Hypergraph Lossless Image Compression

Alain BRETTO and Luc GILLIBERT
Université de Caen, GREYC Département d’informatique UMR-6072, Campus II,
Bd Marechal Juin BP 5186, 14032 Caen cedex, France.
alain.bretto@info.unicaen.fr, lgillibe@info.unicaen.fr

Abstract

Hypergraphs are a very powerful tool and can represent many problems. In this paper we define a new image representation based on hypergraphs. This representation conducts to a new lossless compression algorithm for images called HLC. We present the algorithm and give some experimental results proving its efficiency. Finally we show that this algorithm can be generalized to three-dimensional images and to parametric lossy compression.

1. Introduction

Image compression addresses the problem of reducing the amount of data needed to represent a digital image. This is done by the removal of redundant data contained in the image. This removal of data may be reversible (i.e. the removed data can be fully reconstructed from compressed data) or irreversible (i.e. the removed data can be only partially reconstructed from compressed data). The first kind of compression is called lossless compression, while the second kind of compression is called lossy compression [4]. The lossless image compression is used in fields where full restoration of original image from compressed one is crucial: business documents, where lossy compression is prohibited for legal reasons, satellite images, where the data loss is undesirable because of image collecting cost, and medical images, where difference in original image and uncompressed one can compromise diagnostic accuracy. In this paper we describe a new method for lossless image compression, based on hypergraphs and called HLC (Hypergraph Lossless Compression). The hypergraphs are a very interesting generalization of the graphs. Introduced in 1960 by C. BERGE [6], they are now used in many domains such as chemistry, engineering and image processing [1, 2, 3]. We give an algorithm making the conversion between a matrix-based representation and the hypergraph representation. We also present some experimental results proving that HLC, combined with a PPM-based [10] data compression algorithm, is very efficient. The resulting compression is better than GIF or PNG and can compete with LOCO-I/JPEG-LS. Finally we show that hypergraph compression can be generalized to lossy compression and to three-dimensional images.

2. Hypergraph image representation

Let \( V = \{x_1, x_2, \ldots, x_n\} \) be a finite set. A hypergraph on \( V \) is a family \( H = \{E_1, E_2, \ldots, E_m\} \) of subsets of \( V \) such that:
- \( E_i \neq \emptyset \) for \( i = 1, 2, \ldots, m \)
- \( \bigcup_{i=1}^{m} E_i = V \)

A simple hypergraph is a hypergraph \( H = \{E_1, E_2, \ldots, E_m\} \) such that \( E_i \subseteq E_j \Rightarrow i = j \). The elements \( x_1, x_2, \ldots, x_n \) are called the vertices, and the sets \( E_1, E_2, \ldots, E_m \) are called the hyperedges of the hypergraph. A partial hypergraph \( H' \) from an hypergraph \( H \) is an hypergraph such that \( H' \subset H \).

Let \( I \) be a matrix represented image. We build a hypergraph \( H(I) \), called the hypergraph representation of the image, as it follow:
- The vertices of \( H(I) \) are the pixel of the images.
- The hyperedges of \( H(I) \) are the maximal rectangles such that inside a rectangle all the pixels have the same color.

This is a simple hypergraph because, by construction, the hyperedges are maximal for the inclusion. Notice that, given the hypergraph representation of an image, the image can be integrally rebuild. So, for storing an image \( I \), storing hyperedges of \( H(I) \) is sufficient.

3. Hypergraph image compression

A rectangle can be stored as a couple of points. But such representation required several bytes. Let’s consider
the case of a one-pixel black/white checkerboard pattern. The hypergraph associated to this image will contain as many rectangles as there are pixels in the image and each rectangle will require more than one byte for storing it. So the rectangles hypergraph will be bigger than the original uncompressed image. For that reason, we introduce an integer $K$ and we use the following process for compressing and image $I$:

- Build $H(I)$, the hypergraph representation of $I$.
- Order the hyperedges of $H(I)$ following surface (the bigger rectangle comes first). If two rectangles have the same surface the first one is the first constructed. The ordered hyperedges are called \{R_1, \ldots, R_m\}. The ordered hypergraph is called $H_0(I)$.
- Extract from $H_0(I)$ a partial hypergraph $H_K(I) = \{R_x \in \{R_1, \ldots, R_m\}; x \in X\}$. The set of indices $X$ is chosen such that for all $x \in X$, $R_x$ contains at least $K$ pixels that are not in $\cup_{i \in X, i \neq x} R_i$ (X can be empty). The partial hypergraph $H_K(I)$ depends on $K$. One can note that if $K = 0$ then $H_K(I) = H_0(I)$.
- Store the hyperedges of $H_K(I)$. With a good representation between 3 and 9 bytes are required for a rectangle, plus the color.
- Create an empty data segment.
- Create an empty surface $S$ of the size of the image with one flag per pixel. Set all the flags to 0.
- Read the hypergraph $H_K(I)$ and draw all the rectangles on $S$. For each pixel drawn, set the corresponding flag to 1.
- Traverse the image and, for each flag set to 0, write the pixel color in the data segment. The colors are written linearly so no additional data are required.
- Compress the hypergraph and the data segment with a PPM-based algorithm. For this paper we chose an open source data compression program based on PPMd [9]. In fact a LZW-based algorithm can also give good results.

With that process it is possible to only store rectangles giving information on at least $K$ pixels. The other pixels of the image are stored in a data segment by a common way: one byte for one pixel (or 3 bytes if the image is RGB). The image is reconstituted with the following process:

- Expand the hypergraph and the data segment with the PPM-based algorithm.
- Create an empty surface $S$ of the size of the image with one flag per pixel. Set all the flags to 0.
- Read the hypergraph and draw all the rectangles on $S$. For each pixel drawn, set the corresponding flag to 1.
- Traverse the image and, for each flag set to 0, give to the pixel the color read in the data segment.

So, by example, if there is no rectangle with a surface at least equal to $K$ pixels, the HLC compressed image, before the PPM compression, will be an empty hypergraph followed by data segment equal to the original image. A nontrivial image example can be seen in table 1.

![Table 1](https://example.com/table1.png)

<table>
<thead>
<tr>
<th>Data</th>
<th>Hypergraph:</th>
<th>Hyperedges</th>
</tr>
</thead>
<tbody>
<tr>
<td>13,14,15,14,8</td>
<td>rect(1,1,3,2,10)</td>
<td>13,14,15,14,8</td>
</tr>
<tr>
<td>9,10,10,10,11</td>
<td></td>
<td>9,10,10,10,11</td>
</tr>
<tr>
<td>6,10,10,10,8</td>
<td></td>
<td>6,10,10,10,8</td>
</tr>
<tr>
<td>91,8,13,14,15</td>
<td></td>
<td>91,8,13,14,15</td>
</tr>
</tbody>
</table>

**Proposition 1** - The number of hyperedges in $H_K(I)$ is inferior or equal to the number of pixels of $I$ divided by $K$ if $K \geq 1$.

**Proof** - Let’s traverse the hypergraph $H_K(I) = \{R_x \in \{R_1, \ldots, R_m\}; x \in X\}$. By construction the set $X$ is chosen such that for each $x \in X$, $R_x$ contains at least $K$ pixels that are not in $\cup_{i \in X, i \neq x} R_i$. For each $x \in X$ we construct the following set:

$$A_x = R_x - R_x \cap (\bigcup_{i \in X, i < x} R_i)$$

If $x_1$ and $x_2$ are two elements of $X$, $x_1 \neq x_2$, then $A_{x_1} \cap A_{x_2} = \emptyset$. For each $x \in X$, $A_x$ contains at least $K$ pixels. So the cardinality of the set $P_K(I) = \{A_x; x \in X\}$ is inferior or equal to the cardinality of the set of pixels of $I$ divided by $K$. Moreover, there is a bijection between $P_K(I)$ and $H_K(I)$:

$$R_x \rightarrow A_x.$$  

So the number of hyperedges in $H_K(I)$ is inferior or equal to the number of pixels of $I$ divided by $K$.

Therefore, with $K$ bigger than the size of a rectangle (in bytes), storing an image hypergraph representation is more economic in memory than storing a matrix representation. There is no negative compression (except the added size of the header that contains the length of the hypergraph segment, this header is 4 bytes long). In the worst case, the image compressed with our HLC algorithm will be 4 bytes bigger than the original one.

**4. The algorithm**

In this code a hypergraph is a list of rectangles and an image is a table of integers. The function BuildHyper(I)
creates the hypergraph $H(I)$ associated to an image $I$. For each pixel $p$ with coordinates $(i, j)$ of the image $I$, the algorithm constructs the list of the maximal rectangles such that $p$ is their upper left corner. Then these rectangles are compared with the rectangles having $(i + 1, j)$ or $(i, j + 1)$ as upper left corner. So, the included rectangles can be deleted during the construction of the hypergraph. This function $\text{Build}_\text{hyper}$ returns a table of list of rectangles. So, for each pixel $p$, it is possible to access directly to the list of rectangles having the pixel $p$ as upper left corner. The function $\text{Build}_\text{rectangles}$ is used for the construction of the maximal rectangles having a pixel $p$ as upper left corner and is called in $\text{Build}_\text{hyper}$. The procedure $\text{Delete}_\text{inc}_\text{rect}(S_1, S_2)$ is also called to each loop in $\text{Build}_\text{hyper}$. Its goal is to compare two lists of rectangles, $S_1$ and $S_2$, and to delete all rectangles in $S_1$ that are included in at least on rectangle of $S_2$. In the function $\text{Build}_\text{hyper}(I)$ the image $I$ is traversed from its lower right corner. In the function $\text{Build}_\text{rectangles}$ the image $I$ is traversed from its upper left corner. The function $\text{Build}_\text{hyper}(I)$ can be written with the following pseudo code:

**Construction of the hypergraph representation of the image $I$.**

```plaintext
hypergraph[][] Build_hyper(image I)
  x = width of I;
  y = height of I;
  H = new hypergraph[x][y];
  for (i = y; i >= 0; i --)
    for (j = x; j >= 0; j --)
      if (j <= x) then
        H[i][j] = Build_rectangles(I,i,j);
      end if;
      if (i < y) then
        Delete_inc_rect(H[i][j + 1], H[i][j]);
      end if;
      if (j < y) then
        Delete_inc_rect(H[i + 1][j], H[i][j]);
      end if;
  end for;
  return H;
End
```

The function $\text{Build}_\text{rectangles}$ is more complex. We will start with an example. Let us suppose that have an image with $p$ its upper left corner (Table 2).

<table>
<thead>
<tr>
<th>$p$</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>13</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Table 2

First we build a rectangle the largest possible and the higher possible for that width. On our example it is a rectangle of width 4. With that width the maximum height is 2, because if we build a rectangle of width 4 and height 3 it will contain a pixel of color 13 (Table 3).

<table>
<thead>
<tr>
<th>$p$</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>13</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Table 3

Then we build the second rectangle the largest possible (and again the higher possible for that width). On our example it is a rectangle of height 3 and width 3 (Table 4).

<table>
<thead>
<tr>
<th>$p$</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>13</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Table 4

Finally we obtain the list of all the rectangles such that $p$ is their upper left corner (Table 5).

<table>
<thead>
<tr>
<th>$p$</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>13</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Table 5

A rectangle is defined by two pixel (its upper left corner and its lower right corner). In the following pseudo code the rectangle constructor is $\text{rect}(x_1, y_1, x_2, y_2, C)$, where $(x_1, y_1)$ are the coordonates of the first pixel, $(x_2, y_2)$ are the coordonates of the second pixel and $C$ is the color of the rectangle.

**Construction of the list of the maximal rectangles such that $(i, j)$ is their upper left corner.**

```plaintext
hypergraph Build_rectangles(image I, int i, int j)
  hypergraph H = \emptyset;
  x = width of I;
  y = height of I;
  v_x = x;
  v_y = j;
  c_color = I[i][j];
  r_color = c_color;
  for (k = i; k < +) # Vertical progression
    c_color = I[k][j];
    if ($r_color \neq c_color$ or $y \leq k$) then
      add rect(i, j, k - 1, v_y, r_color) to H;
      break for;
  end if;
  for (l = j; l < +) # Horizontal progression
```
Proposition 2 - The complexity of the algorithm building the hypergraph associated to an image is $O(n^2)$ in the worst case, where $n$ is the number of pixels in the image. This complexity is reached for a uniformly colored image.

5. Experimental results

We compare our algorithm to the following ones:

- GIF, the most used lossless image compression format on the web. We use non-interlaced v89a files.
- PNG (Portable Network Graphics format [7]). We use non-interlaced files compressed with the plug-in SuperPNG [8] at default settings.
- JPEG-LS, the new standard for the lossless compression. We use the HP Labs' software implementation of LOCO-I/JPEG-LS that can be found at http://www.hpl.hp.com/locos/ [5].

For all the test $K$ is fixed to 16. The uncompressed sizes correspond to uncompressed TIFF files. The following test is based on the miscellaneous image set from [11] and contains mainly some natural image. All the 44 images have been converted to grayscale. All the sizes are in bytes.

<table>
<thead>
<tr>
<th>Compression</th>
<th>Size</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed</td>
<td>11980312</td>
<td>100%</td>
</tr>
<tr>
<td>GIF</td>
<td>9416744</td>
<td>78.60%</td>
</tr>
<tr>
<td>PNG</td>
<td>6335010</td>
<td>52.87%</td>
</tr>
<tr>
<td>HLC&amp;PPMd</td>
<td>6261885</td>
<td>52.26%</td>
</tr>
<tr>
<td>JPEG-LS</td>
<td>6031978</td>
<td>50.34%</td>
</tr>
</tbody>
</table>

Table 6

On that images set hypergraph compression is better than PNG and GIF but less efficient than JPEG-LS. But in fact some of the images of the set are worst cases for the HLC. By example, the partial hypergraph $H_{16}$ associated to the image 4.2.0.6.tiff don’t contains any rectangle. So, for this image, PPMd will provide all the compression. If we extract a subset containing only synthetic images the results are much better (Table 7).

<table>
<thead>
<tr>
<th>Image</th>
<th>PNG</th>
<th>JPEG-LS</th>
<th>HLC&amp;PPMd</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.13.tif</td>
<td>10970</td>
<td>8616</td>
<td>7721</td>
</tr>
<tr>
<td>gray21.512.tif</td>
<td>1461</td>
<td>1998</td>
<td>83</td>
</tr>
<tr>
<td>ruler.512.tif</td>
<td>3788</td>
<td>17348</td>
<td>1417</td>
</tr>
<tr>
<td>testpat.1k.tif</td>
<td>55091</td>
<td>77463</td>
<td>53239</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>71310</strong></td>
<td><strong>105425</strong></td>
<td><strong>62460</strong></td>
</tr>
<tr>
<td><strong>Total%:</strong></td>
<td><strong>4.33%</strong></td>
<td><strong>6.41%</strong></td>
<td><strong>3.80%</strong></td>
</tr>
</tbody>
</table>

Table 7
JPEG-LS files are systematically bigger than hypergraph compressed ones on that image selection. The most interesting case is of course gray21.512.tiff, a regular rectangle-composed picture, that is reduces to 83 Bytes (1998 bytes for JPEG-LS and 1461 for PNG). Our compression is also very good on testpat.1k.tiff that mix natural portions and some geometric figures. It results from the good separation between the geometric compression part of the algorithm and the data segment compression part (based on PPMd). The two parts are powerfully combined as shown on table 8.

<table>
<thead>
<tr>
<th>Image</th>
<th>PPMd</th>
<th>HLC</th>
<th>HLC&amp;PPMd</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.13.tiff</td>
<td>8525</td>
<td>11529</td>
<td>7721</td>
</tr>
<tr>
<td>gray21.512.tiff</td>
<td>1765</td>
<td>102</td>
<td>83</td>
</tr>
<tr>
<td>ruler.512.tiff</td>
<td>8661</td>
<td>11016</td>
<td>1417</td>
</tr>
<tr>
<td>testpat.1k.tiff</td>
<td>62453</td>
<td>101319</td>
<td>53239</td>
</tr>
</tbody>
</table>

| Total:         | 81404| 123966| 62460    |
| Total%:        | 4.95%| 7.54% | 3.80%    |

Table 8

The time required for compressing all the 44 images with HLC and PPMd on a 2 GHz 32 bits Athlon is 88.21 seconds (43.19 seconds for the HLC compression and 45.02 seconds for the PPMd compression of the HLC files). But our C++ implementation was not optimized for speed.

6. Conclusion

The real power of the hypergraph compression is its ability to extract regular geometric data from an image and to isolate them. The others data are left unchanged and are stored in a specific data segment. So, with hypergraphs, geometric compression can be combined with a PPM (prediction by partial matching) data compression algorithm, or with a LZW-based algorithm. The total result is very efficient.

Two main generalizations of the image hypergraph representation can be establish:

1. The representation and the algorithm can be generalized to three-dimensional images simply by using rectangle parallelepipeds in replacement of rectangles.

2. The algorithm can be tweaked for parametric lossy compression by adding an α tolerance. The hyperedges become the maximal rectangles such that inside a rectangle all the pixels have a color at a distance inferior to α. The results depend on the distance chosen on the color space.

References


