Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3143

Parallelisierung und Analyse eines kontextbasierten Entropie Koders für die Bilddatenkompression

Volkan Keles

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. Sven Simon
Betreuer:	Dipl.-Inf. Simeon Wahl
begonnen am:	17. Januar 2011
beendet am:	19. Juli 2011
CR-Klassifikation:	E.4, I.4.2, H.3.1

Abstract

Die Bilddatenkompression ermöglicht die Komprimierung von Rohdaten eines digitalen Bildes auf einen kleineren Wert. JPEG-LS ist ein Standard zum Komprimieren von Bilddaten, welcher eine sehr hohe Kompressionrate bei möglichst geringer Komplexität des Algorithmus zum Ziel hat. Dabei werden die Residuals, die durch einen statischen Prädiktor als Prädiktionsfehler erzeugt werden, mittels Kontext-Modellierung für den nachfolgenden Entropie Kodierer (Golomb-Rice-Coder) optimiert. Der Golomb-Rice-Coder verwendet für die optimale Wahl der Kodewörter ebenfalls Kontextinformationen. Die Kontext-Modellierung im JPEG-LS erschwert die Parallelisierung des Algorithmus, da die Berechnung des Prädiktionsfehler eines Pixels in Abhängigkeit bereits aufgetretener Pixel geschieht. Diese Gegebenheit führte zu der Überlegung, die Kontext-Modellierung aus dem JPEG-LS Algorithmus zu entfernen. Das Entfernen hätte Auswirkungen auf die Kodiereffizienz des kontextabhängigen Golomb-Rice-Coders und die Güte des Prädiktionsfehlers, welche wichtige Kontextinformationen zur Kodierung bzw. Optimierung fehlen würden. Das Ersetzen des Golomb-Rice-Coders durch einen anderen Entropie-Kodierer soll Abhilfe schaffen.

In dieser Arbeit wird der Datenkompressionsalgorithmus Prediction by Partial Matching (PPM) als Alternative für den Kontext-adaptiven Golomb-Rice-Coder im JPEG-LS Standard analysiert und auf Parallelisierbarkeit untersucht. Dazu wird der kontext-abhängige Golomb-Rice-Coder im JPEG-LS entfernt und durch den PPM ersetzt. Der PPM Algorithmus basiert auf Kontextmodellen und Prädiktion. Anschließend werden die Auswirkungen auf die Kompressionsrate analysiert.

Zur besseren Analyse der Kompressionsraten und der Parallelisierbarkeit des PPM-Algorithmus wurde dieser in Matlab implementiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Die Anzahl an digitalen Daten in Form von Bildern im Internet hat in den letzten Jahren enorm zugenommen. Das soziale Netzwerk Facebook zum Beispiel hat alleine 80 Milliarden Bilder auf 60.000 Servern verteilt gespeichert. Eine möglichst effiziente Speicherung der Bilddaten ist hier notwendig, da das Unternehmen dadurch Kosten für den Speicherplatz spart. Eine effiziente Komprimierung hat auch den weiteren Vorteil des höheren Datendurchsatzes, indem für die selbe Menge an Informationen weniger Bytes übertragen werden müssen. JPEG ist ein Standard, das eine effiziente Speicherung von Bilddaten ermöglicht.

Der JPEG Standard wurde 1992 von der Joint Photographic Experts Group vorgestellt. Der Standard kann Bilddaten sowohl verlustbehaftet als auch verlustfrei komprimieren [IC92]. Im verlustbehafteten Modus werden die Bilddaten durch das Ausführen von mehreren Verarbeitungsschritten, die zum Teil verlustbehaftet sind, komprimiert.

- Farbraumumrechnung RGB- nach YCbCr-Farbmodell
- Tiefpassfilterung und Unterabtastung der Farbweichungssignale Cb und Cr
- Einteilung in 8x8-Blöcke und diskrete Kosinustransformation dieser Blöcke
- Quantisierung
- Entropiekodierung

Der verlustfreie Modus des JPEG-Standards basiert auf dem einfachen prädiktiven Codierungsmodell genannt Differential Pulse Code Modulation (DPCM). DPCM ermöglicht eine Datenreduktion, indem es Differenzwerte aufeinanderfolgender Abtastwerte bildet. Die Differenzwerte werden beim JPEG-Standard mit Hilfe von benachbarten Pixelwerten, die bereits kodiert wurden gebildet. Dafür wird zunächst ein Schätzwert aus den benachbarten Pixeln gebildet und anschließend vom Wert des eigentlich zu kodierenden Pixels subtrahiert. Die sich daraus ergebenden Differenzwerte sind alle um dem Wert 0 verteilt und sind damit weniger stark korreliert als vorher. Nachdem alle Pixel bearbeitet wurden, werden die Differenzwerte mit dem Huffman-Code entropiekodiert.

Im Gegensatz zum Internet, wo sich der JPEG-Standard durchsetzen konnte und wofür es völlig ausreichend ist, werden in anderen Bereichen der Datenverarbeitung (z.B. Medizinische Bildbearbeitung) Bilddatenkompressionsverfahren ohne jeglichen Informationsverlust

verlangt. Zudem ist der verlustfreie Modus im JPEG-Standard nicht besonders effektiv und andere Standards wie der JPEG2000-Standard sind relativ rechenintensiv.

Das JBIG/JPEG-Komitee war auf der Suche nach einem effektiveren und schnelleren Komprimierungsalgorithmus, deshalb bat das Komitee Anfang 1994 Entickler um Vorschläge für einen lossless und near-lossless Komprimierungsalgorithmus. Neun Algorithmen wurden eingereicht, von denen sieben auf prädiktiven Techniken und zwei auf transformativen Techniken basierten. Gewonnen hat der von Hewlett-Packard-Mitarbeitern entwickelte Algorithmus LOCO-I (*LOw COmplexity*) vor dem CALIC-Verfahren, welcher bis zuletzt auch als Favorit galt. Obwohl das CALIC-Verfahren eine bessere Kompressionsrate hat, konnte sich der LOCO-I Algorithmus aufgrund der geringeren Komplexität durchsetzten und wurde vom JBIG/JPEG-Komitee als Basis für den neuen JPEG-LS-Standard gewählt.

Gliederung

Die restlichen Kapitel dieser Arbeit sind wie folgt strukturiert.

Kapitel 2 – JPEG-LS Im zweiten Kapitel wird der JPEG-LS genauer untersucht und ein kompakter Algorithmus vorgestellt.

Kapitel 3 – PPM In diesem Kapitel wird auf den PPM Algorithmus eingegangen, indem als erstes die Funktionsweise des Algorithmus beschrieben und anschließend ein Algorithmus entwickelt wird.

Kapitel 4 – Implementierung . Im vierten Kapitel werden die Implementierung des JPEG-LS und des adaptiven arithmetischen Kodierers vorgestellt und analysiert.

Kapitel 5 – JPEG-LS mit PPM Aufbauend auf dem Wissen aus den vorherigen Kapiteln vorgestellten Algorithmen wird hier der PPM-Algorithmus implementiert und validiert. Anschließend werden die Algorithmen JPEG-LS und PPM kombiniert.

Kapitel 6 – Evaluation In diesem Kapitel wird der modifizierte JPEG-LS aus Kapitel 5 analysiert.

2 JPEG-LS

JPEG-LS verwendet wie der verlustfreie Modus im JPEG-Standard auch einen prädiktiven Kodierungsalgorithmus um eine Dekorrelation der Pixelwerte eines Bildes zu erreichen. Die Differenzwerte, die sich aus dem Schätzwert und dem Wert des tatsächlich zu kodierenden Pixels ergeben werden mit einem kontextbasierten Entropiekodierer (Golomb-Rice-Code) kodiert. Für das bessere Verständnis werden einige

2.1 Dekorrelation durch Prädiktion

Methoden zur Dekorrelation von Bildsignalen lassen sich in zwei Hauptgruppen unterteilen. Verfahren der Signaltransformation und Filterbänke realisieren die Dekorrelation durch zerlegen der Bildsignale. Diese Methoden werden hauptsächlich in verlustbehafteten Bilddatenkompressionsalgorithmen verwendet und werden im weiteren nicht näher beschrieben. Die zweite Gruppe bildet die Prädiktion, welche unter Zuhilfenahme bereits verarbeiteter Bildsignale eine Voraussage über nachfolgende Signalwerte trifft. Die Vorgehensweise und Bezeichner, der hier vorgestellten Prädiktionsverfahren, lehnen sich an die von Strutz [Str05] an.

Die statistischen Abhängigkeiten innerhalb eines Signalwerts können durch Prädiktion von Signalwerten aufgelöst werden. Das Ziel dieser Methode ist es, durch Dekorrelation eine Ungleichverteilung der Signalwerte zu erreichen. Dabei werden die Signalamplituden um bestimmte Werte konzentriert, wodurch sich die Signalentropie verkleinert. Dies hat den Vorteil einer besseren Entropiekodierung.

Sei $x[n]$ das n-te Symbol eines zeitdiskreten Signals, das verarbeitet werden soll. Das Prädiktor-Modul (Abbildung 2.1) erzeugt einen Schätzwert $\hat{x}[n]$, indem es eine Voraussage über die Amplitude dieses Symbols trifft. Dieser Wert wird anschließend vom tatsächlichen Wert abgezogen. Der daraus resultierende Prädiktionsfehler $e[n] = x[n] - \hat{x}[n]$ ist abhängig von der Voraussage. Je besser die Voraussage ist, desto kleiner wird der Wert des Prädiktionsfehlers. Der Originalwert $x[n]$ wird durch Addition vom Schätzwert und Prädiktionsfehler wiederhergestellt. Die Prädiktionsfehler werden vom Kodierer berechnet und an den Dekodierer auf der Empfängerseite übermittelt, welcher das gleiche Prädiktor-Modul besitzt. Der Empfänger rekonstruiert das Signal durch Addition von empfangenem Prädiktionsfehler und auf Grundlage der bereits dekodierten Informationen ebenfalls berechneten Schätzwert.

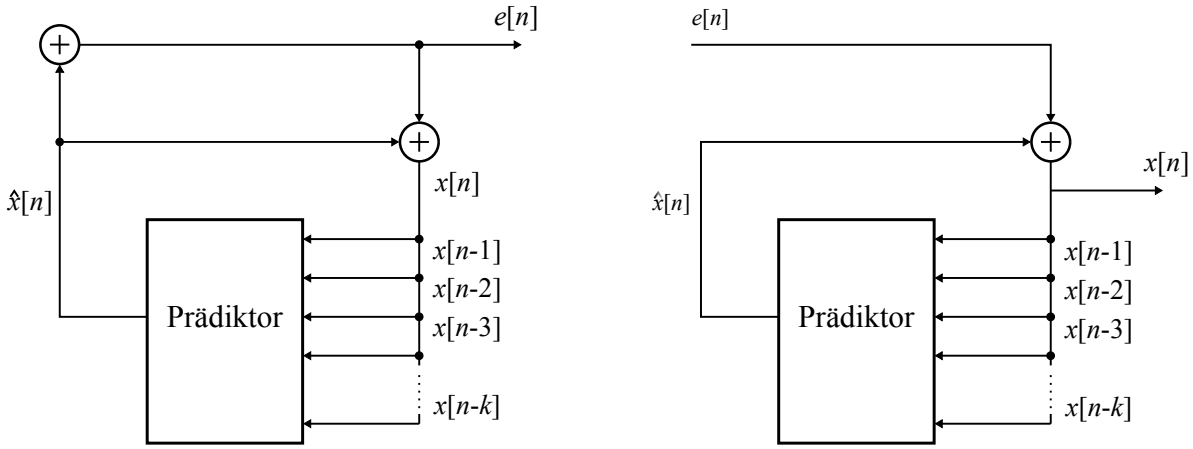

Abbildung 2.1: Prinzip der Signalprädiktion (Kodierer und Dekodierer)

2.1.1 Lineare Prädiktion

Ein einfaches Beispiel einer linearen Prädiktion ist die Voraussage eines Signalwertes durch seinen direkten Vorgänger. Die Differenz aus der Amplitude des aktuell zu kodierenden Pixels und dem des direkten Vorgängers ergibt den Prädiktionsfehler

$$e[n] = x[n] - x[n-1].$$

Im folgenden wird die Varianz des Schätzfehlersignals mit der Varianz des Originalsignals verglichen und analysiert. Voraussetzung für die Analyse ist, dass sowohl die Signale $x[n]$ und der Prädiktionsfehler mittelwertfrei sind ($\sum_n e[n] = 0$). Daraus folgt die Varianz ${\sigma_e}^2$, die dem Erwartungswert von $e^2[n]$ entspricht. Setzt man nun die Prädiktionsformel ein, erhält man

$$\begin{aligned} {\sigma_e}^2 = \mathrm{E}[e^2[n]] &= \mathrm{E}[x^2[n] - 2x[n] \cdot x[n-1] + x^2[n-1]] \\ &= \mathrm{E}[x^2[n]] - 2 \cdot \mathrm{E}[x[n] \cdot x[n-1]] + \mathrm{E}[x^2[n-1]]. \end{aligned}$$

Die Erwartungswerte $\mathrm{E}[x^2[n]]$ und $\mathrm{E}[x^2[n-1]]$ sind gleich und entsprechen der Varianz des Originalsignals ${\sigma_e}^2$. Dagegen ist der Erwartungswert $\mathrm{E}[x[n] \cdot x[n-1]]$ an der Stelle $m = 1$ gleich der Autokorrelationsfunktion $K_{xx}[m]$ von $x[n]$. Die Autokorrelationsfunktion ist für $m = 0$ gleich der Varianz des Signals. Daraus folgt

$${\sigma_e}^2 = 2 \cdot {\sigma_x}^2 - 2 \cdot K_{xx}[1] = 2 \cdot {\sigma_x}^2 \cdot \left(1 - \frac{K_{xx}[1]}{{\sigma_e}^2}\right) = 2 \cdot {\sigma_x}^2 \cdot \left(1 - \frac{K_{xx}[1]}{K_{xx}[0]}\right).$$

Für den weiteren Verlauf der Analyse ist der Korrelationskoeffizient notwendig, dazu wird die Autokorrelationsfunktion auf die Signalvarianz normiert

$$\rho_{xx}[m] = \frac{K_{xx}[m]}{K_{xx}[0]}.$$

Der Korrelationskoeffizient trifft eine Aussage über die Stärke der Korrelation eines Signalwertes mit einem Wert im Abstand von m. Mit Hilfe des Korrelationskoeffizienten lässt sich die Varianz des Prädiktionsfehlers berechnen

$$\sigma_e{}^2 = \sigma_x{}^2 \cdot (2 - 2 \cdot \rho_{xx}[1])$$

Vergleichen wir nun die Varianz des Prädiktionsfehlers mit der Varianz des Originals, sieht man eine Verkleinerung der Varianz, falls die Korrelation zwischen zwei benachbarten Signalwerten größer als 0.5 ist. Außerdem gilt, je größer die Korrelation ist, desto kleiner wird die Varianz.

Eine Verbesserung des Prädiktionsfehlers kann durch Anpassen der Vorgängeramplitude erreicht werden

$$\hat{x}[n] = a \cdot x[n-1].$$

Um einen geeigneten Wert für das Gewicht a zu finden, muss zwischen stationären und instationären Signalen unterschieden werden.

Stationäre Signale sind Signale, deren Eigenschaften wie Autokorrelationsfunktion, Mittelwert, Leistungsdichtespektrum, usw. unabhängig von der Zeit oder vom Ort sind. Bei stationäre Signalen gibt es einen optimalen Wert für das Gewicht a. Die Formel für den Prädiktionsfehler lautet $e[n] = x[n] - a \cdot x[n-1]$ und die Varianz des Prädiktionsfehlers sieht mit $\sum_n x[n] = 0$ wie folgt aus

$$\begin{aligned}
\sigma_e{}^2 &= \mathrm{E}[x^2[n] - 2ax[n] \cdot x[n-1] + a^2x^2[n-1]] \\
&= \left(1 + a^2\right) \cdot \sigma_x{}^2 - 2a \cdot K_{xx}[1] \\
&= \sigma_x{}^2 \cdot [(1 + a^2) - 2a \cdot \frac{K_{xx}[1]}{\sigma_x{}^2} \\
&= \sigma_x{}^2 \cdot [(1 + a^2) - 2a \cdot \rho_{xx}[1]].
\end{aligned}$$

Als nächstes wird untersucht wann der Kodiergewinn am größten ist. Dies ist der Fall, wenn der Faktor $[(1 + a^2) - 2a \cdot \rho_{xx}[1]]$ minimal ist. Daraus folgt das, $2a - 2\rho_{xx}[1]$ gleich 0 sein muss

$$0 = 2a - 2\rho_{xx}[1].$$

Durch Auflösen der Gleichung nach a erhalten wir den optimalen Wert $\rho_{xx}[1]$.

Setzt man nun $a = \rho_{xx}[1]$ in ${\sigma_x}^2 \cdot [(1 + a^2) - 2a \cdot \rho_{xx}[1]]$ ein, erhalten wir

$$\begin{aligned} {\sigma_e}^2 &= {\sigma_x}^2 \cdot [1 + a^2 - 2a^2] \\ &= {\sigma_x}^2 \cdot (1 - a^2) \end{aligned}$$

Anhand eines Beispiels wird gezeigt inwieweit die Gewichtung zur Verbesserung der Varianz des Prädiktionsfehlers beiträgt. Sei $\rho_{xx}[1] = 0.6$ somit folgt

$${\sigma_e}^2 = 2 \cdot {\sigma_x}^2 \cdot [1 - 0.6] = 0.8 \cdot {\sigma_x}^2$$

bzw.

$${\sigma_e}^2 = 2 \cdot {\sigma_x}^2 \cdot [1 - (0.6)^2] = 0.64 \cdot {\sigma_x}^2.$$

Instationäre Signale haben die charakteristische Eigenschaft vom betrachteten Signalabschnitt abhängig zu sein. Im Gegensatz zu stationären Signalen ist das optimale Gewicht a bei instationären Signalen nicht konstant, d.h. dass $a \rightarrow a[n]$ muss fortlaufend in Abhängigkeit vom Prädiktionsfehler $e[n] = x[n] - a[n] \cdot x[n-1]$ werden

$$a[n+1] = a[n] + \mu \cdot x[n-1] \cdot e[n].$$

Die Signaleigenschaften und Signalamplituden sind ausschlaggebend bei der Wahl eines angemessenen Wertes für das Gewicht a. Die Anpassungsgeschwindigkeit des Gewichts wird durch den Parameter μ geregelt. Dieser sollte nicht zu groß gewählt werden, da es sonst zu einem Aufschwingen kommen könnte. Um dies zu vermeiden wird ein Wert zwischen $0 < \mu < 1/{\sigma_x}^2$ vorgeschlagen.

Bilder haben die Eigenschaft, dass sich deren statistischen Eigenschaften bei der Bearbeitung schnell verändern können, z.B. bei Übergängen zwischen homogenen Bereichen und Bereichen mit vielen Kanten. Aus diesem Grund ist das Anpassen der Gewichte kaum möglich. Dieses Problem kann mit Hilfe von kontextbasierten Anpassung umgangen werden. Die Bildpunkte werden abhängig vom Kontext, dem sie zugeordnet sind adaptiert, somit wird pro Kontext nur ein Parametersatz benötigt.

2.1.2 Prädiktion von Bilddaten (2D-Daten)

Der Unterschied von Bildern gegenüber eindimensionalen Signalen ist der, dass die Signale in zwei Dimensionen gegeben sind, d.h. es können mehr Freiheitsgrade bei der Berechnung des Schätzwertes hinzugezogen werden. Der Schätzwert für einen Bildpunkt $x[n, m]$ mit den Koordinaten (n, m) berechnet sich nach

$$\hat{x}[n, m] = \sum_i \sum_j a_{i,j} \cdot x[n - i, m - j].$$

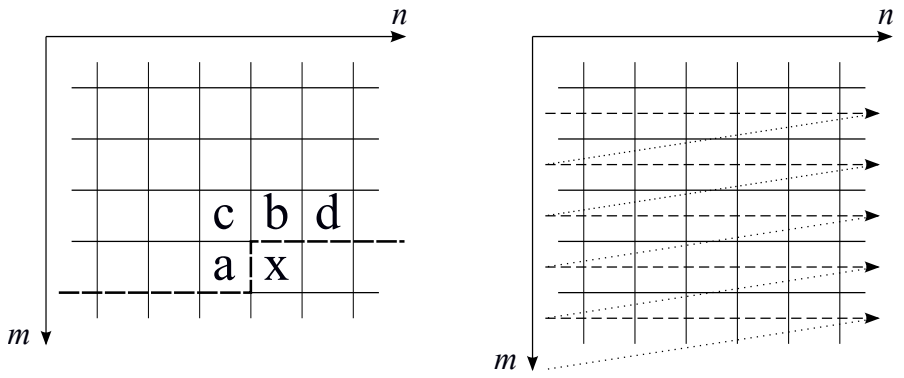

Abbildung 2.2: Prädiktion von 2D-Signalen

Bei der Prädiktion von zwei Dimensionalen Signalen ist die zeitliche Reihenfolge zu beachten, denn um eine Voraussage über ein Bildpunkt $x[n, m]$ zu treffen, dürfen nur die Bildpunkte $x[n - i, m - j]$ verwendet werden, die auch zeitlich davor liegen. In Abbildung 2.2 links wären das die Bildpunkte oberhalb der gestrichelten Linie. Die zeitliche Abfolge wird durch das Raster-line-scan-Verfahren definiert (Abbildung 2.2 rechts). Als Template wird die Menge mit den Bildpunkten bezeichnet, die bei Bestimmung des Schätzwertes verwendet werden. In Abbildung 2.2 wären dies die Bildpunkte *abcd*, die verwendet werden um den Schätzwert $\hat{x}$ vom Bildpunkt x zu bestimmen.

2.1.3 Nichtlineare Prädiktion

Im Vergleich zur linearen Prädiktion wird der Schätzwert $\hat{x}$ bei der nichtlinearen Prädiktion nicht mit einer linearen Berechnungsformel bestimmt, sondern mittels einer nichtlinearen. In Abhängigkeit der naheliegenden Bildpunkte wird hier zwischen verschiedenen Prädiktionsmodi gewählt, um ein besseres Ergebnis zu erzielen. Bei Bilddaten kann somit die Information von horizontalen und vertikalen Kanten mit in die Berechnung des Schätzwertes einfließen. Der LOCO-I Algorithmus [WSSoo] verwendet eine einfache Form der nichtlinearen Prädiktion, indem es nur drei benachbarte Bildsignale für die Kontextbildung verwendet (Bildpunkte a, b und c in Abbildung 2.2). Durch Analyse dieser Bildpunkte wird geschätzt, ob sich im aktuellen Bildausschnitt eine Kante (horizontal oder vertikal) oder ein ebener Signalverlauf befindet.

$$\hat{x} = \begin{cases} \min(a, b), & \text{falls } c \geq \max(a, b) \\ \max(a, b), & \text{falls } c \leq \min(\text{a,b}) \\ a + b - c, & \text{sonst.} \end{cases} \tag{2.1}$$

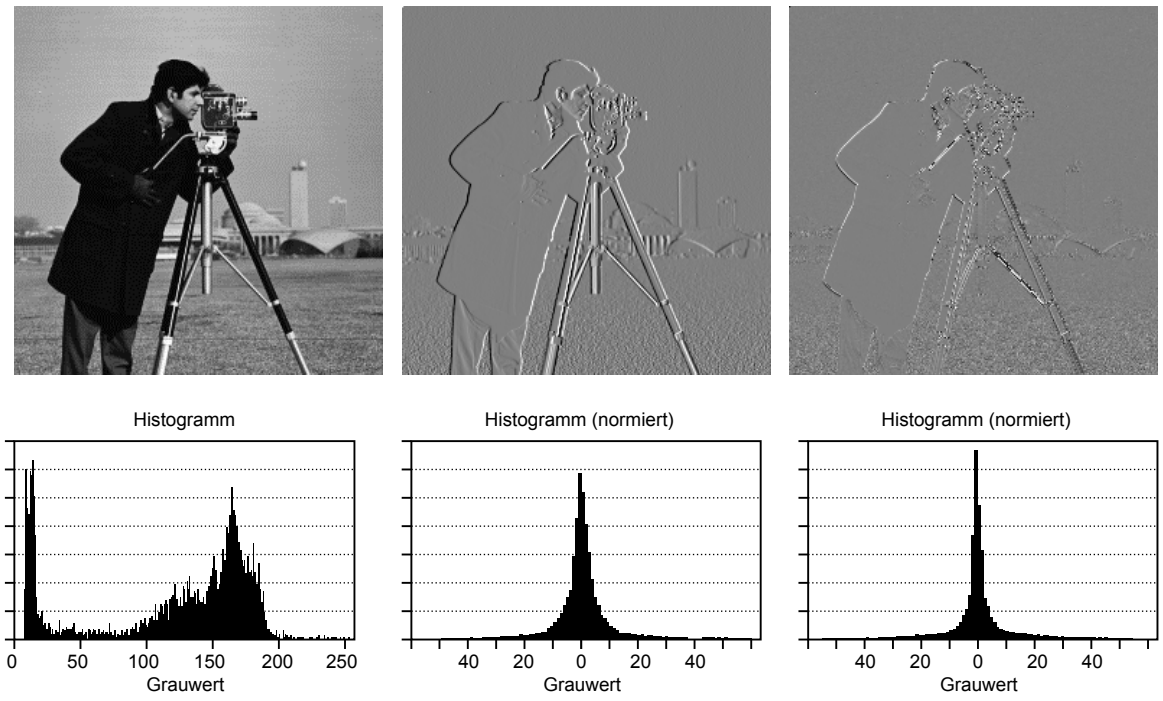

Abbildung 2.3: Original(links), einfache Prädiktion(mitte), nichtlineare Prädiktion(rechts)

Ist der Wert des Bildpunktes c größer als die Werte von a und b und a der kleinste Wert, so wird eine horizontale Kante angenommen. In diesem Fall wird als Prädiktionswert a gewählt, da der aktuelle Bildpunkt x mit großer Wahrscheinlichkeit auf der selben Kante liegt wie a. Wenn c kleiner als a und b ist und a der größte Wert, handelt es sich wieder um eine horizontale Kante, mit dem Unterschied, dass das Vorzeichen des Gradienten umgekehrt ist. Analog hierzu erfolgt die Ermittlung von vertikalen Kanten durch das Vertauschen von a und b. Für den Fall das c zwischen a und b liegt, wird für den Schätzwert ein Annäherungswert aus den Werten a, b und c gebildet. Eine Möglichkeit den Schätzwert zu verbessern ist die Addition eines Ausgleichswertes (Bias) (siehe Kapitel ...).

Abbildung 2.3 zeigt die Häufigkeitsverteilung (Histogramm) eines Beispielbilds für das Originalbild, einer einfachen Prädiktion $x[n,m] = x[\hat{n}-1,m] = a$ und der nichtlinearen Prädiktion nach Gleichung (2.2). Auf den Histogrammen der Grauwerte sieht man eine Konzentration der Signalwerte um den 0-Wert. Dies deutet auf eine Reduzierung der Varianz gegenüber dem Originalbild. Die Varianz ist bei der nichtlinearen Prädiktion am kleinsten.

2.1.4 Prädiktionsfehler im Detail

In diesem Kapitel wird der Wertebereich des Prädiktionsfehlers untersucht. Der Prädiktionsfehler wird aus der Differenz des originalen Signalwerts und dem Schätzwert $\hat{x}$ bestimmt

$$e = x - \hat{x}.$$

Daraus ergibt sich ein theoretischer Wertebereich von $-x_{\max} \leq e \leq x_{\max}$ für den Prädiktionsfehler. In der Praxis ist dieser Bereich jedoch kleiner, da dieser nämlich vom Schätzwert abhängt

$$0 - \hat{x} \leq e \leq x_{\max} - \hat{x}.$$

Diese Eigenschaft wird genutzt um die Varianz des Schätzfehlers weiter zu reduzieren, indem der Wertebereich durch Division auf $\lfloor (x_{\max}+1)/2 \rfloor \leq e \leq \lfloor x_{\max}/2 \rfloor)$ begrenzt wird.

$$\begin{array}{lll} \text{if} & (e > \lfloor x_{\max}/2 \rfloor) & e' := e - (x_{\max}+1) \\ \text{elseif} & (e < -\lfloor (x_{\max}+1)/2 \rfloor) & e' := e - (x_{\max}+1) \\ \text{else} & & e' := e \end{array}$$

Der Empfänger erkennt die Division durch Überprüfen des Wertebereiches, liegt der rekonstruierte Wert $x' = e' + \hat{x}$ außerhalb des gültigen Wertebereichs wird er mit der Division wieder angepasst

$$\begin{array}{lll} \text{if} & (x' > x_{\max}) & x := x' - (x_{\max}+1) \\ \text{elseif} & (x < 0) & x := x' + (x_{\max}+1) \\ \text{else} & & x := x' \end{array}$$

Beispiel Sei $0 <= 0 <= 255$ der Wertebereich der Pixel eines Graustufenbildes. Angenommen ein Pixel im Bild mit dem Wert x=15 wird durch einen zu hohen Schätzwert von $\hat{x} = 175$ vorausgesagt. Damit ergibt sich ein beschränkter Wertebereich des Prädiktionsfehlers von

$$\begin{array}{rll} -\lfloor (255+1)/2 \rfloor & \leq e' \leq & \lfloor 255/2 \rfloor \\ -128 & \leq e' \leq & 127 \end{array}$$

Daraus folgt der Schätzfehler

$$e = x - \hat{x} = 15 - 175 = -160.$$

Der Schätzwert liegt außerhalb des gültigen Wertebereichs. Es gilt:

$$(e < -128) \Rightarrow e' = e + (255+1) = 96.$$

Dem Empfänger wird der Schätzfehler $e' = 96$ übermittelt, welcher den ursprünglichen Signalwert wie folgt berechnet

$$x' = e' + \hat{x} = 96 + 175 = 271$$

Dabei gilt für x'

$$(x' > 255) \Rightarrow x = x' - (255 + 1) = 15.$$

Der rekonstruierte Wert lautet 15 und ist damit richtig.

2.2 Golomb-Code

Der Golomb-Code wurde 1966 vom amerikanischen Mathematiker und Ingenieur Solomon W. Golomb vorgestellt [Gol66]. Der Code ist besonders für Zeichen mit einer geometrischen Wahrscheinlichkeitsverteilung geeignet. Dabei werden kleine Zahlen mit wenigen Bits und größere Zahlen mit vielen Bits dargestellt. Dies wird durch einen Parameter gesteuert. Die Anzahl der benötigten Bits ist abhängig von der Größe des Parameters. Je größer dieser ist, desto langsamer wächst die Anzahl, umso mehr Bits werden für kleine Zahlen benötigt.

Der Golomb-Code eines nicht-negativen ganzzahligen Wertes (Integer) n hängt von der Wahl des Parameters m ab. Als erstes werden für die Berechnung des Golomb-Codes eines nicht-negativen Integer-Wertes der Quotient q, der Rest r und c bestimmt.

$$q = \left\lfloor \frac{n}{m} \right\rfloor, \quad r = n - q \cdot m, \quad c = \lceil log_2 m \rceil$$

Anschließend wird der aus zwei Teilen zusammengesetzte Kode berechnet. Der erste Teil ist unär und besteht aus q 1-Bits gefolgt von einem 0-Bit, wobei der zweite Teil binär ist und in Abhängigkeit von c bestimmt wird. Für den Fall, dass $r < 2^c$ gilt, wird r als Binärcode mit der Länge $c - 1$ als zweiter Teil bestimmt. Falls $r \geq 2^c - b$ gilt, wird r als Binärcode mit der Länge c bestimmt.

Beispiel: Sei $b = 4$ und $n = 6$ dann gilt $q = \left\lfloor \frac{6}{4} \right\rfloor = 1$. Daraus ergibt sich für $r = 6 - 1 \cdot 4 = 2$ und $c = \lceil log_2 4 \rceil = 2$. Folglich besteht der erste Teil des Codewortes aus einem 1-Bit gefolgt von einem 0-Bit "10" und der zweite Teil besteht aus der Bitfolge "10" (siehe Tabelle 2.1).

Im JPEG-LS wird eine Variante des Golomb-Codes namens Rice-Code verwendet. Der Parameter b ist im Rice-Code eine Potenz von 2 $b = 2^k$, welches sich einfacher durch Bitshiften und logischen Operationen umsetzten lässt. Dabei gilt für q und r

$$q = n \gg k, \quad r = n \wedge (b - 1).$$

Die Reihenfolge in der das Kodewort zusammengesetzt wird, hat keine größere Bedeutung, solange der Kodierer und Dekodierer übereinstimmen. In diesem Fall entspricht der erste Teil des Codes den letzten k Bits des zu kodierenden Zahlenwertes n. Der zweite Teil

r	m=1	m=2	m=3	m=4
0	0	0 0	0 0	0 00
1	10	0 1	0 10	0 01
2	110	10 0	0 11	0 10
3	1110	10 1	10 0	0 11
4	11110	110 0	10 10	10 00
5	111110	110 1	10 11	10 01
6	1111110	1110 0	110 0	10 10
7	11111110	1110 1	110 10	10 11
8	111111110	11110 0	110 11	110 00
9	1111111110	11110 1	1110 0	110 01
10	11111111110	111110 0	1110 10	110 10
⋮	⋮	⋮	⋮	⋮

Tabelle 2.1: Golomb-Codes für Parameter m=1,2,3 und 4

besteht aus dem Rest $q = n \gg k$ und wird unär ausgegeben. Der Parameter k kann durch Codierungsadaption optimal gewählt werden. Im JPEG-LS Algorithmus wird dies mit Hilfe der Kontext-Parameter umgesetzt.

r	k=0	k=1	k=2	k=3
0	0	0 0	00 0	000 0
1	10	1 0	01 0	001 0
2	110	0 10	10 0	010 0
3	1110	1 10	11 0	011 0
4	11110	0 110	00 10	100 0
5	111110	1 110	01 10	101 0
6	1111110	0 1110	10 10	110 0
7	11111110	1 1110	11 10	111 0
8	111111110	0 11110	00 110	000 10
9	1111111110	1 11110	01 110	001 10
10	11111111110	0 111110	10 110	010 10
⋮	⋮	⋮	⋮	⋮

Tabelle 2.2: Rice-Codes für Parameter k=0,1,2 und 3

Beispiel: Sei $n = 7$ (binär ”111”) und $k = 2$, dann lauten die untersten k Bits der binären Darstellung von n”11”. Diese Bits werden ausgegeben gefolgt von der Bitfolge ”10”, welcher sich durch die Bitshift-Operation $n \gg k$ und der unären Darstellung der sich daraus

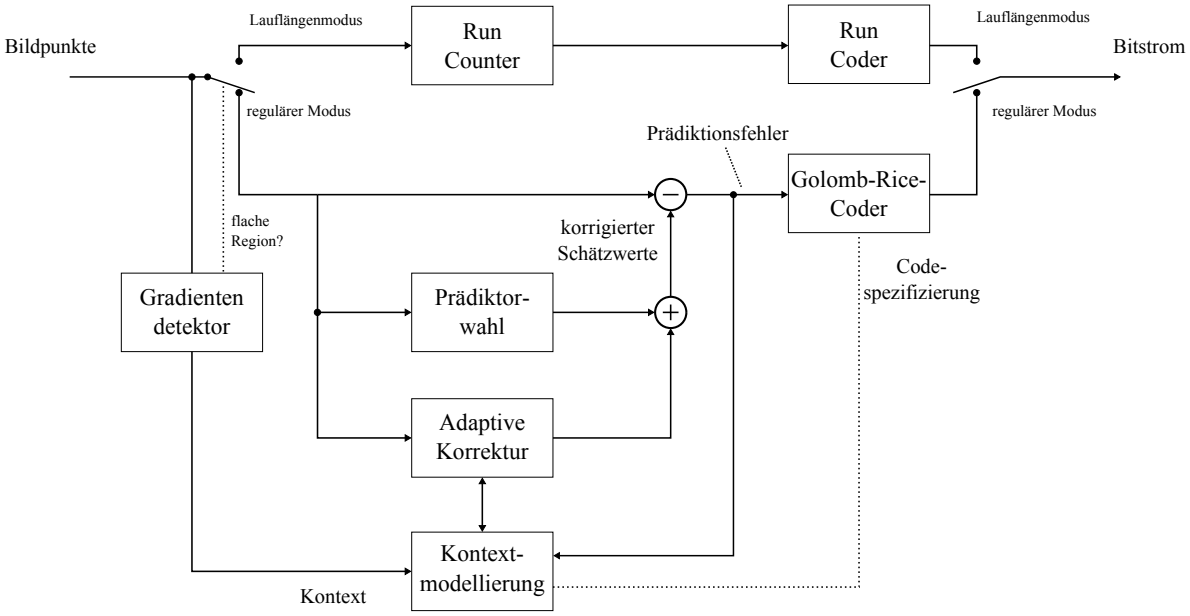

Abbildung 2.4: Modell vom JPEG-LS Algorithmus

ergebenden Zahl 1 ergibt (Tabelle 2.2). Für die Darstellung des unären Teils können die Bits auch vertauscht werden.

2.3 JPEG-LS Algorithmus

Abbildung 2.4 zeigt das Modell des JPEG-LS. Die Aufgabe des Gradientendetektors ist die Analyse der Umgebung des aktuell zu kodierenden Bildpunktes und die Festlegung des Kontextes für die Kodierung. Abhängig vom Signalverlauf aktiviert der Gradientendetektor entweder den Lauflängenmodus (run mode) oder den regulären Modus (regular mode). Der Lauflängenmodus wird aktiviert, wenn eine konstanter Signalverlauf erkannt wird. Anschließend werden die Bildsignale mit einem Lauflängenkodierer kodiert. Für den Fall, dass kein konstanter Signalverlauf erkannt wird, wird durch Kantendetektion (Kapitel 2.1) eines der drei Prädiktoren gewählt (MED-Prädiktor). Der Schätzwert wird durch die Addition eines Ausgleichswertes $(off[cx])$, der in Abhängigkeit vom Kontext gebildet wird, verbessert (Adaptive Korrektur)

$$\hat{x} = off[cx] + \begin{cases} \min(a,b), & \text{falls } c \geq \max(a,b) \\ \max(a,b), & \text{falls } c \leq \min(\text{a,b}) \\ a+b-c, & \text{sonst.} \end{cases} \tag{2.2}$$

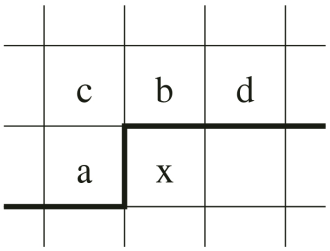

Abbildung 2.5: JPEG-LS Template

Der Prädiktionsfehler wird gebildet indem der korrigierte Schätzwert vom Wert des originalen Bildpunktes abgezogen wird. Der Golomb-Rice-Coder, der den Prädiktionsfehler kodiert, ist kontextabhängig und wird über den Kontextmodellierer gesteuert.

2.3.1 Initialisierungsphase

Während der Initialisierungsphase werden relevante Parameter für die Kodierung definiert. Für die Wertebereichsbegrenzung des Prädiktionsfehlers wird der Wertebereich der zu verarbeitenden Werte benötigt. Dazu wird der Parameter RANGE eingeführt mit

$$\text{RANGE} := \left\lfloor \frac{\text{MAXVAL} + 2 \cdot \text{NEAR}}{\text{NEAR} \cdot 2 + 1} \right\rfloor + 1.$$

Bei dem NEAR-Wert handelt es sich um einen Parameter, der die Art der Komprimierung festlegt. Für $\text{NEAR} = 0$ codiert der Algorithmus verlustfrei. Dagegen wird für $\text{NEAR} > 0$ verlustbehaftet ($\pm$NEAR) codiert. MAXVAL wird der Wert des größten zu codierenden Bildsignales zugewiesen. Vier weitere Parameter in Form von Arrays werden benötigt um die 367 Kontexte zu modellieren. Die ersten 355 Kontexte sind für den regulären Modus bestimmmt und die restlichen zwei $[365, 366]$ für den Lauflängenmodus. $A[0 \ldots 366]$ enthält die Summe aller Prädiktionsfehlerbeträge pro Kontext, $B[0 \ldots 366]$ einen Wert für das Ausgleichen des Bias, $C[0 \ldots 366]$ Werte für die Korrektur des Schätzwertes und $N[0 \ldots 366]$ die aktuelle Anzahl von Ereignissen pro Kontext. Die Elemente von B und C werden zu Beginn des Algorithmus auf Null gesetzt, die von $N[...]$ auf 1. Letztlich müssen noch die Elemente von $A[...]$ definiert werden mit

$$\max\left(2, \left\lfloor \frac{\text{RANGE} + 2^5}{2^6} \right\rfloor\right).$$

Der RESET Parameter hilft bei der Adaption des Algorithmus an die Signalstatistik. Dafür wird im Laufe der Codierung $N[...]$ ständig mit dem Wert von RESET verglichen. Falls RESET

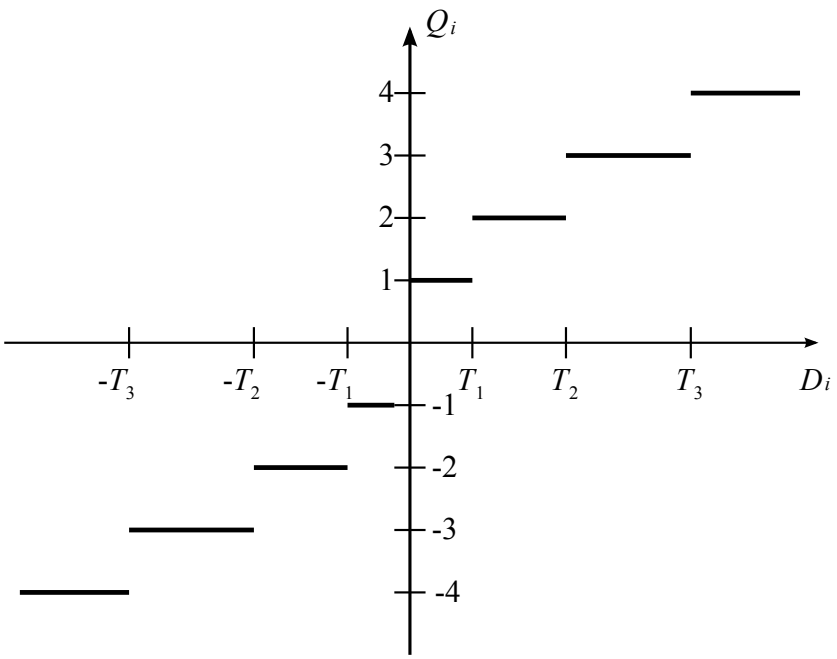

Abbildung 2.6: Quantisierung der Gradienten

überschritten wird, werden die Werte von $A[...]$, $B[...]$ und $N[...]$ halbiert. Die Wahl eines korrekten Wertes muss gut überlegt sein. Ein kleiner Wert kann zwar die Reaktionszeit der Anpassung verringern, dafür kann es aber auch zu Ausreißern in der Statistik kommen. Als Standardwert für RESET wird 64 angegeben.

Zwei weitere Parameter werden für lange Rice-Codewörter benötigt. qbpp $= \lceil \log_2(\text{RANGE}) \rceil$ und LIMIT $= (2 \cdot (bpp + \max(8, bpp) - qbpp - 1)$, mit bpp $= \max(2, \lceil log_2(\text{MAXVAL} + 1) \rceil)$.

Die Anzahl der Kontexte, wird durch Quantisierung begrenzt, da ohne Quantisierung sich bei einer Abhängigkeit von n Nachbarn und bei 8 Bits pro Bildpunkt 256^n verschiedene Kontexte ergeben würden. Die Quantisierung der Kontexte im JPEG-LS Algorithmus benötigt drei Schwellwerte $T1 = 3, T2 = 7$ und $T3 = 21$.. Für die Schwellwerte gilt

$$\begin{aligned} NEAR + 1 \quad &\leq T1 \leq \quad MAXVAL \quad \text{or} \quad T1 = 0, \\ T1 \quad &\leq T2 \leq \quad MAXVAL \quad \text{or} \quad T2 = 0, \\ T2 \quad &\leq T3 \leq \quad MAXVAL \quad \text{or} \quad T3 = 0. \end{aligned}$$

2.3.2 Kontextmodellierung

Gradientendetektion

Sowohl die Gradienten als auch die Kontexte werden mit Hilfe des Templates in Abbildung 2.5 ermittelt. Für die Bildpunkte an den Randpositionen müssen Sonderregeln definiert werden, da diesen Bildpunkten Informationen des Umfeldes fehlen. Deshalb werden für Bildpunkte der ersten Zeile b, c und d der Wert 0 zugewiesen. Bei Bildpunkten der ersten Spalte wird a der Wert von b und c der Wert von a aus der Vorgängerzeile zugewiesen. Zuletzt wird für Bildpunkte der letzten Spalte d mit b belegt.

Bei der Gradientendetektion werden die Gradienten in der naheliegenden Umgebung des zu kodierenden Bildpunktes ermittelt

$$D1 := Rd - Rb, D2 := Rb - Rc, D3 := Rc - Ra.$$

Wobei die Bezeichner Ra, Rb, Rc und Rd die rekonstruierten Bildpunkte darstellen und völlig identisch mit denen aus Abbildung 2.5 [IT98] sind. Der Lauflängenmodus wird gewählt, wenn alle Gradienten den Wert gleich Null haben bzw. kleiner gleich dem NEAR-Wert sind, trifft dies nicht zu wird der reguläre Modus gewählt.

Kontextbestimmung

In diesem Abschnitt wird der Ablauf der Kodierung im regulären Modus beschrieben. Dazu werden zunächst die Kontexte, welche in Abhängigkeit der Gradienten $D1$, $D2$ und $D3$ gebildet werden, bestimmt. Wie bereits in Kapitel 2.3.1 erwähnt, kann die Menge aller Kombinationen enorme Werte annehmen. Aus diesem Grund werden die Gradienten durch die Schwellenwerte $T1$, $T2$ und $T3$ auf $Q1$, $Q2$ und $Q3$ abgebildet. Der Kontext wird somit durch den Vektor $(Q1, Q2, Q3)$ gebildet. Die Anzahl der Kontexte wird durch die Quantisierung auf $9 \cdot 9 \cdot 9 = 729$ reduziert.

$$\begin{array}{lll}
\text{if} & (Di \leq -T3) & Qi = -4 \\
\text{else if} & (Di \leq -T2) & Qi = -3 \\
\text{else if} & (Di \leq -T1) & Qi = -2 \\
\text{else if} & (Di \leq -NEAR) & Qi = -1 \\
\text{else if} & (Di \leq NEAR) & Qi = 0 \\
\text{else if} & (Di < T1) & Qi = 1 \\
\text{else if} & (Di < T2) & Qi = 2 \\
\text{else if} & (Di < T3) & Qi = 3 \\
\text{else} & & Qi = 4
\end{array}$$

Die Kontextnummer kann mit der Formel $cx' = 9 \cdot (9 \cdot Q1 + Q2) + Q3$ berechnet werden. Durch eine Vorzeichen/Betrag-Darstellung kann die Anzahl unterscheidbarer Kontexte nochmals auf 365 verringert werden. Dabei gilt für $sign$ und cx'

$$\begin{array}{lllll} \text{if} & (cx' < 0) & cx = -cx' & \wedge & sign = -1 \\ \text{else} & & cx = cx' & \wedge & sign = 1 \end{array}$$

Prädiktion

Der Schätzwert des aktuellen Wertes x wird durch einen nichtlinearen Prädiktor (Kapitel 2.2) bestimmt

$$\begin{array}{lll} \text{if} & (Rc \geq \max(Ra, Rb)) & \hat{x} = min(Ra, Rb) \\ \text{else if} & (Rc \leq \min(Ra, Rb)) & \hat{x} = max(Ra, Rb) \\ \text{else} & & \hat{x} = Ra + Rb - Rc. \end{array}$$

Anschließend wird der hieraus resultierende Schätzwert kontextsensitiv korrigiert und auf den gültigen Wertebereich überprüft

$$\begin{array}{lll} \text{if} & (sign == +1) & \hat{x} = \hat{x} + C[cx] \\ \text{else} & & \hat{x} = \hat{x} - C[cx] \\ \text{else} & & \hat{x} = Ra + Rb - Rc \end{array}$$

$$\begin{array}{lll} \text{if} & (\hat{x} > MAXVAL) & \hat{x} = MAXVAL \\ \text{else if} & (\hat{x} < 0) & \hat{x} = 0. \end{array}$$

Nachdem der korrigierte Schätzwert berechnet wurde wird der Prädiktionsfehler in Abhängigkeit vom Vorzeichen berechnet(Gl.2.3) und später quantisiert(Gl.2.4)

$$\begin{array}{lll} \text{if} & (sign < 0) & e = \hat{x} - x \\ \text{else} & & e = x - \hat{x} \end{array} \tag{2.3}$$

$$\begin{array}{lll} \text{if} & (e > 0) & e = \quad (\text{NEAR} + e)/\text{NEAR} \cdot 2 + 1 \\ \text{else} & & e = -(\text{NEAR} - e)/\text{NEAR} \cdot 2 + 1 \end{array} \tag{2.4}$$

Der Wertebereich des Prädiktionsfehlers e liegt theoretisch zwischen -MAXVAL und +MAXVAL. Aufgrund der Tatsache, dass der Schätzwert $\hat{x}$ sowohl dem Kodierer als auch dem Dekodierer bekannt ist, kann der Wertebereich durch eine Modulo Rechenoperation auf einen Bereich von $-\lfloor \text{RANGE}/2 \rfloor$ und $\lceil \text{RANGE}/2 \rceil$ eingeschränkt werden.

$$\begin{array}{lll} \text{if} & (e < -\text{RANGE}/2)) & e = e + \text{RANGE} \\ \text{else if} & (e > (\text{RANGE} - 1)/2) & e = e - \text{RANGE} \end{array}$$

Die Rekonstruktion des Bildpunktes auf der Empfängerseite geschieht durch

$$\begin{array}{lll} \text{Rx} = \hat{x} + sign \cdot \text{e} \cdot (\text{NEAR} \cdot 2 + 1\ ) & & \\ \text{if} & \text{(Rx>MAXVAL)} & \text{Rx= MAXVAL} \\ \text{else if} & \text{(Rx< 0)} & \text{Rx= 0} \end{array}$$

Kodieren des Prädiktionsfehlers

Bevor der Golomb-Rice-Coder mit der Kodierung des Prädiktionsfehlers beginnen kann, müssen zum einen die Prädiktionsfehler mit negativem Wert auf einen positiven Wertebereich abgebildet (Listing 2.1) werden und zum anderen der kontextabhängige Kodierungsparameter k berechnet werden. Die Berechnung des Parameters k in Abhängigkeit vom Kontext soll bei der Auswahl einer geeigneten Kodetabelle helfen. Für die Berechnung von k wird angenommen, dass k Bits zur Speicherung vom mittleren absoluten Prädiktionsfehler ausreichen $2^k \geq A[cx]/N[cx] \Leftrightarrow N[cx] \cdot 2^k \geq A[cx] \Leftarrow (N[cx] \ll k) \geq A[cx].$) Nun kann k einfach bestimmt werden, indem k so lange erhöht wird bis die Bedingung erfüllt ist

$$\text{for}(k = 0; (N[cx] << k) < A[cx]; k++).$$

```
if ((NEAR == 0)&&(k == 0)&&(2 · B[cx] ≤ N[cx]))
{
    if (e < 0) e_M = −2 · (e + 1)
    else       e_M =  2 · e + 1
}
else
{
    if (e < 0) e_M = −2 · e − 1
    else       e_M =  2 · e
}
```

Listing 2.1: Wertebereichskorrektur

2.1 Für die Kodierung von e_M werden zunächst die k untersten Bits abgeschnitten um den Restwert $y = e_M \gg k$ zu ermitteln. Darauf werden y Bits mit Null gefolgt von einem Bit mit dem Wert 1 ausgegeben. Danach werden die untersten k Bits von e_M ausgegeben. Durch Auftreten von sehr großen y Werten kann das Codewort entsprechend lang werden. Dieser Fall wird durch begrenzen von y auf LIMIT Bits mit Null verhindert. Der eigentliche Wert wird mit $qbpp$ Bits übertragen.

Aktualisierung der Kontextvariablen

Nun fehlt noch die Aktualisierung der Variablen $A[cx], B[cx], C[cx]$ und $N[cx]$, durch deren Hilfe der Kodierungsparameter k und die Schätzwerte bestimmt werden. Die Variablen werden nach jedem Kodierungsschritt der Statistik des Bildes angepasst. Für $A[cx], B[cx]$und$N[cx]$ geschieht dies wie folgt

$$
\begin{aligned}
&A[cx] = A[cx] + abs(e)\\
&B[cx] = B[cx] + e\\
&\\
&\text{if } (N[cx] == \text{RESET})\\
&\{\\
&\quad A[cx] = A[cx] \gg 1\\
&\quad N[cx] = N[cx] \gg 1\\
&\quad \text{if } (B[cx] \geq 0)\ B[cx] = B[cx] \gg 1\\
&\quad \text{else } B[cx] = -((1 - B[cx]) \gg 1)\\
&\}\\
&N[cx] = N[cx] + 1
\end{aligned}
$$

Listing 2.2: Skalierung der Variablen A, B und N

$$
\begin{aligned}
&\text{if } (B[cx] \leq -N[cx])\\
&\{\\
&\quad \text{if } (C[cx] > -128)\ C[cx] = C[cx] - 1\\
&\quad B[cx] = B[cx] + N[cx]\\
&\quad \text{if } (B[cx] \leq -N[cx])\ B[cx] = 1 - N[cx]\\
&\}\\
&\text{else if } (B[cx] > 0)\\
&\{\\
&\quad \text{if } (C[cx] < 127)\ C[cx] = C[cx] + 1\\
&\quad B[cx] = B[cx] - N[cx]\\
&\quad \text{if } (B[cx] > 0)\ B[cx] = 0\\
&\}
\end{aligned}
$$

Listing 2.3: Kalkulation des Korrekturwertes C

Der Korrekturwert $C[cx]$ sorgt dafür, dass sich kein Mittelwert ungleich Null innerhalb eines Kontextes einstellt.

2.3.3 Lauflängenmodus

Die Lauflängenkodierung ist gut geeignet für die verkürzte Darstellung von Sequenzen mit dem gleichen Wert. Die Gradienten $D1, D2, D3$ entscheiden ob der Lauflängenmodus gewählt wird oder nicht. Für eine Lauflängenkodierung müssen alle Gradienten für den verlustfreien Modus $(losslessmode)$ gleich Null bzw. für den verlustbehafteten Modus $(near - losslessmode)$ kleiner gleich $NEAR$ sein. Zunächst wird die Lauflänge ermittelt. Der Kodierer liest dazu die Bildpunkte solange weiter, bis die Amplitude des aktuell gelesenen Bildpunktes sich um mehr als $NEAR$ vom Anfangswert unterscheidet. Die Lauflänge wird in Segmente, die variabel sind und sich der Statistik des Bildes anpassen, der Länge (2^l) aufgeteilt und übertragen.

Beispiel: Angenommen, die Lauflänge r sei 11 und die Segmentlänge liege aktuell bei $s = 2^3 = 8$. Nun wird geprüft wie oft s in r passt und anschließend die Bitfolge ausgegeben. Da s einmal in r passt wird ein 1-Bit gefolgt von einem 0-Bit ausgegeben. Der zweite Teil besteht aus dem verbleibenden Rest $11 - 1 \cdot 8 = 3$ mit der Anzahl von l Bits. Die komplette Bitfolge für das Codewort sieht am Ende so aus "10011". Die Segmentlänge ist variabel und kann sich im Laufe der Kodierung einer Lauflänge ändern (Adaption).

2.3.4 Performance

In diesem Abschnitt wird die Performance des JPEG-LS Algorithmus mit andere Bilddatenkompressionsverfahren verglichen und die Möglichkeit der Parallelisierung des Algorithmus betrachtet. Dazu werden die Ergebnisse des JPEG-LS Kompression mit denen verschiedener Bilddatenkompressionsverfahren verglichen. Die Tabelle 2.3 zeigt, dass alternative Kompressionsverfahren mit ungefähr gleicher Komplexität (z.B. FELICS, PNG, JPEG-Huffman) vom LOCO-I/JPEG-LS Duo geschlagen werden [Gol66]. Verglichen mit dem komplexen und starken $CALIC$-Verfahren ist JPEG-LS bei geringer Komplexität nur um wenige Prozent schlechter. Die Parallelisierung des Algorithmus ist wegen der Kontextabhängigkeiten zwischen den Bildpunkten, die der Kontext-Modeller erzeugt, schlecht umzusetzen. Folglich wird im weiteren Verlauf der Arbeit, die Kontextmodellierung genauer analysiert und eine eventuelle Entfernung dieser in Betracht gezogen.

Image	LOCO-I	JPEG-LS	FELICS	Lossless JPEG Huffman	CALIC arithm.	PNG
bike	3.59	3.63	4.06	4.34	3.50	4.06
cafe	4.80	4.83	5.31	5.74	4.69	5.28
woman	4.17	4.20	4.58	4.86	4.05	4.68
tools	5.07	5.08	5.42	5.71	4.95	5.38
bike3	4.37	4.38	4.67	5.18	4.23	4.84
cats	2.59	2.61	3.32	3.73	2.51	2.82
water	1.79	1.81	2.36	2.63	1.74	1.89
finger	5.63	5.66	6.11	5.95	5.47	5.81
us	2.67	2.63	3.28	3.77	2.34	2.84
chart	1.33	1.32	2.14	2.41	1.28	1.40
chart_s	2.74	2.77	3.44	4.06	2.66	3.21
compound1	1.30	1.27	2.39	2.75	1.24	1.37
compound2	1.35	1.33	2.40	2.71	1.24	1.46
aerial2	4.01	4.11	4.49	5.13	3.83	4.44
faxballs	0.97	0.90	1.74	1.73	0.75	0.96
gold	3.92	3.91	4.10	4.33	3.83	4.15
hotel	3.78	3.80	4.06	4.39	3.71	4.22
Average	3.18	3.19	3.76	4.08	3.06	3.46

Tabelle 2.3: Kompressionsergebnisse in Bits pro Pixel

3 PPM

PPM (Prediction by Parital Matching) ist ein adaptiv-statistischer Entropiekodierer, welcher 1984 von J. Cleary und I. Witten entwickelt [CTW95] und 1990 von A. Moffat verbessert und implementiert [Mof90] wurde. Der Algorithmus basiert auf Kontextmodellierung und Prognosen.

3.1 Einleitung

Der PPM-Algorithmus versucht in Abhängigkeit von Symbolen eines vorausgegangenen Datenstroms das nächste Symbol vorherzusagen. Die Ordnung $(order)$ des PPM legt die maximale Anzahl der vorhergehenden Symbole fest (Kontextlängen), die betrachtet werden $PPM(n)$. Wenn das aktuell zu kodierende Symbol im Kontext der Länge n bisher nicht aufgetreten ist, kann keine Prognose gemacht werden. Die Suche wird im nächst kleineren Kontext mit der Länge $n-1$ fortgesetzt. Der Kontext wird solange verkleinert, bis ein Treffer gefunden wird oder keine Symbole im Kontext verbleiben $(order-0)$. Für den Fall eines Treffers wird die Wahrscheinlichkeit des zu kodierenden Symbols aktualisiert und das zu kodierende Symbol mit dem arithmetischen Kodierer kodiert. Der Wechsel von einem Kontext in den nächst kleineren geschieht mit Hilfe eines Sonderzeichens ($ESCAPE$-Symbol), welches einen Kontextwechsel signalisiert. Die Zuordnung eines Wahrscheinlichkeitswertes für das *ESCAPE*-Symbol stellt ein gewisses Problem dar, auch bekannt als das *zero probability problem*. Dabei stellt sich die Frage, welche Wahrscheinlichkeit einem Symbol, das bisher nicht aufgetreten ist zugeordnet werden soll. Folglich gibt es verschiedene PPM Varianten, die unterschiedlich mit diesem Problem umgehen.

PPM gehört zu den stärksten Kompressionsalgorithmen seiner Klasse. Der Kompressionsgrad von PPM ist verglichen mit den von anderen Entropiekodierern wie dem Lempel-Ziv-Welch-Algortithmus größer [CTW95]. Aufgrund der langsameren Geschwindigkeit gegenüber anderen Algorithmen hat sich PPM in der Praxis nicht durchsetzen können.

3.2 Arithmetischer Kodierer

Der PPM-Algorithmus verwendet zur Kodierung der Symbole den arithmetischen Kodierer. Aus diesem Grund wird in diesem Abschnitt der arithmetische Kodierer beschrieben. Bei der

arithmetischen Kodierung wird die zu kodierende Symbolsequenz auf eine reelle Zahl im Intervall $[0, 1)$ abgebildet. Angenommen die reelle Zahl sei 0.97456, dann wird als Kodewort 97456 ausgegeben, wobei der vordere Teil 0. entfällt.

Für die Kodierung einer Symbolsequenz muss als erstes die Wahrscheinlichkeit der Symbole (Modell) bestimmt werden. Das Modell kann durch das Lesen des gesamten Datenstroms aufgestellt werden. Es ist aber auch möglich, bereits vorhandene Modelle aus anderen Quellen zu verwenden. Tabelle 3.1 zeigt ein Modell für die Symbolsequenz SWISS␣MISS.

3.2.1 Die Intervallgrenzen

Sei M das Modell über dem Alphabet $A = ␣, M, I, W, S$ (siehe Tabelle 3.1) mit den Wahrscheinlichkeiten

$$p(␣) = 0.1, p(M) = 0.1, p(I) = 0.2, p(W) = 0.1, p(S) = 0.5.$$

Das Intervall [0,1) wird in diesem Fall in fünf Teilintervalle aufgeteilt.

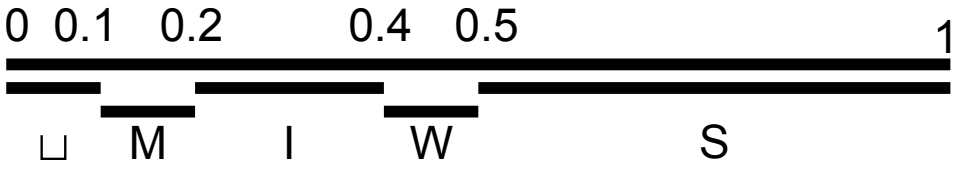

Abbildung 3.1: Aufteilung des Intervalls [0,1)

Die obere bzw. untere Grenze des komplett betrachteten Intervalls werden mit *high* bzw. *low* bezeichnet. Der artithmetische Kodierer benötigt noch die Grenzen der Teilintervalle um während der Kompression die neuen Grenzen (*high* und *low*) zu berechnen. Diese werden aus den kumulierten Wahrscheinlichkeiten gebildet

$$K(a_k) = \sum_{i=1}^{k} p(a_i).$$

Die Werte von $K(a_i)$ sind im Gegensatz zu *high* und *low* konstant und werden bei der Berechnung der Grenzen high und low verwendet.

3.2.2 Kodierung

Das Intervall I ist vor Beginn der Kodierung stets [*low*,*high*), wobei *low* = 0 und *high* = 1 sind. Als nächstes wird I auf ein Teilintervall I' eingeschränkt, dazu muss zunächst das erste Symbol s_1 eingelesen werden. Die Grenzen des neuen Intervalls I' werden wieder als *low*

und *high* dargestellt. Angenommen $s_1 = a_i$ sei das i-te Symbol des Alphabets, dann gilt für die obere bzw. untere Grenze

$$low := \sum_{i=1}^{k} -1p(a_i) = K(a_{k-1})$$

und

$$high := \sum_{i=1}^{k} p(a_i) = K(a_k).$$

Werden alle Buchstaben des Alphabets A in diese Gleichungen eingesetzt, ergeben sich daraus die in Abbildung 3.1 gezeigten Ober- und Untergrenzen der Teilintervalle. Die Größe des Teilintervalls [*low*,*high*) hängt von der Größe der Wahrscheinlichkeit $p(a_i)$ ab. Je größer dieser nämlich ist, desto größer wird das Teilintervall I', desto besser ist die Kompressionsrate. Der Grund dafür liegt in der Länge der Nachkommastellen mit der eine Zahl eindeutig einem Intervall zugewiesen werden kann. Ist das Intervall sehr groß, reicht eine kleine Anzahl von Nachkommastellen, wobei für kleine Intervalle viele Nachkommastellen benötigt werden, was sich negativ auf die Kompressionsrate auswirkt. Da im weiteren Verlauf das Intervall I' nicht mehr verlassen wird und die Werte *low* und *high* diesen eindeutig festlegen, reicht es mit diesen Werten weiter zu rechnen. Nachdem das Teilintervall für das erste Symbol s_1 berechnet wurde, wird nun im nächsten Schritt das Teilintervall für das zweite Symbol $s_2 = a_j$ berechnet. Für die Definition der neuen Grenzen ist die Anpassung der bisherigen Gleichungen zur Berechnung von *high* und *low* notwendig. Diese beziehen sich nämlich auf das Intervall $I := [low, high)$ mit $low = 0$ und $high = 1$, während das Intervall I' ein echtes Teilintervall davon ist. Dazu wird zum einen der Anfang des Intervalls I' und zum anderen ein Ausgleichsfaktor zur Verrechnung mit den Grenzen in die Gleichungen mit aufgenommen.

$$low' := low + \sum_{i=1}^{j-1} p(a_i) \cdot (high - low) = low + K(a_{j-1}) \cdot (high - low) \tag{3.1}$$

$$high' := low + \sum_{i=1}^{j} p(a_i) \cdot (high - low) = low + K(a_j) \cdot (high - low). \tag{3.2}$$

Die neuen Grenzen *low′* und *high′* werden in Abhängigkeit von den vorherigen Grenzen *low* und *high* berechnet. Im nächsten Schritt werden wiederum die Grenzen aktualisiert indem $low := low'$ und $high := high'$ gesetzt wird.

Im folgenden werden die ersten zwei Schritte der Kodierung der bereits vorher erwähnten Zeichenfolge $S = \text{SWISS} \sqcup \text{MISS}$ unter der Annahme des Modells in Tabelle 3.1 gezeigt. Bei dem ersten zu kodierenden Symbol aus S handelt es sich um ein S. Zu Beginn ist das Intervall [0,1), daraus ergeben sich die Grenzen

$$low := 0.0 + (1.0 - 0.0) \cdot 0.5 = 0.5$$
$$high := 0.0 + (1.0 - 0.0) \cdot 1.0 = 1.0.$$

Symbol	Häufigkeit	Wahrscheinlichkeit	Intervall	cumfreq
a_1=␣	1	$1/10 = 0.1$	$[0.0, 0.1)$	0
a_2=M	1	$1/10 = 0.1$	$[0.1, 0.2)$	1
a_3=I	2	$2/10 = 0.2$	$[0.2, 0.4)$	2
a_4=W	1	$1/10 = 0.1$	$[0.4, 0.5)$	4
a_5=S	5	$5/10 = 0.5$	$[0.5, 1.0)$	5
			$Totalcumfreq =$	10

Tabelle 3.1: Häufigkeiten und Wahrscheinlichkeiten von SWISS␣MISS

Dadurch ergibt sich ein Intervall von $[0.5, 1.0)$, mit dem weiter gerechnet wird. Das zweite Symbol, welches im zweiten Schritt kodiert werden soll, ist ein W. Folglich lauten die Grenzen

$$low := 0.5 + (1.0 - 0.5) \cdot 0.4 = 0.70$$
$$high := 0.5 + (1.0 - 0.5) \cdot 0.5 = 0.75.$$

Die volständige Kodierung ist in Tabelle 3.2 zusammengefasst.

Das Intervall verkleinert sich während der Abarbeitung kontinuierlich. Nach Bearbeitung der Sequenz $S =$ SWISS␣MISS hat das Intervall nur noch die Größe $[0.71753375, 0.717535)$. Das Kodewort wird durch Wählen eines geeigneten Wertes (z.B. 0.71754) aus diesem Intervall bestimmt.

Die kontinuierliche Reduzierung des Intervalls führt zu immer längeren Zahlenwerten, die wiederum ein Problem für heutige Rechner darstellen. Dieses Problem wird durch Reskalierung der Intervalle (Kapitel 3.2.5) umgangen.

3.2.3 Dekodierung

Die Dekodierung einer kodierten Sequenz läuft analog zur Kodierung. Die kodierte Sequenz S wird aus dem Code $Z := Code(S)$ welches, in Form einer Zahl gegeben ist und im Intervall $I := [0, 1)$ liegt, konstruiert. Unter der Voraussetzung, dass das Modell gegeben ist, wird nun die Zahl Z analysiert und das dazugehörige Intervall $I' := [K(a_k - 1), K(a_k))$, mit dem dazugehörigen Symbol ausfindig gemacht. In diesem Fall ist a_k das erste Symbol der Originalsequenz. Als nächstes wird untersucht, in welchem Teilintervall von I' die Zahl Z liegt. Dazu müssen, analog zu der Kodierung die Grenzen der Teilintervalle in Abhängigkeit der Grenzen vom Intervall I' angepasst werden. Anschließend wird $I := I'$ gesetzt und nach dem Intervall des nächsten Symbols gesucht. Die Anpassung der Grenzen geschieht mit

$$low' := low + K(a_{i-1}) \cdot (high - low)$$
$$high' := low + K(a_i) \cdot (high - low).$$

Symbol		Berechnung von low und high
S	L	$0.0 + (1.0 - 0.0) \times 0.5 = 0.5$
	H	$0.0 + (1.0 - 0.0) \times 1.0 = 1.0$
W	L	$0.5 + (1.0 - 0.5) \times 0.4 = 0.70$
	H	$0.5 + (1.0 - 0.5) \times 0.5 = 0.75$
I	L	$0.7 + (0.75 - 0.70) \times 0.2 = 0.71$
	H	$0.7 + (0.75 - 0.70) \times 0.4 = 0.72$
S	L	$0.71 + (0.72 - 0.71) \times 0.5 = 0.715$
	H	$0.71 + (0.72 - 0.71) \times 1.0 = 0.72$
S	L	$0.715 + (0.72 - 0.715) \times 0.5 = 0.7175$
	H	$0.715 + (0.72 - 0.715) \times 1.0 = 0.72$
␣	L	$0.7175 + (0.72 - 0.7175) \times 0.0 = 0.7175$
	H	$0.7175 + (0.72 - 0.7175) \times 0.1 = 0.71775$
M	L	$0.7175 + (0.71775 - 0.7175) \times 0.1 = 0.71525$
	H	$0.7175 + (0.71775 - 0.7175) \times 0.2 = 0.717550$
I	L	$0.717525 + (0.71755 - 0.717525) \times 0.2 = 0.717530$
	H	$0.717525 + (0.71755 - 0.717525) \times 0.4 = 0.717535$
S	L	$0.717530 + (0.717535 - 0.717530) \times 0.5 = 0.7175325$
	H	$0.717530 + (0.717535 - 0.717530) \times 1.0 = 0.717535$
S	L	$0.7175325 + (0.717535 - 0.7175325) \times 0.5 = 0.71753375$
	H	$0.7175325 + (0.717535 - 0.7175325) \times 1.0 = 0.717535$

Tabelle 3.2: Ablauf der arithmetischen Kodierung von SWISS␣MISS

Nachdem die neuen Grenzen der Teilintervalle berechnet wurden wird das nächste Symbol a_k gesucht, indem k so gewählt wird, dass Z zwischen den dazugehörigen Grenzen *low* und *high* liegt. Um die ganze Originalsequenz zu bestimmen wird diese Prozedur $|S|$ mal wiederholt.

Im folgenden soll die Dekodierung am Beispiel der Sequenz $SWISS \sqcup MISS$ gezeigt werden. Angenommen der zu dekodierende Kode sei $Z = 0.71754$, welches der Sequenz eindeutig zugeordnet werden kann. Zu Beginn der Dekodierung gilt $low = 0$ und $high = 1$. Im folgenden wird überprüft für welches a_k die Bedingung $low \leq Z \leq high$ gilt

$$low = 0 + 0.5 \cdot (1 - 0) = 0.5$$
$$high = 0 + 1.0 \cdot (1 - 0) = 1.0$$

Damit ist das erste Symbol ein S, da $0.5 \leq 0.71754 \leq 1.0$ gilt. Diese Vorgehensweise wird nun für das zweite Symbol wiederholt.

$$low = 0.5 + 0.4 \cdot (1 - 0.5) = 0.7$$
$$high = 0.5 + 0.5 \cdot (1 - 0.5) = 0.75$$

In diesem Fall entspricht das Symbol einem W, denn es gilt $0.7 \leq 0.71754 \leq 0.75$.

Die Dekodierung kann durch die Umstellung der Gleichung 3.1 vereinfacht werden. Dazu wird low vom Kode abgezogen und anschließend durch range geteilt, wobei $range = (high - low)$ ist.

$$Code = Z - low/range.$$

Im nächsten Schritt wird überprüft in welchem Intervall der Kodewert liegt und somit das gesuchte Symbol a_k bestimmt. Für das Beispiel von vorhin, würde das wie folgt aussehen.Vor Beginn der Dekodierung gilt wieder $low = 0$ und $high = 1$. Das erste Symbol ist ein S, da das Kodewort = 0.71753375 zu Beginn im Intervall $[0.5, 1)$ liegt. Im nächsten Schritt werden die low und high Werte von Symbol S verwendet. low wird zunächst vom Kodewort 0.71753375 abgezogen und anschließend durch das Intervall *range* geteilt $(0.71753375 - 0.5)/0.5 = 0.4350675$. Der resultierende Kode liegt im Intervall [0.40.5). Folglich ist das gesuchte Symbol in diesem Fall 'W'. Die restlichen Schritte sind in Tabelle 3.3 zusammengefasst.

Symbol	Kode-low/(high-low)	range
S	$0.71753375 - 0.5$ = 0.21753375	/0.5 = 0.4350675
W	$0.4350675 - 0.4$ = 0.0350675	/0.1 = 0.350675
I	$0.350675 - 0.2$ = 0.150675	/0.2 = 0.753375
S	$0.753375 - 0.5$ = 0.253375	/0.5 = 0.50675
S	$0.50675 - 0.5$ = 0.00675	/0.5 = 0.0135
␣	$0.0135 - 0$ = 0.0135	/0.0 = 0.135
M	$0.135 - 0.1$ = 0.035	/0.1 = 0.35
I	$0.35 - 0.2$ = 0.15	/0.2 = 0.75
S	$0.75 - 0.5$ = 0.25	/0.5 = 0.5
S	$0.5 - 0.5$ = 0	/0.5 = 0

Tabelle 3.3: Dekodierung der Sequenz SWISS ␣ MISS

Bei der Dekodierung tritt das Problem auf, wann der Rechenvorhang gestoppt werden soll, da der Dekodierer nicht weiß wie die Originalsequenz ausieht. Dieses Problem kann durch Bestimmen der Anzahl von Dekodierungsschritten oder durch Verwenden eines Sonderzeichens (*EOF_Symbol*), welches das Ende der Dekodierung signalisiert, umgangen werden. Letzeres wird auch bei der Implementierung des arithmetischen Kodierers verwendet.

3.2.4 Kodierung als Ganzzahl

Bisher wurden die zu kodierende Symbolsequenz auf reelle Zahlen abgebildet. Das hat zum einen den Nachteil der Ineffizienz, da Operationen mit reellen Zahlen gegenüber ganzen Zahlen zeitintensiver sind und zum anderen können unendlich große reelle Zahlen auf dem Rechner nicht dargestellt werden. Eine vereinfachte Variante des arithmetischen Kodierers, der mit ganzen Zahlen operiert und eine Implementierung dazu haben Witten, Neal, und Cleary im Jahre 1987 vorgestellt [IHW87]. Im folgenden soll dieser gezeigt werden.

Die Wahrscheinlichkeiten werden mit der Verwendung von ganzen Zahlen nicht mehr als Bruchteile von eins angegeben, stattdessen werden die Häufigkeiten der Symbole auf die Gesamtzahl der Symbole normiert. Als untere Grenze wird die Summe der Häufigkeiten der kleineren Symbole angegeben und als obere Grenze die Summe plus die Häufigkeit des zu kodierenden Symbols

$$\begin{aligned} low\_count &= \sum_{i=0}^{\#symbol-1} cum\_freq[i] \\ high\_count &= \sum_{i=0}^{\#symbol-1} cum\_freq[i] + cum\_freq[symbol] \\ &= low + cum\_freq[symbol] \end{aligned}$$

Im folgenden wird die arithmetische Kodierung mittels 4-stelligen Integerwerten vorgestellt. Die höchstmögliche darstellbare 4-stellige Zahl im Dezimalsystem ist 9999 und die kleinste 0000. Um möglichst alle Werte des Intervalls zu nutzen, wird $low = 0000$ und $high = 9999$ gesetzt. Außerdem wird durch die Unterteilung des Bereichs zwischen low und high in total Schritte, die Variable *step* eingeführt mit

$$step = range/total$$

wobei

$$range = high - low + 1.$$

Die Darstellung der oberen Grenze *high* im Dezimalsystem wäre äquivalent zu $9999.\overline{9}$. Durch Entfernen der Nachkommastellen wird die obere Grenze und damit auch das Intervall um 1 kleiner. Um dies zu kompensieren wird wieder eine 1 hinzu addiert. Dies Addition ist auch der Grund weshalb der Bereich am Anfang der Kodierung beschränkt wird (z.B. 16-Bit→ 15-Bit), denn zu Anfang ist die Differenz von der oberen und der unteren Grenze gleich der maximal darstellbaren Zahl. Die Addition der 1 auf diese Zahl würde dazu führen, dass der gültige Bereich überschritten und eine Fortsetzung der Kodierung nicht mehr möglich wird. Nach der Definition aller notwendigen Variablen können nun die Grenzen bestimmt werden.

$$\begin{aligned} low &= low + step \cdot low\_count \\ high &= low + step \cdot high\_count - 1 \end{aligned}$$

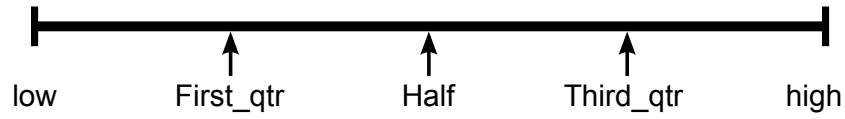

Abbildung 3.2: Intervall-Quadranten

3.2.5 Skalierung bei Über- bzw. Unterlauf

Eine weitere Problematik des arithmetischen Kodierens ist die Annäherung der beiden Grenzen low und high zueinander. Bereits nach wenigen Schritten sind diese nämlich so nahe beieinander, dass eine weitere Kodierung nicht mehr möglich ist. Dies wird auch Überlauf (*overflow*) genannt. Das liegt an der Eigenschaft des arithmetischen Kodierens. Wird ein Intervall einmal betreten, wird dieser nicht mehr verlassen, d.h. irgendwann passiert es, dass die höherwertigen Stellen von *low* und *high* gleich sind und folglich sich nicht mehr ändern. Diese Stellen sind für die weitere Kodierung unwichtig. Deshalb kann die Information gespeichert und die Grenzen verändert werden. Dazu wird zuerst überprüft ob die Grenzen in derselben Hälfte des Intervalls liegen, falls dies der Fall ist und beide Grenzen in der ersten Hälfte liegen ($high < Half$) wird mit der Funktion bit_plus_follow eine 0 ausgegeben. Liegen diese dagegen in der zweiten Hälfte ($low \geq Half$) wird mit derselben Funktion eine 1 ausgegeben. Anschließend werden *low* und *high* wieder auf den gesamten Bereich skaliert. Die Skalierung bei Überlauf reicht allein nicht aus um das Problem der Konvergenz zu beheben. Es kann nämlich immer noch eine Annäherung von *low* und *high* zueinander auftreten, indem, sich *low* von der unteren Hälfte und *high* von der oberen der Mitte ($Half$) des Intervalls annähern, auch Unterlauf (*underflow*) genannt. Die Folge wäre ein immer kleiner werdendes Intervall, mit welchem der Kodierer nicht mehr vernünftig arbeiten könnte. Um das zu verhindern werden zunächst die Konstanten *First_quarter* und *Third_quarter*, welche die Quadrant-Grenzen kennzeichnen (Abbildung 3.2).

Das Ziel hierbei ist das Schrumpfen des Intervalls auf weniger als die Hälfte zu erkennen und zu verhindern. Sobald *low* größer als das untere Viertel (*First_qtr*) und *high* kleiner als das obere Viertel (*Third_qtr*) ist wird das Intervall skaliert, so dass sich *low* und *high* immer im unteren bzw. oberen Viertel bewegen. Die Häufigkeit der Unterlauf-Skalierung, die zwischen den Überlauf-Skalierungen stattfindet wird in der Variable $bits\_to\_follow$ festgehalten und bei der nächsten Ausgabe ($bits\_plus\_follow$) mit ausgegeben.

Listing 3.1: Über- und Unterlauf Skalierung

```
while(true)
{
  if(high<Half)              \\check for overflow
  {
    bit_plus_follow(0);          \\output 0-Bit
  }
```

```
  elseif (low >= Half)           \\check for overflow
  {
    bit_plus_follow(1);          \\output 1-Bit
    low = low - Half;
    high = high - Half;          \\move offset to top
  }
  elseif (low >= First_qtr) && (high<Third_qtr) \\check for underflow
  {
    bits_to_follow = bits_to_follow + 1;
    low = low - First_qtr;
    high = high - First_qtr;       \\move ofset to middle
  }
  else
  {
    break;
  }
  low = 2*low;               \\scale up range
  high = 2*high+1;
}
```

3.2.6 Dekodierung als Ganzzahl

Die Dekodierung ist vergleichbar mit dem Verfahren in Kapitel 3.2.3, mit dem Unterschied, dass hier die Skalierung bei Über- und Unterlauf beachtet werden muss. Die Funktionsweise des Dekodierers kann in zwei Teile gegliedert werden. Im ersten Teil wird das vom Kodierer gespeicherte Symbol bestimmt, indem der Kodewert (*cum*) des Symbols errechnet wird mit

$$range = high - low + 1$$

$$step = (high - low + 1)/total$$

$$cum = (value - low)/step.$$

Die Variable *value* enthält die zu dekodierende Sequenz. Mit Hilfe der Variable *cum* kann nun bestimmt werden, welches Symbol kodiert wurde. Dies geschieht, indem überprüft wird im welchen Häufigkeits-Intervall die Variable *cum* liegt. Nachdem das dazugehörige Symbol bestimmt wurde, werden im zweiten Teil die Grenzen *low* und *high* aktualisiert mit

$$low = low + step \cdot low\_count$$
$$high = low + step \cdot high\_count - 1.$$

Bei der Dekodierung ist zusätzlich zu beachten, dass der Kodewert bei Überlauf und Unterlauf mit skaliert wird.

3.2.7 Adaptive Arithmetische Kodierung

Die arithmetische Kodierung kann durch das Anpassen der Wahrscheinlichkeiten während des Kodierungsprozesses optimiert werden. Bisher waren die Wahrscheinlichkeiten statisch und mussten vor der Kodierung ermittelt werden. Die statistische Verteilung der Symbole wird durch Abtasten der gesamten Nachricht ermittelt und anschließend in einer Kodetabelle abgelegt. Der Dekodierer ermittelt die Symbole durch Ablesen der berechneten und dekodierten Kodewerte aus der Kodetabelle. Beim adaptiven Verfahren wird die statistische Verteilung der Symbole während der Kodierung nach und nach ermittelt. Nach jedem Schritt werden die Wahrscheinlichkeiten, mit denen die Symbole auftreten, angepasst (*update_model*). Die Nachricht muss nicht mehr zu Beginn vollständig gelesen werden um die Kodetabelle aufzubauen. In Folge dessen wird die Kodetabelle nur soweit aufgebaut wie dafür erforderlich. Die Adaption funktioniert nur, wenn der Algorithmus sowohl auf der Kodiererseite als auch der Dekodierer identisch ist. Die adaptive Methode ist zu Beginn sehr ungenau, da noch wenige Symbole gelesen wurden, wird jedoch mit wachsender Länge der Nachricht genauer.

3.3 Markov-Ketten

Das Hauptmerkmal des PPM-Algorithmus besteht darin, das nächste Symbol mit Hilfe der vorausgegangenen Symbole (Kontext) zu bestimmen. Diese Art der Vorhersage eines Symbols wird auch Markov-Kette genannt. Das Ziel von Markov-Ketten ist Wahrscheinlichkeiten für das Eintreten zukünftiger Ereignisse in Abhängigkeit der Vorgeschichte anzugeben. Markov-Ketten haben die Eigenschaft, Prognosen basierend auf einem Teil der Vorgeschichte genauso gut zu bewerkstelligen wie unter Kenntnis der gesamten Vorgeschichte des Prozesses.

Ein Zufallsprozess (stochastischer Prozess) ist eine Folge von elementaren Zufallsereignissen.

$$X_1, X_2, ..., X_t \in \Omega, t \in \mathbb{N}$$

wobei die Zufallswerte Zustände des Prozesses sind. Zum Zeitpunkt t befindet sich ein Prozess im Zustand X_1.

Markov-Ketten sind spezielle Zufallsprozesse mit der Eigenschaft

$$\begin{aligned} &P(X_{t+1} = x_{t+1} | X_t = x_t, X_{t-1} = x_{t-1}, ..., X_0 = x_0) \\ =&P(X_{t+1} = x_{t+1} | X_t = x_t, ..., X_{t-n+1} = x_{t-n+1}). \end{aligned}$$

Dabei gibt die Variable n die Ordnung der Markov-Kette an, d.h. die Wahrscheinlichkeit des aktuellen Zustands hängt von den vorhergehenden n Zuständen ab.

3.4 PPM-Algorithmus

Der PPM-Algorithmus kann in zwei Teile gegliedert werden, das Modellieren und Kodieren. Die Modellierung geschieht mit Hilfe der Markov-Ketten [CW84], mit den es möglich ist Wahrscheinlichkeiten in Abhängigkeit bereits gelesener Symbole anzugeben. Im zweiten Teil wird dann das aktuell zu kodierende Symbol mit dem entsprechenden Wahrscheinlichkeitswert arithmetisch kodiert. Genauer gesagt wird der adaptive arithmetische Kodierer verwendet, da die Häufigkeiten im Laufe des Kodierungsprozesses ständig aktualisiert werden.

Kodierer

Der PPM-Algorithmus bestimmt, wie bereits erwähnt, die Wahrscheinlichkeiten von zu kodierenden Symbolen mit Bezug auf bereits gelesene Symbole und verwaltet diese in einer Datenstruktur (Array, Trie-Baum,...). Die Idee dahinter soll etwas genauer verdeutlicht

werden. In der deutschen Sprache z.B. gibt es Zeichensequenzen, die leicht vorhersehbar sind. Die Wahrscheinlichkeit das auf ein 'q' in der Regel ein 'u' folgt und auf die Zeichensequenz 'ic' ein 'h' ist sehr hoch. Diese Eigenschaft nutzt der PPM-Algorithmus um eine hohe Kompression zu erzielen. Der Kodierer liest das nächste Symbol s_i aus der Symbolsequenz und schaut im Kontext C der Ordnung N nach, ob dieser Kontext in der Vergangenheit bereits auftrat. Abhängig davon wird die Wahrscheinlichkeit $P(s|C_N)$, mit der das Symbol im Kontext C vorkommt bestimmt. Anschließend wird s mit der Wahrscheinlichkeit P arithmetisch kodiert. Findet der Kodierer s_i im Kontext C_N aktualisiert er diesen und führt die Kodierung fort. Wird s dagegen nicht gefunden bedeutet dies, dass s bisher im Kontext C_N nicht vorgekommen ist. Infolgedessen wechselt der Kodierer in die nächst kleinere Ordnung mit der Länge $N-1$. Kommt s auch in diesem Kontext C_{N-1} nicht vor, wird die Ordnung so lange verkleinert bis das Symbol vorkommt. Wird das Symbol auch nicht in der 0-ten Ordnung gefunden, wechselt der Kodierer in die Ordnung -1. Hier sind die Wahrscheinlichkeiten aller vorkommenden Symbole im Alphabet $A = s_1, s_2, ..., s_n$ gleich verteilt

$$P(s_i) = \frac{1}{|A|}. \tag{3.3}$$

Beispiel Angenommen der Kodierer befindet sich im Kontext *sch* der Ordnung 3, welches bereits 24 mal vorkam gefolgt von einem w (10-mal), einem o (7-mal), einem u (5-mal) und einem l (2-mal). Daraus ergeben sich für die Symbole die Wahrscheinlichkeiten 10/24, 7/24, 5/24 und 2/24. Wenn das nächste Symbol ein w ist, wird es mit dem Wahrscheinlichkeitswert $P(w|sch) = 7/24$ arithmetisch kodiert. Im Anschluss werden die Wahrscheinlichkeiten aktualisiert auf 11/25, 7/25, 5/25 und 2/25.

Angenommen das nächste Symbol sei ein a. Die Wahrscheinlichkeit für dieses Symbol im Kontext *sch* ist 0, da der Kontext gefolgt vom Symbol a zuvor nie aufgetreten ist (zero probability problem). Dieses Problem wird durch Wechseln in einen kürzeren Kontext gelöst. Unter der Annahme das a sowohl im Kontext der Ordnung 2 (*ch*) und 1 (*h*) nicht vorkommt, schaltet der Kodierer runter bis zur Ordnung 0. Angenommen die Anzahl der bisher gelesenen Symbole sei 460 und von diesen seien 70 das Symbol a, folglich wird a der Wahrscheinlichkeitswert 70/460 zugeordnet. Kommt das Symbol stattdessen in der Ordnung 0 nicht vor, wird in die Ordnung -1 gewechselt, wo jedes Symbol den festen Wahrscheinlichkeitswert 1/(Anzahl der Symbole) erhält.

Dekodierer

In diesem Abschnitt wird der Dekodierer des PPM-Algorithmus betrachtet. Der Kodierer sieht gegenüber dem Dekodierer das nächste Symbol und kann den nächsten Schritt basierend auf diesem Symbol ausführen. Ein Kontextwechsel findet statt nachdem das zu kodierende Symbol eingelesen wurde. Der Dekodierer dagegen muss das nächste Symbol

ermitteln und kann den Kontextwechsel des Kodierers nicht wiederspiegeln, da dieser das nächste Symbol nicht kennt. Um dieses Problem zu lösen wird ein Symbol aus der Symbolmenge als *escape*-Symbol (*Esc*) reserviert. Durch diese Zusatzinformation ist der Dekodierer in der Lage einen Kontextwechsel zu erkennen. Findet ein Kontextwechsel statt gibt der Kodierer zuerst das *escape*-Symbol im gegenwärtigen Kontext aus. Es ist wichtig, dass die Aktualisierung nicht vor dem *escape*-Symbol geschieht, da zu diesem Zeitpunkt das Modell vom Kodierer synchron zu dem vom Dekodierer sein muss.

Eine Frage die sich bei der Kodierung des *escape*-Symbols stellt ist, mit welcher Wahrscheinlichkeit das *escape*-Symbol kodiert werden soll. Dazu wird der folgende Fall betrachtet. Angenommen der Kodierer befindet sich in der Ordnung N und das Symbol s_i, welches zum ersten mal auftritt und auf Grund dessen in keiner Ordnung vorkommt, wird eingelesen. Das hat zur Folge, dass der Kodierer $N+1$ *escape*-Symbole ausgeben muss, um in die Ordnung -1 zu wechseln in der s_i mit der Wahrscheinlichkeit $P(s_i) = 1/|A|$ kodiert werden kann. Diese Charakteristik tritt vor allem am Anfang der Kodierung auf, wo die Symbole zum ersten mal auftreten. Deshalb ist es sinnvoll, dem *escape*-Symbol einen großen Wahrscheinlichkeitswert am Anfang zu geben und diesen in Abhängigkeit der gesammelten Infomationen zu verkleinern.

PPM-Varianten

Wie in der Einleitung bereits erwähnt gibt es verschiedene Varianten des PPM-Algorithmus, die das *zero probability problem* unterschiedlich handhaben. Die wichtigsten werden im folgenden vorgestellt:

PPM-A Diese Methode weist dem *escape*-Symbol unabhängig von der Anzahl der Kontextwechsel die Häufigkeit 1 zu. Daraus folgt eine Wahrscheinlichkeit $P(Esc\_symbol) = \frac{1}{n+1}$, wobei n die Summe aller bisher aufgetretenen Symbole ist. Für die restlichen Symbole s im dazugehörigen Kontext gilt $P(s) = \frac{\text{Häufigkeit von s}}{n+1}$.

PPM-B Die Idee hinter dieser Methode ist die, dass ein Symbol s, welches bisher einmal in Kontext C auftrat, vom Dekodierer eher unwahrscheinlich wieder angesprungen wird. Folglich werden die Häufigkeitswerte aller Symbole im Kontext C um eins dekrementiert und dem zum selben Kontext angehörenden *escape*-Symbol übergeben. Damit verschwinden alle Einträge mit der Häufigkeit 1 aus der Modell-Tabelle.

PPM-C Diese Methode ist der Methode B sehr ähnlich, mit dem Unterschied, dass anstatt die Häufigkeitswerte von den Symbolen im Kontext C zu dekrementieren, dem *escape*-Symbol die Häufigkeit der Symbole im jeweiligen Kontext zugewiesen wird.

Tabelle 3.4 zeigt das Kontextmodell der Zeichenfolge textitabracadabra bis zur Ordnung 3. Bei der PPM-Variante handelt es sich um PPM-C. Die Kontexte werden in Gruppen

zusammengefasst. Die Ordnung -1 wurde in die Tabelle nicht mit aufgenommen, da die Wahrscheinlichkeit für alle Symbole im Alphabet gleich ist (Gleichung 3.3).

Order-3					**Order-2**					**Order-1**					**Order-0**			
Context			c	P	Context			c	P	Context			c	P	Context		c	P
abr	$\rightarrow$	a	2	$\frac{2}{3}$	ab	$\rightarrow$	r	2	$\frac{2}{3}$	a	$\rightarrow$	b	2	$\frac{2}{7}$	$\rightarrow$	a	5	$\frac{5}{16}$
	$\rightarrow$	*Esc*	1	$\frac{1}{3}$		$\rightarrow$	*Esc*	1	$\frac{1}{3}$		$\rightarrow$	c	1	$\frac{1}{7}$	$\rightarrow$	b	2	$\frac{2}{16}$
											$\rightarrow$	d	1	$\frac{1}{7}$	$\rightarrow$	c	1	$\frac{1}{16}$
bra	$\rightarrow$	c	1	$\frac{1}{2}$	ac	$\rightarrow$	a	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	3	$\frac{3}{7}$	$\rightarrow$	d	1	$\frac{1}{16}$
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$						$\rightarrow$	r	2	$\frac{2}{16}$
										b	$\rightarrow$	r	2	$\frac{2}{3}$	$\rightarrow$	*Esc*	5	$\frac{5}{16}$
rac	$\rightarrow$	a	1	$\frac{1}{2}$	ad	$\rightarrow$	a	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{3}$				
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$									
										c	$\rightarrow$	a	1	$\frac{1}{2}$				
aca	$\rightarrow$	d	1	$\frac{1}{2}$	br	$\rightarrow$	a	2	$\frac{2}{3}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$				
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{3}$									
										d	$\rightarrow$	a	1	$\frac{1}{2}$				
cad	$\rightarrow$	a	1	$\frac{1}{2}$	ca	$\rightarrow$	d	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$				
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$									
										r	$\rightarrow$	a	2	$\frac{2}{3}$				
ada	$\rightarrow$	b	1	$\frac{1}{2}$	da	$\rightarrow$	b	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{3}$				
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$									
dab	$\rightarrow$	r	1	$\frac{1}{2}$	ra	$\rightarrow$	c	1	$\frac{1}{2}$									
	$\rightarrow$	*Esc*	1	$\frac{1}{2}$		$\rightarrow$	*Esc*	1	$\frac{1}{2}$									

Tabelle 3.4: Kontexte, Häufigkeit c und Wahrscheinlichkeit P der Zeichenfolge *abracadabra*

Kontextlänge n

Dieser Abschnitt widmet sich der Kontextlänge n im PPM-Algorithmus und deren Einfluss auf die Kompression. Die Kontextlänge bestimmt die Anzahl der Vorgängersymbole, welche für die Kontextbildung betrachtet werden. Für $n = 0$ kodiert der PPM-Algorithmus ähnlich dem adaptiven arithmetischen Kodierer, wo jedes Symbol einzeln betrachtet wird. Dagegen führt ein zu hoher Wert für n zu einer ungleichen Verteilung der Wahrscheinlichkeiten in höheren Kontexten. Außerdem hat zur Folge, dass ein Symbol in höheren Kontexten immer seltener gefunden wird und aus diesem Grund ein *Esc*-Symbol ausgegeben wird, um ein Kontextwechsel zu signalisieren. Die Anzahl der Bits, die benötigt werden um das Symbol zu Kodieren, erhöht sich mit der Anzahl der ausgegebenen *Esc*-Symbole.

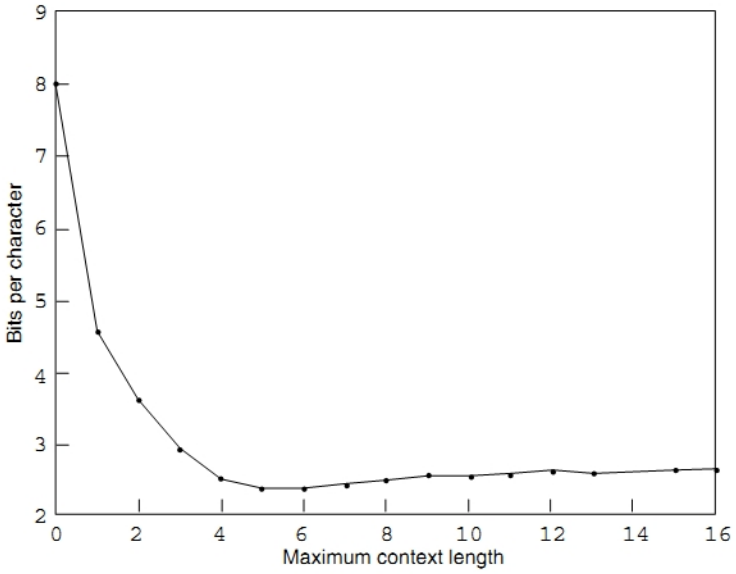

Abbildung 3.3: Kompressionsrate als Funktion der Kontextlänge (aus [CTW95])

Die Praxis hat gezeigt, dass die Kompressionsrate bei Textinhalten am höchsten ist, wenn für n der Wert 5 oder 6 gewählt wird. Abbildung 3.3 zeigt die Anzahl von Bits pro Zeichen (bpc) in Abhängigkeit von der verwendeten Kontextlänge, die für die Komprimierung der Textdatei *book*1[1] benötigt werden.

[1]formatlose ASCII Text-datei mit dem Inhalt des Romanes 'Far from the Madding Crowd' von Thomas Hardy.

4 Implementierung

Der Versuch den JPEG-LS Algorithmus durch Parallelisierung zu beschleunigen scheitert an der Kontextmodellierung. Dieser führt bei der Komprimierung von Bilddaten zu Abhängigkeiten zwischen den Bildpunkten, das wiederum verhindert eine parallele und unabhängige Bearbeitung der Bildsignale. Aus diesem Anlass wird in diesem Kapitel die Kontextmodellierung im JPEG-LS Algorithmus entfernt und die daraus auftretenden Problematiken analysiert. Anschließend wird nach einer Lösung in Form von Alternativen gesucht.

4.1 Studie des Golomb-Rice-Code Parameters *k*

Das Entfernen der Kontextmodellierung wirkt sich auf die Berechnung des Prädiktionsfehlers Px und des Parameters k. Der Schätzwert kann ohne den Korrekturwert C, welcher im Kontextteil bestimmt wird, das nicht mehr korrigiert werden. Außerdem ist eine optimale Berechnung des Parameters k, welcher die Kodewortlängen des Golomb-Rice-Coder bestimmt und für jeden Bildpunkt individuell berechnet wird, nicht mehr möglich.

Im folgenden wird versucht einen optimalen statischen Wert für k zu finden, indem diverse Bilddaten mit verschiedenen k-Werten komprimiert werden. Anschließend werden die Kompressionsraten mit denen des originalen JPEG-LS Algorithmus verglichen.

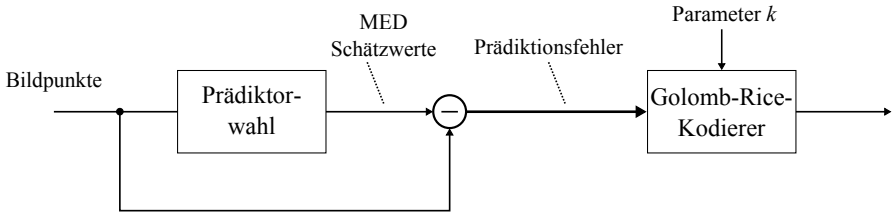

Abbildung 4.1: Modell zur Analyse des Parameters k

Abbildung 4.1 zeigt das Modell des modifizierten Algorithmus, in dem weder eine Kontextmodellierung noch eine Gradientendetektion stattfinden. Stattdessen wird für den Parameter k ein statischer Wert verwendet. Außerdem wird der Schätzwert anhand des Kontextes nicht

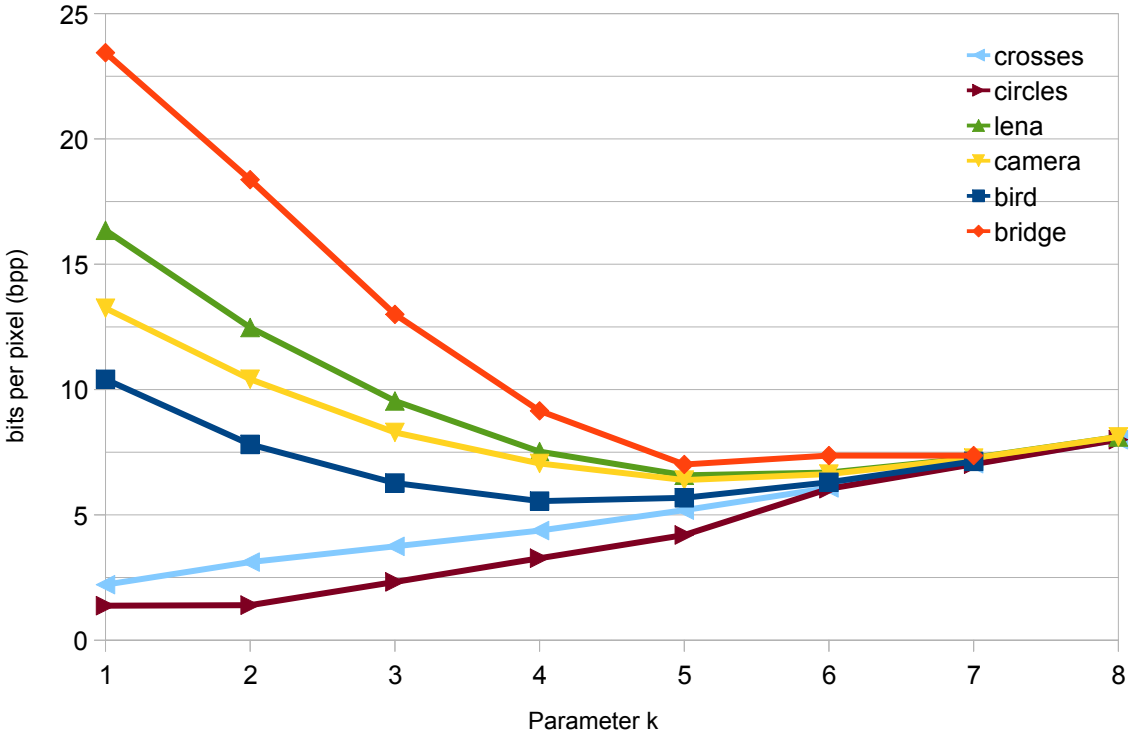

Abbildung 4.2: Kompressionsergebnisse des Golomb-Rice-Coders für verschiedene k

mehr korrigiert, da die Kontextinformationen vom Kontextmodeller fehlen. Der Schätzwert wird folglich nach der Gleichung 2.1 bestimmt und dem Golomb-Rice-Coder übergeben.

Für die Analyse wurden sowohl natürliche (lena, camera, bird, bridge) als auch homogene (circles, crosses) Bilder verwendet, mit dem Ziel einen allgemein verwendbaren k-Wert für den Golom-Rice-Coder, der für alle Bilder gilt, zu finden. Bei den Bildern handelt es sich um Graustufenbilder, die mit 8-Bit pro Pixel (*bpp*) dargestellt werden. Abbildung 3.3 zeigt die Anzahl von Bits, die für die Kodierung der natürlichen Bilder benötigt werden, als Funktion der Kodewortlänge. Für die Kodierung von natürlichen Bildern, die gewöhnlich eine höhere Korrelation aufweisen als homogene, werden bei kleinem k im Vergleich zum originalen JPEG-LS mehr Bits pro Pixel (bpp) benötigt. Bei Bildern die weniger korreliert sind (z.B. bird) findet eine Kompression bereits bei kleinen k statt, wogegen bei Bildern mit starker Korrelation eine Kompression erst ab Kodewortlängen von vier oder größer möglich ist. Zu beachten ist aber das die Kompression mit der Kodewortlänge wieder abnimmt. Bei Bildern die kaum korreliert sind (circles) ist die Kompression bei kleinem k am höchsten. Mit wachsendem k wird hier im Gegensatz zu korrelierten Bildern die Kompression kontinuierlich schlechter.

Bilddatei	orig.	Parameter k (ohne adaptive Korrektur)							
	JPEG-LS	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$
bird	5.14	10.41	7.81	6.27	5.55	5.68	6.30	7.13	8.07
bridge	7.64	23.44	18.37	13.00	9.15	7.37	7.01	7.37	8.11
camera	6.13	13.24	10.41	8.29	7.05	6.39	6.63	7.27	8.12
circles	1.38	1.39	2.32	3.27	4.19	5.09	6.04	7.02	8.00
crosses	1.48	2.22	3.12	3.75	4.37	5.19	6.08	7.04	8.00
lena	6.69	16.35	12.47	9.54	7.53	6.59	6.69	7.28	8.11
Mittelwert	4.74	11.18	9.08	7.35	6.31	6.05	6.46	7.19	8.07

Tabelle 4.1: Kompressionsergebnisse der Analyse von k in Bits pro Pixel (bpp)

Die Kompression ohne adaptive Bestimmung des Parameters k ist im großen und ganzen schlechter als das Original (Tabelle 4.1). Es gibt nur wenige Fälle in denen die modifizierte Variante besser ist. Bilder die stark korreliert sind werden für $k = 5$ unter Umständen besser komprimiert als mit dem originalen JPEG-LS.

4.2 Analyse des Schätzwertes

Die Schätzwertkorrektur ermöglicht eine präzisere Vorhersage über das aktuell zu kodierende Bildsignal und ermöglicht damit eine verbesserte Kompression. In diesem Abschnitt wird untersucht, welchen Effekt die nicht-Korrektur des Schätzwertes auf die Kompressionsrate hat. Dazu wird lediglich der Teil, indem k berechnet wird entfernt und mit konstanten Werten ersetzt.

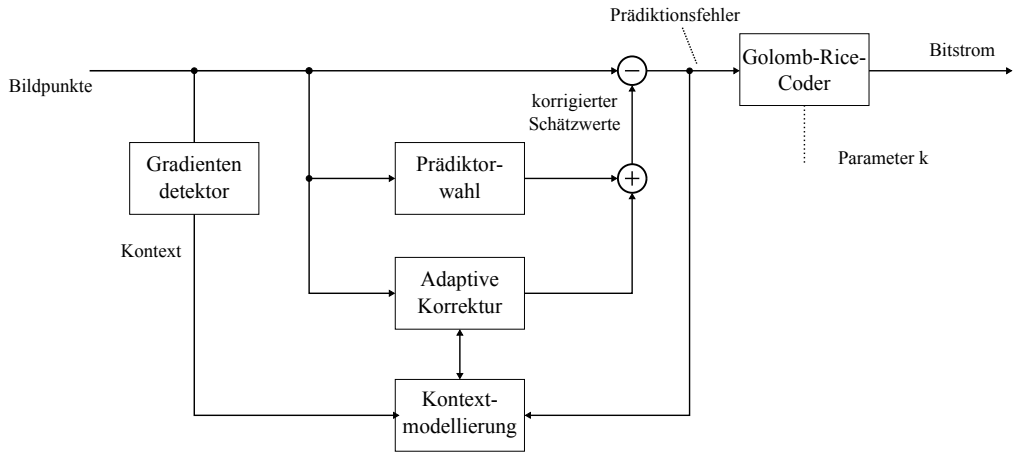

Abbildung 4.3: Modell zur Analyse des Schätzwertes

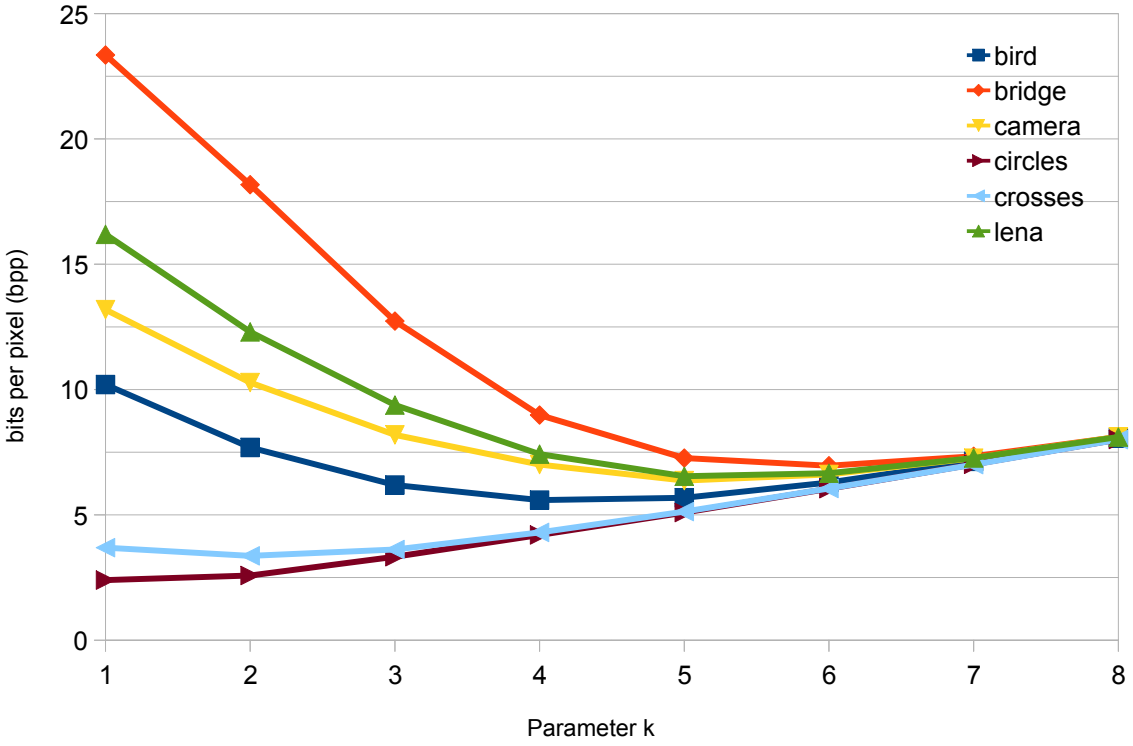

Abbildung 4.4: Kompressionsergebnisse des Golomb-Rice-Coders für verschiedene k

Das in Abbildung 4.3 abgebildete modifizierte Modell des JPEG-LS Algorithmus ist bis auf die Bestimmung des Parameters k identisch mit dem des Originals. Der Schätzwert wird wie schon im Original durch eine adaptive Korrektur auf den Kontext angepasst.

Der Schätzwert wurde bei der Analyse des k in Kapitel 4.1 nicht korrigiert. Im folgenden wird die Kompression mit korrigiertem Schätzwert ermittelt und mit den aus dem vorherigen Abschnitt verglichen. Die Ergebnisse in Abbildung 4.4 zeigen einen kleinen Unterschied gegenüber dem modifizierten Algorithmus ohne Schätzwertkorrektur. Die Schätzwertkorrektur verbessert die Kompression um einen kleinen Wert. Der Grund für den geringen Unterschied kann am statisch festgelegten Parameter k liegen. Um dies zu überprüfen wird der originale JPEG-LS Algorithmus ohne Schätzwertkorrektur bei optimal bestimmten Parameter k ausgeführt. Die Kontextmodellierung wird aufgrund der Berechnung des Parameters k nicht komplett entfernt. Dieser wird anhand der Kontextparameter N und A berechnet. Die Kompression wird durch die Korrektur des Schätzwertes im allgemeinen verbessert. Die Ergebnisse sind in Tabelle 4.2 aufgelistet.

Bilddatei	orig. JPEG-LS	adaptives k ohne adap. Korrektur	Parameter k (mit adaptiver Korrektur) $k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$
bird	5.14	5.27	10.20	7.68	6.19	5.59	5.68	6.29	7.13	8.06
bridge	7.64	7.69	23.35	18.17	12.73	8.98	7.27	6.96	7.34	8.10
camera	6.13	6.18	13.19	10.28	8.19	7.00	6.37	6.61	7.26	8.11
circles	1.38	1.28	2.39	2.58	3.33	4.19	5.09	6.04	7.02	8.00
crosses	1.48	1.47	3.69	3.36	3.62	4.30	5.12	6.05	7.00	8.00
lena	6.69	6.73	16.19	12.30	9.38	7.42	6.54	6.66	7.27	8.11
Mittelwert	4.74	4.77	11.50	9.06	7.24	6.25	6.01	6.44	7.17	8.06

Tabelle 4.2: Vergleich der Kompressionsergebnisse mit und ohne adaptive Schätzwertkorrektur

4.3 Adaptiver Arithmetischer Kodierer

Die Analyse des Parameters k und der Schätzwertkorrektur haben keine allzu große Verbesserung in der Kompression von Bilddaten gezeigt. In diesem Abschnitt wird daher der Golomb-Rice-Coder, der ohne optimales k nicht so gut komprimiert wie mit, ersetzt durch den adaptiven arithmetischen Kodierer.

Zunächst wird der adaptive arithmetische Kodierer, aufbauend auf dem Wissen aus Kapitel 3.2 implementiert. Der adaptive arithmetischer Kodierer unterscheidet sich zum gewöhnlichen arithmetischen Kodierer in der Verwendung der Wahrscheinlichkeiten. Während beim arithmetischen Kodierer die Wahrscheinlichkeitswerte der Symbole statisch sind, werden diese beim adaptiven arithmetischen Kodierer iterativ angepasst. Dazu werden die Häufigkeiten der Symbole nach jedem Auftreten um eins inkrementiert.

Der adaptive arithmetische Kodierer wird in drei Matlab-Dateien gegliedert: Kodierer, Dekodierer und dem Modell.

4.3.1 Modellierung

Die Methode start_model() initialisiert das Modell, indem es die Symbolwahrscheinlichkeiten bestimmt. Dazu werden die Array-Felder cum_freq und freq definiert und mit entsprechenden Startwerten belegt.

Die Häufigkeiten der zu kodierenden Symbole werden mit *cum_freq* und *freq* dargestellt. Das Array freq enthält die Häufigkeit eines Symbols, welches während der Kodierung bei Auftreten des Symbols aktualisiert wird. Das Array *cum_freq* enthält die kumulierten Häufigkeiten eines Symbols und wird zur Normalisierung des Modells verwendet. Mit der

Methode *start_model* werden *freq cum_freq* initialisiert. Die Häufigkeiten *freq* werden alle bis auf das erste Array-Element zu Beginn mit 1 initialisiert. Dagegen wird das erste Element des *freq* Arrays mit 0 initialisiert.

Nach jeder Kodierung oder Dekodierung eines Symbols wird das Modell mit *update_model(symbol)* aktualisiert. Die Methode *update_model* überprüft zunächst ob der kumulative Häufigkeitswert, der in *cum_freq1* gespeichert ist, den Maximalwert *Max_frequency* überschritten hat. Falls dies der Fall ist, werden die Arrays *freq* und *cum_freq* aktualisiert. Dies geschieht in einer for-Schleife in der zunächst die Häufigkeiten *freq* halbiert und die kumulierten Häufigkeiten *cum_freq* entsprechend den Werten in *freq* neu bestimmt werden.

Listing 4.1: Adaptiv arithmetischer Kodierer - Modellierung

```
function model = model()
    No_of_chars = 256;
    No_of_symbols = No_of_chars+1;
    Max_frequency = 16383;
    model.start_model = @start_model;
    model.update_model = @update_model;

    function start_model()       %initialize model
        freq = cell(1, No_of_symbols+1);  %define cell-array
        cum_freq = cell(1, No_of_symbols+1);  %define cell-array

        for i=1:1:No_of_symbols+1
            freq{i} = 1;  \initialize freq
            cum_freq{i} = No_of_symbols - i;   %initialize cum_freq
        end
        freq{1}=0;      %initialize first element of freq with 0
    end

    function update_model(symbol)
       if cum_freq{1} == Max_frequency
           cum = 0;
            for i=No_of_symbols:-1:1
              freq{i} = fix(freq{i}+1/2);
              cum_freq{i} = cum;
              cum = cum + freq{i};
           end
       end

         freq{symbol+1} = freq{symbol+1}+1;

       for i=symbol+1:-1:1
           cum_freq{i} = cum_freq{i}+1;
       end
    end
end
```

4.3.2 Kodierer

Der Kodierer wird zunächst mit der Funktion start_encode() initialisiert. Während der Initialisierungsphase des Kodierers werden die Parameter *Top_value*, *First_qtr*, *Half* und *Third_qtr*, wie in Kapitel 3.2 beschrieben definiert. Außerdem werden die Parameter *low* mit 0 und *high* mit *Top_value* initialisiert. Anschließend können die Schätzwerte, die mit Hilfe des MED-Prädiktor berechnet werden, kodiert werden. Dazu werden die Schätzwerte (*symbol*) mit den zugehörigen Häufigkeitswerten (*cum_freq*) an die Funktion encode_symbol() übergeben. Welcher zunächst den aktuellen Bereich *range* anhand der oberen *high* und der unteren Grenze *high* berechnet. Anschließend wird dieser Bereich unterteilt und das neue Intervall des dazugehörigen Symbols neu berechnet. Dazu werden die neuen Intervallgrenzen *high* und *low* mit der Häufigkeit des entsprechenden Symbols neu bestimmt.

$$low = low + \frac{range \cdot cum\_freq[symbol - 1]}{cum\_freq[1] - 1}$$
$$high = low + \frac{range \cdot cum\_freq[symbol]}{cum\_freq[1]}.$$

Nachdem die neuen Grenzen bestimmt wurden, wird das Intervall nach dem Verfahren in Kapitel 3.2.5 skaliert und die entsprechenden Bits ausgegeben.

Listing 4.2: Adaptiv arithmetischer Kodierer - Kodierung eines Symbols

```
function encode_symbol(symbol, cum_freq)
    range = (high-low)+1;
    high = low + fix((range*cum_freq{symbol})/cum_freq{1})-1;
    low = low + fix((range*cum_freq{symbol+1})/cum_freq{1});

    while(true)
        if high<Half
            bit_plus_follow(0);
        elseif low>=Half
            bit_plus_follow(1);
            low = low - Half;
            high = high - Half;
        elseif low>=First_qtr && high<Third_qtr
            bits_to_follow = bits_to_follow + 1;
            low = low - First_qtr;
            high = high - First_qtr;
        else
            break;
        end
        low = 2*low;
        high = 2*high+1;
    end
end
```

4.3.3 Dekodierer

Der Dekodierer funktioniert analog zum Kodierer. Es wird lediglich eine zusätzlichenVariable *value* benötigt. Das aktuell zu dekodierende Symbol wird aus den Bits der Variablen *value* ermittelt. Dazu wird zunächst der kumulative Wert des zu dekodierenden Symbols *cum* berechnet und anschließend danach gesucht. Die Suche wird mit einer while-Schleife realisiert. Angefangen bei dem Element 1 wird das Array *k*, solange die Bedingung cum_freqsymbol>cum gilt, durchlaufen. Die Variable wird in jeder Iteration um 1 erhöht.

Listing 4.3: Adaptiv arithmetischer Kodierer - Dekodierung eine Symbols

```
function [symbol] = decode_symbol(cum_freq)
   range = (high - low) + 1;
   cum = fix((((value - low) + 1)*cum_freq{1}-1)/range);

   symbol = 1;
   while cum_freq{symbol}>cum
       symbol = symbol + 1;
   end
   symbol = symbol - 1;
    high = low + fix((range*cum_freq{symbol})/cum_freq{1})-1;
    low = low + fix((range*cum_freq{symbol+1})/cum_freq{1});

   while (true)
       if high<Half
       elseif low>=Half
           value = value - Half;
           low = low - Half;
           high = high - Half;
       elseif low>=First_qtr && high<Third_qtr
           value = value - First_qtr;
           low = low - First_qtr;
           high = high - First_qtr;
       else
           break;
       end
       low = 2*low;
       high = 2*high+1;
       value = 2*value+biti.input_bit();
   end

end
```

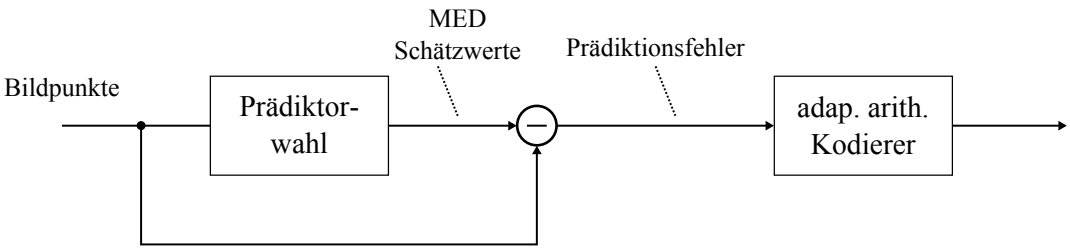

Abbildung 4.5: Modell zur Kompression von Bilddaten mit dem arithmetischen Kodierer ohne adaptive Schätzwertkorrektur

4.3.4 Eingabe und Ausgabe von Bits

Der Kodierer verwendet zum Einlesen von Bits die Methoden *start_inputing_bist* und *input_bit*. Die erste Methode initialisiert die Variablen *bits_to_go* und *garbage_bits* mit dem Wert 0. Die zweite Methode wird zum Einlesen von Bits aus dem Eingabestrom verwendet. Dazu wird zunächst überprüft ob die Variable *bits_to_go* gleich 0 ist. Falls dies der Fall ist wird das nächste Symbol (1*Byte*) in den Puffer *buffer* geschrieben und der Zähler *bits_to_go* auf 8 gesetzt. Ist dagegen *bits_to_go* nicht leer, wird kein neues Symbol eingelesen, sondern das an der Reihe stehende Bit als Return-Wert zurückgegeben. Zum Schluss wird die Variable *bits_to_go* um eins dekrementiert.

Der Dekodierer gibt die dekodierten Bits mithilfe der Methoden *start_outputing_bits*, *output_bit(bit)* und *done_output_bits*. Die Methode *start_outputing_bits* initialisiert die Variablen *std_out*, *buffer bits_to_go*. In der Variablen *stdout* werden die dekodierten Symbole gespeichert. Die Variable *buffer*, welche die Bits enthält und später in stdout geschrieben werden, wird mit 0 initialisiert. Die Variable *bits_to_go* ist der Zähler für den Puffer (*buffer*) und wird mit dem Wert 8 initialisiert. Die Methode *output_bit(bit)* schreibt zunächst das übergebene Bit in den Puffer und dekrementiert den Wert von *bits_to_go*. Falls dieser gleich 0 ist wird der Puffer geleert, indem der Inhalt in *stdout* geschrieben wird. Mit der Methode *done_outputing_bits* werden am Ende des Dekodierens die restlichen Bits im Puffer in *std_out* geschrieben.

4.3.5 JPEG-LS und der adaptive arithmetische Kodierer

Nachdem der adaptive arithmetische Kodierer implementiert worden ist, wird dieser in Kombination mit dem JPEG-LS auf Bilddaten getestet. Dazu wird wird der Golomb-Rice-Coder durch den adaptiven arithmetischen Kodierer ersetzt. Der arithmetische Kodierer hat gegenüber dem Golomb-Rice-Coder keinen einstellbaren Parameter, der die Länge des Kodewortes beeinflusst und vor der Kompression ermittelt werden muss. Daher ist auch eine Kontextmodellierung nicht notwendig und wird aus diesem Grund komplett entfernt.

Bilddatei	orig. JPEG-LS	adap. arithm. Kodierer ohne adap. Korrektur	adap. arithm. Kodierer mit adap. Korrektur
bird	5.14	5.19	5.15
bridge	7.64	7.05	7.00
camera	6.13	5.83	5.81
circles	1.38	0.30	0.87
crosses	1.48	0.65	1.09
lena	6.69	6.32	6.29
Mittelwert	4.74	4.14	4.37

Tabelle 4.3: Kompressionsergebnisse des adaptiven arithmetischen Kodierers mit und ohne Schätzwertkorrektur

Abbildung 4.5 zeigt das Modell welches sich mit der Verwendung des adaptiven arithmetischen Kodierers ergibt. Die Bilddaten werden einzeln eingelesen und der Schätzwert davon gebildet. Dieser wird anschließend vom adaptiven arithmetischen Kodierer mit der entsprechenden Häufigkeit kodiert. Die Wahrscheinlichkeit des Schätzwertes, die sich aus der Häufigkeit ergibt, steigt mit der Anzahl des Auftretens.

Die Ergebnisse der Analyse zeigen eine deutliche Verbesserung der Kompression gegenüber dem originalen JPEG-LS Algorithmus (siehe Tabelle 4.3). Sowohl die Kompression mit Schätzwertkorrektur als auch ohne schneiden besser ab als das Original. Dabei ist auch interessant zu sehen, dass der adaptive arithmetischer Kodierer ohne Korrektur des Schätzwertes bei Bildern mit größeren homogenen Flächen (circles, crosses) deutlich besser ist als mit. Dagegen sieht es bei korrelierten Bildern, wobei der Unterschied nicht so groß ist wie bei homogenen, umgekehrt aus. Von großer Bedeutung ist das Kompressionsergebnis ohne Korrektur des Schätzwertes, für den die Kontextmodellierung benötigt wird. Der original JPEG-LS Algorithmus erzielt gegenüber dem adaptiven arithmetischen Kodierer ohne Schätzwertkorrektur nur für ein Bild eine bessere Kompression. In den restlichen Tests ist der adaptive arithmetische Kodierer um einiges besser.

Eine Verbesserung des adaptiven arithmetischen Kodierers wäre der PPM-Algorithmus. Der PPM-Algorithmus kodiert und dekodiert wie der arithmetische Kodierer, nur in der Modellierung unterscheiden sich die beiden Algorithmen. Der PPM-Algorithmus aktualisiert die Wahrscheinlichkeiten der Symbole in Abhängigkeit bereits vorhergegangener Symbole.

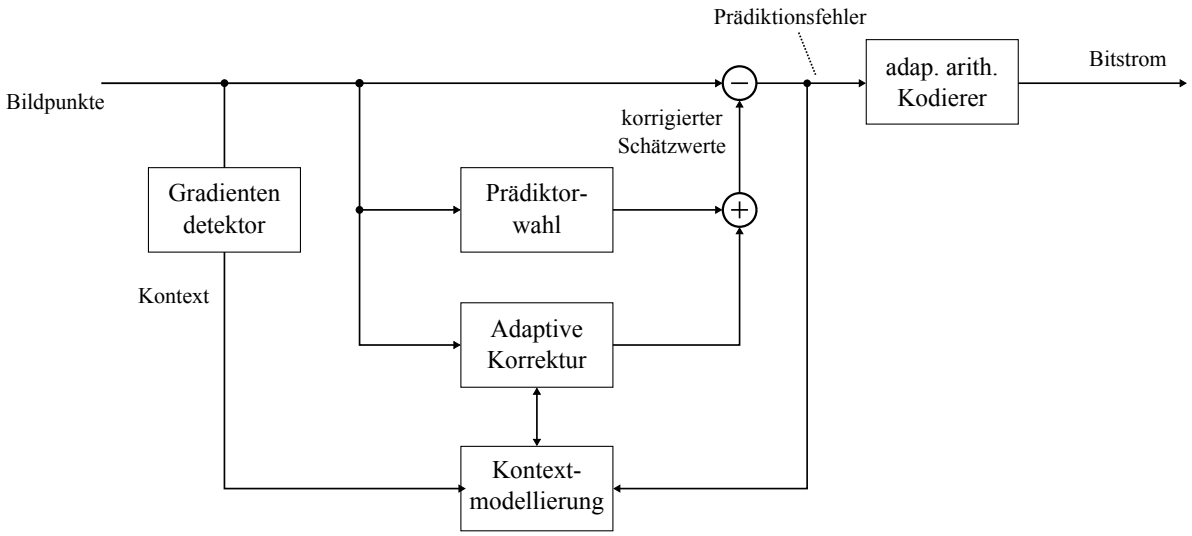

Abbildung 4.6: Modell zur Kompression von Bilddaten mit dem arithmetischen Kodierer und mit adaptiver Schätzwertkorrektur

5 JPEG-LS mit PPM

Nachdem gute Kompressionsergebnisse mit dem adaptiven arithmetischen Kodierer im vorherigen Kapitel erzielt wurde, wird in diesem Kapitel die Variante C des PPM-Algorithmus (PPM-C) implementiert und später als Entropiekodierer im JPEG-LS Algorithmus eingesetzt. Dazu wird zunächst der PPM-Algorithmus mit dem Wissen aus Abschnitt 4.3 und Kapitel 3 implementiert. Im folgenden werden die Funktionsweise und Unterschiede gegenüber dem adaptiven arithmetischen Kodierer erklärt.

5.1 PPM Implementierung

Der PPM-Algorithmus wird in Matlab implementiert. Matlab ist eine Hochsprache, mit der mathematische rechenintensive Aufgaben gelöst werden können. Sie ist auch auf numerische Berechnungen mithilfe von Matrizen spezialisiert. Der PPM-Algorithmus wird in fünf Dateien, bestehend aus *encoder.m*, *decoder.m*, *model.m*, *bit_input.m* und *bit_output.m* gegliedert. Die Dateien *bit_input.m* und *bit_output.m* sind identisch mit denen des adaptiven arithmetischen Kodierers.

5.1.1 model.m

Die Datei *model.m* besteht aus den Methoden *start_model()* und *update_model_ppm()*.

start_model

Die Methode *start_model()* muss vor Beginn des Kodiervorgangs aufgerufen werden, um notwendige Parameter, die auch von anderen Methoden genutzt werden, zu deklarieren. Das wären folgende Parameter:

- No_of_chars = 258
- No_of_symbols = No_of_chars+1
- Max_frequency = 16383
- iP

- iC

iP und iC sind jeweils Array-Felder. Das Array-Feld enthält die Häufigkeiten der Symbole und wird zu Beginn mit Einsen initialisiert. iC enthält die kumulativen Häufigkeiten der Symbole und wird mit

```
for j = 1:No_of_symbols+1
    iC(j) = No_of_symbols − j;
end
iP(1) = 0;
```

initialisiert. Sowohl iP und iC gehören zur Ordnung -1 und werden nicht verändert. Für Ordnungen größer -1 werden die Zweidimensionalen Arrays f und c verwendet. Während eine Dimension von der maximalen Ordnung abhängt, hängt die zweite von der Anzahl der Kontexte, die während der Kodierung bzw. Dekodierung auftreten, ab.

update_model

Die Methode *update_model_ppm* wird nach jedem Kodieren eines Symbols ausgeführt, damit die entsprechenden Wahrscheinlichkeiten aktualisiert werden. Dazu wird die aktuelle Kontextinformation, die vom Kodierer bzw. Dekodierer in der Variablen *in_context* gespeichert wurde, untersucht. Zuerst werden die neu gesammelten Kontextinformationen nach deren Ordnung aufgetrennt und im Array *ContextU()* vorübergehend gespeichert. Anschließend werden diese mit den existierenden verglichen. Falls ein Kontext nicht existiert wird der neue Kontext in das Array *context*, in welchem die Kontexte gespeichert werden, aufgenommen. Wenn jedoch der Kontext bereits existiert wird dessen Häufigkeit f aktualisiert, indem es um eins inkrementiert wird. Nach der Aktualisierung der Häufigkeiten, werden noch die kumulativen Häufigkeiten c aktualisiert.

5.1.2 encoder.m

start_encoding

Der Kodierer wird zunächst mit der Methode *start_encoding* initialisiert. Dabei werden wie bei der adaptiven arithmetischen Kodierung die Parameter für die Intervall-Quadranten definiert mit

```
Top_Value = bitsll(1,Code_value_bits)−1;
First_qtr = fix(Top_Value/4)+1;
Half = 2∗First_qtr;
Third_qtr = 3∗First_qtr;
```

wobei

Code_value_bits = 16;

ist. Der Parameter Codevalue legt die Anzahl der Bits eines Kodewortes fest. Mithilfe des Codevalue Parameters wird der Topvalue Parameter bestimmt. Dieser gibt den maximal möglichen Wert an, den ein Kodewort annehmen kann.

Zusätzlich werden das *ESC*-Symbol, welches dem Dekodierer einen Kontext-Wechsel signalisiert und das *EOF*-Symbol, welches das Ende des Dekodiervorgangs signalisiert wie folgt deklariert:

No_of_chars = 258;
No_of_symbols = No_of_chars+1;
Max_frequency = 16383;
EOF_symbol = No_of_chars+2;
ESC_symbol = No_of_chars+1;

Zu beachten ist, dass in Matlab der Wert 0 als Index eines Arrays nicht verwendet werden darf. Daher gilt Array-Index > 0. Die Werte von 1 bis 256 (2^8) werden für die 8-Bit Bilddaten benötigt und jeweils ein Bit für das *Esc*- und *EOF*-Symbol. Außerdem werden in dieser Methode die Intervallgrenzen mit $low = 0$ und $high = Top_Value$ definiert. Der Parameter *bits_to_follow* wird später für die Ausgabe von Bits benötigt und hier mit 0 initialisiert.

encode_file

Zum Starten des Kodiervorgangs muss die Funktion *encode_file(file, max_content, dest)*, mit den dafür notwendigen Parametern aufgerufen werden. *file* enthält enthält den Namen der zu kodierenden Datei, *max_content* die maximale Kontextlänge (bzw. Ordnung) und *dest* den Namen der Zieldatei. Nach Aufruf dieser Funktion werden als erstes die Parameter *maxContextOrder* mit *max_context* und das Array *f*, wegen der Regel Array-Index > 0 mit der Größe 1 bis $maxContextOrder + 1$ definiert. Somit werden Kontexte der Ordnung 0 in *f1*, Kontexte der Ordnung 1 in 2, usw. gespeichert. Als nächstes werden die Methoden *start_model()* und *start_encode* aufgerufen und die Eingabedatei *file* geöffnet.

Die einzelnen Symbole werden in die Variable *ch* eingelesen und im Buffer *in* mit der Länge des maximal möglichen Kontextes gespeichert (Abbildung 5.1). Dieser ist zu Beginn leer und wird mit dem Kodieren der Symbole aufgefüllt.

Anschließend werden diese in einer Schleife kodiert, indem die Funktion *encode_symbol(Symbol)* mit dem aktuell zu kodierenden Symbol als Parameter aufgerufen wird. Als letztes Symbol nach der Schleife wird das *EOF*-Symbol kodiert und die Methoden *done_encoding()* und *done_outputing_bits* aufgerufen, mit denen die restlichen Bits ausgegeben werden.

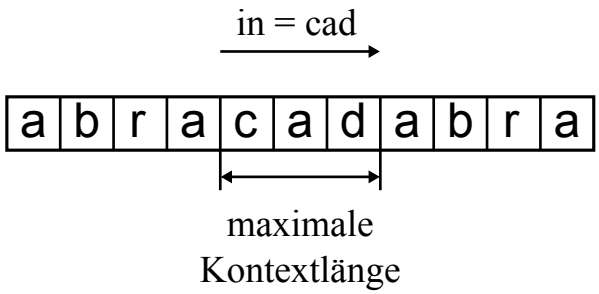

Abbildung 5.1: Eingabepuffer *in*

check_context

In dieser Methode wird mit Hilfe des Puffers *in*, welches das aktuell zu kodierende Symbol und die vorhergehenden Symbole (Kontextinformation) enthält, überprüft ob das Symbol bereits vorher in diesem Kontext aufgetreten ist oder nicht. Dazu wird zuerst die Ordnung (*contextNum*) überprüft in der sich der Kodierer aktuell befindet. Ist dies nicht der Fall wird ein *Esc*-Symbol kodiert und die Suche in der nächstkleineren Ordnung fortgesetzt. Dies wird solange wiederholt bis entweder ein Treffer erzielt oder die Ordnung 0 erreicht wird. Wenn der Kodierer in der Ordnung 0 angelangt ist, überprüft es ob das Symbol bisher alleine auftrat, falls dass der Fall ist wird die Methode *encode_symbol* aufgerufen und das Symbol, sowie die entsprechend kumulative Häufigkeit *ccontextNum* übergeben. Tritt, dagegen das Symbol zum ersten mal auf, wird ein *Esc*-Symbol kodiert und in die Ordnung -1 gewechselt, in der jedem Symbol eine Häufigkeit zugewiesen ist.

encode_file

Diese Methode funktioniert wie der Kodierer des adaptiven arithmetischen Kodierers (Abschnitt 4.3.2).

5.1.3 decoder.m

Bei dem Dekodierer wird vieles analog zum Kodierer implementiert. Deshalb werden hier nur die wichtigsten Unterschiede beschrieben. Die Datei *decoder.m* besteht aus den Methoden *start_decoding*, *d_check_context decode_symbol* und *decode_file*.

start_decoding

Analog zum Kodierer werden in dieser Methode die Intervallgrenzen *high* und *low*, *Top_Value* usw. initialisiert. Zusätzlich dazu wird die Variable *value* mit

```
value = 0;
i = 1;
while i<=Code_value_bits
      value = 2*value+biti.input_bit();
      i = i + 1;
end
```

initialisiert.

decode_file

Die Methode funktioniert überwiegend wie der Kodierer, mit dem Unterschied, dass sobald der Dekodierer ein *EOF*-Symbol dekodiert, es den Dekodiervorgang stoppt und das sich in der Variablen *stdout* befindende Ergebnis, in eine Ausgabedatei speichert.

5.1.4 Validierung des PPM

In diesem Kapitel soll die Implementierung des PPM validiert werden. Dazu wird die book Datei aus dem Calgary Corpus Testdatenset komprimiert und das Ergebnis mit dem aus Abbildung 3.1 verglichen. Aus Skalierungsgründen wird nicht die ganze Datei kodiert, sondern nur ein Teil davon.

Abbildung 5.2 zeigt die Kompression in Bit pro Zeichen (Bit per Character) als Funktion von der maximalen Kontextlänge. Das Ergebnis zeigt nicht nur einen ähnlichen Verlauf der Kennlinie wie in Abbildung 3.1 sondern auch fast identische Werte. Der Dekodierer wurde auch auf Funktion geprüft und bestätigt.

Nachdem die Implementierung von PPM validiert wurde, kann der Algorithmus im JPEG-LS Algorithmus eingesetzt und analysiert werden.

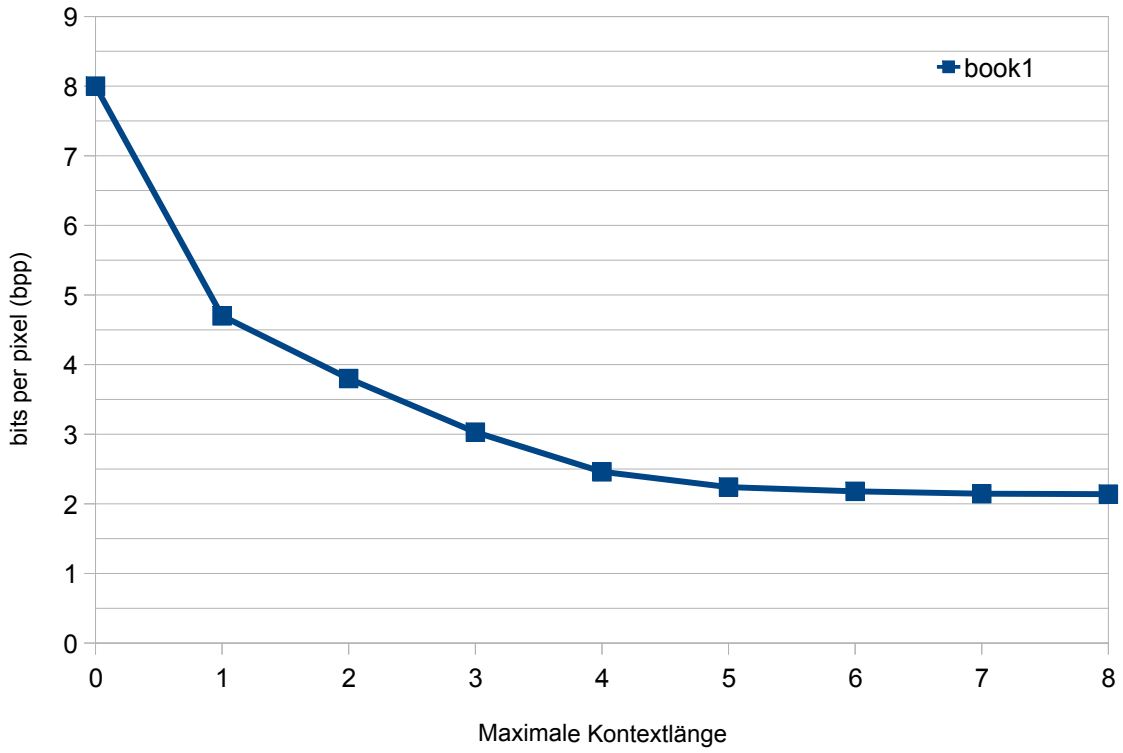

Abbildung 5.2: Kompressionsergebnis der PPM-C Implementierung

5.2 JPEG-LS und PPM

Im folgenden wird der JPEG-LS in Kombination mit den im vorherigen Kapitel implementierten PPM-C Algorithmus analysiert. Dazu wird der modifizierte JPEG-LS, wie bereits in vorherigen Untersuchungen geschehen, sowohl mit korrigiertem Schätzwert (Abbildung 5.3) als auch ohne korrigiertem ausgeführt. Die Ergebnisse werden in Kapitel 6 mit den Ergebnissen aus den vorherigen Untersuchungen verglichen und analysiert.

Während der Implementierung des JPEG-LS mit dem PPM-Algorithmus als Entropiekodierer wurde dieser immer wieder auf dessen Funktion überprüft. Dabei wurde eine nicht so gute Skalierung des PPM-Algorithmus festgestellt. Dies liegt zum einem an Matlab selbst, welcher nicht so gut skaliert wie andere Programmiersprachen (z.B. C++) und zum anderen an der Größe des Alphabets, welches zur Kodierung verwendet wird. Je größer das Alphabet ist desto mehr Kontexte können auftreten. Dies wirkt sich auf die Performance des Algorithmus aus. Die Bildpunkte eines Bildes können Werte zwischen 0 und 255 annehmen. Dies entspricht einer Alphabetgröße von 256 Symbolen. Mit dem Bit-plane-slicing Verfahren können die Bilddaten in einzelne Bit-Ebenen gruppiert werden. Dadurch ergibt sich für jede Ebene eine

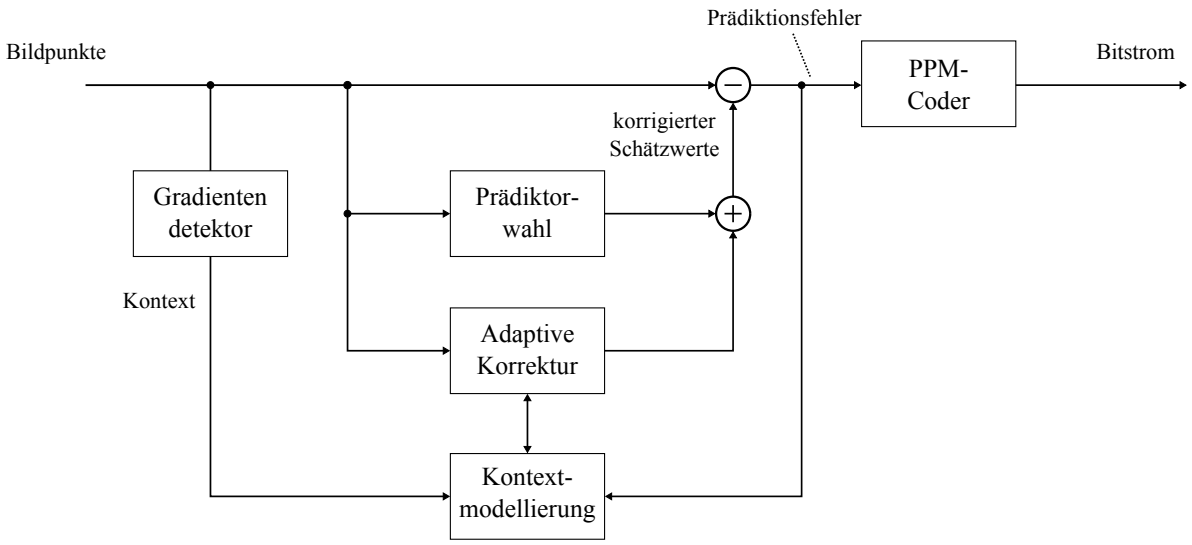

Abbildung 5.3: Modell zur Kompression von Bilddaten mit dem PPM-Kodierer

Alphabetgröße von zwei. Im folgenden wird das Bit-plane-slicing Verfahren vorgestellt, und später in Kombination mit JPEG-LS und PPM untersucht.

5.2.1 Bit-Plane-Slicing

Das Bit-Plane-Slicing Verfahren (Bit-Ebenen-Schachtelung) unterteilt die Pixel (Bildpunkte) eines Bildes in Bit-Ebene. Angenommen, die Pixel eines Bildes werden durch 8-Bit repräsentiert, dann ergeben sich daraus 8 Ebenen (Abbildung 5.4). Dabei befinden sich die wichtigen Bildinformationen in den höherwertigen und die unwichtigen in der niederwertigen Ebene (siehe Abbildung 5.7 und 5.8). Durch diese Methode ist es möglich, die einzelnen Bit-Ebenen zu analysieren und mit unterschiedlichen Kompressions-Algorithmen zu Kodieren. Später soll die Kompressionrate vom PPM-Algorithmus für die verschiedenen Ebenen untersucht werden.

Implementierung

Das Bit-plane-slicing Verfahren zu realisieren ist relativ einfach. Die einzelnen Bits der Ebenen werden in einem dreidimensionalen Array-Feld gespeichert. Die Größe des Array-Feldes hängt von der Anzahl der Bits eines Pixels und der Größenordnung des Bildes ab. Für ein Bild der Größe 512x512 mit $8 - Bit$ pro Pixel entspricht das einem acht mal 512 zu 512 Array-Feld. Nachdem das Array-Feld definiert wurde, werden die Pixel im Rasterscan-Verfahren durchlaufen. Bei jeder Iteration werden die einzelnen Bits des Pixels

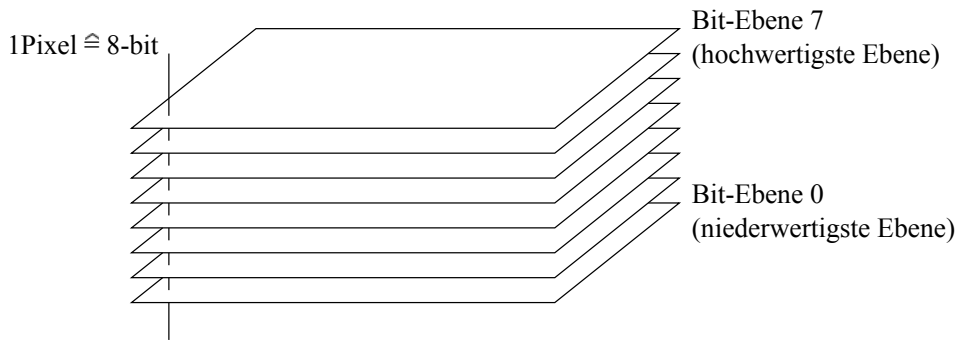

Abbildung 5.4: Bit-Plane-Slicing Darstellung

analysiert und in das entsprechende Array-Feld geschrieben.

Listing 5.1: Bit-plane-slicing Matlab-Code

```
in_img = imread(infile);   //read image
[m n] = size(in_img);             //create image of image size (e.g. 512x512)
plane = cell(1,8);            //create cell with size of 8
for i = 1:m
 for j = 1:n                  //raster-scanning
  b = f(i,j);
  for k = 8:-1:1              //loop through all planes, starting with the highest
     if (b>=2^(k-1))          //check if b has corresponding bit for plane k
         b = b-(2^(k-1));
         plane{1,k}(i,j) = 1;
     else
         plane{1,k}(i,j) = 0;
     end
  end
 end
end
imwrite(plane{1,1},'Cameraman_plane0.bmp','BMP') %// output file
imwrite(plane{1,2},'Cameraman_plane1.bmp','BMP')
imwrite(plane{1,3},'Cameraman_plane2.bmp','BMP')
imwrite(plane{1,4},'Cameraman_plane3.bmp','BMP')
imwrite(plane{1,5},'Cameraman_plane4.bmp','BMP')
imwrite(plane{1,6},'Cameraman_plane5.bmp','BMP')
imwrite(plane{1,7},'Cameraman_plane6.bmp','BMP')
imwrite(plane{1,8},'Cameraman_plane7.bmp','BMP')
}
```

Die Güte der einzelnen Ebenen kann durch eine andere Darstellung der Dezimalzahl gesteigert werden. Für gewöhnlich werden diese in Binärform (Binärcode) dargestellt. Eine Alternative zu der Binärform ist die Darstellung als Gray-Code, mit der die Güte der Ebenen gesteigert werden kann.

Gray-Code

Der Gray-Code ist ein Binär-Code, bei dem sich benachbarte Kodewörter nur in einem Bit unterscheiden. Dezimalzahlen in Binärform b, auch Binärcode genannt, werden in Gray-Codes umgeformt, indem b zuerst um 1 Bit nach rechts verschoben wird (Bit-shift) und das daraus resultierende Kodewort mit b einer XOR-Operation verknüpft wird. Für die Implementierung wird der selbe Kode wie im Listing 4.1 verwendet, mit dem Unterschied, dass b vor der Unterteilung in Bit-Ebenen in ein Gray-Code umgewandelt wird.

Listing 5.2: Bit-plane-slicing mit Gray-Codes Matlab-Code

```
in_img = imread(infile);   //read image
[m n] = size(in_img);             //create image of image size (e.g. 512x512)
plane = cell(1,8);             //create cell with size of 8
for i = 1:m
 for j = 1:n                   //raster-scanning
  b = f(i,j);
  a = bitshift(b,-1);   //right-shift of b
  b = bitxor(b,a);    // a XOR b
  for k = 8:-1:1               //loop through all planes, starting with the highest
     if (b>=2^(k-1))           //check if b has corresponding bit for plane k
         b = b-(2^(k-1));
         plane{1,k}(i,j) = 1;
     else
         plane{1,k}(i,j) = 0;
     end
  end
 end
end
}
.
.
.
```

Abbildung 5.7 und 5.8 zeigen die Bit-Ebenen des Bildes 5.6. Während in der Bit-Ebene 0 mit binärer Kodierung nichts außer rauschen zu erkennen ist, kann in der selben Ebene mit Gray-Code Kodierung bereits kleinere Flächen erkannt werden. Die Gray-Code Kodierung erzielt in allen Ebenen, bis auf die hochwertigste eine bessere Qualität als die binäre Kodierung. Die hochwertigste Ebene ist dagegen identisch mit der Ebene der binären Kodierung. Diese Eigenschaft des Gray-Codes sollte ausreichen um gegenüber der binären Kodierung eine bessere Kompression zu erreichen, wenn auch nur minimal. Einen weiteren Vorteil hat die Aufteilung in Ebenen hinsichtlich der Parallelisierung. Damit wird es möglich die Bilddaten, nach Auftrennen in Bit-Ebenen, unabhängig voneinander und parallel zu kodieren.

Nachdem das Bild in Abbildung 5.6 in die einzelnen Bit-Ebenen aufgeteilt wurde, werden diese zunächst mit dem PPM-Algortihmus ohne Prädiktion für verschiedene Kontextlängen kodiert. Die Ergebnisse der Gray-Code Darstellung sind in Abbildung 5.5 dargestellt. Die Abbildung zeigt eine hohe Kompression der höherwertigen Bits, wobei bei steigender

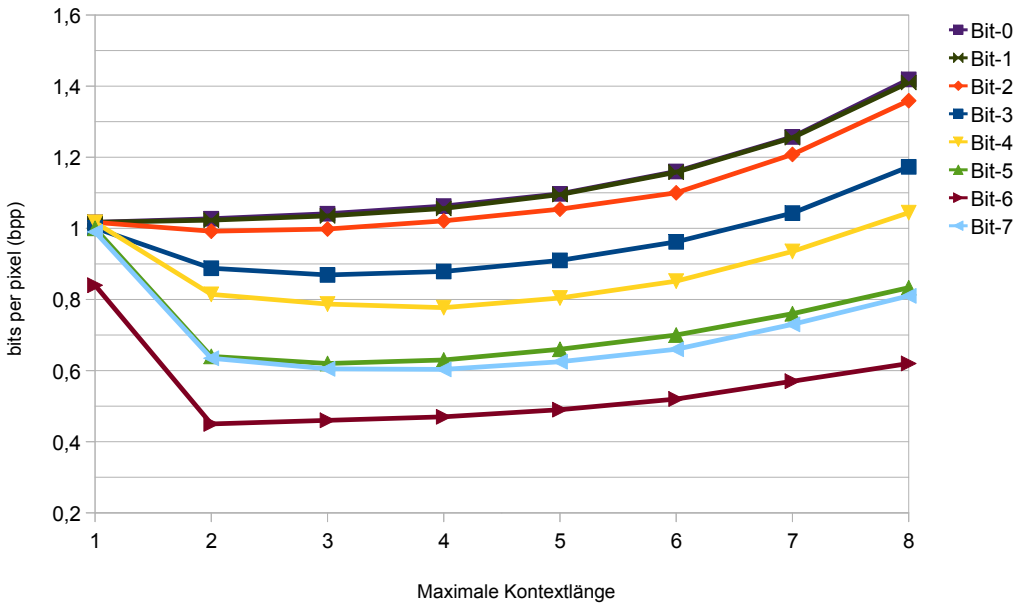

Abbildung 5.5: Bit-plane-slicing: Kompressionsergebnisse der Gray-Code Darstellung

Abbildung 5.6: Original-Bild cameraman

maximaler Kontextlänge die Kompression anfängt abzunehmen. Die Bilddaten der niederwertigen Bits werden dagegen kaum komprimiert. Die Performance vom PPM-Algortihmus in Verbindung mit Bit-plane-slicing hat sich gegenüber der Kodierung von unbearbeiteten Bildern verbessert. Werden jedoch die einzelnen Resultate der Komprimierung zusammenaddiert ergibt das hinsichtlich der Kompression gegenüber der Kodierung des Originalbildes als Ganzes, ein schlechteres Gesamtresultat. Die Kompressionsergebnisse, der Bit-Ebenen in Binärdarstellung, sind gegenüber den in Gray-Code schlechter. Aus diesem Anlass wird dieser nicht weiter untersucht.

Abbildung 5.7: Bit-Plane-Slicing: Binäre Kodierung (links) und Gray-Code (rechts)

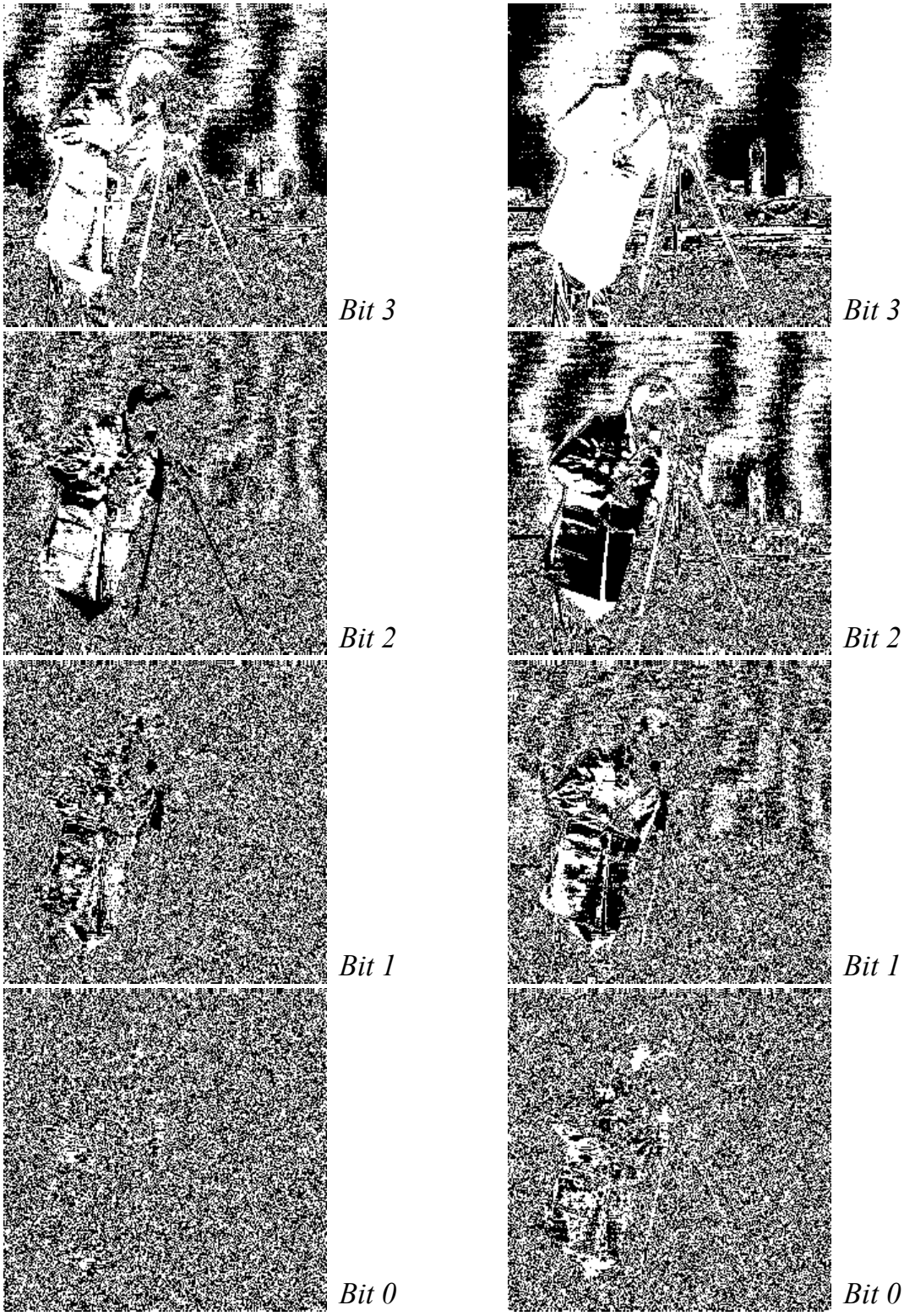

Abbildung 5.8: Bit-Plane-Slicing: Binäre Kodierung (links) und Gray-Code (rechts)

5.2.2 Kontextquantisierung

In diesem Kapitel wird die Kontextanzahl im PPM-Algorithmus analysiert. Die Kontextanzahl hängt von der Order und der größe des Alphabetes ab, mit der PPM ausgeführt wird. Je größer dieser ist desto größer wird die Anzahl der Kontexte. Das wirkt sich wiederum negativ auf die Performance aus. Dies soll durch Quantisieren der Kontexte verhindert werden [ZAo8]. Ähnlich wie im JPEG-LS Algorithmus, wo die Kontextanzahl durch Quantisieren reduziert wird, soll die Anzahl der Kontexte in PPM reduziert werden. Dafür werden zwei Kontexte die sich gleichen (z.B. $C_1 = [100, 67, 89, 205]$ und $C_2 = [98, 69, 91, 204]$) zusammengefasst. Das eignet sich vor allem für Bilddaten, da hier auf Grund der Eigenschaften von Bilddaten sehr viele Kontexte die sich ähneln auftreten können.

Zwei Symbole s_1 und s_2 gelten als ähnlich wenn folgendes gilt:

$$|s_1 - s_2| \leq k$$

wobei k in diesem Fall der Ähnlichkeitsparameter ist, der vom Algorithmus festgelegt wird. Dies wird im folgenden auf die Kontexte adaptiert. Seien zwei Kontexte der Ordnung m mit $C_q = c_q^m c_q^m - 1 \cdots c_q^1$ und $C_d = c_d^m c_d^m - 1 \cdots c_d^1$ gegeben. Dann wird die Menge aller k-ähnlichen Kontexte für C_q und C_d wie folgt definiert:

$$S_q = \{(c_q^m c \pm k) \circ (c_q^{m-1} c \pm k) \circ \cdots \circ (c_q^1 c \pm k)\}$$

$$S_d = \{(c_d^m c \pm k) \circ (c_d^{m-1} c \pm k) \circ \cdots \circ (c_d^1 c \pm k)\}.$$

Das Zeichen $\circ$ stellt eine Konkatenation-Operation dar und der Ausdruck $c_i \pm k$ die Menge aller c_j mit der Distanz k zu c_i mit

$$c_i \pm = \{c_j | c_i - c_j| \leq k\}.$$

Die Konkatenation zweier Mengen A und B ist wie folgt definiert $A \circ B = \{a_i \circ b_j, \forall a_i \in A, \forall b_j \in B\}$. Im Gegensatz zum original PPM bei dem ein Kontextwechsel statt findet, falls ein Symbol nicht gefunden wurde, kann dies hier anders gelöst werden. Dazu wird der Ähnlichkeitsparameter k so lange erhöht bis ein passender Kontext gefunden wird. Dieses Verfahren reduziert die Anzahl der Kontextwechsel und folglich die Ausgabe von *Esc*-Symbolen, was sich positiv auf die Kompressionsrate auswirkt.

6 Evaluation

Nachdem das Bit-plane-slicing Verfahren nicht die erhoffte Verbesserung hinsichtlich der Kompression und Geschwindigkeit gebracht hat, wird in diesem Kapitel das Ergebnis des PPM-C Algorithmus in Verbindung mit JPEG-LS analysiert und verifiziert. Dabei werden die Ergebnisse aus diesem Kapitel mit den Ergebnissen aus den vorherigen Kapiteln verglichen. In Kapitel 4 wurde gezeigt, dass die adaptive arithmetische Kodierung als Ersatz für den Entropiekodierer des JPEG-LS Algorithmus eine bessere Kompression erzielt als das Original. Dies sollte beim PPM-Algorithmus nicht anders sein. Dazu wird dieser zunächst ohne Schätzwertkorrektur für die selben Bilddaten mit verschiedenen Kontextlängen getestet. Abbildung 6.1 zeigt die dazugehörigen Ergebnisse.

Darin wird deutlich, wo die Stärken des PPM-Algorithmus liegen, nämlich in der Modellierung der Kontexte. Aus diesem Grund besteht auch eine gewisse Ähnlichkeit zum JPEG-LS Algorithmus, welcher durch Kontextmodellierung das aktuell zu kodierende Bildsignal vorhersagt. Im Fall von PPM funktioniert das ähnlich, nur das hier die Wahrscheinlichkeiten in Abhängigkeit vom Kontext bestimmt werden. Während des Kodierens werden Kontextinformationen gesammelt, das bedeutet je häufiger ein Symbol in einem Kontext auftritt desto besser wird dieses Symbol später kodiert. Bezogen auf Bilder bedeutet dies, das Bilder mit weniger Korrelation besser kodiert werden als solche mit starker Korrelation. Dies wird sowohl in Abbildung 6.1 als auch in 6.2 bestätigt. Die natürlichen Bilder die eine höhere Korrelation aufweisen als die Bilder *circles* und *crosses* werden entsprechend schlechter kodiert.

Bilddatei	orig.	*adap.arith.*	Maximale Kontextlänge n (ohne adaptive Korrektur)					
	JPEG-LS	*Kodierer*	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
bird	5.14	5.19	5.12	5.50	5.89	6.14	6.22	6.23
bridge	7.64	7.05	7.17	7.79	7.99	7.99	7.99	8.00
camera	6.13	5.83	5.81	6.33	6.69	6.84	6.89	6.90
circles	1.38	0.30	0.11	0.10	0.10	0.10	0.11	0.12
crosses	1.48	0.65	0.27	0.19	0.15	0.13	0.13	0.14
lena	6.69	6.32	6.36	6.91	7.24	7.31	7.31	7.32
Mittelwert	4.74	4.14	4.14	4.47	4.68	4.75	4.78	4,79

Tabelle 6.1: Kompressionsergebnisse des PPM-C Algorithmus

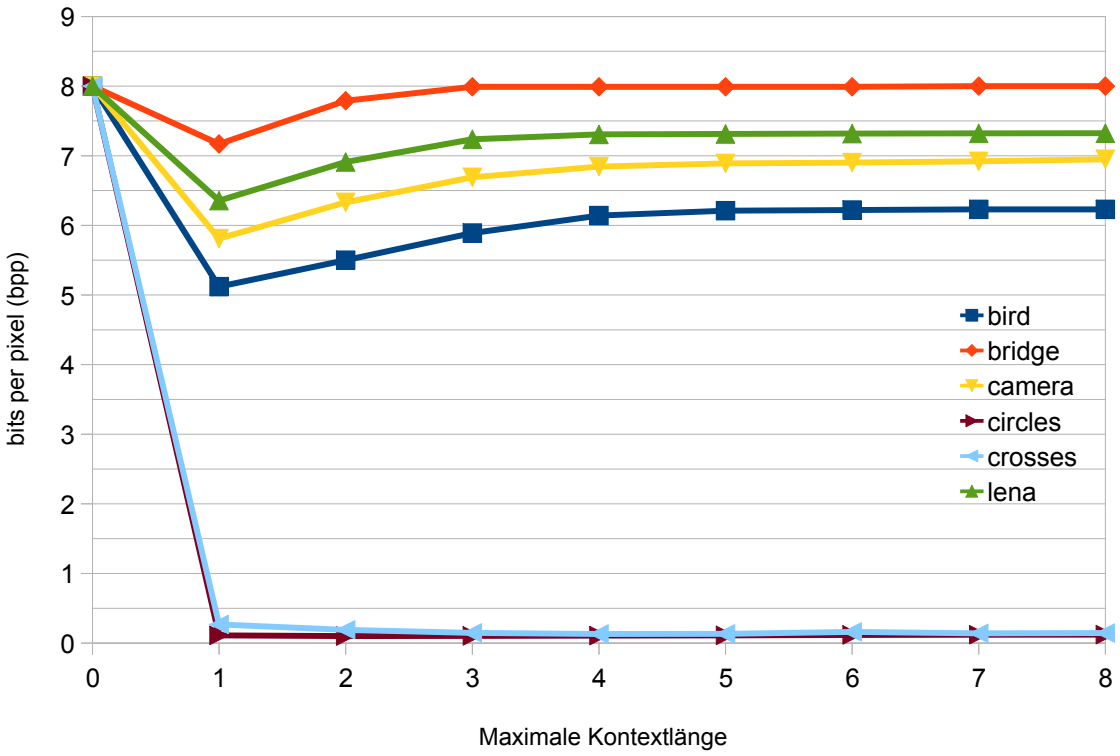

Abbildung 6.1: Kompressionsergebnisse des JPEG-LS mit PPM und ohne Schätzwertkorrektur

In Abbildung 6.2 sind die Kompressionsergebnisse des JPEG-LS Algorithmus mit PPM als Entropiekodierer und Schätzwertkorrektur zu sehen. Die Ergebnisse ähneln der ohne Schätzwertkorrektur, wie schon aus früheren Untersuchungen bekannt wahr. Auch hier liegt die optimale Kontextlänge bei eins und stagniert mit steigender Kontextlänge.

In Tabelle 6.1 sind die wichtigsten Kompressionsergebnisse noch einmal zusammengefasst und als Bit pro Pixel aufgelistet.

Während Textdateien vom PPM-Algorithmus bei einer maximalen Kontextlänge von fünf und mehr maximal komprimiert werden, sieht das bei Bildern anders aus. Diese werden bei einer maximalen Kontextlänge von eins optimal kodiert. Eine Erhöhung der Kontextlänge verschlechtert in vielen Fällen die Kompression. Eine Kontextlänge von eins entspricht theoretisch dem adaptiven arithmetischen Kodierer, da im Fall der Kontextlänge eins das Symbol selbst als Kontext betrachtet wird. Dies ist auch der Grund der ähnlichen Kompression (Tabelle 6.1).

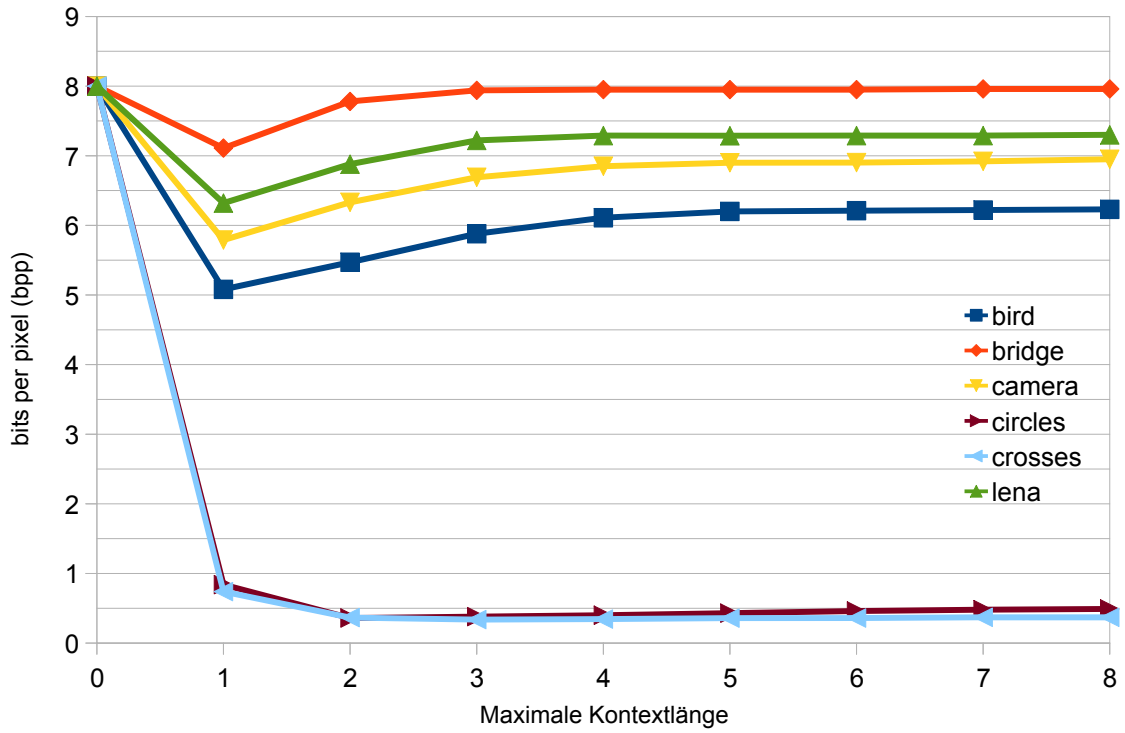

Abbildung 6.2: Kompressionsergebnisse des JPEG-LS mit PPM und Schätzwertkorrektur

Ein Schwachpunkt des Algorithmus beim Kodieren von Bildern ist die Kontextlänge. Es kann durch Erhöhen der Kontextlänge keine Steigerung der Kompression erreicht werden. Das liegt an der Anzahl von *Esc*-Symbol, die kodiert werden müssen, wenn in höheren Kontexten kein Treffer erzielt wird. Dies kann durch eine Quantisierung der Kontexte, wie in Kapitel 5.2.2 beschrieben, verhindert werden. Dabei werden Kontexte nicht nur nach dem Kriterium der vorhergehenden Symbolen aufgeteilt. Stattdessen werden die Kontexte untereinander und an Bedingungen verknüpft.

7 Zusammenfassung und Ausblick

Wie sich zeigte, stellt die Suche nach einem alternativen Entropiekodierer mit dem Ziel den JPEG-LS Algorithmus zu Parallelisieren kein einfaches Unterfangen dar.
In Kapitel 2 wurden die Grundlagen des JPEG-LS Algorithmus umfassend vorgestellt, indem die einzelnen Komponenten des JPEG-LS genauer beschrieben wurden. Durch Prädiktion wird es möglich die Grauwerte eines Bildes um den Wert 0 zu konzentrieren. Diese Methode wird auch in einer leicht verbesserten Variante im JPEG-LS eingesetzt. Der Schätzwert, welcher vom Prädiktor berechnet wird, wird mit der Kontextmodellierung angepasst. Außerdem wird mit Hilfe der Kontextmodellierung ein geeigneter Wert für den Parameter k des Golomb-Rice-Coders bestimmt. Durch Quantisierung gelingt eine Reduzierung der Kontextanzahl, was sich auf die Performance positiv auswirkt. Nachdem der Schätzwert korrigiert wurde, wird dieser vom eigentlich zu kodierendem Bildsignal abgezogen und mit dem Golomb-Rice-Coder kodiert.
Es hat sich herausgestellt, dass eine Parallelisierung des JPEG-LS aufgrund der Kontextmodellierung nicht einfach umzusetzen ist. Folglich wurde das Entfernen der Kontextmodellierung in Erwägung gezogen. Dazu wurde zuerst der Golomb-Rice-Coder mit festem Parameter k initialisiert und analysiert. Nachdem kein optimaler statischer Wert für den Golomb-Rice-Coder gefunden werden konnte, wurde dieser durch den PPM-Algorithmus ersetzt. PPM steht für Prediciton by Partial Matching und ist ein kontextabhängiger Entropiekodierer, welcher auf Textdaten spezialisiert ist und Symbole arithmetisch kodiert. Folglich wurde der arithmetische Kodierer vorgestellt und der adaptive arithmetische Kodierer implementiert, auf dem aufbauend später der PPM-Kodierer implementiert wurde. Beide Verfahren konnten in Kombination mit dem JPEG-LS eine bessere Kompression als der original JPEG-LS Algorithmus erzielen. Wobei höhere Kontexte beim PPM-Algorithmus keinen Vorteil verschaffen konnten, das zum Teil an der großen Anzahl der Kontexte, die durch die Größe des Alphabets auftraten und dem dadurch entstehenden Kontextwechsel, bei dem immer ein *Esc*-Symbol kodiert werden muss, lag.
Um die Größe des Alphabets zu reduzieren wurde das Bild mit dem Bit-plane-slicing und dem Gray-Code in seine Bit-Ebenen aufgeteilt. Anschließend wurden die Bilder die dadurch entstanden sind, separat kodiert. Dabei konnte im Gesamten keine Verbesserung der Kompression festgestellt werden. Folglich wurde der PPM-C Algorithmus als Entropiekodierer für den JPEG-LS eingesetzt und die Ergebnisse in Kapitel 6 analysiert.

7.1 Ausblick

Es gibt Versuche den arithmetischen Kodierer zu parallelisieren [MNFTJ99]. Mit dem Ziel den PPM-Kodierer zu parallelisieren könnte dieses Wissen auf den adaptiven arithmetischen Kodierer adaptiert werden. Genauso gibt es Versuche die Bilddatenkompression mit dem PPM-Algorithmus zu verbessern [ZA08]. Denkbar wäre eine Kombination beider Verfahren, die hinsichtlich der Kompressionsleistung und der Skalierung mit großer Wahrscheinlichkeit zu einer Optimierung des PPM-Algorithmus, führen könnte.

Literaturverzeichnis

[CTW95] Cleary, J. G. ; Teahan, W. J. ; Witten, I. H.: Unbounded length contexts for PPM. In: *Proc. Data Compression Conf. DCC '95*, 1995, S. 52–61 (Zitiert auf den Seiten 7, 31 und 45)

[CW84] Cleary, J. ; Witten, I.: Data Compression Using Adaptive Coding and Partial String Matching. 32 (1984), Nr. 4, S. 396–402. http://dx.doi.org/10.1109/TCOM.1984.1096090. – DOI 10.1109/TCOM.1984.1096090 (Zitiert auf Seite 41)

[Gol66] Golomb, S.: Run-length encodings (Corresp.). 12 (1966), Nr. 3, S. 399–401. http://dx.doi.org/10.1109/TIT.1966.1053907. – DOI 10.1109/TIT.1966.1053907 (Zitiert auf den Seiten 20 und 29)

[IC92] ITU-CCITT: *Recommendation T.81*. 1992. – Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines (Zitiert auf Seite 11)

[IHW87] I. H. Witten, J. G. C. R. M. Neal N. R. M. Neal: Arithmetic Coding for Data Compression. In: *Communications of the ACM* (1987), S. 520–540 (Zitiert auf Seite 37)

[IT98] ITU-T: *Recommendation T.87*. 1998. – Information technology – Lossless and near-lossless compression of continuous-tone still images – Baseline (Zitiert auf Seite 25)

[MNFTJ99] Mahapatra, S. ; Nunez, J. L. ; Foregrino-Thribe, C. ; Jones, S.: Parallel implementation of a multialphabet arithmetic coding algorithm. In: *Proc. IEE Colloquium Data Compression: Methods and Implementations (Ref. No. 1999/150)*, 1999 (Zitiert auf Seite 78)

[Mof90] Moffat, A.: Implementing the PPM data compression scheme. 38 (1990), Nr. 11, S. 1917–1921. http://dx.doi.org/10.1109/26.61469. – DOI 10.1109/26.61469 (Zitiert auf Seite 31)

[Str05] Strutz, Tilo: *Bilddatenkompression : Grundlagen, Codierung, Wavelets, JPEG, MPEG, H.264*. 3., aktualisierte und erweiterte Auflage. Wiesbaden : Vieweg, 2005 (Zitiert auf Seite 13)

[WSS00] Weinberger, M. J. ; Seroussi, G. ; Sapiro, G.: The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS. 9 (2000), Nr. 8, S. 1309–1324. http://dx.doi.org/10.1109/83.855427. – DOI 10.1109/83.855427 (Zitiert auf Seite 17)

[ZA08] Zhang, Yong ; Adjeroh, D. A.: Prediction by Partial Approximate Matching for Lossless Image Compression. 17 (2008), Nr. 6, S. 924–935. http://dx.doi.org/10.1109/TIP.2008.920772. – DOI 10.1109/TIP.2008.920772 (Zitiert auf den Seiten 71 und 78)

Alle URLs wurden zuletzt am 28.7.2010 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Volkan Keles)