-component entries (i.e. 128 * 3 = 384 cells). | The Red component of the 1st color<br>The Green component of the 1st color<br>The Blue component of the 1st color | 1st color<br>(3 bytes) |
| | The Red component of the 2nd color<br>The Green component of the 2nd color<br>The Blue component of the 2nd color | 2nd color<br>(3 bytes) |
| | ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ | |
| | The Red component of the 127th color<br>The Green component of the 127th color<br>The Blue component of the 127th color | 127th color<br>(3 bytes) |
| | The Red component of the 128th color<br>The Green component of the 128th color<br>The Blue component of the 128th color | 128th color<br>(3 bytes) |
| Data Part<br>Using seven-bit-reference pointers. Each seven-bit-reference pointer refers to one and only one of the 128 entries of colors | Pixel Representation<br>Using seven-bit-reference pointers to point out to one of the 128 dictionary entries (Every individual pixel requires only seven bits to store its reference) | Pixels' Data |

Figure 14. Encoding of a block using a class of type "**T4**"

Over again, the data compression of this block class can be constructed by utilizing reference pointers that point out to a special LUD. In this regard, the SRP measure is modeled mathematically by Equation 8:

$$SRP(\text{“}\mathbf{T4}\text{”}) = \left[1 - \frac{2 + 3 * 2^7 + \frac{BL^2}{8/7}}{3 * BL^2}\right] * 100\% = \left[1 - \frac{386 + BL^2 * \frac{7}{8}}{3 * BL^2}\right] * 100\% \quad (8)$$

This equation is similar to Equation 4 except the following differences:

- Dictionary part (LUD): Since there are ($2^8$=128) colors, the expression "3*$2^7$" of the numerator stands for the number of bytes that are required to store the RGB dictionary (i.e. LUD).
- Data part: Since there are (128) colors and each of them required seven bits to code, the number of pixels that can be stored in a single byte should be divided by (8/7) or be multiplied by (7/8). So the expression "$BL^2$/ (8/7)"

is used to determine the number of bytes that are essential to store the seven-bit-reference pointers of each block of this class.

For further clarification, a complete example of this type is introduced at "Appendix A" at the end of this paper. Besides that this example explains the compression process in a simple way, it gives experimental proof to support the validity of Equation 8.

*4.5 Class "**T5**" (129-256 Colors)*

A block is identified as grey if the values of the corresponding three basic RGB components of all pixels of the block are almost equal. Rather than repeating the same information for the three repeated RGB color components, one component is enough to represent the other two components. Though, the red component is selected to represent the other two color components.

Compared with the previous four classes, neither the special dictionary (i.e. LUD) nor the reference pointers are required for the blocks of this class. Rather, the actual red component of the original block is selected and directly stored as it is without any reshaping or rearrangement. Figure 15 illustrates how the data can be constructed for this class of blocks. Each block is just represented by its address (I, J) and the actual red components of its pixels where each pixel needs a single byte.

| Address part | **I value**<br>**J value** | Block address<br>(2 bytes) |
|---|---|---|
| Data Part<br>Pixels' data contain only Red components. Neither the special-purpose dictionary (LUD) nor the reference pointers are essential here. Each pixel occupies only one byte to be stored. | The Red component of the 1st color | 1st color (1 byte) |
| | The Red component of the 2nd color | 2nd color (1 byte) |
| | The Red component of the 3rd color | 3rd color (1 byte) |
| | The Red component of the 4th color | 4th color (1 byte) |
| | The Red component of the 5th color | 5th color (1 byte) |
| | ⋮ ⋮ ⋮ ⋮ ⋮ ⋮ | Pixels' data Using pointers.<br>Since they are grey colors, the red components are enough to be stored. |
| | Pixels' representation of the remaining pixels (1 byte per pixel)<br>&<br>(only the Red components are stored) | |

Figure 15. Encoding of a block using a class of type "**T5**"

Since class "**T5**" is considered as grey, the dictionary is needless and the SRP per block is defined using Equation 9:

$$\text{SRP}(\text{“}\mathbf{T5}\text{”}) = \left[1 - \frac{2 + \text{BL}^2}{3 * \text{BL}^2}\right] * 100\% \qquad (9)$$

The basic difference between the last two Equations, 8 and 9, is that the dictionary is needless in the latter one. Given that there are ($2^8$=256) colors, each color takes up just one byte, hence ($\text{BL}^2/1 = \text{BL}^2$). For a complete example of this class type, see Appendix B at the end of this paper. This example, on the other hand, gives an empirical proof about its validity.

*4.6 Class "**T6**" (more than 256 Colors)*

Different than class "**T5**" which only stores the red component, all the three basic RGB components of the original block are stored in class "**T6**" and, therefore, each pixel occupies three bytes. The representation of these blocks is saved by storing the address (I, J) of the block and the actual pixels' data where each pixel requires three bytes. Figure 16 shows the data structure representation of this class of blocks.

The SRP per block is represented by Equation 10:

$$\text{SRP}(\text{“}\mathbf{T6}\text{”}) = \left[1 - \frac{2 + 3 * \text{BL}^2}{3 * \text{BL}^2}\right] * 100\% \qquad (10)$$

The following points clarify this equation:

- The dictionary and pointers are needless.
- In any case, this equation gives negative results for this class. But the result is approaching zero value and, therefore, the counter loss of storage space can be easily affordable and then disregard without making a big difference.
- The rest of this equation is similar to Equation 4.

For further clarification and understanding, Appendix C at the end of this paper gives a complete example of this type and gives real empirical proof about the validity of Equation 10.

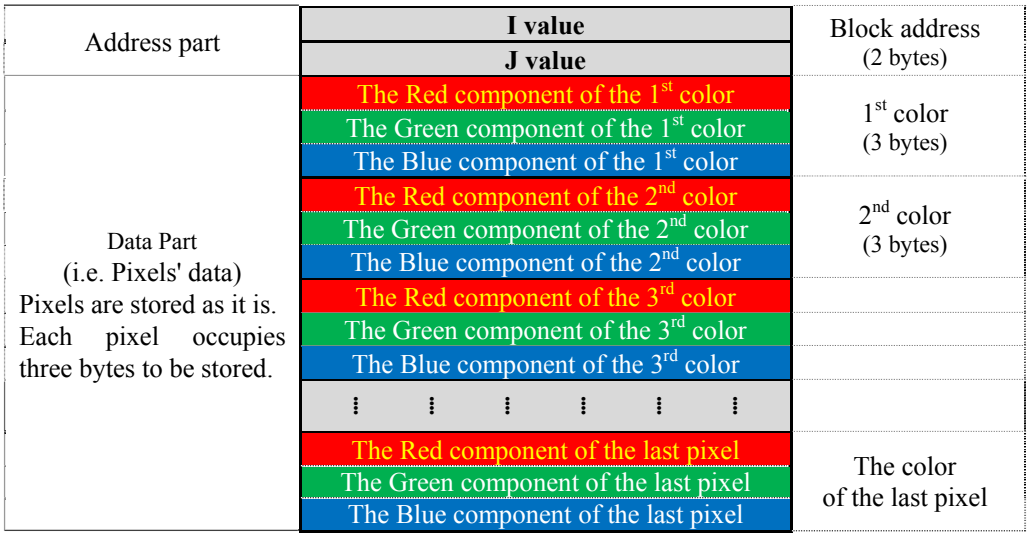


Figure 16. Encoding of a block using a class of type **"T6"**

## 5. Experimental Results & Evaluation

To assist in compare and contrast, a short outline of the six different classes is outlined in Table 11. It is worth remembering that the block length, "**BL**", is only stored once at the first byte of class "**T1**". Since it is reserved as a single byte, the maximum block length is "255". Otherwise, there is a necessity to change the size of the block length.

Table 11. Compare and contrast between the six classes

| Class Type | "BL" | Block Address | Max. No. of Colors | Dictionary Size (byte) | LUD | Reference pointer width | Every three bytes are stored as: |
|---|---|---|---|---|---|---|---|
| **"T1"** | ✓ | ✓ | **1** | 1*3=3 | ✓ | × | Zero bit |
| **"T2"** | × | ✓ | **2** | 2*3= 6 | ✓ | 1 bit | One bit |
| **"T3"** | × | ✓ | 16 | 16*3=48 | ✓ | 4 bits | Four bits |
| **"T4"** | × | ✓ | 128 | 128*3=384 | ✓ | 7 bits | 7 bits |
| **"T5"** | × | ✓ | 256 | × (i.e. zero) | × | × | 8 bits |
| **"T6"** | × | ✓ | >256 | × (i.e. zero) | × | × | 24 bits |

Since a reliable system should be experimented and analyzed on a great number of samples, a certain special-purpose database contains different image types were created as reported in Table 12. This database contains a dataset of 3151 24-bit-RGB-bitmap images of various resolutions distributed among eight categories of three-RGB-component colors. AS it is creative and has a productive service environment, the experiments have been carried out and tested empirically using MATLAB® 9.4 (R2018a) environments. (MathWorks Inc 2019)

After this proposed algorithm has been conducted upon this database, a proportionate reduction in the compression level has been achieved and this empirically-based evidence, on the other hand, shows rapprochement between theoretical and experimental results. To put it another way, all the ten equations stated in this research are proved both theoretically and empirically as being correct. The result is therefore worthy and the saving percentage (SRP) for the whole dataset in terms of storage space reduction is significant, which is

(71.039%). On the way to compare and contrast, this admirable result is totally better than the previous result of El-Omari et al. (2017) which is (87%) but for the documents that contain only texts and graphics" (El-Omari et al. 2017). For further clear investigation and evaluation, Table 13 illustrates these remarks and results for different block classes and block lengths. The strikethrough bolded cells in the last column of this table are introduced to show the cases where the compression ratio is poor due to the fact that:

> *If the data block is of class "**T6**", then the current data are stored as it is along with its block address (I, J) which is a two-byte length.*

Table 12. A special-purpose database created for the purpose of testing.

| Type no. | Image content | Number of images |
|---|---|---|
| 1 | Pure backgrounds | 218 |
| 2 | Pure texts | 556 |
| 3 | Pure graphics | 388 |
| 4 | Pure pictures | 414 |
| 5 | Graphs and pictures without texts | 218 |
| 6 | Texts and pictures without graphics | 286 |
| 7 | Texts and graphics without pictures | 369 |
| 8 | Mixed images | 702 |
| **Total number of images** | | **3151** |

Table 13. A numeric example showing the SRP measure for the six classes using one-byte block length

| Class / "BL" | "T1" Single color Equation 4 | "T2" 2 colors Equation 6 | "T3" 3-16 colors Equation 7 | "T4" 17-128 colors Equation 8 | "T5" 129-256 colors Equation 9 | "T6" >256 colors Equation 10 |
|---|---|---|---|---|---|---|
| 25 | 99.680 | 95.407 | 80.667 | 50.247 | 66.560 | ~~**-0.107**~~ |
| 35 | 99.837 | 95.616 | 81.973 | 60.330 | 66.612 | ~~**-0.054**~~ |
| 45 | 99.901 | 95.702 | 82.510 | 64.479 | 66.634 | ~~**-0.033**~~ |
| 55 | 99.934 | 95.745 | 82.782 | 66.580 | 66.645 | ~~**-0.022**~~ |
| 65 | 99.953 | 95.770 | 82.939 | 67.788 | 66.651 | ~~**-0.016**~~ |
| 75 | 99.964 | 95.786 | 83.037 | 68.546 | 66.655 | ~~**-0.012**~~ |
| 85 | 99.972 | 95.796 | 83.103 | 69.052 | 66.657 | ~~**-0.009**~~ |
| 95 | 99.978 | 95.804 | 83.149 | 69.408 | 66.659 | ~~**-0.007**~~ |
| 105 | 99.982 | 95.809 | 83.182 | 69.666 | 66.661 | ~~**-0.006**~~ |
| 115 | 99.985 | 95.813 | 83.207 | 69.860 | 66.662 | ~~**-0.005**~~ |
| 125 | 99.987 | 95.816 | 83.227 | 70.010 | 66.662 | ~~**-0.004**~~ |
| 135 | 99.989 | 95.819 | 83.242 | 70.127 | 66.663 | ~~**-0.004**~~ |
| 145 | 99.990 | 95.821 | 83.254 | 70.221 | 66.663 | ~~**-0.003**~~ |
| 155 | 99.992 | 95.822 | 83.264 | 70.298 | 66.664 | ~~**-0.003**~~ |
| 165 | 99.993 | 95.824 | 83.272 | 70.361 | 66.664 | ~~**-0.002**~~ |
| 175 | 99.993 | 95.825 | 83.279 | 70.413 | 66.664 | ~~**-0.002**~~ |
| 185 | 99.994 | 95.826 | 83.285 | 70.457 | 66.665 | ~~**-0.002**~~ |
| 195 | 99.995 | 95.826 | 83.290 | 70.495 | 66.665 | ~~**-0.002**~~ |
| 205 | 99.995 | 95.827 | 83.294 | 70.527 | 66.665 | ~~**-0.002**~~ |
| 215 | 99.996 | 95.828 | 83.297 | 70.555 | 66.665 | ~~**-0.001**~~ |
| 225 | 99.996 | 95.828 | 83.300 | 70.579 | 66.665 | ~~**-0.001**~~ |
| 235 | 99.996 | 95.829 | 83.303 | 70.600 | 66.665 | ~~**-0.001**~~ |
| 245 | 99.997 | 95.829 | 83.306 | 70.619 | 66.666 | ~~**-0.001**~~ |
| 255 | 99.997 | 95.829 | 83.308 | 70.635 | 66.666 | ~~**-0.001**~~ |
| **Average** | **99.962** | **95.783** | **83.020** | **68.411** | **66.654** | **-0.013** |

Connected with Table 13, Figure 17 is a graphical evaluation that presents clearly the relation between the block class and the average SRP measure. As clearly shown in this Figure, the overall saving ratio of the proposed algorithm is based on recognizing the class of the block.

By analyzing Table 13 and its associated Figure 17, it can be concluded that the best results of the storage space reduction can be achieved with classes of type "**T1**" where the average compression ratio is (99.962%), which means that the entire image is a background containing one color. The next best results can be extracted with the class of type "**T2**" which is (95.783%). Then, the next one is achieved by the class of type "**T3**" which has (83.020%). Next, classes of types "**T4**" and "**T5**" with the percentages (68.411%), (66.654%), respectively.

Due to the fact that additional two-byte storage is required, the worst case is whenever the blocks are of the class of type "**T6**", which means that the entire image is a picture. In this case, the encoding of this approach is not appropriate and the proposed system is dynamic enough to cancel the encoding process and use another proper encoder. But, this worst-case (i.e. "**T6**") has an average losing percentage that is around zero (precisely 0.013 %) which can be neglected at the expense of the other worthy percentages.

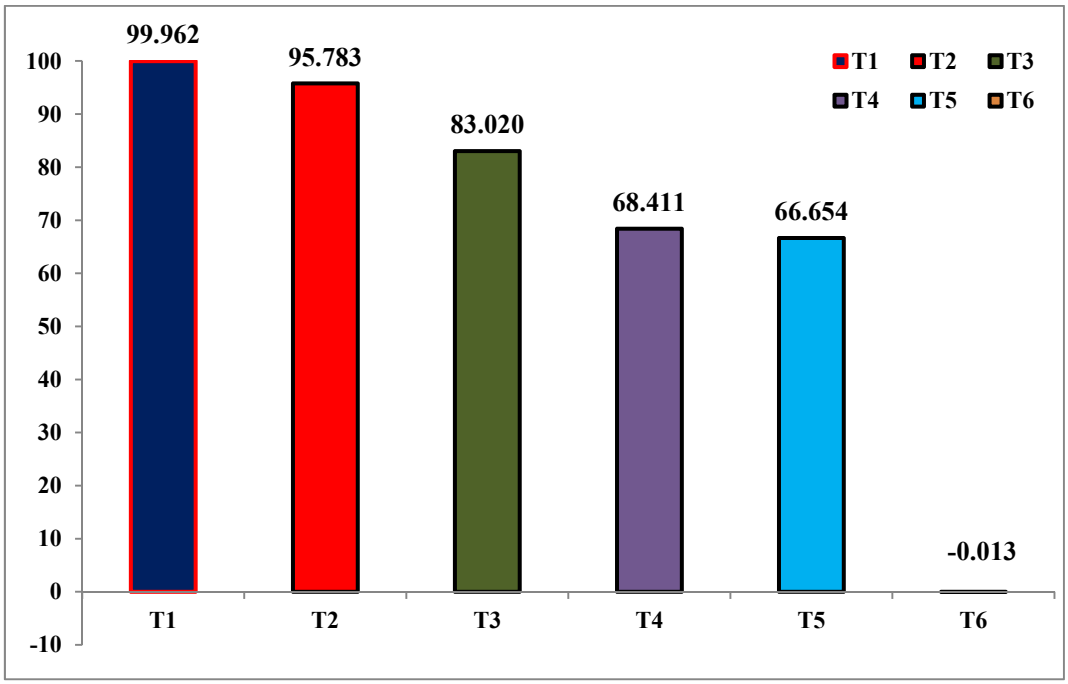


Figure 17. Per class-type compression ratios (using one-byte block length).

On the basis of the above-stated analyses, the block class type has a great impact on the SRP measure. Stated in other words, this measure is highly relying on the image content that leads to the distribution of the original table upon the six classes. By comparing the whole advantages and benefits of this proposed algorithm, it is proved that it is a very efficient alternative and able to produce comparably competitive results.

By a further evaluation of Table 13, the block length, "BL", has an impact on the received results. When the block length is increased, the SRP measure is increased, as well. This proves that SRP is directly proportional to the block length. Hence, duplicating this length is maybe imperative particularly for large-size images. In order to prove this truth, Table 14 and it is a related demonstration of Figure 18 use a two-byte block length. From another point of view, this table assures that the proposed algorithm is a significant one for segmentation and compression compound images.

Similar to the investigation of Table 13 and Figure 17, it can be concluded from Table 14 and Figure 18 that this proposed technique gives the best results for the first five classes. The best results can be achieved with the class of type "**T1**" where the SRP is (99.999%). The next best results that can be achieved are with the classes, "**T2**", "**T3**", "**T4**", and "**T5**" with the percentages (95.832%), (83.327%), (70.786%), (66.666%), respectively. Again, the worst case is of the class of type "**T6**" which is around zero (precisely 0.0002446 %). Since this loss is too small to be observed, it can be neglected without making a big difference.

Table 14. A numeric example showing SRP for the six classes using two-byte block lengths

| Class / "BL" | "T1" Single color Equation 4 | "T2" 2 colors Equation 6 | "T3" 3-16 colors Equation 7 | "T4" 17-128 colors Equation 8 | "T5" 129-256 colors Equation 9 | "T6" >256 colors Equation 10 |
|---|---|---|---|---|---|---|
| 280 | 99.997 | 95.830 | 83.312 | 70.669 | 66.666 | ~~-0.001~~ |
| 305 | 99.998 | 95.830 | 83.315 | 70.695 | 66.666 | ~~-0.001~~ |
| 330 | 99.998 | 95.831 | 83.318 | 70.715 | 66.666 | ~~-0.001~~ |
| 355 | 99.998 | 95.831 | 83.320 | 70.731 | 66.666 | ~~-0.001~~ |
| 380 | 99.999 | 95.831 | 83.322 | 70.744 | 66.666 | 0.000 |
| 405 | 99.999 | 95.832 | 83.323 | 70.755 | 66.666 | 0.000 |
| 430 | 99.999 | 95.832 | 83.324 | 70.764 | 66.666 | 0.000 |
| 455 | 99.999 | 95.832 | 83.325 | 70.771 | 66.666 | 0.000 |
| 480 | 99.999 | 95.832 | 83.326 | 70.777 | 66.666 | 0.000 |
| 505 | 99.999 | 95.832 | 83.327 | 70.783 | 66.666 | 0.000 |
| 530 | 99.999 | 95.832 | 83.327 | 70.788 | 66.666 | 0.000 |
| 555 | 99.999 | 95.832 | 83.328 | 70.792 | 66.666 | 0.000 |
| 580 | 99.999 | 95.833 | 83.328 | 70.795 | 66.666 | 0.000 |
| 605 | 99.999 | 95.833 | 83.329 | 70.798 | 66.666 | 0.000 |
| 630 | 99.999 | 95.833 | 83.329 | 70.801 | 66.666 | 0.000 |
| 655 | 100.000 | 95.833 | 83.329 | 70.803 | 66.667 | 0.000 |
| 680 | 100.000 | 95.833 | 83.330 | 70.806 | 66.667 | 0.000 |
| 705 | 100.000 | 95.833 | 83.330 | 70.807 | 66.667 | 0.000 |
| 730 | 100.000 | 95.833 | 83.330 | 70.809 | 66.667 | 0.000 |
| 755 | 100.000 | 95.833 | 83.330 | 70.811 | 66.667 | 0.000 |
| 780 | 100.000 | 95.833 | 83.331 | 70.812 | 66.667 | 0.000 |
| 805 | 100.000 | 95.833 | 83.331 | 70.813 | 66.667 | 0.000 |
| 830 | 100.000 | 95.833 | 83.331 | 70.815 | 66.667 | 0.000 |
| 855 | 100.000 | 95.833 | 83.331 | 70.816 | 66.667 | 0.000 |
| 880 | 100.000 | 95.833 | 83.331 | 70.817 | 66.667 | 0.000 |
| 905 | 100.000 | 95.833 | 83.331 | 70.818 | 66.667 | 0.000 |
| 930 | 100.000 | 95.833 | 83.331 | 70.818 | 66.667 | 0.000 |
| 955 | 100.000 | 95.833 | 83.332 | 70.819 | 66.667 | 0.000 |
| 980 | 100.000 | 95.833 | 83.332 | 70.820 | 66.667 | 0.000 |
| 1005 | 100.000 | 95.833 | 83.332 | 70.821 | 66.667 | 0.000 |
| **Average** | **99.999** | **95.832** | **83.327** | **70.786** | **66.666** | **0.000** |

Compared with the other approaches, the most important advantage of this proposed algorithm is its simplicity (less than five operations per pixel), clarity and directness, dependency on just a few parameters And, above all, its reliability. Furthermore, this proposed approach combines different compression concepts in order to achieve better compression ratios of the scanned documents; its basic scope is based upon hybridizing the following methods that are already demonstrated in Figure 3:

- Dynamic Algorithms: Since it is relying on capturing more details about the problem of interest, it's a dynamic and content-based algorithm.
- Statistical-based: It is a local statistical thresholding approach where the blocks classification can be achieved by exploiting some prior knowledge relevant to the number of colors that originally exist within the image or one of its blocks.

- Dictionary-based: For the reason that the dictionary of colors is included inside the internal data representation, it is a dictionary-based-compression scheme.
- Block-based encoding: It is a block-based approach where the input image is divided into equal-size-square blocks.
- Multi-layered encoding: As each input image is divided into six regions in view of the number of the detected colors, it is a region-based approach, as well.
- Lossless encoding: When the final image of Phase VII is retrieved back and compared with the original one, the two images are entirely the same and precisely bit-by-bit alike. This guarantees that every bit can be retrieved back precisely to its original value without any level of distortion and hence the process is reversible. This also implies that the proposed algorithm can be recognized as a lossless one or at least near-lossless.

Above and beyond that, not only this approach crosses the aforesaid models but also it is a two-level compression technique (i.e. Phase V and Phase VII). Finally, to conclude the discussion of this section, if the logical operation "XOR" is accomplished on both the encoded input images and the decoded output images, the result is zero (i.e. off or false) which means that both the images are alike. Therefore, the output quality of this phase is (100%) which also reinsures the above-stated conclusion that says: *this technique is a lossless one*.

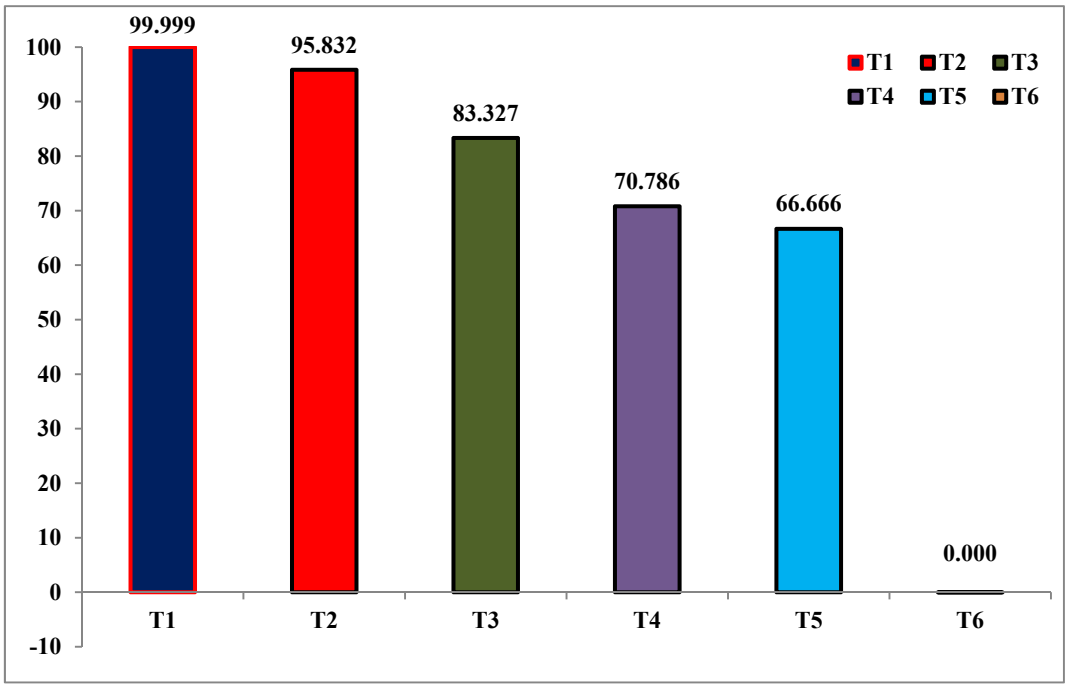


Figure 18. Per class-type compression ratios (using a two-byte block length).

## 6. Conclusion

An increase in the demand of numerous millions of computer users for storing more numerous millions of images paved the way for viewing segmentation and compression techniques and seeing them as more intertwined than ever. And so, the present work proposes a lossless statistical block-based segmentation technique that works in conjunction with other encoding techniques to segment compound or mixed documents that have different content types, such as pictures, graphics, texts, and/or backgrounds. Furthermore, this research has disclosed very stimulating and deep-insight findings that can significantly improve the mechanisms by which the segmentation and compression of compound images are currently evaluated.

With regard to the number of colors detected in each part of the image, this paper involves a seven-phase approach in which an incoming compound document is segmented into a set of multiple image objects, each compressed by the most off-the-shelf applicable compression technique. This approach hybridizes different compression concepts to achieve better compression in terms of space-time complexity. It is a block-based approach where the input image is divided into equal-size-square blocks. It is a region-based approach where the input image is divided into homogeneous regions according to the number of colors. Besides it's a dynamic and

content-based algorithm, it is a threshold approach where the blocks classification is carried out by exploiting the number of colors that exist within the block or the image. Since the lookup dictionary of colors is included in the internal representation of the third phase (i.e. Rearrangement phase) and in the external representation of the six$^{th}$ phase (i.e. Integrating phase), it is a dictionary-based-compression scheme.

Motivated by the purpose of testing the performance of the proposed algorithm, a special database was created. It contains a dataset of 3151 24-bit-RGB-bitmap document images with different image types and rich-mix contents. In view of the empirical findings, the outcomes of the conducted experiments are admirable and the overall average saving rate that has been achieved is (71.039%) for the whole dataset. The important thing is that the relevant matching analysis between the theorizing (i.e. Equations 1 through 10) and the empirically-based results show rapprochement without any discrepancies. Thus, the algorithm is efficient, robust, and has the capability of handling compound documents that have different content types. However, the performance of this solution, like most other image compression algorithms, depends on the content of the file to be compressed. Finally, as the input encoded image and the output decoded image are recognized as the same and recorded as identical up to the last bit, this technique is a lossless one.

## 7. Future Work and Outlook

In order to realize the potential advantages of this proposed technique upon this significant area, further experimental and simulation researches should be carried out and, in turn, several significant issues can be extended for future work to support the achievements of this work. These issues may lead to further improvement related to more storage space reduction and, furthermore, bring to light a great number of new research opportunities that need to be further investigated. On the whole, the research scope can be extended to introduce the following perspectives:

- In the way of maximizing the existing compression ratios, the grey-scale resolution (i.e. bit-depth) can be increased from 24-bit to other values. Then the impact of this modification upon the six-SRP classes should be investigated.
- Because some regions may have more relative importance than the others, this algorithm may take further direction related to the preservation of information. For instance, the regions of the vehicle plates might be more significant to be verified precisely than the other parts of the vehicle (Alghyaline et al. 2019). And, therefore, a “lossless” algorithm is applied to vehicle plates and “lossy” compression is applied for the rest of the image. Hence, “lossless” and “lossy” are used upon this importance.
- According to analysts and specialists, it is extremely rare to see these current days anyone living without Internet access and, above and beyond that, it is foreseen in the next few coming years that there isn't any running business without the innovative Cloud Computing (CC) services (El-Omari 2019). As most Digital Image Processing (DIP) applications are high-productivity and could be deployed remotely with the new vision of the smart world, there is an utmost need to integrate the DIP paradigm to be activated within the CC environment (Yuzhong and Lei 2014)(El-Omari 2019). This is especially true for hosting and delivering this proposed solution; the following motivations reinforces this relevant point and truly ensure that CC is the most tolerable place for hosting DIP systems:
  - The majority of these systems are typically sophisticated and entail high-end communicational capabilities, high-level computational power, well-developed applications, and large mass data storage capacities (Kumar et al. 2019) (M Gokilavani, GP Mannickathan, and MA. Dorairangaswamy 2018)(El-Omari 2019).
  - Most DIP systems require application-specific platforms and real-time or near-real-time applications (El-Omari 2019)(Yuzhong and Lei 2014).
  - Given that CC is moving in the direction of providing highest Quality of Service (QoS) at a lower expense, the underlying hardware of these systems is usually very expensive to be single-owned by the enterprise itself (Mirarab, Fard, and Shamsi 2014)(El-Omari 2019)(Qin et al. 2018).
  - Moreover, the three-field integration (CC, Big Data, and DIP) has recently become the most desirable platform for hosting and delivering DIP functions (El-Omari and Alzaghal 2017)(Mirarab et al. 2014)(Kang and Lee 2016).

  By this, segmentation and compression compound images based on utilizing CC might become a wildly-popular simple practice among ordinary users.
- Since image compression has a positive leading contribution in the security area (Kumar et al. 2019), the data inside the Lookup Dictionary Table (LUD) and the reference pointers can be encrypted at the third phase (i.e.

Phase III: Rearrangement phase) of this proposed algorithm which, as a result, leads to an encrypted integrated file as an output of phase 6.

- In the foreseeable future, there may be many possible arrangements that can be arranged to rapidly accelerate the compression/decompression progress of this proposed technique; some of them have been already highlighted in Figure 3. Followings are two of these arrangements:
  - Building the data compression/decompression process as real-time utility software that may be considered as part of the operating system. By using this strategy, every data file is directly encoded when it is stored and, in contrast, it is automatically decoded when it is retrieved back (i.e. loaded).
  - Building the data compression/decompression mechanism internally as a special-purpose built-in chip. Again as have been stated in the previous point, every file is automatically compressed during the saving process, and vice versa.

Equally important, both arrangements should be designed to be operated automatically without the users' interferences. In addition, these arrangements should be worked without the end-users' awareness of their existence.

**References**

Alghyaline, Salah, Nidhal. K. T. El-Omari, Ra'ed M. Al-Khatib, & Hesham Al-Kharbshh. (2019). RT-VC: An Efficient Real-Time Vehicle Counting Approach. *Journal of Theoretical and Applied Information Technology (JATIT),* 97(7), 2062–75.

Azad, Abul Kalam, Rezwana Sharmeen, Shabbir Ahmad, & S. M. Kamruzzaman. (2010). "An Efficient Technique for Text Compression." 467–73 in *The 1st International Conference on Information Management and Business, The University of South Australia*. The University of South Australia.

Boopathiraja, S., P. Kalavathi, & C. Dhanalakshmi. (2019). *Significance of Image Compression and Its Upshots – A Survey, 5*(2),1203–8. https://doi.org/10.32628/CSEIT1952321

El-Omari, Nidhal. K. T. (2019). Cloud IoT as a Crucial Enabler: A Survey and Taxonomy. *Modern Applied Science, 13*(8), 86–149. https://doi.org/10.5539/mas.v13n8p86

El-Omari, Nidhal. K. T., A. H. Omari, O. F. Al-badarneh, & H. Abdel-jaber. (2012). Scanned Document Image Segmentation Using Back-Propagation Artificial Neural Network Based Technique. *International Journal of Computers and Communications, 6*(4), 183–90.

El-Omari, Nidhal K. T., Ahmad H. Al-omari, Ali Mohammad H. Al-ibrahim, & Tariq Alwada. (2017). Text-Image Segmentation and Compression Using Adaptive Statistical Block Based Approach. *International Journal of Engineering and Advanced Technology (IJEAT), 6*(4), 1–9.

El-Omari, Nidhal K. T. & Arafat A. Awajan. (2009). "Document Image Segmentation and Compression Using Artificial Neural Network Based Technique." 320–24 in *International Conference on Information and Communication Systems (ICICS09), Amman, Jordan*.

El-Omari, Nidhal Kamel Taha. (2008). A Hybrid Approach for Segmentation and Compression of Compound Images. *The Arab Academy for Banking and Financial Sciences,* 1–201.

El-Omari, Nidhal Kamel Taha & Mohamad H. Alzaghal. (2017). "The Role of Open Big Data within the Public Sector, Case Study: Jordan." Pp. 182–86 in *The 8th International Conference on Information Technology (ICIT 2017), Internet of Things, IEEE, Amman, Jordan*. https://doi.org/10.1109/ICITECH.2017.8079997

Gonzalez, Rafael C. & Richard E. Woods. (2017). *Digital Image Processing*. 4th ed. Pearson Education In.

Gonzalez, Rafael C., Richard E. Woods, & Steven L.Eddins. (2009). *Digital Image Processing Using Matlab (DIPUM)*. 2nd ed. Gatesmark Publishing.

Kang, Sanggoo & Kiwon Lee. (2016). Auto-Scaling of Geo-Based Image Processing in an OpenStack Cloud Computing Environment. *Remote Sensing, 8*(8), 662–71. https://doi.org/10.3390/rs8080662

Kumar, N. Udaya, M. Madhavi Latha, E. V. Krishna Rao, & K. Padma Vasavi. (2019). Multi Scale Multi Directional Region of Interest Based Image Compression Using Non Subsampled Contourlet Transform. *SCIREA Journal of Electrics, Communication, 2*(1), 1–18.

M Gokilavani, GP Mannickathan, & MA. Dorairangaswamy. (2018). A Survey of Cloud Environment in Medical Images Processing. *Monthly Journal of Computer Science and Information Technology, 7*(11), 68–73.

MathWorks Inc. (2019). MATLAB The Language of Technical Computing. *The Math Works INC*. Retrieved January 3, 2020 (https://www.mathworks.com/help/matlab/index.html).

Mirarab, Ali, Najmeh Ghasemi Fard, & Mahboubeh Shamsi. (2014). A Cloud Solution for Medical Image Processing. *International Journal of Engineering Research and Applications (IJERA), 4*(7), 74–82.

Petrou, Maria & Panagiota Bosdogianni. (2010). *Image Processing: The Fundamentals*. John Wiley & Sons. https://doi.org/10.1002/9781119994398

Qin, Z., J. Weng, Y. Cui, & K. Ren. (2018). Privacy-Preserving Image Processing in the Cloud. *IEEE Cloud Computing,* 5(2), 48–57. https://doi.org/10.1109/MCC.2018.022171667

Queiroz, Ricardo L. de, Robert Buckley, & Ming Xu. (1999). Mixed Raster Content (MRC) Model for Compound Image Compression. *The International Society for Optical Engineering (SPIE ),* 3653, 1106–17.

Rahman, Atiqur & Mohamed Hamada. (2019). Lossless Image Compression Techniques: A State-of-the-Art Survey. *Symmetry, 11*(10), 1–22. https://doi.org/10.3390/sym11101274

Sharma, Pulkit. (2019). Computer Vision Tutorial: A Step-by-Step Introduction to Image Segmentation Techniques. Retrieved February 2, 2020 (https://www.analyticsvidhya.com/blog/2019/04/introduction-image-segmentation-techniques-python/).

Taha, Nidhal Kamel, Jafar Ababneh, Jamal N. Bani Salameh, Abdel Rahman A. Al Karabsheh, & Ali Mohammad H.Al-Ibrahim. (2012). Innoviate Text-Image Compression Technique. *European Journal of Scientific Research, 88*(4), 603–16.

Wikipedia. (2016). Lookup Table. *Wikipedia, the Free Encyclopedia*. Retrieved January 2, 2020 (https://en.wikipedia.org/wiki/Lookup_table).

Yuzhong, Yan & Huang Lei. (2014). "Large-Scale Image Processing Research Cloud." Pp. 88–93 in *The Fifth International Conference on Cloud Computing, GRIDs, and Virtualization, Venice, Italy*.

**Appendix A: An example of the "T4" blocks (17-128 colors)**

This example aims at putting a focus on how the class of type "**T4**" is fabricated. Suppose that a square block (I, J) = (48, 67) of 64x64 pixels in size has the 39-color CST that is viewed in Table 15. This table is achieved based on Table 1 and as an output of Algorithm II of Figure 8. This table contains four columns, the first three columns are for the RGB components and the last one is for the frequencies of colors that are found within this block. Since the size of the block is 4096 pixels, the total sum of the last column of this table is 4096 pixels.

Note how these 64*64=4096 pixels are distributed among 39 three-RGB-component colors and they are ordered in descending order according to their frequencies. Since this dictionary has 39 (i.e. from sequence 0 to sequence 38) colors, it is of type "**T4**" and so each pixel needs a seven-bit reference to point out to one of the 128 dictionary entries. When the LUD dictionary is built, the first color takes the number $(000)_{10} = (000\ 0000)_2$, the second color takes $(001)_{10} = (000\ 0001)_2$, the third color takes $(002)_{10} = (000\ 0010)_2$, the fourth color takes $(003)_{10} = (000\ 0011)_2$ , and so on until the last color that takes $(127)_{10} = (111\ 1111)_2$ . In order to complete the LUD entries, the remaining unoccupied (i.e. unfilled) entries of colors, from the $40^{th}$ color to the $128^{th}$ color, are fulfilled with null values and there isn't any reference pointer that points out to one of them.

Now, assume that the first twenty pixels of this block contain the following color components RGB:

| | | | | |
|---|---|---|---|---|
| $(255,100,150)_{10}$ | $(000,010,050)_{10}$ | $(000,010,050)_{10}$ | $(077,015,080)_{10}$ | $(167,100,200)_{10}$ |
| $(255,100,150)_{10}$ | $(000,010,050)_{10}$ | $(013,023,033)_{10}$ | $(255,100,150)_{10}$ | $(153,153,153)_{10}$ |
| $(000,010,050)_{10}$ | $(255,100,150)_{10}$ | $(241,100,111)_{10}$ | $(233,200,200)_{10}$ | $(241,100,111)_{10}$ |
| $(018,000,000)_{10}$ | $(000,010,050)_{10}$ | $(000,010,050)_{10}$ | $(109,100,209)_{10}$ | $(109,100,209)_{10}$ |

Inspired by the number of colors detected in this block, this technique treats this block as a class "**T4**". So, the corresponding reference pointers for these twenty pixels are described in Table 16. On the other side, Table 17 illustrates the data schematic construction of this example where a total of 3970 bytes are required for each block

of this type and size. For the remaining 4076 pixels, other than these twenty pixels, the same pattern is used. Related to Table 17, it is vitally important to mention that to simplify the viewing of the seven-bit-reference pointers, eight-bit-reference pointers are viewed instead. In terms of this, the total occupied bytes of the data part is multiplied by (7/8), i.e. 4096*7/8+386=3970.

Table 15. A CST example of 39 colors (data in decimal)

| | Red component | Green component | Blue component | Frequency |
|---|---|---|---|---|
| 0. | 255 | 100 | 150 | 969 |
| 1. | 000 | 010 | 050 | 816 |
| 2. | 077 | 015 | 080 | 395 |
| 3. | 167 | 100 | 200 | 392 |
| 4. | 155 | 105 | 113 | 289 |
| 5. | 100 | 050 | 090 | 199 |
| 6. | 007 | 017 | 027 | 166 |
| 7. | 099 | 100 | 105 | 81 |
| 8. | 016 | 017 | 018 | 59 |
| 9. | 012 | 024 | 036 | 49 |
| 10. | 013 | 023 | 033 | 49 |
| 11. | 225 | 100 | 150 | 49 |
| 12. | 140 | 160 | 190 | 40 |
| 13. | 189 | 189 | 189 | 40 |
| 14. | 208 | 208 | 208 | 40 |
| 15. | 124 | 124 | 124 | 39 |
| 16. | 199 | 199 | 199 | 39 |
| 17. | 198 | 200 | 200 | 39 |
| 18. | 197 | 197 | 197 | 32 |
| 19. | 240 | 240 | 240 | 32 |
| 20. | 241 | 100 | 111 | 28 |
| 21. | 244 | 200 | 200 | 28 |
| 22. | 102 | 111 | 111 | 24 |
| 23. | 153 | 153 | 153 | 20 |
| 24. | 104 | 104 | 104 | 20 |
| 25. | 154 | 154 | 154 | 20 |
| 26. | 217 | 200 | 200 | 19 |
| 27. | 178 | 100 | 100 | 18 |
| 28. | 233 | 200 | 200 | 17 |
| 29. | 001 | 002 | 003 | 16 |
| 30. | 002 | 004 | 006 | 16 |
| 31. | 011 | 113 | 111 | 16 |
| 32. | 017 | 027 | 037 | 12 |
| 33. | 018 | 000 | 000 | 8 |
| 34. | 109 | 100 | 209 | 7 |
| 35. | 255 | 90 | 100 | 4 |
| 36. | 255 | 95 | 110 | 3 |
| 37. | 255 | 90 | 120 | 3 |
| 38. | 255 | 90 | 130 | 3 |
| | | | **Total** | **4096** |

Finally, to conclude the discussion of this example, an important conclusion that should be stated here**:**

> Rather than spending **12288** (i.e. 64 * 64 * 3) bytes in storing this block of (64x64) pixels in size, 4482 bytes are enough. Namely:
>
> **3970 / (64 * 64 * 3) * 100% = 32.308%**
>
> This means that this proposed technique has the capability to encode this class of blocks by using only **32.308%** of the block size. If Equation 8 is recalculated by using BL=64, the result is the same without any discrepancies.

Table 16. The corresponding reference pointers of the example on the class type "**T4**"

| Pixel no. | Pixel data | Decimal color reference | 7-bit binary color reference |
|---|---|---|---|
| 01 | $( 255,100,150 )_{10}$ | $( 00 )_{10}$ | $( 000\ 0000 )_2$ |
| 02 | $( 000,010,050 )_{10}$ | $( 01 )_{10}$ | $( 000\ 0001 )_2$ |
| 03 | $( 000,010,050 )_{10}$ | $( 01 )_{10}$ | $( 000\ 0001 )_2$ |
| 04 | $( 077,015,080 )_{10}$ | $( 02 )_{10}$ | $( 000\ 0010 )_2$ |
| 05 | $( 167,100,200 )_{10}$ | $( 03 )_{10}$ | $( 000\ 0011 )_2$ |
| 06 | $( 255,100,150 )_{10}$ | $( 00 )_{10}$ | $( 000\ 0000 )_2$ |
| 07 | $( 000,010,050 )_{10}$ | $( 01 )_{10}$ | $( 000\ 0001 )_2$ |
| 08 | $( 013,023,033 )_{10}$ | $( 10 )_{10}$ | $( 000\ 1010 )_2$ |
| 09 | $( 255,100,150 )_{10}$ | $( 00 )_{10}$ | $( 000\ 0000 )_2$ |
| 10 | $( 153,153,153 )_{10}$ | $( 23 )_{10}$ | $( 001\ 0111 )_2$ |
| 11 | $( 000,010,050 )_{10}$ | $( 01 )_{10}$ | $( 000\ 0001 )_2$ |
| 12 | $( 255,100,150 )_{10}$ | $( 00 )_{10}$ | $( 000\ 0000 )_2$ |
| 13 | $( 241,100,111 )_{10}$ | $( 20 )_{10}$ | $( 001\ 0100 )_2$ |
| 14 | $( 233,200,200 )_{10}$ | $( 28 )_{10}$ | $( 001\ 1100 )_2$ |
| 15 | $( 241,100,111 )_{10}$ | $( 20 )_{10}$ | $( 001\ 0100 )_2$ |
| 16 | $( 018,000,000 )_{10}$ | $( 33 )_{10}$ | $( 010\ 0001 )_2$ |
| 17 | $( 018,000,000 )_{10}$ | $( 33 )_{10}$ | $( 010\ 0001 )_2$ |
| 18 | $( 018,000,000 )_{10}$ | $( 33 )_{10}$ | $( 010\ 0001 )_2$ |
| 19 | $( 109,100,209 )_{10}$ | $( 34 )_{10}$ | $( 010\ 0010 )_2$ |
| 20 | $( 109,100,209 )_{10}$ | $( 34 )_{10}$ | $( 010\ 0010 )_2$ |

Table 17. A data structure example for the blocks of type "**T4**"

| Data Type | | | Decimal Data | Byte Sequence |
|---|---|---|---|---|
| Block address | | I | $( 048 )_{10}$ | 001 |
| | | J | $( 067 )_{10}$ | 002 |
| Dictionary Part (LUD) A special-purpose dictionary of 128 three-RGB-color-component entries (i.e. 128 * 3 = 384 cells). | 1st color | $R_{001}$ | $( 255 )_{10}$ | 003 |
| | | $G_{001}$ | $( 100 )_{10}$ | 004 |
| | | $B_{001}$ | $( 150 )_{10}$ | 005 |
| | 2nd color | $R_{002}$ | $( 000 )_{10}$ | 006 |
| | | $G_{002}$ | $( 010 )_{10}$ | 007 |
| | | $B_{002}$ | $( 050 )_{10}$ | 008 |
| | 3rd color | $R_{003}$ | $( 077 )_{10}$ | 009 |
| | | $G_{003}$ | $( 015 )_{10}$ | 010 |
| | | $B_{003}$ | $( 080 )_{10}$ | 011 |
| | 4th color | $R_{016}$ | $( 167 )_{10}$ | 012 |
| | | $G_{016}$ | $( 100 )_{10}$ | 013 |

| Data Type | | | Decimal Data | Byte Sequence |
|---|---|---|---|---|
| | | $B_{016}$ | $(200)_{10}$ | 014 |
| | $5^{th}$ color | $R_{005}$ | $(155)_{10}$ | 015 |
| | | $G_{005}$ | $(105)_{10}$ | 016 |
| | | $B_{005}$ | $(113)_{10}$ | 017 |
| | $6^{th}$ color | $R_{006}$ | $(100)_{10}$ | 018 |
| | | $G_{006}$ | $(050)_{10}$ | 019 |
| | | $B_{006}$ | $(090)_{10}$ | 020 |
| | $7^{th}$ color | $R_{007}$ | $(007)_{10}$ | 021 |
| | | $G_{007}$ | $(017)_{10}$ | 022 |
| | | $B_{007}$ | $(027)_{10}$ | 023 |
| | $8^{th}$ color | $R_{008}$ | $(099)_{10}$ | 024 |
| | | $G_{008}$ | $(100)_{10}$ | 025 |
| | | $B_{008}$ | $(105)_{10}$ | 026 |
| | $9^{th}$ color | $R_{009}$ | $(016)_{10}$ | 027 |
| | | $G_{009}$ | $(017)_{10}$ | 028 |
| | | $B_{009}$ | $(018)_{10}$ | 029 |
| | $10^{th}$ color | $R_{010}$ | $(012)_{10}$ | 030 |
| | | $G_{010}$ | $(024)_{10}$ | 031 |
| | | $B_{010}$ | $(036)_{10}$ | 032 |
| | $11^{th}$ color | $R_{011}$ | $(013)_{10}$ | 033 |
| | | $G_{011}$ | $(023)_{10}$ | 034 |
| | | $B_{011}$ | $(033)_{10}$ | 035 |
| | $12^{th}$ color | $R_{012}$ | $(225)_{10}$ | 036 |
| | | $G_{012}$ | $(100)_{10}$ | 037 |
| | | $B_{012}$ | $(150)_{10}$ | 038 |
| | $13^{th}$ color | $R_{013}$ | $(140)_{10}$ | 039 |
| | | $G_{013}$ | $(160)_{10}$ | 040 |
| | | $B_{013}$ | $(190)_{10}$ | 041 |
| | $14^{th}$ color | $R_{014}$ | $(189)_{10}$ | 042 |
| | | $G_{014}$ | $(189)_{10}$ | 043 |
| | | $B_{014}$ | $(189)_{10}$ | 044 |
| | $15^{th}$ color | $R_{015}$ | $(208)_{10}$ | 045 |
| | | $G_{015}$ | $(208)_{10}$ | 046 |
| | | $B_{015}$ | $(208)_{10}$ | 047 |
| | $16^{th}$ color | $R_{016}$ | $(124)_{10}$ | 048 |
| | | $G_{016}$ | $(124)_{10}$ | 049 |
| | | $B_{016}$ | $(124)_{10}$ | 050 |
| | $17^{th}$ color | $R_{017}$ | $(199)_{10}$ | 051 |
| | | $G_{017}$ | $(199)_{10}$ | 052 |
| | | $B_{017}$ | $(199)_{10}$ | 053 |
| | $18^{th}$ color | $R_{018}$ | $(198)_{10}$ | 054 |
| | | $G_{018}$ | $(200)_{10}$ | 055 |
| | | $B_{018}$ | $(200)_{10}$ | 056 |
| | $19^{th}$ color | $R_{019}$ | $(197)_{10}$ | 057 |
| | | $G_{019}$ | $(197)_{10}$ | 058 |
| | | $B_{019}$ | $(197)_{10}$ | 059 |
| | $20^{th}$ color | $R_{020}$ | $(240)_{10}$ | 060 |
| | | $G_{020}$ | $(240)_{10}$ | 061 |
| | | $B_{020}$ | $(240)_{10}$ | 062 |
| | $21^{st}$ color | $R_{021}$ | $(241)_{10}$ | 063 |
| | | $G_{021}$ | $(100)_{10}$ | 064 |

| Data Type | | | Decimal Data | Byte Sequence |
|---|---|---|---|---|
| | | $B_{021}$ | $(111)_{10}$ | 065 |
| | 22nd color | $R_{022}$ | $(244)_{10}$ | 066 |
| | | $G_{022}$ | $(200)_{10}$ | 067 |
| | | $B_{022}$ | $(200)_{10}$ | 068 |
| | 23rd color | $R_{023}$ | $(102)_{10}$ | 069 |
| | | $G_{023}$ | $(111)_{10}$ | 070 |
| | | $B_{023}$ | $(111)_{10}$ | 071 |
| | 24th color | $R_{024}$ | $(153)_{10}$ | 072 |
| | | $G_{024}$ | $(153)_{10}$ | 073 |
| | | $B_{024}$ | $(153)_{10}$ | 074 |
| | 25th color | $R_{025}$ | $(104)_{10}$ | 075 |
| | | $G_{025}$ | $(104)_{10}$ | 076 |
| | | $B_{025}$ | $(104)_{10}$ | 077 |
| | 26th color | $R_{026}$ | $(154)_{10}$ | 078 |
| | | $G_{026}$ | $(154)_{10}$ | 079 |
| | | $B_{026}$ | $(154)_{10}$ | 080 |
| | 27th color | $R_{027}$ | $(217)_{10}$ | 081 |
| | | $G_{027}$ | $(200)_{10}$ | 082 |
| | | $B_{027}$ | $(200)_{10}$ | 083 |
| | 28th color | $R_{028}$ | $(178)_{10}$ | 084 |
| | | $G_{028}$ | $(100)_{10}$ | 085 |
| | | $B_{028}$ | $(100)_{10}$ | 086 |
| | 29th color | $R_{029}$ | $(233)_{10}$ | 087 |
| | | $G_{029}$ | $(200)_{10}$ | 088 |
| | | $B_{029}$ | $(200)_{10}$ | 089 |
| | 30th color | $R_{030}$ | $(001)_{10}$ | 090 |
| | | $G_{030}$ | $(002)_{10}$ | 091 |
| | | $B_{030}$ | $(003)_{10}$ | 092 |
| | 31st color | $R_{031}$ | $(002)_{10}$ | 093 |
| | | $G_{031}$ | $(004)_{10}$ | 094 |
| | | $B_{031}$ | $(006)_{10}$ | 095 |
| | 32nd color | $R_{032}$ | $(011)_{10}$ | 096 |
| | | $G_{032}$ | $(113)_{10}$ | 097 |
| | | $B_{032}$ | $(111)_{10}$ | 098 |
| | 33rd color | $R_{033}$ | $(017)_{10}$ | 099 |
| | | $G_{033}$ | $(027)_{10}$ | 100 |
| | | $B_{033}$ | $(037)_{10}$ | 101 |
| | 34th color | $R_{034}$ | $(018)_{10}$ | 102 |
| | | $G_{034}$ | $(000)_{10}$ | 103 |
| | | $B_{034}$ | $(000)_{10}$ | 104 |
| | 35th color | $R_{035}$ | $(109)_{10}$ | 105 |
| | | $G_{035}$ | $(100)_{10}$ | 106 |
| | | $B_{035}$ | $(209)_{10}$ | 107 |
| | 36th color | $R_{036}$ | $(255)_{10}$ | 108 |
| | | $G_{036}$ | $(090)_{10}$ | 109 |
| | | $B_{036}$ | $(100)_{10}$ | 110 |
| | 37th color | $R_{037}$ | $(255)_{10}$ | 111 |
| | | $G_{037}$ | $(095)_{10}$ | 112 |
| | | $B_{037}$ | $(110)_{10}$ | 113 |
| | 38th color | $R_{038}$ | $(255)_{10}$ | 114 |
| | | $G_{038}$ | $(090)_{10}$ | 115 |

| Data Type | | | Decimal Data | Byte Sequence |
|---|---|---|---|---|
| | | $B_{038}$ | $(120)_{10}$ | 116 |
| | $39^{th}$ color | $R_{039}$ | $(255)_{10}$ | 117 |
| | | $G_{039}$ | $(090)_{10}$ | 118 |
| | | $B_{039}$ | $(130)_{10}$ | 119 |
| | Colors from $40^{th}$ to $128^{th}$ are fulfilled with null values. | ⋮ | ⋮ | ⋮ |
| | | $R_{127}$ | null | 381 |
| | | $G_{127}$ | null | 382 |
| | | $B_{127}$ | null | 383 |
| | | $R_{128}$ | null | 384 |
| | | $G_{128}$ | null | 385 |
| | | $B_{128}$ | null | **128*3+2=386** |
| Data Part<br>Pointer references for the first 15 pixels of the block.<br>In terms of simplicity, eight-bit-reference pointers are viewed instead of a seven-bit-reference. Hence, the number of the total bytes is multiplied by (7/8). | $1^{st}$ byte = $P_1$ | | $(00)_{10}$ | 387 |
| | $2^{nd}$ byte = $P_2$ | | $(01)_{10}$ | 388 |
| | $3^{rd}$ byte = $P_3$ | | $(01)_{10}$ | 389 |
| | $4^{th}$ byte = $P_4$ | | $(02)_{10}$ | 390 |
| | $5^{th}$ byte = $P_5$ | | $(03)_{10}$ | 391 |
| | $6^{th}$ byte = $P_6$ | | $(00)_{10}$ | 392 |
| | $7^{th}$ byte = $P_7$ | | $(01)_{10}$ | 393 |
| | $8^{th}$ byte = $P_8$ | | $(10)_{10}$ | 394 |
| | $9^{th}$ byte = $P_9$ | | $(00)_{10}$ | 395 |
| | $10^{th}$ byte = $P_{10}$ | | $(23)_{10}$ | 396 |
| | $11^{th}$ byte = $P_{11}$ | | $(01)_{10}$ | 397 |
| | $12^{th}$ byte = $P_{12}$ | | $(00)_{10}$ | 398 |
| | $13^{th}$ byte = $P_{13}$ | | $(20)_{10}$ | 399 |
| | $14^{th}$ byte = $P_{14}$ | | $(28)_{10}$ | 400 |
| | $15^{th}$ byte = $P_{15}$ | | $(20)_{10}$ | 401 |
| | $16^{th}$ byte = $P_{16}$ | | $(33)_{10}$ | 402 |
| | $17^{th}$ byte = $P_{17}$ | | $(33)_{10}$ | 403 |
| | $18^{th}$ byte = $P_{18}$ | | $(33)_{10}$ | 404 |
| | $19^{th}$ byte = $P_{19}$ | | $(34)_{10}$ | 405 |
| | $20^{th}$ byte = $P_{20}$ | | $(34)_{10}$ | 406 |
| The remaining 12268 pixels | ⋮ | ⋮ | ⋮ | ⋮ |
| | $12288^{th}$ byte = ( $P_{12288}$ ) | | …… | **4096*7/8+386=3970** |

**Appendix B: An example of the "T5" blocks (129-256 colors)**

This example is viewed to clarify the class of type "**T5**". Suppose that a square block (I, J) = (34, 50) of 64x64 pixels in size has the CST that is viewed in Table 18. This table is achieved based on Table 1 and as an output of Algorithm II of Figure 8. Since the size of the block is 64x64=4096 pixels, the total number of all the colors is 4096 pixels.

Note how these 4096 pixels are distributed among 145 three-RGB-component colors and they are ordered starting from the color that has the highest frequency (here white). Because the three RGB-component values of each row of this table are the same, all the pixels of this range are considered as grey pixels. Since this CST has 145 grey colors, it is of type "**T5**". Now, assume that the first twenty-five pixels of this block contain the following three-RGB-component colors:

| | | | | |
|---|---|---|---|---|
| $(255,255,255)_{10}$ | $(000,000,000)_{10}$ | $(000,000,000)_{10}$ | $(077,077,077)_{10}$ | $(167,167,167)_{10}$ |
| $(255,255,255)_{10}$ | $(000,000,000)_{10}$ | $(013,013,013)_{10}$ | $(255,255,255)_{10}$ | $(153,153,153)_{10}$ |
| $(000,000,000)_{10}$ | $(255,255,255)_{10}$ | $(140,140,140)_{10}$ | $(109,109,109)_{10}$ | $(178,178,178)_{10}$ |
| $(140,140,140)_{10}$ | $(001,001,001)_{10}$ | $(109,109,109)_{10}$ | $(140,140,140)_{10}$ | $(140,140,140)_{10}$ |
| $(208,208,208)_{10}$ | $(208,208,208)_{10}$ | $(208,208,208)_{10}$ | $(208,208,208)_{10}$ | $(208,208,208)_{10}$ |

Table 18. A CST example of 145 colors (data in decimal)

| | Red component | Green component | Blue component | Frequency |
|---|---|---|---|---|
| 0. | 255 | 255 | 255 | 206 |
| 1. | 000 | 000 | 000 | 201 |
| 2. | 077 | 077 | 077 | 200 |
| 3. | 167 | 167 | 167 | 189 |
| 4. | 155 | 155 | 155 | 188 |
| 5. | 100 | 100 | 100 | 180 |
| 6. | 007 | 007 | 007 | 177 |
| 7. | 099 | 099 | 099 | 163 |
| 8. | 004 | 004 | 004 | 163 |
| 9. | 012 | 012 | 012 | 140 |
| 10. | 013 | 013 | 013 | 139 |
| 11. | 225 | 225 | 225 | 137 |
| 12. | 140 | 140 | 140 | 137 |
| 13. | 189 | 189 | 189 | 132 |
| 14. | 208 | 208 | 208 | 123 |
| 15. | 124 | 124 | 124 | 121 |
| 16. | 199 | 199 | 199 | 119 |
| 17. | 198 | 198 | 198 | 119 |
| 18. | 197 | 197 | 197 | 118 |
| 19. | 240 | 240 | 240 | 114 |
| 20. | 241 | 241 | 241 | 113 |
| 21. | 244 | 244 | 244 | 77 |
| 22. | 102 | 102 | 102 | 67 |
| 23. | 153 | 153 | 153 | 58 |
| 24. | 104 | 104 | 104 | 58 |
| 25. | 154 | 154 | 154 | 55 |
| 26. | 217 | 217 | 217 | 38 |
| 27. | 178 | 178 | 178 | 22 |
| 28. | 233 | 233 | 233 | 21 |
| 29. | 001 | 001 | 001 | 14 |
| 30. | 002 | 002 | 002 | 14 |
| 31. | 011 | 011 | 011 | 14 |
| 32. | 017 | 017 | 017 | 13 |
| 33. | 018 | 018 | 018 | 13 |
| 34. | 109 | 109 | 109 | 13 |
| ⋮ | ⋮⋮⋮ | ⋮⋮⋮ | ⋮⋮⋮ | ⋮⋮⋮ |
| ⋮ | ⋮⋮⋮ | ⋮⋮⋮ | ⋮⋮⋮ | ⋮⋮⋮ |
| 130. | 251 | 251 | 251 | 5 |
| 131. | 222 | 222 | 222 | 3 |
| 132. | 224 | 224 | 224 | 3 |
| 133. | 205 | 205 | 205 | 3 |
| 134. | 223 | 223 | 223 | 3 |
| 135. | 229 | 229 | 229 | 3 |
| 136. | 003 | 011 | 063 | 3 |
| 137. | 004 | 010 | 064 | 3 |
| 138. | 006 | 011 | 066 | 3 |
| 139. | 004 | 012 | 067 | 3 |
| 140. | 005 | 015 | 061 | 3 |
| 141. | 235 | 235 | 235 | 2 |
| 142. | 245 | 245 | 245 | 2 |
| 143. | 211 | 211 | 211 | 2 |
| 144. | 219 | 219 | 219 | 2 |
| | | | **Total** | **4096** |

Based on the number of detected colors that are viewed in Table 18, the type of the block involved in this example is "**T5**". Table 19 illustrates the data schematic construction of this example where a total of 4098 bytes are required for each block of this type and size. For the remaining 4071 pixels, other than these twenty-five pixels, the same pattern is utilized. With regard to the aforesaid discussion, the researcher concludes that:

> Rather than using 12288 (i.e. 64 * 64 * 3) bytes to store this three-RGB-component-color block of (64x64) pixels in size, 4098 bytes are enough. To be specific:
>
> **4098 / (64 * 64 * 3) * 100% = 33.349%**
>
> Another time, this means that this proposed technique is capable to encode this class of blocks by using only **33.349%** of the block size. If Equation 9 is recalculated by using BL=64, the same result will be achieved which means that the proposed algorithm is an efficient alternative for this class of blocks and, above all, this outcome shows a rapprochement between the theorizing (i.e. Equations 9) and the empirically-based results.

Table 19. A data structure example for the blocks of type "**T5**"

| **Data Type** | | **Data in Decimal** | **Byte Sequence** |
|---|---|---|---|
| Block address | I | $(034)_{10}$ | 001 |
| | J | $(050)_{10}$ | 002 |
| Dictionary of Colors | | Not needed | |
| Data Part<br>Every pixel needs only one byte to be stored. There is no special-purpose dictionary. | $1^{st}$ byte = $P_1$ | $(255)_{10}$ | 003 |
| | $2^{nd}$ byte = $P_2$ | $(000)_{10}$ | 004 |
| | $3^{rd}$ byte = $P_3$ | $(000)_{10}$ | 005 |
| | $4^{th}$ byte = $P_4$ | $(077)_{10}$ | 006 |
| | $5^{th}$ byte = $P_5$ | $(167)_{10}$ | 007 |
| | $6^{th}$ byte = $P_6$ | $(255)_{10}$ | 008 |
| | $7^{th}$ byte = $P_7$ | $(000)_{10}$ | 009 |
| | $8^{th}$ byte = $P_8$ | $(013)_{10}$ | 010 |
| | $9^{th}$ byte = $P_9$ | $(255)_{10}$ | 011 |
| | $10^{th}$ byte = $P_{10}$ | $(153)_{10}$ | 012 |
| | $11^{th}$ byte = $P_{11}$ | $(000)_{10}$ | 013 |
| | $12^{th}$ byte = $P_{12}$ | $(255)_{10}$ | 014 |
| | $13^{th}$ byte = $P_{13}$ | $(140)_{10}$ | 015 |
| | $14^{th}$ byte = $P_{14}$ | $(109)_{10}$ | 016 |
| | $15^{th}$ byte = $P_{15}$ | $(178)_{10}$ | 017 |
| | $16^{th}$ byte = $P_{16}$ | $(140)_{10}$ | 018 |
| | $17^{th}$ byte = $P_{17}$ | $(001)_{10}$ | 019 |
| | $18^{th}$ byte = $P_{18}$ | $(109)_{10}$ | 020 |
| | $19^{th}$ byte = $P_{19}$ | $(140)_{10}$ | 021 |
| | $20^{th}$ byte = $P_{20}$ | $(140)_{10}$ | 022 |
| | $21^{st}$ byte = $P_{21}$ | $(208)_{10}$ | 023 |
| | $22^{nd}$ byte = $P_{22}$ | $(208)_{10}$ | 024 |
| | $23^{rd}$ byte = $P_{23}$ | $(208)_{10}$ | 025 |
| | $24^{th}$ byte = $P_{24}$ | $(208)_{10}$ | 026 |
| | $25^{th}$ byte = $P_{25}$ | $(208)_{10}$ | 027 |
| | ⋮ ⋮ ⋮ | ⋮ ⋮ ⋮ | ⋮ ⋮ ⋮ |
| | $4096^{th}$ byte = ( $P_{4096}$ ) | …… | **4096 + 2 =4098** |

**Appendix C. An example of the "T6" blocks (more than 256 colors)**

This example is reported here in order to understand how the blocks of class type "**T6**" are fabricated. Assume that the square block (I, J) = (27, 57) of 195x195=38025 pixels in size has a color table with more than 256 colors. Suppose that the first fifteen pixels of this block contain the following color components RGB.

| | | | | |
|---|---|---|---|---|
| $(209,211,239)_{10}$ | $(255,255,254)_{10}$ | $(020,050,090)_{10}$ | $(254,054,253)_{10}$ | $(055,055,055)_{10}$ |
| $(101,101,133)_{10}$ | $(003,009,255)_{10}$ | $(255,255,255)_{10}$ | $(255,255,255)_{10}$ | $(101,101,133)_{10}$ |
| $(000,000,000)_{10}$ | $(255,255,255)_{10}$ | $(199,188,191)_{10}$ | $(197,180,193)_{10}$ | $(192,187,189)_{10}$ |

Inspired by the number of colors detected in this block, this technique treats this block as a class "**T6**". Hence, Table 20 illustrates the data schematic construction of this example where a total of 114077 bytes are required for each block of this type and size. This means that there are only two bytes more than the original size of the block. For the remaining 38010 pixels, other than these fifteen pixels, the same pattern is utilized. Based on the above-mentioned discussion, the researcher can conclude that:

> Rather than using **114075** (i.e. 195 * 195 * 3 * 3) bytes in storing this block of (195x195) pixels in size, **114077** bytes are required. This is due to the fact that additional two-byte storage is required in terms of storing the block address. Namely:
>
> **114077 / (195 * 195 * 3 + 2) * 100% = 0.00175%**
>
> Another time, this means that this proposed technique needs only **0.00175%** as an extra space to store this block. This is relatively very small and can be ignored at the expense of the other worthy percentages. If Equation 10 is recalculated by using BL=195, the same result will be achieved. And so, this outcome is consistent with the theorizing (Equation 10) without any discrepancies.

Table 20. A data structure example for the blocks of type "**T6**"

| Data Type | | | Data in Decimal | Byte Sequence |
|---|---|---|---|---|
| Block address | | I | $(027)_{10}$ | 01 |
| | | J | $(057)_{10}$ | 02 |
| Dictionary of Colors | | | Not needed. | |
| Data Part<br>Every pixel of the block takes three bytes.<br>Just the first 15 pixels of the block are shown here. There is no special-purpose dictionary. | 1st Pixel | $R_{001}$ | $(209)_{10}$ | 03 |
| | | $G_{001}$ | $(211)_{10}$ | 04 |
| | | $B_{001}$ | $(239)_{10}$ | 05 |
| | 2nd Pixel | $R_{002}$ | $(255)_{10}$ | 06 |
| | | $G_{002}$ | $(255)_{10}$ | 07 |
| | | $B_{002}$ | $(254)_{10}$ | 08 |
| | 3rd Pixel | $R_{003}$ | $(020)_{10}$ | 09 |
| | | $G_{003}$ | $(050)_{10}$ | 10 |
| | | $B_{003}$ | $(090)_{10}$ | 11 |
| | 4th Pixel | $R_{016}$ | $(254)_{10}$ | 12 |
| | | $G_{016}$ | $(054)_{10}$ | 13 |
| | | $B_{016}$ | $(253)_{10}$ | 14 |
| | 5th Pixel | $R_{005}$ | $(055)_{10}$ | 15 |
| | | $G_{005}$ | $(055)_{10}$ | 16 |
| | | $B_{005}$ | $(055)_{10}$ | 17 |
| | 6th Pixel | $R_{006}$ | $(101)_{10}$ | 18 |
| | | $G_{006}$ | $(101)_{10}$ | 19 |
| | | $B_{006}$ | $(133)_{10}$ | 20 |
| | 7th Pixel | $R_{007}$ | $(003)_{10}$ | 21 |
| | | $G_{007}$ | $(009)_{10}$ | 22 |
| | | $B_{007}$ | $(255)_{10}$ | 23 |

| Data Type | | | Data in Decimal | Byte Sequence |
|---|---|---|---|---|
| | 8th Pixel | $R_{008}$ | $(255)_{10}$ | 24 |
| | | $G_{008}$ | $(255)_{10}$ | 25 |
| | | $B_{008}$ | $(255)_{10}$ | 26 |
| | 9th Pixel | $R_{009}$ | $(255)_{10}$ | 27 |
| | | $G_{009}$ | $(255)_{10}$ | 28 |
| | | $B_{009}$ | $(255)_{10}$ | 29 |
| | 10th Pixel | $R_{010}$ | $(101)_{10}$ | 30 |
| | | $G_{010}$ | $(101)_{10}$ | 31 |
| | | $B_{010}$ | $(133)_{10}$ | 32 |
| | 11th Pixel | $R_{011}$ | $(000)_{10}$ | 33 |
| | | $G_{011}$ | $(000)_{10}$ | 34 |
| | | $B_{011}$ | $(000)_{10}$ | 35 |
| | 12th Pixel | $R_{012}$ | $(255)_{10}$ | 36 |
| | | $G_{012}$ | $(255)_{10}$ | 37 |
| | | $B_{012}$ | $(255)_{10}$ | 38 |
| | 13th Pixel | $R_{013}$ | $(199)_{10}$ | 39 |
| | | $G_{013}$ | $(188)_{10}$ | 40 |
| | | $B_{013}$ | $(191)_{10}$ | 41 |
| | 14th Pixel | $R_{014}$ | $(197)_{10}$ | 42 |
| | | $G_{014}$ | $(180)_{10}$ | 43 |
| | | $B_{014}$ | $(193)_{10}$ | 44 |
| | 15th Pixel | $R_{015}$ | $(192)_{10}$ | 45 |
| | | $G_{015}$ | $(187)_{10}$ | 46 |
| | | $B_{015}$ | $(189)_{10}$ | 47 |
| The remaining 38010 pixels | ⋮ ⋮ ⋮ | ⋮ ⋮ ⋮ | ⋮ ⋮ ⋮ | ⋮ ⋮ ⋮ |
| | Pixel 38025 | …… | …… | **38025*3+2=114077** |