Hypergraphs for Near-lossless Volumetric Compression

Luc Gillibert  
Université de Caen, GREYC UMR-6072.  
Bd Marechal Juin BP 5186, 14032 Caen cedex, France.  
luc.gillibert@info.unicaen.fr

Alain Bretto  
Université de Caen, GREYC UMR-6072.  
Bd Marechal Juin BP 5186, 14032 Caen cedex, France.  
alain.bretto@info.unicaen.fr

Abstract

A hypergraphs-based image representation is already used for 2D lossless image compression. In this paper we extend this hypergraph representation on 3D-image and we add an $\alpha$-tolerance. This extended representation conducts to a generalisation of the HLC lossless compression algorithm [2] for near-lossless 3D-image compression. We present an algorithm performing that compression and we give some experimental results proving its efficiency. This paper is a detailed version of the poster [3].

1 Introduction

Image compression addresses the problem of reducing amount of data needed to represent a digital image. Lossless or reversible compression refers to compression techniques in which the reconstructed data exactly matches the original. Near-lossless compression denotes compression methods which give quantitative bounds on the nature of the loss that is introduced. Such compression techniques provide the guarantee that no pixel difference between the original and the compressed image is above a given value. Both lossless and near-lossless compression are used for 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 [5, 6]. In this paper we describe a new method for near-lossless 3D-image compression, based on hypergraphs and called HNLC (Hypergraph Near-Lossless Compression). This method is a generalisation of the HLC method (Hypergraph Lossless Compression). The hypergraphs are a very interesting generalisation of the graphs. Introduced in 1960 by C. Berge [8], they are now used in many domains such as chemistry, engineering and image processing [1, 4]. We give an algorithm making the conversion between a 3D-matrix-based representation and the hypergraph representation. We also present some experimental results proving that HNLC, combined with a PPM-based [11] data compression algorithm, is very efficient.

2 Definitions

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 $i = 1, 2, \ldots, m$
- $\bigcup_{i=1}^{m} E_i = V$

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 hyper-edges of the hypergraph. A partial hypergraph $H'$ from a hypergraph $H$ is a hypergraph such that $H' \subset H$. If the set of vertices of $H'$ is equal to the set of vertices of $H$ we say that $H'$ is a partial hypergraph covering. A simple hypergraph is a hypergraph $H = \{E_1, E_2, \ldots, E_m\}$ such that $E_i \subseteq E_j \Rightarrow i = j$.

3 Hypergraph representation

3.1 Formal definition

Let $I$ be a 3-dimensional matrix represented image (if the size of $I$ is $i \times j \times k$, $I$ can be see as a set of $k$ 2D matrix of size $i \times j$). We build a hypergraph $H^\alpha(I)$, called the extended hypergraph representation of the image for the tolerance $\alpha$, as it follow:
The vertices of $H^\alpha(I)$ are the voxels of the 3D-image.

The hyper-edges of $H^\alpha(I)$ are the maximal rectangle parallelepipeds such that inside a rectangle parallelepiped all the voxel colors are at a distance inferior to $\alpha$ of the color of the upper higher left corner of the rectangle parallelepiped.

This hypergraph will depend on the distance chosen on the color space. In the following paper we use grey-scale voxels and the canonic distance on grey-scale voxels (the absolute value of the difference of the grey-scale value of the voxels).

The hypergraph $H^\alpha(I)$ is a simple hypergraph because, by construction, the hyper-edges are maximal for the inclusion. Notice that, given the extended hypergraph representation of a 3D-image (the list of the rectangle parallelepipeds and the color of the upper higher left corner of each rectangle parallelepiped), the image can be integrally rebuild with an error inferior or equal to $\alpha$ on each voxel. If $\alpha$ is chosen equal to zero the hypergraph can be denoted $H(I)$ and is equal to the one used for HLC.

3.2 The algorithm

In this code a hypergraph is a list of rectangle parallelepipeds and an image is a 3D table of integers. Build_hyper$(I, \alpha)$ creates the hypergraph $H^\alpha(I)$ associated to a 3D-image $I$ for the tolerance $\alpha$. For each voxel $v$ with coordinates $(i, j, k)$ of the image $I$, the algorithm constructs the list of the maximal rectangle parallelepipeds such that $v$ is their upper higher left corner. Then these parallelepipeds are compared with the parallelepipeds having $(i + 1, j, k)$, $(i, j + 1, k)$ or $(i, j, k + 1)$ as upper higher left corner. So, the included hyper-edges can be deleted during the construction of the hypergraph. This function Build_hyper returns a 3D-table of list of rectangle parallelepipeds. So, for each voxel $v$, it is possible to access directly to the list of rectangle parallelepipeds having the voxel $v$ as upper higher left corner. The function Build_parallelepiped$(I, \alpha, i, j, k)$ is their upper higher left corner.

$H[I][j][k]$ = Build_parallelepiped$(I, \alpha, i, j, k)$;
if $(i < x)$ then Delete_inc_parallelepiped$(H[i + 1][j][k], H[i][j][k])$;
end if;
if $(j < y)$ then Delete_inc_parallelepiped$(H[i][j + 1][k], H[i][j][k])$;
end if;
if $(k < z)$ then Delete_inc_parallelepiped$(H[i][j][k + 1], H[i][j][k])$;
end if;
end for;
end for;
return $H$;

A rectangle parallelepiped is defined by two voxels (its upper higher left corner and its lower deeper right corner). In the following pseudo code the rectangle parallelepiped constructor is para($x_1, y_1, z_1, x_2, y_2, z_2, C$), where $(x_1, y_1, z_1)$ are the coordinates of the first voxel, $(x_2, y_2, z_2)$ are the coordinates of the second voxel and $C$ is the color of the rectangle parallelepiped.

Construction of the list of the maximal rectangle parallelepipeds such that $(i, j, k)$ is their upper higher left corner:

$H[I][j][k] = \text{Build_parallelepiped}(image\ I, int\ \alpha, int\ i, int\ j, int\ k)$

$H = \emptyset$;
$x = \text{width of } I; y = \text{height of } I; z = \text{deep of } I$;
$v_x = x; v_y = y$;
color = $I[i][j][k]$; r_color = c_color;
for ($p_x = k; p_x + +$) # In-deep progression
color = $I[i][j][p_x]$;
if ($|r_color - c_color| > \alpha$ or $p_x > z$) then
add para($i, j, k, p_x, v_y, p_z - 1, r_color$) to $H$;
break for;
end if;
for ($p_y = j; p_y + +$) # Vertical progression
color = $I[i][p_y][p_z]$;
if ($|r_color - c_color| > \alpha$ or $p_y > y$) then
if ($p_y < v_y$) then
add para($i, j, k, v_x, p_y, p_z - 1, r_color$),

$H[I][j][k] = \text{Build_parallelepiped}(image\ I, int\ \alpha)$

$x = \text{width of } I; y = \text{height of } I; z = \text{deep of } I$;
$H = \text{new hypergraph}[x][y][z]$;
for ($i = x; i \geq 0; i -$)
for ($j = y; j \geq 0; j -$)
for ($k = z; k \geq 0; k -$)
$H[i][j][k] = \text{Build_parallelepiped}(I, \alpha, i, j, k)$;
if ($i < x$) then Delete_inc_parallelepiped$(H[i + 1][j][k], H[i][j][k])$;
end if;
if ($j < y$) then Delete_inc_parallelepiped$(H[i][j + 1][k], H[i][j][k])$;
end if;
if ($k < z$) then Delete_inc_parallelepiped$(H[i][j][k + 1], H[i][j][k])$;
end if;
end for;
end for;
return $H$;

End
\( v_y = p_y; \)
end if;
break for;
end if;
for \((p_x = i; \; p_x + + )\) \# Horizontal progression
\( c_{\text{color}} = I[p_x][p_y][p_z]; \)
if \((|r_{\text{color}} - c_{\text{color}}| > \alpha \text{ or } p_x > x)\) then
if \((p_x < v_x)\) then
add to \( H; \)
\( \text{para}(i, j, v_x, v_y, v_z - 1, r_{\text{color}}); \)
v_x = p_x;
end if;
break for;
end if;
end for;
return \( H; \)
end for;

The procedure \texttt{Sub-hypergraph} extracts from the ordered hypergraph \( H_0^\alpha(I) \) the sub-hypergraph \( H_K^\alpha(I) \) such that each rectangle parallelepiped brings some information on at least \( K \) new voxels.

\textbf{Construction of the partial hypergraph} \( H_K^\alpha(I) \).

\texttt{Sub-hypergraph(hypergraph \( H \); image \( I \))}

\( x = \) width of \( I; \; y = \) height of \( I; \; z = \) deep of \( I; \)
flag = new bool\([x][y][z]\]
for \((t_x = 0; t_x < x; t_x + + )\)
for \((t_y = 0; t_y < y; t_y + + )\)
for \((t_z = 0; t_z < z; t_z + + )\)
flag\([t_x][t_y][t_z]\) = 0;
end for;
end for;
end for;
for \((r \text{ in } H)\)
useful\_parallelepiped = 0; \( c = 0; \)
for \((t_x = r.x; t_x \leq r.x2; t_x + + )\)
for \((t_y = r.y; t_y \leq r.y2; t_y + + )\)
for \((t_z = r.z; t_z \leq r.z2; t_z + + )\)
if \((\text{flag}\[t_x][t_y][t_z]\) = 0) then
\( c++; \)
if \((c \geq K)\) then
useful\_parallelepiped = 1;
end if;
end if;
end for;
end for;
if \((\text{useful\_parallelepiped} = 0)\) then
remove \( r \) from \( H; \)
else
for \((t_x = r.x; t_x < r.x2 + 1; t_x + + )\)
for \((t_y = r.y; t_y < r.y2 + 1; t_y + + )\)
for \((t_z = r.z; t_z < r.z2 + 1; t_z + + )\)
flag\([t_x][t_y][t_z]\) = 1;
end for;
end for;
end for;
end for;

\textbf{Proposition 1 -} 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 voxels in the image.

\section{Hypergraph image compression}

A rectangle parallelepiped can be stored as a couple of points. But such representation required several bytes. Let’s consider the case of a 3D one-voxel black/white checker-board pattern. The hypergraph associated to this image will contains as many rectangle parallelepipeds as there are voxels in the image and each rectangle parallelepiped will require more than one byte for storing it. So the 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 \( I \):

1. Build \( H^\alpha(I) \), the extended hypergraph representation of \( I \) for the tolerance \( \alpha \).
2. Order the hyper-edges of \( H^\alpha(I) \) comparing volume (the biggest rectangle parallelepiped comes first). If two parallelepipeds have the same volume the first one is the first constructed. The ordered hyper-edges are called \( \{R_1, \ldots, R_m\} \). The ordered hypergraph is called \( H_0^\alpha(I) \).
3. Extract from \( H_0^\alpha(I) \) the following partial hypergraph:
\[ H_K^\alpha(I) = \{R_x \in \{R_1, \ldots, R_m\}; \; x \in X\} \]

The set of indices \( X \) is chosen such that for all \( x \in X \), \( R_x \) contains at least \( K \) voxels that are not in \( \bigcup_{0 \leq i < x} R_i \) (\( X \) can be empty). The partial hypergraph \( H_K^\alpha(I) \) depends on \( K \). One can note that if \( K = 0 \) then \( H_K^\alpha(I) = H_0^\alpha(I) \).
4. Store the hyper-edges of \( H_K^\alpha(I) \). With a good representation between 6 and 18 bytes are required for a rectangle parallelepiped (three coordinates per point for two points, so six coordinates, and a coordinate take between 1 and 3 bytes). The colour requires 1 byte or more, depending the image.
5. Create an empty data segment (a list which will be used for the voxels that are not covered by any parallelepipeds).
6. Create a volume $V$ of the size of the image with one flag per voxel. Initialise all the flags to $0$.

7. Read the hypergraph $H_K(I)$ and draw all the hyper-edges on $V$. For each vertex drawn, set the corresponding flag to $1$.

8. Traverse the image with the canonical order and, for each flag set to $0$, write the voxel colour in the data segment. The colours are written linearly so no additional data are required. The flags indicate the voxels that are not covered by any hyper-edges.

9. 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 [10]. In fact a LZW-based algorithm can also give good results.

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

1. Expand the hypergraph and the data segment with the PPM-based algorithm.

2. Create a volume $V$ of the size of the image with one flag per voxel. Set all the flags to $0$.

3. Read the hypergraph and draw all the rectangle parallelepipeds on $V$. For each voxel drawn, set the corresponding flag to $1$.

4. Traverse the image and, for each flag set to $0$, give to the voxel the color read in the data segment.

So, by example, if there is no rectangle parallelepiped with a volume at least equal to $K$ voxels, the HNLC compressed image, before the PPM compression, will be an empty hypergraph followed by a data segment equal to the original image.

Proposition 2 - The number of hyper-edges in $H_K(I)$ is inferior or equal to the number of voxels of $I$ divided by $K$ if $K \geq 1$.

Therefore, with $K$ bigger than the size of a rectangle parallelepiped (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 HNLC algorithm will be 4 bytes bigger than the original one.

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. GIF can be used for compressing short image sequences (and every 3D-image can be see as a 2D-image sequence).

- PNG (Portable Network Graphics format [9]). We use non-interlaced files. PNG only compress 2D-images so the size of a compressed 3D-image will be the added size of all the compressed 2D frames constituting it. PNG is lossless, not near-lossless, but we will use palette reduction.

- JPEG-LS, the new standard for the lossless and near-lossless compression. We use the HP Labs’ software implementation of LOCO-I/JPEG-LS [7].

For all the test $K$ is fixed to 26. All the sizes are in bytes.

We use the two following sets of pictures:

1. 0%-noisy brains from the Brain Web [12]. Each voxel takes a byte (unsigned). The uncompressed sizes correspond to uncompressed raw files. Detailed information about the brain files is in table 1.

2. The first 20 GIF files from the animals section of Horton-Szar animation library. The files are converted to grey-scale.

http://www.horton-szar.net/clipart/animals.php

<table>
<thead>
<tr>
<th>File</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain 1</td>
<td>M=T1, Noise=0%, INU=0%</td>
</tr>
<tr>
<td>Brain 2</td>
<td>M=T2, Noise=0%, INU=0%</td>
</tr>
<tr>
<td>Brain 3</td>
<td>M=PD, Noise=0%, INU=0%</td>
</tr>
</tbody>
</table>

Table 1

<table>
<thead>
<tr>
<th>Compression</th>
<th>Size</th>
<th>Size %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed</td>
<td>7,069,860</td>
<td>100%</td>
</tr>
<tr>
<td>GIF (Lossless)</td>
<td>5,118,423</td>
<td>72.40%</td>
</tr>
<tr>
<td>JPEG-LS (tolerance=2)</td>
<td>1,487,253</td>
<td>21.04%</td>
</tr>
<tr>
<td>HNLC&amp;PPMd (tolerance=2)</td>
<td>2,861,518</td>
<td>40.47%</td>
</tr>
</tbody>
</table>

Table 2 (Set 1)

On this brain set JPEG-LS is better than HNLC. But hypergraph compression [2] is not optimal on natural pictures, and this set of picture is unrelavent for our method. An appropriate reduction of the 8 bits palette (256 values) to a 4 bits palette (16 values) can be considered as a near-lossless compression with a tolerance of 8. So we are able to compare HNLC with JPEG-LS, PNG and GIF for a near-lossless compression.

With a tolerance of 2, i.e. the maximal distance between the original voxel and the compressed voxel is fixed to 2, hypergraph compression combined with PPMd is better than JPEG-LS on this set of pictures and animations. With a tolerance of 8, the difference with JPEG-LS is reduced, but
HNLC is still better. GIF and PNG are also beaten. The efficiency of HNLC 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 3 (lines PPMd alone and HNLC alone).

<table>
<thead>
<tr>
<th>Compression</th>
<th>Size</th>
<th>Size %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed</td>
<td>1,148,880</td>
<td>100%</td>
</tr>
<tr>
<td>GIF (lossless)</td>
<td>227,248</td>
<td>19.78%</td>
</tr>
<tr>
<td>PPMd (alone, lossless)</td>
<td>142,734</td>
<td>12.42%</td>
</tr>
<tr>
<td>HNLC (alone, tolerance=2)</td>
<td>348,426</td>
<td>30.32%</td>
</tr>
<tr>
<td>JPEG-LS (tolerance=2)</td>
<td>191,902</td>
<td>16.70%</td>
</tr>
<tr>
<td>HNLC&amp;PPMd (tolerance=2)</td>
<td>120,414</td>
<td>10.48%</td>
</tr>
<tr>
<td>HNLC (alone, tolerance=8)</td>
<td>290,102</td>
<td>25.25%</td>
</tr>
<tr>
<td>GIF (4 bits palette)</td>
<td>139,279</td>
<td>12.12%</td>
</tr>
<tr>
<td>PNG (4 bits palette)</td>
<td>138,540</td>
<td>12.06%</td>
</tr>
<tr>
<td>JPEG-LS (tolerance=8)</td>
<td>123,729</td>
<td>10.77%</td>
</tr>
<tr>
<td>HNLC&amp;PPMd (tolerance=8)</td>
<td>116,707</td>
<td>10.16%</td>
</tr>
</tbody>
</table>

Table 3 (Set 2)

References


