LISBOA | UNIVERSIDADE DE LISBOA

Lossy Compression of Images

Overview of various approaches

Desenho e Análise de Algoritmos

Professor:
Carlos Lourenço

Authors (Group 1):
Miguel Rodrigues, fc44281
Miguel Falé, fc43556

Context

Compression algorithms encode information into a smaller representation than the original, and are widely adopted in order to reduce storage space usage, as well as bandwidth consumption. However, there is a clear interplay between compression and a series of other important factors, and the usefulness of the resulting time and ratio varies per use case:

- **Space-time complexity**. Do we want faster compression with less reduction, or is the compression ratio more important? Web application operators need to make these trade-offs, i.e. if they want to provide services for video storage or streaming.
- **Cost of hardware resources**. Both time and space issues can be mitigated through the use of more powerful hardware, but is it cost effective? For real-time speech coding, it is often better to process data as fast as possible. In order to reduce bandwidth consumption and delays, and given the redundant nature of voice, discarding and distorting some of the data is perfectly reasonable.
- **Auxiliary storage**. Even though most indicators show that storage devices are getting cheaper, the quantity of information being produced is also increasing, thus maintaining the relevance of these algorithms.
- **Lossy vs Lossless**. Do we want to recover the original data in its full integrity, or do we allow for some losses? In addition to the real-time speech example, images can also lose some information to a certain degree, and a good lossy algorithm can offer significant ratios - often better than their lossless relatives - without noticeable degradation. On the other hand, there is a risk for generation loss if an image is subject to many subsequent encodings.

In this project we decided to shift our focus into the world of **lossy** compression of images. Before we move on, we explain key concepts used throughout this report:

- **Pixel.** Atomic component of an image. Each pixel has an associated color, and this aspect is important when explaining distance and similarity functions later on.
- **RGB.** Popular color system formed by Red, Green and Blue components. A common representation of RGB is the 0-255 scale (per each of the three channels), with each channel data being stored in 1 byte.
- **Mosaic.** This is a piece of an image, composed by a number of pixels.
- **Prototypes.** Mosaics that are most representative of an image. Prototypes play a fundamental role in our approach, since we can take advantage of redundancy and minimum distance in order to produce a simpler (smaller) image by replacing the mosaics of the original image by the most appropriate prototypes.
    - The mosaic/prototype size will determine the seamlessness of the artifacts displayed by the image. Higher mosaic sizes will result in “blocky” images, but allow for faster processing and better reductions.
    - The number of prototypes will determine the range of candidates for potential mosaic replacements. A wider range of prototypes means that our algorithm has to spend more time going through a wider range of similarity verifications, and store a higher number of prototypes.

Overview of the implementation

The following list depicts the main “recipe” used throughout the project. We developed a python script that takes four arguments - the input image, and three additional parameters concerning the mosaic size, the number of prototypes and the alpha value used in the learning algorithm. The result is a highly compressed (non-visual) version of the image, which can be later decompressed into a slightly altered image that also occupies significantly less space than the original; the level of degradation depends on the choice of additional parameters. Even though some algorithms and internal approaches were changed along the way, the same basic scheme was left unchanged from the start.

1. Select Image
2. Select prototypes
3. Re-do image by replacing each mosaic with a prototype
4. Build dictionary
5. Compress such Dictionary
6. Decompress
7. See the results

Approaches

The choice of prototypes (and their sources) was our first main challenge. The function used for similarity was also subject to change along the way. Later on, we improved prototype selection through training.

First approach

Given the input image:

1. Take a collection of images as **sources** for prototypes;
2. Choose the *n* prototypes that are **most similar with each mosaic of the input image**. Our similarity function returns a simple difference between two matrices.
3. Build a new image from the prototype selection

Second approach

Given the input image:

1. Prototypes are obtained **from the input image** rather than a collection. The reason for this is because the previous approach didn’t scale, and the prototypes selected had very few similarity with the final image
2. Choose the *n* prototypes that are most similar with each mosaic of the input image. Our similarity function returns a simple difference between two matrices, again.
3. Build a new image from the prototype selection

Third approach

1. Prototypes are obtained **from the input image** rather than a collection.
2. Choose the *n* prototypes **at random**.
3. Train prototypes using a learning algorithm.
4. Build a new image from the **trained** prototype selection, according to the new Comparison Algorithm, described below.

Learning Algorithm

As for the learning algorithm in the third approach:
n -> selected prototype
y -> most similar prototype
x -> current mosaic
α -> alpha [0,1]
i,j -> mosaic location

$$y^n = y^n + \alpha(x_{ij} - y^n)$$

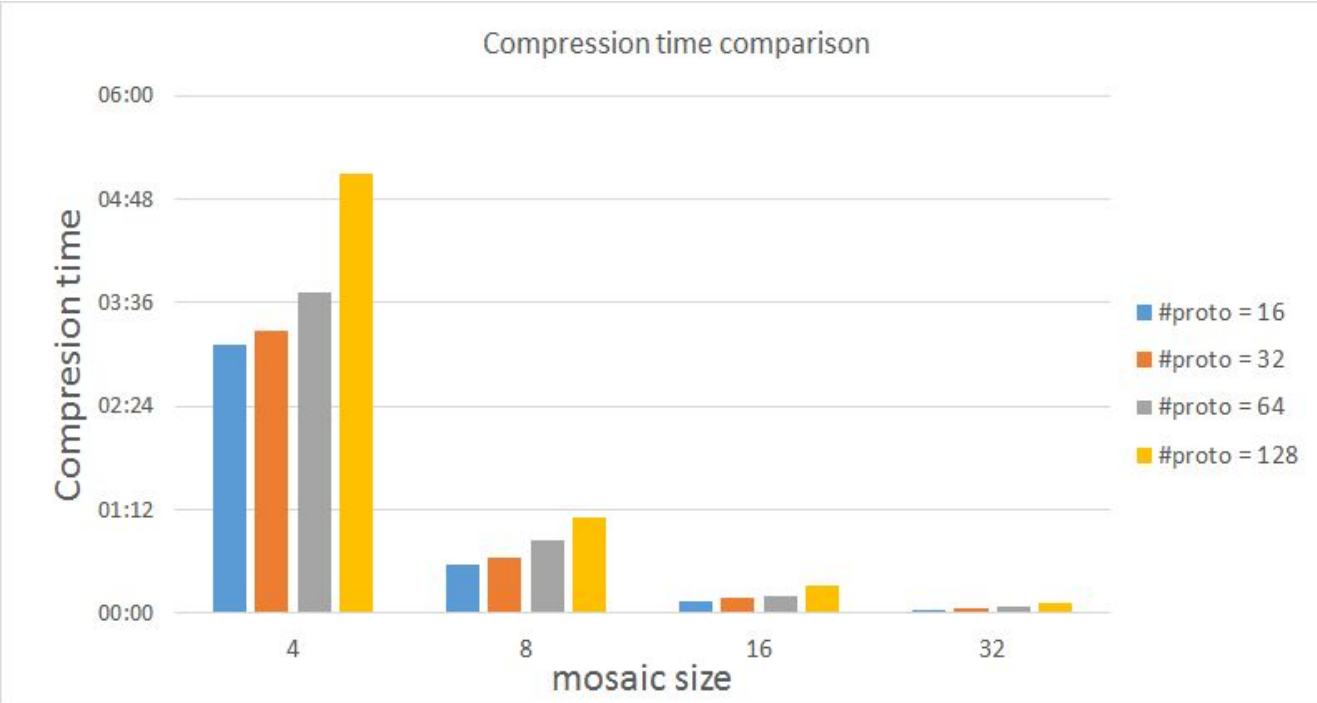
Comparison Algorithm

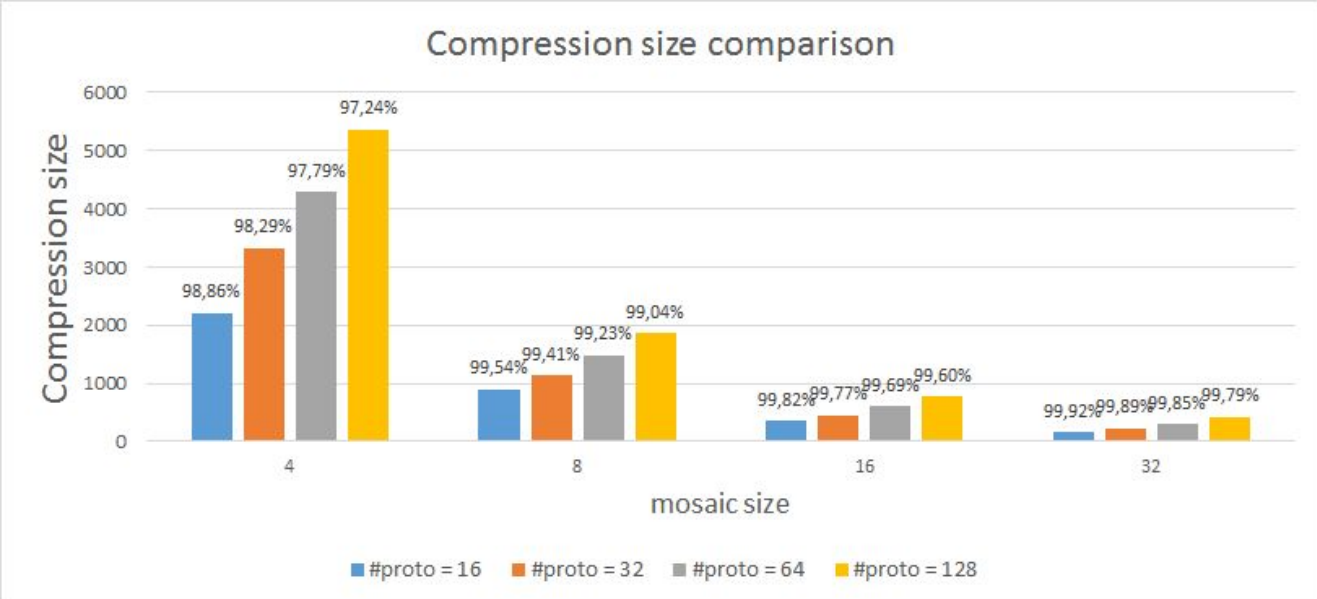
n -> selected prototype
y -> most similar prototype
x -> current mosaic
α -> alpha [0,1]
p -> number of prototypes

$$d(y^n, x) \leq d(y^n, x) \ \forall n$$

$$d(y^n, x) = \sqrt{\sum_{i=1}^{p} (y^n - x^i)^2}$$

Results


Compression time comparison
06:00
04:48
03:36
02:24
01:12
00:00
Compresion time
4
8
16
32
mosaic size
#proto = 16
#proto = 32
#proto = 64
#proto = 128


Compression size comparison
6000
5000
4000
3000
2000
1000
0
Compression size
98,86%
98,29%
97,79%
97,24%
99,54%
99,41%
99,23%
99,04%
99,82%
99,77%
99,69%
99,60%
99,92%
99,89%
99,85%
99,79%
4
8
16
32
mosaic size
#proto = 16
#proto = 32
#proto = 64
#proto = 128

Final considerations

Given that we wanted to alter RGB values according to their respective real differences, and not equally among the three values, we used a specific multiplier to each, according to the real difference that existed for that value.

For example, if we had $\alpha = 0.01$, a RGB of [100 0 200] and we wanted to make it closer to the respective winner of [100 0 20], we would only decrease the value of the last RGB, making the new RGB array like [100 0 (200 + 0.01 (20 - 200)] = [100 0 182]. This way, we would assure a better approximation.

Example

Original Image

Compressed Image

References

1. Steven S. Skiena. 2008. The Algorithm Design Manual (2nd ed.). Springer Publishing Company, Incorporated.
2. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. Introduction to Algorithms (2nd ed.). McGraw-Hill Higher Education.
3. Davis, Z., 2013. Huffman | An Algorithmic Lucidity. May, 2016 in: zackmdavis.net/blog/2013/06/huffman/
4. Md. Rubaiyat Hasan, 2011. Data Compression using Huffman based LZW Encoding Technique. June, 2016 in: www.ijser.org/researchpaper%5CData-Compression-using-Huffman-based-LZW-Encoding-Technique.pdf
5. Anon, 2016. Huffman coding - Rosetta Code. May, 2016 in: rosettacode.org/wiki/Huffman_coding
6. Mark Nelson,2016. LZW Data Compression. June, 2016 in: marknelson.us/1989/10/01/lzw-data-compression/
7. Anon, 2016. LZW compression - Rosetta Code. June, 2016 in: rosettacode.org/wiki/LZW_compression#Python
8. Jia Li. Prototype Methods: K-Means. June, 2016 in: http://sites.stat.psu.edu/~jiali/course/stat597e/notes2/kmeans.pdf
9. Scott Beale, 2009. Generation Loss, What JPEG Compression Looks Like After 600 Saves. June, 2016 in: http://laughingsquid.com/generation-loss-what-jpg-compression-looks-like-after-600-saves/
10. https://en.wikipedia.org/wiki/RGB_color_model
11. https://en.wikipedia.org/wiki/K-means_clustering