1 Introduction

This document is intended to define formally the Polyomino Compressed Image Format (PCIF). It will often refer for clearness to the actual implementation code available at www.researchandtechnology.net/pcif/. To obtain the same results as the implemented algorithm, some choices that do not depend upon the file format must be implemented opportunely, as the choice of filters used in the first phase of the algorithm.

This document should actually be treated as an pre-pre-alpha/development release. You can report problems, errors or ask for further explanations contacting the author at www.researchandtechnology.net/pcif/contacts.php.

This document is structured by describing first the encoding of the simpler structures, and proceeding in a bottom-up fashion up to the file specification itself. First of all, an overview of the compression algorithm is given.

2 Overview of the algorithm

The domain of the algorithm are images of 24-bit color depth. In the following, the three primary colors will be numbered as: 0 for blue, 1 for red, 2 for green. For what regards orientation, we will refer to the point of a matrix (0,0) to be the leftmost bottom point.

The algorithm is defined in three main steps: filtering, decomposition and actual compression. In the first step the data is not reduced in dimension, but its values are changed to obtain a better distribution. In the second step the matrix representing the samples is decomposed in 24 matrices containing only binary values. In the third step these so said layers are compressed separately one from another.

2.1 Filtering

The filtering phase is formed by three substeps:

- The image is scanned, starting from the rightmost samples of the top row, and proceeding from high to low and then from right to left. For every sample an estimation is done, based on the samples that have not been scanned yet. The sample is replaced with its difference with the estimated value module 256. For different zones of the image different filtering functions may be applied.

- The image is scanned again, in any order that will respect the inverse color ordering: for a given pixel, scan before the red sample, then the green sample, and finally the blue sample. For the green and the red sample an estimation is done, based on the values of the samples of the same pixel for of the other colors. Again the sample is replaced with its difference from the estimate module 256.

- Every possible value of the matrix is mapped into another according to a permutation array. This serves to define how the following decomposition phase will act.

Code reference

Filter application: class BmpImage, method applyFilter, line 1102
2.2 Decomposition

After the filtering phase, the decomposition phase takes effect:

- The matrix with the image samples is decomposed into 24 layers containing only binary values. These are numbered as follows: 0-2 represent the least significative bits respectively of blue, green and red. 3-5 represent the second least significative bit, and so on.

At this point, the format allows an operation of layer comparison that has been experimentally tried to improve compression. Since this slows noticeably the encoding and decoding process, and does not usually give any compression gain, it is highly unrecommended. For the same reason, it is actually undocumented.

2.3 Layer compression

Finally, for every so obtained layer, one of the following techniques is applied:

- Compression through polyominoes: The layer is encoded through a series of integers representing the values of 0, and through a series of hv-convex polyominoes that represent the values of 1.
- Compression through RLE and Huffman coding: The layer is encoded through Run Length Encoding and Huffman trees.
- As is storage: The layer is encoded 'as is', without compression. This is useful for very chaotic layers.

3 Filtering functions

In the filtering phase, for the estimation of the point $x$, the four point $a$, $b$, $c$ and $d$ shown in figure 1 are used. The algorithm can apply for every zone of $n \times n$ pixels a different function, where $n$ must be a multiple of 2 smaller than 256. The numbering has been done to order the filters from the most useful (0) to the least; the format allows to specify that only the first $k$ filters are used in the file, allowing to save space in the encoding of the filters in the compressed bitstream. The possible functions to apply are the following:
1. $f(x) = 0$
2. $f(x) = a$
4. $f(x) = b$
5. $f(x) = c$
7. $f(x) = d$
8. $f(x) = a + (b - c)/2$
9. $f(x) = b + (a - c)/2$
3. $f(x) = (a + b)/2$
10. $f(x) = (a + b + c + d + 1)/4$
11. $f(x) = (a + d)/2$

6. $f(x) = a + b - c$ if $0 \leq a + b - c < 256$
   $f(x) = 0$ if $a + b - c < 0$
   $f(x) = 255$ if $a + b - c > 255$

12. $f(x)$ if the Paeth filter, defined by the following code:

   ```c
   p = a + b - c;
   pa = abs(p - a);
   pb = abs(p - b);
   pc = abs(p - c);
   if pa <= pb and pa <= pc then f(x) = a
     else if pb <= pc then f(x) = b
     else return f(x) = c;
   ```

0. $f(x)$ defined by the following:

   If $a \leq c \leq b$, $f(x)$ is equal to the linear filter,
   otherwise $f(x)$ is equal to the Paeth filter

If any point of a filter refers to a sample 'external' to the image, the estimation of that point is replaced by the following:

- $f(x) = a$ if available, otherwise
- $f(x) = b$ if available, otherwise
\[ f(x) = 0 \]

The color filters are as above appliable differently to different zones, also for a different \( n \) than the one chosen for the standard filters. The possible color filters used are the following; \( p[x] \) is the point to estimate, and \( p[i] \) is the sample relative to color \( i \).

1. \( f(p[x]) = 0 \)
   
   0. If \( x = 0 \) then \( f(p[x]) = 0 \), else \( f(p[x]) = p[x - 1] \)
   
   2. If \( x = 0 \) then \( f(p[x]) = 0 \), else \( f(p[x]) = p[0] \)

Code reference

Filtering of a sample: class BmpImage, method filter, line 1757
Color filtering of a sample: class BmpImage, method colorFilter, line 1994

4 Numerical coding

In this section the coding of integer values in the PCIF format will be described. First of all, a description of how bit matrices are handled is given.

4.1 Bit matrices

In the algorithm there is often the necessity to handle matrices of bits, on a side for the layers and on the other for the sequencial writing of the bits on the compressed bitstream. These two cases are treated with the same abstraction. Since the architecture of a computer usually allows to save and store bytes and not bits, the following correspondence will be used: when writing bits sequencially in a compressed stream, the first bits to be written will be those relative to the least significative of the relative byte. A correspondence between a bitstream and the bytes wrote into a file can be seen in table 1

| Bit stream | 11010110010010110011001001101010 |
| Byte stream | 107 | 210 | 76 | 86 |

Table 1: Bit streams and byte streams

4.2 Coding in a fixed format

The simpliest coding in the PCIF format is the coding at a fixed number of bits: an integer is encoded in its bit representation from the least to the most significative bit. The integer is encoded always in \( n \) bits for a given \( n \). Below a table of example where the written bitstream is shown in function to the encoded values.

Code reference

Fixed bit size coding: class bitMatrix, method fwrite, line 585
4.3 Coding in a fixed domain

The coding in fixed domain is used to encode an integer that the decoder will know to be less than $k$. In this case the bits representing the value are written from the least significant to the most, but the encoding or decoding process immediately stops if the next bit that could be written could cause the value to be $\geq k$. The coding in a fixed domain can be done also for values $\geq k'$ and $< k''$; in this case the algorithm will encode $n - k'$ with the limit $k'' - k'$. Below as an example a table of the possible codings in the interval $[0, 6]$.

<table>
<thead>
<tr>
<th>Value</th>
<th>Given fixed bit number</th>
<th>Written bits</th>
<th>Complete written stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>110</td>
<td>111110</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>1011</td>
<td>11111010111</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>01</td>
<td>111110101101</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>11111010101010</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>1110</td>
<td>1111101011011011110</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0100</td>
<td>11111010110110111100100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>11111010110111001001001</td>
</tr>
</tbody>
</table>

Table 2: Result of some fixed size integer encodings

This technique is used to code rapidly values for whom we do not want to build an Huffman tree, and anyway save some bits. It can be extended also to the case of encoding an array $[a_0 \ldots a_l]$ given an array $[m_0 \ldots m_l]$ such that $a_i < m_i$ for every $i$. The algorithm reconducts this case to the previous thanks to the formulas:

$$n = \sum_{i=0}^{l} a_i \prod_{j=0}^{i-1} (m_j)$$

$$k = \prod_{i=0}^{l} m_i$$

An example is in table 4.

<table>
<thead>
<tr>
<th>Encoding in the domain $[0, 6]$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Table 3: Encoding in the domain $[0, 6]$
4.4 Coding in variable bit sizes

The coding in variable bit sizes is done to write values that are virtually unlimited. In this case we suppose that smaller values are less probable, and encode every value with a bit length proportional to its minimal bit representation length. Given the value to encode $n$ and a value $k$ said ‘bit base’, the encoding algorithm is defined as the following:

1. Initialize a value $b = k$
2. While $n \geq 2^b$, impose $n = n - 2^b$ and $b = b + 1$
3. Write $b - k$ bits of value 1, and then a zero
4. Write $n$ at a fixed bit number of $b$ bits

The decoding process consists of the following steps that reobtain the original value $n$:

1. Initialize $b = k$ and $n = 0$
2. While the read bit is equal to 1 impose $n = n + 2^b$ and $b = b + 1$
3. Read an integer at a fixed bit number size of $b$ bits, and sum it to $n$

A pair of examples of variable bit coding are in tables 5 and 6.

Code reference

Variable bit coding: class bitMatrix, method writeVbit, line 658
Variable bit decoding: class bitMatrix, method readVbit, line 680

5 Huffman tree coding

The encoding of an Huffmann tree is defined by the following algorithm; a value $n$ is assumed to be known, such that every simbol in the tree in a positive integer $\leq n$. 

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Array & Associated N & Coding \\
\hline
[0,0,0] & 0 & 0000 \\
[1,0,0] & 1 & 1000 \\
[0,1,0] & 2 & 0100 \\
[1,1,0] & 3 & 1100 \\
[0,2,0] & 4 & 0010 \\
[1,2,0] & 5 & 101 \\
[0,0,1] & 6 & 011 \\
[1,0,1] & 7 & 111 \\
[0,1,1] & 8 & 0001 \\
[1,1,1] & 9 & 1001 \\
[0,2,1] & 10 & 0101 \\
[1,2,1] & 11 & 1101 \\
\hline
\end{tabular}
\caption{Array coding with limitations (2,3,2)}
\end{table}
### Table 5: Variable bit coding with bit base = 0

<table>
<thead>
<tr>
<th>Value</th>
<th>Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
</tr>
<tr>
<td>3</td>
<td>11000</td>
</tr>
<tr>
<td>4</td>
<td>11010</td>
</tr>
<tr>
<td>5</td>
<td>11001</td>
</tr>
<tr>
<td>6</td>
<td>11011</td>
</tr>
<tr>
<td>7</td>
<td>1110000</td>
</tr>
<tr>
<td>8</td>
<td>1110100</td>
</tr>
<tr>
<td>9</td>
<td>1110010</td>
</tr>
<tr>
<td>10</td>
<td>1110110</td>
</tr>
<tr>
<td>11</td>
<td>1110001</td>
</tr>
<tr>
<td>12</td>
<td>1110101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Table 6: Variable bit coding with bit base = 2

<table>
<thead>
<tr>
<th>Value</th>
<th>Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
</tr>
<tr>
<td>2</td>
<td>001</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>10000</td>
</tr>
<tr>
<td>5</td>
<td>10100</td>
</tr>
<tr>
<td>6</td>
<td>10010</td>
</tr>
<tr>
<td>7</td>
<td>10110</td>
</tr>
<tr>
<td>8</td>
<td>10001</td>
</tr>
<tr>
<td>9</td>
<td>10101</td>
</tr>
<tr>
<td>10</td>
<td>10011</td>
</tr>
<tr>
<td>11</td>
<td>10111</td>
</tr>
<tr>
<td>12</td>
<td>1100000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6: Variable bit coding with bit base = 2
1. If the tree is formed by an only leaf, write the bit 1 and the simbol associated with a fixed bit number of \([\log_2 n]\).

2. Otherwise, write the bit 0, encode recursively the left son and then the right son.

The bit encoding for a simbol will be given by the path from the root to the leaf containing the simbol itself. For convention, left sons represent the bit 0, and right one represent the bit 1.

**Code reference**

**Huffman tree coding**: class HTree, method write, line 256

---

6  **Layer encoding through RLE and Huffman**

First of all, two possible methods to scan the layer are defined:

1. From left to right, and then from low to high.

2. Start from the bottom left sample, and iterate the following path (see Figure 2):
   2a. Proceed one sample up
   2b. Proceed one sample right
   2c. Proceed one sample down
   2d. Proceed one sample right
   2e. At the end of a pair of rows, proceed with the lowest leftmost non scanned pixel

![Figure 2: An example of the visiting order 2](image)

The encoding of a layer with this technique consist in the following steps:

1. Scan the layer with the selected scan order, and process the encountered bits with Run Length Encoding. Consider the first sequence to be of zeros (if not, add a 'sequence of zero zeros' at the beginning).

2. Create a series of simbols by coupling the obtained values: the first with the second, the third with the fourth, and so on.

   2a. If in these couples the first value exceeds 63 or the second exceeds 15, split the couple to obtain this condition (ex. \((100, 10)\) becomes \((63, 0)\) and \((37, 10)\)).
3. Encode the obtained couples with an Huffman tree. Since Huffman trees have been
defined for integers, assign to each couple \((a, b)\) an integer value \(v = a \cdot 16 + b\).

**Code reference**

Layer Huffman-RLE coding: class Layer, method writeHuffmanCompressed, line 438

---

7 Coding with polyominoes

In this section the coding through polyominoes will be described. At first the encoding of
a single polyomino, and then of an entire layer through the use of polyominoes.

7.1 Coding of a polyomino

The algorithm makes use of a convenient encoding for hv-convex polyominoes. In respect
to the standard definition of a polyomino, the values of '1' will be considered to be cells.

In an hv-convex polyomino

- All the ones must be connected
- Every one of its columns must be connected
- Every one of its rows must be connected

7.1.1 Step 1 of 3: encoding of the polyomino’s dimensions

The polyomino’s dimensions are defined by the width and the height of the minimal
rectangle with sides parallel to the axes that contain the polyomino. Say \(w\) the width and
\(h\) the height. Execute the following:

a. If \(w < 8, h < 8\) but at least one of them is \(\neq 8\), encode the value \((h - 1) \cdot 8 + w - 1\)
with an Huffman tree.

b. Otherwise, write 63 with the Huffman tree, and write \(w - 1\) in the variable bit format
with bit base equal to 3.

   b1. If the written value is less than 7, write \(h - 1\) with a variable bit number and
   bit base equal to 3.

   b2. Otherwise write \(h - 1\) with a variable bit number and bit base equal to 0.

If \(w > h\), rotate the polyomino of 90 degrees counterclockwise.

7.1.2 Step 2 of 3: encoding of the polyomino’s feet

The feet of the polyomino are defined as the first (looking from left to right) values equal
to one that lie on the upper and lower size of the polyomino. Consider the distance of
these values from the left side of the bounding rectangle; as in the following, the distance
of a cell from a side of the bounding rectangle is defined as the number of values (ones or
zeros) that lay between these two objects. The array formed by these two values (the one
on the inferior side first) is encoded in a limited range where the limiting array has two
elements both equal to the polyomino’s width.
7.1.3 Step 3 of 3: encoding of the polyomino’s body

The remaining information about the polyomino will be coded through a series of integers that represent the first and last value with a one in every column. For the convexity constraints, this is the only information that we need to rebuild all the values in the column.

Since every time that a column is determined some constraints can be defined on the other ones, we can use them to save some bits in the coding of the other columns. We will call $minLow$ and $maxLow$ the arrays that limit the possible values of the distance of the lower ones from the bottom side of the bounding box; symmetrically, we will define $minHigh$ and $maxHigh$ for the higher side. Every encoding of the lowest (highest) one in a column will be done by encoding its distance from the lower (higher) side of the bounding rectangle and writing this value in a limited range defined by the arrays $minLow$ and $maxLow$ ($minHigh$ and $maxHigh$).

The constraints represented by these array will be computed by the following; $pLow$ e $pHigh$ are the positions of the feet as distance from the left side.

- Initialization:
  
  $minLow[i] = 0 \forall i$
  $minHigh[i] = 0 \forall i$
  $maxLow[i] = h - 1 \forall i$
  $maxHigh[i] = h - 1 \forall i$

- After feet encoding/decoding: Said $j$ the index of the lower foot $k$ the one of the upper, impose
  
  $maxLow[j] = 0$
  $maxHigh[k] = 0$

  and
  
  $minLow[i] = 1 \forall i < j$
  $minHigh[i] = 1 \forall i < k$

- Coding of a distance from the inferior side with column index greater than $pLow$: said $k$ the column index and $d$ the distance, impose for convexity
  
  $maxLow[i] = \min(maxLow[i], d) \forall pLow < i < k$
  $minLow[i] = \max(minLow[i], d) \forall k < i < l$

  and if $k > 0$ for connectiveness
  
  $maxHigh[k - 1] = \min(maxHigh[k - 1], h - d - 1)$

- Coding of a distance from the inferior side with column index less than $pLow$: said $k$ the column index and $d$ the distance, impose for convexity
  
  $maxLow[i] = \min(maxLow[i], d) \forall k < i < pLow$
  $minLow[i] = \max(minLow[i], d) \forall 0 \leq i < k$

  and if $k < w - 2$ for connectiveness
  
  $maxHigh[k + 1] = \min(maxHigh[k + 1], h - d - 1)$
• Coding of a distance from the upper side with column index greater than $p_{High}$: said $k$ the column index and $d$ the distance, impose for convexity

$$maxHigh[i] = \min(maxHigh[i], d) \quad \forall \quad p_{High} < i < k$$

$$minHigh[i] = \max(minHigh[i], d) \quad \forall \quad k < i < l$$

and if $k > 0$ for connectiveness

$$maxLow[k - 1] = \min(maxLow[k - 1], h - d - 1)$$

• Coding of a distance from the upper side with column index less than $p_{High}$: said $k$ the column index and $d$ the distance, impose for convexity

$$maxHigh[i] = \min(maxHigh[i], d) \quad \forall \quad k < i < p_{High}$$

$$minHigh[i] = \max(minHigh[i], d) \quad \forall \quad 0 \leq i < k$$

and if $k < w - 2$ for connectiveness

$$maxLow[k + 1] = \min(maxLow[k + 1], h - d - 1)$$

• After the coding of a column of index $k$, for a correct ordering impose

$$maxLow[k] = \min(maxLow[k], h - minHigh[k] - 1)$$

and

$$maxHigh[k] = \min(maxHigh[k], h - minLow[k] - 1)$$

The visiting order is defined by the following pseudo-code to optimize the quantity of information we obtain from these constraints. The initial call is done to $visit(0, w-1)$. Assume that $encode(i)$ encodes the values in column $i$, first the lower value one and then the higher value one.

$$visit(l, r) = \begin{cases} 
\text{encode}(l) \\
\text{encode}(r) \\
visit_{rec}(l + 1, r - 1) 
\end{cases}$$

$$visit_{rec}(l, r) = \begin{cases} 
\text{if } (l < r) \{ \text{med} = (l + r) / 2 \\
\text{encode(med) \\
\text{visit}_{rec}(l, \text{med} - 1) \\
\text{visit}_{rec}(\text{med} + 1, r) \} 
\end{cases}$$

An example of the polyomino coding is in figures 3 through 6. The shaded values are those who have been changed from the previous step.

**Code reference**

Polyomino coding, intestation: class Poly, method write, line 387
Polyomino’s body coding: class Poly, method writeGenPoly, line 681
Array limits handling: class Poly, methods adjustLimitsXX
7.2 Layer encoding with polyominoes

The encoding of a layer through polyominoes is done with a series of alternate encodings of distances, representing the polyominoes’ positions and the zeros of the layer, and encoding of polyominoes. Two Huffman trees will be used for every layer: one for the dimensions of
the polyominos, as defined previously, and one for the distances between them.

First of all we will define a set of known points, defined at a defined moment by the following:

<table>
<thead>
<tr>
<th>minHigh</th>
<th>maxHigh</th>
<th>distances</th>
<th>minLow</th>
<th>maxLow</th>
<th>distances</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 0 0 3</td>
<td>1 0 0 3 3</td>
<td>1 0 1 2 3</td>
<td>1 0 0 2 2</td>
<td>1 0 1 2 3</td>
<td>1 0 0 2 2</td>
</tr>
</tbody>
</table>

Figure 5: An example of polyomino coding (3)

<table>
<thead>
<tr>
<th>minHigh</th>
<th>maxHigh</th>
<th>distances</th>
<th>minLow</th>
<th>maxLow</th>
<th>distances</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0 1 1 3</td>
<td>1 0 1 1 3</td>
<td>1 0 1 2 3</td>
<td>1 0 0 2 2</td>
<td>1 0 1 2 3</td>
<td>1 0 0 2 2</td>
</tr>
</tbody>
</table>

Figure 6: An example of polyomino coding (4)
• Every value belonging to an already encoded polyomino is known.

• For every polyomino formed by an only value, the cells immediately above and to its right are known.

• For every polyomino that has width 1, the cell immediately above it is known.

• For every polyomino that has height 1, the cell immediately to the right of it is known.

In the following it will be clear that for every encoded polyomino these conditions must stand:

• Every polyomino with an only one will have a zero immediately to its right and over it.

• Every polyomino that has width 1 must have the value immediately above equal to zero.

• Every polyomino that has height 1 must have the value immediately to its right equal to zero.

At this point, the layer is scanned from left to right and then from low to high, and the following steps are executed for each value; we will maintain a variable $d$ initialized to zero to memorize the distances.

• If the current value is in the known values, do nothing.

• If the value is equal to zero and not known, then impose $d = d + 1$.

• If the value is equal to one and not known then
  – If $d < 63$, encode it with the apposite Huffman tree.
  – Otherwise ancode the number 63 with its Huffman code and then $d - 63$ in the variable bit coding with bit base equal to 6.
  – Encode in the compressed bitstream a polyomino that contains the encountered value of one.
  – Update information about the known points.

Code reference

Compression with polyominoes: class Layer, method writeCompressedLayer, line 846

8 Filter encoding

The information about which filters have been used for which zones must be encoded in the compressed file. Considering that the maximum index of the used filter is $n$ (this will be written in the file intestation), the indexes of the filters will be written with a series of fixed bit size codings, where every integer is written in $\lceil \log_2 n \rceil$ bits. The filters are written from the first to the last zone, where the first zone starts from the lower left point and the numbering proceeds from left to right and from low to high, as examplified in figure 7.
The bit size of the filters will be $[z \cdot \lceil \log_2 n \rceil / 8] \cdot 8$, where $z$ is the number of filtering zones of the image defined by $z = \lceil (w/d) \cdot (h/d) \rceil$, where $w$ and $h$ are the width and height of the image, and $d$ is the size of the side of a filtering zone, in pixels.

The color filters are encoded exactly with the same procedure, considering that these can eventually have different values for $n$ and $d$.

**Code reference**

Filter encoding: class BmpImage, method writeLineFilters, line 970

---

9 Remapping array encoding

The remapping array will be written into the compressed file as 256 bytes, each one of them representing the value that has been assigned to one of the possible initial values. The destination values will be written ordinately from the one relative to value 0 to the one referring to value 255. Each value will be represented as a byte.

**Code reference**

Remapping array encoding: class BmpImage, method writeMap, line 1932

---

10 Compressed layer encoding

This section will describe how a compressed layer is encoded in the compressed bitstream. Each field is written in a fixed bit size of the specified dimension.

- Layer start code, 16 bits: Two bytes of value 76 ed 89 are written into the bitstream. These correspond to the ASCII of 'LY' and are used to identify the start of each layer.
- Type, 2 bits: The compression type used for this layer. 0 represents a polyomino compressed layer, 1 an RLE/Huffman compressed layer and 2 a layer stored 'as is'.
- Width, 16 bits: The width, in points, of the layer.
- Height, 16 bits: The height, in points, of the layer.
- Color, 2 bits: The color to which the layer refers. Respects the usual convention: 0 for blue, 1 for green and 2 for red.
- Bit, 3 bits: The bit to which the layer refers to, where \( k \) represents the \( k \)-th less significative bit.

- Hash code, 32 bits: An hash code to verify the layer’s correct reconstruction. Defined by the formula \( \text{bit} + \text{color} + \sum_{i=0}^{n} (b[i] \cdot 2^{i \mod 16}) \mod 2^{30} \), where \( b \) is the stream of bytes corresponding to the representation in bytes of the bit matrix of the layer.

- Base layer, 5 bit: Code for layer comparison option, set to 0 if this phase has been skipped (as recommended). Other options are currently undocumented.

- Variable base layers, variable bit number: Field must be specified only if the previous field is equal to 27. This option is actually undocumented.

- Inversion, 1 bit: If equal to one, every value of the layer has been inverted before compression. This option is never used in the actual implementation, but is left in the format to allow different compression techniques.

- Zigzag, 1 bit: If the layer is compressed through RLE and Huffman coding, specifies the visiting order, where 0 represents the standard left-to-right/low-to-up scan order, while 1 refers to the zig-zag order. If the layer is compressed with other methods, this value is ignored and can be arbitrarily written.

- Number of polyominoes, 32 bits: If the layer is compressed through polyominoes, this value counts the number of polyominoes used in the layer. Used to check exactness of the encoding. For layers encoded with other methods, this field should be ignored by the decompressor.

After this intestation, the layer is encoded as the following:

- If the layer is compressed through polyominoes
  - The Huffman tree used for distances is encoded
  - The Huffman tree relative to the polyominoes’ dimensions is encoded
  - The alternate sequence of distances and polyominoes is encoded ad described previously

- In the case of RLE/Huffman compression
  - The used Huffman tree is encoded
  - The couples of RLE sequences are encoded, as described previously

- In the case of ’as is’ storage
  - The bits of the layer are written in the compressed bit stream, proceeding from the leftmost bottom point and proceeding from from low to high and then from left to right.

In all these three cases, there is no ’end of layer’ code: the encoding ends when all the values of the layer have been processed.

**Code reference**

- Intestation encoding: class Layer, method writeIntest, line 768
- Polyomino compressed layers: class Layer, method writeCompressedLayer, line 846
- RLE-Huffman compression: class Layer, method writeHuffmanCompressed, line 438
- As-is layer storage: class Layer, method writeLayer, line 728
11 PCIF file coding

The encoding of the PCIF file intestation is done in integer byte-size fields, allowing a simpler handling. The structure of a PCIF file is the following:

- Identification string, 57 bytes: The PCIF file format is identified by the initial string 'PCF, Polyomino Compressed Format. Author Stefano Brocchi.'

- Version, 1 byte: File version number. For the version described in this document (1.1) must have value 1.

- Sub version, 1 byte: File subversion number. For the version described in this document (1.1) must have value 1.

- Embedded BMP intestation, 54 bytes: The original BMP file intestation is embedded in the PCIF file. Since the algorithm currently works on 24-bit color images, this will always be of 54 bytes of lenght. In bytes 18...21 and 22...25 this contains the image’s width and height, from the least significative byte to the most.

- Hash code, 4 bytes: An integer that can be used to check the correct image reconstruction. Defined by the formula \( P_{w1} = 0 \sum_{i=0}^{w-1} P_{h1} = 0 \sum_{j=0}^{h-1} \sum_{k=0}^{2} m[i][j][k] \cdot (i + 1) \cdot (j + 1) \cdot (k + 1) \mod 2^{32} \), where \( m \) represents the matrix of samples relative to the image.

- Used filter number, 1 byte: This number, imposed to \( n \), is used to indicate that all used filters have index less than \( n \).

- Used color filters, 1 byte: As the previous field, this indicates the number of used color filters.

- Filter determination step, 1 byte: Integer that, imposed to \( n \), states that during compression \( 1/n \) points of the image have been used to chose the different zone filters. Information used only for informative purpose.

- Filtering zone dimension, 1 byte: The size in points of a side of a filtering zone.

- Color filtering zone dimension, 1 byte: The size in points of a side of a color filtering zone.

- Filter coding, variable byte size: Here the used filters are encoded as described previously. If the used byte number is not integer, this field shall be padded with zeros until this condition is accomplished.

- Color filter coding, variable byte size: Here the used color filters are encoded as described previously. If the used byte number is not integer, this field shall be padded as above.

- Remapping array, 256 bytes: Specifies the remapping array used during compression, as described above.

After the intestation, in the file there will be the 24 encoded layers, proceeding in the order blue-green red and then by bit number, from the more significative to the least.

**Code reference**

Entire image coding: class BmpImage, method writeCompressedImage, line 1031
**Intestation coding**: same method, lines 1048-1062

**Layer coding**: same method, lines 1063-1090