Dense Skeletons for Image Compression and Manipulation

M. L. Terpstra

university of groningen

faculty of mathematics and natural sciences

Dense Skeletons for Image Compression and Manipulation

by

M. L. Terpstra

Student number:	2028980	
Supervisors:	Prof. dr. A. C. Telea	University of Groningen
	C. Feng MSc	University of Groningen

Cover Image: Excerpt of a map of Groningen in 1575 skeletonized at graylevel 153.

university of groningen

faculty of mathematics and natural sciences

Abstract

Skeletons are well-known, compact 2D or 3D shape descriptors. Earlier, skeletons have been extended to dense skeletons to encode grayscale images rather than binary images. To do this an image is decomposed in threshold sets which are skeletonized individually. So far, storing images using this approach has not been able to compete with common image compression algorithms such as JPEG.

In this work we attempt to improve the compression quality by exploiting the structure of dense skeletons in order to reduce redundancy and by using sophisticated encoding schemes. We compare these images with conventional image compression methods in terms of size and quality. Moreover, we research the effects of combining well-established image compressors our dense skeleton results.

Previous works have also shown that interesting stylistic effects can occur when an image is processed using dense skeletons. We attempt to introduce new image manipulation techniques by performing *skeleton bundling*. With these operations it can become possible to alter image lighting and perform further image simplification. We will research how these manipulation techniques can influence image size, image quality and how these can create new, interesting effects.

We show that we can reliably generate images using our pipeline of high fidelity at a file size smaller than JPEG using our dense skeleton image encoding and can generate images of very high fidelity at a file size smaller than JPEG by using our method as a JPEG preprocessor. We demonstrate the effects of inter-layer skeleton path bundling as a local contrast enhancement method which generates interesting effect. We also demonstrate that our pipeline can generate extremely simplified representations of images, and extend our method to color images.

Acknowledgments

It would have been impossible to carry out this research without the many people around me during my research. Firstly, I want to express my deepest thanks and gratitude to my supervisors Alex Telea and Cong Feng for giving me the possibility to do this project and their copious amounts of advice, feedback and interest.

I would also like to thank Lianne and my parents, for their continuous support, help, understanding and patience.

Lastly, I would like to thank whomever managed to listen to me whenever I rambled on about something with shapes, bits, and images, in particular Folkert, Han, and Laura.

M. L. Terpstra
Amersfoort, January 2017

Contents

List of Figures ix

1 Introduction 1

2 Related Work: Skeletons 5
- 2.1 Skeletons . . . 5
- 2.2 Computation . . . 6
  - 2.2.1 CPU-method . . . 6
  - 2.2.2 GPU-method . . . 6
- 2.3 Importance . . . 7

3 Related Work: Skeleton Image Coding 9
- 3.1 Layer removal . . . 9
- 3.2 Skeleton simplification . . . 9
- 3.3 Skeleton path encoding . . . 9
- 3.4 Reconstruction . . . 9
- 3.5 Results . . . 10

4 Information Theory & Compression 12
- 4.1 What is information? . . . 12
- 4.2 Limits . . . 13
- 4.3 Techniques . . . 13
  - 4.3.1 Huffman Coding . . . 14
  - 4.3.2 Arithmetic coding . . . 15
  - 4.3.3 Finite-state Entropy. . . . 16
  - 4.3.4 Run-length coding . . . 16
  - 4.3.5 Universal methods . . . 16
  - 4.3.6 Prediction . . . 17
  - 4.3.7 Compaction . . . 17
- 4.4 Assessing compression . . . 18
- 4.5 Image compression & Quality . . . 18
  - 4.5.1 JPEG . . . 18
  - 4.5.2 Image Quality . . . 20

5 Skeleton Compression 24
- 5.1 Pre-filtering . . . 24
- 5.2 Layer selection . . . 25
- 5.3 Skeletonization . . . 26
- 5.4 Signal modification . . . 27
  - 5.4.1 Bundling. . . . 27
  - 5.4.2 Overlap pruning. . . . 29
  - 5.4.3 Delta representation . . . 30
- 5.5 Skeleton data encoding . . . 31
  - 5.5.1 Transformations . . . 32
  - 5.5.2 Other techniques . . . 32
- 5.6 External methods . . . 32
- 5.7 Reconstruction . . . 33

6 Results & Discussion 35
- 6.1 Image results . . . 35
  - 6.1.1 Corpus . . . 35

6.2 Encoding schemes . . . 39
6.3 Overlap pruning effects. . . . 40
6.4 External compression methods . . . 42
6.5 Transcending JPEG . . . 43
6.5.1 Types of images . . . 43
6.5.2 Extreme simplification . . . 49
6.6 JPEG preprocessor . . . 51
6.7 Skeleton path bundling . . . 56
6.8 Color images . . . 57

7 Conclusion & Future work 63

Appendices 65

A How-to 67
A.1 Dependencies . . . 67

B File Format 69

List of Figures

1.1 Severe artifacts in Figure 1.1b due to heavy compression of Figure 1.1a using JPEG . . 2

2.1 Visualisation of the AFMM algorithm. The boundary is monotonically initialized and propagated along the wavefront. Image from [44]. 6
2.2 The influence of boundary noise on the complexity of the skeleton. Images from [41]. . 7

3.1 Some results of Meiburg's pipeline. 10

4.1 The Huffman tree of the sentence "Mississippi river" 14
4.2 The zig-zag encoding of JPEG. Notice the long runs of zeros. 19
4.3 The downside of using errors as a measure for quality. All modified images have the same error but vary wildly in image quality. This is because the human visual system is not taken into account with naive quality measures. [1] 21
4.4 Logistic fit of MS-SSIM scores with the MOS of all images in the LIVE image database. 22

5.1 A high level overview of our encoding scheme and SIR file viewing. A conventional image is the input and a skeletonized representation is the output. 24
5.2 The effect of island filtering on an upper level set. Notice that black is foreground in this image. 25
5.3 The effects of using skeleton image coding as a preprocessor for a matrix method. The images on the bottom are very similar, but much smaller than the original image. 27
5.4 Bundling leading to interesting image effects. 29
5.5 Example of overlap pruning. The original shape is visible in Figure 5.5a. When both upper-level sets are skeletonized parts of the skeleton in Figure 5.5d are made redundant due to the skeleton points created in Figure 5.5e and can safely be discarded. 30
5.6 Overlap pruning in action. Layers t_2 and t_3 are not contributing to the final image as they are entirely occluded by t_4. 31
5.7 Same skeletonization with and without interpolation, clearly Figure 5.7b has a lower psycho-visual error. 33

6.1 Cameraman original (512x512, 257kB) 36
6.2 Commercial for Delft salad oil from 1894 (401x611, 240kB) 36
6.3 Elaine (512x512, 257kB) 36
6.4 Forest (1024x768, 769kB) 36
6.5 Map of Groningen of 1575 (701x601, 417kB) 37
6.6 House (512x512, 257kB) 37
6.7 Lena (128x128, 17kB) 37
6.8 Lena (256x256, 65kB) 37
6.9 Lena (512x512, 257kB) 37
6.10 Smiling people (641x965, 605kB) 37
6.11 Mandril (512x512, 257kB) 38
6.12 Iconic picture of Marilyn Monroe (874x1079, 747kB) 38
6.13 Peppers (512x512, 257kB) 38
6.14 "Starry Night" painting by Van Gogh (750x565, 414kB) 38
6.15 Woman Blonde (512x512, 257kB) 38
6.16 Companion Cube from the game *Portal* (600x375, 220kB) 38
6.17 Comparison of encodings per image on the net file size. 39
6.18 Comparison of encodings per image on the file size before external compression. . . . 40

6.19 Visual comparison of different skeletonizations of Figure 6.2. This image uses 39 most significant layers. 41
6.20 Comparison of file sizes after external compression per image. All parameters are kept constant except for the overlap pruning parameter. 41
6.21 Comparison of the file size after external compression over all images. All parameters are kept constant except for the overlap pruning parameter. 42
6.22 Comparison of external compression methods per image. 42
6.23 Visual comparison of JPEG versus skeletonizations of Figure 6.9. 43
6.24 Visual comparison of JPEG versus skeletonizations of Figure 6.13. 44
6.25 Visual comparison of JPEG versus skeletonizations of “Barbara”. 44
6.26 An example of extreme compression results using dense skeletons with fair quality. . . 45
6.27 Skeletonization of a comical image. 47
6.28 A skeletonization of a non-photographically rendered scene. 48
6.29 Extreme simplifications of various images. 50
6.30 The effects of using skeleton image coding as a preprocessor for a matrix method. The images on the bottom are very similar, but much smaller than the original image. 52
6.31 While the MS-SSIM of Figure 6.31c is higher than that of Figure 6.31d, the JPEG artifacts in the former are much more pronounced and one could say that the quality is lower. . . 53
6.32 Comparison of using skeletonization as preprocessor on the “Marilyn” image (Figure 6.12) 54
6.33 Comparison of using skeletonization as preprocessor on the “Lena” image (Figure 6.9) . 55
6.34 Comparison of using bundling on the “Lena” image (Figure 6.9) 56
6.35 Comparison of using bundling on the “Peppers” image (Figure 6.13) 57
6.36 Different color images of Lena. 59
6.37 Different color images of the Peppers image. 60
6.38 Different color images of the Mandril image. 61

1 Introduction

Ever since digital images are created there has been a need to compress images in order to store and transmit images in an efficient form. Although storage capacity and bandwidth capacity have increased monumentally even over the past decade – let alone compared to 40 years ago – demand for superior image compression algorithms have hardly ever been higher. With the popularization of social networks and smartphones, more images are created, saved and shared than ever before. The current largest social network, facebook, stated that its users share two billion photos every day [5]. Storing these images uncompressed would render the service infeasible due to lack of storage stage, which is an issue already. And while information channels have increased significantly over the decades, many of the current mobile connections – and also some physical in some countries – are capped to a preset volume. Efficient communication is then key in order to avoid exhausting the channel.

Since its introduction, JPEG [49] has been the most common format to store images. This is a low-level effort to compress images by interpreting an image as a matrix where each element symbolizes an intensity or a color of a pixel. This matrix is then sliced in 8×8 macroblocks which are coded using a Discrete Cosine Transform – or `dct` – and subsequently efficiently encoded. This format can easily yield a tenfold compression with little perceptible loss in image quality [20].

In the past twenty years since the introduction of JPEG there were few commercially successful formats created that could compete with JPEG – with the notable exception of PNG – but there has been recent developments. In 2010, Google created the WebP image format based on their WebM video format which boasts an up to 35% smaller file size than a same-quality JPEG by using macroblock prediction algorithms [3]. Another recently introduced format called FLIF claims to outperform PNG, lossless WebP, and has files up to 50% smaller than same-quality JPEG [40]. However, this format is virtually unsupported and still a work in progress. There are also efforts to amend JPEG by using format-specific re-encoding of existing JPEGs without loss. One such effort is lepton, which is led by Dropbox [21]. It claims an average 22% drop in file size.

All these methods share the approach of considering an image as a matrix. While this facilitates, until now, unprecedented compression it comes with a few problems. The first one is a technical problem. One of the biggest downsides of “matrix-based” image compression is that graceful degradation is difficult to achieve. One of the occurring problems of JPEG are various kinds of artifacts, most notably so called “blocking” and “ringing” which is visible in Figure 1.1. This is what happens when the quality is too low such that the macroblocks become painfully visible and is direct consequence of the matrix-based approach to image processing.

The second problem is a more semantic problem. While it makes sense from a computing-science perspective to approach an image as a matrix, it makes hardly any sense to do this from a human perspective. Humans reason about images from a more morphological perspective: it has shapes, edges, colors. Most of the time it even transcends this perspective and we reason about faces, plants, buildings, and other high-level features present in an image.

Suppose we are able to discern important or salient ‘shapes’ or ‘features’ in an image. Then we would be able to encode more important shapes in greater detail and less important shapes in lesser

(a) The well-known, original "Lena" image (b) Figure 1.1a with extreme JPEG artifacts

Figure 1.1: Severe artifacts in Figure 1.1b due to heavy compression of Figure 1.1a using JPEG

detail. This is the basis for a lossy[1] image compression technique. There are several methods available that attempt to capture such features. Skeletons are among the most important classes for shape processing, medial axis skeletons in particular. They attempt to be a compact representation of the topology and geometry of a binary shape.

In a previous work [26] it was attempted to employ medial axis skeletons for grayscale image encoding and reconstruction by thresholding an image in upper level sets and transform each set using the Medial Axis Transform and subsequent filtering to obtain salient skeletons. These are encoded in a compact manner in order to save space. Moreover, they have also found that when simplifying and encoding an image using skeletons, interesting effects can occur. For example, it was previously noted that when images are simplified using skeletons they resemble a painting-like effect which can be generated with far greater ease than especially crafted algorithms as, for example, described by Papari et al. [29]. It turned out that these "artifacts" can be favorable and therefore we will generate new types of image modifications which allow us to *manipulate* image structures and will generate new, interesting artistic effects.

While they demonstrated that multi-scale skeleton for image encoding, simplification, and compression works, the results are not on par with modern image formats. Same quality images are larger than their JPEG counterpart but low-quality skeleton images are much more aesthetically pleasing than low-quality JPEGs as they do not suffer from blocking or ringing artifacts.

In this work we attempt to answer two questions:

1. How we can use skeletons for *efficient* and *effective* image compression?

For such a method to succeed we have to describe methods which yield an effective image encoding and compression which are quality and size-wise comparable or better than state-of-the-art image compression methods. Moreover it must be possible to compute such representation with high efficiency to enable a reasonable time frame as to compete with current standards. In chapter 2, we describe skeletons, how one can compute them in a robust, efficient and fast way. It also describes how dense skeleton image coding has been performed in the past. In chapter 4 we discuss in detail how general compression techniques work as well as how a state-of-the-art method as JPEG compresses images. It will also provide a theoretical foundation for assessing image quality and compression quality. In chapter 5 we study in detail how to compress the structure of dense skeletons and how to store such structure efficiently.

2. How can we use the structure of an image skeletonization to perform new types of image manipulation?

[1]A lossy image encoding is an approximation of an original image, as opposed to a lossless image which is an exact encoding.

Due to the new shape-oriented perspective that dense skeletons provide it also opens up new possibilities for performing *image manipulation*. These techniques can introduce interesting new effects and further aid tasks as image simplification or non-photorealistic rendering. One such technique is *inter-layer skeleton path bundling*. This is an interesting but straightforward manipulation technique which can introduce some rather interesting effects. This technique and its effects are further discussed in subsection 5.4.1.

In the appendix there will be some documentation on how to use the supplementary tool to convert tools back and forth between skeleton images and raster images.

2

Related Work: Skeletons

Before we describe how to store images using shapes it is necessary how we define shapes and how they are represented using skeletons.

2.1. Skeletons

A skeleton is a transformation of a shape which provides a compact and simple descriptor of the original shape. Skeletons find many applications in computer graphics, flow visualization, medical imaging, metrology, and robotics[41][44]. There are many different definitions of the skeleton which are all slightly different but they are all share the properties such that they are

Invertible The original shape can be retrieved from the skeleton,

Compact The skeleton is a subset of the shape – ideally an infinitesimally small shape.

Expressive Skeletons are intuitive descriptors which can capture the 'essence' of a shape.

Another desirable, but not required, property of skeletons is that they are connected as this guarantees homotopy. In our case, we will focus on the Medial Axis Skeleton, which is a type of connected skeleton. This type of skeleton was originally introduced by Blum [7]. This skeleton is defined of the locus of centers of maximally inscribed discs in a shape. There are different ways to extract this medial axis, as detailed here[41]. Or more formally, suppose we have a shape $\mathcal{O}$ which has a boundary $\partial\mathcal{O}$. The *distance transform* $DT_{\mathcal{O}}$ is defined as

$$DT(x \in \mathcal{O}) = \min_{y \in \partial\mathcal{O}} \|x - y\|$$

The skeleton of $\mathcal{O}$ is subsequently defined as

$$S = \{x \in \mathcal{O} \mid \exists y, z \in \partial\mathcal{O}, y \neq z, DT(x) = \|x - z\| = \|x - z\|\}$$

In the continuous case the points of the skeleton are infinitesimally small, as they are single points. In practice, however, this is impossible due to the discrete nature of computers so we have to settle for 1 pixel thick skeletons. The contact points y and z are the points of the circle at x where it touches the boundary. These points are given by the *feature transform* FT of the shape, i.e. a map which associates each point in the shape with its closest point on the boundary. More formally,

$$FT(x \in \mathcal{O}) = \arg\min_{y \in \partial\mathcal{O}} \|x - y\|$$

The set S alone, however, cannot reconstruct $\mathcal{O}$ as the radii of the disks differ for each skeleton point $s \in S$. A full description of $\mathcal{O}$ is thus given by the Medial Axis Transform MAT of $\mathcal{O}$, i.e.

$$MAT(\mathcal{O}) = \{(s, DT(s)) \mid s \in S\}$$

Suppose $D(s, r)$ is a function that places a disc centered at s with radius r. The MAT then reconstructs $\mathcal{O}$ as

$$\overline{\mathcal{O}} = \bigcup \{D(s, r) \mid (s, r) \in MAT(\mathcal{O})\}$$

2.2. Computation

The definition of the skeleton is not constructive; it does not give us an algorithm to compute the MAT. Therefore there are various ways to compute the MAT based on various definitions of it. Some interpretations and approaches work better on some platforms than others based on the properties on the platform. There are two successful methods for computing the MAT based on the DT. There is one that works on regular CPUs and one that works on massively parallel architectures such as GPUs.

2.2.1. CPU-method

The CPU based method can be intuitively explained but is a mathematically hard problem and difficult to implement efficiently. Suppose we have some shape $\mathcal{O}$ which we consider as a patch of grass and its boundary $\partial\mathcal{O}$. Suppose we set the boundary on fire. The fire will burn isotropically from the boundary towards the interior of $\mathcal{O}$ with uniform speed. At those locations where these fire fronts meet will be the skeleton of $\mathcal{O}$.

It turns out that this is can be interpreted as solving the Eikonal Equation $|\nabla T| = 1$ with $T = 0$ on $\partial\mathcal{O}$. The Fast Marching Method (FMM) is an algorithm to solve this problem in $\mathcal{O}(n \log n)$ [34]. This finds the DT of $\mathcal{O}$ efficiently, by propagating a narrow band from the boundary inwards. Skeleton points as these are along singularities in the solved field. However, finding these singularities is no trivial task. It is numerically unstable to find these directly and can lead to false or missed skeleton points, which is undesirable.

The FMM was extended to the Augmented Fast Marching Method (AFMM) to overcome this problem [44]. Prior to solving the Eikonal equation, the boundary is numbered. A random point on the boundary is given the number zero, and is then increased monotonically until all points are numbered. This extra field U is propagated along the narrow band. Afterwards, for each point in $\mathcal{O}$ it is known which is the closest boundary point. Skeleton points are then the points whose difference with u values is larger than 2 as it is impossible for these points to originate from neighboring boundary points and thus have to distinct points FT points. This is illustrated in Figure 2.1. Skeleton points – with a bold border – are marked as such because their u values differ by more than 2.

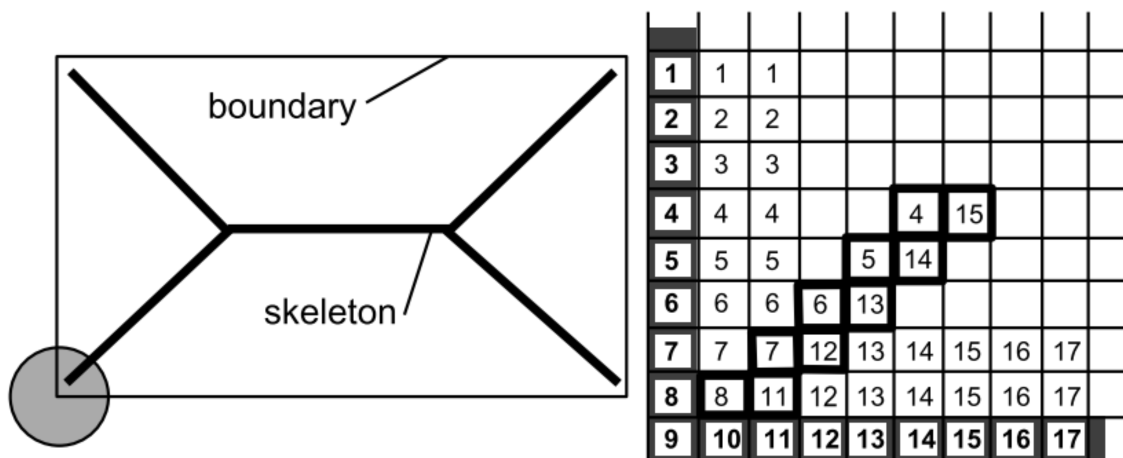


Figure 2.1: Visualisation of the AFMM algorithm. The boundary is monotonically initialized and propagated along the wavefront. Image from [44].

This method runs in $\mathcal{O}(N \log B)$ where N is the number of pixels of the shape's foreground and B is the boundary length of the foreground shape. In practice this is roughly $\mathcal{O}(N \log \sqrt{N})$.

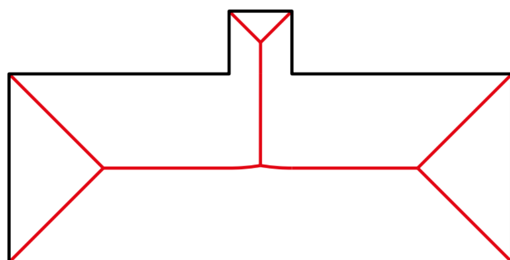
2.2.2. GPU-method

The GPU method is akin to the CPU AFMM method but modified to take advantage of the massively parallel architecture that GPGPU enables to do. It is based on the Parallel Banding Algorithm by Cao et al. [8]. This algorithm can compute the exact DT by a "sweep-and-merge" algorithm By dividing an image in bands, computing voronoi diagrams and merging these results intelligently the EDT can be obtained with very high performance as each band can be processed concurrently. Telea modified this algorithm to obtain the one-point FT from which a 1-pixel thick skeleton and the DT can be derived [42].

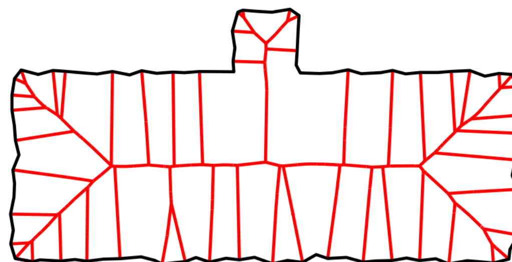
Due to its parallel nature, and because the time complexity is reduced to $\mathcal{O}(N)$, it is significantly faster than the CPU methods. It has been found that this method is in practice 20…80 times faster than the CPU method – or a few milliseconds for a 1024^2 image – thus allowing real-time manipulation of parameters.

2.3. Importance

While skeletons are very powerful descriptors, they have the downside that they are very sensitive to boundary noise. This is illustrated in Figure 2.2. The skeleton in Figure 2.2a is "simple". It has



(a) Skeletonization of a shape without boundary noise. The skeleton is "simple".



(b) Skeletonization of the same shape as in Figure 2.2a but with perturbations of the boundary. This generates a "complex" skeleton.

Figure 2.2: The influence of boundary noise on the complexity of the skeleton. Images from [41].

few branches and the number of skeleton pixels w.r.t. the area of the shape is low. The skeleton in Figure 2.2b on the other hand is not simple while the boundary is perceptually similar to that in Figure 2.2a. It contains many branches to encode the boundary noise, which is not desirable. It is therefore necessary to simplify either the shape such that these branches are not generated, or to simplify the skeleton and prune these branches.

Telea has presented a method to simplify skeleton paths based on simple metrics [43]. The *importance* measure ρx measures the importance of a skeleton point. This is defined by the length of the collapsed boundary between the two feature transform points on the boundary. Small perturbations on the boundary have a very small collapsed boundary length and should be eliminated. Thresholding the importance thus ought to remove the small noise. However, this also rounds off important corners of shapes, which is undesirable.

Therefore the *salience* metric is defined. The salience is based on two properties:

1. Salience is proportional with size. Longer features are more salient than others
2. Salience is inversely proportional with thickness. Features on thick objects are less salient than features on thin objects.

The salience is defined as $\sigma(x) = \frac{\rho(x)}{DT(x)}$. Thresholding this metric will remove small perturbations on the boundary while removing boundary noise.
In short, this means that we can compute the skeleton, with DT and FT of every shape

Very quickly A skeleton is computed in the order milliseconds.

Robustly It always generate 1-pixel thick skeletons for every 2D shape.

Regularized Noise is eliminated robustly and intuitively, while maintaining important shapes.

3
Related Work: Skeleton Image Coding

Skeleton Image Coding was introduced in the thesis by Meiburg [26]. In this work, he proposed the idea of encoding shapes based on skeletons. He encoded images by generating an upper-level set segmentation of a gray scale image to obtain a set binary images representing the original image.

These sets are skeletonized to obtain a set of Medial Axis Transforms, which can be encoded into a container file. Reconstruction of these sets happens by reconstructing each MAT from a low thresholds to high thresholds on top of each other. The reconstructed pixels obtain the intensity of the highest threshold set they appear in. In order to prevent boundary effects, Meiburg's framework offer interpolation options.

Meiburg's framework provided a few parameters:

3.1. Layer removal

Meiburg realized that encoding *all* upper-level sets will not be fruitful as these contain too much and, most important, redundant information. Therefore he posed that many layers can be removed without altering the final image too much. In order to do this, a global threshold is set on the histogram. All intensities that have a pixel count of $\geq \psi$ remain unaltered, and all intensities with a pixel count below this parameter are not skeletonized, thus darkened to the nearest intensity below it.

3.2. Skeleton simplification

Meiburg's computes salient skeletons as explained in chapter 2 using the CPU AFMM method. Meiburg recognized that this can generate disconnected skeletons for each shape and retaining each skeleton branch is expensive. Therefore only the *largest* skeleton is retained. Also, he removes skeleton paths that are either too short – as these are too expensive to maintain or do not encode a large enough area. If the area reconstructed by the skeleton path is small, it will be hardly visible and thus space can be retained by removing that path.

3.3. Skeleton path encoding

Meiburg provides a sparse encoding of skeleton paths using trees. From each upper-level set only MAT is retained rather than the full layer of skeleton pixel and non-skeleton pixels. This is a space-saving measure as it stores less redundant information.

3.4. Reconstruction

To overcome boundary effects, interpolation between reconstructed layers is necessary. Meiburg achieves this by modifying the alpha values near the boundary and has different schemes for this. If only the alpha within the shape is modified, this alters the size of a shape. Therefore there is another reconstruction that makes the border have size b where interpolation from 100% to 50% alpha happens within the border from the original border to $\frac{b}{2}$ pixels within the shape, and the interpolation from 50%

to 0% happens from the border to $\frac{b}{2}$ pixels outside the shape. Therefore, sizes of shapes remain equal and a smooth transition between boundaries happens.

3.5. Results

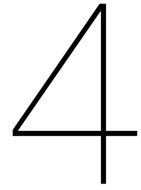
The results of Meiburg's framework were promising but unfortunately not up to par compared to JPEG. The quality was too low, the files too large and computation time was too high. An example is in Figure 3.1. As one can see the quality is acceptable, although there some significant errors, but the file size is thrice that of a better JPEG result, and was computed in 50 seconds or more, rather than < 1 second to compute a better JPEG image.

(a) A result of Meiburg's pipeline using the "Lena" image. The file size is 172kB.

(b) A result of Meiburg's pipeline using the "Mandril" image. The file size is 196kB.

Figure 3.1: Some results of Meiburg's pipeline.

However, their pipeline and method shows great promise and improvement opportunities. So rather than creating a new pipeline, we will study their methods and parameters and introduce new steps where necessary.



Information Theory & Compression

When discussing compression, it is important to note that there are finite limits to what is possible for a general compression algorithm. It is a sensitive set of scales with quality of the signal on one end and size of the signal on the other; as the size of a signal decreases, the other side can only rise so far before the quality of the signal starts to drop. To explore where these tipping points are, it is necessary to discuss what compression entails *in general* before we can apply it to dense skeletons.

4.1. What is information?

It is important that before we try to apply compression what it *means* to compress something. The obvious interpretation is that we take an object which occupies n_1 bytes and we try to store it as n_2 bytes where hopefully $n_2 < n_1$. However, this is not the full extent of what compression encompasses. Here we try to fully extend the meaning of compression so we can achieve an optimal result when we try to compress dense skeletons – aside from learning what optimal *means* in terms of compression.

The basis from compression comes from the central paper in signal processing and mathematical communication *"A Mathematical Theory of Communication"* by Claude E. Shannon [37]. In this paper, he provides a mathematical description for communication which he aptly describes as:

> *The fundamental problem of communication is that of reproducing at one point, either exactly or approximately, a message selected at another point*
>
> *Claude E. Shannon*

This quote already introduces a subtlety to our previous interpretation of compression: It is not about taking a set of values and store in as few bytes as possible, but about finding a signal which enables the receiver to *reconstruct* the original message with as few information as possible. However, this only seems to introduce more questions: What signal will enable this reconstruction of a message? How do you communicate information? How do we measure information?

We can approach this problem by introducing a statistical model. Suppose there is some alphabet $\mathcal{A} = \{\alpha_1, \alpha_2 \dots, \alpha_n\}$ which describes which symbols can be encountered in a message along with probabilities $\mathcal{P} = \{p_1, p_2, \dots, p_n\}$ providing the occurrence probabilities of the corresponding of symbols ($\sum_{p \in \mathcal{P}} p = 1$). Now how can we extract information out of this model? One way to define this information is by *predictability*. If an information source is very predictable, we hardly ever learn new information. One such predictable system is a coin with two heads. Since we know that the result will always be same, obtaining that result does not give us any new information thus the expected value of information of that message is 0. We can say that these events have 0 bits of information. An unpredictable event, however, carries the most information. In the case of a fair coin flip – which has both heads and tails – we will never give a prediction of what the next result is going to be. Then, we will be correct 50% of the time, on average. Since this event carries most information, it contains maximum entropy, or one bit of information.

Moreover, information is additive. The information that events mn are happening, ought to be same

information content that event m is happening and that event n is happening, i.e. $I(mn) = I(m) + I(n)$[1]. To recap, we now have:

1. $I(p) \geq 0$. Each event carries at worst no information
2. $I(1) = 0$. Events that always or never occur carry no information
3. $I(mn) = I(m) + I(n)$. Information is additive.

Shannon has proven in his paper that the only logical definition for $I(p)$ is $I(p) = -\log_2(p)$. Now N events happen according to probability density function $\mathcal{P}$. Then the total information – on average – received $\mathcal{I} = -\sum_i Np_i \log_2(p_i)$. Therefore, the average information each event yields – also called the entropy – is

$$\mathcal{H}(\mathcal{P}) = -\sum_{p \in \mathcal{P}} p \log_2(p)$$

.

So now we have a measurement of information given a probability density function. We can now determine for every sent signal how much information is transmitted and how much is redundant – i.e. information sent with $\mathcal{H}(\mathcal{P}) = 0$. So how can a signal now be compressed from an information theoretical perspective? One can either use the knowledge of the distribution of the signal or transform the alphabet of the signal to find a better suited one such that the information content is smaller. For example, suppose English text needs to be transmitted. A regular ASCII table comprises of 128 different characters which all have the same probability of $I(x) = \frac{1}{128} = 0.0078125$ bits. The entropy of the dataset is thus $\mathcal{H}(\mathcal{A}) = -\sum_{i=1}^{128} \frac{1}{128} \log_2 \frac{1}{128} = 7$. Therefore, there are 7 bits per character necessary to encode the dataset. However, the English language uses only a subset of the glyphs in the ASCII table – i.e. lower- and uppercase letters as well as some punctuation characters. This can significantly reduce the range of symbols needed to transmit to the receiving party. Moreover, the characters of the ASCII table are not uniformly used in the English language; the letter 'q' is hardly used at all and the letter 'e' is the most common. Research[19] incorporating this information have tried to compress classic literary works and found that these works have an entropy of about 1.58 bits per (printable) symbol (tested on a corpus of 20.3 million characters).

4.2. Limits

The measure of entropy has given us a valuable estimate on how much space we need for a message. For a message of length N Shannon estimates we need $N \cdot \mathcal{H}(\mathcal{A})$ bits. However, there are some limits until how far we can compress something.

First off, there are some messages that cannot be compressed using some method. Suppose that there is some file of x bits and some function $c(x)$ that maps x to b bits. Surely, there are only $2^{b+1} - 1$ possible messages if $b < x$. However, there are 2^x possible messages and since $2^{b+1} - 1 < 2^x$ there is at least one message for which $b > x$.

Now suppose there is an encoding $\mathcal{C}$ which maps an ensemble $\{\mathcal{A}, \mathcal{P}\}$ to $\{0, 1\}^+$. A message is uniquely decodeable iff $\forall x, y \in \mathcal{A}^+, x \neq y \rightarrow \mathcal{C}(x) \neq \mathcal{C}(y)$[24]. If an encoding is uniquely decodeable then $\forall x \in \mathcal{C}, |\mathcal{C}(x)| \in [\mathcal{H}(x), \mathcal{H}(x) + 1)$, i.e. the entropy of a message is a lower bound for the expected message size. Therefore we shall call an encoding optimal if the expected code size is as small as possible – i.e. the entropy – while it is still possible to decipher it.

However, all these statements only hold for *lossless* encoding. If we are willing to accept an error one *can* below the Shannon entropy as the entropy of the original dataset decreases. If one is not willing to sacrifice information in order to obtain a better compression one is bounded by the entropy of the signal.

4.3. Techniques

The work of Shannon has given us a lower bound on the size needed to encode a message, but it has not given us a way to *construct* such code. However, there have been several methods constructed over the past decades that attempt to encode a signal in a (near-)optimal way.

[1]This assumes that m and n are both i.i.d.

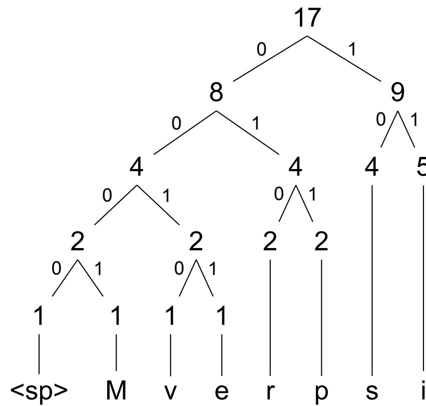


Figure 4.1: The Huffman tree of the sentence “Mississippi river”

Table 4.1: Comparison of ASCII and Huffman codes for the sentence “Mississippi river”

Symbol	Huffman Code	ASCII code
<sp>	0000	00100000
r	010	01110010
v	0010	01110110
e	0011	01100101
M	0001	01001101
p	011	01110000
s	10	01110011
i	11	01101001

These can crudely be defined by three classes: fixed-length code, variable length codes and universal codes. In a fixed-length code, each symbol is encoded with a fixed number of bits.

Fixed-length codes are mainly used when either the PDF is uniform, when very large blocks of data are compressed or when there is a non-zero probability of failure in communication or encoding of the symbols.

A variable-length code, however, assigns each symbol a variable number of bits. It is proven that there exists strategies that can compress signals arbitrarily close the its entropy. A particular type of variable-length encoding are prefix codes. This is an encoding which has the requirement that no code word of a symbol is the prefix of another code word in the code book. Therefore, the code book $\{7,42\}$ is a prefix code whereas $\{7,42,78\}$ is not because the code 7 is a prefix to 78. These code books can reach entropy-sized compression on a symbol basis.

The third class are universal codes. These are also prefix codes that can map integers to binary codes Below are several of such methods.

4.3.1. Huffman Coding

Huffman coding was one of the first optimal prefix codes discovered by David A. Huffman[22]. It can transform a signal using its histogram to a decodeable, compact representation using a surprisingly simple and elegant algorithm in linear time. It is based on Shannon-Fano coding which Shannon proposed himself in [37] which constructs a binary tree from the histogram in a “top-down” approach by recursively splitting it in two subsets of (near-)equal weight. This method, however, does not always generate optimal codes whereas Huffman Coding does. Rather than approaching the problem in a top-down method, Huffman proposes a “bottom-up” method. It constructs a frequency-sorted binary tree in the following way: Suppose we start with N leaf nodes which contain a symbol and its associated frequency. Find the two nodes which have the *lowest* frequencies f_1 and f_2 and merge these into an internal node with associated frequency $f_1 + f_2$ and has the previous two nodes as children. This process is repeated until only one node remains. Codes for each symbol are then obtained by traversing this tree, adding a $'0'$ to the prefix for each left child and a $'1'$ for each right child. In Figure 4.1 example for the sentence “Mississippi river”. Counting carefully shows that encoding the original sentence using

Huffman Coding takes 46 bits – or an average of about 2.70588 bits per symbol as opposed to 136 bits using ASCII at 8 bits per symbol. This is slightly above the entropy of the message, which is about 2.69866 bits per symbol. However, this does not encode the code book which is also necessary at the receiving end to decode the message. This can be circumvented by fixing the dictionary – at the cost of introducing redundancy – or also transmitting the dictionary – at the cost of sending more bytes. There are several methods to overcome this, rather than naively transmitting the dictionary. One is to use the Canonical Huffman Coding variant, which can encode the dictionary in $B2^B$ bits with B the number of bits of a symbol. This is already much better than the naive dictionary transmission approach. Another method is to send the tree rather than the code words and let the client deduce the codes. If the message is large enough, this is the preferred method as it has very little overhead compared to the message length.

There are a few large downsides to this method. One is that it implies that the alphabet and its distribution for a message is known *before* the encoding process. While this seems like a fairly innocuous assumption, there are many situations where this is not the case. Suppose one wants to send a message which is too large to fit in the sender's memory. In that case one cannot fully determine the frequencies of each symbol, and therefore not construct a Huffman tree. There is an amended version of Huffman coding which does not require the frequency count to be available beforehand and can determine it during encoding. This variant, called Adaptive Huffman coding, does not necessarily generate an optimal encoding.

Another is that it is only optimal when considering symbol-by-symbol encoding, which results in an optimal encoding if the symbols are i.i.d. In many situations this assumption fails to hold, thus resulting in larger-than-optimal encoding. A method which can handle this situation better is Arithmetic Coding, discussed in the next section.

Due to the properties of a binary tree, optimality can only be guaranteed if the probability of each symbol follows the function 2^{-l} for some l. If this is the case, then the Huffman tree will approach a full binary tree which has height $\log_2 N$ for N leafs. Therefore, the maximum symbol length is $\lceil \log_2 N + 1 \rceil$. For any other tree, and therefore any other distribution, Huffman coding can result in longer code words. This is especially a problem with short alphabets.

4.3.2. Arithmetic coding

Arithmetic Coding is a method that attempts to alleviate the downsides that can occur with Huffman coding, but intends to be just as optimal in situations where Huffman Coding shines. However, due to its late invention in 1987[53], complex implementation, and possible patent coverings, it has not been as popular or fully replaced Huffman coding while superior. Arithmetic Coding tries to capture an entire message in a fraction. This works in the following way: Suppose the half-open interval $[0, 1)$ and symbol distribution $\mathcal{P}$ such that $\sum \mathcal{P} = 1 \wedge \forall p \in \mathcal{P}, p > 0$. We can divide this interval according to $\mathcal{P}$. Now when a symbol is encountered we can shorten our interval to the higher and lower bound that define that symbol. This new range is again subdivided according to $\mathcal{P}$, until all symbols are encoded. What is left is a lower and higher bound. This entire region is capable of uniquely and fully encode the original message.

For example, consider our previous message of "Mississippi river" as in Huffman Coding. This method is superior to Huffman coding because it not only is symbol-optimal, but is also signal-optimal; It will compress the whole *signal* better due to that certain symbols can be represented with non-integer bits.

However, it shares parts of the same downsides as Huffman coding. It is also required to know the probability of each symbol beforehand. While there is also an adaptive version – where each interval starts out as the same length and is rescaled as symbols are presented – this will not always yield an optimal encoding. Also, implementation-specific details need to be considered. Current computers now often host 32-bit floating point numbers, which are used to define the interval $[0, 1)$. Whenever an interval becomes smaller than the machine precision, it can no longer be represented thus disappearing and rendering an incorrect encoding. Without proper countermeasures, one can encode up to 15 symbols at most on current machines. If one wants to encode longer symbols the interval can be *re-normalized*. Moreover, a very good model is now mandatory otherwise it will perform poorly.

4.3.3. Finite-state Entropy

Entropy coding has been a notoriously slow field regarding new advancements, mostly because it is very hard to create a new optimal entropy coder. This is why Huffman Coding was invented over sixty years ago in the early fifties and Arithmetic Coding was the first real improvement on this, which was invented somewhat 30 years later. However, there has been one recent development with the discovery of finite-state entropy[12]. It attempts to achieve the same superiority that Arithmetic Coding enjoys while having the performance of Huffman Coding. This has lead to the Zstandard algorithm which was published by facebook in the summer of 2016.

4.3.4. Run-length coding

While previous methods were methods to encode a signal with as few entropy as possible it is also possible to modify the signal while still meaning the same. For example, consider the signal `AAAAAAAAAAABBABAAAAAAAAAABBBBBBBBBBBBBC`. There are a lot of repeated elements in this signal and therefore a lot of redundancy. This signal can be broken down in several "runs" of consecutive identical characters as follows: `AAAAAAAAAAA BB A B AAAAAAAAAA BBBBBBBBBBBBB C`. Run-length encoding defines a signal by a set of such runs and encoding these by first denoting the length followed by the symbol of that run. For this signal, this would become `11A2B1A1B10A13B1C`. This is significantly shorter than the original signal, and becomes more effective as runs become longer. It is one of the techniques used in JPEG to reduce signal size as it tries to introduce long runs of consecutive zeros for its high frequencies. Especially used in conjunction with one of the previous two techniques this is a profitable way to encode a signal.

4.3.5. Universal methods

Whereas Huffman- and Arithmetic coding requires you to know the exact distribution, there are also methods which provide good results when only the approximate distribution is known. For example, when one only knows the *ranks* of the symbols in order of occurrence, Universal coding might provide an outcome. Formally, they are a mapping of integers to prefix-free binary codes. Usually these methods impose a distribution such as the distribution 2^{-n}. Universal codes have the desirable property that if one imposes a monotonic probability distribution $\mathcal{P}$ on the set of integers, then for all code length c it holds that $\epsilon C(\mathcal{P}) \geq c$ for some value of $\epsilon \geq 1$ and some function C that gives the optimal encoding of a probability distribution. Or in other words, the length of each code word is bounded by length of the corresponding optimal code word up to some constant. This constant can be made arbitrarily close to 1 by encoding larger blocks of data. As a result, these encoding schemes are often used in audio encoding methods (such as Apple Lossless, FLAC) and video encoding methods (such as H.264, H.265, and MPEG-4 AVC) as well as some image formats as FELICS and JPEG-LS. There are several of such methods such which we will describe shortly.

Unary coding

Unary coding is arguably the simplest method of Universal coding. It is akin to counting with ones fingers: You write as many ones as fingers are up, and terminate it by a zero. Or more formally, to encode an integer n one writes n bits of value one and one extra 0 to terminate the sequence. This sequence is optimal for the discrete probability function $\mathcal{P}(n) = 2^{-n}$. This encoding is used in UTF-8 coding of Unicode symbols and is often used neural network training.

Exp-Golomb coding

Exp-Golomb – or Exponential-Golomb coding – is another universal code which can encode any non-negative integer. The algorithm to encode integers is the following:

- Represent $n + 1$ in binary – this has length $l(n + 1)$
- Write $l(n + 1) - 1$ zeros as a prefix to the previous number.

This code is obviously a prefix code which makes it desirable. This code is also used in H.264 video encoding to encode inter-frame motion vectors[31]. The fact that small values can be written compactly makes it an attractive code. However, as a trade-off, one can shorten the codes for larger integers at the cost of slightly larger small integers. This can be useful if one already knows the range of output values and wishes to shorten the codes at the edge of range while maintaining shorter codes

Table 4.2: Different encodings of natural numbers

Number	Binary	Unary code	Exp-Golomb code
0	00000000	0	1
1	00000001	10	010
2	00000010	110	011
3	00000011	1110	00100
4	00000100	11110	00101
5	00000101	111110	00110
6	00000110	1111110	00111
7	00000111	11111110	0001000
8	00001000	111111110	0001001
9	00001001	1111111110	0001010

for smaller integers. This then becomes the order-k Exp-Golomb code as opposed to the order-0 code as described above. The code is then calculated as follows:

- Encode $\left\lfloor \frac{x}{2^k} \right\rfloor$ using the order-0 Exp-Golomb method
- Append $x \mod 2^k$ to the previous number in binary

In Table 4.2 the non-negative numbers smaller than 10 are encoded in binary, unary and order-0 Exp-Golomb. Note that none of the methods are naturally capable of encoding negative numbers. Luckily, the set $\mathbb{N}$ can be bijected onto $\mathbb{N}_+$ by mapping each positive number n to $2n + 1$ and each negative number to $-2n$. This allows mapping the sequence $(0, -1, 1, -2, 2, -3, 3, ...)$ onto $(1, 2, 3, 4, 5, ...)$. However, this comes at the cost that each positive number now costs twice as many bits as before. And although it is not required to know the distribution beforehand, best results are obtained if the symbol sequence is drawn from a geometric distribution (i.e. Pr(k) $= (1 - p)^k p)$).

4.3.6. Prediction

So far, earlier systems assumed that samples were i.i.d. drawn from some distribution. However, this is very often not the case and this property can be exploited. Suppose we have some predictor available at both ends of the communication channel that can based on the history of written symbols predict what the next one is going to be. If this is a perfect predictor, i.e. one that makes no mistakes in guessing, we do not have to transmit anything but the first symbol as there is after that no new information generated. However, suppose that we have a very good predictor that can guess most of the time correctly. In that case, we are able to significantly ease the encoding of the information. For example, rather than encoding the new state one can encode the difference between the actual next state and the predicted next state. If it is a good predictor, this difference will often be zero and when it is wrong the difference shall be small. Bad predictors are either often wrong, or the difference is very large, or both.

A prediction scheme based in intra-frame macroblocks is successfully used by Google's VP8 video format and WebP image format[6].

It is essential that the predictor has an accurate model to ensure proper prediction. If there is no accurate model there cannot be an accurate prediction. One the more recent methods of acquiring a model is by the Prediction by Partial Matching – or PPM – algorithm. PPM tries to predict the next symbol according to a N-th order statistical model. If it fails to give a good prediction, it reduces itself to a $N - 1$-th order statistical model all the way down to an order-0 model unless a good fit is found. This iterative search for the best prediction is attempted at each symbol, which makes this algorithm rather expensive. In practice, Markov models are used to predict the data. As soon as a prediction is made, this result is entropy coded with, for example, arithmetic coding.

4.3.7. Compaction

Compaction is another efficient and simple compression algorithm based on byte-pair coding[16]. The original algorithm intents to replace consecutive bytes with a new, unused byte. This yields a size improvement if a repetition occurs often enough because it also needs to be encoded which byte represents which two original bytes. If byte pairs are considered one new byte can thus represent two

other bytes for a total cost of three bytes. That means there is a profit if one byte pair occurs more than twice. Consider the following signal: “ABABCABCABD”. If a sliding window of size 2 moves over this signal the following consecutive byte pairs are found: ‘AB’, ‘BA’, ‘AB’, ‘BC’, ‘CA’, ‘AB’, ‘BC’, ‘CA’, ‘AB’, ‘BD’. We can see that the byte-pair ‘AB’ most often so we replace it a new byte E. This yields the signal “EECECED”. We could say that we are now done but this algorithm can be applied again on this signal. We can see that the pair ‘EC’ again occurs twice and can therefore replace it again with a new byte ‘F’. This yields the new signal “EFFED”. This is the final signal and can be transmitted along with the code book which stores pointers to the replacement table.

If one replaces the restriction of 2 consecutive bytes to a sliding window of n bytes one obtains an algorithm very similar to Lempel-Ziv compression[54]. History is tracked back to to see if elements are repeating. If they are, a marker is inserted how many characters need to be read with an offset on how many characters back in the stream that was. This is very useful if one does not know the alphabet beforehand. If one does, one can use the Lempel-Ziv-Welch algorithm instead[52]. This starts out with a basic dictionary and extends it with each run it finds which is not yet in the dictionary.

Now that we have a solid understanding of what compression means, can achieve, and have discussed several techniques of reaching the (near-)optimal case of compression, we can try to apply it to dense skeletons. A takeaway from all compression methods is that we need to find and eliminate redundancy. This has been the point for general compression methods for the past decades. We can see that in all methods it is attempted to impose a statistical model on the data. If there *is* redundancy it should follow from this model which ought to make it possible to remove it or encode it efficiently. We are in the advantage here since we *know* beforehand what our messages will look like. As we can recall from chapter 2, skeleton paths are connected meaning that differences in locations can be regular. This could indicate good compression prospects for techniques as run-length encoding or compaction. These results could be further entropy coded to binary code words using a simple technique as Huffman coding.

4.4. Assessing compression

After compression is done it is often useful to know how well a compression scheme performed compared to the original signal or other compression schemes. One common metric is the compression ratio which is defined as

$$\text{DC} = \frac{\text{Uncompressed size}}{\text{Compressed size}}$$

While this gives the direct *output* performance of a compression algorithm it may not be the most useful because it does not take compression time into account. This might be important because for some applications the compression time is critical for certain goals or a satisfactory user experience.

The Weissman score is a recently developed which takes both space saving and time into account. While created as a bogus measure to give a realistic feeling to the comedy show *Silicon Valley* it turns out that this can be a useful metric. It is defined as

$$W = \alpha \frac{r}{r'} \frac{\log T'}{\log T}$$

where r is the compression ratio, T is the compression ratio, the primed counterparts are the same features of a competing algorithm and α a scaling constant.

4.5. Image compression & Quality

Image compression is about as old as the creation of digital images itself. Earlier image compression methods as PackBits, TIFF, and GIF were not very advanced image compression algorithms in terms of compression quality, flexibility, or image quality. JPEG was an algorithm that was created later and is still one of the most popular to this date. We will review it in detail so we can determine its strengths and weaknesses.

4.5.1. JPEG

JPEG was created in 1992 and was one the first sophisticated image compression algorithms specifically designed to compress natural images and take human perception into account. The standard

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 4.2: The zig-zag encoding of JPEG. Notice the long runs of zeros.

published by the Joint Photographic Experts Group describes how to efficiently encode an image into bytes and how to decode it back into an image. What it does *not* describe is an image file format, which is described separately in the JFIF (JPEG File Interchange Format) standard.

JPEG compresses images based on two assumptions on human vision:

1. Humans are not very sensitive to changes in color
2. Humans are bad at distinguishing high frequency details

These two assumptions allow for for two techniques to come in play to allow image compression. One is that since we cannot distinguish color differences very well we do not need to store color information in the same fidelity as intensity information. JPEG exploits this by performing *chroma sub-sampling*; color information is stored in half or a quarter of the original resolution. Changing merely the colorspace rather than a combination of the intensity and colorspace as RGB exposes, the image is converted to the YCbCr colorspace which has a intensity channel (Y) and two *chroma* channels – the difference in red (Cr) and the difference in blue (Cb). The latter channels are thus sub-sampled and stored at lower resolution.

Each channel is subdivided in 8×8 *macroblocks*. Each of these blocks is processed and stored independently of the other blocks. Rather than storing the blocks directly they are processed using the Discrete Cosine Transform (DCT). This decomposes the signal into a sum of cosines of different amplitudes and frequencies. For each macroblock M the DCT transformed block M' is defined as

$$M(u,v)' = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} M(x,y)\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right]$$

with u and v the relative coordinates within the DCT macroblock, and $\alpha(x)$ normalizing scale factors such that $\alpha(x) = \frac{1}{\sqrt{2}}$ if $x = 0$ and 1 for other values of x. Since we observed before that humans cannot distinguish high-frequency intensity changes, the amplitudes of those frequencies are quantized to zero. Other amplitudes are also rounded and quantized. After the DCT, the DC component and amplitudes corresponding to lower frequencies will be concentrated in the top-left corner while high-frequency amplitudes are towards the bottom-right corner. To encode the block it is traversed in a “zig-zag” fashion to have most information in the beginning of the signal while having long runs of zeros towards the end of the signal. This is illustrated in Figure 4.2. This is then encoded using a custom run-length encoding scheme which is subsequently further compressed using an arithmetic coder – the most common being Huffman coding.

Removal of high-frequency intensity components and an efficient encoding scheme of these intensities are what makes JPEG very successful at image compression with a low psycho-visual error. However, they are also the source of a very sudden degradation of quality when the compression factor is too high. Sensitive intensity changes are deleted resulting in an abruptly high psycho-visual error.

These are points which dense skeleton image coding can attempt to beat while possibly providing more graceful image quality degradation.

4.5.2. Image Quality

As mentioned before, there is a sensitive trade-off between file size and quality when compression is concerned. Size, on the one hand, is easily quantifiable and compared across different signals. Quality, however, is not as easily quantified or compared across signals, especially image quality. In simpler signals – i.e. a sequence of letters – one can compute the reconstruction error if one knows the original signal. This reconstruction error is a simple and effective measure for quality.

A naive extension might be to apply the same to images. That is, measure the sum of squared differences for each pixel. This can be a measure for quality but it is, however, not a very good one. It fails to take into account the human visual system. Consider Figure 4.3. Figures 4.3b, 4.3c, 4.3d, 4.3e and 4.3f have about the same reconstruction error as measured by the MSE compared to Figure 4.3a. It is, however, easily seen that the image quality of Figure 4.3b is higher than that of Figure 4.3f despite their seemingly same perceptual image quality. Therefore, we need a more advanced system to objectively judge the perceptive image quality which *does* take into account the human visual system.

It turns out that humans are excellent judges of images quality even when no reference image is present[48]. When presented an image they can give an opinion score regarding the quality of the image. The mean opinion score (MOS) of each image can then be considered a ground truth of image quality. The downside of this approach is, however, that asking humans to judge thousands of images is very time-consuming and taxing on the psychological well-being of the judges. Therefore, to adequately judge the image quality compared to some ground truth image, it is required that this score correlates with the MOS.

The state of the art method of assessing image quality is Multi-Scale Structural Similarity (MS-SSIM) [50]. This is an extension of the original SSIM metric[51]. This was already an advanced top-down interpretation of the human visual and compares two images based on degradation of structural information of some image with respect to a ground truth. It gives a score to an image between 0 and 1 based on a luminance component, contrast component, and a structure component. These are, respectively defined as

$$\begin{aligned} l(x,y) &= \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \\ c(x,y) &= \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_1}, \\ s(x,y) &= \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \end{aligned}$$

with $\mu$the means of the respective images, $\sigma$the standard deviation, σ_{xy} the correlation, $C_1 = (K_1L)^2$, $C_2 = (K_2L)^2$, and $C_3 = C_2/L$. In these equations L is the dynamic range of the image and K_1 and K_2 are small constants. These are approximated by the mean of the images, the standard deviation of the images, and the correlation between the images, respectively. In total it is computed by

$$\text{SSIM} = [l(x,y)]^\alpha \cdot [c(x,y)]^\beta \cdot [s(x,y)]^\gamma = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

when $\alpha = \beta = \gamma = 1$. There are still some limitations to this system, however. As the human visual system can be considered a non-linear, multi-scale system the current SSIM metric does not hold well when images are compared at different resolutions or at angles. Moreover, as the human visual system is highly non-linear, detection of features and important structures is also poorly approximated by a single linear system. To do this correctly, a multi-scale approach is required.

As mentioned before, the MS-SSIM is a multi-scale extension of the SSIM introduced to alleviate previously mentioned objections and to provide a metric with higher correlation with the human visual system. In the case of the MS-SSIM, the image is M times sub-sampled and down scaled and the

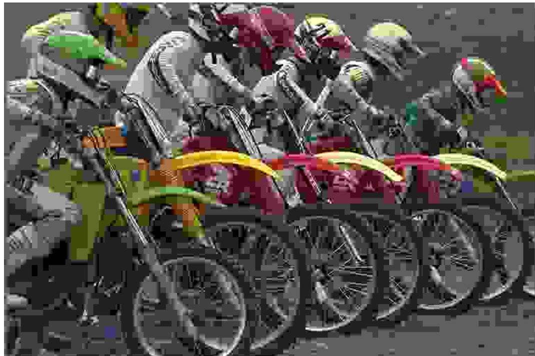
[2]Images courtesy of VideoClarity (`http://videoclarity.com/videoqualityanalysiscasestudies/wpadvancingtomulti-scalessim/`)

(a) Original image

(b) Figure 4.3a with a contrast enhancement.
MSE = 74. MS-SSIM=0.9956

(c) Figure 4.3a with a Gaussian blur filter.
MSE = 75. MS-SSIM=0.6609

(d) Figure 4.3a with a Gaussian noise.
MSE = 74. MS-SSIM=0.9592



(e) Figure 4.3a with JPEG compression artifacts.
MSE = 78. MS-SSIM=0.6609

(f) Figure 4.3a with salt and pepper noise.
MSE = 75. MS-SSIM=0.4145

Figure 4.3: The downside of using errors as a measure for quality. All modified images have the same error but vary wildly in image quality. This is because the human visual system is not taken into account with naive quality measures. [2]

contrast and structure component are computed M. The product of these components, in addition to the luminance factor are the final quality score given to the image. So all in all it is computed as

$$\text{MS-SSIM} = [l_M(x,y)]^{\alpha_M} \cdot \prod_{j=1}^{M} \left[c_j(x,y)\right]^{\beta_j} \cdot \left[s_j(x,y)\right]^{\gamma_j}$$

where each jth component is sub-sampled and down scaled j times. This provides a good correlation with mean opinion scores on the LIVE image database[38], as visible in Figure 4.4.

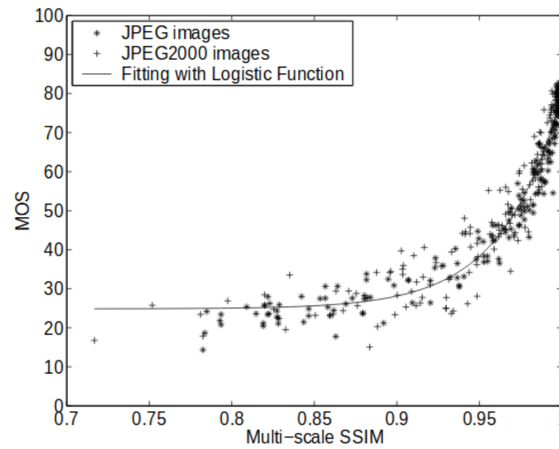


Figure 4.4: Logistic fit of MS-SSIM scores with the MOS of all images in the LIVE image database.

5

Skeleton Compression

Now that we have a basis of what skeletons are, how to compute them – as described in chapter 2 – and what compression entails – described in chapter 4 – we can discuss image coding and compression. From a very high-level, we need a pipeline that accepts and input image, performs some skeletonization steps, and returns a file which can be reconstructed into an image. There are, however, certain important intermediate steps whose influence on the final result cannot be underestimated. Our full pipeline is visible in Figure 5.1. It accepts a conventional raster image image which is encoded into a SIR file (Skeleton Image Representation). This is produced by the method of Meiburg as described in chapter 3. This SIR file can subsequently be reconstructed into a conventional raster format. Each step in the pipeline can be influenced by some parameters and have some influence on the final result. These will be discussed separately.

5.1. Pre-filtering

As Meiburg [26] noticed is that when an image is thresholded in an upper-level set, noisy edges are introduced. These noisy edges create a lot of small objects surrounding larger objects. This means that relatively a lot of skeleton points are "spent" on a small surface area, which is problematic for image compression. Moreover, as these regions are small, they are relatively unimportant as they are hardly visible.

It is therefore necessary to remove these small structures. This is performed by an area opening filter which removes small "islands". A connected component labeling of the image is made, such that connected "islands" smaller than a certain size are inverted. This certain size can either be expressed as a fixed number of pixels or as a percentage of the image dimensions. A typical value for this parameter is to invert islands smaller than 10 pixels or $\approx 1 ... 5\%$ of the image dimensions.

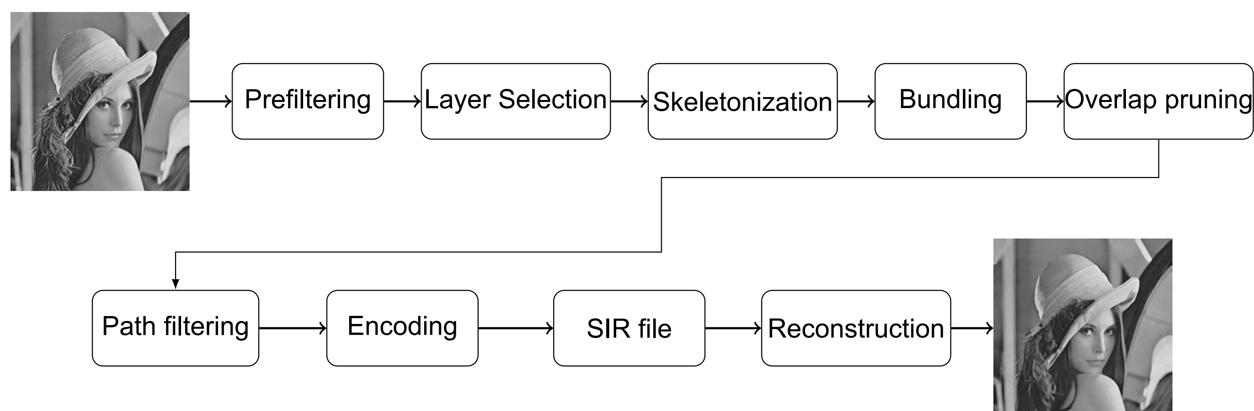
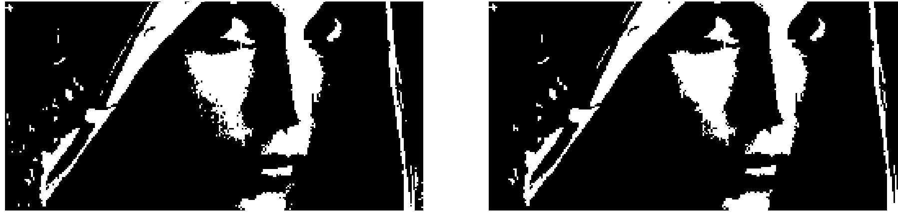


Figure 5.1: A high level overview of our encoding scheme and SIR file viewing. A conventional image is the input and a skeletonized representation is the output.



(a) Upper level set of Lena at 155. Note the noisy boundaries.

(b) Island filtering of Figure 5.2a using a size of 5 pixels – or $\approx 1\%$ of the image dimensions. Notice that there is less boundary noise.

Figure 5.2: The effect of island filtering on an upper level set. Notice that black is foreground in this image.

5.2. Layer selection

One of the main observations of skeleton image compression is that many layers are not relevant enough to encode. These layers, for example, contain only non-salient shapes, or other shapes which are already encoded by other layers. Therefore we do not need to encode every layer that is in the original image. Since we know that the digital shape of every layer is embedded in the upper-level sets below it, we can remove it without too much loss in the resulting image. After all, if we remove a layer then the pixel intensity is reduced to the layer below it. Therefore, we slightly darken the pixel which can be called an acceptable loss.

It is now key to remove the correct layers because if we remove too many important layers, we would end up with an image that is too distorted whereas if we select too many layers we end up with an image containing a lot of redundant information which is too large. There are several approaches to define importance, of which we give a subset.

Meiburg[26] selects relevant layers by histogram thresholding. Their justification is that layer importance is directly measurable by the number of pixels that have exactly that intensity, which is what the histogram represents. This histogram, $\mathbb{H}$, is subsequently L_∞-normalized i.e.

$$\mathbb{H}' = \frac{\mathbb{H}}{\max \mathbb{H}}$$

This means that the layer with most pixels is adding most to the image, so its value is 1. All other layers have an importance $0 \leq i < 1$. Therefore, this new histogram $\mathbb{H}'$ is easily and intuitively thresholded by some value t such that a layer set $\mathbb{L} = \{h \in \mathbb{H}', h \geq t\}$ is obtained. While this is a reasonable measure of importance and allows one to delete numerous layers it is not perfect. This method often rejects small, but visually important features such as highlights. These are often small regions, thus have very few pixels but are important visual cues thus resulting in a higher reconstruction error. Besides, to obtain an aesthetically pleasing result one has in practice to set the threshold such that at least 100 layers are selected, resulting in a large file size.

Another method described by van der Zwan et al. [55] tries a different approach. Suppose an image consists of n distinct layers. To determine layer importance they repeatedly remove each layer one at a time and determines the reconstruction error without that layer. This is then repeated n times, once for each layer. Those layers that when they are removed have the highest reconstruction error are the most important ones. Thus one could select the ℓ layers that without those layers result in the highest reconstruction error. While this seemingly gives a good reconstruction result there are a few downsides to this approach. One is that reconstructing an image n times is an expensive operation, especially when n is large. Another one is that layer selection now has become very opaque. While one can say that he wants the ℓ most relevant layers, there is no intuition which layers these will be. This is not necessarily a bad thing but it disables the user of having full control over layer selection. Finally, this is an inherently greedy approach which might have unexpected results. It might be an image where deleting layers n trough m *individually* has a low impact on an image, but removing *all* layers n through m has a very large impact. This situation is poorly detected by this method but a real possibility.

Contribution: We have found another method to select relevant layers. We select layers that

are *local maxima* in the histogram. As these regions "stand out" with respect to similar image intensities so these must be important. We consider the amount by which they stand out with respect to surrounding layers a measure for importance; if it has much more pixels than surrounding layers it must be important. The layers which are selected are the n most important local maxima.

Using this technique, the number of selected layers can be reduced from 256 intensities to about 15 with acceptable results. However, finding these local maxima in the histogram is no trivial task. The standard technique of finding the local maxima is by the first-order derivative but histograms are rarely differentiable anywhere as they are almost never C_0-continuous. One could try to compute the discrete derivative by central-order differencing but this process is very numerically unstable. This could somewhat be averted by smoothing the histogram but this shifts the peaks and might therefore yield unreliable results. Another method is to fit a very high order (i.e. 20) polynomial through the points of the histogram. These polynomials are often badly conditioned, numerically unstable and yield improper results.

Currently we select detect peaks by a look-ahead and look-back. If a layer has the highest histogram value within a certain distance from it, we consider it a local maximum.

Traditionally layers are removed by mapping them towards the nearest layer below the layer that is about to be removed. This can, however, introduce a very large reconstruction error when very few layers are selected as the image is darkened a lot and a strong banding effect.. This is illustrated in Figure 5.3.

It makes more sense to map the removed layers towards the closest selected layers, regardless of direction. The result is that all pixels with a "peripheral" intensity are only changed slightly as their closest selected layer is closer than when it is only darkened. Moreover, the number of pixels with a changed intensity is relatively few as important layers are selected by maxima in the histogram, i.e. the layers with most pixels. With Meiburg's method, at least 75 layers are necessary to produce high-quality results, whereas with this method good results are obtained with as few as 15 layers.

5.3. Skeletonization

At this stage we have selected the relevant upper level sets which have been filtered for optimal skeletonization. Now every element of this set can be skeletonized. This means that every upper level set is modified to a skeleton map – an image in which pixels that represent skeleton point sites are foreground and non-sites are background – and a distance transform map – an image in which each foreground pixel in the upper-level set contains a floating-point value with the distance to the boundary, and background pixels are zero.

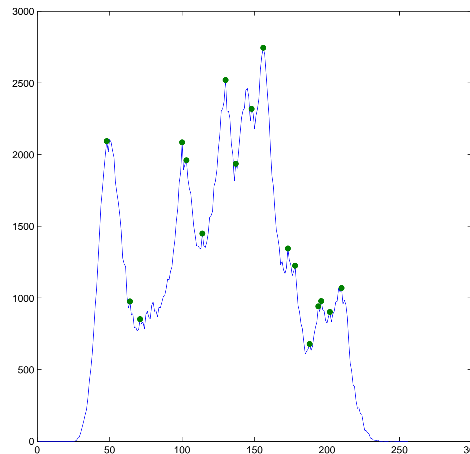
Skeletonization can be influenced by a few parameters, the most important being the *salience* parameter. As one can recall from chapter 2, this parameter can simplify skeletons such that noise is removed but important features remain. Skeleton simplification is a very important component as otherwise skeleton points would remain that encode noise which would take up the majority of the space. Other parameters are with respect to filtering "small" branches. This can either refer to to the *length* of the branch – with the rationale that short branches have high cost of bytes per skeleton points and are probably hardly visible – or the *area* – if a branch encodes a small area it will not have a high visual impact and can therefore safely be removed.

After these removal processes we have a set of salient and visible skeleton points and the subsequent distance transform map. As we can recall from chapter 2, we know that the skeleton branches are one-pixel thick, 8-connected "paths". It makes sense to represent these paths in a tree because this makes it possible to have a deterministic beginning and end for each path. Regular skeleton paths do not always have such beginning or end because these paths can be circular. If we were to represent them as a tree, however, one can pick any point on that path and promote it to root of the tree. From there, a DFS search along the path while marking which nodes are visited creates a skeleton tree from a path regardless of the topology, even one with cycles.

This process is repeated until all skeleton points in an upper level set are collected and stored in a tree. Since an image is a collection of upper-level sets, we now have a structure for representing an entire image.

In our pipeline we employ the GPU skeletonization method using CUDA – as described in chapter 2 – which computes the skeleton of a single layer within a few milliseconds and the skeleton of an entire image within 1 second.

(a) Original Lena image as visible in Figure 6.4.



(b) Histogram of Figure 5.3a. Notice the local maxima are the most relevant layers (marked in green).

(c) Skeletonization of Figure 5.3a using the 17 most relevant layers.

(d) Skeletonization of Figure 5.3a with peripheral intensities projected onto the most relevant layers

Figure 5.3: The effects of using skeleton image coding as a preprocessor for a matrix method. The images on the bottom are very similar, but much smaller than the original image.

5.4. Signal modification

There are two ways to store a signal in a smaller manner. One can either remove redundant information or one can represent non-redundant information smarter. Here, we attempt to do both.

5.4.1. Bundling

Suppose that there are several skeleton points sufficiently close together across different upper-level sets. If one is willing to accept a small reduction in image quality or otherwise small change in the image, one can much more efficiently encode an image by *bundling* several skeleton points together in one skeleton point. This forces fewer unique points at the cost that the boundary of the shapes move compared to the original image. Image results can be altered on how bundling is applied. This is akin to graph bundling[13, 47], in which graphs that are often too complex, big or detailed to display in a useful way from which one can derive sensible information are altered to emphasize important information and simplified for usefulness, enhanced understanding, and aesthetics. We can apply this same strategy to skeleton points between thresholds: skeleton points at some higher threshold that are close to skeleton points at a lower threshold can be bundled into one skeleton point at a shared

location.

The goal for doing this is twofold. On the one hand it can improve image compression, as fewer distinct skeleton points ought to be encoded. The other, more interesting goal is that it opens new ways for image manipulation. Traditional image manipulation techniques operate either on pixel level (e.g. setting a pixel value, inserting a sprite), fixed local neighborhood (e.g. Gaussian blurring), or a global level (e.g. contrast stretching). On the other hand there are classical morphological filters such as the opening or closing of shapes. These filters are however somewhat limited as these feature fixed structuring elements, which usually requires prior knowledge of your image. Skeleton-based bundling, however, is an almost fully automatic new image filter which can introduce subtle but interesting effects.

To do this, one must find points that are sufficiently close by some measure and combine these points into one new point shared by all relevant skeleton path. There are different measures that can be taken into consideration when determining "closeness" when considering skeleton points. One obvious measure is the position or distance. This is easily determined by some measure such as the Euclidean distance. However, relying on the distance alone might give a skewed perception of similarity. If there is a large discrepancy between the distance transform values for each skeleton point then these might be too dissimilar to bundle. There are therefore three options when considering points:

- Leave the DT values out of consideration, only looking at position.
- Attenuate the distance by DT values according to an inverse proportional function
- Modify the DT values of each skeleton such that they become closer together when bundled, sharing the "error" between layers rather than accumulating this on one layer.

There are also several strategies to be considered where to place the bundled points. One can either place the points at the location of the "darker" skeleton point, moving highlights closer to the boundary of the shape. The opposite would be to place the bundled points at the locations of the "lighter" points, placing highlights more centered with respect to the shape. There is also a "middle ground" approach by advecting skeleton points according the DT field. This is a meet-in-the-middle approach, placing the bundled skeleton point on neither of the original points yet sharing the "error" introduced, thus staying most true to the original image.

If the visual error is to be kept at a minimum, the bundling radius to to be kept rather small to prevent large movements of the shapes which could distort an image and not focus the entire reconstruction error on one specific layer, but spreading these over all layers thus "middling-out" the total reconstruction error. However, having a large reconstruction error can also be beneficial.

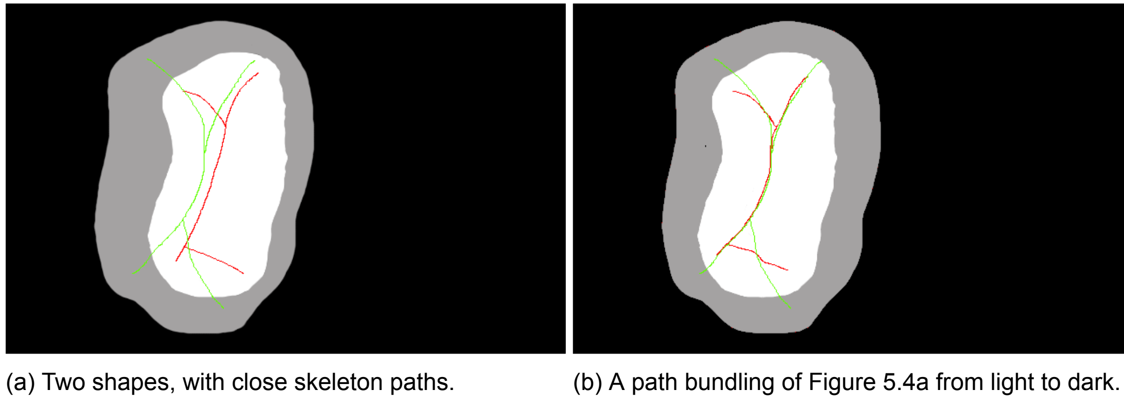
An important assumption in our image encoding scheme is that upper level sets are nested, i.e. $\forall \mathbf{x} \in T_{i+1} \rightarrow \mathbf{x} \in T_i$. It is important to maintain this principle. This is easiest maintained when only skeleton points at higher thresholds are moved towards skeleton points at lower thresholds. If we apply this operation layer by layer it becomes a rather straightforward algorithm. Per skeleton point in the higher layer, find the closest skeleton point in the lower layer and move it towards it.

Finding the closest skeleton point per skeleton point is no trivial task. A naive linear search over the skeleton images makes this operation $\mathcal{O}(n^2)$ in image size. An easier way to find these skeleton points is by computing the feature transform of the skeleton at the lower layer. Recall that the FT is a map to the closest point on the boundary from a foreground point. If we consider the skeleton as the boundary and non-skeleton points as the foreground this will give for each non-skeleton point the closest skeleton point. Ergo, the closest skeleton point in T_i for some skeleton point $\mathbf{x}_{i+1}$ in T_{i+1} is given by $\mathbf{x}_i = \mathrm{FT}_{T_i}(\mathbf{x}_{i+1})$. We can find the bundled skeleton point $\mathbf{x}'_{i+1}$ by moving $\mathbf{x}_{i+1}$ towards $\mathbf{x}_i$. We do this by introducing an attraction parameter $\alpha \in [0, 1]$. This defines $\mathbf{x}'_{i+1}$ as a linear combination of $\mathbf{x}_{i+1}$ and $\mathbf{x}_i$ by

$$\mathbf{x}'_{i+1} = (1-\alpha)\,\mathbf{x}_{i+1} + \alpha\,\mathbf{x}_i,$$

i.e. as α nears 1, it reduces to $\mathbf{x}'_{i+1} \approx \mathbf{x}_i$, the closest point in the skeleton in the lower threshold set. Likewise, if α nears 0 it will stay close to its original location, so $\mathbf{x}'_{i+1} \approx \mathbf{x}_{i+1}$. In order to limit the reconstruction error and not move the boundaries of shapes too much, the total shift should be limited by $\left|\mathbf{x}'_{i+1} - \mathbf{x}_{i+1}\right| \leq \epsilon$ where ϵ is some value representing the maximum number of pixels a point can move.

An example of what might happen in Figure 5.4. As one can see in Figure 5.4a, there are two shapes with their (supposed) skeletons; the skeleton of the darker shape in green and the skeleton of



(a) Two shapes, with close skeleton paths.

(b) A path bundling of Figure 5.4a from light to dark.

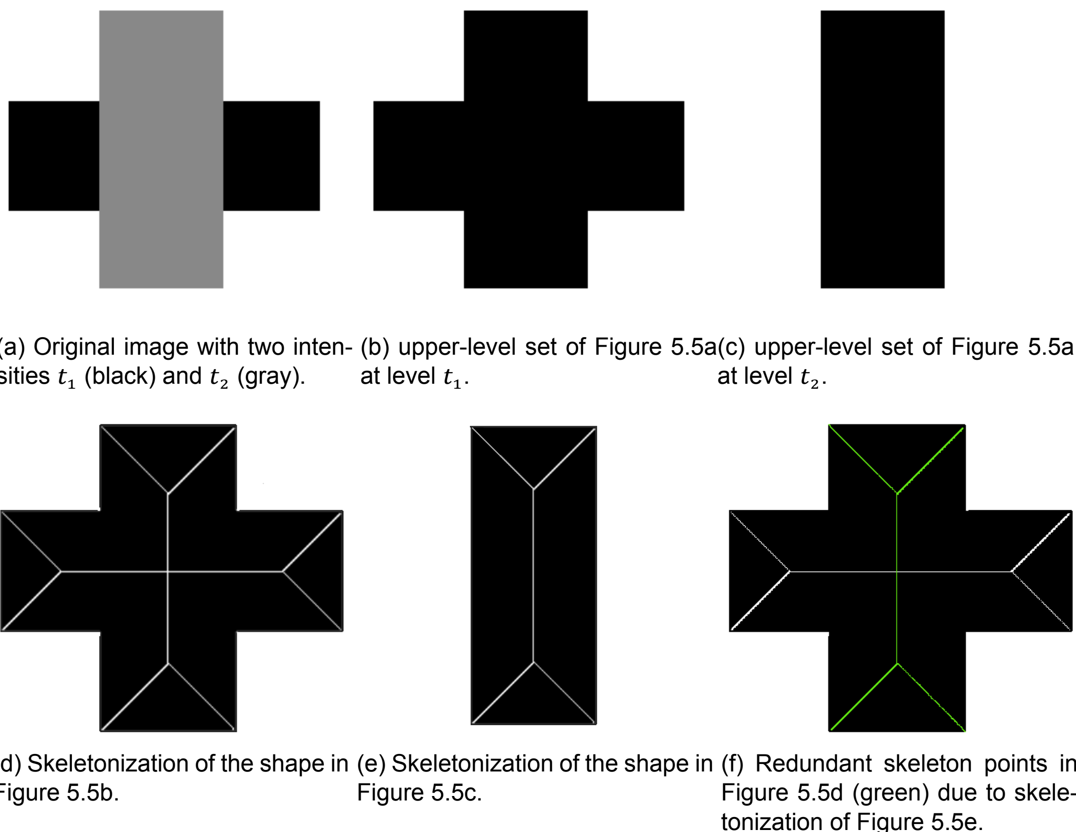
Figure 5.4: Bundling leading to interesting image effects.

the lighter shape in red. These paths can be bundled as most points are sufficiently close together. Note that this notion of closeness is relative. In Figure 5.4b the lighter shape has moved toward the center of the darker shape. Note that the shape outlines remain the same although the skeletons moved. A possible result of this filter is thus a local increase in contrast; the distance between boundaries between thresholds is increased, which can be perceived as more dark pixels surrounding a brighter area, which can increase the perception of contrast or highlight enhancement. Moreover, we see in Figure 5.4b that the number of distinct skeleton point locations has reduced drastically as a result of this operation. This can aid compression of skeleton points significantly – i.e. storing these two layers requires about 10% less space – as skeleton points are expressed by distinct locations. The results of this are presented in section 6.7.

5.4.2. Overlap pruning

Overlap pruning is an attempt to remove redundant skeleton points that do not contribute to the final image because the disc corresponding to that skeleton point is – or at least partially – occluded by discs at higher levels. Since the binary images during skeletonization consist of upper-level sets it will regularly occur that a region does not change does not occur when transitioning from a threshold level t_1 to a higher threshold level t_2 as this region has a higher intensity of t_3 (where $t_1 < t_2 < t_3$). Therefore, when the full skeletons of these layers are stored this results in encoding redundant skeleton points. This redundancy is visualized in Figure 5.5. We see a shape in Figure 5.5a which consists of two distinct grayscale intensities, which we shall refer to as t_1 (black) and t_2 (gray). Their respective upper-level sets are visible in Figures 5.5b and 5.5c. If we were to skeletonize these shapes we can see there is an overlap in discs between the skeletons of t_1 and t_2. In Figure 5.5f we have marked the redundant skeleton points of Figure 5.5d green as a result of the skeleton points in Figure 5.5e, as their corresponding discs are are fully occluded by discs at the higher layer. Not encoding these green points results in about 33% fewer skeleton points to be stored for the full image.

This is an effective way to reduce the number of skeleton points, or to “prune” the skeleton trees. However, it is important to realize when it is possible to prune in a lossless manner. One of the observations is that the digital shape embedded in the upper-level set at threshold t_2 is always fully embedded in the upper-level set at t_1. This implies that $\forall p \in \Omega_{t_1}, DT_{t_2}(p) \leq DT_{t_1}(p)$. That is, for all pixels in the upper-level at the higher threshold is always smaller or equal to that at the lower threshold. The proof for this statement is trivial. Clearly, the upper-level set Ω_{t_2} is a subset of Ω_{t_1} as only pixels are removed when a grayscale image is thresholded at higher level. Therefore, the digital binary shapes in Ω_{t_2} are smaller then, or equal to, those in Ω_{t_1}. Since the binary shapes are smaller, the distance transform is also smaller except in the case when the digital shape stays the same – pixel for pixel, that is – then the Distance Transform also stays the same. However, when they are equal then the shape at the lower threshold will not contribute to the final image. Figure 5.6 tries to illustrate what it means if the digital shape does not change. What we see is one $\{x, y\}$ point – which happens to be a skeleton point – with all corresponding radii at different threshold levels. We can assume that t_1 is the lowest intensity level and t_5 is the highest intensity level. Clearly, at threshold t_1 contributes to the final image as there are some non-occluded pixels. Also at level t_5 *all* pixels contribute to the final image because there is no



(a) Original image with two intensities t_1 (black) and t_2 (gray).
(b) upper-level set of Figure 5.5a at level t_1.
(c) upper-level set of Figure 5.5a at level t_2.
(d) Skeletonization of the shape in Figure 5.5b.
(e) Skeletonization of the shape in Figure 5.5c.
(f) Redundant skeleton points in Figure 5.5d (green) due to skeletonization of Figure 5.5e.

Figure 5.5: Example of overlap pruning. The original shape is visible in Figure 5.5a. When both upper-level sets are skeletonized parts of the skeleton in Figure 5.5d are made redundant due to the skeleton points created in Figure 5.5e and can safely be discarded.

layer above it. However, at thresholds t_2 through t_4 the radii – therefore DT values – are equal. This means that only the radius at t_4 needs to be stored as those at t_2 and t_3 are entirely occluded and will be overwritten by that at t_4. This method ought to be lossless because no visible pixels at the end result will be modified as the restoration of the pruned discs has no influence whatsoever on the final image.

This can also easily be transformed in a lossy method for skeleton compression. Rather than deleting the lower skeleton points whose radius is equal than at some higher level, one could also delete all lower radius values where $D_L - D_H \leq \epsilon$ for some integer ϵ and D_L is the lower skeleton point and D_H is the higher skeleton point. This difference will always be ≥ 0 as $D_H \leq D_L$.

This method can for most images greatly reduce the number of encoded skeleton points, as it typically removes more than 50%, and up to 90%, of the total number of skeleton points in all layers. The full results are discussed in section 6.3.

5.4.3. Delta representation

Our original tree per layer is a tree of location tuples $\{x, y, DT(x, y)\}$ where x and y represent some location in the image relative to the origin and the DT value is the distance from that point to the boundary of the shape. Assuming the image size is less than $2^{16} = 65536$ pixels wide or high a naive approach might be to store each point as two 16-bit values and a 32-bit floating point value – or 8 bytes per skeleton point. This clearly is not going to provide a better compression scheme than traditional raster image formats as this is very wasteful. If we were to use this encoding scheme the final file size would be several orders of magnitude higher than the corresponding JPEG. One of the first observations that we can use is that a skeleton path is 8-connected. This means that the difference of location between each skeleton point is at most one pixel in either x or y direction, or both – i.e. $\Delta x, \Delta y \in \{-1, 0, 1\}$. For the radii this difference is somewhat subtler. However, Meiburg [26] has provided a topological proof that this

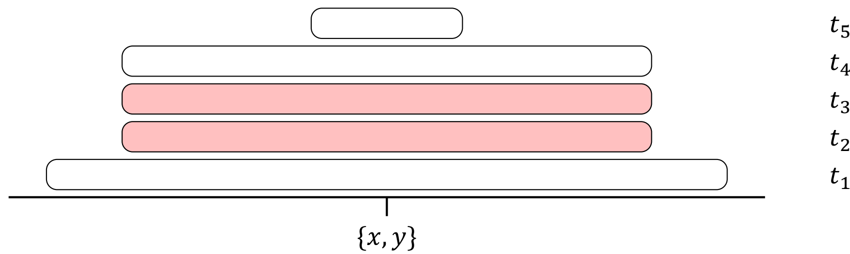


Figure 5.6: Overlap pruning in action. Layers t_2 and t_3 are not contributing to the final image as they are entirely occluded by t_4.

difference can never be more than $\sqrt{2}$. Also, since we are implementing a lossy compression scheme, we are not interested in sub-pixel accuracy with respect to recovering a digital shape. Therefore, with rounding issues taken into account, the range for $\Delta r = \{-2, -1, 0, 1, 2\}$. This means we can encode each skeleton point in at most one byte per skeleton point with respect to some starting point. This encoding scheme is thus very efficient as stores only skeleton points rather than a full skeleton image, and it stores each skeleton point very compactly at 1 byte per skeleton point.

5.5. Skeleton data encoding

We have tried to apply several encoding methods to this delta-representation of each skeleton tree. One of such methods is a direct encoding. If the skeleton tree is eight-connected, we have already established that $\Delta x, \Delta y \in \{-1, 0, 1\}$ and $\Delta r \in \{-2, -1, 0, 1, 2\}$. This means that we can represent each point in $2\log_2(3) + \log_2(5) \approx 5.89$ bits, optimally. In practice, however, we use a single byte which is divided as `0xxyyrrr`, meaning two bits each for Δx and Δy and three bits for Δr.

Another direct encoding approach also yields a slightly lower entropy. Consider all possible combinations between the sets that $\Delta x, \Delta y,$ and Δr could consist of. These are 45 distinct points, but one can eliminate 5 combinations because these involve $\Delta x = \Delta y = 0$ which is obviously not possible. If one were to topologically sort this set of points, the points could be encoded by the index in this set. Therefore, entire trees can be encoded by a series of integers of the range $0, 1, \ldots, 39$. This has an information content of $\log_2(40) \approx 5.32$ bits.

There are also different strategies for applying Huffman coding or Arithmetic coding to this dataset. One can either apply this directly to the latter index encoding or independently to each component of the delta encoding. There are different aspects to be considered which one could be called “better”. If it were to apply directly to the components of the delta encoding, it would be more flexible as it allows for encoding a disconnected skeleton. However, directly encoding the indexed approach yields a slightly lower entropy and therefore also a slightly smaller file size. Huffman coding was partly implemented using BCL [17] and Arithmetic coding was implemented using FastAC [33].

There is one final option to encode skeleton points *without* using a tree structure. At the time of encoding we have the following information available: the image size and the distinct skeleton points at all threshold levels. We can use that information to encode all skeleton points at a line-by-line basis through an image. We assign each skeleton point a vector the size of distinct gray levels (i.e. in an 8-bit image each point is assigned a vector of 256 elements). If a gray level is not participating at that point it is assigned the value zero, otherwise it is assigned the difference between that radius and that at the previous participating gray level. Explicitly storing this vector for each point may prove to be arduous and redundant so we can use a better encoding. Rather, we can store only each participating gray level. This can be encoded as a tuple of intensity difference and absolute radius difference[1] or as the number of non-participating intensities – i.e. zero components of the vector – preceding this intensity and the difference in radius. At the beginning of the stream we encode the full start coordinate, followed by the vector encoded by one of the two options described as above, followed by an end-of-vector (EOV) marker. All subsequent lines only need to encode the difference in x-value since we can deduce from the image size when we “overflowed” to the next line. The results of various encoding schemes are discussed in section 6.2.

[1]We do not have to worry about signedness as we know that $r_i > r_{i+n}$ for some non-zero n.

5.5.1. Transformations

Several transformations of the data were also considered before an attempt was made to encode it. Applying one or several transformations to a signal may be advantageous to the encoding because it may allow an entropy reduction or it may enable better compression results when a compression technique is applied. For example, as we have seen before, JPEG applies a DCT to pixel data in order to create long runs of zeros to allow for an efficient run-length encoding compression.

We have considered applying different transformations, specifically the (Fast) Fourier Transform[14][10], Discrete Cosine Transform[1], Walsh-Hadamard Transform[36] and the Move-To-Front transform[32]. We deemed the Fourier Transform unfit as efficiently encoding both real and complex components turned out to be nigh impossible. As the other transforms have real and symmetric output, these are more interesting.

There are three different ways to apply each transformation. One could apply the transformation to each point individually, to each block of points or to every component of each block of points. However, since we have a difference encoding, all values are already centered around 0 and are reasonably small the transformation will be reasonably close to the original value. Subsequent quantization of these values (by rounding directly or rounding after division with a quantization table, for example) will therefore result in large absolute error. This will result in too large error in the decoded points and too large errors in image reconstruction.

One transform that we have implemented is the Move-to-Front transform. Recall our earlier lexicographically sorted list of all possible points. Normal encoding returns the index of where the point occurs in the list. the Move-to-Front transform makes an extra step by moving that point to the *front* of the list after it has been returned. This means that when there is a repetition of the same point, a run of zeros will be emitted and when there is a sequence of multiple points, these will emit low integers rather than their original indexes. Note that this transformation does not alter the entropy of a signal but it does alter the distribution of the symbols of the signal, favoring low numbers over higher numbers. This makes techniques such as run-length encoding and Huffman coding more likely to compress a signal than without this transform. This is discussed in section 6.2.

5.5.2. Other techniques

Besides the techniques described above, a plethora of other techniques was also tried but decidedly was dropped in favor of developing other encodings because the results are lacking. One of these is signal compaction, or dictionary compression such as LZ77. We have used an “off-the-shelf” method before applying the external compression but this resulted in significantly larger files.

A predictive scheme was attempted by either predicting entire next points or predicting each component of the point individually. For predicting entire points we have considered a first-order prediction as well as PPM schemes. While for most[2] points we have found that the next point is the same as the previous, but this does not directly result in compression as this significantly increases the number of possible symbols (instead of 40 distinct symbols there can now be an *offset* of either +40 or -40, increasing the number of symbols to 80).

In the same vein as DCT/FFT transforms, wavelets also turned out to be a poor representation for skeleton points. As skeleton points can be better considered as a singularity on the 2D plane rather than a continuous signal spanning the entire 2D plane, more information is introduced in a wavelet representation than removed.

5.6. External methods

After all points have been encoded, there is still likely a lot of redundancy left in the signal. These are often introduced by having redundant layers – i.e. layers that are largely the same as other layers – and redundant inter-frame encoding. Therefore, it can be fruitful to apply an external compressor to the image format.

The question which rests then is to select *which* external compressor to apply. Meiburg[26] used the LZMA algorithm[30] to compress the data streams. This is a variant of the LZ77 algorithm which uses dictionaries to compress strings. It uses a variant on how dictionaries are used and the resulting bits are encoded using a range coder which employs a Markov Chain to predict bit patterns. Due to the nature of the data, it is uncertain whether this is the best option. This problem is solved rather easily

[2] In this case, most means higher probability than a uniform distribution

with a greedy approach: compress the same data stream with an ensemble of different algorithms and pick whichever obtains the lowest byte size. For this, we use the Squash compression library[28] which implements a plethora of different compression algorithms.

Since we optimize for smallest file size we have currently only implemented a selection of the algorithms which Squash provides, removing the algorithms optimizing for speed rather than compression ratio. Often these algorithms are slow, but since our file sizes are relatively modest this is a justified choice. Currently, we expose the following algorithms: LZMA(2)[30], ZLib[11], LZHAM, Brotli[2], ZPAQ[25], BSC[18], CSC[15], Zstandard[9], and BZip2[35]. We compare these compressors in section 6.4.

5.7. Reconstruction

Reconstruction of a SIR file is a rather simple operation. The first step is decoding a file so it once again becomes a representation of layers and skeleton points. Now restoring an image is simply a matter drawing each circle with the corresponding radius centered at each skeleton point. If one has encoded the image in a lossless manner – i.e. no layers removed, no DT thresholding etc. – this would exactly restore the original image, albeit for a few individual pixels which may stay unrestored due to digitization of circles. However, if one *has* removed many different layers it could result in many “banding” artifacts. In that case there is a steep jump from pixels at a lower intensity to the next intensity, rather than a smooth transition as happens in nature. We can circumvent this by interpolating only those pixels that were drawn in the previous layer but are not replaced when the next layer is drawn. van der Zwan et al. [55] provide this interpolation scheme by the following formula for those pixels as

$$v_{\text{new}} = \frac{1}{2}\left[\min\left(\frac{DT_{T_i}}{DT_{T_{i+1}}}, 1\right) v_{\text{prev}} + \max\left(1 - \frac{DT_{T_{i+1}}}{DT_{T_i}}, 0\right) v\right]$$

where v is the current intensity, v_{prev} was the previous intensity DT_{T_i} was the distance transform value at that point at the previous threshold, and $DT_{T_{i+1}}$ is the distance transform value at that point at the current threshold. Note that since we are interested in the distances *between* boundaries of the threshold sets, DT_{T_i} ought to be of the *foreground* of T_i and $DT_{T_{i+1}}$ ought to be of the *background* of T_{i+1}. This achieves a smooth interpolation and therefore lower psycho-visual error, as visible in Figure 5.7.

(a) Skeletonization of Lena using 12 layers, *without* interpolation. (MS-SSIM 0.8949)

(b) Skeletonization of Lena using 12 layers, *with* interpolation. (MS-SSIM 0.9061)

Figure 5.7: Same skeletonization with and without interpolation, clearly Figure 5.7b has a lower psycho-visual error.

6

Results & Discussion

6.1. Image results

To correctly identify the strengths and weaknesses of our skeleton image representation it is important to apply our pipeline to many different images and determine statistics as file size and image quality.

subsection 6.1.1 will test different parameters of our pipeline. First we provide a corpus of uncompressed images for reference. Next we will test the effects of different encoding schemes of the same image data in section 6.2. Subsequently we will determine the effect of overlap pruning, bundling, and a combination thereof on the file size and image quality in section 6.3 and section 6.7. We identify which “types” of images work best using our dense skeleton image representation in subsection 6.5.1. Also, we compare how images are holding up against extreme simplification compared to JPEG in section 6.5. Moreover, we can use our method as a *preprocessor* to JPEG to combine the “best of both worlds”. Finally, we extend the method to color images in section 6.8.

6.1.1. Corpus

Our corpus was hand-picked with care to carefully represent many kinds of digital images and provide an objective basis. Parts of the images are obtained from the well-known USC-SIPI database [39], whereas some other images are drawn directly from a digital camera. Some other images are computer generated, and some other images again are pieces of art rather than real scenes. We have gathered our results from around 50 distinct images in total.

This provides a varied base of real images, generated images, “soft” images, high- and low-contrast images, and high- and low-frequency images. The images are here below, note that the file sizes are those of the *uncompressed* images.

Figure 6.1: Cameraman original (512x512, 257kB)

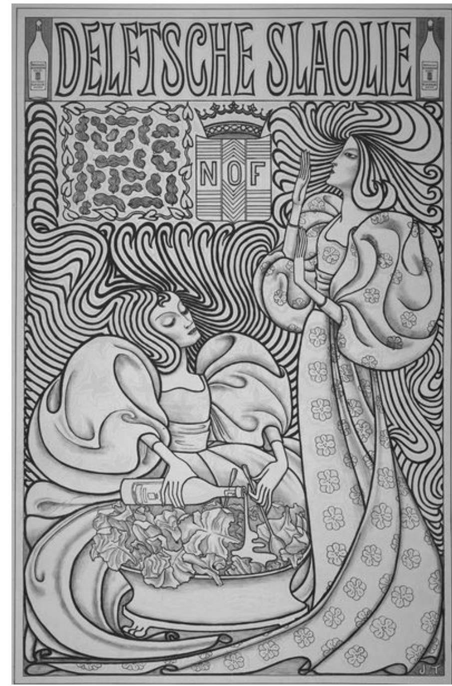


Figure 6.2: Commercial for Delft salad oil from 1894 (401x611, 240kB)

Figure 6.3: Elaine (512x512, 257kB)

Figure 6.4: Forest (1024x768, 769kB)

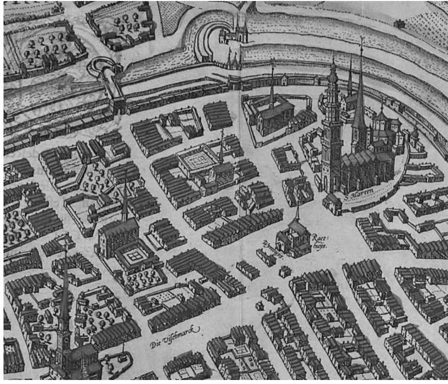


Figure 6.5: Map of Groningen of 1575 (701x601, 417kB)

Figure 6.6: House (512x512, 257kB)

Figure 6.7: Lena (128x128, 17kB)

Figure 6.8: Lena (256x256, 65kB)

Figure 6.9: Lena (512x512, 257kB)

Figure 6.10: Smiling people (641x965, 605kB)

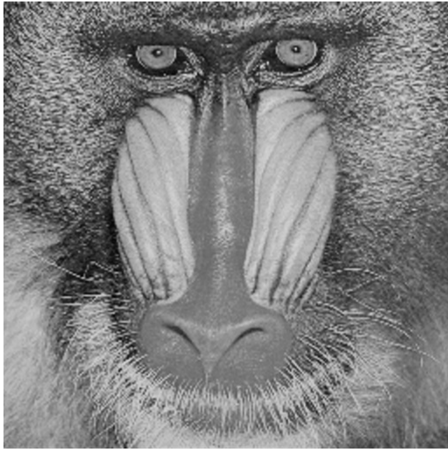


Figure 6.11: Mandril (512x512, 257kB)

Figure 6.12: Iconic picture of Marilyn Monroe (874x1079, 747kB)

Figure 6.13: Peppers (512x512, 257kB)

Figure 6.14: "Starry Night" painting by Van Gogh (750x565, 414kB)

Figure 6.15: Woman Blonde (512x512, 257kB)

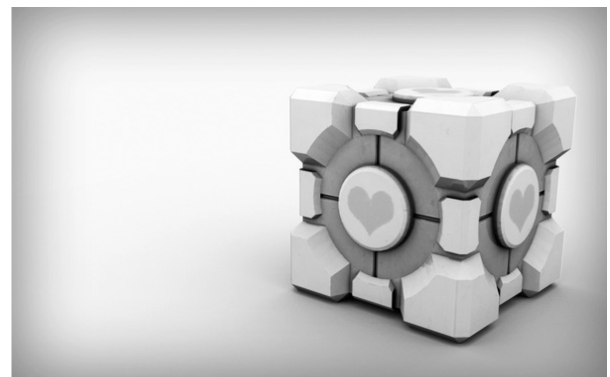


Figure 6.16: Companion Cube from the game *Portal* (600x375, 220kB)

To objectively compare all compression options, we have generated many different configurations per image, varying only the encoding schemes, whether or not an image can be pruned, and which external compression algorithm is used while keeping all the skeleton parameters constant.

6.2. Encoding schemes

We have implemented the encoding schemes presented in section 5.5 for comparison. In Figure 6.17 we compare different encoding schemes for the skeleton images, after external compression. On the vertical axis is the compression ratio as defined in section 4.4. The uncompressed size is that of the original PGM files. In all subsequent box plots in this work, the red line represents the median of the quantity. The blue box is the 25-75% quantile and the "whiskers" of the box the data points not considered outliers of the quantity. The data within the whiskers is within 2.7σ. The crosses above or below the boxes of the quantity are the outliers. As we can see, the Unitary and Exp-Goulomb

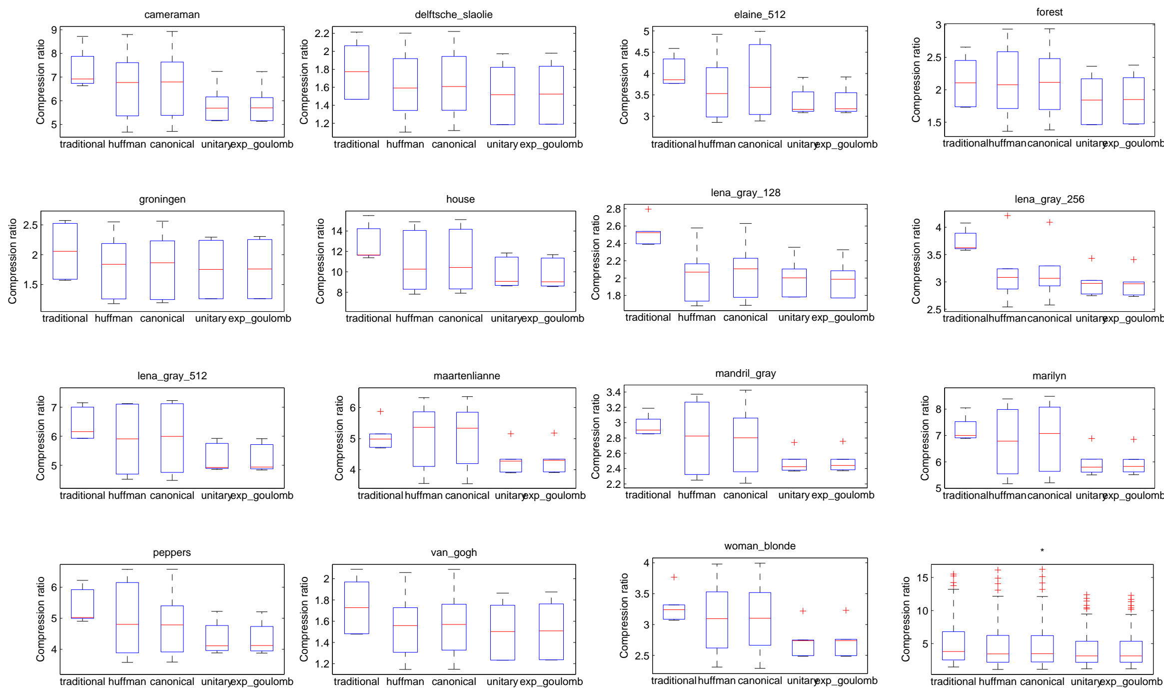


Figure 6.17: Comparison of encodings per image on the net file size.

encodings are most often the least efficient encoding for the average file size. In general, as expected, Canonical Huffman coding performs slightly better than traditional Huffman coding as the codewords are more regular, thus more easily externally compressed. However, in general they perform worse than the traditional delta encoding method. Although these results can be slightly improved by the Move-to-Front transform as discussed in subsection 5.5.1, the results are very marginal and still cannot compete with the delta encoding method. This is because the pattern matching of the external compressor is rendered ineffective because the signal encoding is already approaching the entropy, thus inflating the file size. This is even worse when Arithmetic Coding is applied, which was omitted from the graph because it does not yet work on pruned images which would lead to an unfair testing result.

This image is, however, turned around if the file sizes are considered *before* the external compressor, as visible in Figure 6.18. Here we compare the file sizes of *all* images in the corpus, with all other variables considered "free", using the specified encoding scheme. This file size is projected on the vertical axis. As we can see here, clearly Arithmetic Coding gives the best results, as expected. This is closely followed by the Huffman coding variants, as expected. However, it is also consistently the case that image file size before an external are *always* larger than after. If this compression is subsequently rendered ineffective, that means the entire image compression becomes ineffective. Therefore we can say for now that the "traditional" encoding is currently the best encoding.

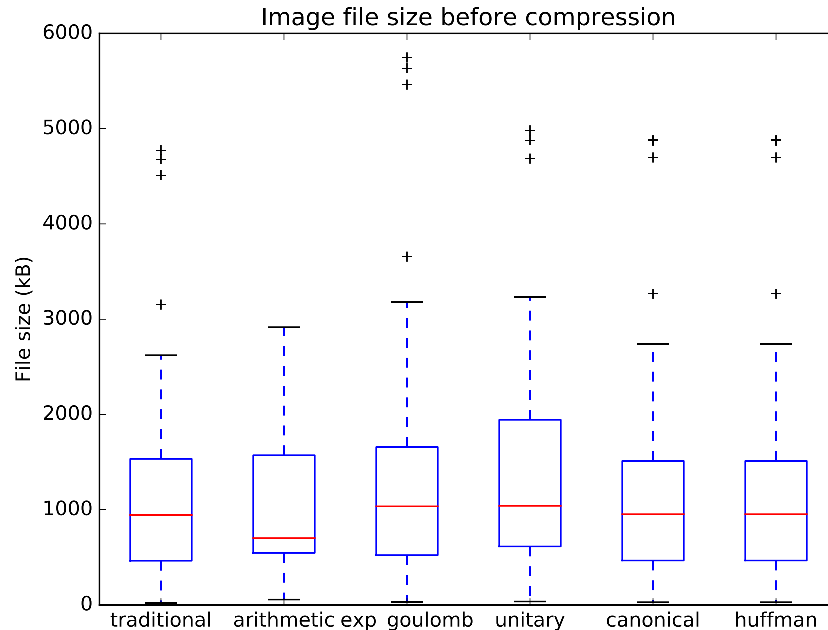


Figure 6.18: Comparison of encodings per image on the file size before external compression.

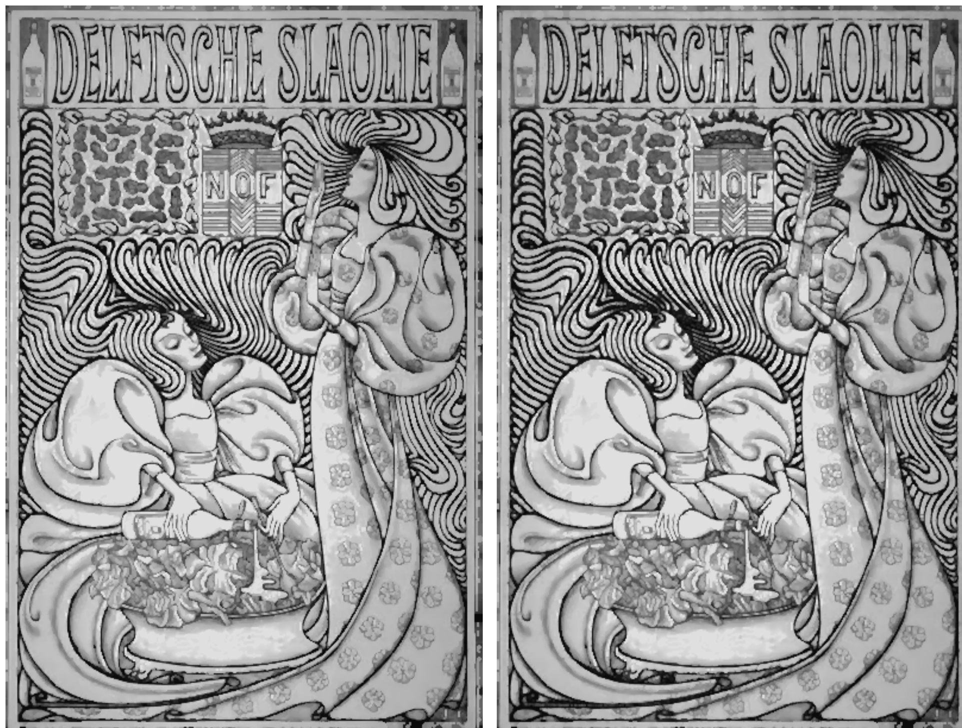
6.3. Overlap pruning effects

In subsection 5.4.2 we presented a technique to delete redundant skeleton points. The image quality can be visually inspected. Take for example the "Delft" image as here below in Figure 6.19. It can be verified that there are very few differences between both images, while Figure 6.19b is significantly smaller than Figure 6.19a.

The file sizes can be compared manually and while we noticed a trend that pruned images are smaller in file size, we decided to subject them to statistical tests. As we cannot assume that the file sizes are drawn from a normal distribution we have used non-parametric tests as the Mann-Whitney U-Test and the Kruskal-Wallis test. We have found that for nearly all images there is a significant reduction in file size ($p \ll 10^{-3}$) with an average reduction in file size of 100kB and up to 500kB.

We have for convenience included a boxplot in Figure 6.20. In this plot we compare between each image the file size after external compression, where each parameter is kept constant except for whether an image is pruned. A zero-reference comparison of *all* images in the same figure can found in Figure 6.21.

Using the same statistical tests we have found that while there is a significant reduction in file size there is in general *no* significant reduction in image quality when reconstructed, using any of the quality measures discussed in subsection 4.5.2 ($p > 0.05$) – i.e. it cannot be posteriori determined from which group an image comes. Combining these results we can say that, in general, overlap pruning is an effective way to reduce file size while maintaining image quality.



(a) Skeletonized, non-pruned image of Figure 6.2 (270kB)

(b) Skeletonized, pruned image of Figure 6.2 (220kB)

Figure 6.19: Visual comparison of different skeletonizations of Figure 6.2. This image uses 39 most significant layers.

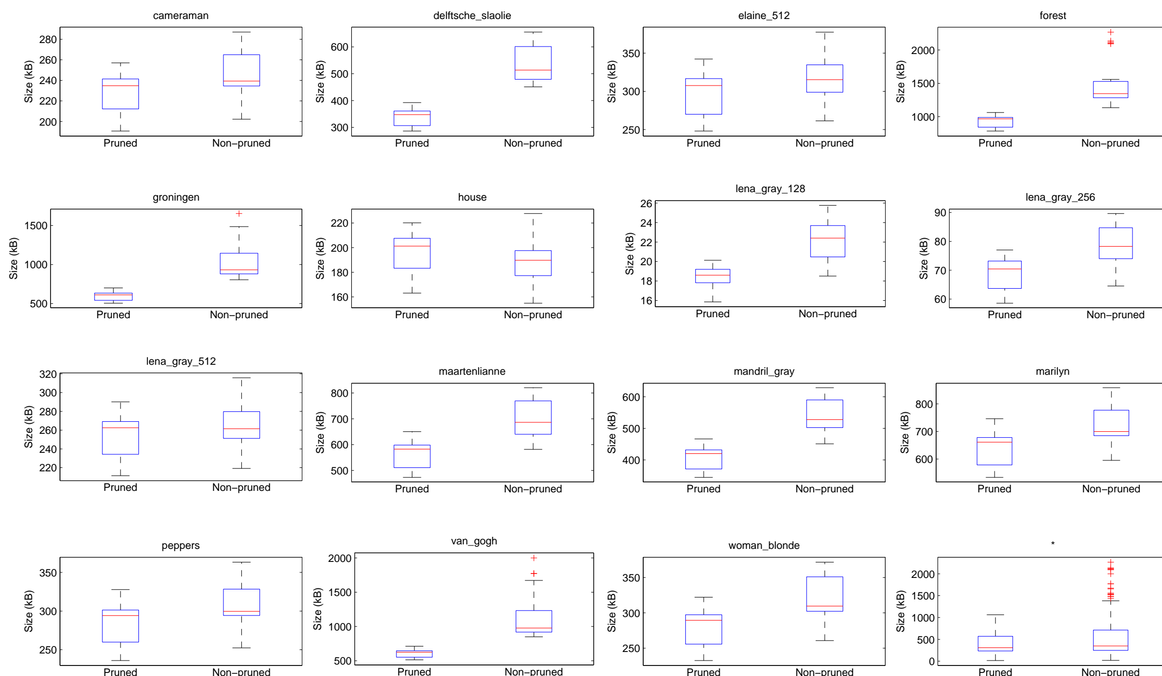


Figure 6.20: Comparison of file sizes after external compression per image. All parameters are kept constant except for the overlap pruning parameter.

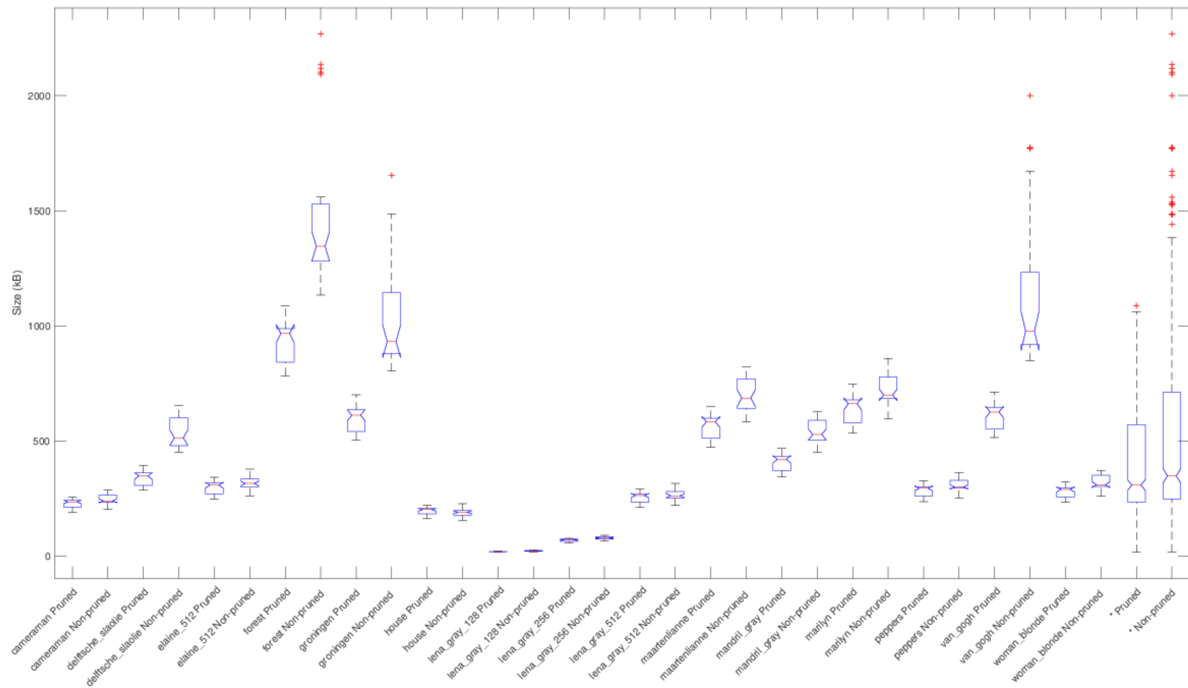


Figure 6.21: Comparison of the file size after external compression over all images. All parameters are kept constant except for the overlap pruning parameter.

6.4. External compression methods

External compression methods can help reach the entropy of the signal. That is, if the right one is chosen. To verify which compression method is the best – i.e. which one obtains the lowest final file size – we have compressed the same signals of our entire corpus multiple times to see which one obtains the lowest file size per image. This result is visible in Figure 6.22. We can clearly see that, in

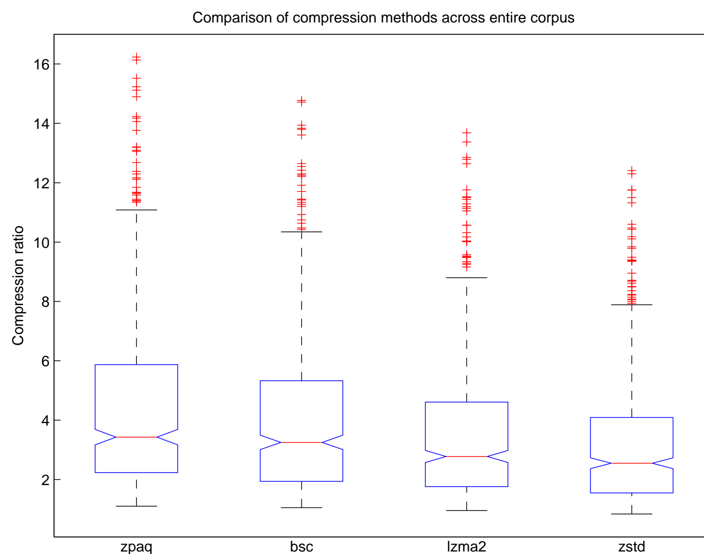


Figure 6.22: Comparison of external compression methods per image.

general, ZPAQ is the best candidate for SIR compression. On the one hand, this was expected because the algorithm was renowned for its excellent compression ratios. However, it was very possible that due to the structure of the data other algorithms outperforms it. On the one hand, this provides a good intuition which external compressor to use. On the other hand, this is a decision that can be greedily

solved without downsides and should therefore not be of much relevance.

6.5. Transcending JPEG

It turns out that even with our best encoding and external compressor, nearing JPEG in image size without sacrificing a lot in image quality is not easy. However, if one is willing to accept a loss, creating images smaller than JPEG with an acceptable quality is in the realm of possibility. However, using overlap pruning, it is possible to retain more layers than before which allows for the creation of passable images. Consider the image in Figure 6.23. This image is 31kB rather than the 64kB of a high-quality JPEG. However, one can clearly see that the image quality is lower than that of the original image in Figure 6.9, but the most salient shapes remain intact whereas small details are lost. This also happens in Figure 6.24. All the most salient shapes remain intact, including shadows but small details disappear and there are sometimes artifacts near boundaries of shapes due to short path removal. Even in Figure 6.25 dense skeleton image compression excels. “Barbara” is a notoriously hard image to compress due to the finely detailed patterns that occur in the image but even with our image compression method these remain mainly intact in Figure 6.25b. If one is willing to accept an ever bigger quality loss, as in Figure 6.25c, the final file size is even more impressive.

(a) JPEG compressed version of Figure 6.9 (64kB).

(b) Skeletonized image of Figure 6.9 using 15 layers (31kB, MS-SSIM 0.9232).

Figure 6.23: Visual comparison of JPEG versus skeletonizations of Figure 6.9.

The key to small images is to only select the relevant layers. This way, psycho-visual error remains as low as possible while the file size also remains within bounds. This is the most effective way to compress images. With a novel encoding and compression as described above, we have shown that it is possible to create images of acceptable quality that are significantly smaller than the corresponding JPEG image.

6.5.1. Types of images

In order to match JPEG images in file size consistently, it is important to identify the types of images that are especially suited for dense skeleton image coding. For example, JPEG is best suited for “natural” photographs and less suited for text images or other monochrome images with clearly defined boundaries. These images have, in general, few high-frequency intensity changes and are thus approximated well by JPEG encoding. In the same way are PNG images best suited for line art, icons and pictures text and less suited for photographs. In these types of images, pixels are often very similar to their neighbors which makes prediction of pixel values, which is the main compression feat of PNG, very successful.

In the same way, we can identify features that would create small, high-quality SIR files. Taken from the characteristics of the skeleton format, we can identify the following features:

(a) JPEG compressed version of Figure 6.13 (77kB).

(b) Skeletonized image of Figure 6.13 using 25 layers (44kB, MS-SSIM 0.8922).

Figure 6.24: Visual comparison of JPEG versus skeletonizations of Figure 6.13.

(a) JPEG compressed version of “Barbara” (78kB).

(b) Skeletonized image of “Barbara” using 15 layers (63kB, MS-SSIM 0.8757).

(c) Skeletonized image of “Barbara” using 15 layers with fewer details (39kB, 0.8708).

Figure 6.25: Visual comparison of JPEG versus skeletonizations of “Barbara”.

Few present intensities
One of the main factors contributing to significant skeleton image compression is the removal of grayscale intensities. By doing this, fewer skeletons sets need to be encoded resulting in small file with a low reconstruction error. However, this can partially averted by implementing intelligent binning as discussed in chapter 5.

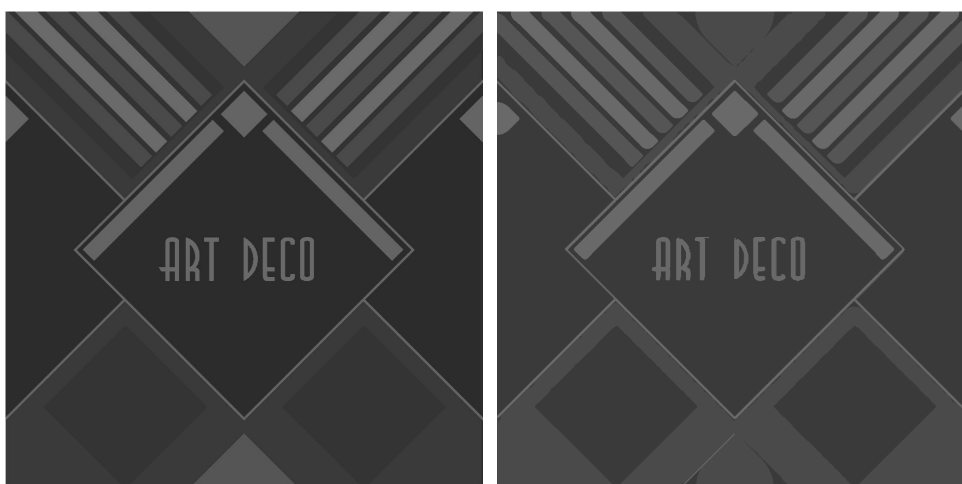
Salient shapes
If all shapes in the upper-level sets are salient there are few skeleton points necessary to describe the shape.

Large shapes
Many small skeletons are more spacious to encode than few large skeleton shapes. Since the size of the shape is somewhat related to the size of the skeleton it is more advantageous to have few relatively larger shapes than many relatively smaller shapes.

High overlap
If shapes change relatively little then many skeleton points can be pruned due to a high number of overlapping discs. This is advantageous to the file size since fewer skeleton points means a smaller file size.

Images that come close to these “virtues” will be images that compress very well. However, when these ideals are present in an image are not always obvious. Surely, when an image has few distinct intensities this will be visible but it is hard to judge when an image contains salient shapes due to the fact that the shapes that we observe does not always correspond to the shape that is skeletonized in some threshold set.

To make dense skeleton image coding successful large shapes are preferred. This is due to ratio of the number of skeleton points to describe a shape over the surface area of a shape grows smaller as the shape area increases. Moreover, large shapes have longer paths. With our delta encoding, anything but the start of the skeleton path requires merely one byte whereas the starting point of the path requires *six* bytes. One such extreme example is visible in Figure 6.26. Here, we have the original image in Figure 6.26a. It is a relatively large image and clearly has few distinct gray levels and large, salient shapes. The skeletonized version is in Figure 6.26. It is still a very high quality image, with few distortions and a contrast reduction, but is more than 11 times smaller than the original image using our dense skeleton encoding. Another such, perhaps less contrived, is visible in Figure 6.27. This



(a) Original image as JPEG (50kB, 800 × 800).

(b) Skeletonized representation of Figure 6.26a (4.4kB, MS-SSIM 0.9445).

Figure 6.26: An example of extreme compression results using dense skeletons with fair quality.

is a promotional still from the animated TV show *Bob's Burgers*. The original image is visible in Figure 6.27a. Notice that this image is quite large, contains not too many distinct grayscale intensities and contains large shapes. A high-quality skeletonization is visible in Figure 6.27b. This is a skeletonization performed using only grayscale binning as described in section 5.2, and overlap pruning. Using these techniques alone achieves a compression ratio of 2.6 compared to the JPEG variant and almost 7 to the uncompressed image.

The same happens in Figure 6.28. These are non-photographically rendered scenes which can be successfully compressed using dense skeletons.

In short, we can say based on these results that dense skeleton image coding is very successful compared to the de-facto standards when the images are *simple*. That is, there are large, salient shapes in each upper level set. These are often in non-photorealistic settings such as comics, animations, non-realistic 3D scenes or "minimal photography".

(a) Original image as JPEG (482kB, 1280 × 1014).

(b) High quality skeletonization of Figure 6.27a (184kB, MS-SSIM 0.9260).

Figure 6.27: Skeletonization of a comical image.

(a) Original image as JPEG (172kB, 1200 × 793).

(b) Skeletonized representation of Figure 6.28a (43kB, MS-SSIM 0.9128).

Figure 6.28: A skeletonization of a non-photographically rendered scene.

6.5.2. Extreme simplification

In many applications it is advantageous to simplify an image to its barest essentials. This is for example done in areas as object detection or shape-based feature detection. In some fields further analysis needs to be done on huge images – e.g. on a large giga- or even terapixel scale[23]. Notable cases where simplified images are used are fields as machine learning[45], multi-spectral imaging[4], and image segmentation[27]. Besides using image simplification as a tool to improve a result of another tool, it is also a goal in itself. JPEG tries to compress images with image simplification by removing high-frequency components of each macroblock in the frequency domain. A downside of this simplification is that it, while it achieves high compression, does not take into account the shapes that are visible on the image thus these images are are much less suited for the aforementioned tasks.

We have performed various simplifications as visible in Figure 6.29. As one can see the images are in the sense as described above very simple and the file sizes are to match this. The most principal shape are still visible in the images but there are also significant artifacts. For example, to the left of the house in Figure 6.29a there are bright discs that were not present in the original image. The dark spots in Figure 6.29b are a lot larger than in the original image, leading to almost removal of an entire pepper in the left of the image.

The Mandril image in Figure 6.29c is a good examples of a simplified image. Here the fine details of the fur are removed while the most salient shapes remain The file size is also significantly reduced while compression artifacts are not introduced and the image remains “continuous”.

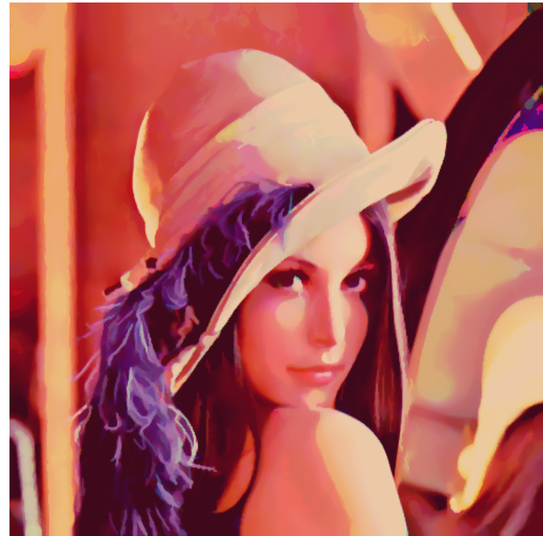
A notable simplification usage is that of color images. These often remain in very high quality while having the ability to simplified significantly. An example is in Figure 6.29d. Clearly, the color space and shapes are simplified whereas the most salient shapes and some of the details remain. This also happens without introducing heavy compression artifacts. Rather, the image looks more like it is overexposed than compressed.

While these images are severely compressed and simplified it should be noted that JPEG is still capable of generating higher-quality images at this file size. It will not be able to generate “simple” images like we present in Figure 6.29, it will generate better looking images at this size. If, on the other hand, the only goal is to remove small details or high-frequency components, our method will generate far better results than JPEG could by DCT thresholding.

(a) Simplification of Figure 6.6. 20 kB.

(b) Simplification of Figure 6.13. 14 kB.

(c) Simplification of Figure 6.11 image. 30 kB.



(d) Simplification of Figure 6.9 color image. 61 kB.

Figure 6.29: Extreme simplifications of various images.

6.6. JPEG preprocessor

Since we have found that reliably generating SIR files smaller than the corresponding JPEG file is not always simple, there might be another way. What we can do using the skeleton representation is simplify an image such that perceptual loss is minimized and important features remain. Moreover, this simplification can have an enhancing effect, because the image will be easier to encode because the intensities have been “regularized” and because of this, the strength of the artifacts will also be smaller because there will be fewer high frequency areas in the blocks so it is therefore possible to *increase* the compression strength of these encoders, for even better compression. This simplified output can be used as a *preprocessor* for existing image encoders to use their ability to optimize for the human visual system. Using a complex image processing pipeline as a preprocessor for a “simple” or traditional image processing algorithm is not new. This was also attempted by Tushabe et al.[46] who preprocessed an image by using max-tree filtering in order to obtain a higher JPEG compression.

Because the image has become very “regularized” – that is, the intra-macroblock variance is decreased due to layer thresholding and the removal of unimportant shapes – we have found that an image can be compressed further than a regular JPEG image. This is tested by encoding a skeletonized image as a low-quality – but passable – JPEG, which we then try to match from the original input image in either file size or quality as measured by the MS-SSIM. For example, in Figure 6.30d we have generated an image at 15% quality of Figure 6.30b. To obtain the same file size from the original image, the quality needs to be set at 7%.

An compression ratio of between 7 and 12 is reached with little quality loss of the original skeletonized image. Compared to a “vanilla” JPEG encoding of an image, it is possible to create images with a better quality at the same file size using our encoder as preprocessor. This is especially visible in Figure 6.31. In Figure 6.31c there are serious artifacts introduced, especially along the dent in the “front” pepper and along the border of the “top” pepper which are not as prominent in Figure 6.31d which is the same JPEG but preprocessed using our method. The difference in MS-SSIM score is mainly due to layer removal – thus resulting in a lower contrast which is heavily punished by the MS-SSIM score – and removal of small skeleton branches as visible in the stem of the “elongated” pepper, But overall it is fair to say that the quality of our preprocessed image is higher.

The strengths of our preprocessing encoding is especially visible in Figure 6.32 and Figure 6.33. In these images there is a very clear transition between intensity boundaries such as in the dress of Marilyn and the transition to the shadow from the floor. This introduces heavy ringing and blocking artifacts near the edges and highly detailed areas, as visible in Figure 6.32c. The same happens As there are fewer distinct intensities and details in Figure 6.32d, the quality parameter can be set higher to obtain an image at the same file size and there are therefor far fewer artifacts. However, highly detailed areas as on the camera in the background are still lost due to the skeletonization process. That is one trade-off that has to be made when using dense skeletons for preprocessing: when the quality of the skeletonization is low, the JPEG compressed result will almost certainly be worse than standard low-quality JPEG encoding while obtaining the same file size. However, when the skeletonization is of sufficiently high quality, the encoded JPEG is of higher quality at the same file size than a standard JPEG.

Concluding, we can generate JPEGs of dense skeletons at a quality that is at least 5 percentage point higher than directly encoding a JPEG image at a same file size, when computed using ImageMagick’s quality tuning parameter. When generating JPEGs of dense skeletons of same *apparant* quality as “vanilla” JPEGs, the compression ratio of the former is usually 1 to 2.5 points higher.

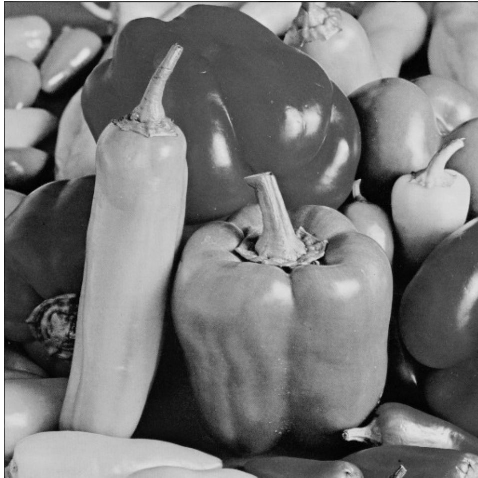
(a) Original forest image as visible in Figure 6.4 (395kB).

(b) Skeletonized representation of Figure 6.30a (729kB, MS-SSIM 0.695).

(c) JPEG at 7% quality of Figure 6.30a (33kB, MS-SSIM 0.86).

(d) JPEG at 15% quality of Figure 6.30b (34kB, MS-SSIM 0.68). The MS-SSIM compared to Figure 6.30b is 0.85.

Figure 6.30: The effects of using skeleton image coding as a preprocessor for a matrix method. The images on the bottom are very similar, but much smaller than the original image.



(a) Original peppers image as visible in Figure 6.13 (75kB).

(b) Skeletonized representation of Figure 6.31a (237kB, MS-SSIM 0.9636).

(c) JPEG at 20% quality of Figure 6.31a (11kB, MS-SSIM 0.9718).

(d) JPEG at 25% quality of Figure 6.31b (11kB, MS-SSIM 0.9528). The MS-SSIM compared to Figure 6.31b is 0.9862.

Figure 6.31: While the MS-SSIM of Figure 6.31c is higher than that of Figure 6.31d, the JPEG artifacts in the former are much more pronounced and one could say that the quality is lower.

(a) Original marilyn image as visible in Figure 6.12 (75kB).

(b) Skeletonized representation of Figure 6.32a (534kB, MS-SSIM 0.9477).

(c) JPEG at 13% quality of Figure 6.32a (25kB, MS-SSIM 0.9634).

(d) JPEG at 20% quality of Figure 6.32b (25kB, MS-SSIM 0.9335). The MS-SSIM compared to Figure 6.32b is 0.9841.

Figure 6.32: Comparison of using skeletonization as preprocessor on the “Marilyn” image (Figure 6.12)

(a) Original Lena (512) image as visible in Figure 6.9 (64kB).

(b) Skeletonized representation of Figure 6.33a stored as JPEG (20kB, MS-SSIM 0.9461).

(c) JPEG at 20% quality of Figure 6.33a (6.5kB, MS-SSIM 0.9461).

(d) Figure 6.33b experiencing the same quality drop compared to Figure 6.33c (4.5kB, MS-SSIM 0.8930).

(e) Figure 6.33b encoded as a JPEG to have the same file size as Figure 6.33c (6.6kB, MS-SSIM 0.9194).

Figure 6.33: Comparison of using skeletonization as preprocessor on the “Lena” image (Figure 6.9)

6.7. Skeleton path bundling

The technique of inter-layer skeleton path bundling was introduced in subsection 5.4.1. We have implemented this light-to-dark bundling scheme in our application and have collected our results here. The first effects of this bundling is presented in Figure 6.34. In Figure 6.34a is a skeletonized image of Figure 6.9 that has no bundling applied. In Figures 6.34b, 6.34c and 6.34d are bundlings of the same skeletonization using the attraction factor $\alpha = 0.1$, $\alpha = 0.9$, and $\alpha = 1$, respectively. All of these bundlings have the maximal point shift limited by $\epsilon = 4$ pixels.

As we can see there is no large difference between Figure 6.34a and Figure 6.34b. We can observe a small artifact in the upper right corner but overall the image looks quite similar to the original skeletonization. As α nears 1, the interesting image effects start to occur. Dark areas look darker and bright areas look brighter. Therefore this could be considered a local contrast enhancement. When $\alpha = 1$ the artifacts that bundling introduces can be considered too extreme, while when $\alpha = 0.9$ the artifacts are just barely prominent while there is an impression of local contrast enhancement.

(a) Unbundled Lena (512) image as visible in Figure 6.9 (95kB).

(b) Bundling of Figure 6.34a, $\alpha = 0.1, \epsilon = 4$ (95kB).

(c) Bundling of Figure 6.34a, $\alpha = 0.9, \epsilon = 4$ (82kB).

(d) Bundling of Figure 6.34a, $\alpha = 1, \epsilon = 4$ (67kB).

Figure 6.34: Comparison of using bundling on the “Lena” image (Figure 6.9)

The same effects can be observed in Figure 6.35. Here the same values for α are used in the same order, but the bundling effect is made stronger by setting the maximum shift to $\epsilon = 15$. Again, when $\alpha = 0.1$ the effects are barely present and the file size is even increased. This is most likely due to the introduction of irregular skeleton paths. Rather than having “straight” or continuous paths, the paths are now jagged due to rounding of the final location of the bundled skeleton points. Again, when α nears 1 a local contrast enhancement can be observed. The concavity on the front pepper looks deeper due to enlarged shadows and the bottom of the dark pepper on top looks darkened giving a

seemingly enhanced depth perception.

(a) Unbundled Peppers image as visible in Figure 6.13 (44kB).

(b) Bundling of Figure 6.35a, $\alpha = 0.1, \epsilon = 15$ (46kB).

(c) Bundling of Figure 6.35a, $\alpha = 0.9, \epsilon = 15$ (38kB).

(d) Bundling of Figure 6.35a, $\alpha = 1, \epsilon = 15$ (33kB).

Figure 6.35: Comparison of using bundling on the "Peppers" image (Figure 6.13)

.

From these experiments we can conclude that this form of bundling has an interesting effects on images. The enhanced shadow and local contrast enhancement gives interesting results. Moreover, it can be seen that when α is made large it has a very positive result on the image compression results. This is due the fact that fewer different distinct skeleton points need to be encoded, yielding a better compression result of about 30% as compared to an unbundled result.

6.8. Color images

The extension to encoding color images using dense skeletons from grayscale images is rather trivial. One can decompose a color image into a three-component image using some colorspace. Popular colorspaces for decomposition are RGB, YCbCr, and HSV. These are popular because of their connection to computer interpretation, separation of intensity and color, and connection to human color perception, respectively.

With dense skeletons, a color image is encoded applying our pipeline as visible in Figure 5.1 to each component of the colorspace. Reconstruction is then also simple. Each component can be recovered in the same way a grayscale image is recovered, converted appropriately into the RGB colorspace and inserted in the right color channel.

In Figure 6.36 we see the Lena color image skeletonized using the RGB colorspace (Figure 6.36b), HSV colorspace (Figure 6.36d), and the YCbCr colorspace (Figure 6.36c). The HSV colorspace seems

to have the truest colors compared to the original, whereas the RGB is slightly darker red. The YCbCr colorspace has the smallest file size but also contains severe color artifacts due to too few chroma layers. All in all, the quality of all these images is fairly high and in the same range as the JPEG file size. Also interesting is Figure 6.36f where paths have been bundled within each layer. Also here the contrast seems enhanced.

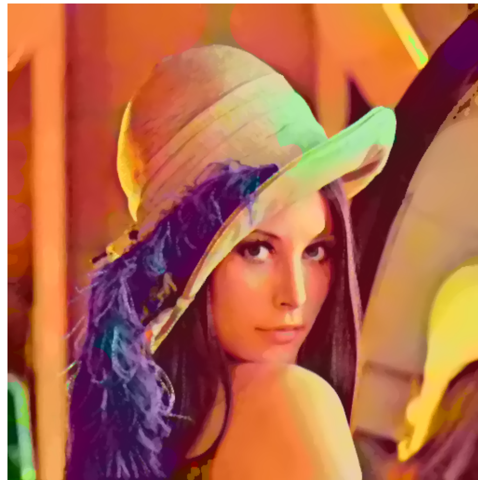
These results are largely the same in the peppers image set (Figure 6.37). Only here, YCbCr contains fewer artifacts and gives the impression of over-saturation whereas the HSV decomposition does show some artifacts.

In the Mandril image set (Figure 6.38) the YCbCr seems to give the truest result.

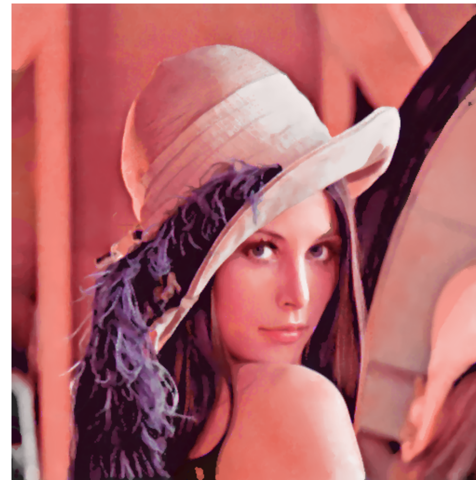
It should be noted that all channels of the color images are processed in their full form. That means that there is no chroma subsampling, resolution enhancements or different treatment of channels during skeletonization or filtering. This means there is still fairly much potential to be gained compared to JPEG which *does* perform these “tricks” in order to reduce file size, even though our method is already quite close to the JPEG result.

(a) Original color image of Lena. (512 x 512, 769kB)

(b) Skeletonization of Figure 6.36a using 30 layers per channel in the RGB color space. (180kB, SSIM 0.8966).

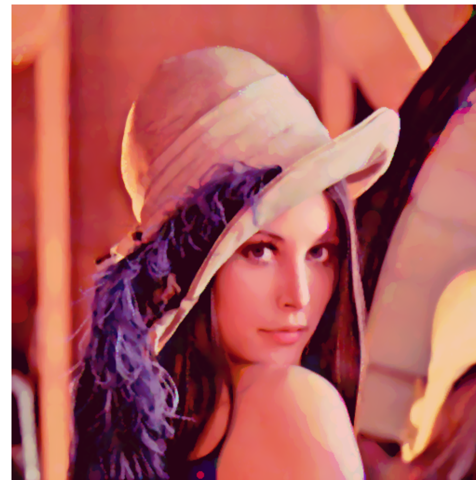


(c) Skeletonization of Figure 6.36a using 10 layers per channel in the YUV color space. (63 kB, SSIM 0.8270).



(d) Skeletonization of Figure 6.36a using 30 layers per channel in the HSV color space. (156 kB, SSIM 0.8779).

(e) JPEG compressed version of Figure 6.36a. (104kB, SSIM 0.9938).



(f) Skeletonization of Figure 6.36a using 30 layers per channel in the RGB color space, with skeleton bundling. (150kB, SSIM 0.8846).

Figure 6.36: Different color images of Lena.

(a) Original color image of peppers. (512 x 512, 769kB)

(b) Skeletonization of Figure 6.37a using 20 layers per channel in the RGB color space. (125kB, SSIM 0.9049).

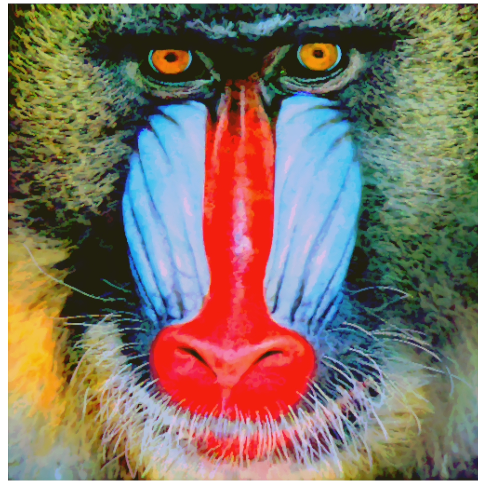
(c) Skeletonization of Figure 6.37a using 20 layers per channel in the HSV color space. (135 kB, SSIM 0.8837).

(d) Skeletonization of Figure 6.37a using 20 layers per channel in the YCbCr color space. (81 kB, SSIM 0.8088).

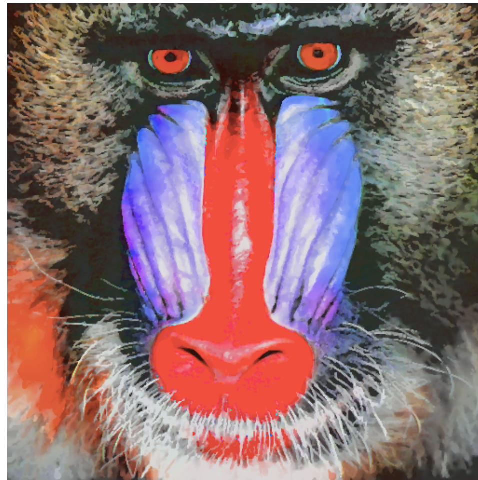
(e) JPEG compressed version of Figure 6.37a. (128kB, SSIM 0.9912).

Figure 6.37: Different color images of the Peppers image.

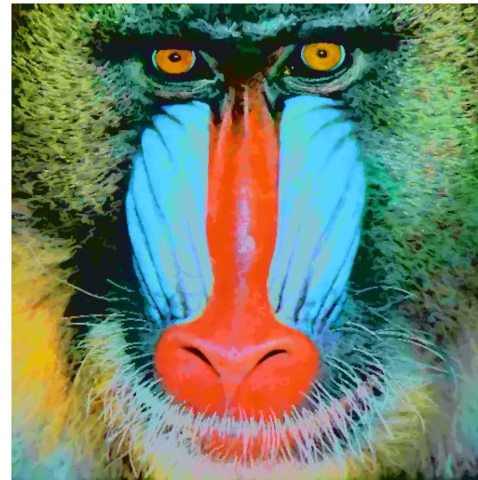
(a) Original color image of mandril. (512 x 512, 769kB)



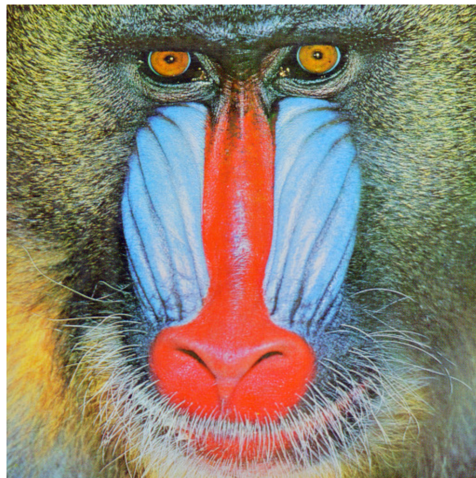
(b) Skeletonization of Figure 6.38a using 20 layers per channel in the RGB color space. (288kB, SSIM 0.6338).



(c) Skeletonization of Figure 6.38a using 20 layers per channel in the HSV color space. (250 kB, SSIM 0.5545).

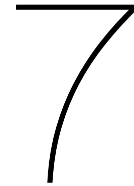


(d) Skeletonization of Figure 6.38a using 20 layers per channel in the YCbCr color space. (189 kB, SSIM 0.5981).



(e) JPEG compressed version of Figure 6.38a. (224kB, SSIM 0.9768).

Figure 6.38: Different color images of the Mandril image.



Conclusion & Future work

We have presented a further exploration of using skeletons for shape representation and simplification. With this research we have described a general image compression and manipulation framework which can handle any type of image – be it a natural photograph, simple or complex image, images with text, computer generated, etc. It works on binary, grayscale and color images of arbitrary bit depth. It has a simple set of parameters and can encode images in general within one second, even though we assume there is still enough room for optimization. This makes it applicable in a very wide context.

We have studied and extended the original pipeline of Meiburg and extended it with various new options. These new options include an improved layer selection algorithm – described in section 5.2, exploited inter-level skeleton coherence by introducing overlap pruning, as described in subsection 5.4.2 and skeleton path bundling – described in subsection 5.4.1, and improved interpolation for image reconstruction as described in section 5.7.

Whereas the original pipeline was too time consuming – taking around two minutes of computation time on most images – and generated too large images – often double the size of a higher-quality JPEG file – we have improved it to the point where processing an image now typically takes 1 or 2 seconds and produces images of fair quality – i.e. MS-SSIM > 0.9 – at file sizes two to twelve times smaller than high-quality JPEG. However, high-quality skeleton images are still larger than high-quality JPEG images but we have shown that these high-quality skeleton images can successfully be used as a *preprocessor* for a JPEG encoding which allows for 10% fewer JPEG quantization at the same file size compared to “vanilla” JPEG.

The introduction of skeleton-path based bundling introduces new possibilities for image manipulation. We have shown that this can be implemented efficiently and presented its interesting new effects in subsection 5.4.1. It offers local contrast enhancement/“specular” highlighting which can introduce desirable effects in some cases. Moreover, the skeleton path bundling in different color spaces also shows interesting “artifacts”, albeit in color space or in shapes. Besides of the effects, it also aids compression by increasing the inter-layer coherency, resulting in 30% smaller file sizes.

From here, several steps can be taken for future research. Compression of images can be further improved in the following ways. One possible option is implementing a custom run-length encoding. It often happens that several consecutive skeleton points share the same difference either in terms of location displacement or radius difference. This could be exploited for a more compact skeleton path encoding.

Moreover, we have identified a better algorithm for layer selection, but all layer selection algorithms considered so far are *global*. It might be be useful to select a number layers *per shape* rather than per image.

Another possible way of improving compression is not by storing the full skeleton paths but approximations of paths by splines in 3D $\{x, y, DT\}$ space. By storing only the control points of the splines rather than the full points additional space might be saved.

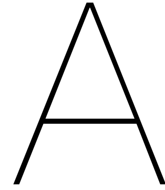
The entropy coders currently used for Huffman- and Arithmetic coding are very rudimentary and not very sophisticated. Throughout the past years several more advanced entropy coders have been developed for several video and image formats such as CABAC for H.264/H.265, VP8 for WebM/WebP,

or MANIAC for the FLIF image format. These could be incorporated in our pipeline for a better compression result.

It might also be possible to extend our method to a video format now that the performance is significantly higher. This allows for another source of information for skeleton data encoding; rather than intra- and inter-layer encoding, also information between frames can be taken into account.

Our dense skeleton image representation allows for image simplification while maintaining aesthetics. One motivation for image simplification is to remove unwanted features and maintain only wanted features. It is worthwhile to verify that these features indeed remain for some of the intended applications such facial recognition, object recognition or preservation of regions of interest.

Appendices



How-to

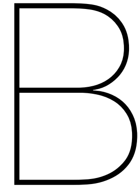
This section describes how the software can be used and how results can be reproduced. The project is divided in two folders, `imConvert` and `imShow`. Each directory contains a Makefile, running `make` in the directory should be sufficient when all dependencies as listed below are satisified. Possibly the paths to CUDA libraries need to be modified to match your system. This produces two executables: `skeletonify` and `show_skeleton`, respectively. `skeletonify` accepts a configuration file – example below – and produces a SIR file. `show_skeleton` accepts a SIR file, which it then shows.

A.1. Dependencies

Dependencies that are required are a C++ compiler (Tested with gcc 6.3.1 and clang 3.9.1), CUDA (version 8), make (version 4.2.1), cmake (version 3.7.2), boost (version 1.63), ragel (version 6.9), and vala (version 0.34). Moreover it requires an OpenGL implementation, for which we used GLUT (freeglut version 3) and GLEW (version 2). Here is an example configuration file with which `skeletonify` can be called:

```
# input image. Must be PGM or PPM
filename = ../examples/lena_gray_512.pgm
# Verbose output
outputLevel=v
# number of layers
num_layers = 20
# layer selection method, other allowed value is thresholding
layer_selection = peaks
# skeleton connected size threshold
sThreshold = 5
# Saliency treshold
ssThreshold = 2
# Island size threshold
siThreshold = 3
# Output file name
outputFile = ./lena512-bundled.sir
# external compression algorithm
compression_method = zpaq
# perform overlap pruning?
overlap_pruning = false
# allowed radius difference before pruning
overlap_pruning_epsilon = .05
# encoding method
encoding = traditional
# perform bundling?
```

```
bundle = false
alpha = 1
epsilon = 5
```



File Format

For reliable reconstruction and comparison, it is necessary to have a standard container format representing our image file. Therefore, we a container format for Skeleton Image Representation (SIR) files. The container has a header describing the data and the data itself. The header has the following layout:

Version The version of the SIR file with which this file is encoded (16 bits).

Width Image width (16 bits).

Height Image height (16 bits).

Colorspace The color space of the image, starting from one it can be respectively Grayscale, RGB, HSV or YCbCr.

Clear color The lowest intensity present in the original image. Necessary for maintaining contrast in the final image.

After this, follow the image planes. If the colorspace is gray, only one plane follows otherwise three planes follow. The meaning of each plane is determined by the colorspace. A plane consist of a byte giving the number of intensities i, followed by so many bytes where each byte is an intensity. After this, the dense skeleton data begins, which consists of i layers. Each layer consists of two bytes denoting how many paths occur in that layer, followed by that many paths. Each path consists of a starting point – which has two bytes for each component of the skeleton point. Each starting point is followed by a chain of tree-encoding points according to the previously determined encoding scheme. Each chain point is either a difference point (1 byte), a fork tag (1 byte, `0x38`) followed by two bytes whose value is how far one has to go back in the tree to move to the next branch, or an end tag (1 byte, `0x39`) denoting the end of a path. A layer contains thus as many end tags as there are paths in the tree.

Bibliography

Nasir Ahmed, T Natarajan, and Kamisetty R Rao. Discrete cosine transform. *IEEE transactions on Computers*, 100(1):90–93, 1974.

J Alakuijala and Z Szabadka. Brotli compressed data format. Technical report, 2016.

Jyrki Alakuijala. Lossless and transparency encoding in webp. URL `https://developers.google.com/speed/webp/docs/webp_lossless_alpha_study`.

Suzanne Angeli, Arnaud Quesney, and Lydwine Gross. *Image Simplification Using Kohonen Maps: Application to Satellite Data for Cloud Detection and Land Cover Mapping*. Citeseer, 2012.

K Bandaru and Patiejunas K. Under the hood: Facebook's cold storage system, 2015. URL `https://code.facebook.com/posts/1433093613662262/-under-the-hood-facebook-s-cold-storage-system-/`.

Jim Bankoski, Paul Wilkins, and Yaowu Xu. Technical overview of vp8, an open source video codec for the web. In *ICME*, pages 1–6, 2011.

Harry Blum. Biological shape and visual science (part i). *Journal of theoretical Biology*, 38(2):205–287, 1973.

Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel banding algorithm to compute exact distance transform with the gpu. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 83–90. ACM, 2010.

Y Collet and C Turner. Smaller and faster data compression with zstandard, 2016-08-31. URL `https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/`.

James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, 1996.

Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

Ozan Ersoy, Christophe Hurter, Fernando Paulovich, Gabriel Cantareiro, and Alex Telea. Skeleton-based edge bundling for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2364–2373, 2011.

Joseph Fourier. *Theorie analytique de la chaleur, par M. Fourier*. Chez Firmin Didot, père et fils, 1822.

fusiyuan2010. Csc: A loss-less data compression algorithm inspired by lzma, 2016.

Philip Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, 1994.

Marcus Geelnard. Basic compression library, 2007.

Ilya Grebnov. libbsc: A high performance data compression library, 2011.

Fabio G Guerrero. A new look at the classical entropy of written english. *arXiv preprint arXiv:0911.2284*, 2009.

Richard F Haines and Sherry L Chuang. The effects of video compression on acceptability of images for monitoring life sciences experiments. 1992.

Daniel Reiter Horn. Lepton image compression: saving 2215mb/s, 2016. URL https://blogs.dropbox.com/tech/2016/07/lepton-image-compression-saving-22-losslessly-from-images-at-15mbs/.

David A Huffman et al. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

Jan Kazemier. Connected morphological attribute filters on distributed memory parallel machines, 2016.

David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

Matt Mahoney. The zpaq compression algorithm. 2015.

Yuri Meiburg. Dense-Field skeleton-based image representations. 2011.

Fernand Meyer. Levelings, image simplification filters for segmentation. *Journal of Mathematical Imaging and Vision*, 20(1-2):59–72, 2004.

Evan Nemerson. Squash - compression abstraction library.

Giuseppe Papari, Nicolai Petkov, and Patrizio Campisi. Artistic edge and corner enhancing smoothing. *IEEE Transactions on Image Processing*, 16(10):2449–2462, 2007.

I Pavlov. 7-zip and lzma sdk,", 1999.

Iain E Richardson. *The H. 264 advanced video compression standard*. John Wiley & Sons, 2011.

Boris Yakovlevich Ryabko. Data compression by means of a "book stack". *Problemy Peredachi Informatsii*, 16(4):16–21, 1980.

Amir Said. Fast arithmetic coding (fastac) implementations, 2004.

James A Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

Julian Seward. bzip2, 1998.

John L Shanks. Computation of the fast walsh-fourier transform. *IEEE Transactions on Computers*, 18 (5):457–459, 1969.

Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 1948.

Hamid R Sheikh, Zhou Wang, Lawrence Cormack, and Alan C Bovik. Live image quality assessment database release 2, 2005.

USC SIPI. The usc-sipi image database, 2016.

Jon Sneyers and Pieter Wuille. Flif: Free lossless image format based on maniac compression. In *Image Processing (ICIP), 2016 IEEE International Conference on*, pages 66–70. IEEE, 2016.

Andrea Tagliasacchi, Thomas Delame, Michela Spagnuolo, Nina Amenta, and Alexandru Telea. 3d skeletons: A state-of-the-art report. In *Computer Graphics Forum*, volume 35, pages 573–597. Wiley Online Library, 2016.

A.C. Telea. Real-time 2d skeletonization using cuda, 2014. URL http://www.cs.rug.nl/svcg/Shapes/CUDASkel.

Alexandru Telea. Feature preserving smoothing of shapes using saliency skeletons. In *Visualization in Medicine and Life Sciences II*, pages 153–170. Springer, 2012.

Alexandru Telea and Jarke J Van Wijk. An augmented fast marching method for computing skeletons and centerlines. In *Proceedings of the symposium on Data Visualisation 2002*, pages 251–ff. Eurographics Association, 2002.

Kinh Tieu and Paul Viola. Boosting image retrieval. *International Journal of Computer Vision*, 56(1-2): 17–36, 2004.

Florence Tushabe and MHF Wilkinson. Image preprocessing for compression: Attribute filtering. In *Proceedings of International Conference on Signal Processing and Imaging Engineering (ICSPIE'07)*, pages 1411–1418. Citeseer, 2007.

Matthew van der Zwan, Valeriu Codreanu, and Alexandru Telea. Cubu: Universal real-time bundling for large graphs. 2016.

Andre M van Dijk, Jean-Bernard Martens, and Andrew B Watson. Quality asessment of coded images using numerical category scaling. In *Advanced Networks and Services*, pages 90–101. International Society for Optics and Photonics, 1995.

Gregory K Wallace. The jpeg still picture compression standard. *IEEE transactions on consumer electronics*, 38(1):xviii–xxxiv, 1992.

Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Seventh Asilomar Conference on*, volume 2, pages 1398–1402. Ieee, 2003.

Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

Matthew Van Der Zwan, Yuri Meiburg, and Alexandru Telea. A Dense Medial Descriptor for Image Analysis. *Proceedings of the eighth VISAPP conference*, pages 285–293, 2013. URL `http://www.cs.rug.nl/{~}alext/PAPERS/VISAPP13/dskel.pdf`.