Refinements of Data Compression Using Weighted Finite Automata

Vom Fachbereich Elektrotechnik und Informatik
der Universität Siegen
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

von
Frank Katritzke

1. Gutachter: Prof. Dr. Wolfgang Merzenich
2. Gutachter: Prof. Dr.-Ing. Bernhard Freisleben

Tag der mündlichen Prüfung:

Contents

Vorwort

Der effizienten Speicherung und Übertragung von Bildern kommt in den letzten Jahren verstärkte Bedeutung zu, etwa bei Multimedia- oder WWW-Anwendungen. In den letzten Jahren haben sich Anwendungen wie Internet-Browser oder Multimedia-Lexika bedeutend ausgebreitet. Die dabei übertragenen und gespeicherten Bilder nehmen selbst in komprimierter Form mit gängigen Bildkompressionsverfahren wie GIF oder JPEG einen großen Teil des benötigten Speicherplatzes ein. Aus diesem Grund beschäftigt man sich weiterhin intensiv mit der Kompression von digitalisierten Bildern.

Die Entwicklung solcher Kompressionsverfahren verlief dabei sowohl über verlustlose Verfahren wie Lempel-Ziv- und Huffman-Kodierung, als auch verlustbehaftete Verfahren wie Vektorquantisierung und Transformationskodierung. Eine allgemeine Übersicht solcher Kompressionsverfahren befindet sich beispielsweise in [Kat94, NH88]. Zu den verlustbehafteten Kodierungsverfahren gehören auch die sogenannten fraktalen Kodierer, bei denen sich insbesondere Kodierer für iterierte Funktionensysteme (iterated function system, IFS) als auch die gewichteten endlichen Automaten (weighted finite automaton, WFA) hervorgetan haben. Die meiste Aufmerksamkeit gilt dabei den IFS-Kodierern, soweit man das an der Anzahl der dedizierten Literatur ablesen kann. Um dieses Ungleichgewicht zumindest geringfügig zurechtzurücken, wird in dieser Arbeit das Gewicht auf die Erforschung der Methode der WFA-Kodierung gelegt. Dazu wurde ein moderner WFA-Kodierer namens AutoPic entwickelt und implementiert, der unter anderem folgende Eigenschaften besitzt:

- bottom-up Erzeugung des Automaten,
- Benutzung von Bintrees,
- lineare und Matching-Pursuit-Approximation, die das Verhältnis von Fehler und Kosten abwiegt. Diese Abwägung wird für alle Entscheidungen benutzt, die die Approximation betreffen,
- Matching Pursuit Approximation mit Second Chance Heuristik,
- Entropie-Kodierung mit einem arithmetischen Kodierer,
- Modellbildung mit endlichem Kontext und Kontextmischung,
- Optimierung der Kodierereffizienz durch genetische Algorithmen,
- Speicherung der Baumstruktur und vereinfachten HV-Partitionierung mittels einer Mischung aus Lauflängenkodierung und statistischer Kodierung,
- Kombination mit Wavelet-Transformationen,
- Nachbearbeitung des dekodierten Bildes durch Kantenglättung.

Diese Arbeit führt außerdem neue Techniken in die WFA-Kodierung ein, wie

- eine abgeschwächte Version der HV-Partitionierung, die Light-HV Partitionierung genannt wird,
- die Einführung einer (exakten) Kosinus-Basis als Menge von Initialzuständen,
- die Einführung eines statistischen Modells, das die exakte Berechnung der Kosten für die Speicherung von WFA-Parametern unter bestimmten Umständen erlaubt,
- eine neue Approximations-Alternative, die *Bad Approximation* genannt wird,
- unabhängige Zuweisung von Domainvektoren in verschiedenen Ebenen des WFA-Baumes,
- eine Mischung zwischen BFS- und DFS-Aufbau des WFA-Baumes,
- Aufteilung der Matching-Pursuit-Vektoren in folgende Teile:
  - DC-Teil,
  - AC-Teil sowie
  - MP-Teil.

Obwohl der Hauptaspekt dieser Arbeit auf der Implementierung eines effizienten Bildkodierungsalgorithmus liegt, wurde ein einfacher Video-Kodierer mit bi-direktionaler Bewegungskompensation implementiert. Dessen Kompensationsentscheidungen beruhen ebenfalls auf einer Abwägung von Approximationsfehler und erzeugten Speicherungskosten. Auf diese Weise wird gezeigt, daß sich die Technik der WFA-Kodierung ebenfalls für Bildsequenzen eignet.

Preface

The efficient storage and transmission of digital images is gaining increasing interest, for example in multimedia and WWW applications. Pictures require a huge amount of storage capacity even in compressed form (as GIF or JPEG). For this reason, many researchers study the area of digital image compression extensively.

The development of compression algorithms evolved from lossless techniques such as Lempel-Ziv and Huffman coding to lossy techniques such as vector quantization and transform coding. An overview of such methods can be found in [Kat94, NH88]. One of the major techniques for state of the art image compression is the fractal-based technique. One version of these are based on iterated function systems (IFS) which have drawn by far the most attention as far as one can tell from the amount of dedicated literature. Other fractal-based techniques are the codecs based on weighted finite automata (WFA) which have been neglected by most researchers. In order to slightly correct this bias, this thesis concentrates on the topic of image coding with WFAs. We have implemented a state of the art WFA coder called AutoPic, which has the following features:

- bottom-up generation of the automaton,
- usage of bintrees,
- rate–distortion constrained linear and matching pursuit approximation. The rate–distortion feature is exploited for all decisions which concern approximation,
- matching pursuit approximation with the second chance heuristic,
- entropy coding by an arithmetic coder,
- finite context modeling with blending,
- optimization of encoder efficiency with genetic algorithms,
- storage of the tree structure and light HV partitioning with a mixture of run length and statistical coding,
- combination with wavelet transforms,
- post-processing of the decoded image by edge smoothing.

This thesis also introduces several new techniques to WFA coding as

- a restricted version of HV partitioning, called light HV partitioning,
- the utilization of an (exact) cosine basis as WFA initial states,

- a statistical model allowing the computation of the exact cost for the storage of WFA parameters under certain circumstances,
- a new approximation alternative called “bad approximation”,
- independent assignment of domain pool vectors in the levels of the WFA tree,
- a mixture of BFS and DFS generation of the WFA tree,
- splitting the matching pursuit vectors into:
  - DC part,
  - AC part and
  - MP part.

Although the main aspect of this thesis is the implementation of an efficient image encoder, a simple video codec was also implemented. It offers rate–distortion constrained bidirectional motion compensation to show that the WFA technique is also suited for video coding.

Outline of the Thesis

This thesis is designed to present several new aspects of image and video coding using weighted finite automata. In order to rate the enhancements, we have implemented an experimental image and video codec named AutoPic, which implements most of the techniques described in this thesis. In order to reach a wide audience, we have added the basics of most aspects surrounding the topic of digital image and video compression. As we will see, a wide variety of enhancements to WFA coding is examined.

In chapter 1 we present fundamentals of digital image compression, e.g. some statistical prerequesites, foundations of information, coding and approximation theory.

In chapter 2 deals with some basics of entropy coding. We briefly describe the best known statistical encoder, the arithmetic codec. This chapter also describes some kinds of statistical models from which we have to choose the best for use in WFA image compression.

Chapter 3 presents image partitioning techniques and the definition of WFAs. This chapter also provides encoding and decoding algorithms for weighted finite automata. The main aspect of this chapter is to choose the best suited construction algorithm to apply.

In chapter 4 we will examine several ways to enhance the compression efficiency of our WFA encoder. Some of the topics considered are the image partioning techniques, the approximation technique, statistical models, domain pool administration and optimization of coding parameters. The chapter is concluded by some results of the WFA encoder and several further research topics.

Chapter 5 is devoted to the reduction of tiling effects by combining the WFA technique with wavelet transforms. Therefore we will first describe some fundamentals of wavelet transforms and some postulations the utilized transform should fulfill. The chapter is concluded by some results of the combined encoder.

The next chapter is intended to show that the WFA technique may be combined with video coding methods. We will present some techniques which may be combined with WFA coding and will choose the most suitable.

Chapter 7 deals with some implementational issues. We will present the user interface and package structure of AutoPic. This chapter also presents several optimizations of the WFA codec.

The last chapter concludes the thesis and gives some suggestions for further research.

The last part of this thesis is filled by the appendix. Here we will add some themes required for further reading. Some of the topics are color spaces, measures of image fidelity, image bases and scalar quantization.

Chapter 1

Preliminaries

This chapter introduces some mathematical preliminaries and notations required for reading this thesis. The importance of some topics will become clear in the next chapters.

1.1 Representation of an Image

Mathematically, an image is interpreted as a function $p : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. The main difference to files examined in classical data compression is the two-dimensionality. In analogy to existing optical systems, the range of $p(.,.)$ is considered to be limited. Because the sight angle of optical systems is limited, we also assume the domain of $p(.,.)$ to be limited. Without loss of generality we can expect the domain to be rectangular. We call the value $p(x, y)$ the *gray value* of the image p at the position (x, y) where we assume Cartesian coordinates as usual in image definitions. The more general case of color images will be examined later.

In order to operate on the image function with a digital computer, we have to take a finite set of samples into account. This operation is called *sampling* in image processing. A commonly used sampling technique is to take the samples at lattice points. This means that the sampling points are equidistant[1] in x and y direction. To summarize these assumptions, we use integer coordinates in the range $0 \leq x < m$ and $0 \leq y < n$.

As samples are real values, we include another processing step called *quantization*. In this step, the sample values are replaced by numbers which are members of a finite set. The assigned values are called *quantization steps* and the resulting error is called *quantization noise* due to its origin in audio coding. For methods to reduce quantization noise see for example [Kat94]. The result

[1] The assumption that the sampling points are equidistant is not mandatory but common because of technical reasons. This also simplifies many mathematical models.

of sampling and quantization is commonly called a *digital image* and its components are called *pixels*[2]. If no confusion may occur, we will often write the x and y coordinates as indices. This representation is commonly called *PCM*[3]. For an illustration of sampling and quantization see Figures 1.1 and 1.2 (because of illustrational reasons we drew the PCM coding of a one-dimensional signal). Note that a discrete image may be interpreted as a matrix.

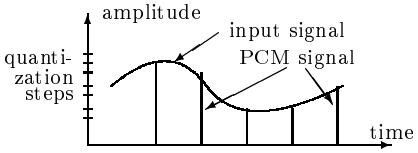


Figure 1.1: PCM coding.

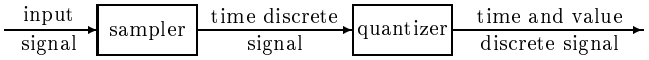


Figure 1.2: Discretization of a signal.

In practice, the sampling of a real valued function is typically performed at a frequency called the Nyquist rate[4]. The quality of PCM coding is mainly determined by the following two parameters [Llo82]:

1. the number of quantization steps and
2. the sampling frequency.

Early lossy image compression schemes used the fact that the human visual system is insensitive to sampling errors. This method is called *sub-sampling* [Kat94]. A frequently used variant is *DPCM* (differential pulse code modulation) described in [Jai81, Kat94, NL80].

For some image processing techniques, an ordering of the pixels or image blocks is required. Therefore, we assume that the origin of the coordinate system is the upper left corner of the image (see Figure 1.3) and that we traverse the rows from left to right and operate the columns top down (*raster scan order*). For this and some other orderings see Figure 1.4. More information about image representations can be found in [NH88].

Treatment of Boundaries Many image processing applications such as filtering or wavelet transforms assume infinite signal lengths. In order to use

[2]Also called *pel* as an abbreviation for picture element.

[3]PCM is an abbreviation for pulse code modulation.

[4]The Nyquist rate is equal to twice the highest occurring frequency in the Fourier transform of the input signal. By sampling at at least this frequency the input signal can be reconstructed exactly [NH88].

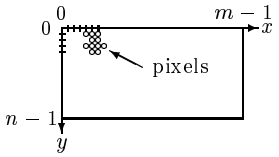


Figure 1.3: Coordinate system of a digital image.

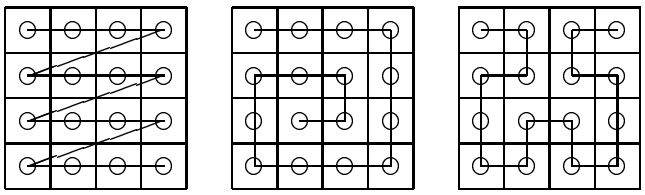


Figure 1.4: Some orders of image blocks. They are called (from left to right): *raster scan order*, *spiral order* and *Hilbert order*.

such applications, one has to extend the image over the boundaries. There are several possibilities to extend images, including

- The missing pixels are filled with constant values, for example 0 or the value of the nearest pixel in the boundary.
- The values are extended periodically. This method is often used in Fourier analysis.
- The values are extended symmetrically. This type of boundary treatment is often used in wavelet analysis.

Note that there are also at least four subtypes of periodic and symmetric extension, as the first or the last sample may be repeated. Since such differences may lead to unwanted side effects, the type of boundary treatment has to be chosen carefully for the specific application to utilize.

Distortion Measures Used in this Thesis In this thesis we adapted the distortion measures to the work of Hafner [Haf99], who measures the distortion between two gray valued images $e, \tilde{e} : \{0, \ldots, m-1\} \times \{0, \ldots, n-1\} \to \{0, \ldots, 255\}$ as

$$\text{PSNR}=20\log_{10}\frac{255}{\text{RMSE}} \tag{1.1}$$

(peak signal to noise ratio) with

$$\text{RMSE}=\sqrt{\frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left(e_{i,j}-\tilde{e}_{i,j}\right)^2} \tag{1.2}$$

(root mean squared error). The bit rate is calculated in the common bpp notation (bits per pixel)

$$\text{bpp}=\frac{\text{size of output stream in bits}}{\text{number of pixels in the image}}. \tag{1.3}$$

All rate–distortion diagrams are made of tuples (rate, distortion) where the rate is measured in bpp and the distortion is the PSNR measured in dB. For a more detailed discussion on distortion measures of images see appendix C on page 169.

1.2 Statistical Prerequisites

This section is provided to define the notation used. We only introduce statistical properties necessary for understanding the fundamentals of data compression. First we introduce some definitions of continuous probability spaces and continue with discrete probability spaces.

1.2.1 Continuous Probability Spaces

In the following, we consider a statistical experiment. The set of the possible outcomes is called Ω and its elements are written as ω.

A *σ-algebra* $\mathcal{S}$ is a subset of the power set of Ω with the following properties:

$$\Omega \in \mathcal{S}, \tag{1.4}$$

$$A \in \mathcal{S} \;\Rightarrow\; \Omega \setminus A \in \mathcal{S}, \tag{1.5}$$

$$A_i \in \mathcal{S} \;\Rightarrow\; \cup_{i\in I} A_i \in \mathcal{S} \text{ for } I \subseteq \mathbb{N}. \tag{1.6}$$

The elements of $\mathcal{S}$ are called *events*. Two events have a special meaning: the *impossible event* $A = \varnothing$ and the *sure event* $A = \Omega$.

Let $\mathcal{T} \subseteq \mathcal{P}(\Omega)$ be a system of sets. The *smallest σ-algebra* $\sigma(\mathcal{T})$ containing $\mathcal{T}$ is given by

$$\sigma(\mathcal{T})= \bigcap_{\mathcal{T} \subseteq \mathcal{C} \subseteq \mathcal{P}(\Omega),\, \mathcal{C}\ \sigma\text{-algebra}} \mathcal{C}. \tag{1.7}$$

In general, the probabilities are defined by a *probability measure*

$$P : \mathcal{S} \to [0, 1] \tag{1.8}$$

with the following properties:

$$P(\Omega) = 1, \tag{1.9}$$

$$P(\cup_{i\in I} A_i) = \sum_{i\in I} P(A_i) \text{ for } I \subseteq \mathbb{N}, A_i \in \mathcal{S} \text{ pairwise disjoint.} \tag{1.10}$$

The term *pairwise disjoint* means that $A_i \cap A_j = \varnothing$ for $i \neq j$. Note that $P(\varnothing) = 0$. A *probability space* is a triplet $(\Omega, \mathcal{S}, P)$. A *measurable space* $(S, \mathcal{B})$

is a set S with σ-algebra $\mathcal{B}$. A mapping $X : \Omega \to S$ is called *measurable* with respect to σ-algebras $\mathcal{S}$ and $\mathcal{B}$ iff[5]

$$X^{-1}(B) \in \mathcal{S} \quad (B \in \mathcal{B}). \tag{1.11}$$

A measurable mapping from one probability space to a measurable space is called *random variable*. The random variable X induces a probability measure $(XP)(B)=P(X^{-1}(B))$ on $\mathcal{B}$, called *distribution* of X. From now on, we will set $S = \mathbb{R}$ and introduce the notations

$$\{X = x\} = \{\omega : X(\omega) = x\}, \tag{1.12}$$

$$\{X \leq x\} = \{\omega : X(\omega) \leq x\}, \tag{1.13}$$

$$\{x_0 \leq X \leq x_1\} = \{\omega : x_0 \leq X(\omega) \leq x_1\}. \tag{1.14}$$

These notations can be similarly defined for other comparison operators.

Let $X : (\Omega, \mathcal{S}, P) \to (\mathbb{R}, \mathbb{B})$ be a real valued random variable. Here $\mathbb{B}$ indicates the *Borelean σ-algebra* generated by the semi infinite intervals $(-\infty, t]$, i.e. $\mathbb{B}=\sigma(\{(-\infty, t] : t \in \mathbb{R}\})$. The distribution of X applied to the *half rays* $(-\infty, t]$ with $t \in \mathbb{R}$ defines the *distribution function*

$$F(t) = XP((-\infty, t]) \tag{1.15}$$

$$= P(X^{-1}((-\infty, t])) \tag{1.16}$$

$$= P(\{X \leq t\}) \tag{1.17}$$

with the following properties

1. $\lim_{t\to\infty} F(t) = 1$,
2. $\lim_{t\to-\infty} F(t) = 0$,
3. F is monotonically increasing.

If F is differentiable, we call $f(t) = F'(t)$ the *(Lebesgue-)density* of F with respect to the probability measure XP with distribution function F.

The *expectation value* (or shorter *expectation*) of the random variable X is defined as

$$E(X)= \int_{-\infty}^{\infty} t f(t)\, dt, \tag{1.18}$$

if this integral exists. The value

$$\mathrm{Var}(X)= \int_{-\infty}^{\infty} (t - E(X))^2 f(t)\, dt \tag{1.19}$$

is called *variance* of the random variable X and its square root is called the *standard deviation* of the random variable X, where the existence of the integral is assumed.

[5] In keeping with mathematical literature, *iff* means "if and only if".

1.2.2 Discrete Probability Spaces

From now on, we assume that $\Omega = \{\omega_i : i \in I, I \subseteq \mathbb{N}\}$ and $\mathcal{S} = \mathcal{P}(\Omega)$. The probability measure P is uniquely defined by the values on the singletons (one-elemental subsets) by

$$p_i{=}P(\{\omega_i\}). \tag{1.20}$$

Then we have

$$\sum_{i\in I} p_i = 1. \tag{1.21}$$

In this case we define the *expectation* of the random variable X as

$$E(X){=}\sum_{i\in I} X(\omega_i)p_i \tag{1.22}$$

(if this sum exists) and the *variance* of X as

$$\mathrm{Var}(X){=}\sum_{i\in I}(X(\omega_i) - E(X))^2 p_i. \tag{1.23}$$

Conditional Distributions

Let

$$X: \qquad (\Omega, \mathcal{P}(\Omega), P) \to (S, \mathcal{B}) \tag{1.24}$$

and

$$Y: \qquad (\Omega, \mathcal{P}(\Omega), P) \to (T, \mathcal{C}) \tag{1.25}$$

be discrete random variables and $B, C \subseteq \Omega$. Then the term

$$P(Y \in C|X \in B){=}\begin{cases} \dfrac{P(\{X \in B, Y \in C\})}{P(\{X \in B\})} & \text{if } P(\{X \in B\}) > 0 \\ P(\{Y \in C\}) & \text{if } P(\{X \in B\}) = 0 \end{cases} \tag{1.26}$$

is called *conditional probability* of $Y \in C$ given $X \in B$. The function

$$F(t) = P(Y \leq t|X \in B){=}\begin{cases} \dfrac{P(\{X \in B, Y \leq t\})}{P(\{X \in B\})} & \text{if } P(\{X \in B\}) > 0 \\ P(\{Y \leq t\}) & \text{if } P(\{X \in B\}) = 0 \end{cases} \tag{1.27}$$

is correspondingly called *conditional distribution function* of Y given $X \in B$.

1.2.3 Arrays of Random Variables

We now generalize the concept of random variables and therefore consider the family of functions

$$X_s(\omega_i) \text{ with } s \in \mathcal{I} \tag{1.28}$$

where $\mathcal{I}$ is an interval of the Euclidean space. If $\mathcal{I}$ is one dimensional, we call $X_s(\omega_i)$ a *stochastic process*; if $\mathcal{I}$ is higher dimensional, then we call $X_s(\omega_i)$ a *random variable array*. In the following, we focus on the discrete two dimensional case.

1.2.4 Images Considered as Arrays of Random Variables

Since we do not know in advance which image will be operated on, we may interpret the gray values of an image as random variables. In order to make the following formulas more readable, we assume that the mean value of the random variables is zero:

$$E(X_{i,j}) = 0 \text{ for all } i, j \in \mathcal{I}. \tag{1.29}$$

In the following considerations, we introduce a measure for dependency of the gray values of adjacent pixels. In the following sections we assume that the variance of the random variables is finite, i.e. $\mathrm{Var}(X_{i,j}) < \infty$. Empirical investigations on digital images have shown that the *autocorrelation function*

$$R(i_0, j_0, i_1, j_1) = E(X_{i_0,j_0} X_{i_1,j_1}) \tag{1.30}$$

can be approximated by the function

$$\sigma^2 e^{-(\alpha|i_0 - i_1| + \beta|j_0 - j_1|)} \tag{1.31}$$

with $\alpha, \beta, \sigma \in \mathbb{R}$ chosen properly [WH71]. The autocorrelation function therefore depends only on the relative positions of the considered pixels and not on their position in the image. In this case the autocorrelation function is called *homogeneous*. In the following, we will assume the autocorrelation function to be homogeneous and write

$$R(i, j) = R(a, b, a + i, b + j), \tag{1.32}$$

for arbitrary values of a and b. The matrix $R(i, j)$ is called *correlation matrix*.

Little is known about the distributions of gray values of digitized images. Researchers often use mixtures of Gaussian and Laplacian distributions (see appendix J.1 on page 209) as an approximation. The lack of precise models for images is surely one of the main reasons for the many heuristics in image compression [Jai81].

1.3 Some Foundations of Information and Coding Theory

Let E and F be countable sets of symbols with at least two elements each which we call the *input alphabet* and *output alphabet*, respectively. With the word *code* we specify a non-empty set $C \subseteq F^+$. The elements of C are called *code words*. The number $|F|$ denotes the *order of the code* ($|F|$-ary code). For the rest of this thesis we assume $F = \{0, 1\}$ if nothing else is stated. With the word *coding* one denotes an (injective) mapping from E^+ to F^+. Note that for many codes, one has to restrict the input alphabet to finite cardinality. In accordance with

many articles about data compression, we use the less precise word *code* instead of coding.

A *block code* is a code whose code words have a fixed length in contrast to a *code of variable length*. In this thesis, we will only examine *uniquely decodable codes* meaning that each sequence of code words can only be decoded in one way. For example $\{1, 10, 01\}$ is not uniquely decodable, because the sequence 101 can be decoded as 1 01 and as 10 1. An example for uniquely decodable codes are the prefix codes which we will introduce later.

1.3.1 Information Theory

In the following, we define the mathematical term information content. In this definition, we do not assign any semantic meaning to the symbols but define the information gain of the receiver of a given message based only on the probabilities of the received symbols.

Our "surprise" about receiving the symbol $e \in E$ is anti-proportional to its probability. Therefore, we assign to symbols with smaller probability a higher information content than symbols with higher probability. Symbols with zero probability[6] are not allowed. We now assign the *information content*[7] I_e to a symbol $e \in E$ with probability $p_e > 0$ as

$$I_e = -\log_m(p_e) \text{ with } m \in \mathbb{R}, m > 1. \tag{1.33}$$

In this context, we choose $m = 2$ and call the unit of information *bit*[8], the base m is omitted if no confusions can occur. There are alternative definitions for information, but here we examine the information theory of C. E. Shannon, which perfectly fits our applications.

Note: There is a confusing ambiguity between the measure unit bit defined above and the storage unit bit. The two measure units have the strong relationship that an (electrical) device which can distinguish two states can *at most* store one bit of information. In German literature the former measure unit is therefore commonly called *bit* and the latter one is called *Bit* (as proposed in [HQ89]). In this thesis we call both measure units bit, in accordance with English literature. ■

Entropy

The term entropy was introduced in 1948 by C. E. Shannon in connection with the analysis of the information content of the English language. The expectation

[6] One could assign the symbols with probability 0 an information content of ∞, but that is not necessary in this thesis.

[7] Note that I is also a random variable.

[8] For $m = e$ one calls the unit of information *nat*.

H of the random variable $I : E \to \mathbb{R}$ is called *entropy* (of first order) and is thus defined by

$$H{=}E(I) = \sum_{e \in E} p_e I_e = -\sum_{e \in E} p_e \log p_e. \tag{1.34}$$

The entropy function is therefore a function which maps probability measures to real numbers. It would be more precise to write $H(P)$ but following the usual notation we also write H. In [HQ89] it is shown that H is continuous and takes its maximum (for finite alphabets) at the *uniform distribution* $p_e = 1/|E|$ for all $e \in E$. In this case, the entropy is given by $H = \log |E|$.

The entropy can be interpreted as a measure for the "randomness" and it serves as a lower bound for the number of bits required on the average to encode a single symbol (see section G.1 on page 197). With the entropy we have a criterion for the efficiency of (lossless) codes. For a graphical representation of the entropy function see Figure 1.5. There, the entropy of an information source with three symbols is plotted. The domain is two dimensional in this case, since the third probability is given by $p_0 + p_1 + p_2 = 1$. See [HQ89] for further details.

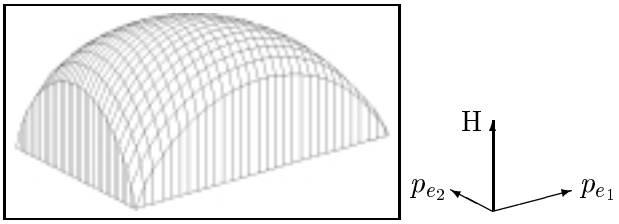


Figure 1.5: Entropy function.

1.3.2 Efficiency and Redundancy of a Code

Let $E = \{e_0, \ldots, e_{n-1}\}$ be an input alphabet with probability distribution $P = (p_0, \ldots, p_{n-1})$ with $p_i = P(e_i)$ and $C : E \to F^+$ a coding to an alphabet, with code words of lengths $l_i = |C(e_i)|$ with $i \in \{0, \ldots, n-1\}$. We denote the expectation of the code word length of the code C as

$$L_C{=}\sum_{i=0}^{n-1} p_i l_i. \tag{1.35}$$

We call

$$\mathrm{Eff}_C{=}\frac{H}{L_C} \tag{1.36}$$

the *efficiency of the code* C and

$$\mathrm{Red}_C{=}L_C - H = \sum_{i=0}^{n-1} p_i l_i - \sum_{i=0}^{n-1} -p_i \log p_i \tag{1.37}$$

the (statistical) *redundancy of the code* C. Let us remark that it is not precise to speak of an average code word length of a *code*, but this notion is commonly used and will not be changed here.

1.3.3 Prefix Codes

Prefix codes are codes of variable length in which no code word is the prefix of another code word. Therefore, a more precise notion for these codes would be *prefix free codes*. This condition ensures the decodability without lookahead, which is also called *instant decodability* or *Fano property*. In the following statements about prefix codes, we assume that the input alphabet is finite ($|E| < \infty$).

In order to illustrate the concept of prefix codes, we introduce the concept of the *code tree*, a tree with input symbols associated to the nodes (at most one symbol per node). We explain this without loss of generality for binary output alphabets. A code tree is a special interpretation of a *trie* (retrieval tree), which is a (binary) tree where the symbols of the input alphabet are stored in the leaves. One interprets the code word of a given symbol as the path from the root to the associated node. Therefore, we encode a 0 if we branch to the left subtree and otherwise encode a 1. An illustration of the trie concept is shown in Figure 1.6.

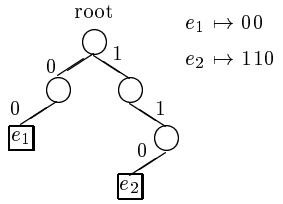


Figure 1.6: Code tree.

It can be shown that the optimal compression rate regarding symbol codes[9] can be achieved using a prefix code [HQ89]. This is the reason, why research about symbol coding may be limited to prefix codes.

Prefix codes are desirable because they allow relatively simple coders and decoders. At the coding stage, one can concatenate the code words without the need to insert extra symbols to separate the code words. The first code word is determined uniquely because of the prefix property (without lookahead). This code word is cut from the data stream and the next code words are successively obtained by the same procedure.

A *minimal prefix code* is a prefix code such that the following condition holds: if $\tilde{p}$ is a prefix of a code word, $\tilde{p}f$ with $f \in F$ is either a prefix of a code word or is itself a code word. This property prohibits that code words are longer

[9] A *symbol code* is a code assigning each input symbol a code word.

than necessary to ensure that the code is uniquely decodable. For this reason, we will only examine minimal prefix codes in the following. A minimal prefix code is always represented by a complete trie, meaning that each inner node always has two children. The trie in Figure 1.6 is not a complete trie, but can be transformed into a complete trie by cutting inner nodes. Because we restrict ourselves to complete tries, we can state that the coding tree possesses exactly $|E|$ leaves and $|E-1|$ inner nodes. If a coding tree T for a prefix code is given, one can easily calculate how many bits are required to encode a given message. For the generation of such codes see appendix F.3 on page 192. More details about information theory, entropy and prefix codes may be found in [MZ97, HQ89].

1.3.4 Coding of Alphabets with Infinite Cardinality

In some cases, one does not know the size of the input alphabet prior to coding. For this application there exists a huge amount of coding methods. Many of the coding methods described in this section have the additional advantage that the set of code words is fixed, simplifying both coding and decoding. In the following definitions we denote the concatenation operator of codewords by the symbol $\cdot$. If no confusion may occur, the concatenation operator is omitted.

Universal Codes A code C is called *universal* iff its average code word length L_C has the upper bound $k_0 H + k_1$ with $k_0, k_1 \in \mathbb{R}$ for each statistical distribution, provided that the input symbols are arranged in descending order of their probabilities. Such a code is called *asymptotic optimal* iff $k_0 = 1$ holds.

Unary Coding Unary codes $\alpha : \mathbb{N} \setminus \{0\} \to \{0,1\}^+$ are defined recursively in the following manner:

$$\begin{aligned} \alpha(1) &= 1, \\ \alpha(n+1) &= 0 \cdot \alpha(n). \end{aligned} \tag{1.38}$$

This code shows bad performance regarding data compression since $|\alpha(n)| = n$ but is introduced to build more complex codes.

Binary Coding The binary coding $\beta : \mathbb{N} \setminus \{0\} \to \{0,1\}^+$ is defined as

$$\begin{aligned} \beta(1) &= 1, \\ \beta(2n) &= \beta(n) \cdot 0, \\ \beta(2n+1) &= \beta(n) \cdot 1. \end{aligned} \tag{1.39}$$

Note that sequences of binary codes are not decodable unless the lengths of the codewords are known. Since $|\beta(n)| = \lfloor \log n \rfloor + 1$ the code[10] is very efficient.

[10] $\lfloor . \rfloor$ denotes the floor function $\lfloor x \rfloor = \max\{z \in \mathbb{Z} | z \leq x\}$.

The coding $\hat{\beta}$ is the coding β with the most significant bit removed (which is always 1). Because of this truncation of one symbol it follows that $|\hat{\beta}(n)| = \lfloor \log n \rfloor$.

The First Code of Elias The *first code of Elias* $\gamma : \mathbb{N} \setminus \{0\} \to \{0,1\}^+$ consists of $\lfloor \log n \rfloor$ zeroes followed by the binary representation of n:

$$\gamma(n) \quad = \quad \alpha\left(\left|\hat{\beta}(n)\right|\right) \cdot \hat{\beta}(n). \tag{1.40}$$

It follows that $|\gamma(n)| = 2\lfloor \log n \rfloor + 1$.

The Second Code of Elias The *second code of Elias* $\delta : \mathbb{N} \setminus \{0\} \to \{0,1\}^+$ maps a natural number n to a code word, consisting of $\gamma(\lfloor \log n \rfloor + 1)$, followed by the binary representation of n without the leading 1. The resulting code word has the length $2\lfloor \log(1 + \lfloor \log n \rfloor) \rfloor + 1 + \lfloor \log n \rfloor$:

$$\delta(n) \quad = \quad \gamma(|\beta(n)|) \cdot \hat{\beta}(n). \tag{1.41}$$

The Third Code of Elias The procedure as used to build the code δ can be applied recursively to build the third code of Elias $\omega : \mathbb{N} \setminus \{0\} \to \{0,1\}^+$. According to [HL87], this is worthwhile only for "astronomically high" numbers. The recursion can be performed in the following way: first the code $\beta(n)$ is written. The codeword $\beta(|\beta(n)| - 1)$ is written to the left of this codeword. This procedure is called recursively with each earlier code to the left being the β representation of the following codeword subtracted by 1. The recursion is stopped if the last codeword has the length 2. In order to uniquely mark the end of the code, a single 0 is appended since the code β always starts with a 1. The codes δ and ω are asymptotical optimal [HL87, BCW90] as opposed to γ. For Elias codes of some numbers see table 1.1. For more information on Elias codes see [Eli75].

Fibonacci Codes Another universal code is based on the Fibonacci numbers. Although the Fibonacci codes are not asymptotically optimal [HL87], they are an alternative for "small numbers".

The *Fibonacci numbers* (of order 2) are defined for this purpose as follows:

$$F_{-1} = F_0 = 1, \tag{1.42}$$

$$F_i = F_{i-2} + F_{i-1} \text{ for } i \geq 1. \tag{1.43}$$

Each natural number n has exactly one binary representation of the form $n = \sum_{i=0}^{k} d_i F_i$ (with $d_i \in \{0,1\}$ and $k \leq n$) without adjacent ones in the representation. A Fibonacci code word is generated by writing the d_i without leading zeroes one after the other, mirroring them and appending a 1. Because in the above representation there are no adjacent ones, the end of the code word

decimal	α	β	$\hat{\beta}$	γ	δ	ω
1	1	1		1.	1.	0
2	01	10	0	01.0	010.0	10.0
3	001	11	1	01.1	010.1	11.0
4	0001	100	00	001.00	011.00	10.100.0
5	00001	101	01	001.01	011.01	10.101.0
6	000001	110	10	001.10	011.10	10.110.0
7	0000001	111	11	001.11	011.11	10.111.0
8	00000001	1000	000	0001.000	00100.000	11.1000.0
9	000000001	1001	001	0001.001	00100.001	11.1001.0
10	0000000001	1010	010	0001.010	00100.010	11.1010.0

Table 1.1: Some codewords of the integers 1–10. For clarity the parts of the codes γ, δ and ω are separated by a dot, which is not part of the code.

decimal	F
1	11
2	011
3	0011
4	1011
5	00011
6	10011
7	01011
8	000011
9	100011
10	010011

Table 1.2: Some Fibonacci codewords.

is determined uniquely. The first ten Fibonacci codewords of order 2 are shown in table 1.2.

One can show that the Fibonacci codes are universal, but not asymptotical optimal. Despite that fact these code words are short for small[11] numbers. For further information about universal codes, see also [BCW90, HL87].

1.3.5 Adjusted Binary Codes

We now consider how to encode integers more efficiently if an upper limit is known. Henceforth we call this upper limit $m \in \mathbb{N}$. We can code an integer $n \in \mathbb{N}$ with $k = \lceil \log m \rceil$ bits by using normal binary coding. If the upper limit m is not a power of two, some code words remain unused. For that case we can use *adjusted binary codes*, which use binary codes with $k-1$ bits if $i < 2^k - m$. If the coded integer is greater than or equal to this limit, the number $i + 2^k - m$ is coded using k bits. Note that this coding increases coding and decoding time. For an example of such a coding see table 1.3. At the decoder side, one has to read $k-1$ bits first and afterwards eventually the last bit. These codes are also commonly called *phased binary codes*. In order to construct more elaborate codes with adjusted binary codes we denote this coding by β_m.

1.3.6 Golomb Codes

Golomb codes are parameterized by a natural number m. The code is constructed in the following way

$$g_m(n) \quad = \quad \alpha(n/m) \cdot \beta_m(n \bmod m) \text{ for } n \in \mathbb{N}. \tag{1.44}$$

The special case when m is a power of two has been examined by Rice. Note that in this case no phasing is required for the binary coding. It has been shown that Golomb codes can be adjusted to yield optimal prefix codes for given exponential distributions [GV75]. For some examples of a Golomb code see table 1.3.

1.4 Approximation with the Matching Pursuit Algorithm

A problem we will encounter in the next chapters (especially section 3.2.1 on page 39), is that we have to approximate a given vector by linear combinations of vectors taken from a set of vectors. Since our goal is data compression, we have to approximate a vector with as few vectors as possible and as accurately as possible. Obviously these two requirements have to be balanced.

[11] The Fibonacci codes are shorter up to $n = 514.228$ [HL87] than the codeword lengths of the code δ.

decimal	β_5	g_5
0	00	0.00
1	01	0.01
2	10	0.10
3	110	0.110
4	111	0.111
5	–	10.00
6	–	10.01
7	–	10.10

Table 1.3: Some adjusted binary and Golomb codewords.

The approximation with matching pursuits (MP) was introduced by Mallat and Zhang in [MZ93]. In this technique, an approximation is constructed step by step using a greedy strategy. Let us first explain the idea of this technique in an informal manner.

The first approximation of a vector v is constructed by selecting the best matching vector from a given codebook. The component of this best matching vector is then subtracted from v. The remaining residue is now encoded in the same way as v and the process is continued until a given abortion criterion is fulfilled. Note that the codebook vectors do not need to be orthogonal. This approximation technique is also called *projection pursuit* in statistics.

1.4.1 The Standard Matching Pursuit Approximation

Let $p, n \in \mathbb{N}$ and $\mathcal{D} = \{g_0, ..., g_{p-1}\}$ with $g_i \in \mathbb{R}^n (i \in \{0, ..., p-1\})$ be a set of $p \geq n$ non-zero vectors. The set $\mathcal{D}$ is often called *dictionary*, *codebook* or *domain pool*. With $\text{span}(\mathcal{D})$ we denote the set of all linear combinations of vectors in $\mathcal{D}$. In order to make the following calculations a bit easier we assume that, without loss of generality, each vector in the dictionary has unit Euclidean norm. If $\text{span}(\mathcal{D}) = \mathbb{R}^n$ then $\mathcal{D}$ contains a set of n linear independent vectors. The *matching pursuit algorithm* begins by projecting v on a dictionary vector g_{i_0} and computing the residue Rv by

$$v = \langle v, g_{i_0} \rangle g_{i_0} + Rv. \tag{1.45}$$

Since Rv is orthogonal to g_{i_0}, the following equation holds:

$$||v||^2 = |\langle v, g_{i_0} \rangle|^2 + ||Rv||^2. \tag{1.46}$$

From that equation one can see that we have to choose g_{i_0} such that $|\langle v, g_{i_0} \rangle|$ is maximal, since $||Rv||$ has to be minimized.

The next iteration of the procedure continues with Rv instead of v, such that the following iteration is computed:

1. $R^0 v = v$.

2. The next residues are computed by

$$\begin{aligned} R^m v &= \langle R^m v, g_{i_m} \rangle g_{i_m} + R^{m+1} v, \\ ||R^m v||^2 &= |\langle R^m v, g_{i_m} \rangle|^2 + ||R^{m+1} v||^2 \end{aligned} \tag{1.47}$$

   and choosing g_{i_m} in the way that $\langle R^m v, g_{i_m} \rangle$ is maximized.

Summing up the last equation in m from 0 to a stopping index $M-1$ yields

$$v = \underbrace{\sum_{m=0}^{M-1} \langle R^m v g_{i_m} \rangle g_{i_m}}_{\text{approximation for } v} + \underbrace{R^M v}_{\text{remaining approximation error}} . \tag{1.48}$$

1.4.2 Improvements of the Greedy Method

In the last section, we have introduced the standard matching pursuit algorithm with the greedy strategy. But as in many areas in computer science, this strategy is not optimal[12]. In the case of the matching pursuit algorithm, the suboptimal case can be observed when the vectors in the codebook are not orthogonal. If we take m elements in the linear combination, the best strategy is to evaluate all

$$\binom{|\mathcal{D}|}{m} \tag{1.49}$$

combination of indices. Since in practice the evaluation of such an approximation is infeasible, many researchers have tried to get suboptimal solutions, which are better than the standard greedy method. Despite all problems, it could be interesting to limit the size of the MP vectors and test all possible combinations of indices.

Second Chance Matching Pursuit

A well-known variant of the standard matching pursuit method is the *second chance variant*. In this variant the approximation algorithm is run twice: the first run is the normal matching pursuit algorithm as described in the last section. After that the best matching vector of the first run is removed from the dictionary and the matching pursuit algorithm is run with that modified dictionary. Afterwards the best of the two alternative approximations is selected. [Haf99] reports that this variant improves the image quality up to 0.3 dB PSNR at the same compression factor. Our implementation also achieved a significant improvement. Note that the first run of the second chance can be computed in the first run of the first chance.

[12] Note that the determination of the optimal solution is an NP hard problem.

Orthogonal Matching Pursuit

In this variant the dictionary vectors are orthogonalized in the same manner as in the Gram Schmidt orthogonalization procedure (see appendix J.2 on page 210). This method is reported to achieve a significant gain over the standard algorithm.

Remaining Problems

Matching pursuit approximation is often utilized in data compression. Since it has proved to be a powerful tool for constructing linear combinations, we decided to use this technique. As we will see in the next chapters, the following additional problems have to be solved:

- the vector v may be not contained in $\text{span}(\mathcal{D})$,
- the achieved coefficients have to be quantized,
- the storage cost of the coefficients may differ.

One of the most severe drawbacks is that a huge number of scalar products has to be calculated. In section 4.3 on page 68 we show how these techniques can be further refined to satisfy the requirements for state of the art WFA coding.

Chapter 2

Entropy Coding

The efficient lossless coding of data is the basis of a high performance image and video coding system. Since the speed, memory consumption and compression efficiency of the entropy coder influences all parts of our implementation of the WFA encoder, we pay special attention to this type of coding. In this chapter, we introduce a highly efficient statistical encoder and present some types of models from which we have to choose the best suited for our project. In chapter 3 on page 35 we argue how the statistical model has to be built to support WFA coding in a highly efficient way.

2.1 The Arithmetic Encoder

The arithmetic coder eliminates the drawback of prefix coders that each symbol has to be encoded by an integer number of bits. This enhancement is obtained by using a totally different way of coding. Another benefit of this coder is the clear separation of statistical model and coder. This benefit allows the design of more flexible models and an easy adaptation of the coder to those models. However, these enhancements are achieved by a higher implementation expense and a running time generally slower by a constant factor.

2.1.1 The Idealized Algorithm

In order to introduce the arithmetic encoder, we assume that the input alphabet is finite. The codec[1] describes the encoded message by an element of a half open interval which is a subinterval of the interval $[0, 1)$. When the message gets longer during the coding process, the output interval narrows at each encoded symbol by a factor dependent on the probability of this symbol. For this reason, the number of bits which have to be used to encode that interval increases. Symbols having a higher probability narrow the output interval less than symbols with lower probability and thus append less bits to the output.

[1] The word *codec* is the merging of the words encoder and decoder.

Each symbol e of the input alphabet will be assigned a half-open interval $I(e)$ with a length equal to the probability of the occurrence of that symbol: $|I(e)| = p_e$. These intervals form a disjoint covering of the interval $[0, 1)$. After each coded symbol, the output interval is narrowed by a factor equal to the probability of the occurrence of that symbol. The initial interval where no symbols are encoded is the interval $[0, 1)$. In order to mark the end of the message, a special EOF[2] symbol could be encoded. Another alternative is to transmit the length of the message at the beginning. Note that not both limits have to be transmitted but only one arbitrary number in the final coding interval (for example the lower bound). In listings 2.1 and 2.2, pseudo implementations of the arithmetic encoder and decoder in its pure form are given.

```
/**
* encodes a string of symbols.
*
*@return a number representing the input message.
**/
float encode()
//high and low denote upper and lower limit of the current
//encoding interval.
//high(e) and low(e) denote upper and lower limit of the
//encoding interval of the symbol e.
  {
  float low=0;
  float high=1;
  while (not EOF)
    {
    e=next input symbol;
    range=high-low;
    high=low+range*high(e);
    low=low+range*low(e);
    }//end while
  return low;
  }//end encode
```

Listing 2.1: Arithmetic encoder.

```
/**
* decodes a real number.
*
*@param z is the number to be decoded.
**/
void decode (float z)
//high(e) and low(e) denote upper and lower limit of the
//encoding interval of the symbol e.
  {
  while (not EOF)
    {
    find the symbol e such that z ∈ I(e);
    output e;
    range=high(e)-low(e);
```

[2] EOF $\simeq$ end of file.

```
    z-=low(e);
    z/=range;
    }//end while
  }//end decode
```

Listing 2.2: Arithmetic decoder.

Example: Assume that the input alphabet is $\{a, b, c\}$. The assigned probabilities are $p_a = \frac{1}{2}$ and $p_b = p_c = \frac{1}{4}$. The assigned intervals could then be determined as shown in table 2.1.

symbol	probability	interval
a	$\frac{1}{2}$	$\left[0, \frac{1}{2}\right)$
b	$\frac{1}{4}$	$\left[\frac{1}{2}, \frac{3}{4}\right)$
c	$\frac{1}{4}$	$\left[\frac{3}{4}, 1\right)$

Table 2.1: Assigning intervals to probabilities.

For the following encoding procedure we refer to Figure 2.1, in which the message *aac* shall be transmitted. After reading the symbol a, the initial interval $[0, 1)$ is narrowed to $\left[0, \frac{1}{2}\right)$. The encoding of the second a narrows the message interval further to $\left[0, \frac{1}{4}\right)$ and the input of the symbol c finally creates the output interval $\left[\frac{3}{16}, \frac{1}{4}\right)$.

The decoder receives an arbitrary number lying in the interval $\left[\frac{3}{16}, \frac{1}{4}\right)$ (for example $\frac{3}{16}$) and can thus reconstruct that the first symbol of the received message is the symbol a, as the received number lies in the interval $\left[0, \frac{1}{2}\right)$. After that the reverse of the first coding step is applied to that number and the next symbol is decoded until the end of the message is reached.

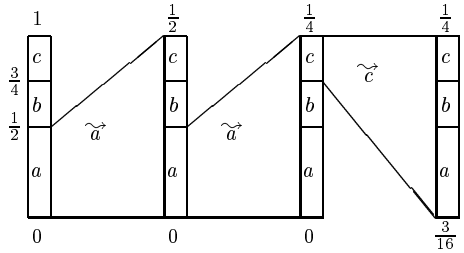


Figure 2.1: Coding of *aac* with arithmetic coding. The left interval is the initial interval, the next intervals are obtained by successively coding the message.

■

Implementation of the Arithmetic Codec

At first glance, the technique of arithmetic coding seems to be impracticable for an implementation as floating point operations seem to be needed. But as we will see, the required operations can be performed using integer arithmetic. Another problem is that we seemingly have to operate on a number with all its digits which in practice can be some millions. In the following implementation proposal, equal digits[3] is pushed out to the left of the window and will not be calculated on in the following operations. This can be done because the next interval is always contained in the current interval and so the digits pushed out will not be changed.

The required operations are explained without loss of generality for the case of decimal numbers. At the beginning of the coding operation the two limits are set to 0 and 0.999... respectively. In order to operate on these limits with integer arithmetic, we only store the digits after the decimal point. Since we can only handle a finite number of digits, we interpret the numbers as a finite window to an infinite precision number. In case that the windows are displaced to the right, one has to append 0s to the lower bound and 9s to the upper bound. If the left (highest) digits of the windows are equal, the windows have to be displaced to the right and the next digits have to be filled in as mentioned before. The process of window displacement is illustrated in Figure 2.2.

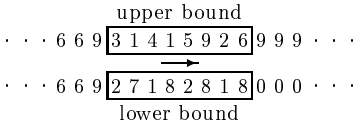


Figure 2.2: Windows to numbers with infinite precision.

Handling of Underflow

A problem with this implementation occurs if the limits approach and the highest digits of these limits differ. In this case, when the highest digits differ only by 1, the encoder has to test if the second highest digit of the lower bound is a 9 and the second highest digit of the upper bound is a 0 (see Figure 2.3). In order to solve this problem, the second highest digits of the two registers have to be eliminated by pushing the following digits to the left but retaining the highest digit. The execution of this operation has to be transmitted separately to the receiver. For more information about arithmetic coding see [Nel92, WNC87]. Results of this coder are given in [Pin90] and in this thesis.

The implementation of the arithmetic codec used in AutoPic is based on the codec presented in [WNC87]. Changings were made especially concerning the statistical model. In order to achieve the highest flexibility, AutoPic uses a

[3]The numbers are assumed to be in Motorola notation meaning that the most significant digits are to the left.

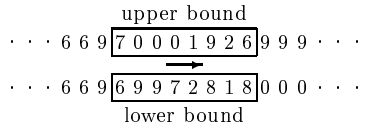


Figure 2.3: The underflow problem of the arithmetic codec.

windowing technique which can be adapted to non-stationary distributions and can also roll back symbols to support the required backtracking operations. Furthermore, the end-of-file symbol was removed from the codec.

Note: Let us remark that a *quasi arithmetic codec* which needs only integer shifts instead of multiplications is also found in the literature [RM89]. However, this coder has the drawback of needing the probabilities in the form $2^j/2^k$ with $j, k \in \mathbb{N}\backslash\{0\}$. Because the arithmetic encoder has only a minor influence on the running time of the WFA algorithm, we decided not to use the quasi arithmetic encoder and thus obtain optimal compression results with small running time punishment. ■

2.2 Statistical Models

A data compression scheme can normally (roughly) be divided into two parts: the *model*[4] and the *encoder*. The model describes the (more or less well) estimated[5] probabilities of the input symbols which the encoder uses to write to the output stream (see Figure 2.4). The problem of encoding is already solved, while the statistical models are still under research.

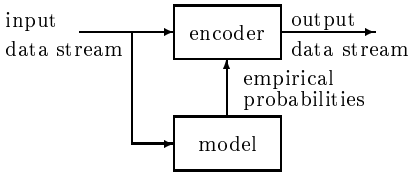


Figure 2.4: Structure of a data compression scheme.

The necessity of predicting the probability of a symbol is of enormous importance for a statistically-based compression scheme. One form of redundancy in a text is the dependency of a symbol from one or more of its predecessors. For example, we consider texts using the German language. In this language, the probability is very high that after the symbol q the symbol u will occur. For this reason, we introduce *higher models*, which estimate the probability of

[4] Note that in the topic of data compression, the term model is used for an implementation of the statistical term probability distribution. In this context, we adapt it to data compression literature.

[5] It is assumed that the exact probabilities are not known.

a symbol's occurrence dependent on a finite number of predecessors. The order of these models denotes the number of considered predecessors.

Models of higher order have the benefit that the probability of the occurrence of a symbol can be estimated more precisely in many cases, but this benefit is obtained by higher storage cost. This cost grows exponentially depending on the order of the model.

2.2.1 Block Coding

Another commonly used type of model does not encode single symbols but combines several symbols to a *block* and thus creates a new alphabet. Such a process is called *alphabet augmentation*. We write the combined components as $e_0, \ldots, e_{n-1}$ and the resulting n-tuple as

$$\boldsymbol{e} = (e_0, \ldots, e_{n-1}). \tag{2.1}$$

The *entropy of order* n (which is also called *block entropy*) is defined on these tuples as

$$H_n = -\sum_{\boldsymbol{e} \in E^n} p_{\boldsymbol{e}} \log p_{\boldsymbol{e}} \ . \tag{2.2}$$

It can be shown that the inequality

$$H_n \leq nH \text{ with } n \geq 2 \tag{2.3}$$

holds where equality is reached iff the components are statistically independent [NH88]. With this method especially prefix encoders can be improved (respectively compression performance). A problem with this type of encoding is the exponential growth of the alphabet sizes.

2.2.2 Predictive Encoding

In the following paragraph, we assume that the usual binary arithmetic operators + and − are defined on the input alphabet, which is for example the case if $E = \mathbb{Z}$.

The method of *predictive encoding* is closely related to conditional encoding. Using conditional encoding, the symbol e is assigned a probability depending on the $n-1$ symbols that were encoded previously. In predictive encoding, the next symbol e is predicted with information extracted from the last $n-1$ symbols. Then the difference $e - \tilde{e}$ is encoded instead of e, where $\tilde{e}$ denotes the *prediction* of e. The decoder uses the same predictor and adds the prediction to the received symbol. The performance of a predictive encoder depends strongly on the effectiveness of the predictor. In practice, the encoder needs to transmit less bits since the entropy of the differences is smaller than the entropy of the original sequence if we assume a "good" predictor and certain statistical properties of the sequences. This benefit is exploited for example in DPCM[6] which is explained in [Jai89, Kat94].

[6] DPCM $\simeq$ differential pulse code modulation.

2.3 Non-Context Models

In this and the next section, we present some ways to implement models. We first examine non-context models followed by context models which often use non-context models as a basis.

2.3.1 Static and Adaptive Models

A *static model* is a model which does not change the assigned probabilities of the input symbols during the encoding process, in contrast to *adaptive models*.

A static model either obtains its probabilities from a fixed table or from a spot test of the encoded data stream. In the latter case, the transmission of the statistics is mandatory, thus worsening the compression performance. Adaptive models start with a predefined probability distribution. We also have to assign a non-vanishing probability to symbols not yet transmitted. Because the relative frequencies of such symbols are 0, we are confronted with a problem known as the *zero frequency problem*. When operating with adaptive models, we have to make sure that first the codeword of a given symbol is transmitted and after that we can update the model, because the decoder otherwise has no way to keep its model up to date in order to work with the same model. A commonly used technique for implementing an adaptive model is to use an array F (frequency) with $|E|$ components, which are all set to the initial value 1. After encoding the symbol e_i, the i^{th} component of the array is incremented. In order to estimate the probability of the symbol e_i, the *relative frequency*

$$p_i = \frac{F_i}{\sum_{j=0}^{|E|-1} F_j} \text{ for } i = 0, \ldots, |E| - 1 \tag{2.4}$$

is used.

For the case of non-stationary statistics, the estimated probabilities eventually cannot adapt quickly if a great number of symbols have been encoded. A solution to this problem is the recalculation of the probabilities after a specified number of encoding steps. As a criterion for recalculating the frequency array, the codec could test if the highest frequency exceeds a given limit. The recalculation can be performed by the algorithm in listing 2.3 [Pin90]. Note that the zero frequency problem is avoided by initializing the array F with the values $F_i = 1$. A faster adaptation of the model is obtained by adding a value greater than one to an array component when updating the model. This parameter is called AdaptationSpeed in AutoPic.

```
/**
* recalculates the array of relative frequencies,
* thus omitting overflow and ensuring fast adaptation of
* this model.
**/
void recalculate()
  {
```

```
final c=...;//c ≥ 1
for (i=0; i < |E|; i++)
  F_i = (F_i + c)/2;
}//end recalculate
```

Listing 2.3: Recalculation of an adaptive source model.

2.3.2 Window Models

In order to exploit non-stationary data statistics, a well-known technique is to use a *windowed model*. This kind of model utilizes a window of the last k symbols defining a sample of the data to be encoded. The statistics of this spot check may be used to estimate probabilities for the entropy encoder. For an illustration of this model see Figure 2.5. More information about this coding technique can be found in [FG89].

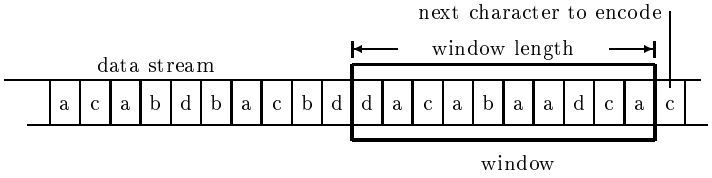


Figure 2.5: A windowed model.

In our implementation of the WFA encoder (AutoPic) we use an even more flexible windowing technique explained in section 4.2.1 on page 67.

2.4 Context Modeling

In this section we examine some kinds of higher order models.

2.4.1 Finite Context Modeling

The technique of *conditional coding* assumes that the components $e_0, \ldots, e_{n-2}$ of the n-tuple $\boldsymbol{e}$ are already known. The component e_{n-1} can then be coded more efficiently by using this information under the assumption that statistical dependency exists between these symbols. Therefore, the technique uses the conditional probability distribution

$$P(e_{n-1} \mid e_0 \ldots e_{n-2}). \tag{2.5}$$

The implementation of conditional models is called *finite context modeling*. This type of model switches to a suitable non-context model, according to a finite tuple of predecessors. We implemented this model in AutoPic since it is reported to achieve best results [BCW90]. Section 4.2.1 on page 65 shows how we refined this technique in AutoPic.

2.4.2 Markov Modeling

Another method of utilizing the context of a symbol is to use state-based models. This method works by processing the input sequence with a finite state automaton. The states of this automaton are associated with non-context models. If a given symbol is encoded, the current non-context model is used to predict probabilities and switch to the next state after coding of the symbol. Therefore, an important question is which structure the underlying automaton should have. There are only a few applications where the automaton can be predefined. For this reason, *dynamic Markov coding* has been established in this area. In this kind of modeling, an initial automaton is created which may be as simple as in Figure 2.6. This initial automaton can be adapted to current probabilities by duplication of states. A state is duplicated if it has more than one incoming transitions and one transition is visited frequently. In this case, the associated model is copied and assigned to a new state. This state's outgoing transitions are also copied from the original state. For an example of the splitting of a state see Figure 2.7.

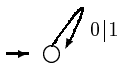


Figure 2.6: A simple Markov model.

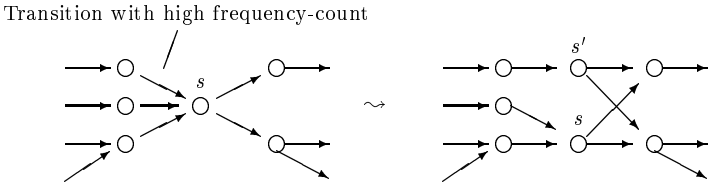


Figure 2.7: Splitting of a state.

Markov models have the benefit of low memory consumption and fast switching of contexts. The main reason not to implement these models in AutoPic is that they are a special case of finite context modeling [How93]. These models could be interesting for further research in WFA coding. For more information about Markov models see [BCW90].

2.4.3 Model Blending

Another well-known technique is the blending of different models. For example, different finite context models could be maintained. As coding starts, the codec assigns the highest weight to the model of order 0, and as coding proceeds assigns higher weight to models of higher order [BCW90].

2.4.4 Further Techniques

The topic of statistical modeling is an area under current research. There are further techniques we do not mention here like grammar models which are used for the compression of data as program source code [BCW90].

Chapter 3

Techniques for WFA Construction

In this chapter, we introduce fractal image coding with weighted finite automata and several methods for WFA construction. We only examine the coding of gray valued and color images. The coding of binary images is described in [CV97]. Most of the algorithms described can easily be adapted to lower dimensional data (for example sound files) or higher dimensional data (for example digital movie files). But these extensions do not necessarily achieve optimal results, which is the reason why techniques like motion compensation are used in video coding. Details about the Hausdorff dimension of WFA-generated images can be found in [MS94].

3.1 Basic Techniques for Image Partitioning

The technique of WFA coding is based on partitioning an image into sub-images. Such an image partitioning was introduced in a scheme named quadtree image compression. It can be refined using bintrees and HV bintrees. In the following sections, we assume that the input image is a square matrix with a side length of 2^k, $k \in \mathbb{N}$. Note that this assumption does not restrict the coding to this subset of images, since images of arbitrary size can be embedded into the next larger square image having side lengths with powers of two. This square image may be encoded by the WFA encoder. The WFA decoder can decode the surrounding square image and afterwards cut the embedded image off this decoded image.

3.1.1 Quadtrees

The original image partitioning in the WFA technique uses quadtrees [CK93]. The *quadtree* is a hierarchical data structure based on the well known principle of recursive partitioning (divide and conquer principle). Such hierarchical data structures are particularly useful because of their ability to cut off certain parts

of the original data. We examine the *region quadtree*, which divides the input image recursively into four quadrants of equal size. The quadtree is therefore a tree of degree four, with each inner node having four children. The division process[1] continues until a given criterion of simplicity is fulfilled. In the original quadtree image compression algorithm, the input image is divided until all pixels in a given quadrant have equal gray values.

For an illustration of this procedure see Figure 3.1. In this drawing, the nodes are traversed in the order represented by the tree on the right. The leaves store the gray values "B" (black) and "W" (white). The inner nodes are labeled with "G" (gray) to indicate that they have to be partitioned further.

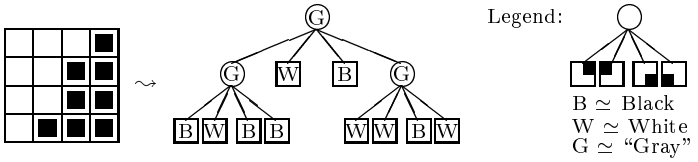


Figure 3.1: Region quadtree. The ordering of the quadrants and the node labels are indicated in the legend on the right.

A quadtree is suited for image compression because large areas can often be stored in a single node. The quadtree can also be used for lossy image coding if the recursive division is stopped if a special "homogeneity property" is fulfilled by the addressed quadrant. The associated node of the quadtree could store a gray value which minimizes a given distortion function. The average value is often used for this task.

Efficient Storage of the Quadtree Structure

The efficient storage of the quadtree is an important task in WFA coding since for efficient image coding, each part has to be stored with the minimum number of bits.

If there is an ordering of the nodes in the tree (as for example introduced by an ordering of the quadrants), only niveaus of the leaves in this ordering have to be stored. For the quadtree in Figure 3.1, this would be the sequence "2 2 2 2 1 1 2 2 2 2". For a more efficient encoding, we could also store these levels in a differential way. The coding of the quadtree referenced above would be "2 0 0 0 −1 0 +1 0 0 0", where the first value was left unchanged.

Another possibility for storing the quadtree is the coding of the division and merging operations. The quadtree in Figure 3.1 could be coded as "↓ ↓ B W

[1]Some authors distinguish between the terms *quadtree* and *region quadtree*. The former partitions the image up to the pixel level whereas the latter stops the division process if the image segments assigned to the children fulfill the homogeneity property, thus collecting image segments in a single node. Since in data compression we are only interested in region quadtrees, we drop the prefix and consider only region quadtrees without mentioning this fact explicitly.

B B W B ↓ W W B W", where the sign ↓ signals the descent in the tree and the quadrants are arranged in the ordering defined in the legend of Figure 3.1. This storage is similar to a DFS traversal where the sign ↓ indicates that a further recursive call has to be performed. Besides the partitioning information of the quadtree, there is also additional information to store, such as the average brightness of the image quadrants associated with the leaves. The storage of such information can be done in any traversal order.

Finally, we can draw the conclusion that the quadtree structure can be stored using four bits per node if in each node each of the four branches is marked with a bit determining whether that quadrant is to split any further or not. This value can be improved with appropriate entropy coding since the probability for dividing an image segment is much higher at the top of the tree than at the bottom.

Note that under the assumption that the receiver of the message knows the minimal and maximal division depth of the tree, the number of bits required for transmitting quadtrees can be further reduced. For further details about the storage of quadtrees see [Far90].

3.1.2 Bintrees

An important variant of quadtrees are the *bintrees*, where at each node the image is subdivided only in two parts instead of four. Horizontal and vertical subdivisions are applied alternately in the niveau of the tree. If only even numbered leaf niveaus were permitted, the bintree division would be the same as that of quadtrees. Using bintrees the division process can be stopped earlier, so the associated parts of the image contain more pixels and can therefore be coded more efficiently. For this reason, we utilized bintrees in our implementation of the WFA image coder. We observe that the bintree structure can be stored using two bits per node without entropy coding in the same way as explained for quadtrees. For an illustration of a bintree see Figures 3.2 and 3.3. Let us remark that the decision, whether the first split is made horizontally or vertically, is arbitrary.

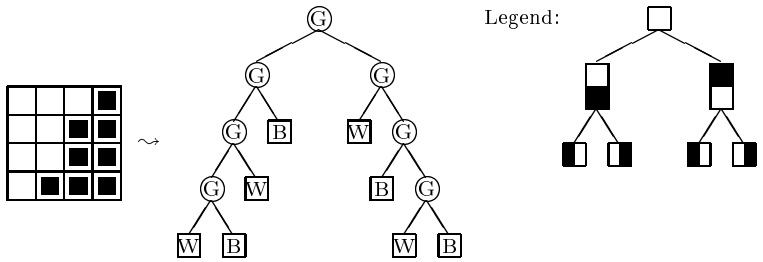


Figure 3.2: Region bintree. The ordering of the segments and the node labels are indicated in the legend on the right.

Figure 3.3: Bintree partitioning of the image Lenna.

3.2 Weighted Finite Automata

This section introduces the main topic of the thesis: the weighted finite automaton (WFA). Before we examine image coding with WFAs in detail, we first explain the idea of this coding method. The basic WFA coding method has a close relationship to image compression with quadtrees. Instead of storing gray values in the leaves, coefficients[2] are stored if a further subdivision is not profitable. These coefficients may contain both linear combinations of a hard-wired initial basis but also linear combinations of other nodes in the WFA tree, which yields the fractal character of that coding method.

There is also a strong relationship to IFS coding, which is discussed in appendix H on page 201. Because of the relationship between these coding methods, many ideas can be exchanged. We have also noticed that some extensions of the IFS method are a special case of the WFA algorithm. The main drawback of the IFS method compared to the WFA method is the need to encode contractions.

3.2.1 Definition of a WFA

Image coding with WFAs was introduced by K. Culik and J. Kari [CK93]. In order to simplify the approach, we introduce this kind of structure with quadtrees. In a similar manner, WFAs may be defined using bintrees and even HV bintrees.

WFA coding can be seen as an enhancement of quadtree or bintree coding. Instead of storing a gray value in the nodes, the coder stores coefficients of linear combinations in the tree nodes. In order to examine images with methods of formal languages, we interpret an image as a function over the set $[0,1] \times [0,1]$ and associate words over the alphabet $\Sigma = \{0,1,2,3\}$ with 2-dimensional image segments. Let us first introduce basic notations of formal languages: let Σ be an alphabet, then the set Σ^n denotes all words (strings) of length n and $\Sigma^* := \bigcup_{n \geq 0} \Sigma^n$. An image quadrant with size $2^{-n} \times 2^{-n}$ is associated with a word in the set Σ^n by interpreting the quadtree as a trie. Each symbol of Σ matches a quadrant of a square image and paths are constructed by concatenating the matching symbols recursively (see Figure 3.4). The coding is done by using a structure which can formally be interpreted as a weighted (nondeterministic) finite automaton.

We call a function $f : \Sigma^* \to \mathbb{R}$ a *multi-resolution image*. To assign gray values to the image segments, we consider only multi-resolution images with the following property:

$$f(\sigma) = \frac{1}{|\Sigma|} \sum_{q \in \Sigma} f(\sigma q) \quad (\sigma \in \Sigma^*). \tag{3.1}$$

[2] We denote by the term coefficients a representation of a given approximation to a vector. In our implementation, we use matching pursuits (see section 1.4 on page 20) since this representation of coefficients fits excellently to our application.

One can interpret such a function as a quadtree for $|\Sigma| = 4$ (for bintree, $|\Sigma| = 2$) with infinite depth, where each node incorporates the average brightness of its children. A function which satisfies equation 3.1 is called *conservative* (or *average preserving*, [CK93]).

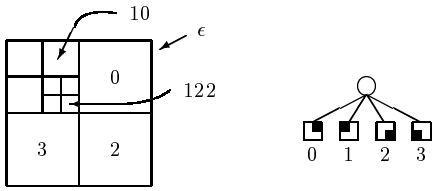


Figure 3.4: Assignment of paths to image segments.

Now we interpret the set of functions $f : \Sigma^* \to \mathbb{R}$ as a real vector space V with the following definitions of the required operations:

$$(f_0 + f_1)(\sigma) = f_0(\sigma) + f_1(\sigma) \quad (f_0, f_1 \in V, \sigma \in \Sigma^*), \tag{3.2}$$

$$(cf)(\sigma) = cf(\sigma) \quad (f \in V, c \in \mathbb{R}, \sigma \in \Sigma^*). \tag{3.3}$$

It is obvious that the set of conservative functions $\mathcal{D}$ is a linear subspace of the vector space V.

Definition 3.2.1 *A* weighted finite automaton *(WFA)* $\mathcal{A} = (S, \Sigma, W, I, F)$ *is specified by*

1. *a finite set of* states $S = \{s_0, \ldots, s_{|S|-1}\}$,
2. *a finite* alphabet $\Sigma = \{a_0, ..., a_{|\Sigma|-1}\}$,
3. *a* weight function $W : S \times \Sigma \times S \to \mathbb{R}$,
4. *an* initial distribution $I : S \to \mathbb{R}$,
5. *a* final distribution $F : S \to \mathbb{R}$.

For our purposes, we either choose $\Sigma = \{0, 1, 2, 3\}$ (quadtree) or $\Sigma = \{0, 1\}$ (bintree). If nothing else is stated, we will regard the quadtree case. We write $(W_a)_{i,j} = W(s_i, a, s_j)$ with $a \in \Sigma$, $s_i, s_j \in S$ and say that (s_i, a, s_j) is an *edge* of the weighted finite automaton iff $(W_a)_{i,j} \neq 0$. In order to perform calculations with the WFA $\mathcal{A}$, we will consider W_a as a real valued $|S| \times |S|$-matrix, I and F as $|S|$-dimensional (column) vectors.

We define for each state $s_i \in S$ a multi-resolution image Φ_i (*state image*) by

$$\Phi_i(a_0 \ldots a_{k-1}) = e_i^t (W_{a_0} \ldots W_{a_{k-1}} F) \tag{3.4}$$

where e_i is the i^{th} canonical unit vector ($(e_i)_j = \delta_{i,j}$ and $\delta_{i,j}$ is the Kronecker symbol[3]).

[3] $\delta_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{else.} \end{cases}$

We define the multi-resolution image given by the WFA $\mathcal{A} = (S, \Sigma, W, I, F)$ as a linear combination of the state images Φ_i

$$f_{\mathcal{A}} = \sum_{j=0}^{|S|-1} I_j \Phi_j, \tag{3.5}$$

or expressed in another way

$$f_{\mathcal{A}}(a_0 \dots a_{k-1}) = I^t (W_{a_0} \dots W_{a_{k-1}} F) \tag{3.6}$$

for $k \in \mathbb{N}$ and $a_0 \dots a_{k-1} \in \Sigma^*$.

Note: Equation 3.6 defines a (slow) WFA decoding algorithm, since we can calculate the associated brightness for each pixel in the target image with this formula. Since there are better alternatives, we will not investigate this algorithm any further. ■

A WFA is called *conservative* iff the associated multi-resolution image is conservative. If F is an eigenvector with eigenvalue $|\Sigma|$ of the matrix $\sum_{q\in\Sigma} W_q$, we can conclude that

$$\left(\sum_{q\in\Sigma} W_q\right) F = |\Sigma| F \tag{3.7}$$

$$\Rightarrow \quad \left(\sum_{q\in\Sigma} W_q F\right) = |\Sigma| F \tag{3.8}$$

$$\Rightarrow \quad \sum_{q\in\Sigma} I^t W_{a_0} \dots W_{a_{n-1}} W_q F = |\Sigma| I^t W_{a_0} \dots W_{a_{n-1}} F \tag{3.9}$$

$$\Rightarrow \quad \sum_{q\in\Sigma} f_{\mathcal{A}}(\sigma q) = |\Sigma| f_{\mathcal{A}}(\sigma) \tag{3.10}$$

for arbitrary $\sigma = a_0, \dots, a_{n-1} \in \Sigma^*$. Equation 3.10 expresses that the multi-resolution image $f_{\mathcal{A}}$ is conservative. Note that this is a rather theoretical result since the conservativeness can be checked for the WFA construction algorithms by induction (see section 3.6 on page 54).

From now on, we will only consider conservative WFAs and functions. There are more general structures than WFAs such as generalized weighted finite automata which are explained in [CR96] or m-WFAs which are described in [Haf95]. A lot of material can be found in [Kar].

3.3 The First WFA Coding Algorithm Introduced by Culik and Kari

In order to construct a WFA for a given image, we have to construct the vectors I, F and the matrices defined by W_a ($a \in \Sigma$). Since the vector I depends on

the encoding algorithm, we specify this vector for each algorithm. From the definition of the state images (equation 3.4) we can infer by setting $k = 0$

$$\Phi_i(\epsilon) = e_i^t F = F_i \tag{3.11}$$

that F_i equals the average brightness of the state image Φ_i. We examine equation 3.4 further to see how the matrices W are constructed,

$$\begin{aligned}
\Phi_i(a_0 a_1 \dots a_{k-1}) &= e_i^t W_{a_0} W_{a_1} \dots W_{a_{k-1}} F && (3.12)\\
&= \left(e_i^t W_{a_0}\right)\left(W_{a_1} \dots W_{a_{k-1}} F\right) && (3.13)\\
&= \left(e_i^t W_{a_0}\right)\begin{pmatrix} e_0^t W_{a_1} \dots W_{a_{k-1}} F \\ \vdots \\ e_{|S|-1}^t W_{a_1} \dots W_{a_{k-1}} F \end{pmatrix} && (3.14)\\
&= \left((W_{a_0})_{i,0}, \dots, (W_{a_0})_{i,|S|-1}\right)\begin{pmatrix} \Phi_0(a_1 \dots a_{k-1}) \\ \vdots \\ \Phi_{|S|-1}(a_1 \dots a_{k-1}) \end{pmatrix} \\
&= \sum_{j=0}^{|S|-1} (W_{a_0})_{i,j} \Phi_j(a_1 \dots a_{k-1}). && (3.15)
\end{aligned}$$

In this equation, we see that the quadrants of the state images are linear combinations of other state images. With these preliminaries, we can define a WFA construction algorithm designed for lossless image coding.

In order to ease the next explanations, we introduce the following notation for conservative functions $\Psi \in \mathcal{D}$:

$$\Psi_{\sigma_0}(\sigma_1) = \Psi(\sigma_0\, \sigma_1) \quad (\sigma_0, \sigma_1 \in \Sigma^*) \tag{3.16}$$

meaning that we zoom Ψ into the quadrant $\sigma_0 \in \Sigma^*$.

The method[4] in listing 3.1 first creates a state representing the whole image and afterwards tries to approximate the quadrants of a given state image by other state images. If that does not work, a new state is generated for that quadrant. For convenience we will call pointers created by the second alternative *partition pointers*.

```
/**
* constructs a WFA which approximates a conservative function.
*
*@param Φ is the function to be represented.
*@return a WFA that represents Φ.
**/
WFA operateImage(conservative Function Φ);
```

[4]Note that listing 3.1 is in a strict sense no algorithm since a multi-resolution image is generally not representable in a finite way. Due to usual notation we will write algorithm instead of mathematical method.

```
{
create a state which represents the whole input image;
while (there are unprocessed states)
  {
  choose an unprocessed state;
  operate on all quadrants of the assigned state image:
    {
    if (the current quadrant can be linearly combined by existing states)
      store these coefficients;
    else
      {
      create a new state representing this quadrant;
      store a pointer to this state;
      }
    }
  }
I=(1,0,...,0);
return the resulting WFA;
}//end operateImage
```

Listing 3.1: The first WFA coding algorithm.

A multi-resolution image Φ can be represented exactly by a WFA iff the set of functions defined by all quadrants spans a finite dimensional subspace in $\mathbb{R}^{\Sigma^*}$ [CK93]. The WFA generated by the algorithm in listing 3.1 possesses the minimal number of states of all WFAs that generate Φ [CK93]. One can see that the WFA tree given by this algorithm is generated top down in BFS[5] order, meaning that first a node for the whole image is generated and afterwards the nodes for image segments having longer addresses will be created. For this reason, we will call that algorithm top down (TD) WFA coding algorithm. For more details concerning this algorithm see [CK95].

Note: The coefficient vectors in the above algorithm may be interpreted as the rows of the weight matrices W_a for $a \in \Sigma$. So the coefficient vector belonging to the quadrant a in the state s_i is the i^{th} row of the matrix W_a. ■

Example: For an example of this coding process see Figure 3.5, where the image on the top is approximated. Assume that the quadrants are processed[6] in the order 1, 2, 0, 3. The generated automaton has four states and Figure 3.5 shows the state images of these states. The upper state image shows Φ_ϵ (the original image), the three lower state images are (from left to right) Φ_1, Φ_2 and Φ_0.

The created WFA $\mathcal{A} = (S, \Sigma, W, I, F)$ has the following components:

- $S = \{s_0, s_1, s_2, s_3\}$, the state images of these states are shown in Figure 3.5 (top to bottom, left to right),

[5] BFS $\simeq$ breadth first search [CLR91, Sed88].

[6] Note that the order in which the quadrants are processed is arbitrary. We have chosen this processing order because of visualizing reasons.

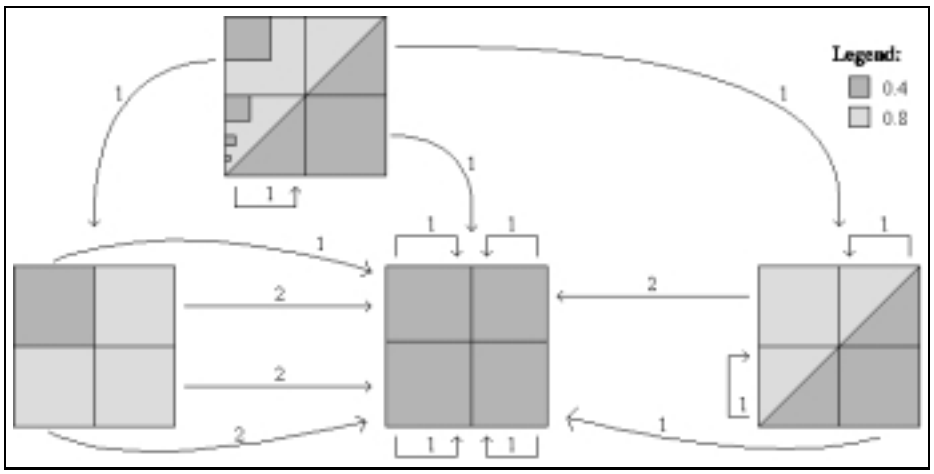


Figure 3.5: Construction of a WFA. The edges are labeled with the corresponding weights and the labels of the edges are defined by the quadrants of the starting points.

- $\Sigma = \{0, 1, 2, 3\}$ since in this example we apply quadtrees for image partitioning,
- $I^t = (1, 0, 0, 0)$ since the original image is the state image of the first state,
- $F^t = (0.5\overline{6}, 0.7, 0.4, 0.6)$ (these are the average brightnesses of the state images),
- and the following weight matrices:

$$W_0 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad W_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 \end{pmatrix},$$
$$W_2 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad W_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

■

Initial States An important observation is that the WFA construction process has not to start with an empty set of states. It is a natural idea to start with a number of predefined states (*initial states*) whose state images may represent patterns which are expected to occur often in the target image. Since these initial states can be hard-wired in the WFA codec, neither the structure nor the average brightnesses of the initial states have to be transmitted. Due to common nomenclature we will call this set of states also *initial basis*.

Optimization of the First Coding Algorithm Despite the fact that the automaton generated by the algorithm above is minimal with respect to the number of generated states [CK93], the automaton is not necessarily minimal with respect to the number of edges. In [Rob95], a modification of this algorithm was implemented with the result that the storage space for automata constructed for natural images were almost always bigger than the original image size. A better approach to image coding is the algorithm described in section 3.5 on page 48.

Note: Let us remark some words about the running time of the WFA construction algorithm. We only determine the maximal number of multiplications performed by linear approximation (calculation of the scalar products). If we assume that we

- use bintree partitioning,
- zoom up all state images to full resolution with p pixels,
- create one state for each image segment defined by the bintree with two pixels, that are (if we sum the number of states in the niveaus from bottom to top)
$$\frac{p}{2} + \frac{p}{4} + \frac{p}{8} + \ldots + 1 = p - 1$$
states,
- approximate each state quadrant with all previously created states. This task requires
$$2 \sum_{i=1}^{p-1} i = (p-1)p$$
scalar products.

Since each scalar product requires p multiplications, we conclude that $p^3 - p^2$ multiplications have to be performed. In case of a 512×512 image we have $p = 262144$ and thus have to perform approximately $1.8 \cdot 10^{16}$ scalar multiplications.

Although the assumption that the image has to be partitioned to such small segments is not realistic, we had to optimize the WFA construction algorithm carefully (see section 7.7 on page 142). ■

Quadtree Coding as a Special Case of WFA Coding Now consider the case that one predefined state (initial state) with constant gray value 1 is added[7] to the WFA and only linear combinations which refer to that state are allowed. In this case the image segments will be partitioned until the gray value in the

[7] A state with constant gray value may be easily constructed as a WFA. We omit the construction of this state since in later sections we will show that this state not even has to be constructed in this manner but can be pre-calculated.

segments are constant. The one (and only) linear coefficient referring to the constant state image has thus the brightness of the constant brightness of the image segment. This process is thus exactly the same as quadtree coding.

3.3.1 Different Representations of the WFA Tree

Let us now take a look at two different representations of the WFA tree. Assume that we are encoding an image segment and have already generated a state for that segment.

- The first way is to approximate the current segment and if that does not work, mark the current state as divided and add pointers from the current state to the new states.
- The second alternative is to approximate the subsegments of the current image segment and if that does not work, store pointers to new states which are to be generated. This representation also has been chosen by Culik et al. [CK93].

Both methods have their benefits and drawbacks. Using the first method, we can cut the WFA tree a level earlier than in the second and the smallest WFA tree contains only one coefficient vector. However, one has to partition the whole segment or has to approximate the whole segment.

With the second method, the choice of approximating or partitioning is in some sense deferred. But here we can choose whether to partition or approximate for each quadrant. We also have the benefit of equal representation since the choice of partitioning can also be represented by a coefficient vector which contains only one element with the index of the newly generated state and 1 as the coefficient. Because WFA trees in our application generally contain thousands of states, we chose the second alternative.

3.4 The Fast WFA Decoding Algorithm

Before we discuss a WFA decoding algorithm, we note that a WFA is resolution independent. In order to decode a WFA we fix the resolution of the output image. In our experiments examining the compression performance of WFAs, we decoded the images at the same resolution as the original images. However, it is also of interest to decode an image at a higher resolution than the original image (fractal interpolation, see section 4.4.13).

In order to decode a WFA at a resolution of $2^n \times 2^n$, we have to calculate the weights of all paths of length n in $\mathcal{A}$ and add these weights to get the gray values of the pixels. This decoding algorithm has a running time of $O(|S|(|\Sigma||S|)^n)$, which is unacceptable for practical purposes. For this reason, we will not consider this decoding algorithm further. Fortunately, there is a more efficient

decoding procedure which is given in listing 3.2 [CK95]. This procedure constructs state images consecutively from low resolutions up to the desired resolution. The algorithm is initialized with state images of resolution 1×1 by setting the pixel brightnesses to the average brightnesses of these states. The next step is the doubling of the resolutions. We construct a quadrant of a state image of resolution $2^{k+1} \times 2^{k+1}$ by adding the state images of resolution $2^k \times 2^k$ according to the linear combinations. This technique is illustrated in Figure 3.6.

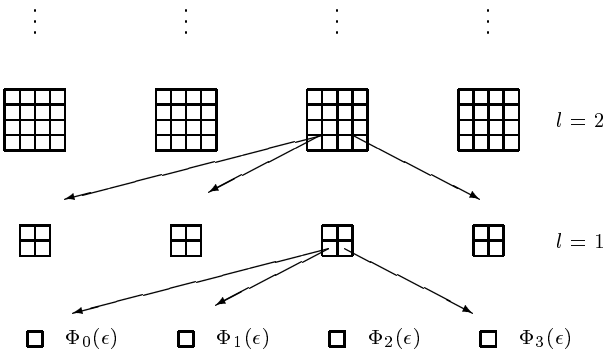


Figure 3.6: Fast decoding of a WFA. To simplify the figure, we have only drawn the coefficient vectors of quadrant 2 and 3 of the state s_2.

This algorithm has a running time of only $O(|S|^2|\Sigma|^n)$, which is bought at the cost of a higher memory consumption of $O(|S||\Sigma|^n)$. Other techniques for efficient WFA decoding can be found in [URP96].

```
/**
* decodes a WFA.
*
*@param A is the WFA to be decoded.
*@param n is the decoding resolution.
*@return the decoded image with resolution 2^n × 2^n
 if quadtree partitioning is used. For bintree
 partitioning double the parameter n.
**/
void WFADecode (WFA A, int n)
  {
  for (i=0; i < |S|; i++)
    Φ_i(ϵ)=F_i;
  for (l = 1; l ≤ n; l++)//construct levels bottom up.
    for (i = 0; i < |S|; i++)//operate all states.
      for (q ∈ Σ)//operate all quadrants.
        for (σ ∈ Σ^(l-1))//operate all pixels in quadrant.
          Φ_i(qσ)=∑_{j=0}^{|S|-1} W_q(i,j)Φ_j(σ);
  for (σ ∈ Σ^n)
    f_A(σ)=∑_{i=0}^{|S|-1} I_i Φ_i(σ);
  return the image consisting of the pixels f_A(Σ^n);
  }//end WFADecode
```

Listing 3.2: WFA decoding algorithm.

Note: In practice, we have to restrict the number of coefficients used to approximate a quadrant to a small number. With this restriction, the running time is reduced to $O(|S|\,|\Sigma|^n)$. ■

Memory Consumption of the Decoding Algorithm This process shows high memory consumption. Since a WFA has in practice between two and six thousand states and the state images are represented by floating precision numbers (four bytes), the decoding process would take approximately

$$2000 \cdot 4 \cdot 512^2 = 2097152000 \text{ bytes } \approx 2 \text{ GB} \tag{3.17}$$

where we assume that we have to generate 2000 state images at a moderate resolution of 512×512. Because of this high memory consumption, our implementation only computes the state images which are required for the construction[8] of the output image.

Note: The fast WFA decoding algorithm constructs the state images bottom up by combining the lower resolution state images corresponding to the coefficients. Thus the initial states may not be defined in a WFA consistent manner but their state images can be calculated in an arbitrary way (for example pre-calculated or as a codebook as in vector quantization). In section 4.4.4 on page 70 we will examine how we chose the initial states. ■

3.5 Top Down Backtracking WFA Construction

In the following considerations we drop the vectors I and F. The vector F holds average brightnesses of the states (see equation 3.11 on page 42) and can be easily calculated. The vector I determines which state image should be decoded. This is normally the unit vector e_0.

Culik [CK94] has proposed a major improvement of the WFA compression algorithm. This new algorithm is a kind of backtracking algorithm and is designed from scratch for lossy coding. In each step, the algorithm chooses one of two alternatives, namely whether the considered quadrant has to be approximated by previously generated state images or that quadrant has to be partitioned.

Since we now address lossy image compression, we first choose a quality factor $q \in \mathbb{R}$. The algorithm minimizes the value b (henceforth called *badness*) defined by

$$b = c + e \cdot q \tag{3.18}$$

where c is the storage cost[9] for the generated automaton (for example in bit) and e is the resulting approximation error (which is an arbitrary metric defined

[8] This process is similar to *lazy evaluation* of parameters in programming languages.

[9] The storage costs are the information gains of the transmitted symbols. This is one of the reasons, why we have to investigate the design of the statistical models for WFA coding in detail.

on multi-resolution images). In each approximation of a quadrant of the original image there are two alternatives: *approximation* and *partition*. In the processing of a quadrant, the badnesses for these two alternatives are calculated and the alternative with the lower badness is chosen. In the pseudo implementation in listing 3.3, these badnesses are called approxBadness and partitionBadness. The quality factor q of equation 3.18 is called `qualityFactor` in the pseudo implementation. Since this process is made recursively in each quadrant, the algorithm is a backtracking algorithm.

The algorithm has to be started with constructWFA(ϵ) to construct a WFA approximating the whole input image Ψ.

```
/**
* creates a WFA that approximates
* the multi-resolution image Ψ.
*
*@param p is the path of the image segment to be
* approximated.
**/
constructWFA(Path p)
//External variables: n is the number of created states,
//initially set to zero.
//Φ is an array of multi-resolution images representing the
//state images.
//· represents the concatenation operator defined on paths.
  {
  n++;
  create a new state s_{n-1};
  Φ_{n-1}=Ψ_p//set the state image of s_{n-1} to Ψ_p;
  for (q ∈ Σ)
    {//operate the image segment p·q.
    //calculate direct approximation.
    find a coefficient vector r = (r_0, r_1, ..., r_{n-1}) ∈ ℝ^n
      such that Ψ_{p·q} is "approximated well" by
      r_0Φ_0 + r_1Φ_1 + ... + r_{n-1}Φ_{n-1};
    let c_a be the cost for storing r (in bit);
    let e_a be the approximation error caused by r;
    approxBadness=c_a + e_a*qualityFactor;//calculate badness.

    //calculate partition recursively.
    int oldn=n;
    constructWFA(p·q);
    let c_p be the cost (in bit) for the new "sub-WFA" constructed
      in the recursion;
    let e_p be the approximation error caused by the "sub-WFA"
      constructed in the recursion;
    partitionBadness=c_p+e_p*qualityFactor;//calculate badness.

    //choose best alternative.
    if (approxBadness<partitionBadness)
      {//direct approximation is "better".
      n=oldn;
      delete the states with indices ≥ n created in the recursion;
```

```
    for (each non-zero component r_i)
     {
     add an edge with label q and weight r_i from the
        current state s_{n-1} to the state s_i;
     }
  else
    {//partition is "better".
    add an edge with label q and weight 1 from the
        current state s_{n-1} to the first (top) state of the sub-WFA created
        in the recursion;
    }
}//end constructWFA
```

Listing 3.3: Outline of the top down WFA creation algorithm.

Next we refine the first backtracking algorithm in some directions. The first modification is that we split the procedure `constructWFA` to the procedures `createState` and `operateStateQuadrant`. The procedure `createState` creates a state approximating a given image segment. The procedure `operateStateQuadrant` constructs all state transitions (coefficient vectors) pointing from the current state to other states. The reason for this splitting is (besides that it is easier to comprehend), that the procedure `createState` will be refined further in the next section. The second modification of the pseudo implementation is that the notation will be much more technically. This is due to the fact that we will explain some technical details of the implementation.

For the explanation of the pseudo implementation the following classes and objects have to be made clear:

- `FloatPicture` is a class modeling a mixture of image and vector. We can apply both methods of image processing as the cutting of certain quadrants but also the methods of a vector like the calculation of inner products on an instances of this class. The method used to calculate the approximation error is called `distance`, which is for example mean squared error defined in appendix C on page 169). This class was introduced to treat images as vectors.

- `domainPool` is an instance of the class `DomainPool` which approximates vectors in the current resolution. Unlike a normal basis, states are inserted instead of vectors. In order to tell the domain pool that a new state has been created, the procedure `notify` is called after which the domain pool can rearrange its dictionary (see section 4.4.4 on page 70). Coefficients obtained by the domain pool are already quantized.

- `encoderInterface` is an instance of the class `EncoderInterface` providing a simple interface to the entropy encoder. This class incorporates thus the arithmetic encoder and the adaptive source model[10]. This object

[10] Due to data compression literature we use the term *model* to denote an implementation for estimation of the statistical distribution.

is required to estimate the cost for storing data which is needed for the reconstruction of the automaton.

- `wfaTree` is a representation of the current WFA tree. The nodes of the tree can be accessed randomly.
- `Path` is a class whose instances incorporate paths to image quadrants (words of Σ^*).
- `State` is a class whose instances incorporate states of the WFA (precalculated and created states). The composition of this class is further explained in section 7.5 on page 137.

A pseudo implementation of the algorithm is given in listings 3.4 and 3.5, where we approximate the image `originalImage` and denote the concatenation of paths by using the $\cdot$ operator. The algorithm is started with `createState(`ϵ`)`. The desired approximation of the original image is the state image of the state s_0 after termination of the algorithm. Thus we set $I = (1, 0, \ldots, 0)$.

```
/**
* creates a state that approximates an image segment.
*
*@param path is the path of the image segment to be
* approximated.
*@return the badness of the created state.
**/
float createState(Path path)
  {
  State quadState= new State(path);
  quadState.setStateImage(originalImage.cutSegment(path));
  wfaTree.add(quadState);
  domainPool.notify(quadState);
  float stateBadness=0;
  for (q ∈ Σ)
    stateBadness+=operateStateQuadrant(quadState, q);
  return stateBadness;
  }//end createState
```

Listing 3.4: Top down state creation algorithm.

```
/**
* creates a coefficient vector of a given quadrant
* of a state.
*
*@param state is the state to operate on.
*@param q is the number of the coefficient vector
* to be created.
*@return the acquired badness.
**/
float operateStateQuadrant(State state, Σ q)
  {
  // · represents concatenation of paths.
```

```
Path quadPath=state.getPath() · q;
FloatPicture quadVector=
              originalPicture.cutSegment(quadPath);
//calculate approximation.
MPVector coeffs=domainPool.approximate(quadVector);
approxError=
      quadVector.distance(domainPool.getVector(coeffs));
approxCost=encoderInterface.cost(coeffs);
approxBadness=approxCost+qualityFactor*approxError;
//calculate partitioning.
int oldExistingStates=wfaTree.size();//memorize size.
partitionBadness=createState(quadPath);

if (approxBadness<partitionBadness)
  {//delete the states created in the recursion.
  int newExistingStates=wfaTree.size();
  for (l=oldExistingStates; l<newExistingStates; l++)
    wfaTree.removeElementAt(oldExistingStates);
  domainPool.reduceTo(oldExistingStates);
  state.setQuadrantWeights(coeffs, q);
  //restore the adaptive source model to the state before
  //the recursion.
  encoderInterface.rollBack();
  //update the adaptive source model
  //with the new coefficients.
  encoderInterface.updateModel(coeffs);
  return approxBadness;
  }
else
  {//insert pointer to the new tree.
  MPVector pointer=new MPVector(quadState.index(), 1.0);
  state.setQuadrantWeights(pointer, q);
  return partitionBadness;
  }
}//end operateStateQuadrant
```

Listing 3.5: The state operation algorithm.

Note that if the tree is drawn with the root at the top, the algorithm constructs the tree top down in DFS[11] order. The main drawback of this technique is that the encoder uses other state images at encoding time than at decoding time. A variant of this algorithm avoiding this drawback is described in the next section.

Let us make an important observation: the best known statistical models are adaptive source models [BCW90], which adapt the statistical model after each transmitted symbol. Note that the cutting of the subtrees causes the model used for the construction of the WFA to differ from the model used for the storage of the WFA. Since storage costs are estimated with the help of the statistical model, the costs are no longer exact if the data stored in cut subtrees (which are

[11] DFS $\simeq$ depth first search [CLR91, Sed88].

not transmitted) affect the statistical model. Thus the statistical model has to be restored to the state before the construction of the subtree if such a subtree has to be cut. Due to database terminology we call this restoration step a *rollback operation*. See section 4.2 on page 64 for details of our implementation of the rollback operation.

3.6 Bottom Up DFS Construction of the WFA Tree

In this variant of the WFA inference algorithm, the WFA tree is constructed bottom up in DFS order. In the same way as in top down WFA construction, we try to approximate the current quadrant by a linear combination of previously created state images. In this WFA construction algorithm, we use only states which are processed completely, i.e. all quadrants are already approximated. On the other hand, the alternative is tested to create a better approximation by splitting the quadrant. We then choose the better of these two alternatives. The desired approximation of the original image is—after termination of the algorithm—the state image of the most recently created state. With these explanations, the pseudo implementation in listing 3.6 can be given.

```
/**
* creates a state that approximates an image segment.
*
*@param path is the path of the image segment to be
* approximated.
*@return the badness of the created state.
**/
float createState(Path path)
  {//bottom up creation of a WFA.
  State quadState= new State(path);
  float stateBadness=0;//badness of the state.
  for(q ∈ Σ)//operate all coefficient vectors.
    stateBadness+=operateStateQuadrant(quadState, q);
  quadState.setStateImage(quadState.decode());
  domainPool.notify(quadState);
  wfaTree.addElement(quadState);
  return stateBadness;
  }//end createState
```

Listing 3.6: Bottom up state creation algorithm.

The images which are used in the method `operateStateQuadrant` are not extracted from the original image but rather the state images from generated and initial states. The principle is that the WFA decoding procedure can be applied to the subtrees which are created during the recursive call. For a practical implementation, a caching procedure is used such that a given state image has to be computed only once. This cache implies high memory usage, which seems acceptable according to our experience. An important observation is that this method of creation of the automaton diminishes the necessity for transmitting the average brightnesses of the states. This fact also improves the compression

ratio. Another benefit of this algorithm is that both errors introduced by the approximation of the vectors or by the quantization of the coefficients can be compensated by a feedback loop, shown in Figure 3.7. Note that this figure is simplified since some aspects like the quantizer and other classes have been omitted. In this figure, we can see that an input image is passed to the WFA encoder which cuts the image to image segments and passes them to the domain pool. The domain pool approximates an image segment by state images generated by the WFA decoder. The cost of this approximation is estimated by an encoder interface. With this cost and the approximation error, the badness can be calculated and passed back to the WFA encoder.

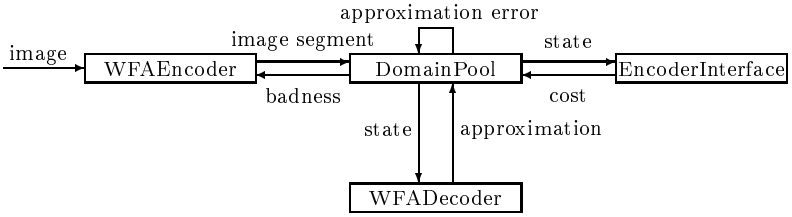


Figure 3.7: The feedback loop used in the WFA encoder.

The algorithm is relatively well natured to a change in parameters (as for example the accuracy of coefficient's quantization) because of the feedback loop. This behavior is due to the fact that with large approximation or quantization errors, the current badness increases and the addressed image segment is further divided into parts by the procedure `operateStateQuadrant`. We will see in section 4.6 on page 87 that this technique of WFA construction shows the best compression efficiency.

A drawback of this method is that a state cannot reference itself, but only states that have already been processed. In empirical tests we have observed that such self similarity rarely occurs in "natural" images.

Another side effect of this bottom up (BU) method is that an initial basis with at least one state is required. However, this fact is no drawback, since Culik already remarked in [CK93] that such an initial basis could also be helpful for the top down WFA construction algorithm.

Note that since the states are put to the WFA tree as late as they have been fully processed, the state approximating the whole input image is inserted lastly. Thus the initial distribution vector has the form $I = (0, ..., 0, 1)$.

Conservativeness Revisited By examining the BU-DFS WFA construction algorithm we can argue why the WFA constructed using this technique is conservative. If we assume that the initial states show conservative state images (which has to be guaranteed by construction), the composed states are likewise conservative, since they are constructed by other states via linear combinations. Since we have seen that the conservative functions span a linear subspace, all

state images are conservative, and thus also the multi-resolution image defined by the whole WFA. For a sketch of the induction step see Figure 3.8.

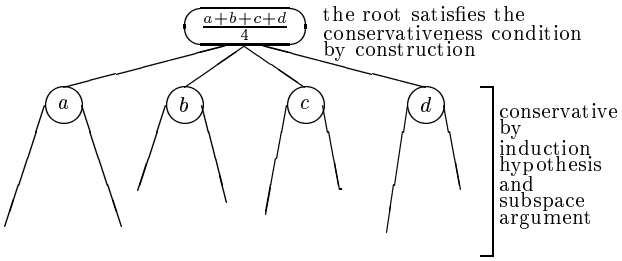


Figure 3.8: Induction step of the proof that all WFA state images are conservative.

3.7 Breadth First Order Generation of WFA Trees

In the last sections we presented the generation of WFA trees top down and bottom up in DFS order. During the optimization of the entropy coding process, we observed that the BFS order is the best order to store the WFA tree components. The drawback of this method is that the encoding cost of the stored parameters is no longer exact, since the order of storage influences the adaptive source model. For this reason, the generation of the WFA tree in BFS order has to be investigated. Again, we study the generation of the WFA tree in two directions: top down and bottom up.

As in the DFS WFA generation, the top down direction does not permit the exact calculation of the encoding error and is therefore not investigated any further.

The bottom up BFS generation of the WFA tree is the next alternative we will check. We explain the algorithm using quadtree partitioning. The proposed algorithm is similar to the generation of Huffman codes (see appendix F.3.2 on page 193) and works as follows: first we generate a WFA tree for each pixel of the original image. These trees consist of an initial basis (containing at least a state with a constant non-vanishing multi-resolution image) and one state representing the associated pixel. Each pixel has to be approximated with the initial basis, which could be easily performed by using one coefficient. By doing so, the associated badness can be stored in this node. In this way we obtain a set of WFAs (automaton pool).

After this initialization, we start the next stage by merging these one pixel automata to four pixel automata. This procedure is performed successively for each level of the paths in a predefined order. The merging process ends when only one tree remains in the automaton pool representing the desired approximation. Since each merging process reduces the number of trees, the process terminates.

When operating successively the quadrants of a given state image, the algorithm first tries to approximate the quadrant of the state image and compares the calculated approximation badness with the badness of the state that has the path of that image segment. Now, as in the DFS version of WFA tree construction, the two badnesses are compared. If the approximation badness is smaller, the coefficient vector is inserted and the automaton for that quadrant of the state image is deleted. If the partition badness is smaller, the state inserts a pointer to the state approximating the current quadrant. Since the merging process is steered by the badness concept, the WFA is built in a rate–distortion constrained manner.

A pseudo implementation is given in listings 3.7 and 3.8, the process is illustrated in Figures 3.9–3.11.

```
/**
* generates a WFA.
**/
void bfsWFA()
  {
  for (each pixel)//initialization.
    pool.addAutomaton(pixel);

  //n shall be the path length of the pixels.
  for (pathLength=n-1; pathLength>=0; pathLength--)
    {//merging stage.
    for (each Path p with |p|==pathLength)
      {
      State state= new State(p);
      float badness=0.0;
      for (q ∈ Σ)
        badness+=operateStateQuadrant(state,q);
      state.setBadness(badness);
      state.setStateImage(state.decode());
      pool.add(state);
      domainPool.notify(state);
      }
    }
  }//end bfsWFA
```

Listing 3.7: Bottom up BFS WFA creation algorithm.

```
/**
* operates a quadrant of a state.
*
*@param state is the state to be processed.
*@param q index of the coefficient vector to be operated on.
*@return the badness of the approximation of the operated
* quadrant.
**/
float operateStateQuadrant(State state, int q)
  {
  Path quadPath=state.getPath()·q; //augment the path.
  FloatPicture quadVector=
```

```
                originalPicture.cutSegment(quadPath);
//calculate the approximation.
MPVector coeffs=domainPool.approximate(quadVector);
approxError=
     quadVector.distance(domainPool.getVector(coeffs));
approxCost=encoderInterface.cost(coeffs);
approxBadness=approxCost+qualityFactor*approxError;
//get the partitioning.
quadState=pool.getState(quadPath);
if (approxBadness<quadState.badness())
  {//insert approximation and delete quadState.
  pool.remove(quadState);
  state.setQuadrantWeights(coeffs, q);
  return approxBadness;
  }
else
  {//retain the state quadState.
  MPVector pointer= new MPVector(quadState.index(), 1.0);
  state.setQuadrantWeights(pointer, q);
  return quadAutomaton.badness();
  }
}//end operateStateQuadrant
```

Listing 3.8: The state operation algorithm for BFS WFA construction.

This variant of WFA tree generation has the following drawbacks:

- The method poses high computational cost because we cannot apply a cutting of the recursion tree. A solution to this problem is the creation of WFA trees up to a given level via the BU-DFS algorithm and merge the trees with the BU-BFS algorithm to one tree.
- The rollback of the statistical models is impossible because the data is not cut at the "end" as in the DFS algorithm but can be cut at any place. We think that this is the most serious problem with the BFS generation of WFAs.

We have implemented the BFS WFA construction algorithm. But the achieved results were worse than the DFS WFA construction algorithm, and so we decided not to develop this technique any further and concentrated on DFS WFA construction.

3.8 Conclusion

We have experienced that the BU-DFS method for WFA construction yields the best results of all considered techniques. In the next chapters we focus our attention to this generation method. For a rate–distortion diagram comparing the BU-DFS and BU-BFS method for WFA construction see Figure 3.12.

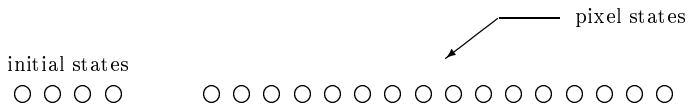


Figure 3.9: Initialization of BFS WFA. The figure shows only partitioning edges.

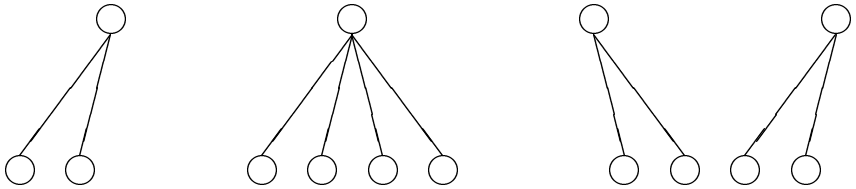


Figure 3.10: Merging operations of BFS WFA obtained using the nodes shown in Figure 3.9. The figure shows only partitioning edges.

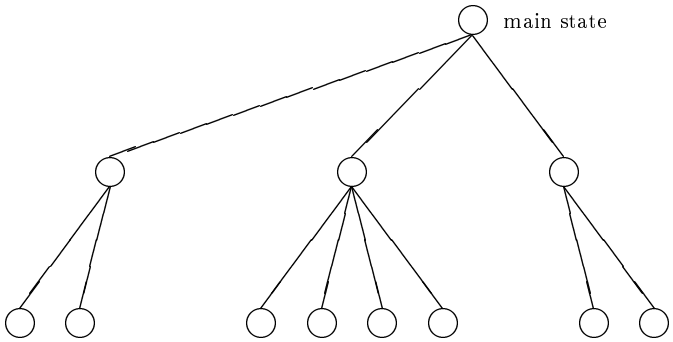


Figure 3.11: Final BFS WFA obtained by merging the trees in Figure 3.10. The figure shows only partitioning edges.

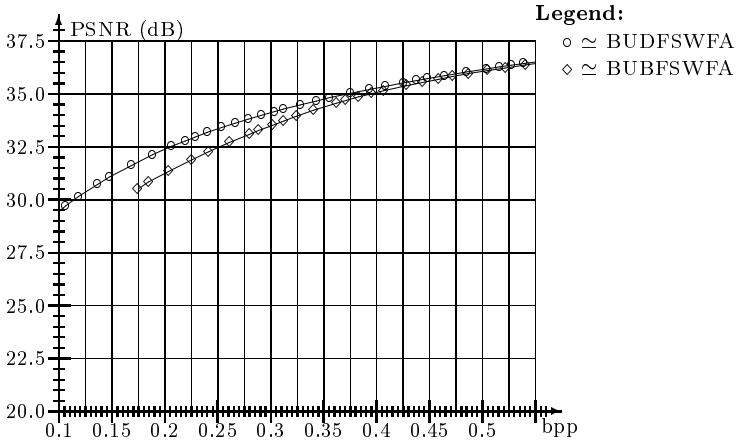


Figure 3.12: Rate–distortion diagram of the image Lenna.

Chapter 4

Further Enhancements of the WFA Coder

In this chapter we examine several further refinements of the WFA coder. We list a huge amount of techniques. Since we almost always have no theoretical basis for deciding which enhancement is worth considering, we had to implement most of them. The decision of which enhancements to choose is finally made by the optimization of the coder using genetic algorithms.

4.1 Enhancements of the Image Partitioning Technique

In section 3.1 we have introduced some basic techniques for image partitioning. Now we examine, how we can improve the WFA compressor using enhanced partitioning methods.

4.1.1 Light HV Partitioning

We can observe an important property of bintrees: if the first bintree partitioning is made horizontally, the image parts belonging to the leaves of the bintree are always either squares or twice as wide as high. Therefore, to give the bintree more flexibility, we have added an option for partitioning the square image segments either horizontally or vertically. We call this method for partitioning *light HV partitioning* since it is a special case of the HV partitioning method introduced in the next section. Note that this method requires the efficient storage of additional data and the improvement in image quality must be high enough to exceed the extra cost.

To explain some difficulties with the light HV partitioning, we express an image from now on as a $n \times m$ matrix with $n, m \in \mathbb{N}$ and not as a multi-resolution image as in chapter 3. Such an image can be interpreted as a nm vector by

using the techniques defined in section 1.1 on page 8. If nothing else is stated, the raster scan order is utilized.

A problem of the light HV partitioning scheme is that different shapes of the image segments occur. We explain this difficulty by using an example. Consider that an image segment of size 16×16 is to be partitioned. When the image segment is partitioned vertically, we would have to approximate two image segments of size 8×16. From the mathematical point of view, we could utilize all image segments of the sizes 8×16 and 16×8 and approximate the given image segment with these vectors. But in WFA coding this obvious technique yields bad results because of the following reason: if some image segments had the resolution 16×8 and are now interpreted as 8×16 image segments we would obtain such scattered images as can be seen in Figure 4.1 (right side). Such high frequencies arise because neighboring relations (and thus correlations) are destroyed.

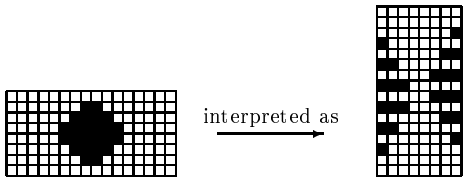


Figure 4.1: Interpretation of a vector as an image of wrong resolution.

Another solution to this approximation problem would be to scale the images of resolution 16×8 up by the factor two in y-direction and afterwards scale the image down by the factor two in x-direction. This procedure wastes details in the image segment and can in the worst case lead to pixelization of the image segment and therefore to bad results. Another possibility would be to scale such image segments up to the next square image and approximate on that resolution. This procedure would both require more memory and twice the time to calculate the required scalar products. Our solution for this problem is the transposition of the image matrix. This technique avoids all drawbacks mentioned above and leads to good results.

For the determination of the partitioning line, we use the heuristic of Fisher which is described in the next section. For an illustration of this partioning scheme see Figure 4.2. As you can see in Figures 4.3–4.6, the image quality is enhanced by approximately 0.25 dB PSNR at the same data rate by using the light HV partitioning scheme.

4.1.2 HV Partitioning

A more flexible approach for partitioning an image was introduced by Y. Fisher and S. Menlove in [Fis95b] with good experimental results using an IFS encoder (see appendix H on page 201). In this partitioning method, the codec is able to

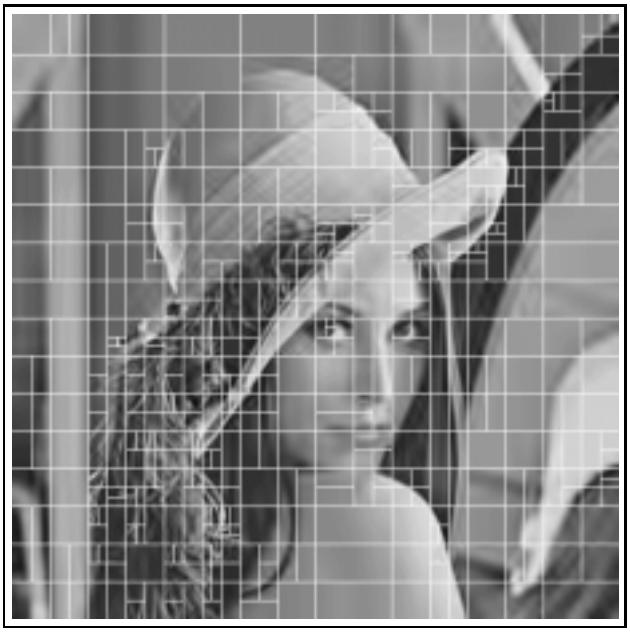


Figure 4.2: Light HV partitioning of the image Lenna.

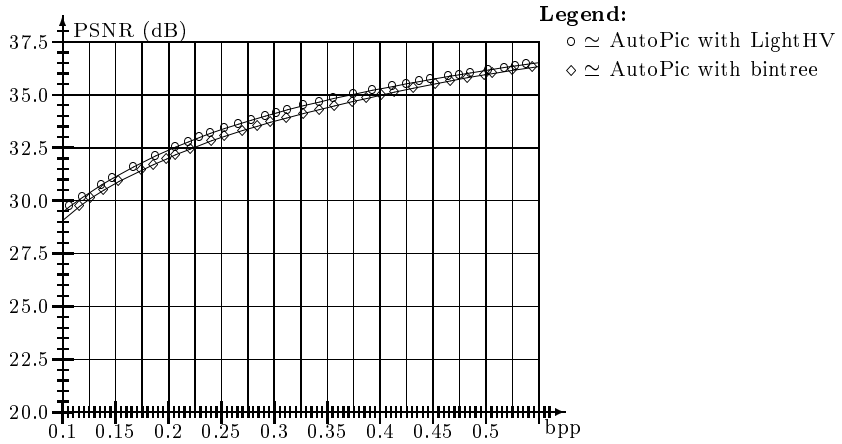


Figure 4.3: Rate–distortion diagram of the image Lenna comparing the different techniques.

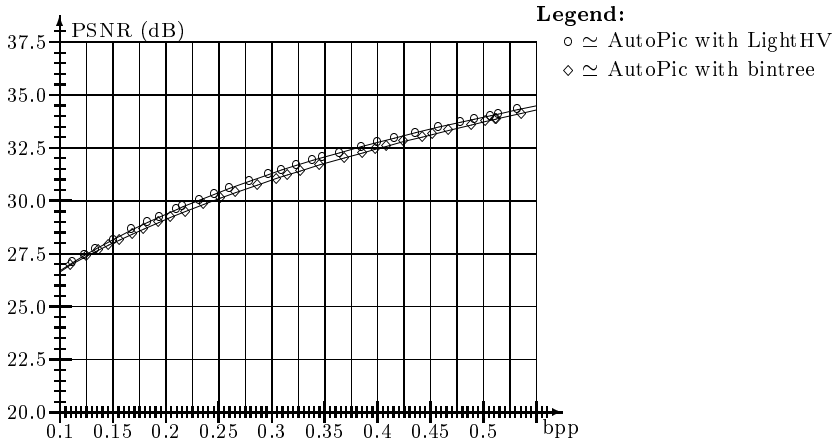


Figure 4.4: Rate–distortion diagram of the image Boat comparing the different techniques.

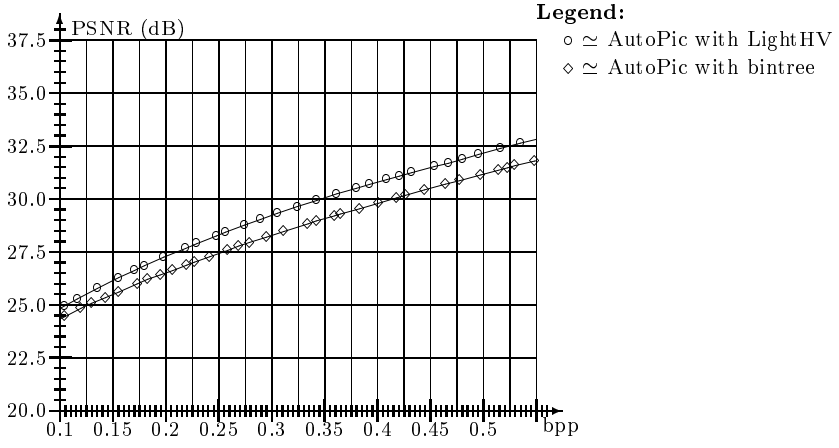


Figure 4.5: Rate–distortion diagram of the image Barb comparing the different techniques.

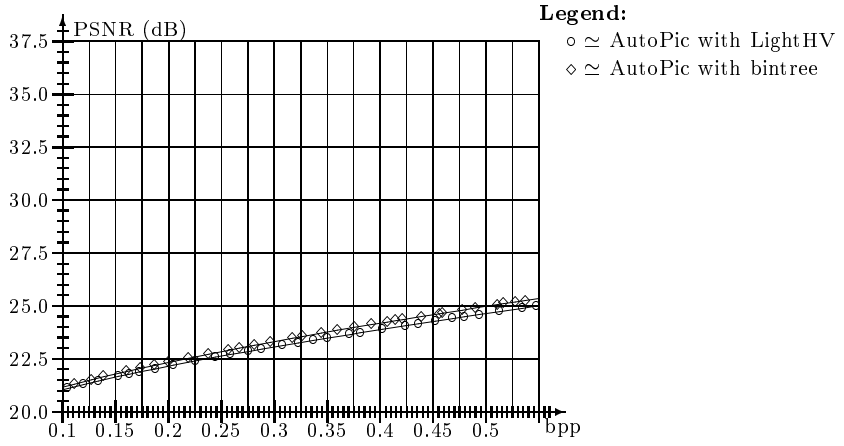


Figure 4.6: Rate–distortion diagram of the image Mandrill comparing the different techniques.

move the partitioning line from the middle towards outer regions of the image segment.

This approach was also tested in our implementation of the WFA codec. However, we observed that the gain in image quality did not compensate the extra cost for storing the information added to the bintree structure. For this reason, the light HV partitioning mentioned above was introduced, where extra storage cost is small. Another reason why this scheme does not yield good results in our implementation could be the problem of scaling. As mentioned in the last paragraph, image details are lost by scaling down. On the other hand, scaling up is not feasible.

We implemented the heuristic used in [Fis95b] to determine the partitioning line. For an $m \times n$ image segment with gray values $r_{i,j}$ ($i \in \{0,\ldots,n-1\}$, $j \in \{0,\ldots,m-1\}$), Fisher computes the *biased differences*

$$h_j = \frac{\min(j, m-j-1)}{m-1} \left| \sum_{i=0}^{n-1} (r_{i,j} - r_{i,j+1}) \right| \tag{4.1}$$

and

$$v_i = \frac{\min(i, n-i-1)}{n-1} \left| \sum_{j=0}^{m-1} (r_{i,j} - r_{i+1,j}) \right|, \tag{4.2}$$

chooses the biggest value of the set $\{h_0,\ldots,h_{m-2},v_0,\ldots,v_{n-2}\}$ and splits the image segment at the corresponding index (horizontally if the biggest value is one of the h_i and vertically otherwise). The partitioning obtained is said to cut the image segments at strong horizontal and vertical edges while avoiding narrow image segments.

A better solution would be backtracking to determine the best partitioning line by checking which choice leads to the best results. That alternative has not

been implemented because the running time would be too high for a practical solution.

However, we currently can not rule out the possibility that HV partitioning can be refined in another way to cooperate with WFA encoding. This point could be an interesting research topic for the future of WFA encoding.

4.1.3 Other Methods for Image Partitioning

There are several other image partioning schemes. For some examples see Figure 4.7. A drawback of the hexagonal image partitioning is the problematic treatment of image boundaries. The drawback of a triangular image partitioning is the treatment of approximation where neighborhood relations may be destroyed as in HV coding. Other segmentation schemes may be found in [RD79].

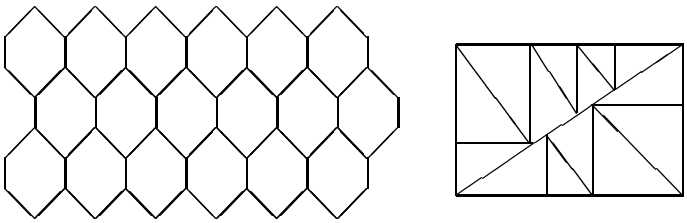


Figure 4.7: Hexagonal and triangular image partitioning.

4.2 The Statistical Model for WFA Coding

As stated earlier, the statistical models are of outstanding importance for data compression. In the next sections, we describe a sophisticated model supporting all operations required for WFA coding.

We do not know in advance the exact statistics of the stored parts (for example coefficients or indexes) of the WFA. Thus we need a model exhibiting the following features:

- it should be adaptive since the statistics differ from image to image,
- it should adapt to non-stationary statistics, since image statistics differ in different image regions,
- it should support finite context modeling and delta coding to benefit from prior coded symbols,
- it should support rollback to accurately trace statistics even if subtrees are cut from the WFA,

- it should ease the calculation of storage cost of single symbols even with "fractions of bits",
- the initial distribution of the model should be tuned to arbitrary distributions,
- it should work properly for "short sequences" despite high order of context modeling,
- the WFA coder should be able to switch fast between different models, as the coefficients model and the index model,
- the calculation of densities and distributions should be as fast as possible,
- the memory requirements should be as low as possible since many models of the different constituents of a WFA have to be accessed,
- the output of the models should be given in a way able to work with the best known statistical encoder, the arithmetic encoder,
- all properties should be independently adjustable to tune them to the current application.

4.2.1 Finite Context Modeling and WFA Coding

In order to incorporate a finite context model to our coder, we have to solve the problem of supporting backtracking. For that purpose, we implemented a data structure called *lazy stack*, a stack that also allows access to elements lying below the top element. Therefore, we overloaded the top method with an integer parameter telling the lazy stack which layer of the stack is to be accessed. The behavior of the extended top method is shown in Figure 4.8. The finite context model with rollback works in the same way as the standard finite context modeling with the following differences:

- Update operation: we first update the current model, afterwards push the obtained symbol onto the stack and change to the next context, which is accessed via the extended top method.
- The rollback operation is performed in reverse order: first the stack is popped. Afterwards the context is switched using the obtained stack and finally the current non-context model is rolled back.

An example for such a model is shown in Figure 4.9 and the described manipulation is illustrated in Figure 4.10. In these figures we write the relative frequencies to the right of the associated symbols.

Note that these models have the following drawbacks:

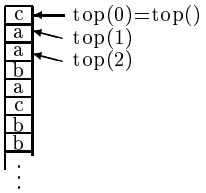


Figure 4.8: Extended stack.

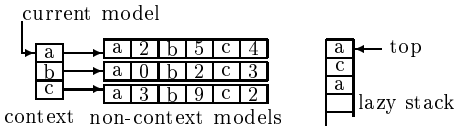


Figure 4.9: A model with context size 1 and alphabet size 3.

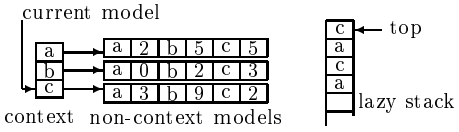


Figure 4.10: The model of Figure 4.9 after updating a "c".

- The memory consumption of the model grows exponentially with the context size. However, in practice there is no need to instantiate all non-context models.

- Since often only short sequences have to be encoded, the non-context models cannot adapt fast enough because the input symbols are spread to many non-context models. A solution to this problem are blended (mixed) models which are described in section 2.4.3 on page 33.

These drawbacks have led many image compression researchers to neglect this coding method. Despite that, we have observed considerable improvements in compression performance for some automaton parameters.

For the storage of all parameters of a WFA, an adaptive statistical model (`wfa.coder.Model`) is used. As entropy coder, an arithmetic coder (`wfa.coder.ArithmeticEncoder` and `wfa.coder.ArithmeticDecoder`) was used. This coder was selected because it also works for small input alphabets near the entropy limit and can easily be adjusted to adaptive source models. In our implementation, we have exploited the fact that the storage cost for automaton parameters can be estimated very precisely. For this estimation, the adaptive source model was used. Because a backtracking algorithm is used for the creation of the automaton, the model has to be rolled back in case that a subtree is cut from the automaton. For this reason, a window was used that holds a fixed[1] number of input symbols. The symbols in this window are the last k symbols the model was last updated with. In order to support unlimited rollback operations, a *deque* (double ended queue) is used. This data structure is an enhanced FIFO-queue, able to insert and delete data elements at both ends [HN86]. If a symbol is inserted into the model, it is inserted at the front of the deque. But now the deque is one symbol too long. So the last symbol is removed from the deque and pushed onto a stack. If an element is rolled back, the element at the front of the deque is removed and the element on top of the stack is popped and inserted at the end of the deque. In this manner, a flexible model is obtained, supporting all operations required for the creation of the WFA. For an illustration of this model see Figure 4.11.

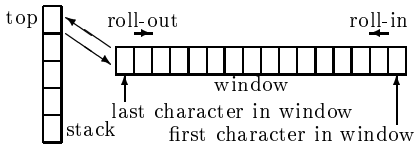


Figure 4.11: Window-based model with rollback.

[1] It is also possible to vary the window length during coding time, which was not applied here because of running time problems.

4.3 Matching Pursuit and WFA Coding

In this section we treat some special features of the matching pursuit approximation due to WFA coding.

Rate–Distortion Constrained Matching Pursuit In our implementation, we employed the concept of badness in different ways. Two of them concern the matching pursuit algorithm. The first usage is for the selection of the best matching MP element where we choose the element with the smallest badness in the standard matching pursuit algorithm (see section 1.4 on page 20). The second usage is for abortion of the approximation algorithm where we stop when the lowest badness of all elements lies above a given threshold. Note that this technique can be combined with the orthogonalization procedure of Gram and Schmidt.

4.4 Additional Refinements

In this section we represent some minor refinements of the WFA coding technique implemented in AutoPic. Note that all enhancements of the algorithm should be "compatible", meaning that they can be independently switched on and off. This is important because the algorithm parameters are optimized via a genetic algorithm. Many problems arise at the combination with the rollback support, as for example at the incorporation of run length encoding. More refinements of WFA coding can be found in [Kra95].

4.4.1 Calculation of the Scalar Products

Hafner [Haf94] and also Zimmermann [Zim97] use a recursive function to compute scalar products of state images. Zimmermann [Zim97] argues that the direct computation of the required scalar products is not feasible. However, this argument is not correct as the state images need not be zoomed to the resolution of the original image. The image segments can be approximated in the resolution of that segment. Another fact is that computers are now much faster than at the time the WFA technique was invented.

A drawback of the recursive calculation is that the initial states also have to be represented by a WFA structure. This disadvantage is avoided by our technique. On the other hand, this manner of calculating the scalar products leads to a gain in flexibility. As we will see, many of the optimizations of WFA coding considered here would not be possible without the ability of applying arbitrary image manipulations to the state images.

4.4.2 Choice of the Function Numbering

Since the function systems we use are (potentially) of infinite dimension, we have to choose a finite subset of these functions. We therefore have to decide which subset to choose and the numbering of the functions in our domain pool. In our implementation, we have in an early stage experimented with the usual zigzag order of the functions (see Figure E.2 on page 189) commonly used in transform coding. But this scheme did not yield the required compression performance. This is the case because of numerical instabilities occurring when approximating vectors in different resolutions.

For this reason, we have experimented with other numberings. We made the best experience with the numbering in Figure 4.12 where the numbers in the boxes define the numbering of the basis functions in the matching pursuit vectors (the sequency number of a function denotes the number of sign changes in the definition range).

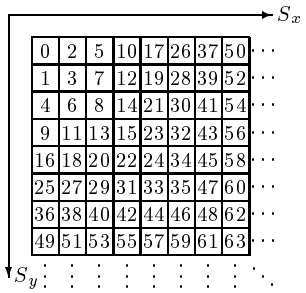


Figure 4.12: Order of the basis functions. S_x and S_y denote the sequency number in x and y direction, respectively.

4.4.3 Calculation of Storage Cost

An interesting implementation problem is the question how to obtain the required storage cost. There are two alternatives:

- The encoder could simulate the storage process and return how many bits would be stored by the entropy encoder. This method has the benefit of reflecting the real situation of the entropy coder.
- The second method is to calculate the information content (see section 1.3 on page 13) of the symbol to encode with respect to the statistical models. This method has the benefit of giving a slightly more precise calculation since the (in our case arithmetic) encoder is able to store "fractions of bits".

The second method achieves slightly better results and is also faster in our implementation.

4.4.4 Domain Pool Administration

In order to limit the required computational resources, we have decided to use a domain pool of fixed size instead of using all states for approximation. At the beginning of the encoding process, the pool is filled with initial states. The initial states can be replaced successively by states created during the generation of the automaton. Because rollback operations have to be supported without the need to store extra information about the update process of the domain pool, the choice of update strategies is limited. The possible strategies are taken from the theory of operating systems, where the most interesting strategies are the first in first out (FIFO) strategy and the least recently used (LRU) strategy. In our implementation, we employed the LRU strategy because it is reported in [Haf99] to be the best strategy for limited domain pool administration. Slightly better results were reported for an unlimited domain pool using all states (initial and generated states) for the approximation. We did not implement this strategy as we expect exorbitant running times.

Independent Administration of the Domain Pool for the Levels in the WFA Tree

An important observation is that often many objects of the same size and shape are present in natural images. An example for this phenomenon is an image of a juggler operating with many balls of the same size. In order to exploit this kind of redundancy, we enhanced Hafner's approach to independent domain pool administration in several levels in the WFA tree. We did this in the following way: if a domain vector is used, it is marked as used only in the current level. In this way it is appreciated that some state images are more adequate for approximation in one level than in others. In order to ensure that state images can be evaluated at all levels, a new state is inserted in all levels of the domain pool.

Choosing the Initial Basis

As mentioned above, the domain pool of the WFA coder has to be filled initially with base vectors. We made tests with different initial bases. An important observation, as remarked in [CK93], is the fact that the vectors of the initial basis do not have to be computed by a WFA but can be calculated directly. Since we are interested in approximating image segments with linear combinations of base vectors, we utilized function systems that are popular in image coding. In our WFA codec, currently the basis vectors of the sine, cosine, Slant, Walsh, Haar and Hadamard transform are implemented. We obtained the best results with the cosine transform also used in the popular image compression standard JPEG. For some images of the initial bases see Figures 4.13–4.18 (the function order is the order defined in section 4.4.2). For a more thorough treatment of these bases, see appendix E on page 179.

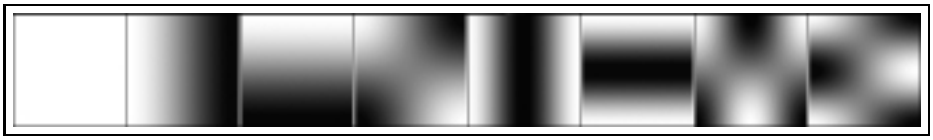


Figure 4.13: Some cosine images.

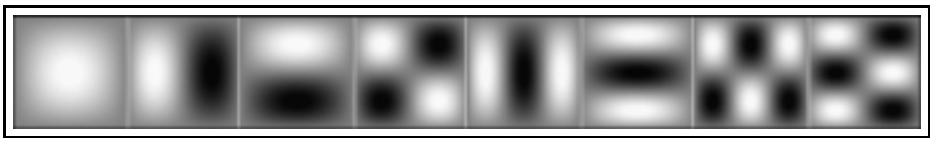


Figure 4.14: Some sine images.

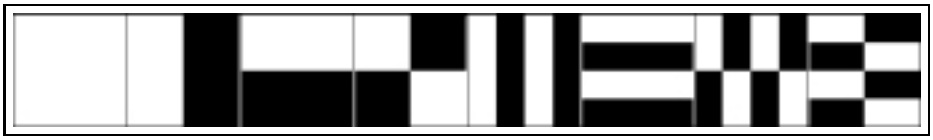


Figure 4.15: Some Walsh images.

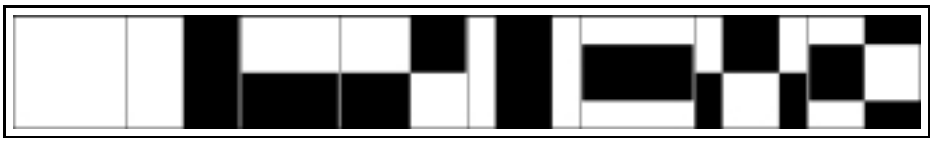


Figure 4.16: Some Hadamard images (sequency ordered).

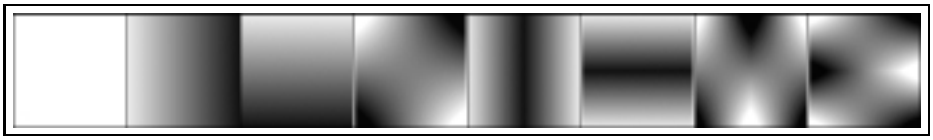


Figure 4.17: Some Slant images (sequency ordered).

Figure 4.18: Some Haar images.

Another way to construct the initial basis is similar to the codebook construction process in vector quantization. We did not implement this alternative since the codebook construction process (for example using the k-means algorithm) is very time consuming. For further details on vector quantization see [Kat94, NH88].

4.4.5 Modified Orthogonal Matching Pursuit

Since we use a dynamically changing domain pool, we are confronted with a severe problem when implementing orthogonal matching pursuit. Since the new vectors of the domain pool are inserted at an arbitrary position in the domain pool, the basis change matrices have to be recalculated each time a new state is created. In order to reduce the computational complexity, we decided to loosen the property of orthogonality. We split the domain pool into a fixed and a dynamic part (see Figure 4.19). The fixed part is filled with domain vectors concentrating the greatest part of image energy (see appendix E on page 179). These vectors are fixed during adaptation of the domain pool. The dynamic part of the domain pool is administered as described in section 4.4.4.

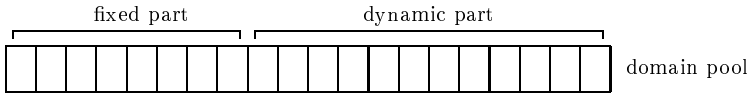


Figure 4.19: Splitting the domain pool in a fixed and dynamic part.

4.4.6 Stopping Criteria of the Approximation Algorithms

We stopped the linear approximation and matching pursuit algorithm if the badness of the next element exceeds a given threshold. An important observation is that this greedy strategy is not optimal. Hafner made studies in which he let the matching pursuit algorithm run some steps further and cut the sequence afterwards at the "global" maximum. Because Hafner observed only little improvement in compression performance while the encoding time was approximately doubled, we did not pursue this optimization. However, we think that this strategy is worth a trial for the linear approximation. But we have to defer this step to a moment when it is clear if this manner of approximation is worthwhile at all.

4.4.7 Limiting the Fractal Transform

We noticed that states near the top and near the bottom of the WFA tree are not used frequently by the approximation algorithm. For this fact, we limited the insertion to the domain pool to states belonging to niveaus between the limits 5 and 10.

4.4.8 Non-Fractal Coding

At WFA coding in its original form, the currently operated image segment can be approximated by using state images of all created states. Because of the exact[2] cosine basis, one can see that the coding has the following advantages if the approximation is performed using only initial states:

- A gain in compression speed because less state images have to be decoded.
- Less memory usage since less state images have to reside in memory.
- For rather small domain pools there was also a slight gain in compression efficiency.

4.4.9 Rate–Distortion Constrained Approximation

In standard rate–distortion constrained matching pursuit approximation, the required coefficients are computed by scalar products and afterwards have to be quantized. But the generated coefficient (the coefficient with least quantization error) does not necessarily achieve the most favorable badness, since not only the approximation error is implied but also the produced storage cost. The optimal solution would be to compute the badness for all coefficients, but this solution has the drawback of computational infeasibility. In order to find a compromise between these two extremes, we decided to additionally examine the second best matching coefficient, which probably has lower storage cost. This procedure is especially powerful in linear approximation since the cost of the coefficients is dominant. Since we did not find this simple heuristic in the literature, we called it *bad approximation alternative*. For an illustration of this concept, see Figure 4.20.

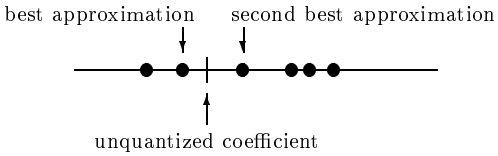


Figure 4.20: Heuristic for rate–distortion constrained approximation. The filled circles represent quantization values and the vertical line represents the unquantized value.

[2]Note that the cosine function is not precisely representable in a digital computer. We use the term *exact* to express that the basis vectors are calculated directly using the library function Math.cos. This technique is contrasted by the WFA coder of Hafner [Kra95], where the cosine basis is approximated with WFA states.

4.4.10 Storage of the Automaton

In this section we give a coarse description of how the WFA parameters and image partitioning have to be stored. More details will be given in section 7.3.1 on page 132.

Storage of the Tree Structure

Now let us take a look at the storage of the WFA tree structure without such specialties as HV partitioning. As mentioned earlier, the tree structure may be stored without entropy coding in two bits per node. For that case, we need to store only one bit for each half of a node. This bit defines whether the segment was approximated by matching pursuits or whether it points to a new node. In the former case, we have to subsequently encode the matching pursuit vector, while in the latter case, we can restore the index of the node to which the node points by the order in which the points are stored (either BFS or DFS order). Since it is clear that the probability to split a node is high at the root of the WFA tree while it is low at the bottom of the tree, the information whether to approximate or partition a quadrant is stored in BFS order. The DFS order for storing the tree structure was also tested, but had a slightly poorer compression performance. In order to mark the end of the tree structure, we first store the number of nodes. Another strategy is the storage of an end-of-tree-sign. One can read in many articles about signal compression that it is not clear how to assign the probability for such a symbol. Since we do not know the exact range of this number in advance, we employ Fibonacci codes for that task.

Another modification would be to make a run length encoding of that true false sequence. Since we observed that long runs only appear at the beginning and the end of the sequence, we have chosen to store the length of the true respectively the false sequence at the beginning of the sequence and to store the middle of that sequence by an adaptive source model. Since we know in advance the range of these numbers, we decided to store the two numbers in binary integer representation. We agree with the work of Hafner [Haf99] that the average cost of storing the tree structure is about 1.5 bits per node.

Storage of the Light HV Partitioning

The storage of the light HV partitioning (a binary sequence) is done in the same way as the storage of the tree structure. For any node belonging to a square state image having a niveau greater than zero, the decision whether this state image is partitioned horizontally or vertically has to be stored. This true false sequence is stored nearly in the same way as the tree structure using run length encoding at beginning and end and an adaptive source model for the remaining booleans. The storage rate of this part is typically less than 1.5 percent of the whole space.

Storage of the Matching Pursuit Vectors

As mentioned in [Haf99], the indices of the matching pursuit vectors take approximately 60 percent of the storage space of the encoded automaton. In order to reduce the overall storage cost, we have developed a coding scheme in three parts where the first two parts can be encoded without the storage of indices.

These three parts are addressed in the following paragraphs. The naming conventions are partially leaned on the notation of JPEG. These parts are

1. *DC part* which is the coefficient belonging to the constant gray value state image. Since the DC part is almost always addressed, this part is always stored. For this reason, there is no need to store an index or the size of that part, only the coefficient.

2. *AC part* is a part where the next coefficients after the DC part (lower frequency coefficients) are enqueued successively. Since the size of the AC part is held variable, we have to store the size of the AC part and the assigned coefficients. We also made experiments with a fixed number of AC coefficients, but with slightly worse results.

3. *MP part* is the part holding the matching pursuit components of the approximation. Since this part holds coefficients to arbitrary state images, we additionally have to store the indices (see section 4.4.4 on page 70) of the states which are used.

Note that the utilization of DC and AC coefficients also significantly speed up the implementation since such coefficients can be calculated in a single sweep over the domain pool. More details about the storage of WFAs can be found in [KF94].

4.4.11 Quantization Strategy

AutoPic only uses uniform quantization of the coefficients. This method is justified by results of coding theory stating that this method is optimal if the presence of an entropy coder is assumed. Nevertheless, Hafner has made experiments with non-uniform quantization leading to essentially the same results on the average as uniform quantization [Haf99].

There are also methods to perform adaptive quantizations. This family of techniques adapts the quantization bins during coding to a spot check distribution of coefficients seen so far. Note that for these methods to work, the codec has to memorize the unquantized coefficients of the spot check and requires significant amount of computations to recalculate the quantization bins. Up to date, we have not utilized this method because of the drawbacks stated above. For further details about adaptive quantization see appendix I on page 207 and [Ohm95].

Accuracy of the Coefficients We specify the precisions of the coefficients by using three parameters. The first parameter is a boolean parameter stating whether to encode a sign or not. The second part is an integer parameter specifying the binary precision before the dual point. It is important to know that digits to the left of the number is filled with zeroes. This parameter may also take on negative values meaning that we fill in zeroes right *after* the dual point. Finally, the third part is the binary accuracy of the number after the dual point. With these conventions, we can conclude that the number of bits used to represent a binary number is (without entropy coding)

$$\langle \texttt{beforePointPrecision} \rangle + \langle \texttt{afterPointPrecision} \rangle \tag{4.3}$$

where a 1 has to be added in case a signed number model is used. An example for this bit allocation is given in Figure 4.21.

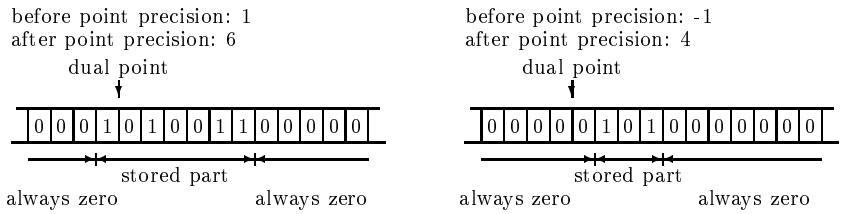


Figure 4.21: Description for the precision of the coefficients.

Storage Orders As we have seen, in the standard mode of AutoPic the nodes (states) of the WFA tree are built bottom up in DFS order. In order to make the storage more flexible, we implemented some iterators to access the WFA states in BFS and DFS order. The storage orders may also be reversed. In this way, four orders can be specified for each part of the WFA specification which can be independently switched.

Coding the Lengths of the Lists The question of how to encode the lengths of the AC and MP parts is similar to the question of how to encode the length of a string in a programming language. In PASCAL, the data of a string is preceded by the length of the string. In C, a string is backed up by a special end-of-string symbol. As we have seen, it is essential in data compression to know in advance (at least approximately) the probability of all symbols. We propose that the calculation of PASCAL-like length coding is easier to handle by statistical models.

Storage of the DC Parts We have observed that the preferred storage order of the DC coefficients is the BFS order. The genetic algorithm determined that a non-signed representation using 1 bit precision before the dual point and 6 bits after the dual point should be used. The DC part takes approximately ten percent of the whole compressed file size. The utilized window length is about eighty symbols.

Storage of the AC Parts The AC part consists of the storage of the length of that part and the corresponding AC coefficients. Here we use a signed number model with typically zero bit before the dual point and five bit after the dual point. The percentage to the whole storage consumption is approximately fifteen percent.

Storage of the Matching Pursuit Parts The storage of MP parts consists of the lengths of that parts, the indices and the corresponding weights. We have observed that the MP parts take approximately fifty percent of the entire storage size.

Up to now, the three pieces of this part are stored in the same order. We have observed that the best storage order for this part is also the BFS order. Once more computing power is available, we will examine whether it is preferable to store the three pieces in different orders.

The length of that parts is stored by using an adaptive source model. Since we expect that in the upper part of the WFA tree the MP lists will be longer while descending towards the lower part of the WFA tree, the encoding is performed in BFS order.

As mentioned in [Haf99], the coding of the matching pursuit indices is the most space consuming and therefore the most important part for image compression. We observed that it is slightly better not to encode the indices directly, but instead the differences to the last stored index (*delta coding*). Indices are stored with 7 bits accuracy. Note that each added bit nearly doubles the running time of the WFA encoding algorithm.

The last section for the storage of the MP part is the storage of the MP coefficients. We have observed that the quantization to -1 bit precision before and 5 bit precision after the dualpoint is most efficient with a window length of approximately sixty symbols.

4.4.12 Coding of Color Images

For the representation on computer monitors, color images are usually stored in the RGB color model, where the color of a pixel is given by its red, green and blue intensities. These values are typically stored using eight bits per value (*true color representation*). The three color channels may be interpreted as separate gray valued images. These images may be approximated by three automata. Since the color channels in the RGB space are highly correlated, it is a well-known technique to use a decorrelating color transform. The three color channels are encoded successively and the created states of the last two channels are enqueued after the first channel. For an illustration of the analysis and synthesis of a color image see Figures 4.22 and 4.23.

Since the color transform does not decorrelate the color channels completely, we can benefit from similar state images and distributions. By this type of WFA

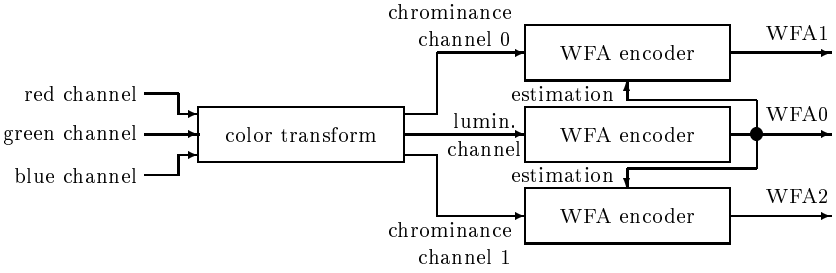


Figure 4.22: Encoding of color images with the WFA technique.

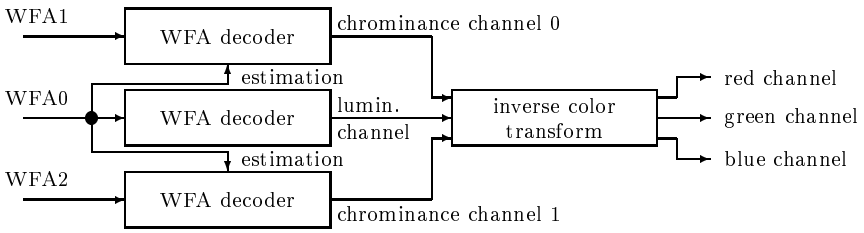


Figure 4.23: Decoding of color images with the WFA technique.

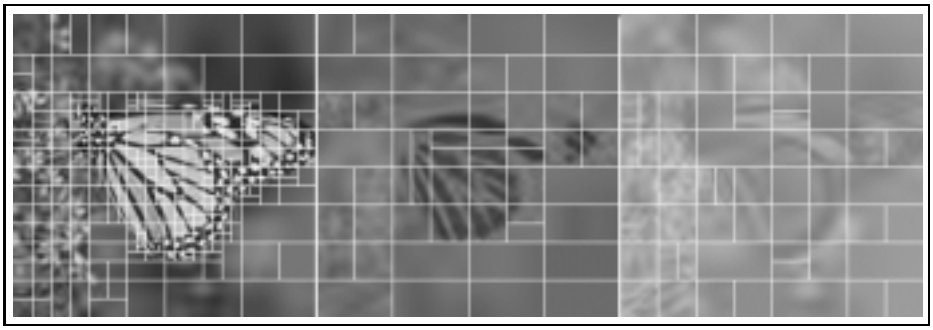


Figure 4.24: Partitioning of the color image Monarch in YC_bC_r color space, ordered Y, C_b, C_r from left to right. See also appendix B on page 167.

coding, the remaining correlations can be efficiently employed. There is also a storage overhead required to tell the decoder which states have to be decoded.

We made tests with RGB, YUV, YIQ and YC_bC_r color systems. The best results were obtained in YC_bC_r color space. A valuable observation is that these color channels are still correlated. This correlation can be exploited by fractal coding segments using segments of previously encoded color channels at the same spatial positions. For an illustration of this process see Figure 4.24.

Another method to exploit the remaining correlations between the color channels is to use the statistical model of the luminance channel as a starting point for the statistical models for the chrominance channels.

We observed that the data rate of the luminance component takes up about 75 percent of the whole data rate of a colored image. The quality reduction of the chrominance components are steered by a reduction of the quality factor q in the WFA construction algorithm.

There are other techniques to reduce the quality of the chroma channels:

- In the JPEG and MPEG data compression standards, the chrominance components are down-sampled by a factor of two prior to compression. We did not implement this kind of quality reduction since image details are diminished by this kind of transform.
- An interesting technique to reduce chroma band quality is the reduction of quantization precision of the resulting coefficients. We did not choose this alternative, since the use of the statistical models of the luminance channel for estimation of the models of the chrominance channels would be made less effective.

4.4.13 WFA-based Zooming

Since a WFA provides image information in a resolution independent manner, it can be decoded to a digital image at any resolution. When the image is

decoded at a larger size than the original image, details beyond this resolution are created by the WFA decoding algorithm. This kind of zooming was investigated with IFS and wavelet representations of images, called fractal and wavelet zoom respectively. In classical zooming, pixels are repeated several times. This method shows blocking artifacts (commonly called *pixelization*) which is the reason why other methods were invented, such as interpolation and the methods considered above. We present only two WFA-zoomed images (see Figures 4.25 and 4.26) for completeness since our WFA coder was mainly designed for image compression (the original is shown in Figure 4.34 on page 88).

Figure 4.25: WFA zoom of 128×128 image Lenna to resolution 512×512 (RMSE=9.86, PSNR=28.25 dB).

4.4.14 Edge Smoothing

The blocking artifacts at low fidelity coding are one of the most annoying drawbacks of the WFA codec when used without wavelet transform (see chapter 5 on page 103). In order to suppress the edge effect, we have implemented a smoothing procedure applied at the edges of the resulting tiles. To adjust the effect of the smoothing operator, we use a smoothing parameter $s > 1$. See Figure 4.27 for details. Note that the lower the parameter s the higher the smoothing effect is.

Figure 4.26: WFA zoom of 256×256 image Lenna to resolution 512×512 (RMSE=5.3526, PSNR=33.55 dB).

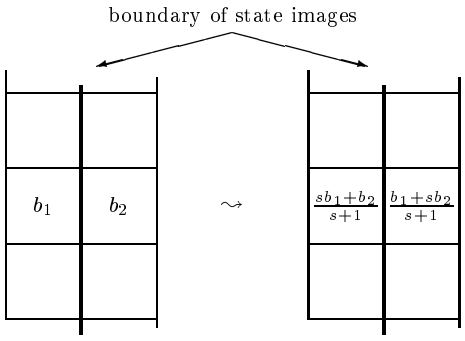


Figure 4.27: Edge smoothing with smoothing parameter s.

Despite the fact that this technique is often applied in fractal image compression [Fis95b, Haf99], an edge smoothing algorithm does not enhance the compression performance of the WFA codec. One reason for this behavior could be the exact cosine basis of our implementation. The smoothing operator could easily be extended to use a wider neighborhood of the edge, but due to disappointing results, we did not pursue this technique but moved on to the wavelet technique which should yield better results.

4.4.15 Coding with Varying Quality

It is a well-known fact that the content of an image is partitioned to *regions of interest* (for example the face of a portrait) and the background. It is an obvious idea that objects in the regions of interest should be coded more accurately than objects in the background. Since such regions often are part of the middle of the image, we implemented a scheme that adjusts the quality factor of the WFA encoder to a higher value than in other regions. See Figure 4.28 for a sketch of the regions with higher quality factor.

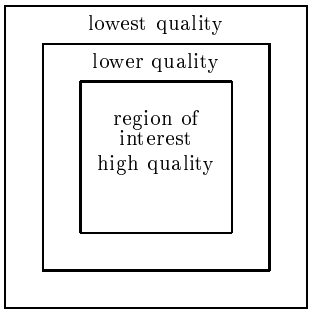


Figure 4.28: Image regions with different quality factors.

Note that this technique only affects the subjectively perceived quality. In PSNR sense, the quality is worsened. For this reason, we give no rate–distortion diagram of this modification but only a decoded image to visualize the result (see Figure 4.29).

The information whether or not a given path is contained in the inner region is determined by adding the first two symbols of the quadtree path. The path then lies in the inner region of the square iff that sum equals to 3 (see Figure 4.30). For a path shorter than 2, it is assumed for consistency to be in the region of interest. The next outer range (marked in Figure 4.28 with "lower quality") is determined in the same way by examining all combinations of 2-tuples of the first three symbols in the path. For the calculation in bintrees (bintree path $b = b_0 b_1 b_2 b_3 \ldots \in \{0,1\}^*$), two symbols have to be merged to become a quadtree address (quadtree path $q = q_0 q_1 q_2 q_3 \ldots \in \{0,1,2,3\}^*$) by the formula

$$q_i = 2 * b_{2i} + b_{2i+1} \tag{4.4}$$

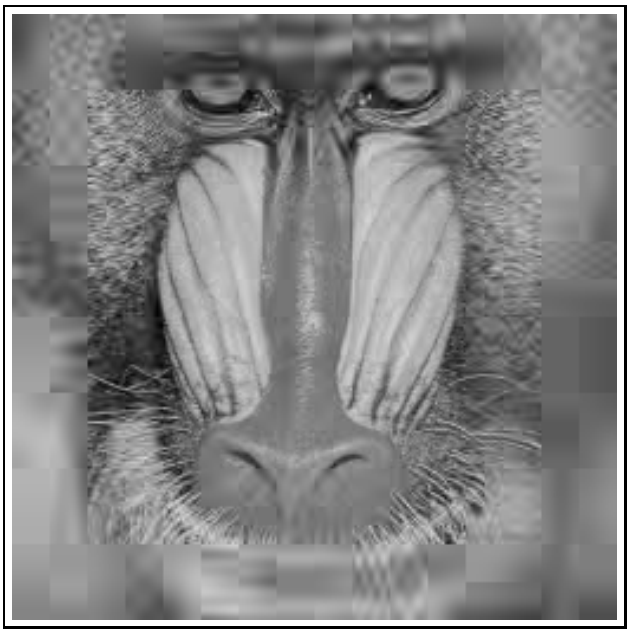


Figure 4.29: WFA decoded image with varying quality factor. Note that we have chosen an extremely low quality factor for the outer region to make the effect clearly visible.

and thereafter calculating the scheme considered above. Note that this scheme is not a total function, as it is not defined for odd length bintree paths, but needs to be calculated only for the first four and six symbols respectively. The corresponding segment is assumed to lie in the region of interest if the path is shorter.

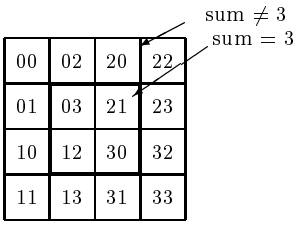


Figure 4.30: Determination of the inner square by using the path.

We did not develop this technique any further since the automatic determination of the region of interest is impossible due to the same reasons as the measurement of image quality (see appendix C on page 169). However, we think that this technique is valuable for special image types, for example portrait images.

4.4.16 Progressive Decoding of WFAs

In case of a slow data channel, the user may have to wait a long time until the decoded image can be seen. For that case, the *progressive decoding mode* of AutoPic is useful. In this mode, a coarse representation of the decoded image is shown first, which is refined successively. In order to quickly get a first approximation, we first transmit the DC, afterwards AC and finally MP coefficients. For an illustration of this concept see Figure 4.31. The WFA is fully decoded after receiving all DC, AC and MP coefficients. This feature has not yet been implemented to the user interface of AutoPic but is available only via the command line parameter `-progressiveNumberOfParts` in the class `WFADecoder`.

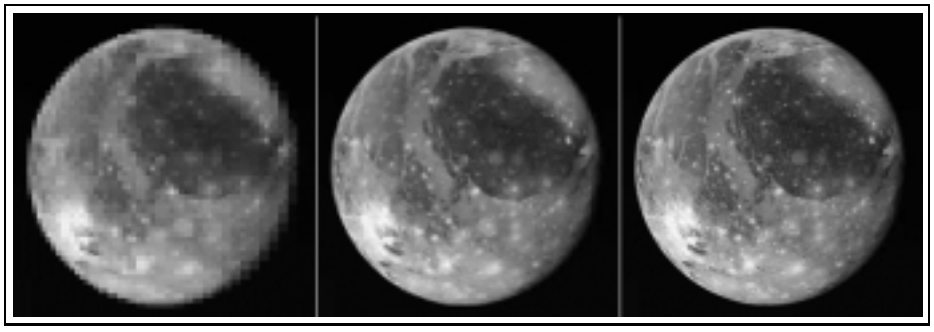


Figure 4.31: Progressive decoding of a WFA.

4.5 Optimization of Coding Parameters

The compression efficiency depends on a vast set of parameters. In the following enumeration we mention some of them, an example of the protocol file is given in section 7.3.1 on page 132.

- Choice of the *initial basis:* we have implemented the bases of cosine transform (also used in JPEG), sine, Hadamard, Slant and Walsh transform. We observed the best results with the cosine basis. Note that the basis functions have to be selected carefully since the images have to be "compatible", meaning that the conservativeness has to be ensured.
- *Quantization parameters:* signed or unsigned, precision before and after the dualpoint. These parameters have to be chosen for all three parts and sub-parts of the matching pursuit vectors.
- The *maximal sizes* of the AC and MP parts.
- *Adaptation speed, window lengths* and *context sizes* of the source models. These parameters have to be chosen carefully for each model since they are of main importance for compression efficiency. There are currently more than ten models affecting the compression performance.
- It has to be stored if the *light HV-partitioning, bad approximation, second chance matching pursuit* etc. has to be chosen.
- *Precision* of the *matching pursuit indices.* This parameter is crucial for the running time of the encoder. The parameter also has a great impact on time and space complexity.
- In the context of color coding, the type of *color model* and the *qualities* of the three color channels have to be encoded. Since the chrominance channels obey completely different statistical distributions, all parameters of the models have to be multiplied by 3.

4.5.1 Utilization of Genetic Algorithms

In order to manage such a huge number of parameters of the codec, the brute force approach for the optimization is infeasible. Therefore, we have utilized the method of genetic algorithms.

The genetic analyzer is adjusted so that at first a generation is created by using random genes. Afterwards the half with the worst genes are replaced by new genes obtained by crossing and mutating the best genes of that generation. For this task, two genes of the better half are selected randomly. The crossing operator selects a random cutting point and copies the bits from position zero to the cutting point from the first gene to the newly created gene. The remaining bits from the cutting point to the end are taken from the secondly selected gene.

For an illustration of the cross operator see Figure 4.32. After the creation of new genes by crossing, a given percentage of the gene pool are mutated, meaning that at random positions in the DNA code bits are inverted (see Figure 4.33). The obtained new generation is treated in the same way as the former generation. For an overview about these techniques see [SP94].

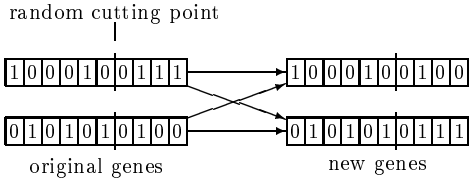


Figure 4.32: Cross operator.

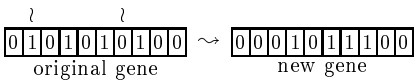


Figure 4.33: Mutate operator.

At the calculation of the fitness function arises the question what constitutes a good compression algorithm. We have drawn the conclusion that it is adequate to optimize the coding parameters with a given image at a predetermined quality. Therefore, a binary search is used which adjusts a quality factor so that the given quality is reached with little deviation. A problem of this search for the right quality factor is that the mapping from quality factor to image quality is of course not continuous, but surprisingly not even monotonic. These problems lead to heuristics in the binary search algorithm which searches a quality factor with which the WFA encoder achieves an image fidelity at least as high as desired, but has to be as close as possible to that limit.

A problem for the GA is the huge amount of parameters considered above. By encoding these parameters directly, we would require hundreds of bits for the representation of the DNA codes. For this reason, we encoded these parameters in a differential way. The informations in the DNA code are therefore added to certain offsets. Afterwards the GA is run and after "convergence" of the GA[3], the parameter offsets are adapted and the GA is started again.

Speeding up the Genetic Algorithm We have observed that the GA converges fastest if a cross rate of 60 percent is used and the best gene is retained in the gene pool. The mutation rate has to be raised from a low value at the beginning (two percent) to a high value (about eighty percent) if the process stagnates.

[3]In order to speed up the convergence, we first started the GA with an image size of only 256×256 and afterwards we started the GA using the normal test images with resolution of 512×512.

For a faster calculation of the genetic algorithm, we have developed a simple but efficient method to solve the problem in parallel on a cluster of computers. Because of the fact that the (by far) highest computing resources are required to calculate the fitnesses of the genes and only a small amount of resources are needed for the genetic algorithm, we store the function genes to fitnesses in a file and let the computers read this file periodically and write it back with the newly calculated genes. Under the assumption that the genes are calculated independently, we therefore have a speedup almost linear to the number of computers operating on the same image.

Another method for speeding up the gene calculation would be a master process and many slave processes on different computers. We have decided to use the heuristic considered above since it minimizes the communication overhead and is almost immune against breakdown of some processes.

The Utilization of Gray Codes In an early stage of optimization with GA, we used binary encoding for the differences. A significant speedup was achieved by using Gray codes instead. A possible explanation for this speedup is the property of Gray codes that adjacent codewords of a given codeword can be obtained by switching only a single bit.

4.6 Results

As there are no satisfying statistical models for images, the algorithm was tested with a number of well-known test images at a resolution of 512×512. The compression results depend on the image type. We target to compress "natural" images using a quadratic fidelity measure. Thus we have the same target as the well-known compression schemes JPEG and SPIHT which are utilized for comparison.

4.6.1 Utilized Test Images

In order to estimate the compression performance of the AutoPic WFA codec, we utilized several test images with a diverse set of characteristics. The characteristics vary from Lenna, a smooth image yielding high compression ratios, to Mandrill with an extensive high frequency domain. You can see the images[4] in Figure 4.34.

We have decided to test and optimize our image compressor using gray valued images because of the following reasons:

- We want to focus our attention on an image compression algorithm and not on color coding.

[4] The images can be downloaded at [Wat]. Another site for well-known test images is [USC].

- Most researchers give results in this form. Thus we can compare our results more easily with other compressors.
- Color spaces are often tuned to the human visual system. Thus statements in PSNR-sense are not visually meaningful. For problems with such distortion measures see appendix C on page 169.
- There is a wide variety of color spaces and the usage of such would make a comparison infeasible.
- One of the most frequently used color models today is the YC_bC_r model which is a relative modern model. Using such a model would make it impossible to compare an algorithm with older compressors.

One of the most popular test images is Lenna shown in Figure 4.34. Several variants of this image are in usage, thus making it hard to compare our results with others. In order to make the choice of images clear, we have used the widespread images of the Waterloo Fractal Image Coding Project [Wat].

Figure 4.34: Gray scale images of the Waterloo image site. The images are called (from left to right, top to bottom): Barb, Boat, Goldhill, Lenna, Mandrill, Peppers, Washsat, Zelda. In accordance with most publications concerning image compression, we left out the images Washsat and Zelda.

4.6.2 Some Decoded Images

We observed a tiling effect at high compression ratios, which commonly occurs at block-based algorithms. Alternatives for reducing this effect are smoothing of the edges or using a wavelet basis.

Already in an early stage of the implementation with a crude parameter setting, a relatively high compression performance was observed due to the feedback loop considered in section 3.7 on page 54. Some decompressed images are

shown in the Figures 4.35–4.36. For the interested reader, we added H-plots describing the encoding fidelity in a multi-resolution dependent manner (see section C.1.1 on page 171).

For comparison issues, we used the reference image compressors

- the JPEG implementation of the independent JPEG group as an example for an established codec[5] and
- SPIHT, a zerotree wavelet coder from W. A. Pearlman and A. Said representing one of the best image codecs available. This codec is described in [PS96] and is available at [PS].

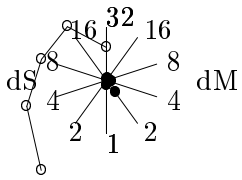


Figure 4.35: Decoded image Lenna and the corresponding H-plot (PSNR 32.89 dB, 0.2246 bpp, 880 states).

4.6.3 Absolute Running Times of the Codec

In order to present absolute running times of the WFA codec, we have made tests on a 500 MHz Pentium III computer with 128 MB RAM. We utilized the just in time compiler of IBM JDK 1.1.8. The probes for different data rates can be found in table 4.1.

[5] The current JPEG implementation is available at [IJG].

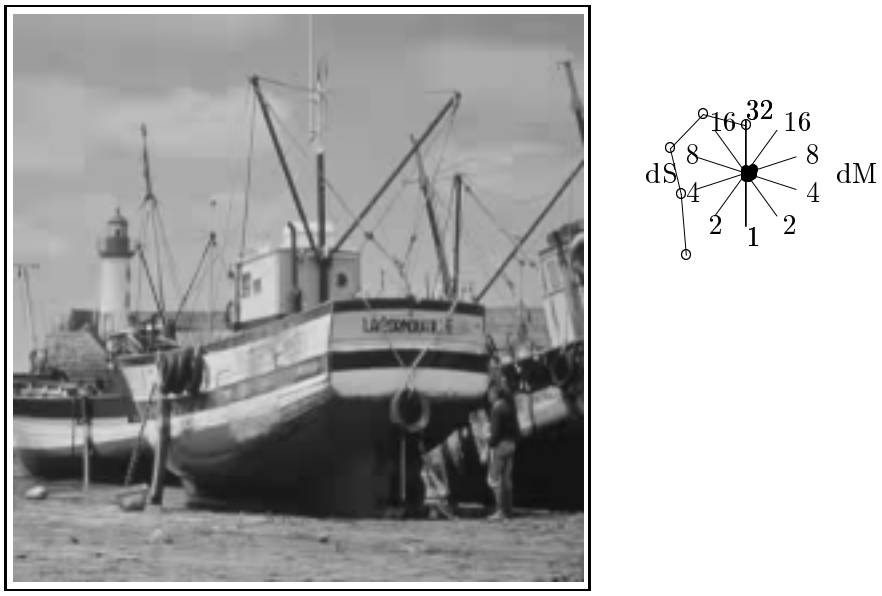


Figure 4.36: Decoded image Boat and the corresponding H-plot (PSNR 31.85 dB, 0.3344 bpp, 1276 states).

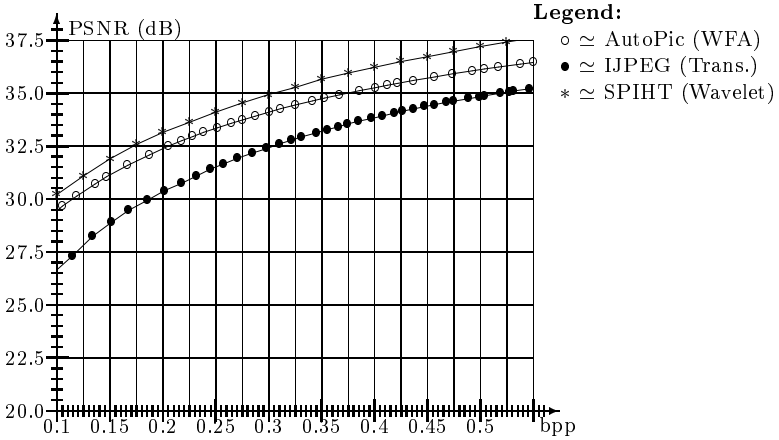


Figure 4.37: Rate–distortion diagram of the image Lenna.

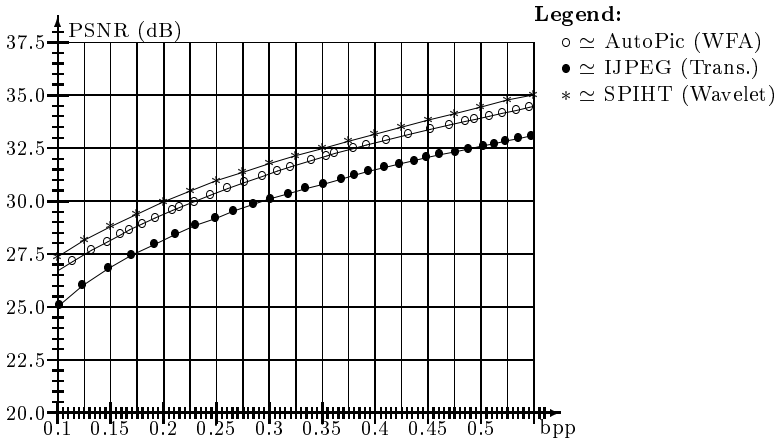


Figure 4.38: Rate–distortion diagram of the image Boat.

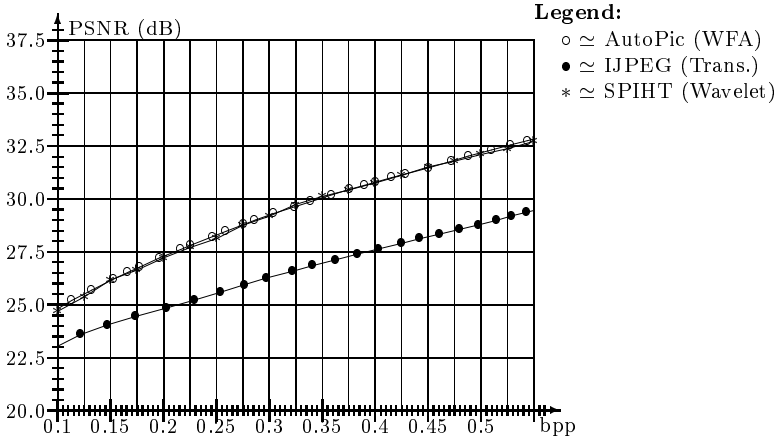


Figure 4.39: Rate–distortion diagram of the image Barb.

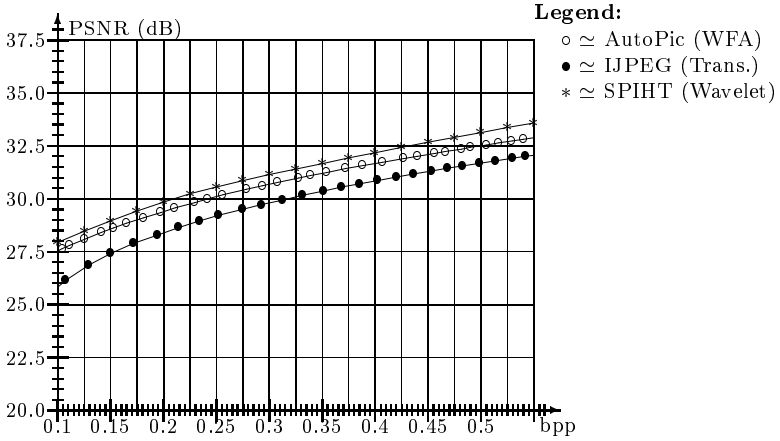


Figure 4.40: Rate–distortion diagram of the image Goldhill.

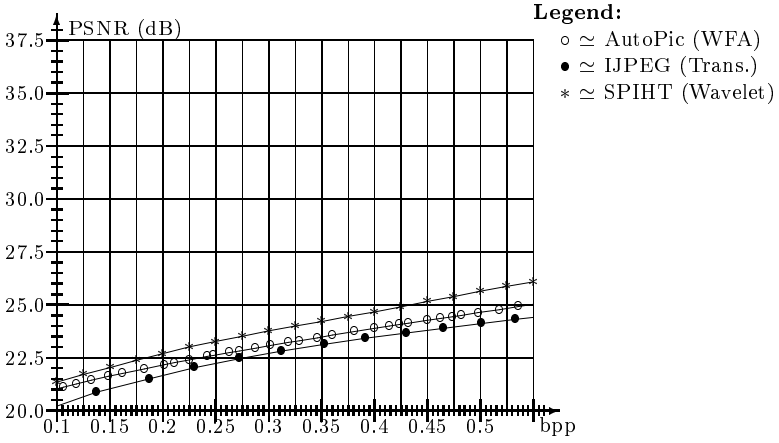


Figure 4.41: Rate–distortion diagram of the image Mandrill.

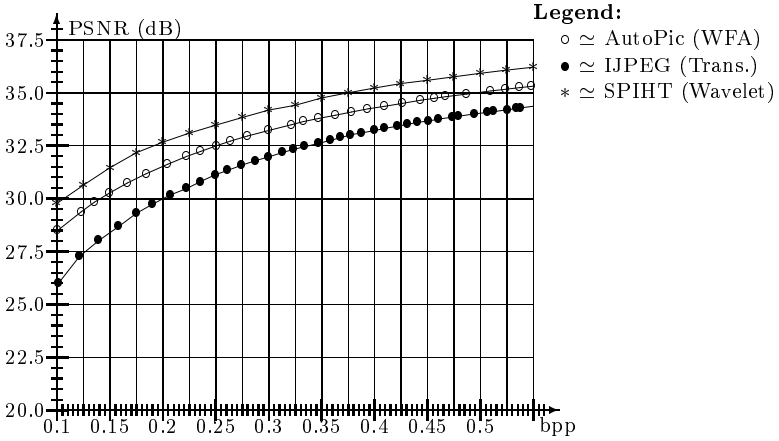


Figure 4.42: Rate–distortion diagram of the image Peppers.

data rate (bpp)	enc. time (sec.)	dec. time (sec.)
0.1396	50.6	0.64
0.1700	55.7	0.76
0.2057	60.1	0.84
0.2468	65.5	0.97
0.2750	66.0	1.05
0.3685	73.0	1.25
0.4626	78.1	1.57
0.5881	80.3	1.97

Table 4.1: Absolute running times of the WFA coder and decoder. The codec was tested with the test image Lenna.

4.7 Statistical Distributions of the Stored Parameters

The biggest part of the storage consumption of the encoded automaton have the indices of the matching pursuit vectors. For the coefficients, we detected distributions lying between Laplace distribution and normal distribution. For the lengths of the parts we detected a normal distribution. The program X TREMES [RT97] was used for the statistical analysis. On the following pages some diagrams for statistical analysis of the images Lenna and Boat are given, two well-known test images with fairly different statistics. The statistical data can be evaluated for designing quantizers and approximation algorithms. The diagrams show statistical parameters for DC, AC and MP coefficients. Figure 4.43 shows the distribution all DC coefficients at storage time for the image Lenna. Figure 4.44 shows deviations of the AC coefficients of the MP vectors in the first positions and Figure 4.45 shows the means of the individual AC coefficients. Afterwards statistical data of the MP coefficients are given. Note that several other statistical data could be interesting, for example the coefficients that are rolled out of the WFA or the distributions of the MP indices and data for tree construction. We have printed only the most important diagrams. More diagrams may be requested directly from the author of this thesis.

4.8 Further Research Topics

The research of WFAs is not completed by far. In this section, we list some of the outstanding problems to be investigated.

4.8.1 Embedding GA to WFA

An interesting but at the moment infeasible modification of the WFA coding algorithm is the inclusion of the GA. The procedure works in the way that the WFA encoder is first adjusted to the input image using the GA, and afterwards transmitting the WFA using the optimized parameters. For this task, a suitable abortion criterion has to be stated. This could be that the best fitting gene has not changed in the last x generations. The DNA code thus obtained could then also be transmitted with the image. The acquired storage overhead may be compensated by a better coding fidelity since the DNA code consists in practice only of a few bytes. The decoder would also have to adjust to those parameters and afterwards decode the automaton. The main problem with that method are the extreme running times since the WFA construction process has to be performed several times.

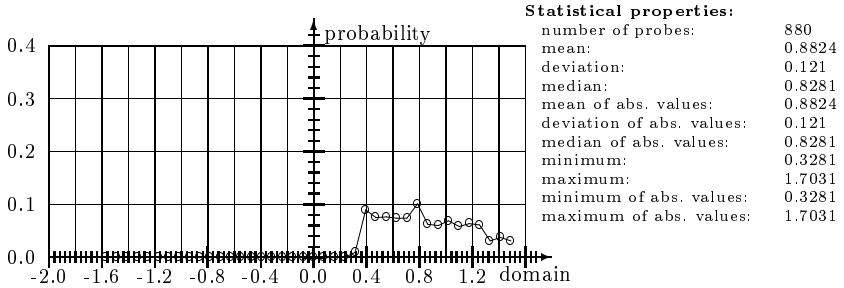


Figure 4.43: Distribution of DC coefficients at storage time (Lenna).

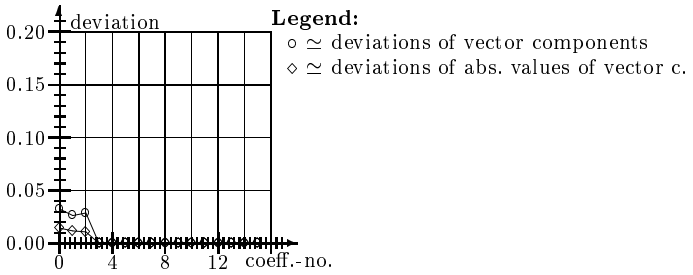


Figure 4.44: Deviations of AC coefficients at storage time (Lenna).

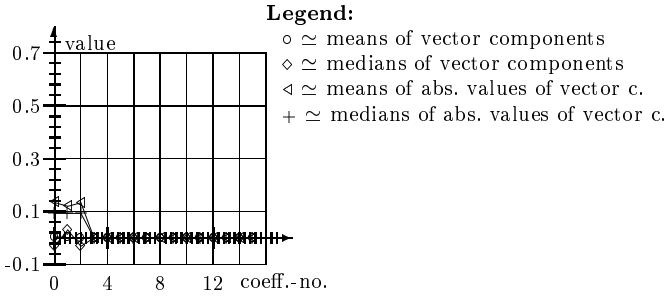


Figure 4.45: Means and medians of AC coefficients at storage time (Lenna).

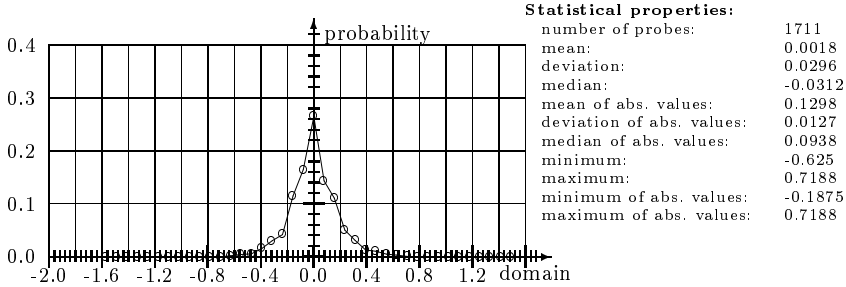


Figure 4.46: Distributions of AC coefficients at storage time (Lenna).

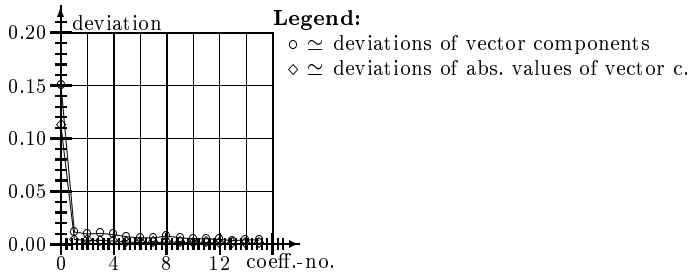


Figure 4.47: Deviations of MP coefficients at storage time (Lenna).

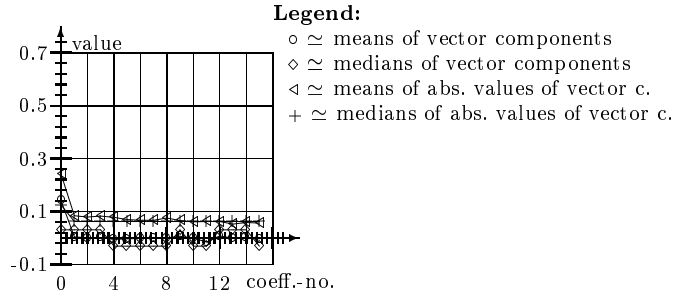


Figure 4.48: Means and medians of MP coefficients at storage time (Lenna).

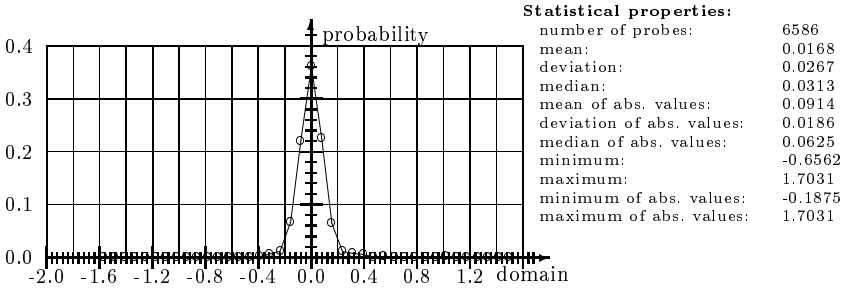


Figure 4.49: Distributions of MP coefficients at storage time (Lenna).

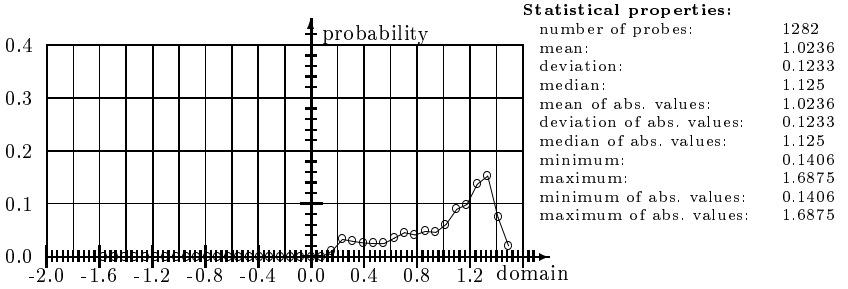


Figure 4.50: Distribution of DC coefficients at storage time (Boat).

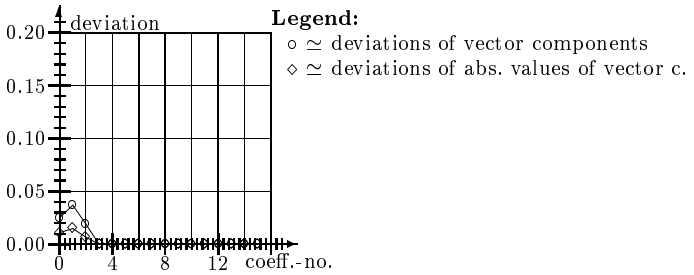


Figure 4.51: Deviations of AC coefficients at storage time (Boat).

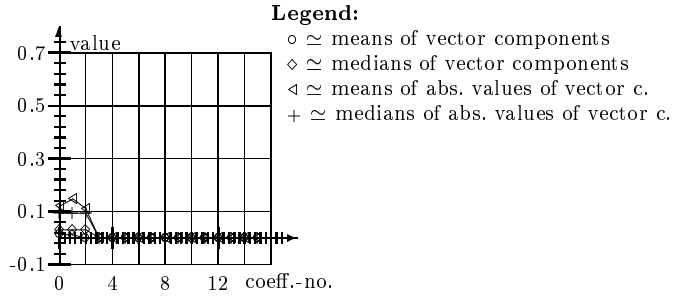


Figure 4.52: Means and medians of AC coefficients at storage time (Boat).

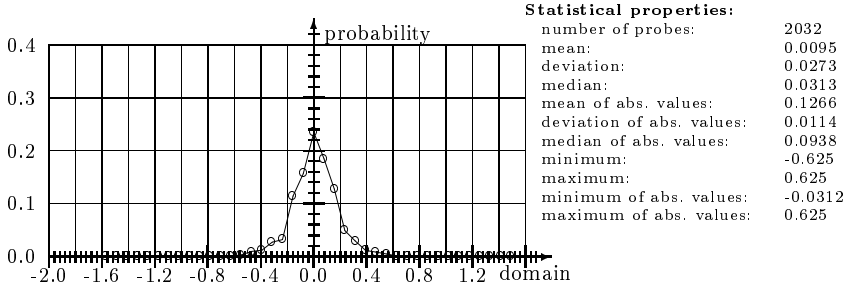


Figure 4.53: Distributions of AC coefficients at storage time (Boat).

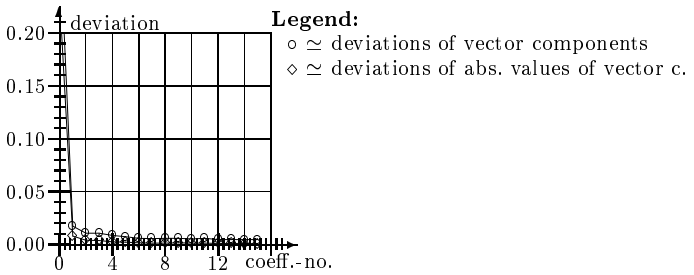


Figure 4.54: Deviations of MP coefficients at storage time (Boat).

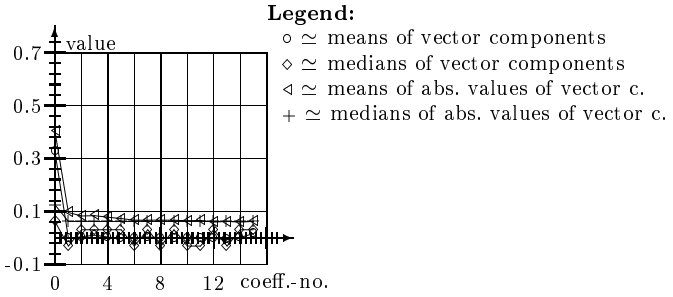


Figure 4.55: Means and medians of MP coefficients at storage time (Boat).

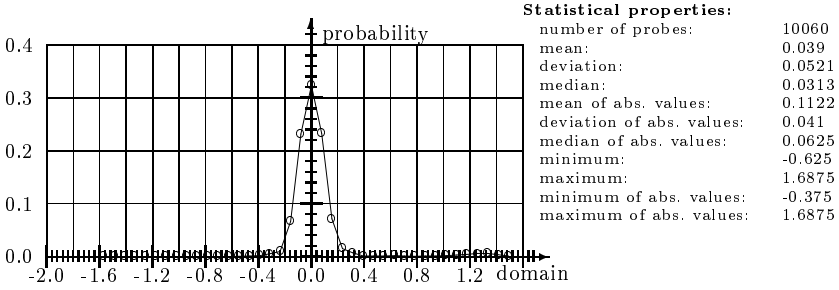


Figure 4.56: Distributions of MP coefficients at storage time (Boat).

4.8.2 Calculation of the Color Space

An interesting optimization could be made to color coding. In a similar way to the Hotelling transform (see section E.4.1 on page 184), a preprocessing step to the WFA encoder could be the calculation of an own color space decorrelating the color channels better than a hard-wired color transform like the YC_bC_r transform.

The drawback of this method is the computational overhead. This is a small time punishment since the main calculation is the determination of the eigenvectors of a 3×3 matrix. Another drawback is that the matrix coefficients and the means would have to be transmitted to the decoder. We think that this transmission overhead of the twelve values is only some bytes if the difference to a hard-wired transform is transmitted.

We think that these drawbacks could easily be compensated by an improved decorrelation of the color transform. However, this method has not been implemented up to date.

4.8.3 WFAs in Use for Pattern Recognition

The WFA construction algorithm may be utilized for image pattern recognition. Since the WFA construction procedure collects huge image segments, the automaton structure could give resolution independent clues about the image content. For this task a neural network may be used. This network is fed with automaton parameters and trained by supervised learning to distinguish, for example, indoor images from outdoor images.

4.8.4 Postprocessing of WFAs

An interesting option is the postprocessing of WFAs. Since all state images can be accessed after construction of a WFA, the reorganization of the edges can be performed to approximate state quadrants using state images constructed after the processed state. This technique seems to be complicated since the skeleton of the WFA is no longer a tree but a general directed graph. Special techniques will have to be applied if circles (loops) occur. The postprocessing of WFAs has not been implemented yet.

4.8.5 Near Lossless Coding

An interesting variant of lossy image coding is *near lossless coding* which applies maximum norm instead of the squared norm. This kind of coding is especially interesting in medical image processing where the image encoder has to guarantee that the reconstructed pixel values do not deviate more than a given threshold. In [BW95] such a scheme has been successfully implemented using WFAs.

4.8.6 Asymmetric Running Times of the Coding and Decoding Algorithm

It is clear that due to the backtracking algorithm and calculation of scalar products (which are only needed to be performed for the encoding process), there is a strong asymmetry respective the running times of coding and decoding an image. With the inclusion of GA this asymmetry would be much more biased, since the DNA code would only have to be decoded at the decoding side. This property of the WFA codec can be used in WORM applications[6].

Image Compression as a Client Server Application

Because of the high resource requirements of the WFA encoding algorithm, we considered the client server implementation of the application. It is conceivable that a (typically slow) client uses the hardware resources of a (typically fast) server. The client would therefore send an image in a typical image format as the Targa format via a data conduction to the server computer. The server computer would convert the image to the WFA format and send it back to the client computer.

The usage of the decoding algorithm in the client server application is also conceivable, but this algorithm can usually be computed by a computer with low resources. The AutoPic manual on page 125 describes how it can be used as a simple client server application.

4.8.7 Adaptation of Coding Parameters to the Image Content

It is conceivable to examine the image content before the coding process. This information could be used for the adaptation of some coding parameters. This approach has not yet been investigated.

4.8.8 Combination of the WFA Coding Algorithm with the Laplacian Pyramid

An early version of multi-resolution analysis is the famous image coder of Burt and Adelson [BA83]. This technique utilizes special low and high pass filters combined to provide an estimation for the current image. The low pass filter produces an image of half resolution of the original image. Using the corresponding high pass filter, the original image can be approximately reconstructed. This technique of resolution decimation is applied recursively to build a pyramid as shown in Figure 4.57.

Now we sketch the reconstruction[7] algorithm. A step in the *reconstruction process* is performed as follows:

[6] WORM $\simeq$ write once read many.

[7] We describe the reconstruction algorithm instead of the construction algorithm since the effect of error propagation is more clearly visible in this process.

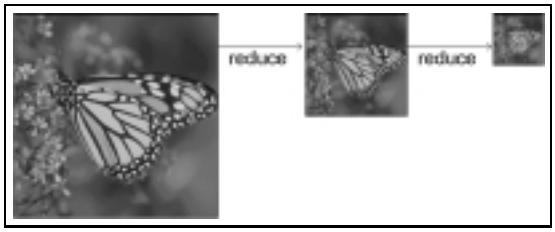


Figure 4.57: Construction of the Laplacian pyramid.

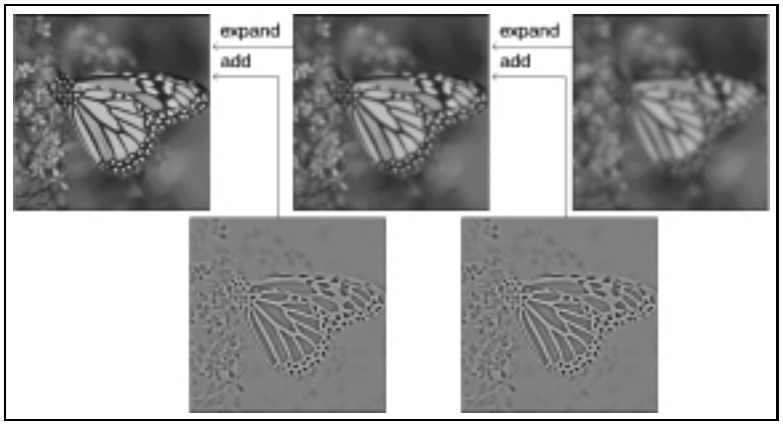


Figure 4.58: Reconstruction of the original image.

- First the current image is scaled up by using the expand operator.
- The remaining error is transmitted using a WFA. The decoded image is added to the up-scaled image.

This procedure is repeated until the entire resolution is accomplished. For an illustration of this process see Figure 4.58. Note that the expanded image is not exactly the same as obtained by down-sampling the image of the pyramid level above because of losses in former pyramid stages. This effect has to be diminished by a feedback loop in the construction algorithm to obtain optimal results.

We have experimented with this kind of coding but observed that this method does not erase the tiling effect entirely. Another drawback of this method is that the amount of data to encode by the WFA encoder is blown up. For these reasons we decided to use wavelet techniques instead. These techniques are explained more thoroughly in chapter 5 on page 103. See [BA83, Ohm95] for further details concerning Laplacian pyramids.

4.8.9 Incorporation of Image Transforms to WFAs

A promising add-on to WFAs is the incorporation of allowing the state images to be transformed in various manners. In [CR96] all rotations by 90 degrees,

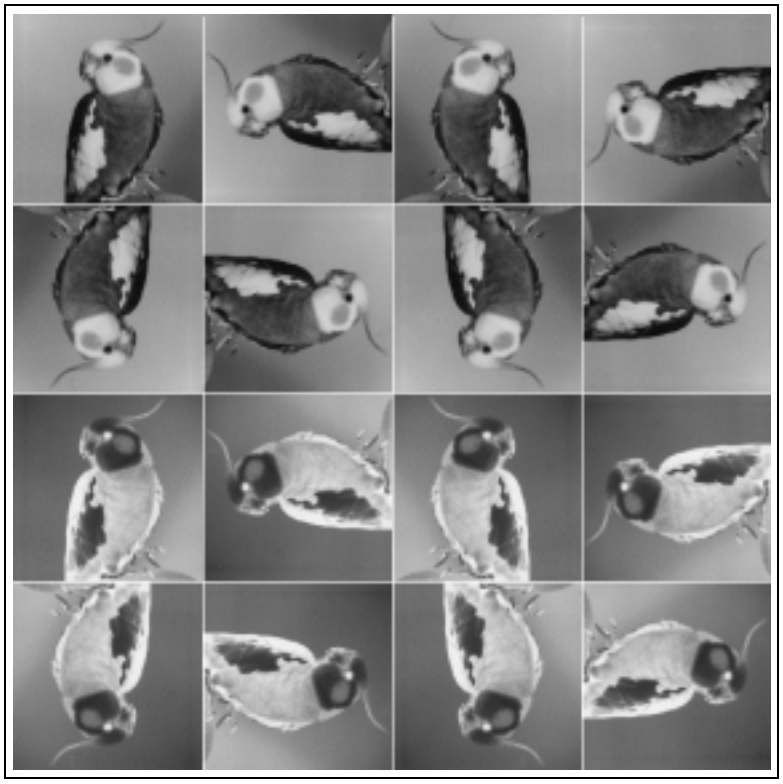


Figure 4.59: Image transforms used in [CR96].

mirroring and negation were allowed. These transforms may be combined to yield the 16 image transforms visualized in Figure 4.59. This generalization is called *GWFA* (generalized weighted finite automaton). We have not implemented this technique yet.

Chapter 5

Combining WFAs with Wavelets

Since the blocking artifacts are probably the most severe problem in WFA coding, we have to think about the reduction of this phenomenon. A frequently used technique is the coding with wavelets. In this chapter, we shortly introduce some of the main ideas concerning wavelet analysis and show that this technique may be embedded efficiently in the WFA encoder of AutoPic.

5.1 Introduction

Wavelets are function systems $\{\Psi_i\}$ used to decompose a function f similar to linear transforms:

$$f = \sum_i a_i \Psi_i. \tag{5.1}$$

The "smoothness" of wavelets is often characterized by the number of vanishing moments. A function f defined over an interval $[a, b]$ is said to have n *vanishing moments* iff

$$\int_a^b f(x) x^i \, dx \tag{5.2}$$

vanishes for $i \in \{0, 1, \ldots, n-1\}$. Wavelets with a high number of vanishing moments can approximate polynomials of a high degree and thus are well suited for approximating smooth signals that occur often in image processing.

Multi-Resolution Analysis Consider the vector space of square Lebesgue-integrable functions in $\mathbb{R}$:

$$\mathrm{L}^2(\mathbb{R}) = \left\{ f : \int_{-\infty}^{+\infty} f^2(x) \, dx < \infty \right\}. \tag{5.3}$$

From now on, we will consider functions differing only on a set of (Lebesgue-) measure zero as equivalent. We may introduce a sequence of closed subspaces of the space $\mathrm{L}^2(\mathbb{R})$ such that the subspaces V_j with $j \in \mathbb{Z}$ are nested

$$\ldots \subset V_{-2} \subset V_{-1} \subset V_0 \subset V_1 \subset V_2 \subset \ldots, \tag{5.4}$$

($V_{j-1} \neq V_j \, \forall j \in \mathbb{Z}$) their union

$$\bigcup_{j=-\infty}^{+\infty} V_j \tag{5.5}$$

is dense in $\mathrm{L}^2(\mathbb{R})$ and their intersection contains only the constant function zero, i.e.

$$\bigcap_{j=-\infty}^{+\infty} V_j = \{0\}. \tag{5.6}$$

A decomposition with the properties 5.4, 5.5 and 5.6 is henceforth called *MRA* (multi-resolution analysis).

We consider only *dyadic wavelets* meaning that they fulfill the following equations:

$$v(x) \in V_j \iff v(2x) \in V_{j+1} \quad (\textit{dilation}), \tag{5.7}$$

$$v(x) \in V_0 \iff v(x+1) \in V_0 \quad (\textit{translation}). \tag{5.8}$$

Definition 5.1.1 *A function $\Phi \in V_0$ with the property that*

$$\{\Phi(x-k) : k \in \mathbb{Z}\} \tag{5.9}$$

forms a basis for V_0 is called father function *or* scaling function. *For V_j with $j \neq 0$ we define*

$$\Phi_{j,k}(x) = 2^{j/2}\Phi(2^j x - k). \tag{5.10}$$

Since the subspaces V_j are nested, we may decompose a subspace V_{j+1} into V_j and the orthogonal complement of V_j in V_{j+1}, denoted by W_j:

$$V_j \oplus W_j = V_{j+1}, \tag{5.11}$$

$$V_j \perp W_j. \tag{5.12}$$

Thus a signal's representation in the space V_j can be considered as a "low resolution" or "low pass" representation of a signal in V_{j+1}. Analogously W_j can be interpreted as the remaining "high resolution" or "high pass" part lost by transition from V_{j+1} to V_j.

Since the union of the V_j is dense in $\mathrm{L}^2(\mathbb{R})$ we can see that

$$\overline{\bigoplus_{j=-\infty}^{+\infty} W_j} = \overline{\bigcup_{j=-\infty}^{+\infty} V_j} = \mathrm{L}^2(\mathbb{R}). \tag{5.13}$$

Analogously to definition 5.1.1 we now define the wavelet function:

Definition 5.1.2 *A function $\Psi \in W_0$ with the property that*

$$\{\Psi(x-k) : k \in \mathbb{Z}\} \tag{5.14}$$

forms a basis for W_0 is called mother function *or* wavelet function. *For V_j with $j \neq 0$ we define the basis functions*

$$\Psi_{j,k}(x)=2^{j/2}\Psi(2^j x-k). \tag{5.15}$$

5.2 The Filter Bank Algorithm

Since both V_0 and W_0 are contained in V_1, we can express Φ and Ψ with basis functions of V_1 as

$$\Phi = 2\sum_k h_k \Phi_{1,k}, \tag{5.16}$$

$$\Psi = 2\sum_k g_k \Phi_{1,k}. \tag{5.17}$$

Due to the MRA property, we conclude that V_j and W_j are contained in V_{j+1} and so these equations hold for arbitrary j.

The coefficients g_k and h_k are called *filter coefficients* and uniquely define the functions Φ and Ψ. We examine only transforms where the number of coefficients of the filters g and h is finite, called *FIR* (finite impulse response) *filters*. For wavelets defined in this way a fast transform algorithm, the *filter bank algorithm*, can be applied.

Now assume that f is a function represented by coefficients for the basis functions of V_{j+1}. Since $V_{j+1} = V_j \oplus W_j$, we may write uniquely

$$f = \sum_k \lambda_{j+1,k}\Phi_{j+1,k} \tag{5.18}$$

$$= \sum_l \lambda_{j,l}\Phi_{j,l} + \sum_l \gamma_{j,l}\Psi_{j,l}. \tag{5.19}$$

with [SS98]

$$\lambda_{j,l} = \sqrt{2}\sum_k h_{k-2l}\lambda_{j+1,k}, \tag{5.20}$$

$$\gamma_{j,l} = \sqrt{2}\sum_k g_{k-2l}\lambda_{j+1,k}. \tag{5.21}$$

Since the number of filter coefficients is fixed and the amount of data is halved at each step, the number of operations performed is linear to the length of the input.

5.3 Orthogonal Wavelets

Early wavelet transforms used *orthogonal wavelets* satisfying the conditions

$$V_j \perp W_j, \tag{5.22}$$
$$\langle \Phi_{j,l}, \Phi_{j,l'} \rangle = \delta_{l,l'}, \tag{5.23}$$
$$\langle \Psi_{j,l}, \Psi_{j',l'} \rangle = \delta_{j,j'}\delta_{l,l'}. \tag{5.24}$$

We can calculate the required coefficients of the analysis

$$f = \sum_l \lambda_{j,l}\Phi_{j,l} + \sum_l \gamma_{j,l}\Psi_{j,l} \tag{5.25}$$

with inner products

$$\lambda_{j,l} = \langle f, \Phi_{j,l} \rangle \text{ and } \gamma_{j,l} = \langle f, \Psi_{j,l} \rangle. \tag{5.26}$$

5.4 Biorthogonal Wavelets

In order to gain more flexibility, the constraint of orthogonality may be relaxed. With *biorthogonal wavelets* we have two multi-resolution analyses, *primal* and *dual* analysis. We mark the dual counterpart by using a tilde:

- primal analysis: V_j, W_j, $\Phi_{j,k}$, $\Psi_{j,k}$,
- dual analysis: $\tilde{V}_j$, $\tilde{W}_j$, $\tilde{\Phi}_{j,k}$, $\tilde{\Psi}_{j,k}$.

The following conditions have to be satisfied:

$$\tilde{V}_j \perp W_j, \tag{5.27}$$
$$V_j \perp \tilde{W}_j, \tag{5.28}$$
$$\langle \tilde{\Phi}_{j,l}, \Phi_{j,l'} \rangle = \delta_{l,l'}, \tag{5.29}$$
$$\langle \tilde{\Psi}_{j,l}, \Psi_{j',l'} \rangle = \delta_{j,j'}\delta_{l,l'}. \tag{5.30}$$

We can calculate the required coefficients using scalar products using the dual basis functions

$$\lambda_{j,l} = \langle f, \tilde{\Phi}_{j,l} \rangle \text{ and } \gamma_{j,l} = \langle f, \tilde{\Psi}_{j,l} \rangle. \tag{5.31}$$

Because of this construction, we may use the filter bank algorithm using the dual filter pair $(\tilde{h}, \tilde{g})$ for the wavelet transform and the primal filter pair (h, g) for the reconstruction.

5.5 The Wavelet Decomposition Tree

Since each subspace V_j can be split in two subspaces V_{j-1} and W_{j-1}, we may decompose the highest resolution subspace V_0 recursively as

$$\begin{aligned} V_0 &= V_{-1} \oplus W_{-1} \\ &= V_{-2} \oplus W_{-2} \oplus W_{-1} \\ &= V_{-3} \oplus W_{-3} \oplus W_{-2} \oplus W_{-1} \\ &\vdots \end{aligned} \tag{5.32}$$

thus yielding a decomposition tree as illustrated in Figure 5.1.

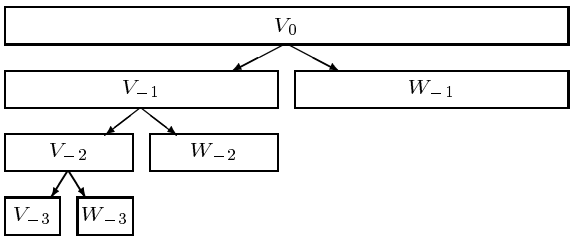


Figure 5.1: A wavelet decomposition tree.

Since the amount of data operated on is halved at each level, the complexity of the full wavelet transform is still $O(n)$.

5.6 Generalization to Higher Dimensions

Since we aim at utilizing wavelets for image coding, we have to perform the wavelet transform in two dimensions. Similarly to the DCT in the JPEG image compression standard, we perform the transform independently for each dimension. Analogously, we write the wavelet transform as a transform matrix and due to associativity of the matrix multiplication, the order of the two transforms is not important. This procedure is equivalent to a two dimensional wavelet transform where the basis functions are tensor products of the according one dimensional basis functions. After one transformation step we use the basis functions

$$\begin{aligned} &\Phi \otimes \Phi, \quad \Phi \otimes \Psi, \\ &\Psi \otimes \Phi, \quad \Psi \otimes \Psi. \end{aligned} \tag{5.33}$$

There are also techniques for constructing multidimensional wavelets directly, which we do not consider here (see [SJ99]).

5.7 The Wavelet Packet Transform

In contrast to the ordinary wavelet transform, the *wavelet packet transform* splits not only the low frequency part of the data but also the high frequency part to two components. This procedure produces a decomposition tree as shown in Figure 5.2. Note that the time complexity of the wavelet packet transform is raised to $O(n \log n)$ compared to the ordinary wavelet transform.

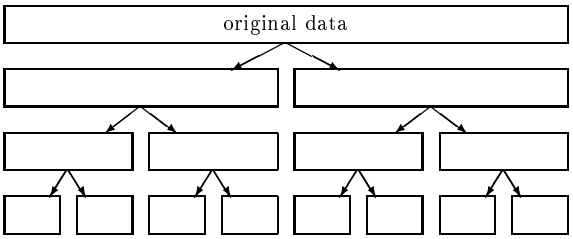


Figure 5.2: A full wavelet packet decomposition tree.

Since the wavelet packet decomposition tree can be cut, the effect of the wavelet packet transform is a search for the "best basis", a set of basis functions decorrelating the input data the most. For an illustration of a cut wavelet package tree see Figure 5.3.

In WFA coding, the decision whether to split the data could also be performed in a rate–distortion constrained manner. This may be achieved by comparing the resulting badnesses of the decisions. This technique seems to be highly promising for the future of research concerning WFA coding.

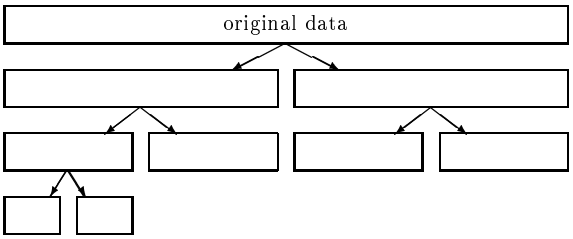


Figure 5.3: A cut wavelet packet decomposition tree.

5.8 The Lifting Scheme

In this section we introduce the *lifting scheme*, an efficient technique for calculating the wavelet transform. In contrast to ordinary wavelet techniques, the lifting scheme does not rely on Fourier transform and is thus easier to implement. We introduce the lifting scheme only for the one dimensional case. The generalization to the multidimensional case and wavelet packets may be performed using the same techniques as described in sections 5.6 and 5.7.

5.8.1 The Lazy Wavelet Transform

As we have already seen, at each level of the wavelet transform the data is split into two parts, the low and the high pass part. These parts are obtained by applying the corresponding wavelet filters.

Let us denote the low resolution part of the level $j+1$ by λ_{j+1}. This part is split into the low and high pass parts of the next level, called λ_j and γ_j, respectively. This division is performed by splitting the data set λ_{j+1} to even and odd samples. This step is commonly called the *lazy wavelet transform*. Note that this transformation step does not decorrelate the input data.

5.8.2 Primal and Dual Lifting

The decorrelation of the input data is performed by the application of a sequence of *lifting steps*. A *dual lifting step* does the same as predictor coding: the data elements of γ_j are predicted by data elements in λ_j. Thus γ_j is replaced by $\gamma_j - \mathcal{P}(\lambda_j)$, where $\mathcal{P}$ is a *prediction function* depending on the data of λ_j.

It is clear that the dual lifting step alters some properties of the data (for example the mean value) which eventually should be conserved. In order to restore such properties, a second step named *primal lifting step* is performed where the set λ_j is updated with the new set γ_j. We write this step by replacing λ_j with $\lambda_j + \mathcal{U}(\gamma_j)$, where $\mathcal{U}$ is called an *update function*.

In general, a sequence of dual and primal lifting steps may be performed in order to get from level $j+1$ to level j. The whole procedure of a wavelet transformation step using lifting is shown in listing 5.1.

```
/**
* performs a step in a wavelet construction.
*
*@param λ_{j+1} is the data set to be transformed.
*@return the next level in the transform.
**/
dataSetTuple transformLevel(dataSet λ_{j+1})
  {
  (λ_j, γ_j)=split(λ_{j+1}); //perform lazy wavelet transform.
  for (i=0; i<numberOfLiftingSteps; i++)
    {
    γ_j -= P_i(λ_j); //perform dual lifting.
    λ_j += U_i(γ_j); //perform primal lifting.
    }//end for
  return (λ_j, γ_j); //return the next level.
  }//end transformLevel
```

Listing 5.1: Wavelet lifting step.

An inverse transformation step is done by simply reverting the steps considered above: revert the order of the lift operations, invert the signs in the lifting steps and perform merging instead of splitting.

The lifting scheme may be interpreted as a generalization of the ordinary[1] wavelet transform, since these kinds of transforms can be rewritten to lifting steps by "factoring out" the FIR filters with the Euclidean algorithm (see [DS97] for details). More details about the lifting scheme may be found in [SJ99].

5.9 Combination of WFAs and Wavelets

The procedure uses the WFA coder as a quantization scheme for wavelet coefficients as follows:

- First the original image (Figure 5.6) is transformed using a wavelet transform (Figure 5.7).
- The transformed image is coded as described above with the WFA coding algorithm and this WFA is transmitted.

The decoder first decodes the WFA and afterwards applies the inverse wavelet transform. The composition of this new codec is illustrated in Figure 5.4. A reconstructed image using the combined codec is shown in Figure 5.8.

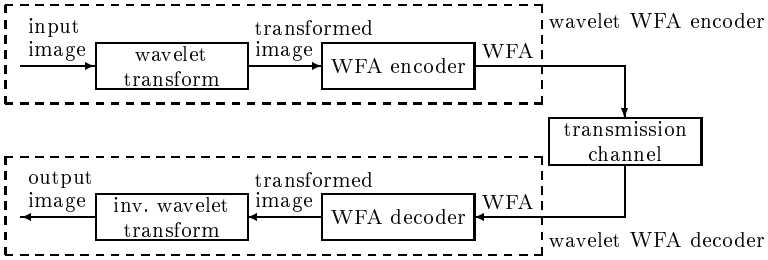


Figure 5.4: Combination of wavelet and WFA coding.

For the use of wavelet transforms in conjunction with WFA coding, we have to pose some constraints to the wavelets we use:

- The first requirement is the *perfect reconstruction property*: since the combined WFA coder should operate at arbitrary high fidelity, it is necessary that the wavelet transform should not introduce errors, besides quantization errors produced by rounding errors in floating point arithmetic.
- The boundaries of the high and low pass parts of the transformed data should coincide with the edges of the tiles produced by the WFA algorithm. Since our implementation uses light HV partitioning (and not HV partitioning), this property is obtained by using dyadic wavelets.

[1] With "ordinary wavelet transforms" we denote transforms defined by FIR filters.

- Since one of the most important aspects of the wavelet decomposition is reducing tiling effects, the wavelets should be as smooth as possible. This smoothness is characterized by the number of vanishing moments, which should be as high as possible. Such smooth wavelets are well suited to approximate smooth areas in "natural" images.

- The wavelet transform should only produce a small computational overhead. This is the main reason why we implemented the wavelet transform using the lifting scheme instead of finite impulse response filters. Surprisingly, the computational overhead produced by the wavelet transform is more than compensated by the fact that the WFA encoder is sped up, since the image detail is concentrated to a small portion of the transformed image. However, the computational overhead does slow down the WFA decoder.

- The wavelet transform should be calculated in-place, meaning that no auxiliary memory is allocated. This property is supported by the lifting scheme considered above and is crucial since image data often requires a huge amount of internal computer memory.

- The decomposition in the wavelet basis should be stable: if the function f is changed slightly, the resulting wavelet coefficients should also vary only slightly. This constraint is fulfilled by biorthogonal wavelets since they form a *Riesz basis* meaning that there exist constants $A, B \in \mathbb{R}_+$ such that

$$A||f|| \leq \sum_l \lambda_{j,l}^2 + \sum_l \gamma_{j,l}^2 \leq B||f||. \tag{5.34}$$

  Note that equation 5.34 reduces to Parseval's identity (see section E.2.4 on page 183) for the case $A = B = 1$. This special case is fulfilled for orthogonal wavelets.

In cause of these requirements, we decided to use wavelets obtained by the well-known "(9-7)" filter pair named by the fact that the associated analysis filter $\tilde{h}$ uses nine coefficients while the synthesis filter h is built by seven coefficients. A graph of the (9-7) wavelet function is shown in Figure 5.5. See [DS97] for an efficient implementation using the lifting scheme. The implementation is approximately 64 percent faster[2] than the ordinary computation with the filter bank algorithm.

Nevertheless, it is not yet clear which wavelet transform is "best" suited for WFA coding. This topic seems to be an interesting area for further research in WFA coding. For more details about wavelet transforms see [Cod92, Mal98, PS93].

[2]The elementary operations counted in this analysis are the sum of additions and multiplications.

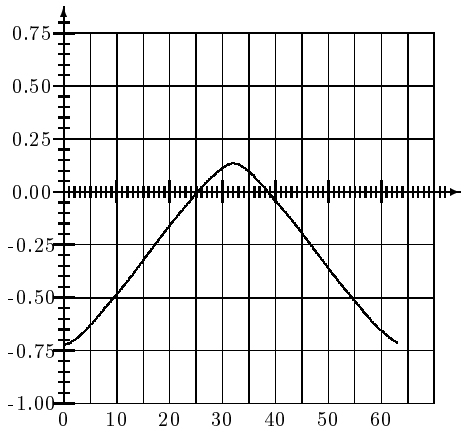


Figure 5.5: (9-7) wavelet function.

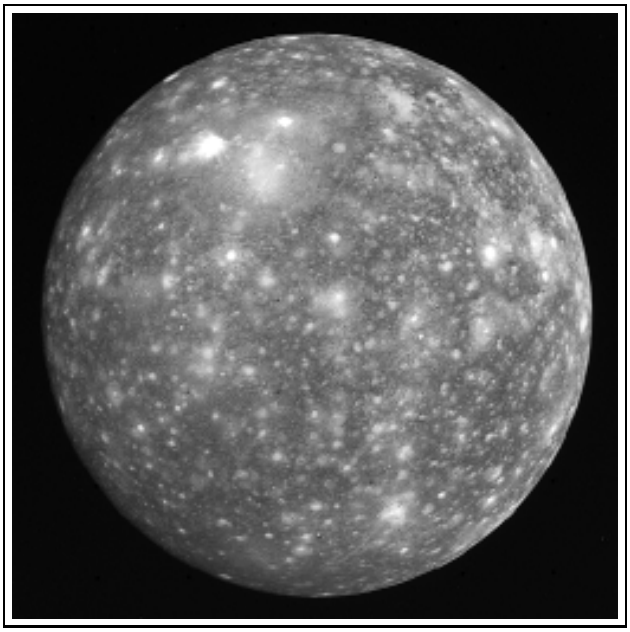


Figure 5.6: Image Callisto.

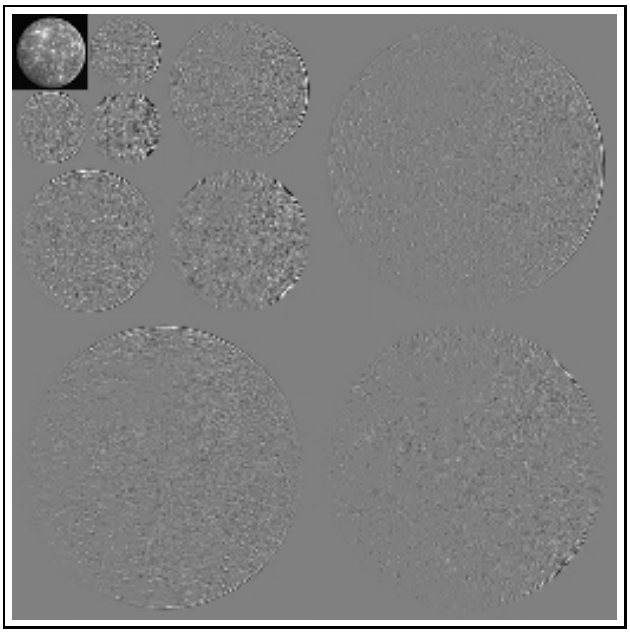


Figure 5.7: Wavelet transformed image Callisto. The values of the outer regions were added to a constant 128 to also visualize negative values.

Figure 5.8: Reconstructed image Callisto.

Progressive Mode In the same way as shown in section 4.4.16 on page 84, progressive decoding of wavelet enhanced WFAs is possible. See Figure 5.9 where the progressive mode is shown in the wavelet enhanced mode.

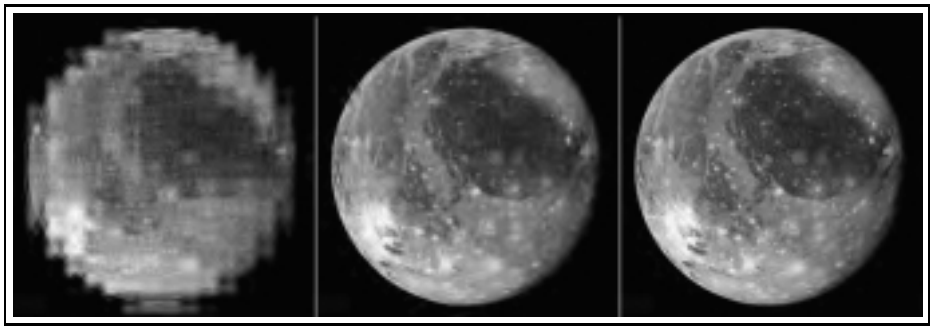


Figure 5.9: Progressive decoding of a WFA in wavelet mode.

Results In most cases the compression results using the combined WFA encoder are worse than the standard mode. We think that this mode is an interesting area for further research since blocking artifacts are diminished and the encoding speed is enhanced. For some rate–distortion diagrams comparing the WFA encoder with and without wavelet mode see Figures 5.10–5.13.

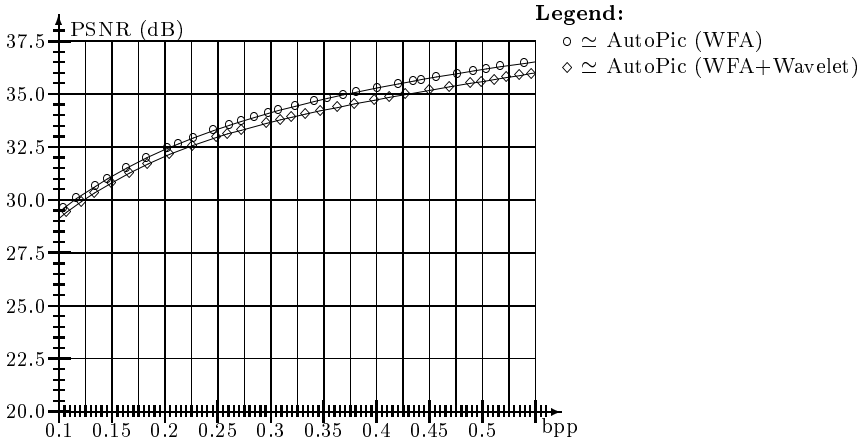


Figure 5.10: Rate–distortion diagram of the image Lenna.

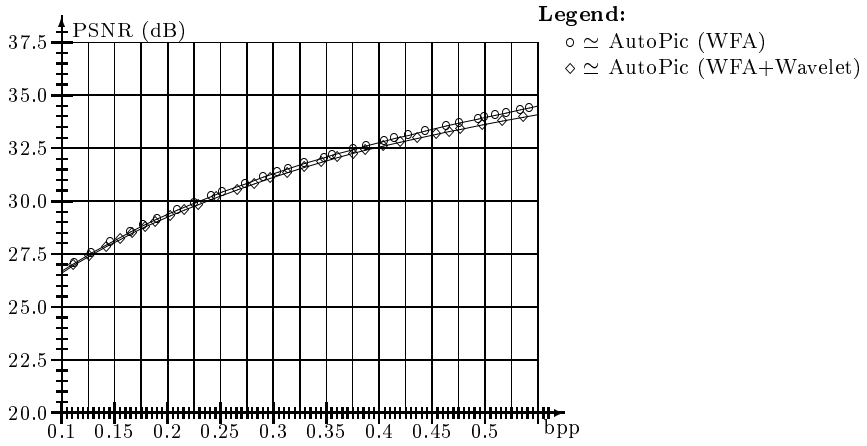


Figure 5.11: Rate–distortion diagram of the image Boat.

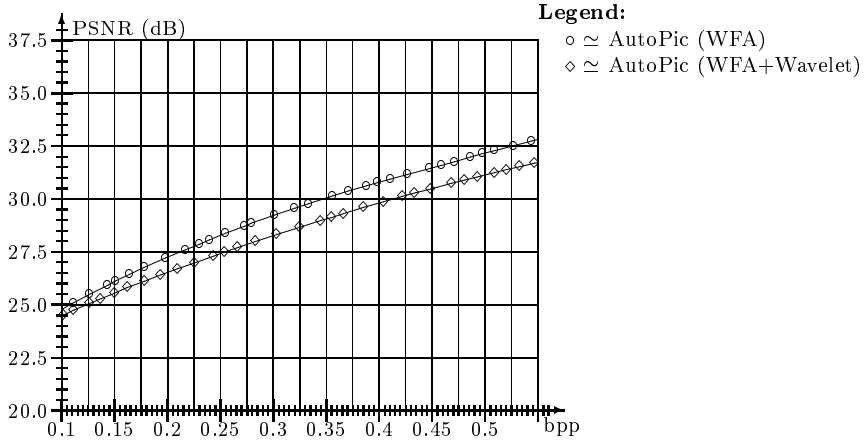


Figure 5.12: Rate–distortion diagram of the image Barb.

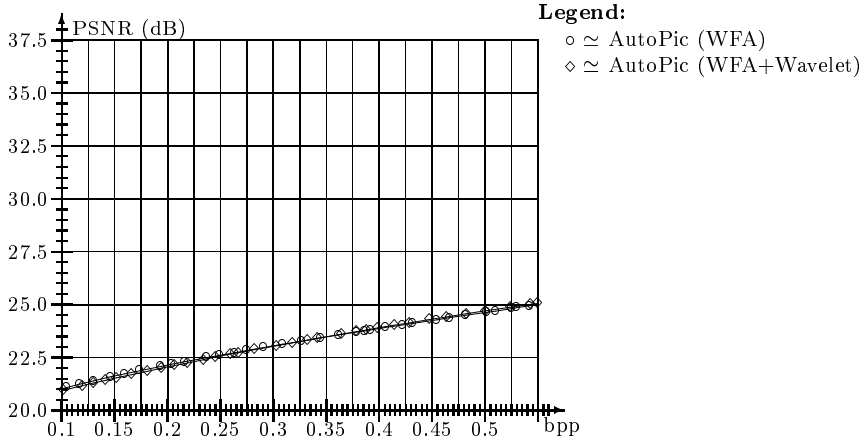


Figure 5.13: Rate–distortion diagram of the image Mandrill.

Chapter 6

Video Coding with WFAs

The coding of image sequences is becoming one of the most important topics of data compression. As we have already seen, the storage of images takes a huge amount of data to be stored. In the application of video coding, this amount of data is multiplied by the number of frames in the video sequence. A true color image with a resolution of 512×512 takes 768 K-Bytes to be stored in PCM coding. Thus we could only store approximately a 30 seconds long video sequence at a rate of 25 frames per second on a CD-ROM. We have seen that the WFA technique may be utilized to reduce spatial redundancy of the image frames. However, we will see that this technique can be refined to also reduce temporal redundancy in image sequences.

In this chapter, we examine how to encode image sequences efficiently by the use of the WFA technique. For this task we have some alternatives to choose from.

- A first approach to encode image sequences is the application of the WFA technique in three dimensions instead of two. Since the time and memory requirements of the WFA technique in two dimensions are already very high, we decided not to use this approach.
- The coding of individual frames and reusage of states of previously created WFAs seems to have a good chance of high compression performance. A drawback of this approach is that the WFA image partition does not reflect natural alterations of the frames as motion. This technique has been implemented by M. Unger with no satisfying results [Ung95].
- Another way for reducing temporal redundancy is to use motion compensated prediction.

We decided to base our implementation on existing techniques for reducing temporal redundancy by motion compensation and adapt them in a natural way to WFA coding.

6.1 MPEG Video Compression

The well-known MPEG[1] 4 technique reduces the data volume of image sequences essentially by two techniques:

- motion compensation for reducing redundancy caused by the similarity of successive images.
- discrete cosine transform (DCT) for reducing the redundancy of individual images (see appendix E.4.2 on page 186).

More details about MPEG can be found in [Kou95].

Instead of the DCT, we use WFA coding to reduce spatial redundancy. We adapt to MPEG nomenclature and thus call the individual pictures of an image sequence *frames*.

6.2 Block-based Motion Compensation

The high similarity of successive frames makes it possible to use a previously coded image (called reference image R) for the estimation of the gray values of the current image B. Typically, the images differ in single objects having moved by a small distance. Therefore, the image is partitioned into blocks of equal size which will be assigned a *motion vector* describing the translation of blocks of B with respect to R. The translated block in R will thus be used as an estimation for the current image block. For an illustration of this process see Figure 6.1.

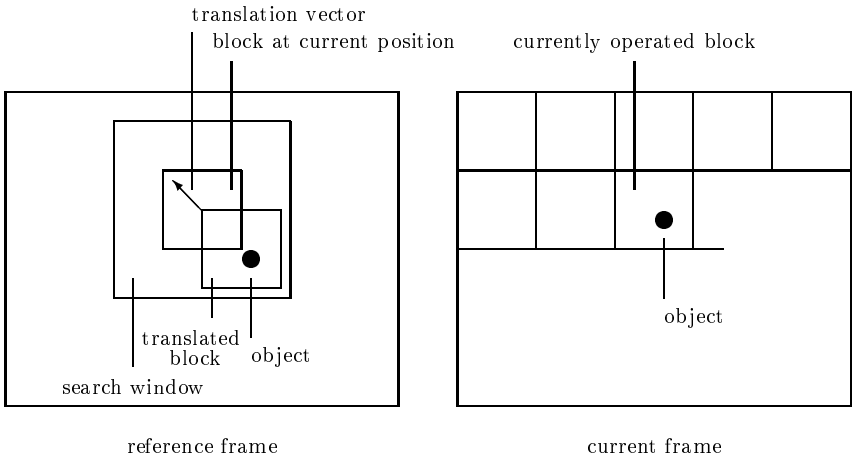


Figure 6.1: Block-based motion compensation.

[1] MPEG $\simeq$ Motion Picture Experts Group.

The encoder searches a neighborhood of the current block in the reference frame and subtracts this new block from the current block, where we regard the blocks as vectors. The image obtained by processing all blocks in the form described above is called MCPE (motion compensated prediction error) and is transmitted by a frame encoder, which is in our case the WFA encoder. If the prediction is functioning "well", most values of the MCPE are small and can be compressed efficiently by a WFA encoder. For an example of a MCPE, see Figure 6.2. Note that we have intensified the contrast of this image to make the effect more clearly visible.

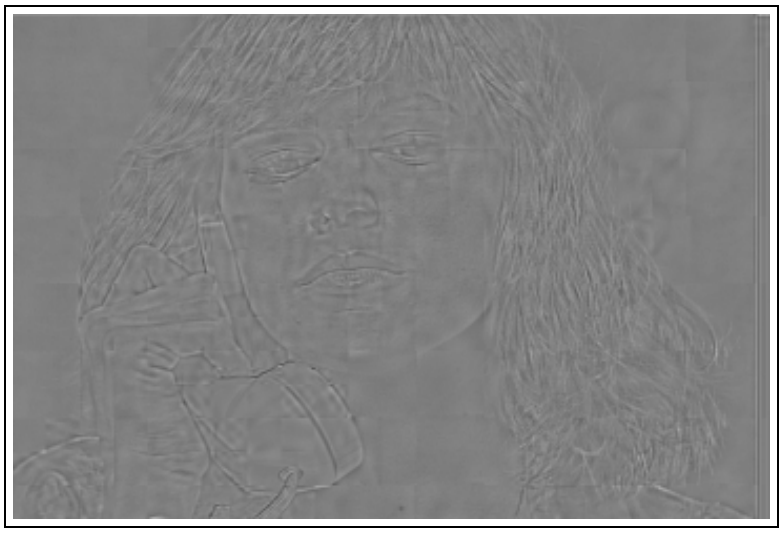


Figure 6.2: MCPE of frame 1 of the image sequence Susie (see Figure 6.5 on page 124). Reference frame is frame 0.

For the video decoder, the situation is reversed. The frame decoder yields an approximation of the MCPE to which the same estimation as used in the encoder is added. Thus a reconstruction of the next frame is obtained.

Note that for an exact reconstruction of the estimation, an efficient encoder should use reconstructed frames instead of original frames to suppress error propagation. This concept leads to the design of our video encoder as shown in Figure 6.3.

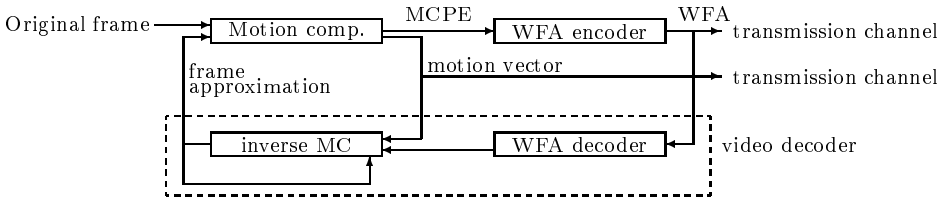


Figure 6.3: Structure of the WFA video encoder. The dashed box is the structure of the WFA video decoder.

6.3 Rate–Distortion Constrained Motion Compensation

An important question of this simple block-based motion compensation is which translation is the most suitable for the current block. In MPEG, this problem is solved by taking the translation causing the lowest mean absolute difference. In our implementation, we use the resulting badnesses of the translations as used in the WFA coder (see equation 3.18 on page 48) for this decision. In this way, we add the motion compensation in a coherent way to the ongoing WFA coding step.

6.4 Bi-Directional Motion Prediction

Since objects may be covered by other objects in image sequences, motion compensation may be performed in two directions: the *forward prediction* described in the last section and *backward prediction* which is performed using a reference frame in the future.

In MPEG, bi-directional motion compensation is performed by estimation in two directions and averaging the two estimations. The disadvantage of this method is that for bi-directional motion compensation, two motion vectors have to be transmitted. In our implementation, we decided to transmit only the best prediction and thus only one motion vector. The decision whether forward or backward prediction is used is transmitted separately to the decoder by an arithmetic encoder. In this way we store an average overhead of at most 1 bit per frame. Note that in order to suppress error propagation, the order of the frames has to be permuted (see next section).

6.5 Image Types

In accordance with MPEG nomenclature, we distinguish the following three image types differing by the type of motion compensation applied:

- *intra coded frame* or *I-frame*: no motion compensation is performed,
- *predicted frame* or *P-frame*: only forward prediction is used,
- *bi-directional predicted frame* or *B-frame*: a frame of this type can be predicted by forward or backward prediction.

In MPEG 4, P- and B-frames may also contain intra coded blocks.

In our video coder, we cut an image sequence to small image sequences of variable length called GOP (group of pictures). These image sequences may contain all three types of frames. For an illustration of a GOP see Figure 6.4.

Note that too many B-frames cause a high distance between the processed image and the associated reference image, thus decreasing the efficiency of motion compensation.

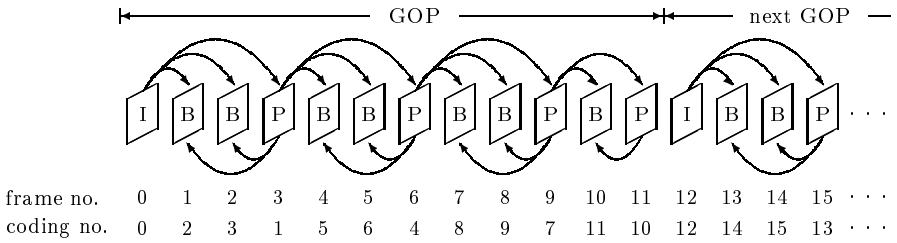


Figure 6.4: Partitioning of an image sequence. The arrows mark the direction of the prediction.

6.6 Subpixel Precise Motion Compensation

An important observation is that motion detection does not have to be restricted to discrete pixel values. MPEG offers a mode for motion compensation with half pixel precision. The gray values at the intermediate positions are calculated by piecewise linear interpolation of the surrounding pixels. If the pixel values are given by $p_{i,j}$, the intermediate values are given by

$$\begin{aligned} p_{i+1/2,j} &= 1/2(p_{i,j} + p_{i+1,j}), \\ p_{i,j+1/2} &= 1/2(p_{i,j} + p_{i,j+1}), \\ p_{i+1/2,j+1/2} &= 1/4(p_{i,j} + p_{i,j+1} + p_{i+1,j} + p_{i+1,j+1}). \end{aligned} \tag{6.1}$$

A disadvantage of this method is the far more expensive calculation and the possibly more expensive storage cost of the motion vectors. However, these disadvantages are in general compensated by a significantly better approximation. This technique is so successful that it is mandatory in the MPEG2 standard. Our video coder also obtains approximately ten percent higher compression factors at the same coding quality.

A well-known speed improvement to half pixel accurate motion compensation is the *two level search technique*: first the MC tests exhaustively all integer coordinates of the displacement vector and memorizes the "best" coordinate. After this first stage, the eight adjacent half pixel coordinates are searched for the best displacement vector.

We also made experiments with even higher precision which yielded no significant improvement. Note that the interpolation could also be improved by higher order interpolations. This technique has not been investigated yet.

6.7 Storage of the Displacement Vectors

In MPEG and H.263 video coding standards, a static table of variable length codes for the storage of the displacement vectors is used. In order to achieve the highest possible compression performance, we used an arithmetic codec with an adaptive source model for this task.

Delta Coding of Motion Vectors An often stated improvement is the differential coding of the motion vectors since motion vectors of adjacent blocks are highly correlated. Nevertheless, we observed no significant improvement if we encode just differences of motion vectors. This fact is substantiated by the observation that we use context models in the entropy coder instead of a static Huffman code like MPEG.

6.8 Motion Compensation of Colored Sequences

As we have seen in section 4.4.12 on page 77, the coding of the luminance channel takes approximately 80 percent of the whole storage requirement. Because the motion vectors for the chrominance channels are highly correlated with the motion vectors belonging to the luminance channel, it seems unreasonable to encode motion vectors for the chrominance channels. Similarly to MPEG, we transmit only one motion vector which is calculated using the luminance channel. We apply the inverse motion compensation at the video decoder with this motion vector in all three channels.

6.9 Further Research Topics

The topic of video coding with WFAs is not treated exhaustively within this thesis. We mention some further points (see [Ung95] for details).

- Since the displacement vectors are highly correlated, the collection of equal displacement vectors could be managed by a quadtree or bintree, or even by using light HV partitioning. This method is called *hierarchical motion compensation.*
- The reuse of previously generated states could be combined with motion compensation. Such a method has not been investigated yet.
- The motion compensation could be integrated in the WFA generation as a third alternative besides approximation and partition. Such a scheme has been successfully implemented in [HFUA97, Ung95].

- The displacement vectors could be stored relative to a frame in the neighborhood of the current frame. This method also offers a significant enhancement since regions of activity often stay the same for a large number of frames.

- For multimedia applications, the image sequences have to be combined with audio data. The audio data could eventually be stored by a WFA. This technique seems to be an interesting topic of further research.

- The technique considered above only considers translatory motion in image sequences. In a similar way, one could easily capture for example dilation and rotation of image blocks. There are also other techniques as object oriented motion compensation and split-merge motion compensation, described in [Ohm95].

- The image tiling for motion compensation could be accomplished with other techniques than with block tiling. One could imagine techniques using triangular or hexagonal image partitionings. We have decided to use rectangular tilings since most of the tile boundaries will coincide with the tiles produced by the WFA algorithm and thus do not introduce further tiling effects.

- The implemented method aims to achieve maximal compression at a given video quality. Often the video stream has to be transmitted via a data channel with fixed data rate. The WFA video codec can be easily modified to match such applications by using input or output buffers with fixed length. We sketch the adaptation process at the encoder side (the input buffer at the decoder side will have the same fill rate with a short delay):

  - The output buffer will be empty at the beginning of the coding process and the WFA encoder will encode at a high quality.
  - If the buffer is filled above a given threshold, the quality factor of the WFA encoder is reduced by a given amount. In this way, the temporal bit rate is also reduced and the buffer is emptied.
  - If the buffer fill rate falls below another given threshold, the quality factor can be raised again and thus the buffer fill rate will also be raised.

  A more sophisticated scheme could adapt the quality factor proportional to the deviation of the buffer fill rate to a given optimal rate.

Figure 6.5: The frames 0, 10, 20 and 30 of the image sequence Susie. This sequence is available at [CU].

Chapter 7

Some Implementational Remarks

In this chapter we give a short description of the program package AutoPic. This is an experimental image processing application with support for WFA image compression. The user interface may also be used to examine interactively the coding artifacts produced by the WFA coder. We discuss some important running time optimization techniques and parts of the class structure of the project. The current implementation is available at [Kat].

7.1 Instruction Manual for the Program AutoPic

The program AutoPic[1] is constructed as a normal image processing tool. The main difference is that some extra image formats were added, for example the WFA format. Strictly speaking, AutoPic is a program package designed for lossy and lossless image compression, image manipulation and text compression. The command line programs are discussed in appendix A on page 153. Most of the functionality of AutoPic may be accessed via the user interface described in this section. Because of the uncomplicated usage, we recommend that the the user shall first use this mode of AutoPic to gather experience.

We now make a small tour through the user interface of AutoPic. The screen shots were made under the operating system Microsoft[2] Windows 95. Note that the user interface may have a different look and feel with other operating systems or Java versions. After the start of the program two windows displaying the start image (start.tga in the working directory) and an additional control window, containing some buttons and menus (see Figure 7.1) are shown. The control interface is magnified in Figure 7.2. If the image start.tga is not in the working directory, the image windows remain black.

[1] The name AutoPic is derived from the Java notation for automaton coded picture.
[2] Microsoft is a registered trademark.

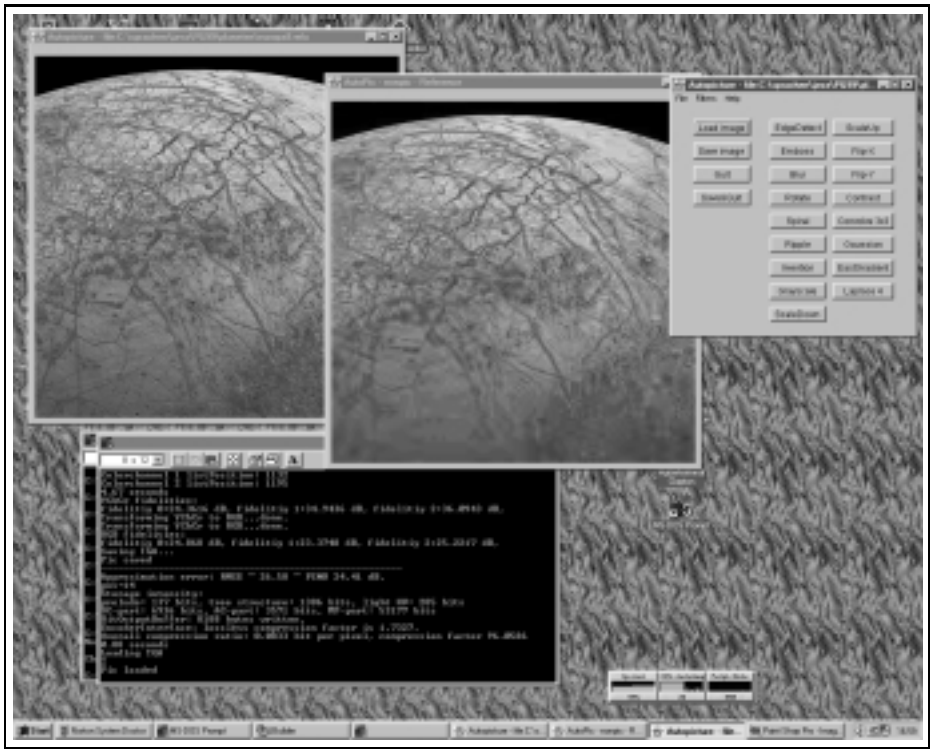


Figure 7.1: The four windows of AutoPic.

Figure 7.2: User interface of AutoPic.

After pressing the `Load Image` button, a file chooser is opened. The user can select the file location on Internet or local network. If the file is to be loaded locally, the path has to start with the prefix `file:`. If the image is to be loaded via a network, the file descriptor begins with `http:` or `ftp:`, depending on the transfer protocol the user wants to apply. Another possibility is to select the file descriptor interactively using a file selection dialogue, started after pressing the `Search File` button. Afterwards a file selection window is shown. The usual operations for directory selection are supported. After confirming the dialogue, the desired image is loaded and displayed in the main image window of AutoPic, marked with "original".

The image format is determined by the suffix of the file descriptor. The current implementation supports the following image formats:

- `.tga` is the Targa format, a lossless true color format. This format is also used for auxiliary images (see section 7.3) since it can be read and written very fast.
- `.gif` is the GIF format[3], a lossless format with 256 colors or gray values. Because of copyright reasons, only loading of GIF images is possible.
- `.jpg` is the JPEG format[4], a lossy true color format.
- `.pgm` is a format for lossless gray value image coding.
- `.bmp` is a lossless true color image format.
- `.wfa` is the main experimental format, a lossy true color format. The `.wfa` format is aimed at nearly the same set of applications as JPEG with the difference that a more modern technique is used.
- `.zfc` is an experimental image format, a lossy true color format. The format is created by our implementation of an IFS encoder, see appendix H on page 201.
- `.lli` is an experimental format for lossless true color image coding (lossless image format). For further details about this format, see section 7.4.1 on page 136.
- `.tc` is an experimental format for text compression. For further details about this format, see section 7.4.2 on page 137.

Since all image formats are handled similar to any commercial image processing tools, the handling of the program is shown with the WFA format.

When the button `Save Image` is pressed, a file descriptor with the suffix `.wfa` will be given, for example: `file:start.wfa`. Afterwards a dialogue asking for the quality of the stored image is displayed. This dialogue will of course

[3] GIF $\simeq$ Graphics Interchange Format, a registered trademark of Compuserve.
[4] JPEG $\simeq$ Joint Photographic Experts Group.

not appear for lossless image formats. The quality factor is preset to the arbitrary value 100 yielding a moderate image quality. After confirming the quality factor, the image compressor starts its work. With the `Abort` button, the coding process can be stopped. The terminal window belonging to AutoPic will afterwards display a number of dots and percent statements indicating the continuation of the coding operation (see Figure 7.3). Once the coding operation is finished, the obtained image quality is displayed as RMSE[5] and PSNR[6] in dB and the obtained compression factor (see Figure 7.4). After this process, the image approximation[7] is displayed in the second window (marked with "approximation"). If the subjective image quality does not match the requirements of the viewer, the coding can be repeated after alteration of the quality factor. The reading of WFA coded images can be obtained by using a file descriptor also ending with `.wfa`. Since the WFA coder works internally with squares having side length of dual powers, the image may be filled with a gray border to get the desired size.

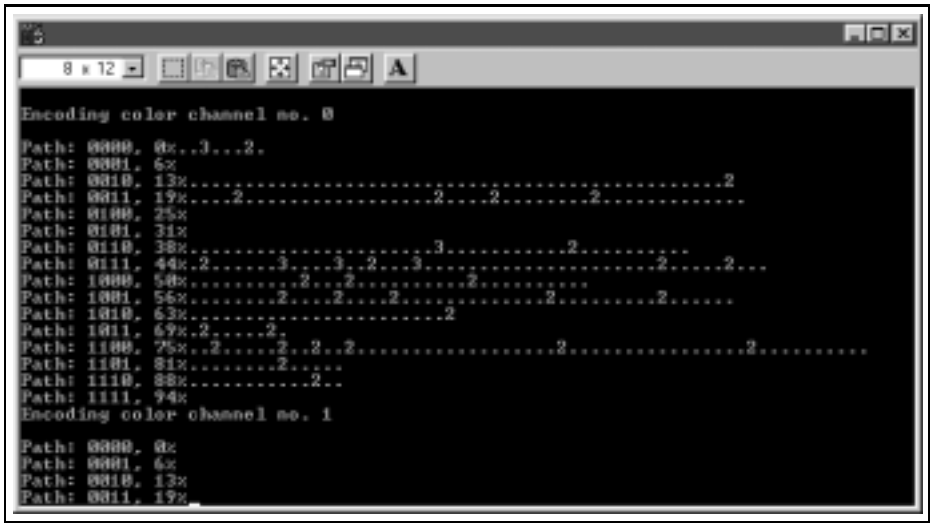


Figure 7.3: Continuation of the WFA coding operation.

Fine Tuning of AutoPic If AutoPic is in developer's mode, you can edit some extra parameters before coding the image. See Figure 7.5 for details. Note that only a small fraction of parameters can be edited. The meaning of the parameters are explained in section A.1.1 on page 154.

[5] RMSE $\simeq$ root mean squared error.
[6] PSNR $\simeq$ peak signal to noise ratio.
[7] The approximation constructed by the WFA encoder is stored in the file `approx.tga`.

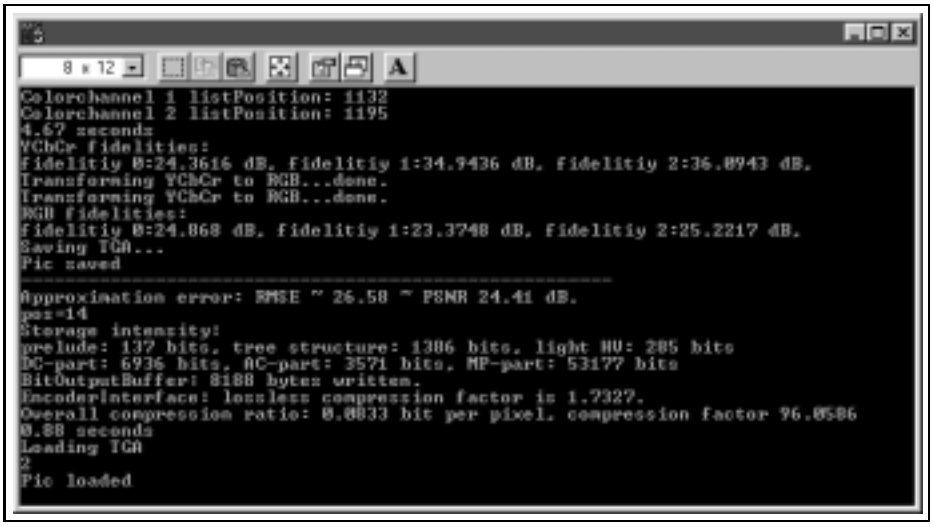


Figure 7.4: Output of AutoPic after WFA coding.

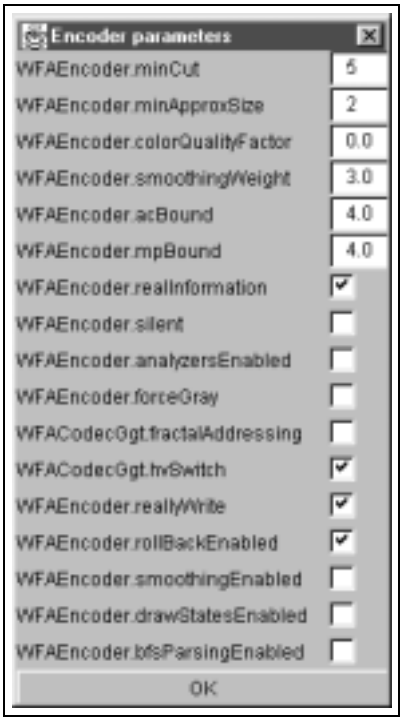


Figure 7.5: Additional parameters of WFA encoder.

7.2 Image Filters Available via the User Interface of AutoPic

A great variety of image manipulation filters is implemented in AutoPic. Some of these filters were embedded to the user interface for interactive examination of coding distortions. In this section we only describe filters required for the task of this thesis. The filter functions always refer to the main window of AutoPic. It often makes sense to apply a given filter repeatedly to an image. In case of color images the image operations are independently applied to all channels in the RGB representation. Because of space problems, we do not describe all filters available, more details can be found in [Jai89, RK82] and appendix D on page 175.

- `Edge Detect` highlights edges of the input image.
- `Blur` smoothes the input image. Here the gray value of a given pixel is replaced by the average brightness of his neighbors. This procedure is carried out for all pixels.
- `Rotate` rotates an image by 45 degrees.
- `Invert` inverts all pixel values. More precisely, a pixel gets the brightness $255-\langle\text{old brightness}\rangle$.
- `Gray Scale` converts color images to gray scale images with 256 gray values by extracting the luminance component of the color space YIQ of the image.
- `Scale Down` scales the image to half size by taking the average of four neighboring pixels.
- `Scale Up` scales the image to double size by repeating the pixels.
- `Flip-X` mirrors the image at the middle horizontal line.
- `Flip-Y` mirrors the image at the middle vertical line.
- `Contrast` changes the contrast of the image by multiplication of the pixel brightnesses by a constant factor.
- `Convolve 3×3` lets the program compute all pixel brightnesses by a weighted sum of the neighboring pixels. With this function it is possible to specify a user defined image filter.
- `Gaussian` blurs the input image.
- `East Gradient` applies a gradient filter to the input image.
- `Laplace4` applies a Laplace filter with neighborhood size four.

Figure 7.6: Original image for filter operations.

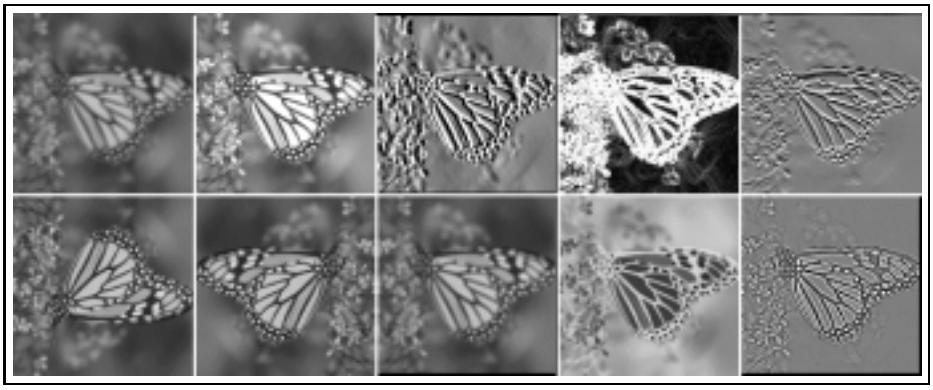


Figure 7.7: Outputs of some image filters. The filters are (left to right, top to bottom): Blur, Contrast, EastGradient, Edge Detect, Emboss, Flip-x, Flip-y, Gaussian, Invert, Laplace4.

For the results of some image filters applied to image 7.6 see Figure 7.7.

Note that while coding an image with the WFA algorithm a file is created, which can be read by statistical software such as XTREMES[8]. The file is named `xtremes.dat` and contains the coefficients obtained at the storage of the WFA. The numbers are written in ASCII code and separated by a newline symbol. The coefficient lists are separated by an empty line. With this option we could find out many statistical properties of the stored values.

Errors or warnings during encoding are protocolled in the files `AutoPic.err` and `AutoPic.war`. The messages contained in these files are beneficial, especially in case that parameters are analyzed over a long time period.

7.3 Hidden Auxiliary Functions

Here we describe some functions which are not useful for the average user, but only for developers of the program.

7.3.1 Optimization of the Encoding Parameters

This function is called if an image is stored using the suffix `.gen`. A genetic algorithm is called, setting encoding parameters in an infinite loop and determining the obtained compression factor. The image quality at which the parameters are tested and the parameters for the genetic algorithm can be adjusted in the class `GenePool`, which the WFA coder calls in this case. The results of the genetic algorithm will then be put in in ASCII code in the file mentioned above, for example `test.gen`. The resulting DNA code can be inserted into the WFA encoder via the method `setParameters`. The WFA encoder stores the resulting parameters after one coding of a WFA image into the file `para.dat` (ASCII format), which can afterwards be hard-wired in the classes `WFAEncoder`, `WFACodecGcd` and `PrecisionDescriber`. For a "good" set of parameters, the GA has to run several weeks. An example of a genetic analysis file is given below:

```
Genetic analysis of lenna.tga.
Image size is 512x512.
crossPercent is 60.0, initial mutatePercent is 5.0.
Generation size is 16, DNA code length is 24 bit.
Writing to disk all 120(1) minutes, Image fidelity is 32.0 dB,
Started at Sun Dec 05 05:27:48 GMT+03:30 1999. goodDNA is 5080183.
--------------------------------------------------------------------------------
System properties:
java.vendor:        IBM Corporation
java.version:       1.1.8
java.class.version: 45.3
java.compiler:      jitc
os.name:            Linux
```

[8] XTREMES is a statistical software tool designed by the Statistical Group of Siegen under pilotage of Prof. Dr. R.-D. Reiss.

```
os.version:         #10 Tue Nov 16 19:15:09 MET 1999.2.2.13
os.arch:            i686
user.name:          frankka
--------------------------------------------------------------------------------

Genes per day: 101.67 (Overall:101.67, multiprocessor speedup:1.0).
Generation no. 0: Maximal fitness is 43.7846
DNA code: 010011011000010001110111 = 5080183
[6] [15] [34] [45] [55] [74]
Genes per day: 94.18 (Overall:319.84, multiprocessor speedup:3.4).
Generation no. 24: Maximal fitness is 43.813
DNA code: 000011010010010011110111 = 861431
[84] [90] [97] [100] [108] [115] [132]
Genes per day: 90.53 (Overall:394.46, multiprocessor speedup:4.36).
Generation no. 74: Maximal fitness is 43.8139
DNA code: 100011010001010011111111 = 9245951
[141] [156] [165] [187] [198] [222]
```

An example of a file `para.dat` is given next. The parameters with the prefix `pd.` are data elements of the class `PrecisionDescriber`. As you can read in the class hierarchy, the objects of this class are designed for carrying huge amounts of coding parameters and making them accessible for many classes. This class will be eliminated after the optimization phase of the program since it violates the information hiding principle. The other parameters belong to `WFACodecGcd` and `WFAEncoder`.

```
Encoding parameters of AutoPic at Tue Oct 05 23:16:18 CEST 1999
pd.coeffBeforePointPrecision[0]=1 //Precision of DC coefficients before the dualpoint.
pd.coeffAfterPointPrecision[0] =6 //Precision of DC coefficients after the dualpoint.
pd.coeffBeforePointPrecision[1]=0 //Precision of AC coefficients before the dualpoint.
pd.coeffAfterPointPrecision[1] =5 //Precision of AC coefficients after the dualpoint.
pd.coeffBeforePointPrecision[2]=0 //Precision of MP coefficients before the dualpoint.
pd.coeffAfterPointPrecision[2] =5 //Precision of MP coefficients after the dualpoint.
pd.mpIndexPrecision    =6 //Precision of MP indices.
pd.coeffWindowLength[0]=1100 //model window length for DC coefficients.
pd.coeffWindowLength[1]=420 //model window length for AC coefficients.
pd.coeffWindowLength[2]=220 //model window length for MP coefficients.
pd.mpIndexWindowLength =290 //model window length for MP indices.
pd.acSizeWindowLength  =530 //model window length for sizes of AC parts.
pd.mpSizeWindowLength  =110 //model window length for sizes of MP parts.
pd.approximatedWindowLength =20 //model window length for tree structure.
pd.lightHVWindowLength =380 //model window length for HV partition.
pd.cuttingPoint[1]     =4 //maximal size of AC parts(including DC part).
pd.cuttingPoint[2]     =35 //maximal size of MP parts(including DC and AC part).
pd.acSizeAdaptivitySpeed=2 //adaptation speed of the m. of the sizes of the AC parts.
pd.mpSizeAdaptivitySpeed=2 //adaptation speed of the m. of the sizes of the MP parts.
pd.mpIndexAdaptivitySpeed=5 //adaptation sp. of the m. of the indices of the MP parts.
pd.coeffAdaptivitySpeed[0]=2 //adaptation speed of the model of the DC coefficients.
pd.coeffAdaptivitySpeed[1]=18 //adaptation speed of the model of the AC coefficients.
pd.coeffAdaptivitySpeed[2]=5 //adaptation speed of the model of the MP coefficients.
pd.deltaCodingEnabled     =true //delta coding of MP indices.

stateCostOffset   =55.0 //badness offset for states.
rollBackEnabled   =true //rollback of statistical models.
realInformationEnabled   =true //information instead of written bits for badness calc.
acBound            =2.0 //bound for the badness in linear approximation.
mpBound            =0.0 //bound for the badness in MP approximation.
smoothingWeight    =3.0 //smoothing weight for reducing tile effect.
lowerHVBound =64 //lower bound for HV partition.
upperHVBound =4096 //upper bound for HV partition.
hvBias =0.05     //bias of HV partition.
minCut =6        //minimal partition depth of bintree.
```

```
fractalAddressingEnabled =false //fractal addressing of domain pool.
hvSwitch=true //light HV partitioning.
lightHVSaveOrder  =true //save order of light HV partitioning (BFS or DFS).
dcSaveOrder       =true //save order DC parts (BFS or DFS).
acSaveOrder       =false //save order AC parts (BFS or DFS).
mpSaveOrder       =true //save order MP parts (BFS or DFS).
dcRevertIterator  =false //save order inversion of DC parts (forward or backward).
acRevertIterator  =false //save order inversion of AC parts (forward or backward).
mpRevertIterator  =false //save order invention of MP parts (forward or backward).
acSizeContextSize =0 //context size of the model for the sizes of AC parts.
mpSizeContextSize =0 //context size of the model for the sizes of MP parts.
mpIndexContextSize=0 //context size of the model for MP indices.
coeffModelContextSize[0]=0 //context size of the model for DC coefficients.
coeffModelContextSize[1]=0 //context size of the model for AC coefficients.
coeffModelContextSize[2]=0 //context size of the model for MP coefficients.
lightHVContextSize=2 //context size of the model for light HV partitioning.
approximatedContextSize=0 //context size of the model for the tree structure.
lightHVDeltaCodingEnabled     =false //delta coding of light HV partitioning.
approximatedDeltaCodingEnabled=false //delta coding of tree structure.
acSizeDeltaCodingEnabled     =false //delta coding of sizes of AC parts.
mpSizeDeltaCodingEnabled     =false //delta coding of sizes of MP parts.
mpIndexDeltaCodingEnabled    =false //delta coding of MP indices.
coeffDeltaCodingEnabled[0]   =false //delta coding DC coefficients.
coeffDeltaCodingEnabled[1]   =false //delta coding AC coefficients.
coeffDeltaCodingEnabled[2]   =false //delta coding MP coefficients.
DomainPool.badApproxEnabled  =false //bad-approximation of DomainPool.
```

As you can see in the method `WFAEncoder.setParameters`, we encode only differences in the parameters in order to use less bits for the DNA codes. The genetic algorithm therefore has to be engaged several times with updating the offsets to achieve a global minimum. We recommend that this mode is run in Unix-like environments using the `nohup`[9] and `nice`[10] commands.

7.3.2 Rate–Distortion Diagram

If an image is stored using the suffix `.pic`, the compression performance will be analyzed. The so called rate–distortion diagram is written to the `.pic` file in ASCII code, which can then be inserted into the text processing system LaTeX. For an example of such a diagram, see Figure 4.37 on page 90. It is recommended that this mode is run in Unix-like environments using the `nohup` and `nice` commands.

7.3.3 Auxiliary Files

An overview of the files created by AutoPic is given in the following list:

- At first there are several image formats described above.
- The image that is loaded by the user interface of AutoPic during the startup process is the image contained in the file `start.tga`.

[9] `nohup` $\simeq$ no hangup is used to execute processes although the user is already logged off the system.

[10] `nice` is used to assign a given process a low execution priority.

- The approximation produced by the WFA encoder is stored in the file `approx.tga`. This image is exactly that produced by the WFA decoder when called using the predefined settings. This image is automatically loaded by the user interface of AutoPic when WFA coding has been performed.
- `xtremes.dat` is a file containing statistical data. The format of this file is tuned for the statistical software package XTREMES .
- `AutoPic.err` is a file containing a protocol of the last occurred serious error in ASCII format.
- `AutoPic.war` is a file containing a protocol of the last occurred warning in ASCII format.
- ⟨name⟩`.pic` is a rate–distortion diagram for the analyzed image in LaTeX format. There are also several other types of diagrams which will be created as means and medians of generated coefficients which can be seen in this thesis.
- ⟨name⟩`.gen` is a protocol of the genetic analysis in ASCII format.
- ⟨name⟩`.dna` contains genetic information (function genes to fitnesses) of an analyzed image in Java internal format. This file type is created during genetic analysis and is used for interchange of information when many computers operate on the same image. It is also helpful in case of system crashes.
- ⟨name⟩`.out` contains output of the program which is normally sent to the standard output channel. These files are created during batch processing.
- ⟨name⟩`.ids` are files containing initial distributions for the statistical models. If these files do not reside in the current directory, the uniform distribution is used.

In order to ensure platform independency[11], the program was written in Java. This language also supports several kinds of data structures as dictionaries, vectors and enumerations which shortened the implementation time and also the source code. For the direct invocation and command line parameters of AutoPic see appendix A on page 153.

Batch Mode For batch mode, the program can be run without the graphical user interface of the program. This mode has two benefits:

- The program works in almost all Java environments because most incompatibilities of Java environments are due to GUI problems.

[11] The feature of platform independency was in our case of eminent importance since we used this feature to engage computers of several computer pools using different operating systems for the optimization of the program.

- The second advantage is that in this mode the program can be started in Unix-like environments using the `nohup` command[12].

In order to start the program in this mode, one has to type the command

`java wfa.main.AutoPic` ⟨firstFileDescriptor⟩ ⟨secondFileDescriptor⟩

where an image is loaded described by ⟨firstFileDescriptor⟩ and saved as ⟨secondFileDescriptor⟩. In this way the normal WFA compressor, genetic algorithm, draw rate–distortion diagrams and other functionality may be accessed. This mode is also interesting for the creation of runtime profiles since no running time is needed for the user interface.

7.4 Other Experimental Data Formats

In this section, we present other experimental formats implemented as side products of the work on the WFA codec. We have implemented

- a lossless image format,
- a lossless text compression format,
- and a lossy true color image format using IFS codes

described in the next paragraphs.

7.4.1 An Experimental Lossless Image Format

In our experiments with the WFA encoder, we often worried about the fact that there is no effective and widespread format for lossless true color image coding. So we decided to build our own format for this use. The format is called lli for lossless image format.

We built a simple forward estimation scheme for pixel brightnesses and combined it with the entropy coder also used for coding of WFA parameters. The most important step is the estimation by already coded pixels in the neighborhood of the current pixel. Since we store the pixel brightnesses in raster scan order, we use the neighborhood as shown in Figure 7.8.

The average brightness of the predicted pixel is subtracted from the brightness of the current pixel and the resulting estimation residue is transmitted by an arithmetic encoder as described in section 2 on page 25.

Since lossless image coding is not the topic of this thesis, we did not optimize this encoder and do not present compression results. However, we think that it is a valuable tool and so we embedded this technique to AutoPic. An image can be stored in this mode using the suffix `.lli`. For further details and command line parameters of this coder see appendix A.1.5 on page 162.

[12] In this manner the program was optimized for several months.

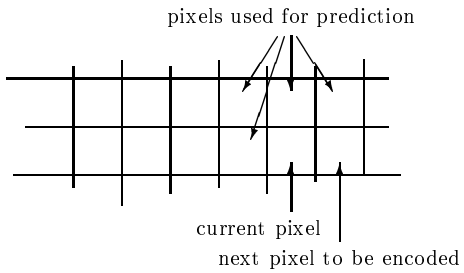


Figure 7.8: Neighborhood of a pixel used for brightness estimation.

7.4.2 An Experimental Text Compression Format

The task of lossless compression of arbitrary data files is called *text compression*. These files may contain a wide variety of data as binary data, program source code and text files. Since lossless compression is the fundamental of WFA coding, we implemented a text compressor using the models described in section 4.2 on page 64. The text compressor was built to compare the lossless compression performance with well known compressors as FELICS [How93]. However, since this type of compression is not the main topic of this thesis we do not present compression results. For further details and command line parameters of this coder see appendix A.1.6 on page 163.

7.4.3 An Experimental IFS Codec

In our image coding application we also integrated an IFS codec (see appendix H on page 201). This codec works with an adaptive image partition with rectangles. The image is first partitioned by an initial tiling. This tiling will be refined if the approximation error on a given square supersedes a given threshold. In this manner the coding quality can be easily adjusted to a given level. In order to store an image in the experimental IFS format, the file suffix `.zfc` is used. Since this coder is not the main topic of this thesis, we do not present results here. For further reading about IFS codes we refer to [Jaq93].

7.5 The Package Structure of AutoPic

Before describing the package and class structure of AutoPic, we have to pay attention to a software engineering problem which became clear during the design stage of the project. Note that classes for the WFA tree, the WFA coder and the states (this is an oversimplification, actually there are five classes modeling the WFA states) have to be specified. The remaining problem is the question: which class implements the WFA encoding algorithm? There are several alternatives in designing the class structure:

- On the one hand, one could argue that this routine belongs to the class `State` because a state has to operate itself to hold all data elements private.
- On the other hand, a state does not stand alone and has to request the state images of other states. We think that the WFA tree and the entropy encoder should be data elements of the WFA encoder.
- Another possibility is that these elements are made static elements of the class modeling the states. This alternative is also not natural.

We have implemented the second alternative and put the function `createState` into the class `WFAEncoder` and implemented routines for the insertion of matching pursuit vectors in the class `State` (`setMatchingPursuitVector`).

Another problem in the sense of software engineering is the implementation of the feedback loops of the WFA algorithm. As already shown in Figure 3.7 on page 54, the class `DomainPool` has to work in a close relation to the classes `WFAEncoder`, `WFATree` and `EncoderInterface`. We have solved this problem by utilizing the package structure of Java. The variables are declared in a way that they are only visible in the package containing the classes mentioned above.

Another big problem was the adjustment of the parameters. For this task we have to control the parameters from a single class. As this control class we have to choose a class inhabiting the greatest number of parameters. In our case this is the class `WFAEncoder`. For the "transport" of the remaining parameters we have to choose one of the following alternatives:

- The parameters may be altered via adjustment procedures. This method has the drawback of complicating the class interfaces because the `WFAEncoder` object does not have direct access to all objects.
- A new class inhabiting the remaining parameters may be created and lets the classes access the parameters as needed.

We decided on the second way and therefore implemented a class called `PrecisionDescriber` holding the parameters as public elements. This class incorporates a violation of software engineering rules. But this violation is only temporary since after optimization the class could be deleted and the parameters could be made hidden constants in the classes where they are needed.

The project is split into the following packages:

- `wfa` contains utilities used in this project.
- `wfa.automata` is a package inhabiting WFA related classes, such as classes for the various kinds of states, `WFATree`, `WFAEncoder` and `WFADecoder`.
- `wfa.entropy` contains classes related to entropy coding.

- `wfa.genetic` contains classes for the genetic algorithm.
- `wfa.graphics` is a graphics package used for the screen representation of images.
- `wfa.lli` contains classes concerning the LLI encoder.
- `wfa.main` is a package incorporating the user interface of the program.
- `wfa.math` is a package for providing mathematical classes and routines.
- `wfa.net` contains classes for file and network usage.
- `wfa.statistics` is a package for statistical evaluation, for analyzing the matching pursuit vectors and other parts of the WFA.
- `wfa.text` contains a text compressor and decompressor.
- `wfa.video` is a package containing the video encoder and decoder and related classes for video coding.
- `wfa.zfc` contains classes required for our experimental IFS codec.

These packages contain the following classes:

- `wfa`:
  - `BitModifier` contains functions to modify single bits of numbers.
  - `Counter` implements a counter.
  - `DeQue` implements a double ended queue.
  - `Diagram` is used for drawing diagrams.
  - `FifoQueue` is an implementation of a first in first out queue.
  - `LazyStack` contains an implementation of a stack, at which some elements next to the top element can also be accessed.
  - `ListElement` implements the structure of an element of a linear list.
  - `MathUtil` contains mathematical utilities required for AutoPic.
  - `RateDist` is used for drawing rate–distortion diagrams as used in this thesis.
  - `SmallStack` contains a stack with limited size.
  - `Stack` represents a stack (LIFO queue).
  - `StopWatch` is used for measuring the required time for specific operations.
  - `TimeObject` is used to create objects memorizing their creation time and which can dump their lifetime.
  - `Util` contains many utilities for AutoPic.
- `wfa.automata`:

  - `ComposedState` implements states composed by other states.
  - `DecoderState` implements states used by the WFA decoder.
  - `EncoderState` implements states used by the WFA encoder.
  - `InitialState` implements states put initially to the WFA. Their state images are computed by predefined function systems.
  - `IntDouble` implements a tuple of an integer and a double precision value. This class is designed for use in MP vectors.
  - `Path` is an implementation of a path, a description of size and position of an (image-) segment.
  - `PrecisionDescriber` is used to "transport" parameters in a comfortable way.
  - `State` is the super class of all kinds of states. This class therefore contains the basic interface of WFA states. Also most caching is implemented in this class.
  - `WFACodecGcd` is the super class of `WFAEncoder` and `WFADecoder` and thus contains fundamental properties contained in both subclasses.
  - `WFADecoder` implements the WFA decoder.
  - `WFAEncoder` implements the WFA encoder.
  - `WFALoadDialog` implements the dialogue for fine tuning of the WFA decoder.
  - `WFASaveDialog` implements the dialogue for fine tuning of the WFA encoder.
  - `WFATree` implements the behavior of the WFA tree. This class thus holds WFA states and returns them by the use of iterators.

- `wfa.entropy`:
  - `BitInputBuffer` is used to buffer and read single bits.
  - `BitOutputBuffer` is used to buffer and store single bits.
  - `BlendedModel` represents a blended model.
  - `ContextModel` implements a context model.
  - `CodecInterfaceGcd` is the super class for `EncoderInterface` and `DecoderInterface` and thus implements the common behavior and data elements of these classes.
  - `DecoderInterface` contains a comfortable interface to the entropy decoder.
  - `EncoderInterface` contains a comfortable interface to the entropy encoder.
  - `EntropyCodecGcd` is the super class of `EntropyEncoder` and `EntropyDecoder` and thus implements the common behavior of these classes.

  - `EntropyDecoder` is an implementation of an entropy decoder, in this case an arithmetic decoder.
  - `EntropyEncoder` is an implementation of an entropy encoder, in this case an arithmetic encoder.
  - `Model` is the most advanced kind of statistical model and implements delta coding.
  - `WindowModel` implements a windowed model.

- `wfa.genetic`:
  - `Gene` implements a gene for GA.
  - `GenePool` contains the genes used for GA.

- `wfa.lli`:
  - `LLICodecGcd` is the super class of `LLIEncoder` and `LLIDecoder`.
  - `LLIDecoder` implements the LLI decoder.
  - `LLIEncoder` implements the LLI encoder.

- `wfa.main`:
  - `AutoPic` is the graphical user interface and thus is the main class of this project.
  - `ViewerApplet` represents the client application used for the client server model.
  - and some minor classes concerning the GUI.

- `wfa.math`:
  - `DomainPool` is used for domain pool administration and approximation of image segments.
  - `ColorPicture` is used for the representation of color images. This class is built from three instances of the class `FloatPicture`.
  - `FloatPicture` is an implementation of a monochrome image with floating point numbers as pixels.
  - `MPVector` implements the matching pursuit vectors.
  - `MathVector` implements mathematical vectors. This is the superclass of `FloatPicture`.
  - `MathVectorHeap` is a heap structure for the class `MathVector`. This class was built to save time by avoiding initialization of `MathVector`s and garbage collector runs.
  - `MotionVector` implements displacement vectors for motion compensation in video coding.
  - `QMatrix` is an implementation of quadratic matrices.

- `wfa.net`:

  - `Connection` represents an Internet connection.
  - `Daemon` implements the client application of the client server model.
  - and some minor classes for net usage.

- `wfa.statistics`:
  - `MPAnalyzer` analyzes MP vectors.
  - `ScalarAnalyzer` analyzes scalar values.

- `wfa.text`:
  - `TextCodecGcd` implements the common behavior of `TextEncoder` and `TextDecoder`.
  - `TextEncoder` implements the text compressor.
  - `TextDecoder` implements the text decompressor.

- `wfa.video`:
  - `VideoCodecGcd` implements the common behavior of `VideoEncoder` and `VideoDecoder`.
  - `VideoDecoder` implements the video decoder.
  - `VideoEncoder` implements the video encoder.

7.6 Some Excerpts from the Class Hierarchy

In this section we describe parts of the class hierarchy of AutoPic. There are more than hundred classes in the project AutoPic. For this reason, the diagrams are not complete in order not to complicate the view. For a more comprehensive description look at the JavaDoc documentation available at [Kat].

The Notation for Class Diagrams We utilize the notation of [GHJV95]. For a short description see Figure 7.9.

7.7 Cutting of the Recursion Trees

In chapter 3, we simplified the backtracking WFA construction algorithms by omitting the cutting of the recursion trees. We show these algorithms now. This kind of running time optimization is one of the most important of our project. The recursion trees are cut when it is impossible to improve the current result. The procedures have to be called with `createState(`ϵ`,` ∞`)`.

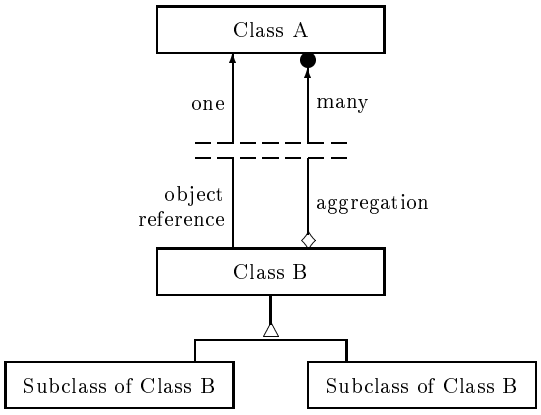


Figure 7.9: Notation for class diagrams used in this thesis.

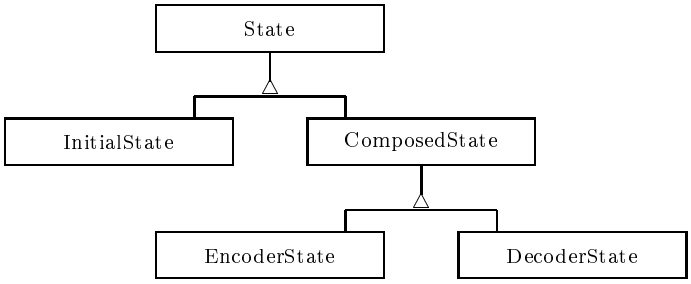


Figure 7.10: The various kinds of states and their interrelation.

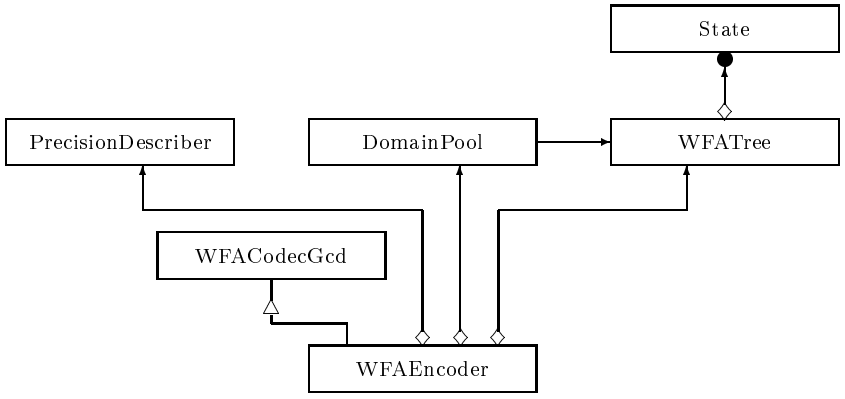


Figure 7.11: The composition of `WFAEncoder`. The class `WFADecoder` is built in a similar way.

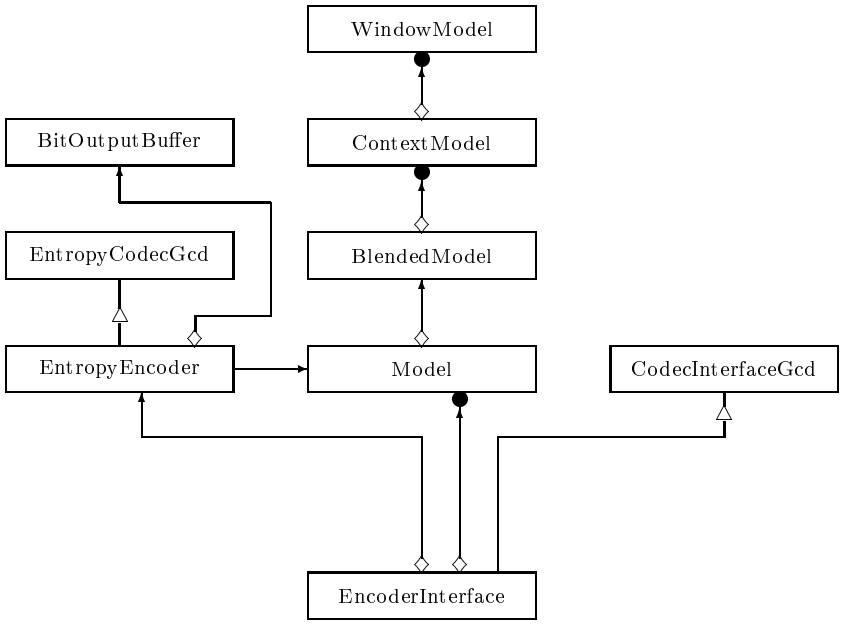


Figure 7.12:
The composition of `EncoderInterface`. The class `DecoderInterface` is built in a similar way.

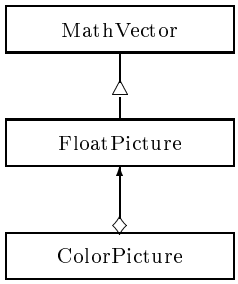


Figure 7.13: The composition of `ColorPicture`.

7.7.1 The Top Down Backtracking DFS Algorithm

```
/**
* creates a state.
*
*@param path is the path of the image segment to be
* approximated.
*@param maxBadness is the maximal badness of the
* constructed approximation.
*@return the badness of the created state.
**/
float createState(Path path, float maxBadness)
  {
  if (maxBadness< 0)
    return ∞;//cutting of the recursion tree.
  State quadState= new State(path);
  quadState.setStateImage(originalPicture.cutSegment(path));
  wfaTree.add(quadState);
  domainPool.notify(quadState);
  float stateBadness=0;
  for (q ∈ Σ)
   stateBadness+ =operateStateQuadrant(quadState, q,
                              maxBadness−stateBadness);
  return stateBadness;
  }//end createState
```

Listing 7.1: Top down WFA creation algorithm with recursion cutting.

```
/**
* creates a coefficient vector for a quadrant
* of a given state.
*
*@param state is the state to be processed.
*@param q is the number of the coefficient vector that
* is to be processed.
*@param maxBadness is the maximal badness that may
* be obtained.
*@return the obtained badness.
**/
float operateStateQuadrant(State state, int q,
                              float maxBadness)
  {
  Path quadPath=state.getPath() · q;
  FloatPicture quadVector=
                      originalPicture.cutSegment(quadPath);
  //calculate approximation.
  MPVector coeffs=domainPool.approximate(quadVector);
  approxError=
        quadVector.distance(domainPool.getVector(coeffs));
  approxCost=encoderInterface.cost(coeffs);
  approxBadness=approxCost+qualityFactor*approxError;
  //calculate partition.
  int oldExistingStates=wfaTree.size();//memorize size.
```

```
partitionBadness=
      createState(quadPath, min(maxBadness, approxBadness));

if (approxBadness<partitionBadness)
  {//delete states created in the recursion.
  int newExistingStates=wfaTree.size();
  for (l=oldExistingStates; l<newExistingStates; l++)
    wfaTree.removeElementAt(oldExistingStates);
  domainPool.reduceTo(oldExistingStates);
  state.setQuadrantWeights(coeffs, q);
  encoderInterface.rollBack();
  encoderInterface.updateModel(coeffs);
  return approxBadness;
  }
else
  {//insert pointer to the new state.
  MPVector pointer= new MPVector(quadState.index(), 1.0);
  state.setQuadrantWeights(pointer, q);
  return partitionBadness;
  }
}//end operateStateQuadrant
```

Listing 7.2: WFA state operation algorithm with recursion cutting.

7.7.2 The Bottom Up Backtracking DFS Algorithm

```
/**
* creates a state for a given image segment.
*
*@param path is the path of the image segment to
* be approximated.
*@param maxBadness is the maximal badness
* for that approximation.
*@return the badness of the created state.
**/
float createState(Path path, float maxBadness)
  {//Bottom up DFS creation of a WFA.
  if (maxBadness<0)
    return ∞;//cut recursion tree.
  State quadState= new State(path);
  float stateBadness=0; //badness of the created state.
  for(q ∈ Σ)//operate all quadrants of the image segment.
    stateBadness+=operateStateQuadrant(quadState, q,
                                       maxBadness-stateBadness);
  quadState.setStateImage(quadState.decode());
  domainPool.notify(quadState);
  wfaTree.addElement(quadState);
  return stateBadness;
  }//end createState
```

Listing 7.3: Bottom up WFA creation algorithm with recursion cutting.

7.8 General Optimizations

Although in the literature it is often suggested not to optimize programs in early coding stages, we were frequently forced to violate this rule. The reason for this behavior is mainly the handling of large data structures like images. Without optimization, the considered algorithms could not be tested because of the high time and memory consumption.

In this section we do not describe well-known optimizations, which can be applied to virtually all programs, such as code motion, loop unrolling, strength reduction, common subexpression elimination and the like [Rin92]. Instead, we describe the most important optimizations concerning especially WFA codecs.

7.8.1 Caching

The most important optimizations are the caching procedures at different levels. The first level is the memorization of the produced state[13] and quadrant images of the generated states. This level of caching is the most important and we think that the running time of our algorithm is intractable without this caching. Note that the caching of large vectors as state images ensures a threefold speedup by reducing

- the time required to compute the state image,
- the time consumed by the garbage collection for freeing the unhooked memory and
- the time required for the internal initialization of the arrays with the value 0.

Because of our good experiences with this caching, we employed caching at all conceivable places.

An important observation is that the initial state images are the same for all WFAs. Since the computing of these images (we mainly use DCT basis vectors) is a very time consuming process, we decided to cache these images separately. Thus we implemented a cache that is statically linked to the class `InitialState` (not to the objects). This cache is especially useful in video coding since the DCT basis vectors need to be computed only once.

One of the most important caches for the parameter optimization procedure is the caching of the function genes to fitnesses to ensure that the fitness of a gene is almost never calculated twice. Another cache is used for the generation of the rate–distortion diagrams because we use a bisection procedure to produce the (rate, distortion) tuples and the same tuples are accessed many times by the diagram routine.

[13] Note that due to the light HV partitioning we have two independent caches: one for flat and one for high image segments.

An important and time consuming part of the WFA construction algorithm is the calculation of storage cost. Since we observed that the best method is to compute the statistical information gain produced by the symbol to encode, the algorithm needs to compute millions of logarithms to encode a single image. Since we observed that often the same logarithms are required many times, we implemented a special caching procedure for the logarithms. This procedure caused a significant speedup of the WFA construction algorithm.

There are many values cached by the WFA algorithm which we cannot mention all, for example quadratic norms of images etc.

Scalar Products The most time consuming process in the WFA inference algorithm is the calculation of scalar products for the matching pursuit algorithm. One optimization of the matching pursuit algorithm is of eminent importance. Because the matching pursuit vectors are in general very small (we observed average lengths of approximately 2.9) we can achieve large gains in compression speed by stopping the matching pursuit algorithm in an early stage. Because we stop this algorithm using the value of the lowest achieved badness, we had to employ this property for the cutting of the matching pursuit process. We therefore check before each iteration if the lowest achievable badness is low enough to obtain a worthwhile enhancement in the approximation. With this cutting procedure we achieved approximately twice the compression speed without affecting the result. Another important optimization already described earlier is that we do not zoom the images up but instead approximate in the current resolution.

We also examined the usage of Java threads to evaluate scalar products to achieve higher compression speed on multi processor computers. We intended to generate many thread objects each evaluating one scalar product and collect the results of the computations after all objects have reported the completion of its operation. The computation could in this manner be spread over many processors of a single machine. But at the time of this writing, the Java environment does not really spread the computational resources over all processors but concentrates the work on only one processor. Because of the administration cost for the threads we observed even a slow down of the compression process. The utilization of other parallelization techniques as jada (java linda) has still to be investigated. For an overview of Objective Linda see [FK99]. A parallel implementation of a WFA coder is presented in [Haf94].

Other Methods In order to achieve maximum performance of other parts, we made extensive use of Java built-in optimizations. An important feature of the Java compiler is the usage of the `-O` switch. With that optimization the compiler generates inline code of certain routines. In order to help the compiler in this optimization process we declared time critical routines as `final`. The running time of the program is also strongly dependent on the just in time compiler. Another important optimization is the exploitation of built in

methods of the Java virtual machine running in native code such as `System.arraycopy`.

Especially in early implementation phases, we made linear approximations by using an object called multiBasis providing bases in different resolutions. A running time problem with this method occurred especially with the required matrix multiplications. We employed especially in this phase the property that the used matrices are triangular. For vector–matrix multiplication, we could thus achieve the double speed and for matrix–matrix multiplication we achieved a four-fold speedup. The inversion of the basis exchange matrices also posed problems with numerical stability which have been diminished by using the post iteration algorithm of Newton (see appendix J.3 on page 211). We will not consider these methods any further because we have observed that this method of approximation is inferior to matching pursuit approximation. But we still think that this method of approximation is interesting because it is much faster than matching pursuit.

An important optimization method is to use the Java Native Interface (JNI) to embed platform dependent implementations to the software project. A problem with this method is the question *what* to program in native code. We have the following alternatives:

- The first approach is to compute only the scalar products in native code. This would mean that we have to transfer the vectors which are multiplied to the native class. A drawback of this method is that the transfer of the array pointers is a significant overhead especially at small arrays.
- The second approach is to implement the whole `DomainPool` in native code. Because of the feedback loop this would mean that the native class would access many Java classes making it more attractive to write the whole program in native code (which is infeasible).

Because of these reasons we have implemented the first approach. In this way, we have achieved approximately twofold speedup of the encoder. By the use of fallback routines we are able to run the project on each platform even when the JNI routines do not exist.

Read and write buffer sizes are specially tuned for the Linux system since the optimization of the WFA parameters was mainly performed on a Linux computer pool. As we have seen, the process requires a highly efficient data transfer.

Despite the benefits there are some features of the language Java slowing down "number crunching" applications like WFA coding.

- Floating point arrays are initialized with the value 0.0. This is extremely annoying since at the decoding process large arrays are handled where the values need only to be copied from other arrays. In order to solve this problem, we implemented our own memory heap. But this also did not lead to satisfying results.

- The array length is a part of the array structure. This feature makes it difficult to calculate state images in place of another state image.
- There is no way to apply pointer arithmetic. In AutoPic, large arrays are handled element by element frequently. For example, vector norms are calculated by scanning the vector elements successively while summing the squares of the elements. The drawback of Java is that at each step the position of the element in the array has to be calculated without using the position of the former element. A language like `C++` supports consecutive addressing of array elements without the need for multiplications. Since this operation calculates with pointers, Java does not allow this for reasons of security.

Compatibility Problems of AutoPic It shall also be mentioned that the project has been developed using Borland JBuilder 2.0. We have seen many problems on different platforms especially with methods concerning the user interface of the program. Although the Java implementations improved significantly during the coding stage of this project, we can never assure that problems due to incompatible platforms will not arise again. This is also the reason why we implemented only a simple user interface without using special features as for example the Swing library.

Chapter 8

Conclusion

We have implemented a state of the art WFA image coder. It even outperforms in some cases the well-known SPIHT codec of W. A. Pearlman and A. Said [PS, PS96]. The enhancement was achieved by the examination of nearly all aspects of the WFA coding algorithm: the image partition, statistical models, error propagation, approximation, storage of auxiliary data, cost estimation, parameter optimization, domain pool administration, initial states, coding with varying quality, post processing of the decoded image, different initial bases and different methods to generate the WFA tree.

Although the main aspect of this thesis concerns lossy gray scale image coding, we considered also the following aspects: color coding, embedding wavelet coding in WFA coding, video coding, progressive coding, IFS coding, pyramid coding and lossless image coding. For the examination of coding artifacts we have implemented a thorough image processing system with a graphical user interface including various image filters.

We did not spend much effort on the analysis of running time and space complexities of the algorithms. This decision is justified by the fact that these features only play a subordinate role in image coding. The major decision criterion[1] to choose among given algorithms is their compression performance.

Another reason for the neglection of running time analyses is the dependence on the number of states. The time and space consumption of the WFA encoding algorithms depend on the number of states to be created. Since we do not know any algorithm to calculate the number of WFA states that will be created for this image, we lack the most important parameter for the desired time and space analysis. Because of these reasons we have presented only analyses for WFA decoding procedures.

We think that the future of WFA coding lies in WORM applications as the storage of images on CD ROM and not on symmetric applications as video telephony since the encoding process is very time consuming.

[1] If compression performance would be not the major decision criterion, each researcher would choose PCM as the coding algorithm.

We hope that we have added a small contribution for the development of compression schemes and especially those using weighted finite automata.

8.1 Acknowledgments

I want to thank the following people for their helpful comments and their never ending assistance at the carrying out of this project:

- Prof. Dr. W. Merzenich, who made me aware of this interesting area of research and many valuable stimuli in the topics of genetic algorithms and WFAs.
- Prof. Dr. F. Freisleben for his efforts in making this thesis possible.
- Prof. Dr. D. Spreen for valuable stimuli around the topic of WFA coding.
- Dr. M. Thomas, who gave me thorough assistance and help in the topics of statistics, software engineering and approximation.
- Dr. U. Hafner for the interesting collaboration, experience exchange and comparison to his WFA codec.

I also want to express my thanks to the following persons: M. Hammel, who programmed parts of the early implementation and had good ideas for speeding up the coder, Dipl.-Inform. G. Rößling helped at different topics around Java and corrected thousands of mistakes in this thesis, J. Schöw played an important role in keeping the optimization process running, Dipl.-Math. H. Schulz had some good ideas for enhancing the WFA codec, M. Fick for the work on the ZFC codec and Dipl.-Ing. T. Gutting gave several hints concerning the Unix system.

I apologize to the people who are not mentioned in the above enumeration, as for example some students of the University of Siegen who helped implementing this project.

Appendix A

Direct Invocation of the Programs

Most of the functionality of AutoPic can be accessed directly by invocation of the appropriate program. The direct invocation of the programs has the following benefits:

- Less amount of storage space is required for the execution,
- the programs may be controlled by batch processing and
- the programs may be executed without a graphical monitor (for example by using a VT100 terminal connection).

A.1 Command Line Parameters

In order to understand which command line options are accessible for the programs, we now look at a small portion of the class structure (see Figure A.1). The notation for class diagrams is based on [GHJV95] and is explained shortly in section 7.6 on page 142. The classes `WFAEncoder`, `WFADecoder`, `VideoEncoder`, `VideoDecoder`, `Daemon`, `RateDist`, `GenePool`, `TextEncoder` and `TextDecoder` contain a main procedure and can thus be executed directly. All classes in the diagram contain variables which may be altered by the user. Since a class may alter the parameters of its superclass and all its aggregates, the parameter control flow takes place transitively along the arrows. As an example, an instance of the class `VideoEncoder` may alter parameters of the classes `VideoCodecGcd`, `WFAEncoder`, `WFACodecGcd`, `DomainPool` and `PrecisionDescriber`. In order to avoid repeating command line parameters, we describe only command line parameters local to the classes. If the respective classes display their command line help, the whole set of parameters is displayed. For the sake of shortness, we also omit messages which are common to all classes, for example the way of how parameters are coded.

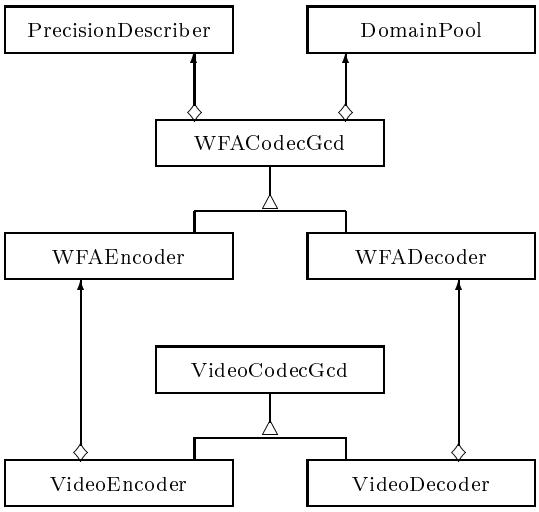


Figure A.1: A portion of the class hierarchy of AutoPic.

A.1.1 The WFA Codec

In this section we describe the invocation and command line parameters of programs concerning the WFA codec.

The DomainPool Class The class `DomainPool` is used for approximation of vectors with state images. The recognized parameters are the following:

```
Options of DomainPool:
Parameter name,                type,   min,    max, default
-------------------------------------------------------------
-badApproxEnabled,            bool,  false,   true,   false
 switches on "bad approximation".
-maxFractalLevel,              int,      0,    100,       6
 sets the maximum tree level for fractal addressing.
-minFractalLevel,              int,      0,    100,      10
 sets the minimum tree level for fractal addressing.
-orthogonalNumber,             int,      0,   1000,      10
 sets the number of states to which a new state is
 orthogonalized to.
```

The PrecisionDescriber Class As we have seen, this class is used to "transport" parameter settings to other classes. The recognized parameters are the following:

```
Options of PrecisionDescriber:
Parameter name,                type,   min,    max, default
-------------------------------------------------------------
-acSizeAdaptationSpeed,         int,     0,     20,       2
 sets   the   adaptation   speed  for  the  stat. model for
 the sizes of the AC part.
-acSizeWindowLength,            int,     1,   1000,     510
```

```
 sets  the  window length for the stat. model for the sizes
 of the AC part.
-approximatedWindowLength,      int,     1,    200,      20
 sets  the  window  length  for the stat. model for the WFA
 tree structure.
-coeffAdaptivitySpeed[0],       int,     0,     20,       2
 sets  the  adaptation speed for the stat. model for the DC
 coefficients.
-coeffAdaptivitySpeed[1],       int,     0,     20,      15
 sets  the  adaptation speed for the stat. model for the AC
 coefficients.
-coeffAdaptivitySpeed[2],       int,     0,     20,      15
 sets  the  adaptation speed for the stat. model for the MP
 coefficients.
-coeffAfterPointPrecision[0],   int,     0,     10,       6
 sets  the  binary  precision of the stat. model for the DC
 coefficients.
-coeffAfterPointPrecision[1],   int,     0,     10,       5
 sets  the  binary  precision of the stat. model for the AC
 coefficients.
-coeffAfterPointPrecision[2],   int,     0,     10,       5
 sets  the  binary  precision of the stat. model for the MP
 coefficients.
-coeffBeforePointPrecision[0],  int,   -10,     10,       1
 sets  the  binary  precision of the stat. model for the DC
 coefficients.
-coeffBeforePointPrecision[1],  int,   -10,     10,       0
 sets  the  binary  precision of the stat. model for the AC
 coefficients.
-coeffBeforePointPrecision[2],  int,   -10,     10,       0
 sets  the  binary  precision of the stat. model for the MP
 coefficients.
-coeffSign[0]                  bool, false,   true,   false
 sets  the sign of the stat. model for the DC coefficients.
-coeffSign[1]                   int, false,   true,    true
 sets  the sign of the stat. model for the AC coefficients.
-coeffSign[2]                   int, false,   true,    true
 sets  the sign of the stat. model for the MP coefficients.
-coeffWindowLength[0]           int,     1,   2000,    1080
 sets  the  window  length  for  the stat. model for the DC
 coefficients.
-coeffWindowLength[1]           int,     1,   2000,     400
 sets  the  window  length  for  the stat. model for the AC
 coefficients.
-coeffWindowLength[2]           int,     1,   2000,     200
 sets  the  window  length  for  the stat. model for the MP
 coefficients.
-cuttingPoint[1]                int,     1,    200,       4
 sets the maximal number of AC coefficients -1 (should be a
 power of 2).
-cuttingPoint[2]                int,     1,    200,      35
 sets the maximal number of MP coefficients
 -cuttingPoint[1]-1 (difference should be a power of 2).
-deltaCodingEnabled            bool, false,   true,    true
 switches on delta coding of MP indices.
-lightHVWindowLength            int,     1,   1000,     380
 sets  the  window length for the stat. model for the light
 HV partition.
-mpIndexAdaptivitySpeed         int,     0,     20,       5
 sets  the  adaptation speed for the stat. model for the MP
 indices.
-mpIndexPrecision               int,     1,     20,       7
 sets  the  binary  precision of the stat. model for the MP
 indices.
-mpIndexWindowLength            int,     1,   1000,     310
 sets  the  window  length  for  the stat. model for the MP
```

```
 indices.
-mpSizeAdaptivitySpeed          int,     0,     20,       2
 sets  the  adaptation  speed  for  the stat. model for the
 sizes of the MP part.
-mpSizeWindowLength             int,     1,   1000,     110
 sets  the  window length for the stat. model for the sizes
 of the MP part.
```

The WFACodecGcd Class The class `WFACodecGcd` is the common superclass of `WFAEncoder` and `WFADecoder` (in some sense the "greatest common divisor"). The class is used to implement common methods of the two subclasses and—of more interest here—common data elements. The recognized parameters are the following:

```
Options of WFACodecGcd:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-acRevertIterator,            bool,  false,   true,   false
 enables  the  reversion  of  the  storage of the AC parts.
-acSaveOrder,                 bool,  false,   true,   false
 switches the order of the storage of the AC parts
 (false=DFS, true=BFS).
-acSizeContextSize,            int,      0,     10,       0
 sets the context size for the stat. model for the sizes of
 the AC parts.
-approximatedContextSize,      int,      0,     10,       0
 sets the context size for the stat. model of the WFA tree.
-approximatedStateBits,        int,      0,     15,      11
 sets  the  number  of  bits used for the RLE coding of the
 lower part of the WFA tree.
-coeffModelContextSize[0],     int,      0,     10,       0
 sets  the  context  size  for  the  stat.  model   for  DC
 coefficients.
-coeffModelContextSize[1],     int,      0,     10,       0
 sets  the  context  size  for  the  stat.  model   for  AC
 coefficients.
-coeffModelContextSize[2],     int,      0,     10,       0
 sets  the  context  size  for  the  stat.  model   for  MP
 coefficients.
-dcRevertIterator,             bool,  false,   true,   false
 enables  the  reversion  of  the  storage of the DC parts.
-dcSaveOrder,                  bool,  false,   true,   false
 switches  the  order  of  the  storage  of  the  DC  parts
 (false=DFS, true=BFS).
-dialogEnabled,                bool,  false,   true,   false
 enables  a  display  for  fine   tuning  the   WFA   coder
 respectively  decoder.
-fractalAddressingEnabled,     bool,  false,   true,   false
 enables  fractal  addressing  of  MP  indices.
-horizontalStateBits,          int,      0,     15,      11
 sets  the  number  of  bits used for the RLE coding of the
 upper part of the light HV partition.
-hvSwitch,                     bool,  false,   true,   false
 enables the light HV partition.
-initialBasisFunction,          int,      0,     7,       0
 sets  the  initial basis function the WFA codec works with
 (0=cosine,  1=sine,  2=Walsh,  3=Hadamard,  4=seq. ordered
  Hadamard, 5=Slant, 6=ordered Slant, 7= Haar).
-internalColorSpace,           int,      0,     10,       2
 sets  the  internal  color  space the WFA codec works with
 (1=RGB, 2=YCbCr, 3=YUV, 4=corrected YUV, 5=YIQ,
 6=corrected YIQ).
```

```
-lightHVContextSize,           int,      0,     10,       1
 sets the context size for the stat. model for the light HV
 partition.
-lightHVSaveOrder,            bool,  false,   true,    true
 switches  the  order  of  the  storage  of  the  light  HV
 coefficients (false=DFS, true=BFS).
-mpIndexContextSize,          int,      0,     10,       0
 sets  the context size for the stat. model for MP indices.
-mpRevertIterator,            bool,  false,   true,   false
 enables  the  reversion  of  the  storage of the MP parts.
-mpSaveOrder,                 bool,  false,   true,   false
 switches  the  order  of  the  storage  of  the  MP  parts
 (false=DFS, true=BFS).
-mpSizeContextSize,            int,      0,     10,       0
 sets the context size for the stat. model for the sizes of
 the MP parts.
-partitionedStateBits,         int,      0,     15,       9
 sets  the  number  of  bits used for the RLE coding of the
 upper part of the WFA tree.
-pyramidHeight,                int,      0,     15,       3
 sets  the  pyramid  height  of  the  wavelet  transform.
-silent,                      bool,  false,   true,   false
 switches  the WFA codec to silent mode (less output on the
 the console, no beeping).
-standardFileSuffix,        string,      -,      -,  '.tga'
 sets  the  standard  file suffix (image format) for output
 and auxiliary image coding.
-waveletEnabled,              bool,  false,   true,   false
 enables wavelet transform.
-lightHVDeltaCodingEnabled    bool, false,   true,    false
 switches  on  delta  coding  of  light  HV  partition.
-approximatedDeltaCodingEnabled,   bool, false, true, false
 switches  on  delta  coding  of  tree  structure.
-acSizeDeltaCodingEnabled     bool, false,   true,    false
 switches  on  delta  coding  of  sizes  of  AC  parts.
-mpSizeDeltaCodingEnabled     bool, false,   true,    false
 switches  on  delta  coding  of  sizes  of  MP  parts.
-mpIndexDeltaCodingEnabled    bool, false,   true,    false
 switches  on  delta  coding  of  MP  indices.
-coeffDeltaCodingEnabled[0]   bool, false,   true,    false
 switches  on  delta  coding  of  DC  coefficients.
-coeffDeltaCodingEnabled[1]   bool, false,   true,    false
 switches  on  delta  coding  of  AC  coefficients.
-coeffDeltaCodingEnabled[2]   bool, false,   true,    false
 switches  on  delta  coding  of  MP  coefficients.
```

The WFAEncoder Class The class `WFAEncoder` incorporates the WFA encoder. The following command line help gives instructions for using the class directly:

```
usage of WFAEncoder:
java wfa.automata.WFAEncoder baseFileName [options]
In the simplest case, the (image) file '[baseFilename].tga'
is loaded and stored as '[baseFileName].wfa'.
Parameters are preceded by a '-' and followed  by  the  new
value.  The new values have to be printed in Java notation.
Example:  '-silent   true  -qualityFactor123.4  -minCut  7'
Warning:  most  of the parameters  are  so  called "wizard
switches",  meaning that you should know exactly  what  you
do. Most of the parameters need to be switched  to  exactly
the  same   value   for  encoding  and  decoding.  This  is
especially  the case for parameters contained  in  the  Gcd
classes,  the   PrecisionDescriber   and   DomainPool.  The
```

following parameters are available (for a thorough documentation see the AutoPic manual). The order is alphabetic.

```
Options of WFAEncoder:
Parameter name,               type,    min,    max, default
-----------------------------------------------------------
-acBound,                    float,  -10.0,   10.0,     2.0
 sets the badness threshold below which AC coefficients
 shall be inserted.
-analyzersEnabled,            bool,  false,   true,   false
 switches all statistical and Hosaka analyzers on. The
 output are latex image files "imXXXXXX.pic".
-bfsParsingEnabled,           bool,  false,   true,   false
 switches to BFS generation of the WFA tree.
-chrominanceQualityFactor,   float,    0.0,    1.0,    0.05
 sets an additional multiplicator to qualityFactor for the
 chrominance channels.
-drawStatesEnabled,           bool,  false,   true,   false
 forces the encoder to mark the boundaries of the tiling.
-forceGrayEnabled,            bool,  false,   true,   false
 switches off automatic color switching.
-hvBias,                     float,    0.0,    2.0,     0.1
 switches the bias of the HV partition.
-lowerHVBound,                 int,      0, 100000,      64
 sets the tree level, above which HV partitioning shall be
 performed.
-lowerRotQualityFactor,      float,    0.0,    2.0,     0.5
 sets an additional multiplicator to qualityFactor for the
 next outside region of interest.
-lowestRotQualityFactor,     float,    0.0,    2.0,     0.3
 sets an additional multiplicator to qualityFactor for the
 far outside region of interest.
-minApproxSize,               int,      0,   1000,       4
 sets the minimum number of pixels of a tile to perform
 approximation.
-minCut,                      int,      0,   10,         6
 sets the minimum tree height.
-mpBound,                    float,  -10.0,   10.0,     0.0
 sets the badness threshold below which MP coefficients
 shall be inserted.
-qualityFactor,              float,    0.0, 1000.0,     100
 sets the quality factor.
-realInformationEnabled,      bool,  false,   true,    true
 switches to cost estimation by statistical models instead
 of simulated writing.
-rollBackEnabled,             bool,  false,   true,    true
 switches on the rollback of stat. models when cutting WFA
 subtrees.
-rotAdjustEnabled,            bool,  false,   true,   false
 switches on quality variation for region of interest.
-smoothingEnabled,            bool,  false,   true,   false
 switches on edge smoothing for reducing the tile effect.
-smoothingWeight,            float,    0.0,  100.0,     3.0
 sets the smoothing strength for reducing the tile effect
 (higher number means less smoothing).
-stateCostOffset,            float,    0.0,  200.0,    55.0
 sets the cost offset for states to compensate the cost for
 tree storage etc.
-upperHVBound,                 int,      0, 100000,    4096
 sets the tree level below which HV partitioning shall be
 performed.
-writeInitialDistributionsEnabled,   bool,  false,    false
 forces the encoder to write initial distributions (".ids"
 file).
```

Other available options are the ones described by the class diagram.

The WFADecoder Class This class incorporates the WFA decoder. The following command line help gives instructions for using the class directly:

```
usage of WFADecoder:
java wfa.automata.WFADecoder baseFileName [options]
In the simplest case, the (image) file '[baseFilename].tga'
is loaded and stored as '[baseFileName].wfa'.

Options of WFADecoder:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-logSize,        ,             int,     -1,     20,      -1
 sets  the  logarithm of the image resolution to decode to.
 If  a  negative  value is used, the original image size is
 restored.
-progressiveNumberOfParts,     int,      1,      3,       3
 sets  number  of parts for progressive decoding. 1 decodes
 the  DC,  2  decodes additionally the AC part and 3 the MP
 part.
```

A.1.2 The Video Codec

In this section we describe the invocation and command line parameters of programs concerning the video codec.

The VideoCodecGcd Class This class contains common features of the classes `VideoEncoder` and `VideoDecoder`. The accepted command line parameters are the following:

```
Options of VideoCodecGcd:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-framesPerSecond,              int,      1,   1000,      25
 sets the number of frames displayed per second.
-gopLength,                    int,      1,   1000,       1
 sets the number of frames contained in a group of
 pictures.
-mcBlockHeight,                int,      1,     64,      32
 sets the block height of the motion compensation.
-mcBlockWidth,                 int,      1,     64,      32
 sets the block width of the motion compensation.
-mcDeltaSwitch,               bool,  false,   true,   false
 enables  delta  coding  of  motion  compensation  vectors.
-mcXDeltaCoding,               bool,  false,   true,   false
 enables  delta coding of x coordinates motion compensation
 vectors.
-mcYDeltaCoding,               bool,  false,   true,   false
 enables  delta coding of y coordinates motion compensation
 vectors.
-mcXAdaptationSpeed,            int,      0,     20,       5
 sets  the  adaptation  speed  for  the stat. model for the
 x coordinates of the displacement vectors.
-mcXAfterPointPrecision,        int,      0,      3,       1
 sets  the  binary  precision  after the dual point for the
 x coordinates of the displacement vectors.
```

```
-mcXContextSize,               int,      0,     10,       1
 sets  the  context  size  for  the  stat.  model  for  the
 x coordinates of the displacement vectors.
-mcXWindowLength,              int,      1,     64,       8
 sets  the  window  length  for  the  stat.  model  for the
 x coordinates of the displacement vectors.
-mcYAdaptationSpeed,           int,      0,     20,       5
 sets  the  adaptation  speed  for  the stat. model for the
 y coordinates of the displacement vectors.
-mcYAfterPointPrecision,       int,      0,      3,       1
 sets  the  binary  precision  after the dual point for the
 y coordinates of the displacement vectors.
-mcYContextSize,               int,      0,     10,       1
 sets  the  context  size  for  the  stat.  model  for  the
 y coordinates of the displacement vectors.
-mcYWindowLength,              int,      1,     64,       8
 sets  the  window  length  for  the  stat.  model  for the
 y coordinates of the displacement vectors.
-rangeX,                       int,      0,     64,       3
 sets  the  range  of the x coordinates of the displacement
 vectors.
-rangeY,                       int,      0,     64,       3
 sets  the  range  of the y coordinates of the displacement
 vectors.
```

The VideoEncoder Class This class incorporates the video encoder. Usage and parameters are as follows:

```
usage of VideoEncoder:
java wfa.video.VideoEncoder baseFileName [options]
In the simplest case, the (image) files
'[baseFileName]000.tga',  '[baseFileName]001.tga',  ... are
successively loaded and stored as '[baseFileName].wfa'.

Options of VideoEncoder:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-estimationPics,              bool,  false,   true,   false
 enables the displaying of estimations for the MC.
-firstFrameNumber,             int,      0,  10000,       0
 sets the first frame number to be coded.
-frameStepSize,                int,      1,  10000,       1
 sets the step size of the frames to be coded.
-lastFrameNumber,              int,      0,  10000,       0
 sets the last frame number to be coded.
-mcpePics,                    bool,  false,   true,   false
 enables the displaying of MCPEs.
-mcpeSave,                    bool,  false,   true,   false
 enables the saving of MCPEs  (filenames are "mcpe000.tga",
 "mcpe001.tga", ...).
-qualityFactor,              float,    0.0, 1000.0,   100.0
 sets  the  quality  factor  used  for  WFA  coding  and RD
 constrained MC.
```

The VideoDecoder Class This class incorporates the video decoder. Usage and parameters are as follows:

```
usage of VideoDecoder:
java wfa.video.VideoDecoder baseFileName [options]
In the simplest case, the (video) file '[baseFilename].wfa'
```

```
is stored as '[baseFileName]000.wfa',
'[baseFileName]000.wfa',....

Options of VideoDecoder:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-animate,                     bool,  false,    true,   true
 enable image animation after decoding.
-estimationPics,              bool,  false,    true,  false
 enabled displaying of estimation images.
-framePics,                   bool,  false,    true,  false
 enable displaying of single frames (still images).
-saveFrames,                  bool,  false,    true,  false
 enable storage of single frames. The file names are
 "frame000.tga", "frame001.tga", ...
```

A.1.3 The Genetic Analyzer

In this section we describe the invocation and command line parameters of programs concerning the genetic analyzer.

The GenePool Class This class incorporates the genetic analyzer. Usage and parameters are as follows:

```
usage of GenePool:
java wfa.genetic.GenePool baseFileName [options]
In the simplest  case, the image file [baseFilename].tga is
loaded  and genetic analysis is performed.  The  result  is
stored in the  file  baseFileName.gen, intermediate results
are  stored  in  the file baseFileName.dna. Note  that  the
procedure  lasts  many  weeks  in general and needs  to  be
aborted if the result stagnates.

Options of GenePool:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-approxQuality,              float,   10.0,   60.0,    32.0
 sets  the  PSNR  quality  in  dB  to  which  the  coder is
 adjusted.
-crossPercent,                 int,    0.0,  100.0,    60.0
 adjusts  how  many  genes of this pool are crossed at each
 generation.
-distanceProtocolTime,         int,      0,  infty,21600000
 sets the minimal time between two protocols.
-distanceWriteTime,            int,      0,  infty,  600000
 sets  the  minimal  time  between  two  writes of the gene
 cache.
-goodDNA,                      int,      0,  infty,   45345
 sets a starting point for genetic analysis.
-mutatePercent,                int,    0.0,  100.0,     5.0
 adjusts how  many  genes  of this pool are mutated at each
 generation.  This  value  automatically  adjusts to higher
 values if many genes are cached.
-picName,              string,      -,      -, baseFileName
 sets the picture name to be stored.
```

A.1.4 The Performance Analyzer

In this section we describe the invocation and command line parameters of programs concerning the performance analyzer.

The RateDist Class This class incorporates the implementation for drawing rate–distortion diagrams. Usage and parameters are as follows:

```
usage of RateDist:
java wfa.RateDist baseFileName [options]
In  the  simplest  case,  the image file [baseFilename].tga
is  loaded   and  a  rate--distortion diagram is drawn. The
result is stored in the file baseFileName.pic.

Options of RateDist:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-lowerBound,                 float,   10.0,   60.0,    37.0
 sets the PSNR quality in dB  above which the RD diagram is
 to be drawn.
-maxBPP,                     float,   0.01,    8.0,     0.5
 sets the maximum bpp rate for the diagram to be drawn.
-numberOfProbes,               int,      0,  infty,     128
 sets the maximal number of probes to be filled in the
 diagram.
-upperBound,                 float,   10.0,   60.0,    21.0
 sets the PSNR quality in dB  up to which the RD diagram is
 to be drawn.
```

A.1.5 The Lossless Image Codec

In this section we describe the invocation and command line parameters of programs concerning the LLI codec.

The LLICodecGcd Class The class `LLICodecGcd` is the common superclass of `LLIEncoder` and `LLIDecoder`. The class is used to implement common methods of the two subclasses and—of more interest here—common data elements. The recognized parameters are the following:

```
Options of LLICodecGcd:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-adaptationSpeed,              int,      0,     20,       5
 sets the adaptation speed of the stat. model.
-contextSize,                  int,      0,     10,       1
 sets the context size of the stat. model.
-deltaCoding,                 bool,  false,   true,   false
 enables delta coding.
-windowLength,                 int,      1,     64,       8
 sets the window length of the stat. model.
```

The LLIEncoder Class This class incorporates the LLI encoder. Usage and parameters are as follows:

```
usage of LLIEncoder:
java wfa.lli.LLIEncoder fileName [options]
In the simplest case, the (image) file
'[baseFileName].tga' is loaded and stored as
'[baseFileName].lli'.

Options of LLIEncoder:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-analyzersEnabled,            bool,  false,   true,   false
 enables the stat. analyzers.
-reallyWrite,                 bool,  false,   true,    true
 enables writing of the output.
-silent,                      bool,  false,   true,   false
 disables most console output.
```

The LLIDecoder Class This class incorporates the LLI decoder. Usage and parameters are as follows:

```
usage of LLIDecoder:
java wfa.lli.LLIDecoder fileName [options]
In the simplest case, the (image) file
'[baseFileName].lli' is loaded and stored as
'[baseFileName].tga'.
```

A.1.6 The Text Codec

In this section we describe the invocation and command line parameters of programs concerning the text codec. The usage of this codec is similar to the well-known compressor `gzip`.

The TextCodecGcd Class The class `TextCodecGcd` is the common superclass of `TextEncoder` and `TextDecoder`. The class is used to implement common methods of the two subclasses and—of more interest here—common data elements. The recognized parameters are the following:

```
Options of TextCodecGcd:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-adaptationSpeed,              int,      0,     20,       5
 sets the adaptation speed of the stat. model.
-contextSize,                  int,      0,     10,       1
 sets the context size of the stat. model.
-deltaCoding,                 bool,  false,   true,   false
 enables delta coding.
-windowLength,                 int,      1,     64,       8
 sets the window length of the stat. model.
```

The TextEncoder Class This class incorporates the text encoder. Usage and parameters are as follows:

```
usage of TextEncoder:
java wfa.text.TextEncoder fileName [options]
In the simplest case, the (text) file
'[filename]' is loaded and stored as
'[fileName].tc'.

Options of TextEncoder:
Parameter name,               type,    min,    max, default
-------------------------------------------------------------
-analyzersEnabled,            bool,  false,   true,   false
 enables the stat. analyzers.
-reallyWrite,                 bool,  false,   true,    true
 enables writing of the output.
-silent,                      bool,  false,   true,   false
 disables most console output.
```

The TextDecoder Class This class incorporates the text decoder. Usage and parameters are as follows:

```
usage of TextDecoder:
java wfa.text.TextDecoder fileName [options]
In the simplest case, the file
'[filename].tc' is loaded and stored as
'[fileName]'.
```

Appendix B

Color Spaces

In this appendix we present some well-known facts about color spaces. Computer monitors utilize the three components[1] red, green and blue (RGB coding), where without loss of generality the values are restricted to the interval $[0, 1]$. A color may be described as a point in the unit cube (see Figure B.1). Color images can be encoded by applying the compression algorithm independently to the three color components, thus reducing the problem of color compression to the coding of gray scale images. A more efficient technique is to transform the RGB values to another color system. Well suited for this task are the color systems used in the TV norms NTSC[2], PAL[3] and SECAM[4] which are called (Y, I, Q) and (Y, U, V) color spaces. Here the luminance- (Y) and chrominance information ((I, Q) or (U, V), respectively) are coded independently. Since the human visual system is more sensitive for distortions in the luminance information, the chrominance information can be encoded using less precision.

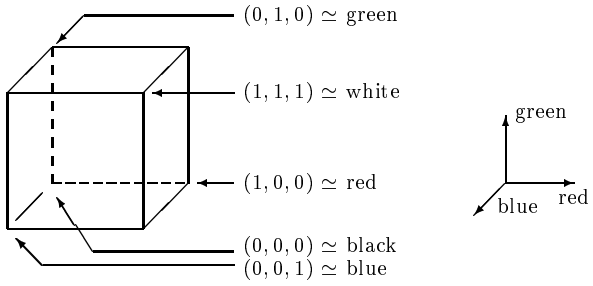


Figure B.1: Color cube of the RGB system.

The transform of a RGB signal to the color space YUV used in the color TV

[1] The gray values of the components are typically uniformly quantized using eight bits per component, respectively.

[2] NTSC is a short form for National Television System Commitee.

[3] PAL is a short form for phase alternation line.

[4] SECAM is an abbreviation for Sequentiel Couleur avec Memoire.

Figure B.2: Decomposition to RGB components.

norms PAL and SECAM is done by usage of the following equations:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B, && \text{(B.1)} \\ U &= -0.147R - 0.289G + 0.436B, && \text{(B.2)} \\ V &= 0.615R - 0.515G - 0.1B. && \text{(B.3)} \end{aligned}$$

The inverse transform is done by the following equations obtained by inverting the above matrix:

$$\begin{aligned} R &= Y + 1.14V, && \text{(B.4)} \\ G &= Y - 0.394U - 0.581V, && \text{(B.5)} \\ B &= Y + 2.03U. && \text{(B.6)} \end{aligned}$$

JPEG uses the YC_bC_r *color space*. This transform yields slightly better results than the color spaces described above:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B, && \text{(B.7)} \\ C_b &= -0.1687R - 0.3313G + 0.5B + 128, && \text{(B.8)} \\ C_r &= 0.5R - 0.4187G - 0.0813B + 128. && \text{(B.9)} \end{aligned}$$

The inverse transform is done by the following equation:

$$\begin{aligned} R &= Y + 1.402(C_r - 128), && \text{(B.10)} \\ G &= Y - 0.34414(C_b - 128) - 0.71414(C_r - 128), && \text{(B.11)} \\ B &= Y + 1.772(C_b - 128). && \text{(B.12)} \end{aligned}$$

A $\mathrm{YC_bC_r}$ decomposed color image is shown in Figure B.3.

Corrected YUV and YIQ Note that the means of the chrominance channels of the YUV and YIQ color spaces vanish. In contrast, the mean of the chrominance channels of the $\mathrm{YC_bC_r}$ color space is 128. For convenience reasons we adapted the chrominance channels of YUV and YIQ to give a mean of 128. These new color spaces are called *corrected YUV* and *corrected YIQ*, they can be accessed by the command line parameters of `WFACodecGcd`. For more details about color spaces see [RK82, Sch98].

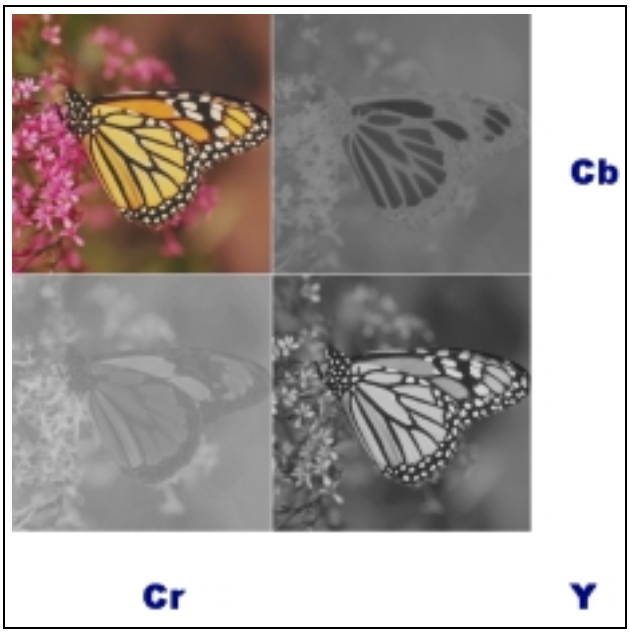


Figure B.3: Decomposition to YC_bC_r components.

Appendix C

Measures for Image Fidelity

In the following section, we discuss measures for the fidelity of image manipulations as lossy coding schemes. This section can give only a short introduction to image fidelity. A more detailed description is beyond the scope of this thesis. The definition of such fidelity measures poses problems, especially because it has to mirror the subjective perception of the *human* observer. Coding algorithms are therefore commonly tested with a great number of observers. Because such tests are both very time consuming and mathematically unemployable, we introduce other methods for measuring image fidelity.

Another important observation is that the distortion measure also depends on the application in which the images are used. One important application is the recognition of objects where the contours of image objects have to be reproduced exactly.

Depending on the distortion measure used, image coding is often dissipated to the following applications:

- *lossless image coding*: no coding error is acceptable at all. This type of coding is commonly used in technical applications.
- *near lossless image coding*: the coding error is measured by the maximum metric. This type of coding is used in applications where it is important that the brightnesses in the images do not differ by more than a given threshold. A common application to such coding schemes is the use of images for medical purposes.
- *lossy image coding*: the coding error is measured by a quadratic error measure. This measure is frequently used to approximate human error perception despite the fact that this measure is controversial.

In the next paragraphs we will discuss some measures of major importance in image coding.

An often used measure for the distortion of an $m \times n$ image is the *mean squared error (MSE)*

$$\mathrm{MSE} = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (p_{i,j} - \tilde{p}_{i,j})^2 \tag{C.1}$$

where $p_{i,j}$ and $\tilde{p}_{i,j}$ are original and reproduced pixel brightnesses, respectively. The square root of the mean squared error is called the *root mean squared error (RMSE)*.

In many articles on image compression, the two following fidelity measures are used (SNR is an abbreviation for signal to noise ratio):

1. $\mathrm{SNR}_0 = \log_{10} \frac{(\text{highest} - \text{lowest representable value of the original image})^2}{\mathrm{MSE}}$. In the case of byte-oriented calculators, one usually employs 256 gray values. For this special case we can define $\mathrm{SNR}_0 = \log_{10} \frac{(255)^2}{\mathrm{MSE}}$.
2. $\mathrm{SNR}_1 = \log_{10} \frac{(\text{highest} - \text{lowest occurring value of the original image})^2}{\mathrm{MSE}}$,
3. $\mathrm{SNR}_2 = \log_{10} \frac{\sigma_e^2}{\mathrm{MSE}}$ where σ_e^2 is the spot check variance of the original image.

Both measures have no unit of measure, but to indicate that the utilized logarithm has the base 10, the pseudo measure unit *Bel* is used. Note that SNR_2 is better adapted to human perception than SNR_0 and SNR_1. Due to usual notation we employ SNR_0 and write henceforth PSNR (peak signal to noise ratio). However, as we see below, this measure is also not very well adapted to human perception because the spatial structure of the distortions is not included in the calculation.

An ambiguity not addressed in the above definition is caused by the "real valued" reconstruction of a WFA. For the final representation on graphic cards, the reconstructed pixel brightnesses (respectively colors) have to be quantized. The above error measure does not address whether to use the reconstructed values or the quantized values. In our rate–distortion diagrams, we decided to use the quantized values to give a honest measure and note that the distortions of the original reconstructed values are slightly lower.

C.1 The Tile Effect

Many lossy image encoding algorithms, including our implementation of the WFA coder, split the input image into small blocks. The size of such blocks can be varying as in AutoPic or fixed as in older techniques like JPEG where 8×8 blocks are chosen. For an example of such a tiling see Figure C.1.

A problem with such codecs is the *tile effect*[1]: at high losses in image fidelity the image segments used in the coding algorithm will become visible (see Figure

[1] The tile effect is also called *block effect*.

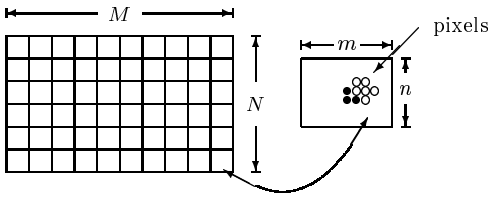


Figure C.1: Splitting an image into blocks.

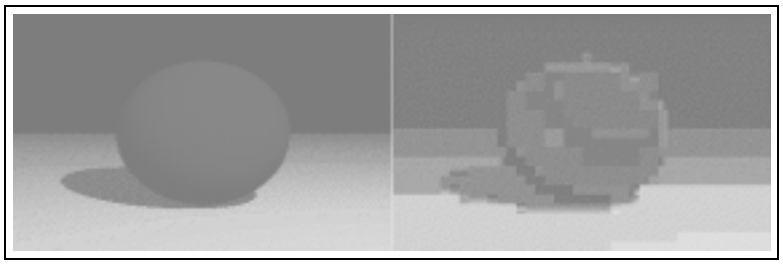


Figure C.2: Tile effect (left: original, right: heavily compressed image).

C.2). The source for such effects is that the blocks were processed independently without taking care of the neighboring blocks. The human eye is very sensitive to such effects, especially in image segments with smooth transitions. Many techniques were introduced to suppress these effects, because especially these effects are responsible for the restrictions of block-based coders. We list some of them:

- Overlapping of the blocks. At the reconstruction stage of the image, gray values of regions of overlapping blocks are reconstructed as the average value of the pixels of the blocks containing these pixels. This method increases computational resources and may diminish the compression performance.
- The edges of the blocks are smoothened after decoding by special image processing methods. This method is reported to achieve good results because no extra storage cost occurs and sometimes also enhances the fidelity in the PSNR measure.
- Transforms allowing less distortion at the edges of the blocks than in the inner regions (for example Legendre transform) may be introduced.
- Instead of blocks, one could use image segments representing the shapes of image objects.

C.1.1 Hosaka Diagrams

The tile effect considered above is more disturbing than noise distributed uniformly over the image. This is the reason why sometimes a striking difference

of the distortion in mean square sense and and the subjective distortion is observed. The most striking difference is observed for a chess board whose blocks contain only one pixel. If this board is reversed, the error is maximal in the SNR sense but is almost not observable by the human eye. Another interesting effect is dithering, introducing distortions in the SNR sense but letting the image appear more pleasant to the human observer.

Hosaka diagrams are a means to give a more detailed description of the image distortion than the mean squared error. The first step for the generation of these diagrams (also called *H-plots*) is the splitting of the original image into quadratic blocks whose variance does not exceed a given limit[2]. For a pseudo implementation see listing C.1. These blocks are then classified by their size and two properties are calculated for each class.

```
/**
* classifies blocks for Hosaka diagrams.
*
*@param S is the variance limit of the blocks.
*@param n is the initial block edge size (power of 2).
**/
void classify(float S, int n)
  {
  divide the image into n × n-blocks;
  for (each block)
    {
    σ²=variance of the block;
    if ((σ² > S) and (n > 1))
      {
      split this block into its four quadrants;
      split these quadrants, until they do not satisfy the
       condition of this if-statement;
      }
    }
  }//end classify
```

Listing C.1: Classification of the blocks for Hosaka diagram.

Let n_k be the number of blocks in the class k of blocks having edge size k. Now each class is processed separately. The mean value of the n^{th} block in the class k is calculated by

$$\mu_k(n)=\frac{1}{k^2}\sum_{i=0}^{k-1}\sum_{j=0}^{k-1} e_{i,j} \text{ for } n=0,1,\ldots,n_k-1 \tag{C.2}$$

and the *spot check variance* of this block is calculated by

$$\sigma_k^2(n)=\frac{1}{k^2-1}\sum_{i=0}^{k-1}\sum_{k=0}^{k-1}\left(e_{i,j}-\mu_k(n)\right)^2 \text{ for } n=0,1,\ldots,n_k-1. \tag{C.3}$$

[2]In Hosaka's original paper, an initial block size of 16 and a limit of 10 were used.

The average of the mean values of the classes is then calculated for each class by

$$\mu_k = \frac{1}{n_k} \sum_{n=0}^{n_k-1} \mu_k(n). \tag{C.4}$$

and m is calculated as the arithmetic mean of the μ_i. Now the properties of the image can be defined: the first property is

$$dm_k = \mu_k - m \tag{C.5}$$

and the second property of the blocks is the average spot check variance

$$\sigma_k = \frac{1}{n_k} \sum_{n=0}^{n_k-1} \sigma_k(n). \tag{C.6}$$

We measure these two properties both in the original image (dm_k, σ_k) and in the reconstructed image (dm'_k, σ'_k), where the blocks in the reconstructed image are the same blocks created by the block classification in the original image. The Hosaka diagram is now defined by the values

$$dS_k = |\sigma_k - \sigma'_k| \tag{C.7}$$

and

$$dM_k = |dm_k - dm'_k|. \tag{C.8}$$

The values are inserted in polar coordinates into the diagram, where the radius is the assigned error value and the angles are chosen equidistant. For an example of a H-plot see Figure C.3.

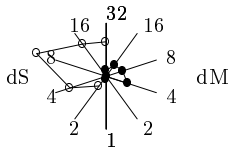


Figure C.3: A typical Hosaka diagram. The length of the rays define the value 1.

In order to determine the properties of Hosaka diagrams, let us see what happens if we manipulate the images in a controlled way:

- If a constant value is added to each gray value (brightness shift), the values of the Hosaka plot are all zero and the diagram consists only of the origin. This feature reflects that the image quality is not changed if we alter the brightness of the image.
- If noise with zero mean is added to the gray values, the dM_i will have small values and the dS_i get mean values proportional to the noise power.

For the detection of distortions caused by the tile effect, one has to assure that the Hosaka blocks do not match the image segments used by the encoder. In [Far90] the edges were cut and the Hosaka diagram has been applied to the remaining image. For further details about Hosaka diagrams see [Far90] (pages 104–108) or [Hos86].

For all H-plots in this thesis we have used the value $S = 10$ and an initial block size of 32.

Measuring Distortions in Color Images In color images, distances are usually measured in the RGB color space using Euclidean metrics. So the distance between two points $e = (r, g, b)$ and $\tilde{e} = (\tilde{r}, \tilde{g}, \tilde{b})$ is defined as

$$d(e, \tilde{e}) = \sqrt{(r - \tilde{r})^2 + (g - \tilde{g})^2 + (b - \tilde{b})^2}. \tag{C.9}$$

Note: Let us remark that this measure is not adapted to human color sensitivity since the error is measured equally on all channels.

- It is a well-known fact that the human visual system is able to distinguish much more green than blue graduations. This phenomenon is explained by the fact that primeval humans had to distinguish many plants by the color grade while the color blue occurs rarely in this area.
- Another well-known fact is that females are able to distinguish more graduations in the red area than males. This phenomenon is commonly explained by the fact that primeval females were engaged by the breeding of the children. In this often emotional work the precise determination of the color grade of the human skin is useful.
- It is known that at low luminances no color sensitivity is possible at all. This fact could be reflected by a chrominance weighting function diminishing for low luminances.
- The spatial resolution of human color sensitivity is much lower than the resolution for luminance sensitivity. This fact is commonly utilized by down-sampling the chrominance channels.
- The color sensitivity curves vary individually from human to human. There are proposals for color systems using more than three color channels.

Nevertheless, such criteria for human color vision are controversial. This is the main reason why we do not examine this interesting area any further. For more details about human color sensitivity see [Ohm95]. ■

Appendix D

Digital Image Filters

Digital image filters are used in image processing for many kinds of image manipulations like enhancement of image quality and highlighting of certain image properties for pattern recognition. In this thesis, we are mainly interested in oppressing the artifacts generated by lossy WFA coding. We still have to investigate whether digital filters can be used as a preprocessing step to encoding an image to enhance compression performance. Such a step could for example smooth image patterns which are hard to encode.

Most of the filters considered here operate locally on a limited neighborhood of a pixel, which could for example be one of the set shown in Figure D.1. Because we can give only a short introduction to digital filters, we refer for further reading to [Jai89, RK82].

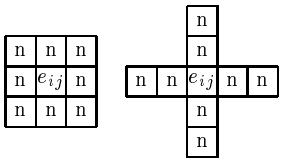


Figure D.1: Neighborhoods for digital filters.

D.1 A Neighborhood-Based Filter in AutoPic

AutoPic allows the application of neighborhood-based filters to the main image window. If the button `convolve 3×3` is pressed, a dialog (see Figure D.2) is shown allowing the input of arbitrary filter coefficients based on a neighborhood as shown in Figure D.1 (left side).

If the filter matrix C contains the values

$$C = \begin{pmatrix} c_{-1,-1} & c_{0,-1} & c_{1,-1} \\ c_{-1,0} & c_{0,0} & c_{1,0} \\ c_{-1,+1} & c_{0,+1} & c_{1,+1} \end{pmatrix}, \tag{D.1}$$

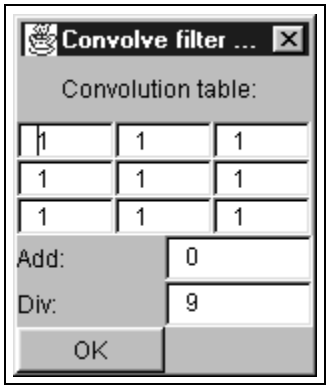


Figure D.2: The general neighborhood filter of AutoPic.

the `Add` field contains the value a and the `Div` field contains the value d, then each pixel brightness $p_{i,j}$ (in case of color images the filters are applied independently to the channels R, G, and B) is replaced by the value

$$\frac{\begin{array}{c} p_{i-1,j-1}c_{-1,-1} + p_{i,j-1}c_{0,-1} + p_{i+1,j-1}c_{1,-1} \\ +p_{i-1,j}c_{-1,0} + p_{i,j}c_{0,0} + p_{i+1,j}c_{1,0} \\ +p_{i-1,j+1}c_{-1,+1} + p_{i,j+1}c_{0,+1} + p_{i+1,j+1}c_{1,+1} + a \end{array}}{d} \tag{D.2}$$

This process is applied for each pixel, where the brightness values of the original image are used, not the new values computed by the filter. At the image boundaries we use the brightness of the next pixel in the image for the filter. With this general filter, a huge amount of image filters can be constructed for applications like edge detection and digital smoothing. For convenience, some filters are predefined and other filters are implemented which can not be constructed by this simple scheme.

Below we will give only some hints how to construct image filters for some specific tasks and give examples of how to combine filters.

D.2 Digital Smoothing

In *digital smoothing* the gray value of a given pixel is replaced[1] by a specially weighted combination of its neighboring values. One can use for example the following formula:

$$\tilde{e}_{i,j} = \frac{4e_{i,j} + e_{i-1,j} + e_{i+1,j} + e_{i,j-1} + e_{i,j+1}}{8}. \tag{D.3}$$

After processing an image by using this operator, edges appear slightly smoother. The filter can be applied several times to enhance the desired effect.

[1] This technique is reported in [Pin90] to enhance compression performance up to 25 percent with only slight visual degradation by the image filter.

Another way to implement a smoothing filter is by the use of image transforms. If an image is transformed by methods described in appendix E on page 179 and coefficients belonging to high sequency basis vectors are set to zero prior to reconstruction, sharp edges will often be removed from the image. The effect depends on the utilized transform and the number of retained coefficients. We also implemented more complex versions of low pass filters, like the *Butterworth low pass filter* [RK82] attenuating the high sequency coefficients by multiplying it by a factor which is the lower, the higher the sequency number is. Note that these filters can also be combined with wavelet techniques.

Nevertheless, the drawback of these filters is that it smoothes all edges contained in the given image. See section 4.4.14 on page 80 on how to implement an edge smoothing algorithm only altering artifacts produced by the WFA algorithm.

The Median Filter This filter replaces the gray value of the processed pixel with the median of the gray values of the neighboring pixels. The median of a set of numbers is the value m such that one half of the numbers is smaller or equal to the value m, and the other half is greater than m. This filter could be used for preprocessing to enhance compression performance and for post processing to eliminate distortions generated by lossy WFA coding.

D.3 Edge Detection

An important application of image filters in AutoPic is the detection of edges (produced by the tiling effect of WFA coding). For this task often image filters based on neighborhoods are applied.

Operators of First Order A simple way to detect edges is obtained by applying one of the filter matrices

$$\Delta_x = \begin{pmatrix} 0 & 0 & 0 \\ -1 & +1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ or } \Delta_y = \begin{pmatrix} 0 & -1 & 0 \\ 0 & +1 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{D.4}$$

This operator can also be designed in a similar way to detect edges in other directions. Other variants of this filter are

$$\begin{aligned} \Delta_x^P &= \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \ (\textit{Prewitt operator}), \\ \Delta_x^P &= \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \ (\textit{Sobel operator}), \\ \Delta_x^K &= \begin{pmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{pmatrix} \ (\textit{Kirsch operator}). \end{aligned} \tag{D.5}$$

Operators of Second Order A common technique is to build filters similar to the second order derivative. In this way the following filters may be built:

$$\begin{aligned}
\Delta_x^2 &= \begin{pmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{pmatrix} \ (\textit{second derivation operator}), \\
\Delta^L &= \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \ (\textit{Laplace operator}), \\
\Delta^{BL} &= \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 \\ 2 & -12 & 2 \\ 1 & 2 & 1 \end{pmatrix} \ (\textit{binomial Laplace operator}).
\end{aligned} \tag{D.6}$$

The Laplace filter is named by the well-known Laplace operator in differential analysis and is often used to detect edges in x- and y-direction. The binomial Laplace filter is obtained by applying first a binomial smoothing of the image and afterwards applying the Laplace edge filter. This filter is therefore often used to detect only very strong edges.

Another way to implement an edge detecting filter is by the use of a *high pass filter*. These filters are defined analogously to low pass filters, but instead of retaining the low sequency coefficients, we now retain the high sequency coefficients. We also implemented more complex versions of low pass filters, as the *Butterworth high pass filter* which is defined analogously to the Butterworth low pass filter.

Appendix E

Image Bases

As we stated earlier, the WFA codec requires an initial basis to start the encoding process with at least one state image. As already stated in [Haf99], the compression results are better if an initial basis with more than one state is used. Here the question arises which basis vectors have to be chosen in order to achieve best compression results. Hafner uses initial states obtained by storing an image containing several images of the discrete cosine transform. The reconstructed state images thus do not represent the images exactly, but on the other hand Hafner's implementation is faster[1] than ours.

In this chapter, we first introduce the topic of transform coders and present some image transforms whose basis vectors can also be used as initial state images in WFA coding.

E.1 Transform Codecs

Image transforms as cosine or Walsh transform have the goal to give an almost uncorrelated representation of a given image. The transform itself does not achieve compression, but the obtained coefficients can often be coarsely quantized. Transform codecs were for a long time the state of the art codecs and are up to now the most widespread image coders, especially the JPEG compression standard which is thoroughly described in [Wal91].

[1] Hafner computes the scalar products of state images by a recursion scheme exploiting the WFA structure. Thus the direct calculation of scalar products is omitted. The drawback of this scheme is that also initial states have to be provided by a WFA structure. We will not explain this scheme and refer to [Haf99].

E.2 Orthogonal Function Systems

A function system $\{f_i(x)\}$ with $i \in I$ and $I \subseteq \mathbb{N}$ is called *orthogonal* with respect to the integration interval $[x_0, x_1]$ with $x_0, x_1 \in \mathbb{R}$ iff

$$\int_{x_0}^{x_1} f_i(x) f_j(x)\, dx = c_i \delta_{i,j} \quad (i, j \in I), \tag{E.1}$$

where $\delta_{i,j}$ is the Kronecker symbol. If $c_i = 1$ $(i \in I)$, the function system is called *orthonormal*. In the following sections we want to restrict to real valued orthonormal function systems.

E.2.1 Sum Representation with Orthonormal Functions

Let $f(x)$ be a real valued function, quadratically integrable in the interval $[x_0, x_1]$:

$$\int_{x_0}^{x_1} (f(x))^2\, dx < \infty. \tag{E.2}$$

Now we want to approximate $f(x)$ in $[x_0, x_1]$ with orthonormal functions, meaning that we want to obtain a representation

$$\sum_{i=0}^{n-1} \alpha_i f_i(x) \tag{E.3}$$

minimizing a given distortion measure for a given $n \in \mathbb{N} \setminus \{0\}$. As distortion measure we use the quadratic error

$$\frac{1}{x_1 - x_0} \int_{x_0}^{x_1} \left(f(x) - \sum_{i=0}^{n-1} \alpha_i f_i(x) \right)^2 dx. \tag{E.4}$$

Theorem E.2.1 *The coefficients minimizing the quadratic error are given by*

$$\alpha_i = \int_{x_0}^{x_1} f(x) f_i(x)\, dx. \tag{E.5}$$

Proof: We assume that we have another approximation for the function with $\sum_i \beta_i f_i(x)$. Then we can calculate the quadratic error[2] as

$$\begin{aligned} \text{MSE} &= \int_{x_0}^{x_1} \left(f(x) - \sum_{i=0}^{n-1} \beta_i f_i(x) \right)^2 dx \\ &= \int_{x_0}^{x_1} f^2(x)\, dx - 2 \sum_{i=0}^{n-1} \beta_i \underbrace{\int_{x_0}^{x_1} f(x) f_i(x)\, dx}_{\alpha_i} \end{aligned} \tag{E.6}$$

[2] We leave out the constant factor $1/(x_1 - x_0) > 0$.

$$+\int_{x_0}^{x_1}\left(\sum_{i=0}^{n-1}\beta_i f_i(x)\right)^2 dx \tag{E.7}$$

$$= \int_{x_0}^{x_1} f^2(x)\,dx - 2\sum_{i=0}^{n-1}\beta_i\alpha_i + \sum_{i=0}^{n-1}\beta_i^2 \tag{E.8}$$

$$= \int_{x_0}^{x_1} f^2(x)\,dx - \sum_{i=0}^{n-1}\alpha_i^2 + \sum_{i=0}^{n-1}(\beta_i-\alpha_i)^2\,. \tag{E.9}$$

The last term vanishes if $\beta_i = \alpha_i$ for all $i \in \{0,\ldots,n-1\}$. □

Because MSE ≥ 0 we can conclude the *inequality of Bessel:*

$$\sum_{i=0}^{n}\alpha_i^2 \leq \int_{x_0}^{x_1} f^2(x)\,dx. \tag{E.10}$$

The orthonormal system $\{f_i(x)\}$ is called *complete* iff the following equation holds for each quadratically integrable function f:

$$\lim_{n\to\infty}\int_{x_0}^{x_1}\left(f(x)-\sum_{i=0}^{n}\alpha_i f_i(x)\right)^2 dx = 0. \tag{E.11}$$

In this case the inequality of Bessel turns to the *equality of Parseval*:

$$\sum_{i=0}^{\infty}\alpha_i^2 = \int_{x_0}^{x_1} f^2(x)\,dx. \tag{E.12}$$

E.2.2 Representation of a Function for a Finite Number of Sampling Points

In this thesis we need the approximation of a finite number of sampling points. Thus we turn the back to the above representation where an infinite number of samples were approximated. Therefore, we assume that we have n ($n \in \mathbb{N}\backslash\{0\}$) sample points x_k and n functions for that task.

Analogously to equation E.1 on the preceding page we define the orthogonality of the function system $\{f_i(x)\}$ as

$$\sum_{k=0}^{n-1} f_i(x_k)f_j(x_k) = \delta_{i,j}. \tag{E.13}$$

The sum generation is done using the formula

$$e_k = \sum_{i=0}^{n-1}\alpha_i f_i(x_k) \tag{E.14}$$

where the coefficient α_i is calculated by

$$\alpha_i = \sum_{k=0}^{n-1} e_k f_i(x_k). \tag{E.15}$$

The reason for this kind of representation is given in the next section where we examine these representations using methods of linear algebra.

E.2.3 Representation with Transform Matrices

The above sum can be represented by the multiplication of the sample vector[3] $\underline{e} = (e_0, \ldots, e_{n-1})$ with a transform matrix T whose column vectors are the function values of orthonormal functions[4] (if $\underline{e}$ is an n-ary vector then T is a $n \times n$ matrix). The transform can thus be written as

$$\underline{f} = \underline{e}T \tag{E.16}$$

and the inverse transform can be written as

$$\underline{e} = \underline{f}T^* \tag{E.17}$$

where T^* is the conjugated transposed matrix of T. The coefficients of the basis vectors can then be found in the vector $\underline{f} = (f_0, \ldots, f_{n-1})$.

Now we may write the orthonormality condition more conveniently as

$$T^*T = I, \tag{E.18}$$

where I denotes the n-dimensional unity matrix. The equation E.18 means that T is a *unitary matrix*. The multiplication of a vector with such a matrix can be interpreted as a basis changing from one orthonormal basis to another orthonormal basis. Equation E.15 can be interpreted as scalar product with basis vectors and equation E.14 as linear combination of basis vectors. Furthermore, in this representation equation E.18 corresponds to equation E.13. If we refer in the following calculations to the matrix components of T, we write

$$T = \begin{pmatrix} t_{0,0} & \cdots & t_{0,n-1} \\ \vdots & & \vdots \\ t_{n-1,0} & \cdots & t_{n-1,n-1} \end{pmatrix}. \tag{E.19}$$

[3] In order to emphasize that we transform a vector, we write in this paragraph $\underline{e}$. In the next paragraph, we transform an image block, written as $\underline{\underline{e}}$. This convention is leaned on books on physics and will only be used in this section.

[4] Besides that we augment our representation at this point to complex valued function systems.

Two Dimensional Transform

Since for the application of image compression one not only wants to benefit of correlation in one dimension but in two dimensions, a two dimensional transform has to be performed. We assume that the image or the image segment is represented as a $n \times n$-matrix $\underline{\underline{e}}$. This is done by multiplying the transposed matrix from the other side. So the transform can now be written as

$$\underline{\underline{f}} = T^t \underline{\underline{e}} T, \tag{E.20}$$

and the inverse transform is written as

$$\underline{\underline{e}} = \overline{T} \underline{\underline{f}} T^*. \tag{E.21}$$

The term $\overline{T}$ denotes the matrix $\left(\overline{t}_{i,j}\right)$.

E.2.4 Image Energy

Unitary transforms conserve the *image energy*

$$||\underline{e}||^2 = \sum_{i=0}^{n-1} e_i^2 = \underline{e}\,\underline{e}^t, \tag{E.22}$$

as can be seen at the following identities:

$$\begin{aligned} ||\underline{e}||^2 &= \underline{e}\,\underline{e}^t && \text{(E.23)} \\ &= (\underline{f}T^*)(\underline{f}T^*)^t && \text{(E.24)} \\ &= \underline{f}(T^*T)\underline{f}^t && \text{(E.25)} \\ &= \underline{f}\,\underline{f}^t && \text{(E.26)} \\ &= ||\underline{f}||. && \text{(E.27)} \end{aligned}$$

Note that this equation is also called *equality of Parseval*. Because of the linearity of the transform, the quantization error is also the same, independent of whether the error is measured in the original or in the transformed image. This fact can be used for the quantization of the transform coefficients. Because of the conservation of image energy we can also explain the frequently used term *compaction of image energy*: unitary transforms put the image energy of "normal images" in only a few coefficients and have many coefficients at values near zero.

E.3 A Statistical Experiment

As another motivation for image transforms we consider a (fictive) statistical experiment: we assume that we had some digitized images and cut these image

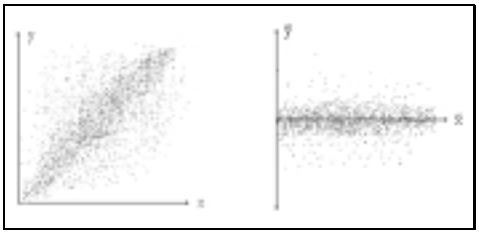


Figure E.1: A statistical experiment.

into blocks of size 1×2. Now we choose randomly a great number of blocks from the generated domain. Because the two pixels in the blocks are adjacent, it is likely that both pixels in the block have a similar gray value. If we now drew a scatter plot of these gray values, meaning that we draw a point for each block, which coordinates are both gray values of the block, so we would obtain a scatter plot similar to that in Figure E.1 (left side). At this drawing one makes the observation that most points lie around the line $y = x$, because of the correlated gray values considered above.

Now we assume that we rotate the coordinate system around the angle $\pi/4$ in mathematical positive direction respective the origin and obtain so the coordinate axis $\tilde{x}$ and $\tilde{y}$ (see Figure E.1 (right side)). We observe that the values are almost decorrelated.

At both scatter plots we can also make another observation: if we assume we had to cut a coordinate (set to 0), we would have big distortions in the first set of data in both cases. In the right set of data we could set the $\tilde{y}$ coordinates to zero without obtaining big distortions. This effect is utilized by transform codecs with the only difference that bigger image blocks are used. More details about image energy can be found in [Jai89, GW92, Win72].

E.4 Some Linear Transforms

In the next sections we present some well-known linear transforms which are frequently utilized in transform coding. In order to test the performance of these transforms with WFA coding, we have implemented most of the transforms listed below.

E.4.1 The Hotelling Transform

The *Hotelling transform*[5] is defined as the transform annulling the pairwise correlation of each pair of transform coefficients. It can therefore be seen as the statistical optimal transform for data compression.

[5]The Hotelling transform is often called the *discrete Karhunen-Loève transform*.

Theorem E.4.1 *The basis vectors t_i of the Hotelling transform are the real valued, orthonormalized eigenvectors*[6] *of the correlation matrix R with*

$$r_{i,j} = E\{(e_i - E\{e_i\})(e_j - E\{e_j\})\}, \tag{E.28}$$

where we assume that R is non-singular. Therefore, the equation

$$Rt_k = \lambda_k t_k, \tag{E.29}$$

holds where the λ_k are the eigenvalues of R for $k \in \{0, \ldots, n-1\}$.

Proof: Without loss of generality we assume $E\{e_i\} = 0$ ($i \in \{0, \ldots, n-1\}$). We observe that

$$\begin{aligned}
r_{i,j} &= E\{e_i e_j\} && \text{(E.30)}\\
&= E\left\{\left(\sum_{k=0}^{n-1} f_k t_{k,i}\right)\left(\sum_{l=0}^{n-1} f_l t_{l,j}\right)\right\} && \text{(E.31)}\\
&= \sum_{k=0}^{n-1}\sum_{l=0}^{n-1} t_{k,i} t_{l,j} E\{f_k f_l\} && \text{(E.32)}\\
&= \sum_{k=0}^{n-1}\sum_{l=0}^{n-1} t_{k,i} t_{l,j} \lambda_l \delta_{k,l} && \text{(E.33)}\\
&= \sum_{l=0}^{n-1} t_{l,i} t_{l,j} \lambda_l && \text{(E.34)}
\end{aligned}$$

where in step E.33 the correlation matrix of f_i was assumed as a diagonal matrix corresponding to uncorrelatedness. The multiplication of both sides with $t_{k,j}$ and summation using index j leads to

$$\begin{aligned}
\sum_{j=0}^{n-1} r_{i,j} t_{k,j} &= \sum_{j=0}^{n-1}\sum_{l=0}^{n-1} t_{l,i} t_{l,j} t_{k,j} \lambda_l && \text{(E.35)}\\
&= \sum_{l=0}^{n-1} t_{l,i} \lambda_l \underbrace{\sum_{j=0}^{n-1} t_{l,j} t_{k,j}}_{\delta_{k,l}} && \text{(E.36)}\\
&= \sum_{l=0}^{n-1} t_{l,i} \lambda_l \delta_{k,l} && \text{(E.37)}\\
&= \lambda_k t_{k,i}. && \text{(E.38)}
\end{aligned}$$

In vector notation this can be written as

$$Rt_k = \lambda_k t_k. \tag{E.39}$$

[6] Because the correlation matrix is real valued and symmetric, the existence of the orthonormalized basis consisting of eigenvectors is ensured.

□

If the eigenvectors are sorted in descending order with respect to the associated eigenvalues, the image energy is compacted[7] in the lower coefficients. [NH88] gives a proof that the Hotelling transform produces the smallest distortion among all linear transforms due to the cutting of coefficients. Despite the benefits of this transform, it is almost never used in image coding because of the following disadvantages:

- The covariance matrix is often non-stationary. For this reason, this matrix would have to be recalculated for each image block to achieve optimal results.
- Fast transform algorithms only exist for special cases [Jai89].
- Because image statistics differ from image to image, one would have to transmit extra information. In the application of data compression this would produce an overhead which is not diminished by the benefits of this transform.

Because of these drawbacks, researchers are looking for transforms achieving similar results as the Hotelling transform without its drawbacks. A frequently used approximation to this transform is the cosine transform which is addressed in the next section.

E.4.2 The Discrete Cosine Transform

The discrete cosine transform is related to the discrete Fourier transform and is commonly assumed to be the best[8] suited transform for image compression due to its very good compaction of energy [NH88]. For this reason, this transform is often used in practice, as for example in the JPEG and MPEG standards.

If the set

$$\left\{\frac{1}{\sqrt{n}}\right\} \cup \left\{\sqrt{\frac{2}{n}}\cos\left(\frac{2j-1}{2n}(i-1)\pi\right) : i = 2,\ldots,n\right\} \tag{E.40}$$

or stated otherwise

$$\left\{\sqrt{\frac{1+\kappa_i}{n}}\cos\left(\frac{2j-1}{2n}(i-1)\pi\right) : i = 1,\ldots,n\right\} \text{ with } \kappa_i = \begin{cases} 0 & \text{if } i = 1 \\ 1 & \text{else} \end{cases} \tag{E.41}$$

[7] The term "compaction of the image energy" refers to the fact that the coefficients mentioned above have the greatest influence on the reconstructed image. Or stated differently, the cutting of these coefficients produces high distortions in the sense of the mean squared error. See also section E.2.4 on page 183.

[8] This statement is based on empirical tests.

is chosen as orthogonal basis vectors (discrete) for $j = 1, \ldots, n$, the resulting transform is called *cosine transform.* The transform matrix is therefore built as

$$t_{i,j} = \sqrt{\frac{1+\kappa_j}{n}} \cos\left(\frac{2i-1}{2n}(j-1)\pi\right). \tag{E.42}$$

For more details on the DCT see [AN83].

E.4.3 The Hadamard Transform

The *Hadamard transform* takes on values in the set $\{-1, +1\}$. Since this transform is defined only for signal lengths having powers of 2, we write $N = 2^n$ for $n \in \mathbb{N} \setminus \{0\}$. We define this transform by the use of transform matrices in the following way:

$$\begin{aligned} H_1 &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \\ H_{n+1} &= \frac{1}{\sqrt{2}} \begin{pmatrix} H_n & H_n \\ H_n & -H_n \end{pmatrix} \text{ for } n \in \mathbb{N} \setminus \{0\}. \end{aligned} \tag{E.43}$$

Since the basis vectors of the Hadamard transform can also be defined by uniformly sampling the so-called *Walsh functions*, this transform is also called the *Walsh-Hadamard transform.* The basis vectors are not sequency ordered, and thus the ordered version is called the *ordered Hadamard transform.* The Hadamard transform can be implemented so that it is performed in $O(N \log N)$ time without multiplications. The (ordered) Hadamard transform was used frequently in transform codecs, but is now rarely used since the cosine transform has proven superior.

E.4.4 The Slant Transform

Since the Slant transform is only defined for powers of 2, we write $N = 2^n$ for $n \in \mathbb{N} \setminus \{0\}$. The transform is defined by matrix multiplication in the following way:

$$S_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{E.44}$$

and

$$S_n = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 0 & \mathbf{0} & 1 & 0 & \mathbf{0} \\ a_n & b_n & & -a_n & b_n & \\ \mathbf{0} & & I_{N/2}-2 & \mathbf{0} & & I_{N/2}-2 \\ 0 & 1 & \mathbf{0} & 0 & -1 & \mathbf{0} \\ -b_n & a_n & & b_n & a_n & \\ \mathbf{0} & & I_{N/2}-2 & \mathbf{0} & & -I_{N/2}-2 \end{pmatrix} \begin{pmatrix} S_{n-1} & \mathbf{0} \\ \mathbf{0} & S_{n-1} \end{pmatrix} \tag{E.45}$$

for $n \geq 2$ where $\mathbf{0}$ denotes the zero matrix of the appropriate size and the parameters a_n and b_n are defined by the following equations

$$a_{n+1} = \sqrt{\frac{3N^2}{4N^2-1}} \text{ and } b_{n+1} = \sqrt{\frac{N^2-1}{4N^2-1}}. \tag{E.46}$$

Let us remark that the basis vectors of the Slant transform are not sequency ordered. See [Jai89] for how to order the basis vectors efficiently by construction. Extensive information about the Slant transform can be found in [PCW74].

E.4.5 The Haar Transform

This transform is also only defined for powers of two ($N = 2^n$). The discrete Haar transform is obtained by first constructing continuous Haar functions $h_k(x)$ with $k \in \{0, \ldots, N-1\}$ over the interval $[0, 1)$ and afterwards sampling these functions equidistantly. The index k can be uniquely written as

$$k = 2^p + q - 1 \text{ with } p \in \{0, \ldots, n-1\} \text{ and } \begin{cases} q \in \{0,1\} & \text{for } p = 0 \\ q \in \{1, \ldots, 2^p\} & \text{for } p \neq 0 \end{cases}. \tag{E.47}$$

Now we can define the Haar functions by

$$h_0(x) = h_{0,0}(x) = \frac{1}{\sqrt{N}} \text{ for } x \in [0,1) \tag{E.48}$$

$$h_k(x) = k_{p,q}(x) = \begin{cases} \frac{2^{p/2}}{\sqrt{N}} & \text{for } x \in \left[\frac{q-1}{2^p}, \frac{q-1/2}{2^p}\right), \\ \frac{-2^{p/2}}{\sqrt{N}} & \text{for } x \in \left[\frac{q-1/2}{2^p}, \frac{q}{2^p}\right), \\ 0 & \text{for other values } x \in [0,1). \end{cases} \tag{E.49}$$

The basis vectors of the Haar transform are obtained by sampling the Haar functions at the values m/N with $m \in \{0, \ldots, N-1\}$. The Haar functions are sequency ordered. This transform is not very interesting for energy compaction reasons, but it is acknowledged as the "grandfather" of wavelet transforms. It is interesting to see that this transform can be done extremely fast in $O(N)$ time using only additions and subtractions. For further details concerning this transform see [SS98].

E.5 Quantization of the Coefficients

The following statements are made for "usual" image models. Using only transforms achieves no compression, but the transforms may achieve a representation suited better for compression than the original representation. In case that one changes a value in the transformed data, the distortion is no longer local, but spreads over the whole transform block (for the transforms considered in this thesis). The Hotelling transform decorrelates the input data completely, which is only approximated by the other transforms. Most information of the image

segment is put in the coefficients belonging to the lower-sequent[9] functions. If the basis functions are ordered by sequency and placed in the transform matrix, most image energy is found after transform in the upper left corner of the coefficient matrix. In order to evaluate this energy compaction, the coefficients are stored in "zigzag order", as shown in Figure E.2 (this order simplifies the cutting of the zero coefficients). In this figure, the number of sign changes is hinted by f_x and f_y in x- or y-direction respectively.) The upper left coefficient has special meaning: since it belongs to the constant basis function[10] in x- and y-direction, this coefficient is therefore called *DC coefficient*. The other coefficients are usually called *AC coefficients*.

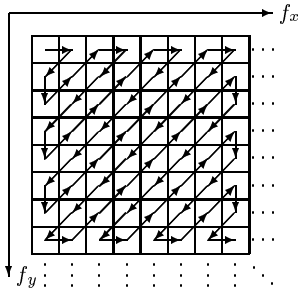


Figure E.2: Zigzag order of the coefficients.

A convenient method to achieve data compression is to encode a fixed or variable number of coefficients, but this method does not achieve high compression performance. Better compression performance is achieved by individual quantization where more bits are assigned to the lower-sequential coefficients. The bit assignment can be done by a quantization matrix, for which you can see an example in Figure E.3. In practice, predefined quantization matrices are often used instead.

$$\begin{pmatrix} 6 & 4 & 3 & 3 & 2 & 1 & 0 & 0 \\ 4 & 3 & 2 & 2 & 1 & 1 & 0 & 0 \\ 3 & 2 & 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure E.3: Typical bit assignment matrix for transform codecs (0.75 bit/pixel).

Transform coding achieves high compression ratios if one can tolerate moderate distortions. In the literature one can find other transforms, such as the Hartley

[9] The sequency number is the number of sign changes of that function in a given interval [Har68].

[10] Some orthogonal function systems do not contain a constant function, but they have no widespread use in image coding and will thus not be examined here.

transform, but these transforms have proven to be inferior to the transforms described in this thesis (see [Far90] for a comparison). For further reading we refer to [Jai89].

Appendix F

Elementary Coding Techniques

F.1 Suppression of Zeroes

The suppression of zeroes is a data compression technique searching a data stream for sequences of zeroes[1] and storing them more efficiently. If such a sequence is found, it is replaced by a special (usually shorter) sequence. As the first symbol of this new sequence, a special symbol (also called *escape sequence*) is used indicating that at this point zero suppression has been applied. The second symbol is the number of zeroes which have been replaced. The coding process is illustrated in Figure F.1 where 0 is the suppressed symbol and the sign $\underline{Z}$ indicates the zero suppression.

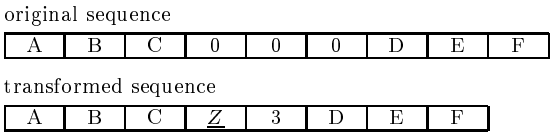


Figure F.1: Zero suppression.

The technique of zero suppression should only be utilized if the escape sequence and the counter require less bits to store than the sequence in plain text. Note that the extra symbol can be generated in an easy way by an arithmetic codec. A variant of this technique is that a fixed number of zeroes can be encoded by one escape symbol (an example for this is the tabulator key on each computer keyboard).

[1] In this context, zero refers to a special symbol in the data stream which is repeated frequently. For text coding, this could be the space symbol. The term zero is due to its historical use and will not be changed here.

F.2 Run Length Encoding

The method of *run length encoding (RLE)* is a special form of zero suppression and is commonly used in coding of facsimile coding. Because long sequences of equal gray values occur often in digital images, this property may be applied to data compression. Instead of repeating the gray values, the gray value followed by the length of the current sequence is encoded.

Example: We assume that we wish to encode the following sequence of numbers: "3, 3, 3, 1, 1, 5, 5, 5, 5". With run length encoding we would obtain the following sequence of 2-tuples: "(3, 3), (1, 2), (5, 4)". ■

High compression performance may be achieved especially for binary images[2], since often long sequences of equal gray value can be found in such images. The storage of the gray values may be avoided since the gray values are alternating. The efficiency of run length encoding depends especially on the average length of the encoded sequences. In this case escape sequences may also be applied. Such sequences can be encoded with runs of zero length. RLE is frequently used in image coding applications such as facsimile, but is inferior for example on images with text because there one rarely finds long runs. In facsimile compression, RLE is often combined with an end-of-line-symbol to restrict the results of transmission errors to small image areas.

Note that RLE has also been extended to two dimensions where the lengths of the runs are encoded differentially. More information about run length encoding can be found in [NH88, Pin90].

F.3 Some Prefix Encoders

This kind of coding tries to lower the average code word length by assigning symbols with higher probability shorter code words and assign the longer code words to symbols with lower probability. This kind of coding, as the arithmetic coding mentioned above, is called *entropy coding* since the goal is to assign an input symbol a code word length resembling its information gain. The problem of prefix encoders is that they approach the entropy limit not as closely as the arithmetic coder. We are mentioning these coders only to show an analogy between the WFA encoder and these coders.

F.3.1 The Shannon Fano Coder

The *Shannon Fano coder* is an initial stage to the Huffman encoder presented in the next section. Its codes are not necessary optimal, but the encoder is popular because of its simplicity.

[2] A *binary image* has at most two gray values.

At the first stage for the construction of the code, the data has to be analyzed to calculate the probabilities of the input symbols. Afterwards, one has to sort the input symbols descendingly regarding their probabilities.

The next step is to partition the set of input symbols to two subsets with approximately the same overall probability. The symbols of the first group are assigned a 0 as the first bit in their code word and the symbols of the second group are assigned a 1 as the first bit in their code word. After that the process is repeated independently on both groups until no further division is possible.

Now it is clear that the Shannon Fano coding provides a minimal prefix code and has a time complexity of $O(|E| \log |E|)$ for the construction of the coding tree. The construction of the coding tree is illustrated in Figure F.2 (left side), where the division is illustrated by lines of differing lengths. The resulting coding tree is shown in Figure F.2, having an average code word length of 2.2 bit per symbol. This value is close to the lower bound of approximately 2.16 bit per symbol.

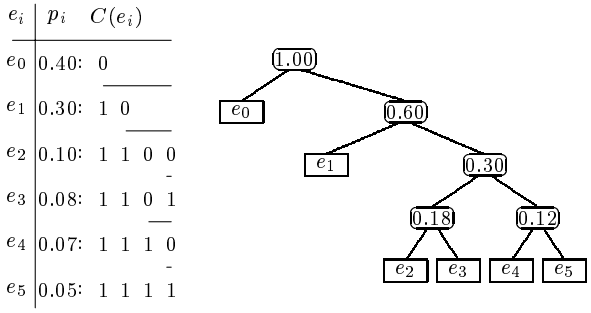


Figure F.2: Construction of the Shannon Fano code tree.

The Shannon Fano code is useful when the splitting to subsets is done to groups with exactly the same probabilities. But due to the optimality of the Huffman code, the Shannon Fano code is in general inferior. But the Shannon Fano code approximates an optimal code if the order of the input alphabet tends to infinity [NH88].

F.3.2 The Huffman Coder

The Shannon Fano coder was nearly completely displaced already a few years after its invention. The Huffman encoder yields with slightly higher implementation effort an optimal prefix code meaning that there exists no prefix code achieving a lower average code word length. This improvement is obtained by the creation of the coding tree bottom up instead of top down as in the case of Shannon Fano coding.

Construction of the Huffman Code Tree

A prerequisite for the construction of the Huffman coding tree is, as is the case for all statistically-based coders, that the probabilities of the input symbols are known. At the generation of the code tree we interpret the trees as binary trees. The assigned probabilities are stored in the nodes after completion of the generation process as code tree.

In the first step, we generate a single node for each input symbol $e \in E$ with $p_e > 0$. Afterwards we search the two nodes with lowest probabilities and create a new node to which the two nodes with lowest probability are appended. This new tree is assigned the sum of the two probabilities of its children. We continue in this manner and merge trees with lowest probabilities until only one binary tree remains. The merging of n trees ($n \leq |E|$) requires exactly $n - 1$ merging steps. Note that the symbols with the lowest probability will be in the lower part of the resulting code tree and the symbols with higher probability are near the root of that tree. This process is illustrated in Figure F.3 where the steps are shown from top to bottom and the assigned probabilities are stored in the nodes of the tree. The tree obtained in this example does not surpass the average code word length of the Shannon Fano code tree of 2.2 bit per symbol (see Figure F.2), as the two obtained coding trees can be transformed to each other by the swapping of subtrees at the same niveau.

The construction of a Huffman code tree is performed by using a heap and has a time complexity of $O(|E| \log |E|)$, where E is the input alphabet. A pseudo implementation is given in listing F.1.

```
/**
* constructs a Huffman code.
*
*@return a code with minimal average code word length
* in code tree form.
**/
CodeTree Huffman()
//assumes that  p_e > 0 ∀e ∈ E holds.
  {
  while (there is more than one tree)
    {
    determine the two trees with the lowest probabilities;
    merge these trees to a new tree with the sum of the two
     probabilities in the root and the two trees as children;
    }//end while
  return the resulting tree;
  }//end Huffman
```

Listing F.1: Creation of the Huffman code tree.

For a proof that the so obtained code tree is optimal and adaptive versions of Huffman coding we refer to [Kat94].

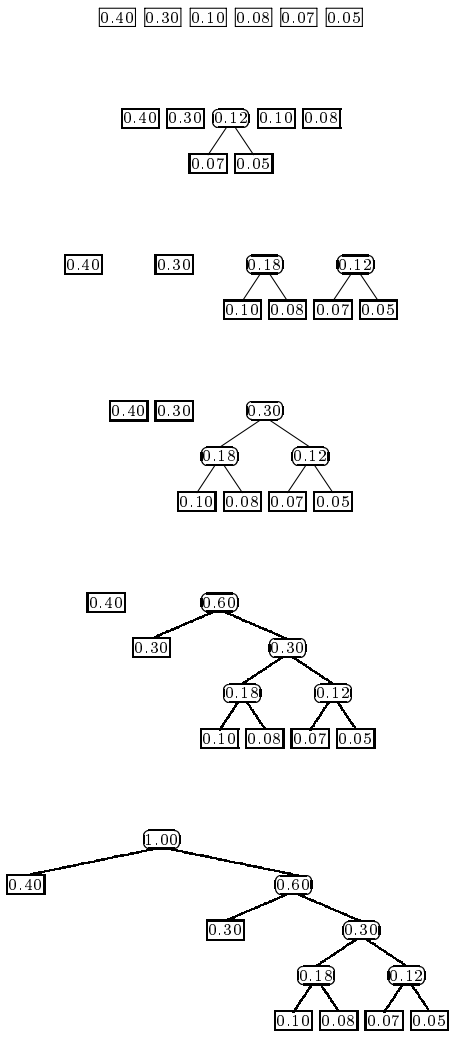


Figure F.3: Construction of the Huffman code tree.

Appendix G

Bounds for Coding Efficiency

In this section we discuss bounds for the compression efficiency of codecs. C. E. Shannon has proved that these bounds can be approached arbitrarily close. For this reason, we will have an efficiency test for encoders in case that the required bound can be calculated. Unfortunately these bounds give no hint for constructing efficient encoders approaching these bounds.

G.1 Bounds for Lossless Encoding

An important result of Shannon states that the average code word length cannot be smaller than the entropy H of the encoded source. However, the result we need here is that the empirical entropy is a lower bound for the average code word length:

$$L_C \geq H = -\sum p_i \log p_i \tag{G.1}$$

where $E = \{e_i\}$ is an input alphabet with relative frequencies n_i/N, n_i is the frequency of the symbol e_i and $N = \sum_{e_i \in E} n_i$.

Proof: The proof is done by induction on the number of transmitted symbols. The induction hypothesis is written as

$$L_C^{(N)} \geq -\sum \frac{n_i}{N} \log \frac{n_i}{N}, \tag{G.2}$$

where $L_C^{(N)}$ is the average code word length of N transmitted symbols.

The induction base, here the statement for $N = 1$, is fulfilled, since the entropy is in this case 0 and the average code word length is non-negative. Induction step: now we assume that the statement is fulfilled for an $N \in \mathbb{N} \setminus \{0\}$. At first we split the set of input symbols into two classes: the class E_0 contains the symbols having as first code bit a 0 and E_1 contains the input symbols having as first code bit a 1. The number of transmitted symbols in these classes are called N_0 or N_1 respectively. We write this as

$$N_0 = \sum_{i:e_i \in E_0} n_i \text{ and } N_1 = \sum_{i:e_i \in E_1} n_i. \tag{G.3}$$

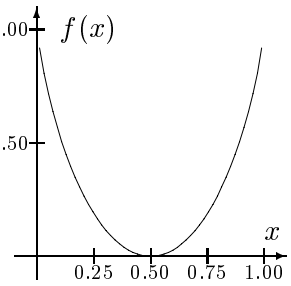
We can also see that $N_0, N_1 \geq 1$ since otherwise the code would not be optimal because we would waste bits. With $N_0 + N_1 = N + 1$ we can conclude that $1 \leq N_0 \leq N$ and $1 \leq N_1 \leq N$ and we can apply the induction hypothesis to E_0 and E_1:

$$
\begin{aligned}
L_C^{(N+1)} &\geq 1 - \frac{N_0}{N+1} \sum_{\{i:e_i \in E_0\}} \frac{n_i}{N_0} \log \frac{n_i}{N_0} \\
&\qquad - \frac{N_1}{N+1} \sum_{\{i:e_i \in E_1\}} \frac{n_i}{N_1} \log \frac{n_i}{N_1} && \text{(G.4)} \\
&= 1 - \sum_{\{i:e_i \in E_0\}} \frac{n_i}{N+1} \log \frac{n_i}{N_0} \\
&\qquad - \sum_{\{i:e_i \in E_1\}} \frac{n_i}{N+1} \log \frac{n_i}{N_1} && \text{(G.5)} \\
&= 1 - \sum_{\{i:e_i \in E_0\}} \frac{n_i}{N+1} \log n_i - \sum_{\{i:e_i \in E_1\}} \frac{n_i}{N+1} \log n_i \\
&\qquad - \frac{N_0}{N+1} \log \frac{1}{N_0} - \frac{N_1}{N+1} \log \frac{1}{N_1} && \text{(G.6)} \\
&= - \sum_{\{i:e_i \in E\}} \frac{n_i}{N+1} \log n_i \\
&\qquad + 1 - \frac{N_0}{N+1} \log \frac{1}{N_0} - \frac{N_1}{N+1} \log \frac{1}{N_1} && \text{(G.7)} \\
&= - \sum_{\{i:e_i \in E\}} \frac{n_i}{N+1} \left(\log \frac{n_i}{N+1} - \log \frac{1}{N+1} \right) \\
&\qquad + 1 - \frac{N_0}{N+1} \log \frac{1}{N_0} - \frac{N_1}{N+1} \log \frac{1}{N_1} && \text{(G.8)} \\
&= - \sum_{\{i:e_i \in E\}} \frac{n_i}{N+1} \log \frac{n_i}{N+1} + 1 + \log \frac{1}{N+1} \\
&\qquad - \frac{N_0}{N+1} \log \frac{1}{N_0} - \frac{N_1}{N+1} \log \frac{1}{N_1} && \text{(G.9)}
\end{aligned}
$$

The first term of the right side is the entropy of the new class. Because of that we have now to show that the rest is non-negative. So we consider the expression

$$
\begin{aligned}
& 1 + \log \frac{1}{N+1} - \frac{N_0}{N+1} \log \frac{1}{N_0} - \frac{N_1}{N+1} \log \frac{1}{N_1} && \text{(G.10)} \\
&= 1 + \frac{N_0}{N+1} \log \frac{1}{N+1} + \frac{N_1}{N+1} \log \frac{1}{N+1} \\
&\quad + \frac{N_0}{N+1} \log N_0 + \frac{N_1}{N+1} \log N_1 && \text{(G.11)} \\
&= 1 + \frac{N_0}{N+1} \log \frac{N_0}{N+1} + \frac{N_1}{N+1} \log \frac{N_1}{N+1} && \text{(G.12)} \\
&= 1 + x \log x + (1-x) \log (1-x) =: f(x) && \text{(G.13)}
\end{aligned}
$$

with $x = \frac{N_0}{N+1} \in (0,1)$. This function has a local minimum at the point $\frac{1}{2}$, whose image is $f(\frac{1}{2}) = 1 + \log \frac{1}{2} = 0$ and has no other minima. For the graph of this function see the Figure to the right. Therefore, the entropy is proven to be the stated bound. For further details see [BCW90].



□

G.2 Bounds for Lossy Encoding

As we have seen in the last section, the entropy is a lower bound for lossless coding techniques. For lossy methods, the search for such bounds is much harder. We consider the coding algorithm as a noisy transmission channel. The distortions caused by the compression coder are therefore seen as noise in this channel.

For a given random variable e a strategy to encode the obtained values with a given average distortion

$$d = E\{d(e,f)\} \tag{G.14}$$

has to be found. This technique shall also minimize the number of bits used for the transmission. Here $d(.,.)$ is a measure for the distortion (metric) between the input $e \in E$ and the output $f \in F$.

Let $p(e)$ be the probability of the occurrence of e, $p(f)$ the probability of f, $p(f|e)$ the conditional probability of f under the assumption that e is known (the probability that the symbol f is read out of the channel while e was put into the channel) and $p(e,f)$ the probability of the occurrence of e and f.

The *rate–distortion function* is defined as

$$R(d) = \min\{I(E;F)\} \tag{G.15}$$

where the minimum is taken above all codes, possessing an average distortion less or equal to the bound d. The *average trans-information* $I(E;F)$ is defined by

$$\begin{aligned}
I(E;F) &= E\{I(e;f)\} && \text{(G.16)}\\
&= E\{I(p(f)) - I(p(f|e))\} && \text{(G.17)}\\
&= \sum_{e\in E, f\in F} p(e,f)\,(I(p(f)) - I(p(f|e))) && \text{(G.18)}\\
&= -\sum_{e\in E, f\in F} p(e,f)\,(\log p(f) - \log p(f|e)) && \text{(G.19)}\\
&= -\sum_{e\in E, f\in F} p(e)p(f|e)\log\frac{p(f)}{p(f|e)} && \text{(G.20)}
\end{aligned}$$

$$= \sum_{e \in E, f \in F} p(e)p(f|e) \log \frac{p(f|e)}{p(f)}. \tag{G.21}$$

The function $R(d)$ is therefore only defined for non-negative values and is non-negative. Besides that it can be shown that $R(d)$ is monotonically decreasing, continuous and convex [RK82]. In the above calculation, the definition of *trans-information*[1] $I(e;f)$ is implicitly given in line G.17. For an example of such a rate–distortion function look for example at Figure G.1. Note that for most rate–distortion diagrams in this thesis we employed the fidelity measure PSNR instead of RMSE. We did this as an adaptation to most literature in image compression to ease comparisons with other compressors.

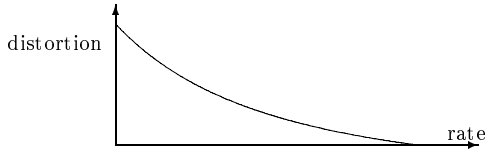


Figure G.1: Typical rate–distortion diagram.

The rate–distortion function due to Shannon gives us a lower bound for the number of required bits to transmit below a given distortion. However, this bound is only theoretical and no encoder may be constructed using this theory operating close to this bound. This theory also brings other problems to our consciousness:

- The rate–distortion function is hard to calculate for many statistical distributions.
- The definition of an optimal distortion measure matching the human visual system is an open problem (see also section C on page 169).
- The required statistics are often unknown.
- The required probabilities are often non-stationary, meaning that they take different values in different regions.
- It is hard to construct an encoder approaching the calculated limit.

[1] Because $I(e;f) = I(f;e)$ the trans-information is also called *mutual information*.

Appendix H

Image Compression with IFS Codes

The method of coding with IFS codes was kept secret for a long time. M. F. Barnsley published many articles about IFS systems for binary images. The automatic encoding algorithm for gray level images was published a few years later by A. E. Jaquin in [Jaq92].

The IFS method can be seen, as the WFA technique, as a fractal compression scheme. In literature concerning image compression the term *fractal* is used for applications using self similarity of the original image in any form for the coding process. Such self similarities can be found in many images. Take for example the image boat displayed in Figure 4.34 on page 88. This image shows a boat on a dry dock. The pylons of that boat differ clearly from the bright sky. These pylons are in general hard to code by a transform or wavelet coder because of the high frequency parts occurring at the edges of the pylons. But fractal techniques use the fact that these edges are nearly parallel and can thus copy parts of pylon edges to other parts of pylon edges. This is the main reason why fractal coders are good for encoding sharp edges while wavelet or transform coders are best for encoding rather smooth transitions. However, on the other hand, fractal encoders are well-known for their blocking artifacts.

For fractal coding, a suitable representation of a fractal has to be found approximating the given image objects. One method we have seen is WFA coding. Another representation are the so-called IFS codes, which is described in this chapter. This technique achieves high compression ratios, but the search for these codes may be very time consuming. We first explain the foundations of IFS coding in its original form: for binary images. Afterwards the automatic coding algorithm of A. E. Jaquin is presented.

H.1 Iterated Function Systems

A (two dimensional) *IFS code* (iterated function system) consists of a set of 2-tuples

$$\{(\omega_0, p_0), \ldots, (\omega_{n-1}, p_{n-1})\}, \tag{H.1}$$

whose first components consist of affine functions $\omega_i : \mathbb{R}^2 \to \mathbb{R}^2$ and whose second components are real numbers p_i with $p_i > 0$ and $\sum_{i=0}^{n-1} p_i = 1$. These numbers p_i shall from now on be called application probabilities of the functions ω_i. For the Lipschitz constants, the following *contraction condition* has to be ensured: $L_i < 1$. The Lipschitz constant L_i is defined as the smallest non-negative real number satisfying

$$||\omega_i(a) - \omega_i(b)|| \leq L_i||a - b|| \tag{H.2}$$

for all vectors a, b from the definition domain of ω_i. With $||.||$ we denote an arbitrary vector space norm. Now let $\mathcal{I}$ be an IFS code. The number

$$L = \max_{i=0}^{n-1} L_i \tag{H.3}$$

is called the *contraction factor* of the IFS code $\mathcal{I}$. Now consider the metric

$$h(A, B) = \max\left\{\max_{a\in A}\min_{b\in B}||a - b||, \max_{b\in B}\min_{a\in A}||a - b||\right\}, \tag{H.4}$$

where A and B are compact subsets of $\mathbb{R}^2$. This metric is called *Hausdorff metric*. The metric space of the compact subsets of $\mathbb{R}^2$ together with the Hausdorff metric we notate by $\mathcal{H}(\mathbb{R}^2)$.

Theorem H.1.1 *Let $\mathcal{I}$ an IFS code with contraction factor L. Then the mapping*

$$W(B) = \bigcup_{i=0}^{n-1} \omega_i(B) \tag{H.5}$$

is a contraction in the space $(\mathcal{H}(\mathbb{R}^2), h(\cdot,\cdot))$ with contraction factor L.

The fixed point $\mathcal{A}$ of the mapping $W(.)$, whose existence and uniqueness is guaranteed by the fixed point theorem of Banach, is in accordance to literature called the *attractor* of the IFS code $\mathcal{I}$. For further details about attractors see [Bar88].

H.2 Construction of an IFS Code

Let $\{(\omega_i, p_i) : i = 0, \ldots, n-1\}$ be an IFS code. The attractor is the object which should be drawn by an IFS decoder. The structure of $\mathcal{A}$ is determined by the affine mappings ω_i. Such an attractor can be constructed by overlapping

itself with affine mappings ω_i, but the determination of the mappings is in general a hard problem. As an example, consider a square. This square may be covered by itself if four squares of the half edge size are used. The probability vector can therefore be for example a vector having 1/4 in each component. This example is illustrated in Figure H.1. In general, such a perfect covering cannot be accomplished by using only a few mappings. This is the reason why an approximation commonly has to be taken.

As a measure for the goodness of such an approximation, the Hausdorff metric mentioned above can be evaluated. For a "good" approximation of the object we want to reproduce, one often has to transmit a lot of mappings, thus worsening the compression factor.

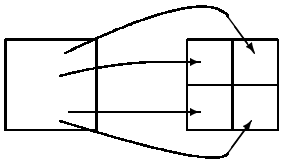


Figure H.1: Construction of an IFS code.

H.3 A Decoding Algorithm for IFS Codes

In the following, we assume that we have an IFS code and now want to determine the attractor of that code. As a starting point for the drawing algorithm a point in the attractor is required. That point can be obtained by determining a fixed point of one contraction. Now the number of iterations has to be determined. This number has to be much larger than the number of pixels of the current resolution in which the desired approximation of the attractor is to be drawn. A mapping according to the probability vector is then selected and applied to the current point. Afterwards the counter of the pixel the current point lies is incremented. Another mapping is then applied to the current point and so on until the predetermined number of iterations is reached. The pixels are afterwards painted corresponding to their counter where the reconstructed brightness is proportional to the value of the pixels' counters. A pseudo implementation of this drawing algorithm is given in listing H.1.

```
/**
* decodes an IFS code.
**/
void  decodeIFS()
  {//A is the attractor of the IFS code.
  Select a starting point (x, y) ∈ A;
  do
    {
    choose a k ∈ {0, ..., n − 1};
    (x, y) = ω_k(x, y);
```

```
    Select the rectangle R[l,m] : (x,y) ∈ R[l,m];
    R[l,m].counter++;
    } while (A is not "approximated good");
  Draw the rectangles corresponding to their counters;
  }//end drawIFS
```

Listing H.1: IFS decoding algorithm.

H.4 The Collage Theorem

The mathematical foundation for the compression with IFS codes is the collage theorem presented here without proof [Bar89].

Theorem H.4.1 (Collage Theorem) *Let $\{(\omega_i, p_i) : i = 0, \ldots, n-1\}$ be an IFS code, $L < 1$ the biggest Lipschitz constant of the affine mappings ω_i and $\epsilon > 0$ a positive real number. Let T be a compact subset of $\mathbb{R}^2$ and the mappings ω_i be selected so that*

$$h\left(T, \bigcup_{i=0}^{n-1} \omega_i(T)\right) < \epsilon. \tag{H.6}$$

Then the inequality

$$h(T, \mathcal{A}) < \frac{\epsilon}{1-L} \tag{H.7}$$

holds.

M. F. Barnsley [Bar88] uses as norm $||.||$ the Euclidean norm. IFS coding works in the way that an IFS code approximating a given image is constructed and only the mappings are stored for reconstruction of the original image.

H.5 Encoding of Gray Scale Images with IFS Systems

The algorithm described above does not work automatically and is designed for binary images. The most important step for enhancing the IFS compression scheme to an acknowledged compression scheme was the publication of A. E. Jaquin (a co-worker of Barnsley), which describes an automatical encoding algorithm for gray scale images. The technique works as follows:

At first we state image segments called *domain blocks* and *range blocks*. The range blocks have to be a covering of the original image. As transforms we can use all mappings from the set of domain blocks to the set of range blocks. For the affine functions to be contractive, the domain blocks have to be chosen in a way that they are larger than the range blocks in both dimensions. In the original implementation of Jaquin, the range blocks were chosen to partition

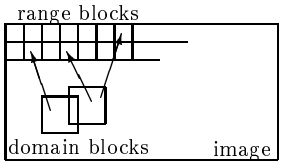


Figure H.2: IFS coding with Jaquin's method.

the image to equal sized quadratic image segments. As domain blocks, Jaquin chose blocks in the image having twice the edge lengths than the range blocks. For an illustration of this algorithm see Figure H.2.

As a refinement of these transforms, Jaquin chooses brightness and contrast scalings. This means that the brightnesses of the pixels are multiplied by a constant factor and added by a constant.

Barnsley has stated that the IFS method is capable of achieving very high compression factors. But Jaquin reports a compression efficiency comparable to compression techniques favored at the time of that writing[1]. The IFS method was developed further than the algorithm described above. On the one hand, IFS researchers have invented some interesting variants, some of them could be embedded to our WFA implementation. On the other hand, one can see some directions in IFS coding stepping toward WFA coding such as the use of matching pursuits [Gha97].

[1] Jaquin compared his scheme with vector quantization and transform coding.

Appendix I

Scalar Quantization

As we have seen in section 3.6 on page 54, the coefficients to build MP vectors have to be quantized. The problem of quantization can only be sketched shortly, for a more comprehensive description see [Max60].

Let e be a real valued random variable with density function $P(e)$. A *quantization function* maps e to a discrete valued random variable $\dot{e}$ drawn from the finite domain $\{r_0, \ldots, r_{Q-1}\}$. This mapping is often a stair function. Let $\{d_0, \ldots, d_Q\}$ be a set of (pairwise different) *decision values* or *transition values*. The values r_i and d_i are assumed as ordered ascendingly, meaning that $d_0 < d_1 < \ldots < d_Q$ holds. If e lies in the half open interval $(d_j, d_{j+1}]$, then e is mapped to $\dot{e} = r_j \in (d_j, d_{j+1}]$ for $j \in \{0, \ldots, Q-1\}$. For an illustration of this fact see Figure I.1. The *quantization problem* is to find optimal decision and reconstruction values minimizing the *quantization error*

$$D = \sum_{i=0}^{Q-1} \int_{d_i}^{d_{i+1}} f(e - r_i) P(e)\, de \tag{I.1}$$

where $P(e)$ is the density function of the random variable e and f is a non-negative, differentiable error function. For the following statements we choose $f(x) = x^2$ and assume that d_0 and d_Q are given.

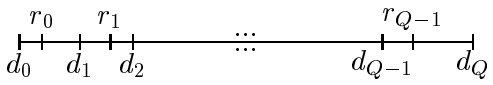


Figure I.1: Typical constellation of decision and reconstruction values.

In order to minimize the quantization error, we partially differentiate with respect to the d_i and the r_i and set the derivatives to zero to obtain necessary conditions[1] for the desired values:

[1] In order to check whether the conditions are sufficient, we have to check if the Hesse matrix of the error function is positive definite.

$$\frac{\partial D}{\partial d_i} = (d_i - r_{i-1})^2 P(d_i) - (d_i - r_i)^2 P(d_i) \stackrel{!}{=} 0 \quad \text{for } i \in \{1, \ldots, Q-1\} \tag{I.2}$$

$$\text{and } \frac{\partial D}{\partial r_i} = -2 \int_{d_i}^{d_{i+1}} (e - r_i) P(e)\, de \stackrel{!}{=} 0 \text{ for } i \in \{0, \ldots, Q-1\}. \tag{I.3}$$

From the equation system I.2 we obtain with the additional condition $P(d_i) > 0$

$$d_i = \frac{r_{i-1} + r_i}{2} \text{ for } i \in \{1, \ldots, Q-1\}, \tag{I.4}$$

meaning that the decision values lie on half length between the reconstruction values. The equation system I.3 yields the equations

$$\int_{d_i}^{d_{i+1}} (e - r_i) P(e)\, de = 0. \tag{I.5}$$

If e is uniformly distributed,

$$d_i = d_0 + (i-1)q \tag{I.6}$$

$$r_i = d_i + \frac{q}{2} \tag{I.7}$$

is obtained with $q = \frac{d_Q - d_0}{Q}$ [Jai89]. This quantizer is called *linear quantizer*. For more complicated density functions, the quantization problem is often hardly possible with this scheme. This is the reason why the quantization problem frequently has to be solved using numerical techniques: if we assume that r_0 is known, then d_1 can be calculated by using equation I.3. With this new information, the value r_1 can be calculated by equation I.4. After this the value d_3 can be calculated in the same way and so on until all required values are calculated. Since r_0 is often not known, one has to estimate this value and calculates all values inclusive d_Q. If this value is too high or too low, the values can be adjusted by varying r_0 (for example using binary search). For more details concerning quantization see for example [Max60, NH88, RK82, Llo82].

Appendix J

Other Mathematical Preliminaries

In this appendix we collect some mathematical principles which have to be added for the sake of completeness.

J.1 Common Statistical Distributions

The *normal distribution* or *Gaussian distribution* is one of the most important distributions in image processing and many other applications. The density of this distribution is given by the equation

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{J.1}$$

where μ denotes the *mean value* and σ the *variance* of this distribution (see Figure J.1).

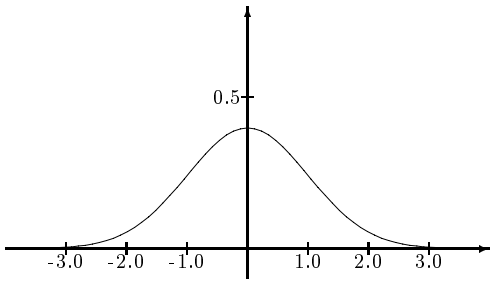


Figure J.1: Normal distribution with mean 0 and variance 1.

For AC coefficients, the *Laplacian distribution* is commonly used and is often employed for the construction of quantizers [NH88]. This distribution could

also be observed in our implementation for AC and MP coefficients. Its density function (see Figure J.2) with mean value μ and variance $\frac{\sqrt{2}}{\lambda}$ is given by

$$p(x) = \frac{\lambda}{2} e^{-\lambda|x-\mu|}. \tag{J.2}$$

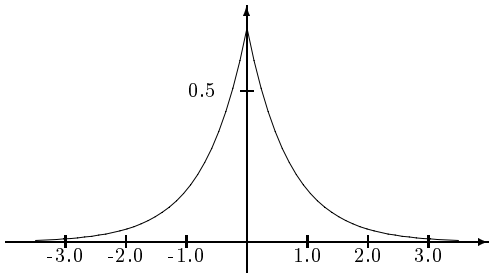


Figure J.2: Laplacian distribution with mean 0 and variance 1.

J.2 The Orthonormalization Procedure by Gram and Schmidt

The *orthonormalization procedure by Gram and Schmidt* is used to construct an orthonormal basis of a subspace of a vector space. Such a basis is often required for linear approximation.

Let E be a vector space with scalar product $\cdot$ above the field $\mathbb{R}$ and $u_0, \ldots, u_{m-1}$ a set of linearly independent vectors in E. The described procedure successively builds a basis $\{w_0, \ldots, w_{m-1}\}$. The first vector w_0 is constructed by normalization of u_0:

$$v_0 = u_0, \tag{J.3}$$
$$w_0 = v_0/||v_0||. \tag{J.4}$$

Afterwards the vectors are "bent" successively in the following way:

$$v_j = u_j - \sum_{k=0}^{j-1} (u_j, w_k) w_k, \tag{J.5}$$
$$w_j = v_j/||v_j||. \tag{J.6}$$

After this procedure, the vectors $\{u_i\}$, $\{v_i\}$ and $\{w_i\}$ span the same subspace, $\{v_i\}$ is an orthogonal system and $\{w_i\}$ even an orthonormal system. For an illustration of this process see Figure J.3.

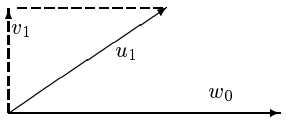


Figure J.3: Orthogonalization procedure.

J.3 Inversion of Matrices

In the early stages of the WFA encoder we had serious problems at the calculation of the inverse matrices for the basis transform. The complete construction of the matrices at the basis extension was prohibited by running time problems. Thus we extended existing transform matrices row by row using the feature of triangularity. In this extension process, we observed numerical instabilities propagated successively from step to step, despite the usage of double precision arithmetic. We solved this problem with the *post iteration algorithm of Newton* described in the following. This technique glares with the following simplicity:

$$X_{i+1} = X_i(E - R_i) \text{ with } R_i = E - AX_i \tag{J.7}$$

where X_0 is an approximation for the inverse matrix of A and E is the unity matrix in the current resolution. We observed that in our application the approximation X_1 suffices. For this reason, additional optimizations for computation speed can be made. We think that the reader does not want us to philosophize about the convergence speed of this technique. For this reason, we refer to [HLP94, MZ97, SH82] for a significantly deeper discussion of this method.

Commonly Used Formula Symbols

Symbol	Meaning
b	badness of an approximation
c	storage costs (in bits)
e	approximation error
$p_e = p(e)$	probability of the symbol $e \in E$
$p(e_1\|e_2)$	conditional probability of e_1 given e_2
$E(X)$	expected value of the random variable X
H	entropy
$I_e = I(p_e)$	information content of the symbol e with probability p_e
L_C	average codeword length of the code C
$\|.\|$	number of elements of a set
$\|\|.\|\|$	norm of a vector
$\mathbb{N}$	set of natural numbers $\{0, 1, 2, ...\}$
$\mathbb{R}$	set of real numbers
$\mathbb{Z}$	set of integer numbers $\{..., -1, 0, 1, ...\}$

Commonly Used Abbreviations

Symbol	Meaning
AC	alternating current, synonym for approximation with a non constant function
BFS	breadth first search
bpp	bits per pixel
BU	bottom up
codec	is a synonym for encoder and deccoder
CR	compression ratio
CCIR	Comité Consultatif International du Radiodiffusion
CCITT	Comité Consultatif International Télégraf et Téléphonique
DC	direct current, synonym for approximation with a constant function
DCT	discrete cosine transform
dens.	density
distrib.	distribution
DFS	depth first search
DPCM	differential pulse code modulation
GA	genetic algorithm
GUI	graphical user interface
HV	horizontal vertical
inv.	inverse
ISO	International Standardization Organization
ITU	International Telecommunication Union
JPEG	Joint Photographic Experts Group

Symbol	Meaning
MC	motion compensation
MCPE	motion compensated prediction error
MP	matching pursuit
MPEG	Motion Picture Experts Group
MSE	mean squared error
MRA	multi resolution analysis
NTSC	National Television Standards Committee
PAL	Phase Alternating Lines
PCM	pulse code modulation
pixel	picture element
PSNR	peak signal to noise ratio
RD	rate distortion
RGB	(red, green, blue) color space
RLE	run length encoding
RMSE	root mean squared error
stat.	statistical
SNR	signal to noise ratio
TD	top down
VLC	variable length coding
WFA	weighted finite automaton
WORM	write once read many

List of Figures

List of Tables

Listings

Bibliography

[AN83] N. Ahmed and T. Natarajan, *Discrete Cosine Transform.* IEEE Transactions on Computers, pp. 90–93, January 1974.

[BW95] P. Bao and X. Wu, *Near Lossless Image Compression Schemes Based on Weighed Finite Automata Encoding and Adaptive Context Modeling.* Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. In Proceedings of the IEEE, vol. 83, no. 2, pp. 930–947, February 1995.

[Bar88] M. F. Barnsley, *Fractals everywhere.* Academic Press, 1988.

[Bar88] M. F. Barnsley, *Fractal Modeling of Real World Images.* Chapter of [PS88], 1988.

[Bar89] M. F. Barnsley, *Lecture Notes on Iterated Function Systems.*, pp. 127–144 in [DK95], 1989.

[BCW90] T. C. Bell, J. G. Cleary and I. H. Witten, *Text Compression.* Prentice Hall, 1990.

[BA83] P. J. Burt and E. H. Adelson, *The Laplacian Pyramid as a Compact Image Code.* IEEE Transactions on Communications, vol. COM-31, no. 4, pp. 532–540, April 1983.

[CU] CityU Image Processing Lab, Department of Electronic Engineering, City University, Hong Kong, *Image Database.* `http://www.image.cityu.edu.hk/imagedb/`.

[Cod92] M. A. Cody, *The Fast Wavelet Transform, Beyond Fourier Transforms.* Dr. Dobb's Journal, April 1992.

[Cul] K. Culik II, *Homepage of Karel Culik.* `http://www.cs.sc.edu/~culik`.

[CK93] K. Culik II and J. Kari, *Image Compression Using Weighted Finite Automata.* Computers and Graphics, vol. 17, no. 3, pp. 305–313, May/June 1993.

[CK94] K. Culik II and J. Kari, *Image-data Compression Using Edge-optimizing Algorithm for WFA Inference.* Journal of Information Processing and Management, vol. 30, pp. 829–838, 1994.

[CK95] K. Culik II and J. Kari, *Inference Algorithms for WFA and Image Compression.* Chapter of [Fis95], pp. 243–258, 1995.

[CR96] K. Culik II and P. C. von Rosenberg, *Generalized Weighted Finite Automata Based Image Compression.* Available at [Cul], 1999.

[CV97] K. Culik II and V. Valenta, *Compression of Silhouette-like Images based on WFA.* Available at [Cul], 1997.

[CLR91] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introductions to Algorithms.* MIT Press, 1991.

[DS97] I. Daubechies and W. Sweldens, *Factoring Wavelet Transforms into Lifting Steps.* September 1996, revised November 1997, available at [Swe].

[DK95] R. L. Devaney and L. Keen, *Chaos and Fractals: The Mathematics behind the Computer Graphics.* Proceedings of Symposia in Applied Mathematics, vol. 39, American Mathematical Society, 1995.

[Eli75] P. Elias, *Universal Codeword Sets and Representations of the Integers.* IEEE Transactions on Information Theory, vol. IT-21, pp. 194–203, March 1975.

[Far90] M. F. Farelle, *Recursive Block Coding for Image Data Compression.* Springer-Verlag New York, 1990.

[Fis95] Y. Fisher (ed.), *Fractal Image Compression.* Springer-Verlag Berlin, Heidelberg, New York, 1995.

[Fis95b] Y. Fisher, *Fractal Encoding with HV Partitions.* Chapter of [Fis95], pp. 119–136, 1995.

[FG89] E. R. Fiala and D. H. Greene, *Data Compression with Finite Windows.* Communications of the ACM, vol. 32, no. 4, pp. 490–505, April 1989.

[FK99] B. Freisleben and T. Kielmann, *Object-Oriented Parallel Programming with Objective Linda.* Available at [Kie], 1999.

[GV75] R. G. Gallager and D. C. Voorhis, *Optimal Source Codes for Geometrically Distributed Integer Alphabets.* IEEE Transactions on Information Theory, vol. IT-21, pp. 228–230, March 1975.

[GHJV95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[Gha97] M. Gharavi-Alkhansari, *Fractal-Based Image and Video Coding Using Matching Pursuit.* PhD Thesis, University of Illinois at Urbana Champaign, 1997.

[GW92] R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Addison-Wesley Publishing Company, 1992.

[Haf] U. Hafner, *Homepage of Ullrich Hafner.* `http://penguinpowered.com/~ulli`.

[Haf94] U. Hafner, *Parallelisierung der WFA Bilddatenkompression unter PVM.* Diplomarbeit, Julius-Maximilians Universität Würzburg, may be downloaded at [Haf], 1994.

[Haf95] U. Hafner, *Asymmetric Coding in (m)-WFA Image Compression.* report no. 132, December 1995.

[HFUA97] U. Hafner, S. Frank, M. Unger and J. Albert, *Hybrid Weighted Finite Automata for Image and Video Compression.* report no. 160 (revised), March 1997.

[Haf99] U. Hafner, *Image and Video Coding with Weighted Finite Automata.* Dissertation, Julius-Maximilians Universität Würzburg, 1999.

[Har68] H. F. Harmuth, *A generalized Concept of Frequency and Some Applications.* IEEE Transactions on Information Theory, vol. IT-14, no. 5, pp. 375–382, May 1968.

[HQ89] W. Heise and P. Quattrocchi, *Informations- und Codierungstheorie.* Springer, 1989.

[HN86] K. Hinrichs and J. Nievergelt, *Programmierung und Datenstrukturen.* Springer Verlag Berlin, Heidelberg, 1986.

[HL87] D. S. Hirschberg and D. A. Lelewer, *Data Compression.* ACM Computing Surveys, vol. 19, no. 3, pp. 261–296, September 1987.

[How93] P. G. Howard, *The Design and Analysis of Efficient Lossless Data Compression Systems.* PhD Thesis, Brown University, Department of Computer Science, Providence, Rhode Island 1993.

[Hos86] K. Hosaka, *A new Picture Quality Evaluation Method.* Proc. International Picture Coding Symposium, Tokyo, Japan, April 1986.

[HLP94] M. Hüttenhofer, M. Lesch and N. Peyerimhoff, *Mathematik in Anwendung mit C++.* Quelle & Meyer, 1994.

[IJG] The Independent JPEG Group, *JPEG software release 6b.* `ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz`, 1998.

[Jai81] A. K. Jain, *Image Data Compression: A Review.* Proceedings of the IEEE, vol. 69, no. 3, pp. 349–389, March 1981.

[Jai81] A. K. Jain, *Advances in Mathematical Models for Image Processing.* Proceedings of the IEEE, vol. 69, no. 5, pp. 502–528, May 1981.

[Jai89] A. K. Jain, *Fundamentals of Digital Image Processing.* Prentice Hall, 1989.

[Jaq92] A. E. Jaquin, *Image Coding Based on a Fractal Theory of Iterated Contractive Image Transformations.* IEEE Transactions on Image Processing, vol. 1, no. 1, pp. 18–30, January 1992.

[Jaq93] A. E. Jaquin, *Fractal Image Coding: A Review.* Proceedings of the IEEE, vol. 81, no. 10, pp. 1451–1465, October 1993.

[Kar] J. Kari, *Homepage of Jarkko Kari.* `http://www.cs.uiowa.edu/~jjkari`.

[KF94] J. Kari and P. Fränti, *Arithmetic Coding of Weighted Finite Automata.* Theoretical Informatics and Applications, vol. 28, no. 3–4, pp. 343–360, 1994.

[Kat] F. Katritzke, *Homepage of Frank Katritzke: Research Interests.* `http://www.informatik.uni-siegen.de/~frankka`.

[Kat94] F. Katritzke, *Über die redundanzvermindernde Codierung digitaler Bilddaten.* Diplomarbeit Universität-Gesamthochschule Siegen, 1994.

[Kie] T. Kielmann, *Homepage of Thilo Kielmann.* `http://www.informatik.uni-siegen.de/~kielmann`.

[Kou95] W. Kou, *Digital Image Compression—Algorithms and Standards.* Kluwer Academic Publishers, 1995.

[Kra95] R. Krampfl, *Optimierungsansätze der WFA-Bilddatenkompression.* Diplomarbeit, Julius-Maximilians Universität Würzburg, October 1995.

[Llo82] S. P. Lloyd, *Least Squares Quantization in PCM.* IEEE Transactions on Information Theory, vol. IT-28, no. 2, pp. 129–136, March 1982.

[MZ93] S. Mallat and Z. Zhang, *Matching Pursuits with Time-Frequency Dictionaries.* IEEE Transactions on Signal Processing, vol. 41, no. 12, pp. 3397–3415, December 1993.

[Mal98] S. Mallat, *A Wavelet Tour of Signal Processing.* Academic Press, 1998.

[Max60] J. Max, *Quantizing for Minimum Distortion.* IRE Transactions on Information Theory, vol. 6, pp. 7–12, March 1960.

[MS94] W. Merzenich and L. Steiger, *Fractals, Dimension, and Formal Languages.* Theoretical Informatics and Applications, vol. 28, no. 3–4, pp. 361–386, 1994.

[MZ97] W. Merzenich and H. C. Zeidler, *Informatik für Ingenieure.* Teubner Verlag, 1997.

[Nel92] M. Nelson, *The Data Compression Book.* M&T Books, 1992.

[NL80] A. N. Netravali and J. O. Limb, *Picture Coding: A Review.* Proceedings of the IEEE, vol. 68, no. 3, pp. 366–406, March 1980.

[NH88] A. N. Netravali and B. G. Haskell, *Digital Pictures, Representation and Compression.* Plenum Press, 1988.

[Ohm95] J.-R. Ohm, *Digitale Bildcodierung.* Springer-Verlag Berlin Heidelberg 1995.

[PS93] W. A. Pearlman and A. Said, *An Image Multiresolution Representation for Lossless and Lossy Compression.* SPIE Symposium on Visual Communications and Image Processing, Cambridge, MA, November 1993.

[PS] W. A. Pearlman and A. Said, *Set Partitioning in Hierarchical Trees.* `http://ipl.rpi.edu/SPIHT`.

[PS96] W. A. Pearlman and A. Said, *A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees.* IEEE Transactions on Circuits and Systems for Video Technology, vol. 6, June 1996, also available at [PS].

[PS88] H.-O. Peitgen and D. Saupe, *The Science of Fractal Images.* Springer-Verlag New York, 1988.

[Pin90] M. Pins, *Analyse und Auswahl von Algorithmen zur Datenkompression unter besonderer Berücksichtigung von Bildern und Bildfolgen.* Dissertation, TU Karlsruhe, 1990.

[PCW74] W. K. Pratt, W.-H. Chen and L. R. Welch, *Slant Transform Image Coding.* IEEE Transactions on Communications, vol. COM-22, no. 8, pp. 1075–1093, August 1974.

[URP96] A. Rao, V. D. Pandit and R. U. Udupa, *Efficient Decoding Algorithms for Weighed Finite Automata.* Department of Computer Science, S.J. College of Engineering, Myosore, 1996.

[RT97] R.-D. Reiss and M. Thomas, *Statistical Analysis of Extreme Values.* Birkhäuser, 1997.

[Rin92] H. Ring, *Programmieren in C++.* McGraw-Hill Book Company Europe, 1992.

[RM89] J. Rissanen and K. M. Mohiuddin, *A Multiplication-Free Multialphabet Arithmetic Code.* IEEE Transactions on Communications, vol. 37, no. 2, pp. 93–98, February 1989.

[Rob95] B. F. Robinson, *Image Encoding Using Weighed Finite Automata.* Dissertation, Department of Computer Science at James Cook University of North Queensland, 1995.

[RD79] A. Rosenfeld and L. S. Davis, *Image Segmentation and Image Models.* Proceedings of the IEEE, vol. 67, no. 5, pp. 764–772, May 1979.

[RK82] A. Rosenfeld and A. C. Kak, *Digital Picture Processing.* Vol. 1, Academic Press, 1982.

[Sch98] A. Schrader, *Evolutionäre Algorithmen zur Farbquantisierung und asymmetrischen Codierung digitaler Farbbilder.* Dissertation, Universität-Gesamthochschule Siegen, 1998.

[Sed88] R. Sedgewick, *Algorithms.* Second edition, Addison Wesley, 1988.

[SP94] M. Srinivas and L. M. Patnaik, *Genetic Algorithms: A Survey.* Computer, vol. 6, pp. 17–26, 1994.

[SH82] F. Stummel and K. Hainer, *Praktische Mathematik.* Teubner Studienbücher, 1982.

[Swe] W. Sweldens, *Homepage of Wim Sweldens.* `http://cm.bell-labs.com/who/wim`.

[SS98] W. Sweldens and P. Schröder, *Building your Own Wavelets at Home.* Available at [Swe], 1998.

[SJ99] W. Sweldens and B. Jawerth, *An Overview of Wavelet Based Multiresolution Analyses.* Available at [Swe], 1999.

[Ung95] M. Unger, *Adaption der WFA-Kompression auf Videosequenzen.* Diplomarbeit, Julius-Maximilians Universität Würzburg, September 1995.

[USC] University of Southern California, *The USC-SIPI Image Database.* `http://sipi.usc.edu/services/database/Database.html`.

[Wal91] G. K. Wallace, *The JPEG Still Picture Compression Standard.* Communications of the ACM, vol. 34, no. 4, pp. 31–44, April 1991.

[Wat] University of Waterloo, Ontario, Canada, *Waterloo BragZone: Comparison of Image Compression Systems.* `http://www.uwaterloo.ca/bragzone.base.html`.

[WH71] A. Habibi and P. A. Wintz, *Image Coding by Linear Transformation and Block Quantization.* IEEE Transactions Commun. Tech., vol. COM-19, no. 1, pp. 50–62, February 1971.

[Win72] P. A. Wintz, *Transform Picture Coding.* Proceedings of the IEEE, vol. 60, no. 7, July 1972.

[WNC87] I. H. Witten, R. M. Neal and J. G. Cleary, *Arithmetic Coding for Data Compression.* Communications of the ACM, vol. 30, no. 6, pp. 520–540, June 1987.

[Zim97] J. Zimmermann, *Gewichtete endliche Automaten mit Anwendung in der Bildcodierung.* Diplomarbeit, Institut für Informatik, Albert-Ludwigs-Universität Freiburg in Breisgau, 1997.

Index