Effiziente Kompression von bi-level Bilddaten durch kontextsensitive arithmetische Codierung

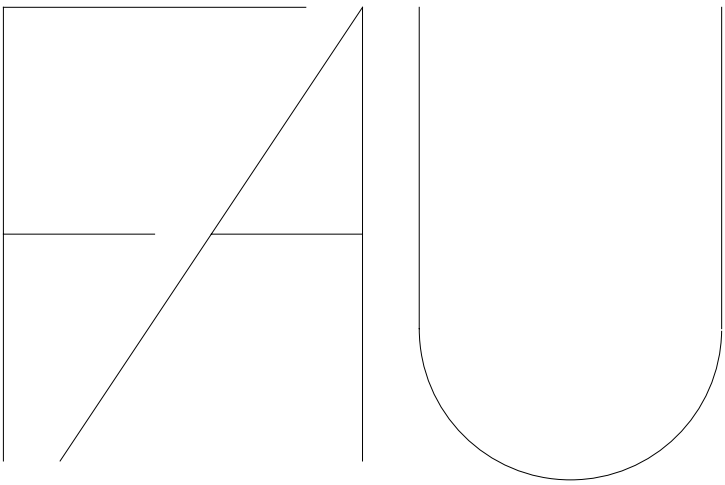
Markus Kuhn

Juli 1995 SA-I4-11-95

Studienarbeit

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)



Effiziente Kompression von bi-level Bilddaten durch kontextsensitive arithmetische Codierung

Studienarbeit im Fach Informatik

vorgelegt von

Markus Kuhn

geboren am 1. Januar 1971 in München

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: ***Prof. Dr. F. Hofmann***
Dipl.-Inf. D. Husemann

Beginn der Arbeit: 01.04.1995
Abgabe der Arbeit: 31.07.1995

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Uttenreuth, den 31.07.1995

Markus Kuhn

Kurzfassung

Im Rahmen dieser Arbeit wurde der JBIG-Algorithmus zur verlustfreien Kompression von bi-level Bilddaten, insbesondere von digitalisierten Dokumenten, als wiederverwendbare portable C Bibliothek implementiert, optimiert und getestet. Damit kann dieser aufwendige Algorithmus sehr einfach in Bild- und Dokumentenverarbeitungssysteme integriert werden. Spezielle Datenkompressionsverfahren für digitalisierte Texte und Dokumente werden in Zukunft eine große Bedeutung bei der Archivierung von wissenschaftlichem Schriftgut und bei dessen Bereitstellung für Bibliotheksbenutzer spielen. Verschiedene gängige Grundmechanismen der Datenkompression unter besonderer Berücksichtigung der in JBIG eingesetzten Verfahren werden vorgestellt, ebenso wie diese Implementation des JBIG-Verfahrens, die besonders im Hinblick auf den Einsatz in interaktiven Systemen entwickelt wurde. Die Leistungsfähigkeit des Verfahrens wird anhand einer Reihe von Beispielen ausgewertet und es zeigt sich, daß abhängig von der Art, Bildqualität und Buchstabendichte des Dokuments Kompressionsfaktoren zwischen etwa 4 und 60 möglich sind. Für digitalisierte wissenschaftliche Veröffentlichungen typische Kompressionsfaktoren liegen im Bereich 8 bis 20. Ein schneller PC kann mit der vorgestellten Implementation komprimierte Daten mit über 64 kbit/s verarbeiten, so daß JBIG-Echtzeitkompression in Fax-Systemen mit den auf ISDN-Kanälen üblichen Übertragungsraten möglich ist.

Abstract

The JBIG algorithm for loss-less compression of bi-level images, especially scanned documents, has been implemented, optimized and tested as a portable C library. This implementation can now easily be included into image and document handling systems. Specialized data compression methods for scanned text and document images are going to play an increasingly important role in the future for archiving scientific publications and keeping them available for library users. Several widely used basic mechanisms for data compression with special emphasis on those used in the JBIG algorithm are described as well as details of this JBIG implementation which is especially designed for the needs of interactive applications. The performance of JBIG on a number of sample documents is evaluated demonstrating that compression ratios between around 4 and 60 depending on the type, image quality and character density of the document are achieved. Typical compression ratios obtained on scanned scientific publications are in the range 8 to 20. With the presented implementation, a fast personal computer can encode and decode at data rates over 64 kbit/s as required by real-time compression for fax transmission on ISDN channels.

Kuhn, M. *Effiziente Kompression von bi-level Bilddaten durch kontextsensitive arithmetische Codierung*. Studienarbeit, Lehrstuhl für Betriebssysteme, IMMD IV, Universität Erlangen-Nürnberg, Erlangen, Juli 1995.

Inhaltsverzeichnis

1 Einleitung

Datenkompressionsalgorithmen sind Verfahren, die einen gegebenen Datenstrom mit charakteristischen statistischen Eigenschaften in eine kompaktere aber äquivalente Darstellung umwandeln. Sie spielen heute in der modernen Kommunikationstechnik eine bedeutende Rolle. Dienste wie etwa der digitale GSM Mobilfunk, Telefax, Multimedia-Anwendungen auf Personal Computern und die kurz vor der Einführung stehende digitale Ausstrahlung von Fernsehprogrammen wären ohne leistungsfähige Kompressionsalgorithmen kaum praktikabel. Nur durch die drastische Reduktion des zu handhabenden Datenvolumens können die meist sehr beschränkten Übertragungs- oder Speichermedien die geforderte Leistung erbringen.

Seit Claude E. Shannon mit seinem Artikel *A mathematical theory of communication* [Sha48] die theoretischen Grundlagen dazu vorbereitete, ist die Datenkompression Gegenstand umfangreicher Forschung. Jedoch erst in den letzten 15 Jahren sind Prozessoren hinreichend leistungsfähig geworden, um die oft schon länger bekannten, aber algorithmisch recht aufwendigen Datenkompressionsverfahren auf breiter Basis einsetzen zu können. Seit Ende der 70er Jahre werden routinemäßig einfachere Verfahren wie etwa die auch in heutigen Telefaxgeräten noch verwendete *run-length-* und Huffman-Codierung eingesetzt, doch erst Ende der 80er Jahre haben wesentlich komplexere Verfahren einen Reifegrad erreicht, der eine Standardisierung zuließ.

Inzwischen sind verschiedene Verfahren genormt worden, so etwa der JPEG Algorithmus (benannt nach der *Joint Photographic Experts Group* der Standardisierungsorganisationen ISO und CCITT) für photoähnliche Bilddaten, der MPEG (Motion Pictures Expert Group) Algorithmus für Videosignale, das V.42bis/LZBT Verfahren für die Kompression in Datenübertragungssystemen und Computernetzwerken oder der DCLZ-Algorithmus für die Datenkompression in Magnetband-Systemen. Ein weiteres gängiges Verfahren ist der Lempel-Ziv-Welch Algorithmus [Wel84], der im UNIX Hilfsprogramm *compress* sowie im auf dem Internet gängigen Bilddatenformat *CompuServe GIF* eingesetzt wird.

Die derzeit gängigen Verfahren lassen sich grob wie folgt unterteilen:

Es gibt eher einfache Algorithmen, die sich prinzipiell auf beliebige Datenströme anwenden lassen. Diese generischen Verfahren basieren häufig auf dem von Lempel und Ziv vorgeschlagenen Prinzip, im Eingangsdatenstrom nach sich wiederholenden Symbolfolgen zu suchen und diese durch kurze Referenzen auf vorangegangene identische Folgen zu ersetzen [Ziv77]. Derartige Techniken haben sich insbesondere bei der Kompression von Text- und Programm-Dateien sowie Netzwerkprotokollen durchgesetzt, werden aber auch auf Bilddaten angewendet.

Die zweite große Klasse von Kompressionsverfahren ist dagegen auf spezielle Anwendungen zugeschnitten und es werden charakteristische Eigenschaften von Datenquelle und -senke ausgenutzt, sowie besondere Anforderungen der Anwendung berücksichtigt. So ist es zum Beispiel bei Bild- und Audiosignalen, die nicht mehr weiter analysiert, sondern ausschließlich für Menschen wahrnehmbar gemacht werden sollen, möglich, große Informationsmengen zu entfernen, die das Auge oder Ohr aufgrund von psychophysiologischen Effekten garnicht wahrnehmen kann, ohne dabei die Qualität der übertragenen Information nennenswert zu beeinflussen. Daraus ergibt sich die wichtige Unterscheidung der anwendungsspezifischen

Verfahren in verlustfreie Algorithmen, die die Ausgangsdaten Bit für Bit identisch wiederherstellen, sowie verlustbehaftete Methoden, bei denen die Ausgangsdaten den Eingangsdaten nur bezogen auf die Anwendung sehr ähnlich sind.

Zu den charakteristischen Eigenschaften von Bildsignalen gehört beispielsweise die meist sehr starken Korrelationen zwischen benachbarten Bildpunkten, die zur Kompression erfolgreich ausgenutzt werden können. Eine spezifische Anforderung einer Anwendung an ein Kompressionsverfahren ist etwa die Möglichkeit, den Datenstrom aufzuteilen, wobei ein Teil bereits zur Darstellung eines qualitativ weniger hochwertigen Ausgangssignals ausreicht. Erst durch Hinzufügen des zweiten Datenstroms kann die volle erforderliche Signalqualität vom Decoder erreicht werden, wobei durch dieses Auftrennen insgesamt nicht wesentlich mehr Daten anfallen dürfen als ohne diesen Mechanismus. Eine konkrete Anwendung dafür ist beispielsweise die gleichzeitige digitale Ausstrahlung von Fernsehprogrammen in der normalen heutigen Bildauflösung sowie in einer künftigen höheren Qualität (HDTV, *high definition television*). Da Sendefrequenzen und damit die vorhandenen Gesamtbitraten knapp sind, soll vermieden werden, daß zusätzlich zu einem bereits gesendeten Programm in heutiger Auflösung teilweise redundant nocheinmal das komplette Bild in hochauflösender Qualität übertragen werden muß. Also empfängt ein HDTV-Gerät sowohl das Signal mit normaler Qualität als auch ein im Vergleich zum vollen HDTV-Signal kompakteres Differenzsignal, welches nur noch die zusätzliche für das HDTV-Bild erforderliche Information enthält und kombiniert beide zu einer hochauflösenden Darstellung.

Mit dem JBIG (*Joint Bi-level Image Experts Group*) Standard [ITU93a] wurde vor zwei Jahren ein spezialisiertes Verfahren für die sehr effiziente verlustfreie Kompression von bi-level Bilddaten veröffentlicht, insbesondere von mit hoher Auflösung digitalisierten Schriftstücken und Zeichnungen. Bei bi-level Bildern können die einzelnen Bildpunkte nur zwei mögliche Werte annehmen, beispielsweise schwarz und weiß. Neben den in der Literatur auch als *textual images* bezeichneten digitalisierten Dokumenten können beispielsweise ebenso die in Banken benutzen Unterschriftenkataloge oder die in der Kriminalistik eingesetzten Fingerabdruck-Archive als bi-level Bilder gespeichert werden.

Derzeit kommt im Rahmen der Entwicklung von sogenannten Multimedia-Anwendungen den Kompressionsverfahren für Photographien sowie Video- und Audiosignalen sehr große Aufmerksamkeit zu. Dagegen führen moderne Kompressionsverfahren für bi-level Bilder in der aktuellen Multimedia-Diskussion zu Unrecht eher etwas ein Schattendasein. Wie die folgenden beiden Anwendungsbeispiele belegen, könnte sich dies aber sehr bald ändern.

Es ist abzusehen, daß in den kommenden 10–15 Jahren ein grundlegender Wandel im wissenschaftlichen Publikationswesen eintreten wird. Auch wenn Lehrbücher und Monographien sicher noch sehr lange Zeit in Buchform erscheinen werden, so ist doch wahrscheinlich, daß ein großer Teil der wissenschaftlichen Fachveröffentlichungen bald nicht mehr in Papierform herausgegeben werden, sondern die Texte digital nur noch über Computer-Netzwerke von den Servern der Verlage und der wissenschaftlichen Standesorganisationen abrufbar sind. Schon lange haben Bibliotheken Probleme mit der kostenintensiven Beschaffung und Lagerhaltung wissenschaftlicher Zeitschriften und heute halten die meisten Bibliotheken ohnehin nur noch eine beschränkte Auswahl der verfügbaren Zeitschriften vor. Die amerikanische *Association for Computing Machinery (ACM)* plant als Herausgeberin zahlreicher Informatik-Fachzeitschriften noch 1995 sämtliche neuen Veröffentlichungen über Datenbanken zum weltweiten Abruf zur Verfügung zu stellen und bereits 1998 sollen daraufhin für die ersten Zeitschriften die Papier-Ausgaben eingestellt werden [Den95]. Bei neu erscheinenden Veröffentlichungen wird dabei eine Textdatei mit Strukturinformation

und *hyperlinks* im SGML-Format veröffentlicht, womit der Artikel zur beliebigen weiteren Bearbeitung zur Verfügung steht, und beispielsweise nicht nur ausgedruckt, sondern auch in verschiedene verteilte Hypertext-, Volltextsuche- und CD-ROM *information retrieval* Systemen weiterbenutzt werden kann.

Wenn jedoch erst einmal die Infrastruktur für die Verteilung von digitalen Veröffentlichungen geschaffen ist, dann wird schnell das Bedürfnis entstehen, auch den vorhandenen Altbestand an Veröffentlichungen auf diese Weise zur Verfügung stehen zu haben. Bei den vor etwa 1990 erschienen Veröffentlichungen müßte dazu der gesamte Text zunächst neu erfaßt werden, was angesichts der großen Anzahl von Texten nicht praktikabel ist. Also wird man sich darauf beschränken, ältere Veröffentlichungen nur sorgfältig zu digitalisieren und als Pixeldateien zur Verfügung zu stellen.

Als Alternative zwischen der sehr arbeitsaufwendigen vollständigen manuellen Neuerfassung einer Veröffentlichung und der recht speicherplatzaufwendigen Archivierung als Pixeldateien besteht die Möglichkeit der automatischen Texterkennung (*optical character recognition*, OCR). Dabei wird ein digitalisiertes Dokument einem Algorithmus vorgelegt, welcher versucht, die einzelnen Buchstaben und Symbole zu separieren und identifizieren. Als Ergebnis entsteht eine Textdatei wie sie in Textverarbeitungsprogrammen weiterbearbeitet werden kann. Jedoch sind derzeit OCR-Algorithmen noch mit einer ganzen Reihe von Nachteilen verbunden. Zum einen liegt die Fehlererkennungsrate je nach Qualität der Vorlage bei mehreren falsch erkannten Zeichen pro Seite. Dies mag bei manchen Anwendungen wie der digitalen Ablage von Geschäftsbriefen akzeptabel sein, aber gerade bei den in wissenschaftlichen Texten oft auftauchenden mathematischen Formeln und Zahlenangaben kann mit jedem einzelnen falsch erkannten Zeichen unwiderruflich wertvolle Information verloren gehen. Darüber hinaus müssen OCR-Algorithmen zunächst auf jede in einem Text auftretende Schriftart trainiert werden. Bei der automatischen Texterkennung gehen oft inhaltlich wichtige Details wie etwa die Hervorhebung durch eine kursive Schrift verloren und selbst wenn OCR-Algorithmen bei Dokumenten die nur Text enthalten zuverlässig arbeiten würden, so sind sie dennoch nicht in der Lage, alle feinen inhaltlich relevanten typographischen Merkmale von mathematischen oder chemischen Formel, Tabellen und Diagrammen zu erfassen. Liegt dagegen ein Dokument als Bilddatei vor, so ist eine künftige automatische Texterkennung immer noch möglich, um beispielsweise einen Stichwortindex zu erstellen. Aufgrund der Gefahr des unwiederbringlichen Informationsverlusts sind daher automatische Texterkennungsverfahren eher als Hilfe bei der manuellen Neuerfassung denn als akzeptable Alternative zur Archivierung als Bilddatei zu sehen.

Eine typische Zeitschriftenseite im Format A4, die für eine akzeptable Reproduktion mit mindestens 300 Punkte pro Zoll (11.8 Punkte/mm) Auflösung abgetastet werden muß, benötigt etwa 1 Megabyte Speicher bei einem Bit pro Pixel. Das bedeutet, daß auf einer heutigen CD-ROM lediglich etwa 650 A4-Seiten ohne Datenkompression gespeichert werden könnten, also nicht einmal der Umfang eines üblichen Zeitschriftenjahrgangs. Dies bedeutet auch, daß ein Anwender über eine Telefonleitung (30 kbit/s) über 4 Minuten pro Seite auf den Zugriff warten muß, und daß ein zentraler Server über einen ATM-Netzwerkanschluß (155 Mbit/s) pro Sekunde weniger als 20 Seiten versenden kann. Diese Beispiele demonstrieren klar die Bedeutung von leistungsfähigen Kompressionverfahren für digitalisierte Dokumente, mit denen das Datenvolumen um typischerweise den Faktor 8–20 reduziert werden kann, für derartige Anwendungen.

Doch nicht nur für die einfache Bereitstellung wissenschaftlicher Literatur sondern auch für deren langfristigen Erhalt ist in naher Zukunft der Einsatz von Kompressionsverfahren

für digitalisierte Texte von großer Bedeutung. Während sorgfältig hergestellte klösterliche Schriften aus dem Mittelalter noch heute in hervorragendem Zustand erhalten sind, so hat ein Großteil der im 20. Jahrhundert gedruckten Literatur nur eine Lebenserwartung von 50–80 Jahren [Rei94]. Da sich seit etwa 1840 in der Papierherstellung eine kostengünstige Harz-Alaun-Leimung durchgesetzt hat, enthält modernes Papier Schwefelsäure und säurebildende Substanzen, die im Zusammenwirken mit der allgegenwärtigen Luftfeuchte die Zellulosefasern des Papiers im Lauf der Jahre brüchig werden lassen. Es wurde beispielsweise in den USA geschätzt, daß dort 70–90% der Dokumente vom Papierzerfall betroffen sind und bereits 15–30% heute nicht mehr benutzbar sind. Eine Untersuchung des Deutschen Bibliotheksinstituts ergab, daß in Deutschland in wissenschaftlichen Bibliotheken etwa 60 Millionen Bücher und in Archiven etwa 350 km Regallänge vom Zerfall bedroht oder schon betroffen sind. Ohne entsprechende Maßnahmen wird damit das geistige Erbe der Menschheit des 20. Jahrhunderts nicht für die Nachwelt erhalten bleiben. Es werden derzeit verschiedene chemische Entsäuerungsverfahren entwickelt und erprobt, doch diese können den Zerfall nur verlangsamen, nicht aufhalten oder rückgängig machen. Eine weitere Alternative ist die Mikroverfilmung, welche die photographierten Texte für etwa 300 Jahre erhalten wird.

Der attraktivste Ausweg besteht auch hier in der Digitalisierung der betroffenen Texte und der komprimierten Speicherung auf digitalen Medien. Magnetische Speichermedien wie Disketten oder Bänder halten leider nur etwa 5–10 Jahre, aber für die einmal beschreibbare CD-WO Platte wird die Datensicherheit auf etwa 100 Jahre und für magneto-optische (MO) Platten auf Glasbasis wird die Haltbarkeit auf mehrere hundert Jahre geschätzt [Ste94]. Da sich digitale Daten beliebig oft verlustfrei kopieren lassen, kann durch die digitale Archivierung Literatur praktisch unbegrenzt gelagert werden, wenn der komplette Datenbestand an einigen großen Zentralarchiven redundant gehalten wird und die Medien nach Bedarf oder beispielsweise alle 50–100 Jahre durch neue ersetzt werden. Bei einer Auflösung von 200 Punkten pro Zoll (7.87 Punkte/mm) wie sie beim hochauflösenden Fax-Modus üblich ist und bei einem Kompressionsfaktor von 20 lassen sich über 25 000 A4-Seiten auf einer Platte mit 650 Mbyte Kapazität unterbringen. Somit wären für die geschätzten 60 Millionen vom Verfall bedrohten Bücher in deutschen wissenschaftlichen Bibliotheken, wenn diese im Mittel etwa 200 A4-Seiten umfassen etwa 500 000 Compact Disks notwendig, welche sogar in einem einzigen Gebäude lagerbar sind. Aneinandergereiht wären diese CDs mit einer Breite von 1 cm auf einem 5 km langen Regal lagerbar. Dabei ist noch nicht einmal berücksichtigt, daß unter den oben erwähnten 60 Millionen Büchern sehr viele Werke mehrfach aufgeführt wurden. Insgesamt erscheint also die digitalisierte zentrale Langzeitarchivierung von Literatur in absehbarer Zeit dank Datenkompression für digitalisierte Dokumente durchaus praktikabel, zumal mit einer weiteren Verbesserung der Massenspeichertechnik zu rechnen ist.

Im Rahmen dieser Arbeit wurde der standardisierte JBIG Kompressionsalgorithmus für digitalisierte Dokumente implementiert [ITU93a]. Es handelt sich dabei um ein sehr leistungsfähiges Verfahren, das neben einem für verlustfreie Algorithmen fast dem heutigen Stand der Technik entsprechenden Kompressionsverhältnis auch über für verschiedene für Anwendungen wichtige Eigenschaften verfügt:

- JBIG wurde dafür ausgelegt, nicht nur in Software, sondern auch direkt in VLSI-Chips implementierbar zu sein.
- Es können im sogenannten progressiven Modus mehrere aufeinander aufbauende Auf-

lösungsstufen eines Bilds getrennt abgespeichert werden, ohne daß das anfallende Datenvolumen wesentlich größer wird als wenn nur die höchste Auflösungsstufe alleine codiert wird. Dadurch ist es möglich, aus Datenbanken bei der Suche nach einem bestimmten Dokument zunächst nur auf schnell übertragbare geringe Auflösungsstufen zuzugreifen, die für die Darstellung auf Bildschirmen ausreichen. Erst wenn die gewünschten Texte identifiziert sind, wird zur hochauflösenden Wiedergabe auf Papier die dazu zusätzlich benötigte Information übertragen. Die progressive Codierung eignet sich auch dazu, zur raschen Suche nach Dokumenten die niedrigauflösende Version auf schnellen aber teuren Magnetplattenspeichern zu halten, während die volle Auflösung nur bei Bedarf von einem wesentlich billigeren aber auch langsameren Magnetband- oder WORM-Archivsystem geholt wird.

- Ein spezieller Algorithmus zur Auflösungsreduktion stellt sicher, daß in den niedrigeren Auflösungsstufen Linien und durch Rasterung (engl. *dithering*) entstandene Grauschattierungen möglichst gut erhalten bleiben.
- Die Anordnung der Daten im JBIG-Datenstrom kann so ausgewählt werden, daß sich entweder der Encoder oder der Decoder mit besonders wenig Speicherplatzbedarf implementieren läßt.
- Es ist möglich, mit der Codierung zu beginnen, wenn die Höhe des Bilds noch nicht bekannt ist, was für den Einsatz in Faxgeräten wichtig ist, die schon während des Papiereinzugs mit der Datenübertragung anfangen.
- Auch wenn das JBIG-Verfahren in erster Linie für die Codierung von bi-level Bildern geschaffen wurde, so lassen sich durch den Einsatz mehrerer einzelner Bitebenen auch Graustufen und farbige Darstellungen im gleichen Datenstrom repräsentieren.

Aufgrund der Flexibilität und Leistungsfähigkeit des JBIG-Algorithmus handelt es sich bei seiner Implementation um ein anspruchsvolles und komplexes Softwaremodul, insbesondere was die Optimierung und die nicht ganz einfache Fehlersuche betrifft. Anwendungsprogrammierer von Dokumentenverwaltungssystemen sollten sich nicht in die Details eines aufwendigen Datenkompressionsverfahrens einarbeiten müssen. Daher wurde in dieser Arbeit ein wiederverwendbares Softwaremodul entwickelt. Dieses kann leicht in andere Systeme integriert werden kann, ohne daß die Entwickler sich in die Interna des JBIG-Verfahrens einarbeiten müssen. Frei verfügbare, einfach handhabbare und gut dokumentierte Softwaremodule sind erfahrungsgemäß eine gute Voraussetzung dafür, daß sich ein formaler Standard wie JBIG auch als de-facto Standard bei Entwicklern und Benutzern durchsetzen kann. Einige Autoren von frei auf dem Internet erhältlichen Bildverarbeitungssystemen haben bereits mit geringem Aufwand dieses JBIG-Modul in ihre Software integriert. Damit besteht die Hoffnung, daß diese Arbeit einen kleinen Beitrag zur vereinfachten Handhabung von bi-level Grafikdaten leisten wird und einen Anstoß zur Entwicklung einer derzeit noch kaum existierenden frei verfügbaren Infrastruktur für die Handhabung digitalisierter Dokumente gibt.

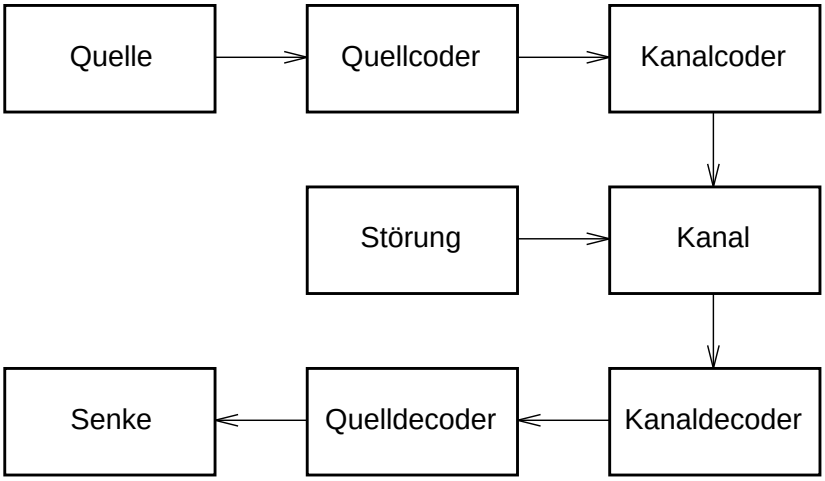
In den folgenden Kapiteln werden zunächst einige für die Datenkompression wichtige Grundlagen der Informationstheorie erläutert. Anschließend wird ein Überblick über verschiedene gängige Grundverfahren der Datenkompression mit besonderem Schwerpunkt auf der bei JBIG eingesetzten und leider noch recht unbekannten arithmetischen Codierung gegeben. Es folgt eine Beschreibung des JBIG-Algorithmus und seiner Implementierung sowie eine kurze Auswertung seiner Leistungsfähigkeit an einer Reihe von Beispielseiten im Vergleich zu gängigen Alternativverfahren.

2 Grundlagen und Verfahren der Datenkompression

2.1 Informationstheorie

Die von SHANNON in [Sha48] begründete Informations- und Kommunikationstheorie beschreibt mathematisch den Informationsgehalt von Nachrichten. Damit liefert sie die theoretischen Grundlagen für Datenkompressionsverfahren, da diese eine Codierung einsetzen müssen, die auf den erwarteten Informationsgehalt einer Nachrichtenquelle hin optimiert ist.

Kommunikationssysteme lassen sich in der Regel in die folgenden einzelnen Bestandteile untergliedern:



Aufgabe des Quellcoders ist es dabei, die Nachrichten aus ihrer ursprünglichen Form in eine für die Übertragung besser geeignete Form umzuwandeln, also beispielsweise eine Digitalisierung und anschließende Datenkompression durchzuführen. Der Kanalcoder paßt dann die vom Quellcoder umcodierten Nachrichten den Eigenschaften des Kanals an, fügt beispielsweise Fehlerkorrekturinformation hinzu wenn der Kanal Störungen aufweisen kann oder wendet eine Modulation an, um die Daten über eine bandbreitenbeschränkte Telefon- oder Funkstrecke übertragen zu können. Kanaldecoder und Quelldecoder wenden die entsprechenden komplementären Verfahren an, etwa eine Demodulation und Fehlerkorrektur im Kanaldecoder und eine Dekompression im Quelldecoder, um die relevante zu übertragende Information aus den Nachrichten der Quelle zu rekonstruieren.

Die Quelle, der Kanal und die Senke lassen sich dabei je nachdem, ob die Betrachtung für ein analoges oder ein digitales Nachrichtensystem durchgeführt wird auf zwei verschiedene Arten formalisieren. Bei der analogen Beschreibung wird eine Quelle als ein- oder mehrdimensionale Funktion $f(t)$ über die Zeit t dargestellt. Ein Beispiel dafür wäre der skalare Spannungsverlauf am Ausgang eines Mikrophonverstärkers oder das dreidimensionale Rot-Grün-Blau-Farbsignal einer analogen Fernsehkamera. Betrachtungen an derartigen analogen Kommunikationssystemen werden beispielsweise nach einer Fouriertransformation des Signals in den Frequenzbereich durchgeführt, wo Bandbreiten, lineare und nicht-lineare Kanaleigenschaften, Signal/Rausch-Abstand und dergleichen untersucht werden können.

Im folgenden werden wir uns dagegen mit der digitalen Variante der Informationstheorie beschäftigen [Arb63, Sha48, Top74]. In dieser wird die Nachrichtenquelle eines Kommunikationssystems beschrieben als der Ausgangspunkt einer Folge von Symbolen, die aus einer endlichen Symbolmenge $A = \{a_1, a_2, a_3, \ldots, a_n\}$ stammen, dem *Alphabet* der Quelle. Diese Symbolmenge können beispielsweise die Buchstaben eines Textes sein, diskrete Meßwerte eines Sensors, Zustände eines Automaten oder wie beim JBIG-Algorithmus die beiden für einen Bildpunkt möglichen Werte *schwarz* und *weiß*. Wie in der Wahrscheinlichkeitsrechnung kann die Quelle als ein Versuch angesehen werden, dessen mögliche Ausgänge jeweils durch ein Symbol des Alphabets repräsentiert werden. Die Informationstheorie beschreibt den Informationsgehalt der Symbole ausschließlich anhand der Wahrscheinlichkeiten, mit denen diese auftreten und nicht anhand deren Bedeutung in einem System. Eine Quelle ist also vollständig charakterisiert durch die bedingten Wahrscheinlichkeiten $p(a_i|w)$ für alle $1 \leq i \leq n$, wobei $a_i \in A$ ein Symbol ist und $w \in A^*$ die Folge aller vor a_i ausgegebenen Symbole der Quelle.

Zunächst betrachten wir Quellen, die Symbole unabhängig von vorangegangenen Symbolen ausgeben, bei denen also $p(a_i|w) = p(a_i|v) = p(a_i)$ für alle $1 \leq i \leq |A|$ und $w, v \in A^*$ gilt. Damit Information quantifiziert werden kann, muß eine Art Meßverfahren dafür gefunden werden. Information kann beispielsweise als Antwort auf eine Frage dargestellt werden. Um dem Problem zu entgehen, den Informationsgehalt einer Antwort bestimmen zu müssen, beschränken wir uns einfach auf binäre Fragen, die ausschließlich mit den beiden Alternativen *ja* und *nein* beantwortet werden können. Die Nachrichtenübertragung erfolgt im folgenden Gedankenexperiment dadurch, daß eine Person Q die zu übertragenden Symbole kennt und diese einer zweiten Person S mitteilen will. Beide Personen kennen bereits das endliche Alphabet A der Symbole aus denen Nachrichten zusammengesetzt sind, sowie die Verteilungsfunktion $p : A \to [0, 1]$. S stellt nun so lange binäre Fragen an Q, bis für S kein Zweifel mehr darüber besteht, um welche Nachricht es sich handelt. Der Informationsgehalt eines einzelnen Symbols $a \in A$ kann nun daran bemessen werden, wieviele binäre Fragen S stellen muß, um a identifizieren zu können. Offensichtlich kann S sich beim Fragenstellen beliebig ungeschickt anstellen, weshalb für die Informationstheorie in erster Linie der *ideelle Informationsgehalt* $I(a)$ eines Symbols a von Interesse ist, die untere Grenze der Fragenanzahl, die zur Übertragung dieses Symbols notwendig ist. Eine wichtige charakteristische Größe der Nachrichtenquelle Q ist der als *ideelle Entropie* bezeichnete Erwartungswert

$$H(Q) = \sum_{a \in A} p(a)\, I(a)$$

der Anzahl der pro Symbol notwendigen Fragen. Die Entropie ist ein Maß für die beim Empfänger herrschende Ungewißheit darüber, was als nächstes Symbol von der Quelle zu erwarten ist.

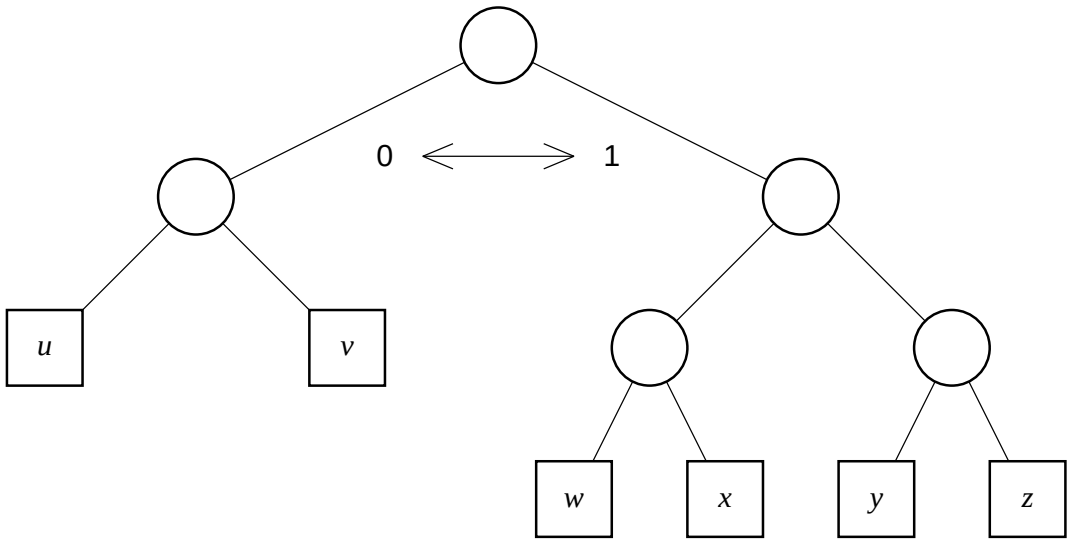
Dieses Frage-Antwort-Spiel läßt sich direkt in die Kommunikationstechnik übernehmen. Aufgrund der bekannten statistischen Eigenschaften der Nachrichtenquelle kann sich S bereits vor Beginn der Kommunikation ein möglichst geschicktes Frage-System ausdenken, das abhängig von den Antworten auf die vorangegangenen Fragen entweder die nächste Frage festlegt oder erkennt welches Symbol gemeint war. Dieses Schema kann auch Q bekannt gemacht werden, so daß Q bereits vorab alle Fragen von S und seine Antworten darauf durchspielen kann. Nun muß Q nur noch für jedes *nein* als Antwort ein Bit 0 und für jedes *ja* ein Bit 1 über einen binären Kanal an S senden und dieser kann anschließend selbst das Frage-Antwort-Spiel durchgehen und daraus das fragliche Symbol ermitteln. So

kann das Frage-Antwort-Spiel direkt in ein digitales Kommunikationsverfahren umgewandelt werden, bei dem keine Fragen mehr übertragen werden. Der mit dem obigen Gedankenexperiment ermittelte Informationsgehalt von Symbolen und die Entropie einer Quelle beschreiben also direkt die minimale Anzahl von Bits, die ein optimaler Datenkompressionsalgorithmus theoretisch nur zu übertragen bräuchte.

Wenn beispielsweise eine Nachrichtenquelle Q_1 mit dem Alphabet $A_{Q_1} = \{u, v, w, x, y, z\}$ und den Auftretenswahrscheinlichkeiten

$$\begin{aligned} p(u) &= 0.35 & p(v) &= 0.2 \\ p(w) &= 0.2 & p(x) &= 0.15 \\ p(y) &= 0.05 & p(z) &= 0.05 \end{aligned}$$

vorhanden ist, so wäre ein möglicher Frage-Algorithmus durch den folgenden Baum gegeben:



Dabei repräsentiert jeder Kreis die Frage, ob sich das zu übertragende Symbol im linken oder im rechten Teilbaum befindet. Die Symbole u, v, w, x, y und z werden folglich durch die binären Antworten 00, 01, 100, 101, 110 sowie 111 repräsentiert. Mit einer Wahrscheinlichkeit von 0.55 werden bei dieser Codierung 2 Bit benötigt und mit einer Wahrscheinlichkeit von 0.45 3 Bit, wodurch im Mittel 2.45 Bit pro Symbol übertragen werden.

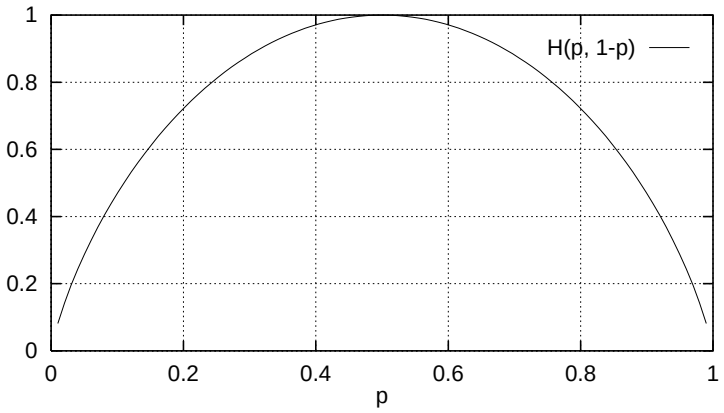
Gemäß dem SHANNON'schen *ersten Hauptsatz der Informationstheorie* beträgt die ideelle Entropie einer Nachrichtenquelle Q mit dem Alphabet A und der Verteilungsfunktion p

$$H(Q) = -\sum_{a \in A} p(a) \log_2 p(a).$$

Damit läßt sich der ideelle Informationsgehalt eines einzelnen Symbols a mit $I(a) = -\log_2 p(a)$ angeben. Für den technisch etwas aufwendigen Beweis wird hier auf [Top74] verwiesen.

Die ideelle Entropie $H(p, 1-p) = -p \log_2 p - (1-p) \log_2 (1-p)$ einer für die Kompression von bi-level Bilddaten besonders relevanten binären Quelle, die mit den Wahrscheinlichkeiten

p und $1-p$ die einzigen beiden möglichen Symbole ausgibt, sieht in Abhängigkeit von p wie folgt aus:



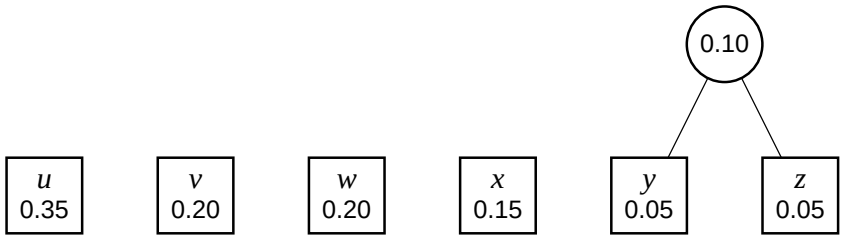
Bei $p = 1 - p = 0.5$ ist die Entropie maximal, da in diesem Fall die Ungewißheit des Empfängers am größten ist. Prinzipiell ist die Entropie einer Quelle mit gegebenem Alphabet immer dann maximal, wenn alle Symbole mit der gleichen Wahrscheinlichkeit auftauchen. Wenn p in der obigen binären Quelle nahe 1 oder 0 liegt, dann weiß der Empfänger bereits *a priori* mit hoher Wahrscheinlichkeit, welches Symbol als nächstes übertragen wird. Er gewinnt dadurch pro übertragenem Symbol weniger Information, da sich die vor der Übertragung auf der Seite des Empfängers herrschende (Un-)gewißheit meistens kaum ändert. In einem Codierverfahren werden entsprechend für die häufigeren Symbole möglichst wenig Bits eingesetzt, dafür für die selteneren umso mehr.

Auf die obige Beispielnachrichtenquelle Q_1 angewendet ergibt der erste Hauptsatz eine ideelle Entropie $H(Q_1) \approx 2.30$ Bit pro Symbol. Der als Beispiel angegebene Code mit 2.45 Bit pro Symbol ist also noch deutlich von der theoretischen Grenze entfernt. Im folgenden werden zwei Verfahren beschrieben, mit denen sich Codierresultate erzielen lassen, die beliebig dicht an der ideellen Entropie liegen.

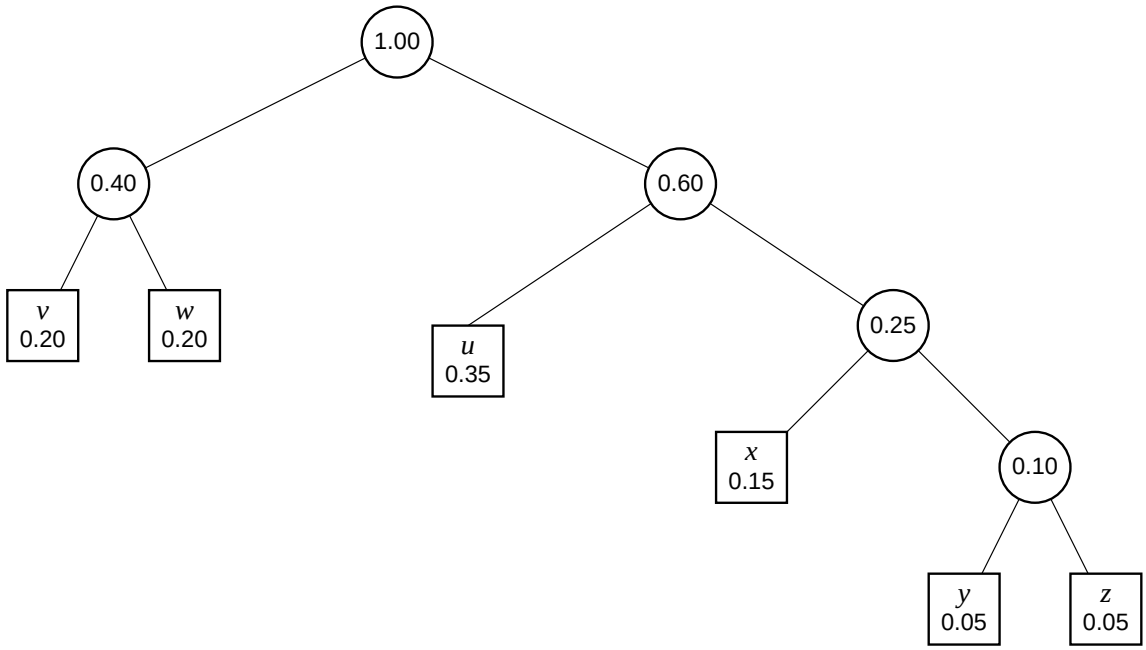
2.2 Codieralgorithmus nach Huffman

Der oben angegebene Codierbaum für die Quelle Q_1 ist nur einer von endlich vielen möglichen Codes. Der Algorithmus von Huffman ist ein Verfahren, mit dem effizient der optimale Codierbaum zu einem Alphabet mit gegebener Verteilungsfunktion konstruiert werden kann. Der Algorithmus iteriert auf einer Menge von Bäumen. Zu Beginn besteht diese Menge aus lauter trivialen Bäumen die alle lediglich aus nur einem einzigen Blatt-Knoten bestehen. Jeder dieser trivialen Bäume repräsentiert ein Symbol des Alphabets. Jedem Baum ist als Wahrscheinlichkeit die Summe der in seinen Blättern enthaltenen Symbole zugeordnet. Es werden nun jeweils die beiden Bäume mit der geringsten Wahrscheinlichkeit aus der Menge entfernt. Diese werden zu einem Baum bestehend aus einem neuen Wurzelknoten und den beiden eben entfernten Bäumen als Teilbäume vereint. Dieser neue Baum wird nun wieder zur Menge der Bäume hinzugefügt und seine Wahrscheinlichkeit ist die Summe der Wahrscheinlichkeiten der beiden Teilbäume, aus denen er entstanden ist. Dieser Baumvereinigungsschritt wird so lange wiederholt, bis die Menge der Bäume nur noch aus einem einzigen Baum besteht, der dann die Wahrscheinlichkeit 1 trägt.

Als Beispiel soll ein Huffman-Code für die Quelle Q_1 konstruiert werden. Die beiden Symbole mit der geringsten Wahrscheinlichkeit sind y und z, also werden sie im ersten Schritt zu einem neuen Baum vereinigt:



Dieser yz Baum ist nun zusammen mit x der kleinste Baum in der Gruppe, also werden beide zusammengefügt. Anschließend werden v und w vereinigt und nach zwei weiteren Schritten ist der komplette Huffman-Baum konstruiert:



Drei Symbole haben einen zwei Bit langen Code, ein Symbol wird mit drei Bits repräsentiert und die beiden recht seltenen Symbole y und z sogar mit vier. Daraus ergibt sich als Erwartungswert 2.35 Bit pro Symbol, was immerhin 0.1 Bit Ersparnis gegenüber dem ersten Beispielcode darstellt. Da es sich hierbei bereits um die optimale Codierung handelt (mit der Einschränkung daß jedem Symbol eine ganze Anzahl von Bits zugewiesen wird) hat die Quelle Q_1 die *reale Entropie* $H_0(Q_1) = 2.35$ Bit pro Symbol.

Der Huffman-Algorithmus weist jedem Symbol a eine ganze Zahl $I_0(a)$ von Bits zu. Die Zahl der Bits $I_0(a)$ entspricht aber nur dem ideellen Informationsgehalt $I(a)$, wenn $p(a) = 2^{-I_0(a)}$ ist. Nur für Quellen, bei denen alle Symbole mit Wahrscheinlichkeiten der Form 2^{-n} ($n \in \mathbb{N}$) auftreten, kann ein Huffman-Code eine Codierung erzielen, die der ideellen Entropie der Quelle entspricht.

Wenn aber, wie etwa bei der Kompression von digitalisierten bi-level Bildern mit den beiden Symbolen *schwarz* und *weiß*, eine binäre Quelle vorliegt oder sonst eine Quelle bei der

ein Symbol eine Wahrscheinlichkeit von über 0.5 hat und daher einen ideellen Informationsgehalt von weniger als einem Bit pro Symbol, dann versagt der Huffman-Algorithmus, da dieser immer mindestens ein Bit pro Symbol benötigt. Ein Ausweg ist die Blockcodierung. Dabei werden die zu codierenden Symbole einer Quelle Q in Wörtern zu jeweils k Symbolen zusammengefaßt. Die Menge A^k der möglichen Wörter wird als Alphabet einer neuen Quelle Q^k aufgefaßt und es wird ein Huffman-Code dafür erstellt. Bei der Blockcodierung wird nicht mehr einem einzelnen Symbol eine eigene Bitsequenz zugewiesen, sondern jeweils einem Wort aus k Symbolen. Wenn die einzelnen Symbole wie vorausgesetzt voneinander stochastisch unabhängig auftreten, dann gilt für die Auftretenswahrscheinlichkeit eines Worts $a_{i_1}a_{i_2}\cdots a_{i_k} \in A^k$:

$$p(a_{i_1}a_{i_2}\cdots a_{i_k}) = \prod_{j=1}^{k} p(a_{i_j}).$$

Die Auftretenswahrscheinlichkeiten von Wörtern sind geringer als die einzelner Symbole und der entsprechende ideelle Informationsgehalt jeweils höher. Dadurch fällt das Problem des Huffman-Verfahrens, daß lediglich eine ganze Zahl von Bits pro Symbol verwendet werden kann, mit steigendem k weniger ins Gewicht und zusätzlich verteilt sich die Differenz zwischen realem und ideellem Informationsgehalt auf k Symbole. Da beispielsweise ein Wort aus k Symbolen immer noch eine Auftretenswahrscheinlichkeit über 0.5 haben kann, ist es denkbar, daß ein ganzes Wort vom Huffman-Blockcodier-Verfahren durch ein einziges Bit repräsentiert wird, wodurch dann pro Symbol in diesem Wort nur $1/k$ Bit anfallen.

Eine weitere Aussage des 1. Hauptsatzes der Informationstheorie ist, daß mit dem Huffman-Blockcodier-Verfahren für $k \to \infty$ die ideelle Entropie der Quelle Q als Codiereffizienz erreicht werden kann [Top84]:

$$H(Q) = \lim_{k\to\infty} \frac{1}{k} H_0(Q^k).$$

Da jedoch mit wachsendem k die Größe des zu bearbeitenden Codierbaums exponentiell ansteigt (der Baum hat $|A|^k$ Blätter und $2|A|^k - 1$ Knoten) wird das Huffman-Blockcodier-Verfahren praktisch kaum eingesetzt und diente in erster Linie der theoretischen Begründung der ideellen Entropie, als die im folgenden vorgestellte arithmetische Codierung noch nicht bekannt war.

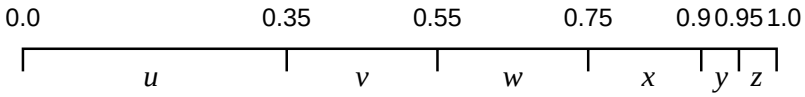
2.3 Arithmetische Codierung

Die *arithmetische Codierung* ist wie der Huffman-Blockcoder ebenfalls ein Verfahren, das Symbolsequenzen binär codiert, ohne dabei wesentlich mehr Bits zu benötigen als die ideelle Entropie der Quelle vorgibt. Im Gegensatz zum wesentlich bekannteren Huffman-Algorithmus wird nicht jedem Symbol eine feste Bitfolge zugeordnet, sondern aus der komplett zu codierenden Symbolfolge wird eine reelle Zahl im Intervall $[0, 1)$ konstruiert, die in binärer Darstellung dem komprimierten Datenstrom entspricht – daher die Bezeichnung *arithmetische* Codierung.

Der Codiervorgang läuft in Form einer Intervallschachtelung ab, d.h. mit jedem weiteren Symbol a wird ein Codierintervall im Bereich $[0, 1)$ um den Faktor $p(a)$ verkleinert und

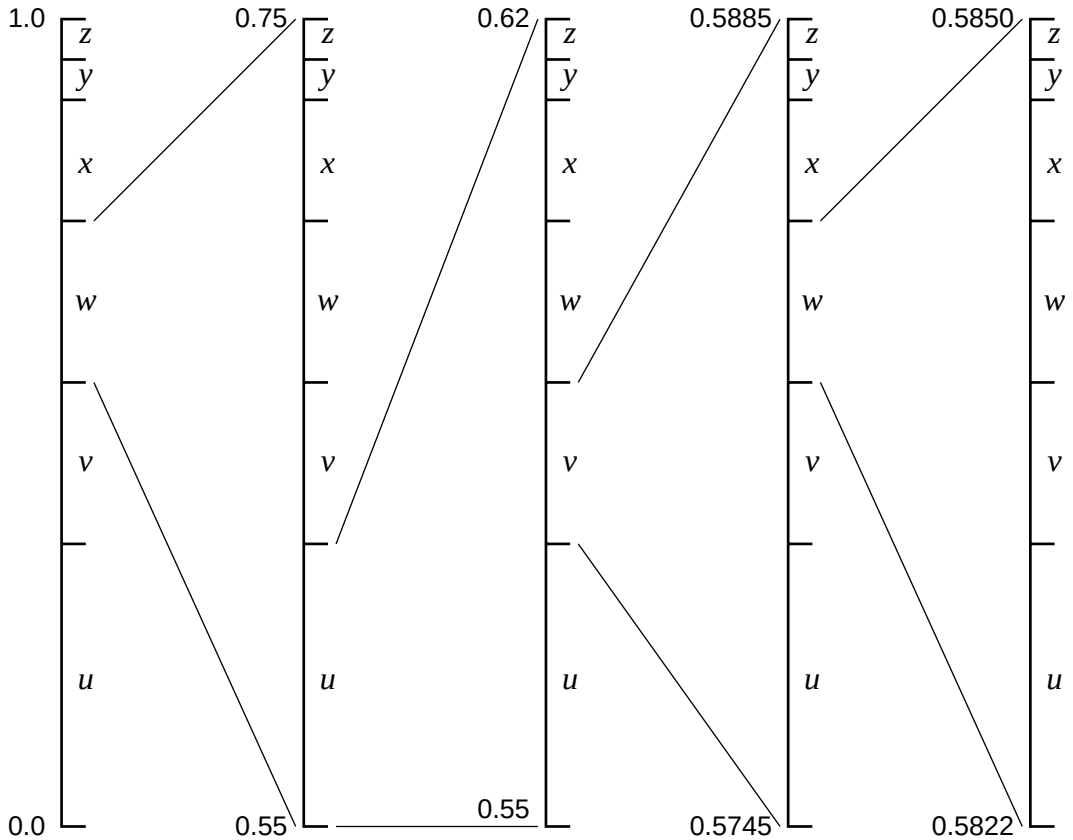
die am Ende ausgegebene Binärzahl liegt in diesem Intervall. Die in der arithmetischen Codierung oft auch als *Modell* bezeichnete Wahrscheinlichkeitsverteilung p der Symbole des Alphabets beeinflußt direkt den Codiervorgang. Daher kann während der Codierung einer längeren Symbolfolge mühelos das Modell dynamisch angepaßt werden, ohne daß wie bei der Huffman-Codierung erst ein Codebaum neu konstruiert werden muß.

Im einzelnen läuft die arithmetische Codierung prinzipiell wie folgt ab: Das aktuelle Codierintervall I wird auf den Bereich $I := [0, 1)$ initialisiert. Nun wird dieses Intervall in n Teilintervalle zerlegt, eines pro Symbol a_i aus dem Alphabet $A = \{a_1, a_2, \ldots, a_n\}$. Die Länge jedes Teilintervalls ist dabei die Auftretenswahrscheinlichkeit $p(a_i)$ multipliziert mit der Größe des aktuellen Codierintervalls. Bei der schon im vorangegangenen Abschnitt verwendeten Beispielquelle Q_1 würde das Intervall I wie folgt aufgeteilt werden:



Nach der Aufteilung in Teilintervalle wird das aktuelle Codierintervall durch das dem nächsten zu codierenden Symbol a_i entsprechende Teilintervall ersetzt. Anschließend wird dieses neue Intervall wieder unterteilt und dieser Vorgang wird für alle folgenden Symbole wiederholt bis kein Symbol mehr zu codieren ist. Nun wird die kürzeste Binärzahl im Intervall $[0, 1)$ gesucht, die im Codierintervall liegt und deren Nachkommastellen werden als Codierergebnis ausgegeben.

Die Codierung der Symbolfolge $wuvw$ aus der Quelle Q_1 läuft beispielsweise wie folgt ab:



Nach den vier codierten Symbolen ist das Codierintervall auf die Länge $|[0.5822, 0.5850)| = 0.0028 = p(w)\,p(u)\,p(v)\,p(w)$ geschrumpft. Zur besseren Anschauung sind im obigen Beispiel die Intervallgrenzen als Dezimalzahlen angegeben. In Binärdarstellung lauten sie

$$0.1001010111000_2$$
$$0.1001010100001_2$$

Die kürzeste Binärzahl in diesem Intervall ist $0.100101011_2 \approx 0.58398$, weshalb die Ausgabe des arithmetischen Coders für die Symbolfolge $wuvw$ die Bitfolge 100101011 ist. Die benötigten 9 Bit liegen nur knapp über dem ideellen Informationsgehalt

$$I(wuvw) = I(w) + I(u) + I(v) + I(w) = log_2 \frac{1}{p(wuvw)} = -log_2 0.0028 \approx 8.48 \text{ Bit}$$

der Nachricht.

Ein idealisierter arithmetischer Encoder hat zwei Festkomma-Variablen l und h mit beliebig steigerbarer Genauigkeit, die die untere und obere Grenze des aktuellen Codierintervalls festlegen. Nach Codierung des nächsten Symbols a_i der Nachricht aus dem Alphabet $A = \{a_1, a_2, \ldots, a_n\}$ hat sich das Codierintervall auf die neuen Grenzen l' und h' mit

$$l' = l + (h - l) \sum_{j=1}^{i-1} p(a_j)$$
$$h' = l + (h - l) \sum_{j=1}^{i} p(a_j) = l' + (h - l)\, p(a_i)$$

verkleinert. Das neue Codierintervall ist dabei um den Faktor $p(a_i)$ kleiner als das vorangegangene. Nach Codierung des letzten Symbols wird l solange aufgerundet, solange noch $l < h$ gilt und die Nachkommastellen von l werden als Codierergebnis ausgegeben.

Der idealisierte arithmetische Decoder liest zunächst die komplette Codesequenz ein und legt sie in der Festkomma-Variable x ab. Anschließend wird wie im Encoder das Codierintervall auf $l = 0$ und $h = 1$ initialisiert und der Übergang auf das jeweils nächst kleinere Codierintervall wird identisch wie im Encoder vorgenommen. Während jedoch im Encoder das bekannte nächste Symbol a_i das nächste Codierintervall aus den n Teilintervallen der Zerlegung auswählt, bestimmt im Decoder die eingelesene Zahl x, welches Teilintervall das nachfolgende Codierintervall ist. Das ausgewählte Teilintervall bestimmt dabei gleichzeitig, welches decodierte Symbol an den Ausgang des Decoders gegeben wird. Im Decoder wird bei einem aktuellen Codierintervall $[l, h)$ dasjenige i gesucht, für welches

$$l + (h - l) \sum_{j=1}^{i-1} p(a_j) \le x < l + (h - l) \sum_{j=1}^{i} p(a_j)$$

gilt. Dann wird das Symbol a_i ausgegeben und das neue Codierintervall lautet wie schon im Encoder $[l', h')$.

Diese Grundidee der arithmetischen Codierung geht auf Elias zurück und wurde erstmals in [Abr63, pp. 61–62] am Rande erwähnt. Um einzusehen, daß die idealisierte arithmetische

Codierung tatsächlich optimal ist in dem Sinn, daß für so codierte Nachrichten nicht mehr Bits, als vom ideellen Informationsgehalt der Nachricht vorgegeben, benötigt werden, ist der folgende Satz maßgeblich:

Satz: Ein Teilintervall $I = [a, b)$ des halboffenen Intervalls $[0, 1) \subset \mathbb{R}$ enthält stets eine Zahl x, die sich mit

$$d = \lceil -\log_2 |I| \rceil$$

binären Nachkommastellen repräsentieren läßt, d.h. für alle Intervallgrenzen $0 \leq a < b < 1$ existieren zwei Zahlen $c, d \in \mathbb{N}$, so daß $a \leq x = c\,2^{-d} < b$ und $d - 1 < -\log_2(b - a) \leq d$.

Beweis: Bezeichne $S_d = \{c\,2^{-d} | c \in \mathbb{N} \wedge 0 \leq c < 2^d\}$ (für $d \in \mathbb{N}$) die Menge aller Zahlen im Intervall [0,1), die sich mit d binären Nachkommastellen darstellen lassen. Zu jeder Zahl $b \in (0, 1)$ und jedem $d \in \mathbb{N}$ existiert genau eine Zahl $s \in S_d$ mit $s < b$ und $b - s \leq 2^{-d}$, nämlich $s = \lfloor b\,2^d \rfloor / 2^d$. Also kann es in $[0, 1)$ kein Teilintervall $[a, b)$ mit $b - a \geq 2^{-d}$ geben welches kein Element aus S_d enthält, denn für $s = \lfloor b\,2^d \rfloor / 2^d$ folgt wegen $b - s \leq 2^{-d}$ und $b - a \geq 2^{-d}$ sofort $a \leq s < b$. Also enthält jedes Intervall $I \subseteq [0, 1)$ ein Element aus S_d mit $|I| \geq 2^{-d}$ was äquivalent ist zu $d \geq -\log_2 |I|$. Das kleinste $d \in \mathbb{N}$ für das S_d mit Sicherheit ein Element aus I enthält, also die geringste Anzahl von binären Nachkommastellen, mit denen sicher ein Element aus I dargestellt werden kann ist folglich $d = \lceil -\log_2 |I| \rceil$, was zu beweisen war.

Eine Nachricht bestehend aus den Symbolen $a_{i_1} a_{i_2} \cdots a_{i_m}$ führt dazu, daß der idealisierte arithmetische Encoder am Ende ein Codierintervall der Länge

$$h - l = \prod_{j=1}^{m} p(a_{i_j})$$

vorliegen hat, womit laut obigem Satz nicht mehr als

$$d = \left\lceil -\log_2 \prod_{j=1}^{m} p(a_{i_j}) \right\rceil$$

Bits ausgegeben werden müssen um dem Decoder eine Zahl x in diesem Codierintervall zu liefern. Die ideelle Entropie der Nachricht beträgt, da per Voraussetzung die Symbole stochastisch unabhängig voneinander auftreten,

$$I(a_{i_1} a_{i_2} \cdots a_{i_m}) = -\log_2 p(a_{i_1} a_{i_2} \cdots a_{i_m}) = -\log_2 \prod_{j=1}^{m} p(a_{i_j}).$$

Also ist die maximale Länge d, die der idealisierte arithmetische Encoder für eine gegebene Nachricht $a_{i_1} a_{i_2} \cdots a_{i_m}$ benötigt, lediglich der auf die nächste ganze Zahl aufgerundete ideelle Informationsgehalt $\lceil I(a_{i_1} a_{i_2} \cdots a_{i_m}) \rceil$ dieser Nachricht. Folglich ist die arithmetische Codierung optimal im Sinne der Informationstheorie. Durch das Weglassen von Nullen am Ende des Codierergebnisses sind in manchen Fällen sogar noch weniger Bits zu übertragen. Da der Decoder keine Möglichkeit hat, festzustellen, wann genau die übertragene Nachricht zuende ist, muß die Länge der Nachricht auf andere Weise dem Decoder mitgeteilt werden. Dies kann entweder durch eine bereits im voraus übertragene Längenangabe für die Nachricht geschehen oder durch ein eigenes Nachrichtenende-Symbol nach dem letzten regulären Symbol.

Trotz der prinzipiellen Einfachheit und Eleganz der arithmetischen Codierung wurde sie über 20 Jahre hinweg im Gegensatz zum Huffman-Algorithmus praktisch nicht eingesetzt. Dies liegt in erster Linie daran, daß die oben dargestellte idealisierte arithmetische Codierung für die direkte Implementation sehr ungeeignet ist. Die dazu benötigten beliebig genauen Festkomma-Variablen und insbesondere die Multiplikationen mit diesen würden eine derartige Implementation sehr ineffizient machen, würden viel Speicherplatz benötigen und keine kontinuierliche Übertragung schon während der Codierung erlauben.

Erst als einige optimierte Abwandlungen der idealisierten arithmetischen Codierung erdacht wurden, gewann das Verfahren an Attraktivität für die praktische Implementation. Die erste Vereinfachung ist, daß die Variablen l und h nur mit endlicher Genauigkeit gespeichert werden. Dazu ist es erforderlich, die Länge des Codierintervalls immer wieder zu renormalisieren, also auf eine Länge von etwa 1.0 zu bringen. Im Zuge der Renormalisierung können dann bereits feststehende führende Ziffern der Intervallgrenzen ausgegeben werden.

Bei der in [Wit87] vorgestellten Implementation etwa werden die Grenzen des Codierintervalls wie folgt renormalisiert, sobald die Länge des Codierintervalls unter 0.5 gefallen ist:

a) Wenn das Codierintervall $[l, h)$ vollständig in der unteren Hälfte $[0, 0.5)$ des Einheitsintervalls $[0, 1)$ liegt, dann wird die bei l und h gemeinsame führende erste Nachkomma-Null ausgegeben und die verbleibenden Bits rücken durch eine Multiplikation von l und h mit 2 nach. In diesem Fall ist das neue Codierintervall $[2l, 2h)$.

b) Wenn $[l, h)$ vollständig in der oberen Hälfte $[0.5, 1)$ des möglichen Bereichs liegt, dann wird entsprechend die bei l und h gemeinsame Eins nach dem Komma ausgegeben und die restlichen Bits rücken nach. Das neue Codierintervall ist dann $[2l - 1, 2h - 1)$.

c) Es verbleibt nun noch der etwas schwierigere Fall, daß die Intervallgrenzen l und h nach dem Komma sich in der ersten Binärziffer unterscheiden, daß also $l \in [0.25, 0.5)$ und $h \in [0.5, 0.75)$. In diesem Fall wird zunächst von l und h vor der Schiebeoperation 0.25 subtrahiert, so daß das Intervall anschließend wieder im Bereich $[0, 1)$ liegt. Das neue Codierintervall ist also $[2l-0.5, 2h-0.5)$, jedoch wird bei dieser Renormalisierung noch kein Bit ausgegeben. Erst das nächste auszugebende Bit wird entscheiden, ob der letztendlich zu erzeugte Wert x im aktuellen Intervall über oder unter 0.5 liegt, also ob bei dieser Renormalisierung eine Eins oder Null hätte ausgegeben werden müssen. Wenn bei der nächsten Renormalisierung Fall b) eintritt und eine 1 ausgegeben werden müßte, dann muß nun 10 ausgegeben werden, da sich dann herausgestellt hat, daß der zu codierende Wert x vor dieser Renormalisierung im Intervall $[0.5, 0.75)$ lag. Wenn dagegen bei der nächsten Renormalisierung der Fall a) auftritt und eine 0 ausgegeben werden müßte, dann lag der Wert dieses Mal in $[0.25, 0.5)$ und es wird daher 01 ausgegeben. In der Implementation wird daher im Fall c) statt ein Bit auszugeben nur ein Zähler z um eins erhöht, um vorzumerken, daß noch ein weiteres noch nicht eindeutig feststehendes Bit hätte ausgegeben werden müssen. Wenn dann bei der nächsten Renormalisierung der Fall a) eintritt, dann wird eine Eins gefolgt von z Nullen, bzw. im Fall b) eine Null gefolgt von z Einsen ausgegeben und wieder $z = 0$ gesetzt.

Diese auf den ersten Blick etwas unübersichtliche Problematik der Überlaufbehandlung in den Variablen mit den Codierintervall-Grenzen ist sicher ein wesentlicher Grund dafür,

warum trotz ihrer Vorzüge die arithmetische Codierung im Vergleich zur Huffman-Codierung lange Zeit kaum implementiert wurde. Hinzu kommt, daß in den USA einige dieser Techniken patentiert wurden und auch daher Bedenken bestanden.

Eine weitere von RISSANEN vorgeschlagene Modifikation zur Effizienzsteigerung der arithmetischen Codierung ist der völlige Verzicht auf die Multiplikation. Wenn bei der Normalisierung die Länge des Codierintervalls immer im Bereich 0.75 bis 1.5 gehalten wird, dann hat das Codierintervall langfristig im Mittel etwa die Länge 1. Also kann darauf verzichtet werden, die Wahrscheinlichkeiten $p(a_i)$ der einzelnen Symbole stets mit der Länge des Intervalls zu multiplizieren. In der Regel ist die Verteilungsfunktion p ohnehin nur eine nicht allzu genaue Schätzung. Daher fällt in der Praxis die Abkehr von der idealisierten und informationstheoretisch optimalen arithmetischen Codierung nach ELIAS durch Festkomma-Variablen von beschränkter Genauigkeit (üblich sind 16 oder 32 Bit) und durch den Verzicht auf die Multiplikation mit der präzisen Länge des Intervalls kaum ins Gewicht und kann leicht durch den erheblichen Geschwindigkeitsgewinn gerechtfertigt werden. Da der Decoder lediglich durch das gleiche Verfahren genau die gleiche Sequenz von Codierintervallen reproduzieren muß, verursachen Vereinfachungen durch zugelassene Rundungsfehler und Näherungen auch keine Probleme auf der Seite des Decoders.

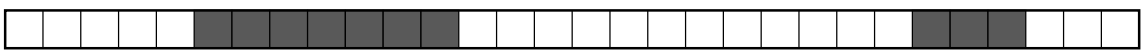
Die drei wesentlichen Vorteile eines derart optimierten arithmetischen Coders lassen sich zusammenfassen mit hoher Ausführungsgeschwindigkeit, hoher Codiereffizienz am Rande des informationstheoretisch möglichen bei Quellen mit stochastisch unabhängig auftauchenden Symbolen sowie eine leichte Anpassbarkeit des Quellmodells p während des Codiervorgangs an Änderungen der stochastischen Eigenschaften der Quelle.

2.4 Lauflängencodierung

Die bisher vorgestellten Verfahren nutzen lediglich die unbedingten unterschiedlichen Auftretenswahrscheinlichkeiten der einzelnen Symbole auf, berücksichtigen aber nicht, daß bei vielen Quellen die Wahrscheinlichkeit der Symbole sehr stark von den vorangegangenen Symbolen abhängt. Andere Verfahren berücksichtigen dagegen auch in gewissen Maße die bedingte Wahrscheinlichkeit $p(a_j|a_{i_1}a_{i_2}\cdots a_{i_m})$ die angibt, mit welcher Wahrscheinlichkeit das nächste Symbol a_j ist, unter der Voraussetzung, daß die m vorangegangenen Symbole $a_{i_1}a_{i_2}\cdots a_{i_m}$ waren.

Das einfachste dieser Verfahren ist die *Lauflängencodierung*. Bei ihr werden lange Ketten aus identischen Symbolen ersetzt durch ein neues Symbol, welches die Länge einer auch als *run* bezeichneten Kette ebenso repräsentiert wie die Symbolart, aus der die Kette besteht.

Bei hochauflösenden digitalisierten Dokumenten ist die Wahrscheinlichkeit, daß ein Bildpunkt die gleiche Farbe wie sein linker Nachbarpunkt hat sehr groß, es wird also häufig lange Ketten aus schwarzen oder weißen Bildpunkten geben. Die Bildzeile



kann daher leicht durch die folgende Symbolfolge wesentlich kompakter dargestellt werden:

Eine leicht modifizierte Form der Lauflängencodierung, die oft bei der einfachen kompakten Abspeicherung von Texten und Graphikdateien eingesetzt wird, erlaubt die Kennzeichnung von Ketten aus sich wiederholenden Symbolen ebenso wie die direkte und effiziente Übernahme von nicht aus zusammenhängenden Ketten identischer Symbole bestehender Bereiche. Dabei wird die zu komprimierende Datei in Blöcke unterschiedlicher aber begrenzter Länge unterteilt. Bei jedem dieser Blöcke handelt es sich entweder um einen Übernahme- oder Wiederholblock. Jeder Block beginnt mit einem Kopf, der ein Bit enthält, das zwischen Übernahme- und Wiederholblock unterscheidet, sowie die Länge n des Blocks. Nach dem Kopf eines Übernahmeblocks folgen n Symbole, die genau so aus der zu komprimierenden Datei übernommen wurden, nach dem Kopf eines Wiederholblocks folgt nur ein einzelnes Symbol, das n mal in der Ausgangsdatei auftauchte. Nach jedem Block folgt unmittelbar der Kopf des nächsten Blocks oder das Dateiende. Die Nachricht

$$ABCAAAAAAAAAABBBBBBBBBCDEFAAAAAA$$

würde dann etwa in der Form

$$\overset{\text{Ü}}{3}ABC\overset{\text{W}}{10}A\overset{\text{W}}{9}B\overset{\text{Ü}}{4}CDEF\overset{\text{W}}{6}A$$

abgelegt werden. Für die Blockköpfe muß die Symbolmenge nicht unbedingt erweitert werden, da durch die in den Blockköpfen festgelegten Blocklängen der Kopf des nachfolgenden Blocks immer eindeutig ermittelt werden kann. Wenn das zu codierende Alphabet beispielsweise 8 Bit lange Bytes sind, dann wäre eine denkbare Codierung für den Blockkopf ein einzelnes Byte, dessen erstes Bit die Unterscheidung zwischen Übernahme- und Wiederholblock erlaubt, während die restlichen 7 Bit eine Längenangabe im Bereich 1–128 erlauben. Im ungünstigsten Fall, also wenn die zu codierende Nachricht niemals mehrere aufeinanderfolgende identische Bytes aufweist, wird die Nachricht nur aus Übernahmeblöcken zusammengesetzt und verlängert sich durch die Blockköpfe nur um etwa den Faktor 129/128, also um etwa 0.78 %. Im besten Fall dagegen besteht eine lange Nachricht ausschließlich aus identischen Symbolen, was eine Kompression um den Faktor 2/128 zuläßt, also etwa auf 1.56 %.

2.5 Lempel-Ziv Algorithmen

Bei Textdateien, bei Software im Quell- und Maschinencode sowie bei vielen anderen Datenquellen hängt die Auftretenswahrscheinlichkeit einzelner Symbole (meistens Bytes) ebenfalls sehr stark von den vorangegangenen Symbolen ab. Dennoch kann eine Lauflängencodierung bei diesen Quellen nur selten eine deutliche Redundanzreduktion erzielen, da die starke stochastische Abhängigkeit der Symbole untereinander nicht durch viele lange Ketten identischer Symbole entsteht. Texte bestehen durch die Wortstruktur der Sprachen aus kurzen, sich oft wiederholenden Symbolfolgen, was ein Codieralgorithmus ausnutzen kann. Programmiersprachen gleichen in dieser Hinsicht den natürlichen Sprachen und auch Maschinencode besteht aus sich sehr häufig wiederholenden kurzen und längeren Byte-Sequenzen für bestimmte charakteristische Sprachkonstrukte wie Schleifen, einfache Fallunterscheidungen oder dem Sichern der Register vor Unterprogrammaufrufen.

Derartige Eigenschaften von Quellen nutzt ein als Lempel-Ziv- oder LZ77-Algorithmus bekanntes universelles Kompressionsverfahren aus [Ziv77]. Die vom Encoder eingelesenen bzw. vom Decoder wieder ausgegebenen Daten werden auf beiden Seiten in einem L Symbole umfassenden FIFO-Puffer zwischengespeichert, wobei L meist im Bereich von mehreren tausend Symbolen liegt. Damit haben sowohl der Encoder als auch der Decoder Zugriff auf die letzten L dem nächsten Symbol vorangegangenen Symbole. Wenn der Encoder erkennt, daß gerade neu ankommende Symbole eine Sequenz ergeben, die bereits vor kurzem schon einmal aufgetaucht ist und sich noch im Puffer befindet, so werden solange weiter Symbole eingelesen bis sich die ankommende Symbolkette von der sich bereits im Puffer befindlichen unterscheidet. Anschließend wird statt der neu eingelesenen, aber bereits bekannten Symbolfolge lediglich ein Verweis auf den Ort der sich noch vom letzten Auftreten her im Pufferspeicher befindlichen Symbolfolge sowie deren Länge codiert. Somit können Wortfragmente, Wörter oder gar ganze Satzteile, die irgendwo in den beispielsweise letzten 100 Zeilen einer Textdatei schon einmal aufgetaucht sind durch einen sehr kompakten Zeiger und eine Längenangabe ersetzt werden. Auch bei Graphikdateien die nur wenige verschiedene Farben enthalten und bei denen bereits eine Lauflängencodierung sehr erfolgreich war haben sich Lempel-Ziv-Algorithmen bewährt. Die annähernde Funktionalität einer Lauflängencodierung ist den Implementationen des Lempel-Ziv-Verfahrens automatisch mitgegeben, da nach einer kurzen Startphase bei Quellen, mit denen sich Lauflängencodierung lohnt, im Pufferspeicher genügend lange Ketten aus identischen Symbolen vorhanden sind, die fast ebenso effizient adressiert werden können wie eine Lauflängencodierung diese Ketten beschreiben könnte.

Der Lempel-Ziv-Algorithmus wurde in vielfältigen Details ergänzt und optimiert, insbesondere um die Suche nach den bereits vorhandenen Symbolketten effizienter zu gestalten. Sehr verbreitet ist beispielsweise die Lempel-Ziv-Welch-Variante (LZW) des Verfahrens [Wel84], bei der statt eines FIFO-Pufferspeichers die angekommenen Symbole einen Baum aufbauen, dessen Wege von der Wurzel zu den jeweiligen Knoten die wiederverwendbaren Symbolketten repräsentieren. Mit jedem Knoten des Baums ist eine Nummer assoziert und nur diese wird abhängig von der aktuellen Größe des Baums als 9–12 Bit langes Wort statt einer Symbolsequenz ausgegeben. Da der Baum erst erweitert wird, nachdem das entsprechende zur Erweiterung des Baums beitragende Symbol übertragen wurde, kann der Decoder synchron mit dem Encoder den gleichen Baum aufbauen und somit die ankommenden Knotennummern durch die entsprechenden Symbolfolgen wieder ersetzen. Wenn der Baum einmal die maximale Größe von z.B. 2^{12} Knoten erreicht hat, wird er auf beiden Seiten gelöscht und einer neuer Baum baut sich mit den folgenden Daten auf. Der LZW-Algorithmus ist besonders effizient, da durch die Baumstruktur des verwendeten Puffers kein Aufwand für die Suche nach identischen älteren Symbolsequenzen auf der Seite des Encoders notwendig ist.

2.6 Anwendungsbeispiele

In diesem Abschnitt wird kurz ausgeführt, wie die eben erläuterten Grundverfahren der Datenkompression in einigen gängigen Systemen eingesetzt werden.

2.6.1 Telefax Kompression

Der in heutigen Gruppe 3 Telefaxgeräten eingesetzte Kompressionsalgorithmus ist eine Kombination aus einer Lauflängencodierung und einem Huffman-Code [ITU93b]. Der in

einem Telefaxgerät eingebaute Scanner digitalisiert zunächst den Text auf einer Breite von 215 mm mit 1728 Bildpunkten pro Zeile und mit 3.81 Zeilen/mm. Diese Standardauflösung entspricht 204×97 Punkte pro Zoll (dpi). Optional sind auch noch höherauflösende Modi mit alternativ 3456 Bildpunkten pro Zeile und 7.7 sowie 15.4 Zeilen/mm verfügbar, was dann einer alternativen horizontalen Auflösung von 408 dpi, sowie weiteren vertikalen Auflösungen von 196 bzw. 391 dpi entspricht. Darüber hinaus wurde der Fax-Standard 1993 um die möglichen Auflösungen 200×100, 200×200, 300×300 und 400×400 dpi erweitert, um eine einfachere Ausgabe auf gängigen Laserdruckern zu ermöglichen.

Beim Gruppe 3 Fax-Kompressionsverfahren wird zunächst durch eine Lauflängencodierung jede Zeile in abwechselnd weiße und schwarze Ketten von Bildpunkten gleicher Farbe unterteilt. Jede Zeile beginnt stets mit einer weißen Kette, die falls das erste Pixel schwarz sein sollte auch die Länge Null haben kann. Die so erhaltenen Kettenlängen werden durch vorgegebene Hufman-Codes ersetzt und ausgegeben. Es wird in einem Faxgerät keine Konstruktion eines Codebaums mit dem Huffman-Algorithmus durchgeführt, sondern der verwendete Hufman-Code wurde bei der Entwicklung des Verfahrens anhand einer Reihe von Beispieldokumenten in der Standardauflösung von 1728 Bildpunkten pro Zeile festgelegt. Da die Verteilungsfunktionen der Längen von weißen und schwarzen Ketten sich deutlich unterscheiden, werden zwei verschiedene Codebäume eingesetzt, die sich laufend abwechseln. Die Codes für die Kettenlängen 0 bis 63 sind sogenannte terminale Codes, da sie alleine bereits vollständig die Länge einer Kette angeben. Für größere Kettenlängen existieren nur die sogenannten Make-up Codes $64, 128, 196, \ldots, 1728$. Kettenlängen über 63 werden durch eine Kombination aus einem Make-up Code und einem terminalen Code dargestellt, die Länge der Kette ergibt sich aus der Summe der beiden durch diese Codes dargestellten Ketten-Längen. Dadurch wird verhindert, daß Faxgeräte zwei Codebäume mit jeweils 1729 Symbolen im Speicher haben müssen.

Bei der Fax-Standardauflösung dominieren in der Verteilung der schwarzen Kettenlängen sehr deutlich die Werte 1–4, da schwarze Ketten in der Regel nur Schnittlinien mit den sehr kurzen und hauptsächlich senkrechten Linien von Buchstaben sind. Diese häufigen Werte werden daher auch mit 2 bis 3 Bit langen Codewörtern abgedeckt. Längere schwarze Ketten als etwa 16 Punkte tauchen sehr selten auf und werden daher mit Codewörtern zwischen 10 und 13 Bit Länge versehen. Die Verteilung der weißen Kettenlängen dagegen ist weniger eng lokalisiert, weshalb die kürzesten Codewörter nicht weniger als 4 Bit Länge im Codebaum für weiße Ketten haben, dafür aber die restlichen Codeswörter deutlich kürzer sind als die entsprechenden im schwarzen Baum. Abgeschlossen wird jede Zeile mit dem Zeilenendecode 000000000001. Da die beiden Codebäume so konstruiert sind, daß niemals 11 Nullen unmittelbar nacheinander stehen können, kann der Zeilenendecode auch erkannt werden, wenn durch Übertragungsfehler der Decoder desynchronisiert wurde. Da in älteren Telefaxgeräten noch keine Fehlerkorrekturprotokolle vorgesehen waren, mußte der Codieralgorithmus sicherstellen, daß gelegentliche Übertragungsfehler höchstens den Inhalt einer Zeile beeinflussen können.

Die folgenden Ausschnitte aus den beiden Codetabellen zeigen deutlich, wie die unterschiedlichen Verteilungsfunktionen der Kettenlängen für schwarze und weiße Bildpunkte die Huffman-Codes beeinflußt hat:

Kettenlänge	Code (weiß)	Code (schwarz)
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
5	1100	0011
6	1110	0010
7	1111	00011
8	10011	000101
9	10100	000100
10	00111	0000100
11	01000	0000101
12	001000	0000111
13	000011	00000100
14	110100	00000111
15	110101	000011000
16	101010	0000010111
...	...	...
63	00110100	000001100111
64	11011	0000001111
128	10010	000011001000
192	010111	000011001001
...	...	...
1728	010011011	0000001100101

Neben der beschriebenen und gängigen 1-dimensionalen Kompression gibt es alternativ noch ein deutlich komplizierteres 2-dimensionales Verfahren, auf daß hier aber nur kurz eingegangen werden soll. Dabei bestimmt ein Parameter K, daß nur jede K-te Bildzeile mit dem 1-dimensionalen Verfahren codiert werden soll, um im Fehlerfall eine Resynchronisation nach maximal K Bildzeilen sicherzustellen. Allen anderen Zeilen werden abhängig vom Inhalt der Vorgängerzeile codiert. Die Grundidee des 2-dimensionalen Fax-Codierverfahrens ist die Beobachtung, daß schwarz/weiß und weiß/schwarz Wechsel in aufeinanderfolgenden Bildzeilen oft an fast der gleichen Position stattfinden. Um eine Kette, zu codieren stehen drei verschiedene Modi zur Verfügung. Im *pass mode* wird ein Farbübergang in der darüberliegenden Zeile übersprungen, im *vertical mode* werden die Grenzen einer Kette relativ zur Lage der Grenzen der darüberliegenden Kette im Bereich -3 bis $+3$ angegeben und schließlich im *horizontal mode* wird vorübergehend wieder auf eine Lauflängencodierung zurückgegriffen, wenn keine passende Kette in der darüberliegenden Zeile vorhanden ist.

2.6.2 Gängige komprimierende Dateiformate

Zur kompakten Auslieferung von Software wird auf Personal Computern und UNIX Rech-

nern häufig der *deflate* Algorithmus eingesetzt. Dieser wurde ursprünglich für die Archivierungssoftware `pkzip` entwickelt und, da der Quellcode frei zur Verfügung stand, später auch in die GNU `gzip` Software [Gai94] sowie als Kompressionsalgorithmus des *Portable Network Graphics (PNG)* Graphikdateiformats eingesetzt.

Es handelt sich bei *deflate* um eine Implementation des LZ77-Algorithmus, die mit dem Huffman-Verfahren kombiniert wurde. Der Pufferspeicher hat die Länge $L = 32768$ Byte und referenzierte Byte-Sequenzen können maximal 258 Bytes lang sein. Alle vom LZ77-Algorithmus ausgegebenen Werte werden mit zwei verschiedenen Huffman-Bäumen codiert. Ein Codebaum enthält Bitsequenzen für alle direkt übernommenen Bytes sowie für alle Längenangaben von referenzierten Sequenzen, der zweite Baum codiert nur die Orte an denen referenzierte Sequenzen im Puffer beginnen. Da die direkt übernommenen Bytes und die Sequenzlängen im selben Baum untergebracht sind, lassen sie sich eindeutig unterscheiden. Nach jeder codierten Sequenzlänge wird für ein Symbol auf den zweiten Codebaum umgeschaltet um auch den Ort der Sequenz abzulegen. Das Ergebnis des LZ77-Algorithmus wird in einem Puffer zwischengespeichert und anschließend werden für diesen Block zwei Huffman-Bäume berechnet. Die Huffman-Bäume selbst werden ebenfalls komprimiert vor dem eigentlichen Datenblock abgelegt, indem nur die Anzahl der pro Symbol vergebenen Bits abgespeichert wird, woraus sich der Decoder eindeutig den Codebaum rekonstruieren kann. Der Encoder trägt alle gelesenen 3-Byte Sequenzen in eine Hash-Tabelle ein und verkettet diese Triplets entsprechend ihrer Ankunftsreihenfolge. Mittels dieser Datenstruktur kann beim Eintragen neu ankommender Bytes sehr schnell festgestellt werden, ob und wo im Puffer die Sequenz bereits einmal auftauchte.

Das in der UNIX Welt gebräuchliche *compress* Programm zur universellen Redundanzreduktion einzelner Dateien ist eine Implementation des bereits erwähnten Lempel-Ziv-Welch-Algorithmus (LZW). Auch das insbesondere beim Graphikdatenaustausch auf dem Internet sehr verbreitete *CompuServe GIF* Graphikdateiformat basiert auf LZW. Dabei werden, falls in einer Originalbild-Datei mehr als 256 verschiedene Farbschattierungen auftreten, durch einen Vektorquantisierer diejenigen 256 Farben ausgesucht, die den vom fraglichen Bild benutzten Farbteilraum am besten repräsentieren. Diese maximal 256 Farben werden in eine Farbtabelle eingetragen und jedem Bildpunkt wird die Nummer des Farbtabelleneintrags zugeordnet, der eine der Farbe des Bildpunkts am naheliegendste Farbe repräsentiert. Diese Farbtabellenindizes werden anschließend zeilenweise an einen LZW-Algorithmus übergeben und vor dessen Ergebnis wird ein Dateikopf mit der Farbtabelle gestellt. Die Kompressionseffizienz von GIF ist etwa mit der von Lauflängencodierverfahren vergleichbar, d.h. nur Bilder, die aus wenigen Farben und großen monochromen Flächen bestehen werden gut komprimiert. Im Vergleich zur Lauflängencodierung kann GIF bei sehr einfachen und regelmäßigen Rasterungsmustern zusätzlich auch wiederkehrende Sequenzen ersetzen und dadurch bei derartigen Bildern deutlich bessere Ergebnisse erzielen.

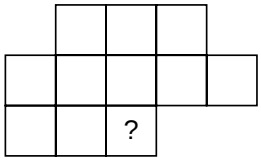
3 Das JBIG-Verfahren

3.1 Übersicht

Digitalisierte bi-level Textdokumente sind 2-dimensionale Felder aus Bildpunkten, von welchen jeder nur die Werte *schwarz* und *weiß* oder etwas allgemeiner formuliert *Vordergrund* und *Hintergrund* annehmen kann. Informationstheoretisch handelt es sich um eine Nachrichtenquelle mit einem binären Alphabet und die Symbolsequenz entsteht in der Regel durch die in deutscher Leserichtung von links nach rechts und von oben nach unten erfolgende Abtastung und Übermittlung der Werte.

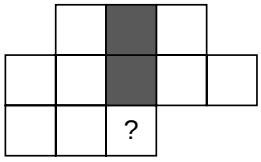
Die Symbole tauchen dabei sehr stark voneinander stochastisch abhängig auf. Diese Abhängigkeit beschränkt sich nicht auf die Beobachtung, daß mit hoher Wahrscheinlichkeit ein Bildpunkt die gleiche Farbe wie sein linker Nachbar hat, die beim Gruppe 3 Fax-Algorithmus und anderen Lauflängencodierverfahren ausgenutzt wird. Europäische Textdokumente sind in erster Linie aus Buchstaben einer oder mehrerer Schriftarten mit selten mehr als 100 verschiedenen Zeichen aufgebaut, sowie aus Geraden und gerasterten Flächen unterschiedlicher Helligkeit. Wenn bereits ein Teil eines Buchstabens übertragen wurde, dann könnte ein Decoder diesen Teil mit anderen bereits vollständig übertragenen Buchstaben vergleichen und daraus eine gute Schätzung für den Wert der folgenden Bildpunkte dieses Buchstabens abgeben. Diese Schätzung der Verteilung für den nächsten Bildpunkt kann mit dem gleichen Algorithmus anhand der bereits übertragenen Daten sowohl im Encoder als auch im Decoder durchgeführt werden. Je besser diese Schätzung möglich ist, desto höher ist die Wahrscheinlichkeit, daß der Encoder nur die Nachricht *der erwartete Wert ist eingetreten* übertragen muß, die aufgrund ihrer hohen Wahrscheinlichkeit einen geringen Informationsgehalt hat und deshalb mit deutlich weniger als einem Bit codiert werden kann. Nur im selteneren Fall, daß nicht der der Schätzung entsprechende Wert auftritt muß diese Nachricht höheren Informationsgehalts mit mehreren Bits übertragen werden.

Diese Schätzungen werden beim JBIG-Verfahren mittels der kausalen Nachbarbildpunkte durchgeführt, also mit den Nachbarpunkten, die aufgrund der Abtastrichtung bereits auch dem Decoder bekannt sind:



Der Bildpunkt mit dem Fragezeichen in der Abbildung ist als nächstes zu übertragen und die Wahrscheinlichkeit, ob es sich um einen schwarzen oder weißen Bildpunkt handelt wird mittels der links und über ihm liegenden 10 Nachbarpunkte abgeschätzt. Das gezeigte Nachbarschaftsmuster ist nur eines von mehreren die zur Auswahl stehen. Die 10 Nachbarpunkte bilden den *Kontext* des zu übertragenden Punkts und da jeder Bildpunkt nur zwei Werte annehmen kann gibt es nur $2^{10} = 1024$ verschiedene Kontexte. Encoder und Decoder führen für alle 1024 verschiedenen Kontexte getrennt eine Statistik darüber, wie oft bereits die beidem Farbalternativen in diesem Kontext aufgetreten sind. So eine Statistik

enthält dann beispielsweise nachdem bereits ein Teil des Dokuments übertragen wurde die Aussage, daß bei einem Kontext der Form



mit einer Wahrscheinlichkeit von beispielsweise 0.042 mit einem weißen und mit einer Wahrscheinlichkeit von 0.958 mit einem schwarzen Bildpunkt zu rechnen ist. Diese Werte können etwa dadurch entstehen, daß der obige Beispielkontext in der Regel ein guter Hinweis darauf ist, daß der zu übertragende Bildpunkt Teil einer senkrechten schwarzen Linie ist. Falls der wahrscheinlichere Fall eintritt wären also nur $-\log_2 0.958 \approx 0.062$ Bits zu übertragen, während im selteneren Fall $-\log_2 0.042 \approx 4.57$ Bit fällig sind, womit die Entropie dieses Kontexts bei etwa 0.25 Bit liegt. Der Graph der Entropie $H(p, 1-p)$ einer binären Quelle aus Abschnitt 2.1 zeigt, wie mit einer zuverlässigeren Vorhersage die Entropie der Quelle und damit die Länge des zu übertragenden Datenstroms stark absinkt.

Das JBIG-Verfahren [ITU93a] setzt einen arithmetischen Coder für binäre Alphabete ein. Aufgrund der in den einzelnen Kontexten seit Beginn einer Seite aufgetretenen Bildpunkte werden die Verteilungen fortlaufend während der Codierung geschätzt. Es werden im Gegensatz zum Gruppe 3 Fax-Verfahren keine einmalig für bestimmte repräsentative Testseiten festgelegte Verteilungen eingesetzt, sondern für jede zu codierende Seite werden während des Codiervorgangs neue Statistiken erstellt. Damit ist JBIG im Gegensatz zum Fax-Algorithmus völlig unabhängig von der Abtastauflösung, der verwendeten Schriftart und deren Größe. All diese Faktoren beeinflussen erheblich die der Kompression zugrundeliegende Statistik und JBIG kann sich jeweils optimal darauf einstellen. Dagegen beruht das Fax-Verfahren auf einer feststehenden Statistik, die versucht, für mit Schreibmaschine geschriebene europäische Geschäftsbriefe, handschriftliche Notizen, technische Zeichnungen, Buchseiten, wissenschaftliche Veröffentlichungen und japanischen Text einen Kompromiß zu finden.

Eine herausragende Besonderheit des JBIG-Verfahrens ist die Möglichkeit zur progressiven Codierung. Dabei wird das Bild in mehreren Auflösungsstufen abgelegt. Beispielsweise könnte ein Bild in den Auflösungen $R_0 = 75$ dpi, $R_1 = 150$ dpi und $R_2 = 300$ dpi abgespeichert werden. In einer Datei würde man die einzelnen Versionen dann in aufsteigender Reihenfolge ablegen, so daß bereits während der noch laufenden Übertragung die 75 dpi Version auf dem Bildschirm dargestellt werden kann, obwohl die wesentlich umfangreichere 300 dpi Fassung noch übertragen wird.

Sei D die Anzahl der zusätzlich zum Original vorhandenen Auflösungsstufen. Dann bezeichnen X_D, Y_D und R_D die horizontale und vertikale Anzahl der Bildpunkte sowie die Auflösung des Originals, und X_0, Y_0 sowie R_0 die entsprechenden Daten der kleinsten Auflösungsstufe. Jede nächst kleinere Auflösungsstufe ist jeweils bis auf Rundung halb so groß wie die darüber liegende, d.h.

$$X_{d-1} = \lceil X_d/2 \rceil$$
$$Y_{d-1} = \lceil Y_d/2 \rceil$$

für alle $0 < d \leq D$. Jedem Bildpunkt einer niedrigeren Auflösungsstufe entsprechen daher vier Punkte der nächst höheren Stufe. Außer bei Schicht 0 werden bei der Codierung aller

Schichten auch noch vier Nachbarpunkte aus der dem Decoder dann bereits bekannten nächst niedrigeren Auflösungsstufe mit in den Kontext einbezogen. Damit wird bei der Codierung von höheren Auflösungsebenen die bereits übertragene Information aus den niedrigeren Stufen mit berücksichtigt, wodurch die progressive Codierung deutlich weniger Daten zur Übertragung benötigt als wenn alle Auflösungen einzeln übertragen werden würden.

Darüber hinaus ist es möglich, ein zu komprimierendes Bild in mehrere horizontale Streifen einzuteilen. Alle Streifen haben in der niedrigsten Auflösungsstufe eine Höhe von L_0 Bildpunkten, nur der letzte Streifen eines Bilds kann kürzer sein. Die Streifenaufteilung erstreckt sich über alle Auflösungsstufen hinweg, d.h. $L_{d+1} = 2L_d$. Die Aufteilung in Streifen erlaubt es insbesondere bei sehr großen komprimierten Dokumenten (z.B. technische Zeichnungen im A0 Format) schneller auf kleine Ausschnitte zuzugreifen, ohne alle darüberliegenden Daten decodieren zu müssen. Ferner dient die Unterteilung in Streifen dazu, bei der Codierung von anderen als bi-level Bildern (z.B. Farb- oder Graustufenbildern) zunächst die ersten Streifen aller Bitschichten zu übertragen, bevor mit den darunterliegenden zweiten Streifen begonnen wird. Dadurch kann Speicherplatz gespart werden, da nicht komplette Bitebenen zwischengespeichert werden müssen.

Ein mit JBIG zu komprimierender Datensatz besteht insgesamt aus $P > 0$ einzelnen Bitebenen (mit $P > 1$ falls mehr als nur 2 Werte pro Bildpunkt codierbar sein sollen), aus $D + 1$ Auflösungsstufen, sowie aus $S = \lceil Y_0/L_0 \rceil$ Streifen. Dadurch werden die Ausgangsdaten in $S(D + 1)P$ Datenblöcke, sogenannte *stripe data entities (SDEs)*, zerlegt. Die Anordnung der SDEs wird durch drei Schleifen über $0 \leq s < S$, $0 \leq d \leq D$ und $0 \leq p < P$ für die Ausgabe von $\mathrm{SDE}_{s,d,p}$ bestimmt. Es gibt $3! = 6$ mögliche Anordnungen für diese Schleifen. Zusätzlich besteht die Option, die Auflösungsstufen entweder in fallender oder ansteigender Reihenfolge abzulegen, wodurch sich insgesamt 12 mögliche Anordnungen von SDEs in einem JBIG Datenstrom ergeben, die alle vom Standard zugelassen sind und durch die vier Bits `HITOLO`, `SEQ`, `ILEAVE` und `SMID` im Kopf des Datenstroms identifiziert werden. Da die Auflösungsstufen im Encoder in fallender Reihenfolge anfallen, im Decoder aber in aufsteigender Reihenfolge benötigt werden, wurde die durch das gesetzte `HITOLO`-Bit gekennzeichnete fallende Auflösungsreihenfolge als Option mit in den Standard aufgenommen, um Encoder mit sehr geringem Speicherplatzbedarf zu ermöglichen. In diesem Fall müßte eine Dokumentendatenbank zwischen Encoder und Decoder die SDEs umsortieren und das `HITOLO`-Bit löschen, da Decoder nur die ansteigende Auflösungsreihenfolge akzeptieren müssen. Die Codierung der einzelnen SDEs hängt nicht von der gewählten Speicherreihenfolge ab, so daß leicht im nachhinein die Datenanordnung entsprechend den Bedürfnissen des Decoders verändert werden kann.

Der Encoder beginnt mit der Bearbeitung von Auflösungsstufe D und erzeugt vor Codierung jeder Stufe außer bei Stufe 0 die nächst niedrigere Auflösungsstufe. Anschließend werden die Bildpunkte der einzelnen Streifen in üblicher Leserichtung abgearbeitet. Zunächst wird ein sogenannter „typische Vorhersage“ Algorithmus (TP = *typical prediction*) angewendet, um Bildpunkte auszusortieren, die einfacheren Gesetzmäßigkeiten entsprechend codiert werden können. In der niedrigsten Auflösungsstufe werden dabei beispielsweise identische aufeinanderfolgende Zeilen entfernt. In allen Stufen außer Stufe 0 wird darüber hinaus ein „deterministische Vorhersage“ Algorithmus (DP = *deterministic prediction*) eingesetzt, der verhindert, daß Bildpunkte vom arithmetischen Encoder bearbeitet werden, deren Wert sich bereits durch Kenntnis der schon übertragenen Bildpunkte (insbesondere auch aus der nächst niedrigeren Stufe) sowie den Eigenschaften des Algorithmus zur Auf-

lösungsreduktion eindeutig ergibt. Alle anderen Bildpunkte eines Streifens, die weder durch den TP- noch durch den DP-Algorithmus bereits entfernt wurden, werden zusammen mit ihren Kontext dem arithmetischen Encoder übergeben, der für jeden Kontext eine Statistik führt und alle diese Punkte in eine einzige Binärzahl im Intervall 0 bis 1 abspeichert. Der Decoder wendet die entsprechenden Schritte in umgekehrter Reihenfolge an, beginnt also mit der niedrigsten Auflösungsstufe und baut die Auflösungsstufen mit den Ergebnissen des arithmetischen Decoders auf, wobei gegebenenfalls der TP- bzw. DP-Algorithmus des Decoders die durch diese Schritte im Encoder aussortierten Bildpunkte, deren Codierung vermeidbar ist, wieder einfügen.

3.2 Der arithmetische Coder

Der im JBIG-Algorithmus eingesetzte arithmetische Encoder basiert auf zwei 32 Bit großen Registern A und C, welche die Länge sowie das untere Ende des aktuellen Codierintervalls enthalten. Die mittels Multiplikation mit 2 (also einer Linksschiebeoperation) durchgeführte Renormalisierung stellt sicher, daß sich A stehts im Intervall $[0\text{x}8000, 0\text{x}10000]$ befindet (die $0\text{x}\ldots$ Notation steht für hexadezimale Zahlenwerte, also z.B. $0\text{x}1\text{f} = 31$).

Aufgabe des Encoders ist es, dem Decoder mitzuteilen, ob das laut Modell wahrscheinlichere der beiden Symbole (MPS für engl. *more probable symbol*) oder das weniger wahrscheinliche Symbol (LPS für engl. *less probable symbol*) eingetroffen ist. Per Konvention wird das Codierintervall stets so aufgeteilt, daß der Bereich des LPS über dem Bereich des MPS liegt. Somit muß auf den beiden Registern im Fall des MPS die Operation

$$A := p_{\mathrm{MPS}} \cdot A$$

und im Fall des LPS die beiden Operationen

$$\begin{aligned} A &:= p_{\mathrm{LPS}} \cdot A \\ C &:= C + p_{\mathrm{MPS}} \cdot A = C + A - p_{\mathrm{LPS}} \cdot A \end{aligned}$$

durchgeführt werden, wobei $p_{\mathrm{LPS}} = 1 - p_{\mathrm{MPS}} \leq 0.5$ die Wahrscheinlichkeiten für die beiden möglichen Symbole repräsentieren.

Dieser Ansatz würde jedoch eine bei den meisten Prozessoren sehr zeitaufwendige 32 Bit Ganzzahlmultiplikation erfordern. Daher wäre es wünschenswert, die Multiplikationsergebnisse fest in einer Tabelle abzulegen statt sie jedes Mal neu auszurechnen. Für p_{LPS} kommen beim JBIG-Verfahren ohnehin nur 113 verschiedene Werte in Frage, da die Symbol-Wahrscheinlichkeiten von einem endlichen Automaten mit 113 Zuständen geschätzt werden. Daher wird statt der Multiplikation mit einem der 32769 möglichen Werte von A die Approximation

$$p_{\mathrm{LPS}} \cdot A \approx p_{\mathrm{LPS}} \cdot \bar{A} =: \mathtt{LSZ}$$

durchgeführt, wobei $\bar{A}$ der Erwartungswert von A ist. Dadurch ergibt sich im längerfristigen Mittel die gleiche Länge des Codierintervalls wie wenn die Multiplikation jedes mal präzise durchgeführt wird. Empirische Beobachtungen haben gezeigt, daß die Wahrscheinlichkeit eines Werts A etwa proportional zu $1/A$ ist, d.h. $p(A) = k/A$ für eine Konstante k mit

$$\sum_{A=0\text{x}8000}^{0\text{x}10000} p(A) = k \sum_{A=0\text{x}8000}^{0\text{x}10000} \frac{1}{A} = 1,$$

woraus sich $k \approx 1.442647$ und der Erwartungswert

$$\bar{A} = \sum_{A=0\text{x}8000}^{0\text{x}10000} A \cdot p(A) = k \sum_{i=0\text{x}8000}^{0\text{x}10000} \frac{A}{A} = k \cdot 0\text{x}8001 \approx 0.721 \cdot 0\text{x}10000 \approx 0\text{xb}893$$

ergibt.

Also reicht es für den Encoder aus, im Fall des wahrscheinlicheren Symbols nur die Operation

$$A := A - \texttt{LSZ} \approx (1 - p_{\text{LPS}}) \cdot \bar{A} = p_{\text{MPS}} \cdot \bar{A}$$

und im Fall des LPS die beiden Operationen

$$\begin{aligned} A &:= \texttt{LSZ} = p_{\text{LPS}} \cdot \bar{A} \\ C &:= C + A - \texttt{LSZ} = C + A - p_{\text{LPS}} \cdot \bar{A} \approx C + p_{\text{MPS}} \cdot \bar{A} \end{aligned}$$

durchzuführen. Sollte dabei A unter 0x8000 fallen, so werden A und C renormalisiert.

Es kann aufgrund der Approximation $p_{\text{LPS}} \cdot A \approx \texttt{LSZ}$ vorkommen, daß das Intervall für das weniger wahrscheinliche Symbol im Einzelfall größer ist als das Intervall für das MPS, also daß $\texttt{LSZ} > A - \texttt{LSZ}$. In diesem Fall werden einfach die beiden Symbole vorübergehend vertauscht. Da der Zustand $\texttt{LSZ} > A - \texttt{LSZ}$ auch auf der Seite des Decoders leicht erkannt werden kann, ist diese Korrekturmaßnahme beim Empfänger wieder umkehrbar. So ist auch sichergestellt, daß nach jedem LPS $A < 0\text{x}8000$ gilt und damit nach jeder Codierung des LPS eine Renormalisierung fällig ist. Die Überprüfung auf $A < 0\text{x}8000$ vor einer Renormalisierung ist daher nur beim MPS notwendig.

Zur Renormalisation werden sowohl A als auch C um ein Bit nach links verschoben. Nach jeder 8. Renormalisation werden 8 Bits aus der oberen Hälfte des 32 Bit Registers C entnommen. Dieses Byte wird dabei nicht sofort ausgegeben, sondern in einer Puffervariable zwischengespeichert, da die Möglichkeit besteht, daß bei weiteren Additionen auf das C Register ein Überlauf auftritt. Sollten alle auszugebenden 8 Bits 1 sein, dann wird nicht der Wert 0xff in die Puffervariable übernommen, sondern nur ein Zähler `SC` um eins erhöht. In `SC` ist vermerkt, wieviele 0xff, die ja alle von einem Überlauf betroffen wären, noch auszugeben sind. Wenn bei der nächsten Renormalisierung in C ein Übertrag auftreten sollte, dann wird die Variable, die sich noch um Pufferregister befindet um Eins erhöht und anschließend `SC` mal der Wert 0x00 ausgegeben. Wenn aber kein Überlauf aufgetreten ist und auch kein weiteres 0xff auszugeben ist, dann können sowohl der Wert aus der Puffervariable als auch die `SC` 0xff-Bytes ausgegeben werden, denn den nächsten Überlauf wird nun das neu aus C entnommene Byte übernehmen, das sich nun in der Puffervariable befindet. Mit diesem Überlaufbehandlungsverfahren kann beliebig lange auf die Variable C mit immer kleineren Werten aufaddiert werden, ohne daß es zu Problemen mit dem nur 32 Bit langen Register kommt, daß für C benutzt wird.

Im einzelnen setzt sich das 32 Bit Register C aus folgenden Bereichen zusammen:

$$C = 0000cbbb\ bbbbbsss\ xxxxxxxx\ xxxxxxxx$$

Die 16 x-Bits sind der Bereich, in dem der maximal 16 Bit große Wert $A - \texttt{LSZ}$ aufaddiert wird. Die 8 b-Bits sind das Byte, das bei jeder 8. Renormalisierung aus C entnommen und an die Überlaufbehandlung übergeben wird. Falls das Bit c den Wert 1 hat, so muß das

zuletzt ausgegebene Byte noch um eins erhöht werden, weshalb es nicht sofort ausgegeben wird, sondern in einer Puffervariable gehalten wird. Die 3 Bits *sss* zwischen dem Bereich in dem addiert wird und den Bits die ausgegeben werden dienen dazu, sicherzustellen, daß niemals über das c-Bit hinaus ein Übertrag stattfinden kann, was die Überlaufbehandlung wesentlich komplexer machen würde.

Charakteristisch für das beim JBIG-Standard eingesetzte arithmetische Codierverfahren ist die Technik, mit der die Wahrscheinlichkeiten p_{MPS} und p_{LPS} abgeschätzt werden, also letztlich der Wert `LSZ` festgelegt wird. Diese Technik wurde im wesentlichen von dem bei IBM entwickelten Q-Coder übernommen [Pen88a]. Für jeden Kontext wird ein endlicher Automat mit 113 verschiedenen Zuständen implementiert. Jedem Zustand entspricht ein geschätzter p_{LPS}-Wert und durch eine Tabelle mit 113 Einträgen wird aus der Nummer des Zustands der zur Codierung notwendige `LSZ`-Wert gewonnen. Zusätzlich zum Zustand ist für jeden Kontext-Automaten auch vermerkt, ob es sich bei der aktuellen wahrscheinlicheren Farbe um die Vordergrund- oder Hintergrundfarbe handelt, also ob das MPS für einen Pixelwert 1 oder 0 steht.

Abgeschätzt werden die Wahrscheinlichkeiten p_{LPS} und $p_{\mathrm{MPS}} = 1 - p_{\mathrm{LPS}}$ anhand der hypothetischen Zähler n_{LPS} und n_{MPS}, die mitzählen, wie oft seit Beginn des Codiervorgangs die beiden Symbole bereits aufgetreten sind. Der Zustand des Automaten beinhaltet nicht nur die aktuelle Wahrscheinlichkeit p_{LPS}, sondern auch wie sicher diese Wahrscheinlichkeit ist, das heißt, wie schnell sich die Schätzung von p_{LPS} durch neu ankommende Symbole beeinflussen lassen soll. Die 113 Zustände des Automaten lassen sich als Punkte in einer n_{LPS}-n_{MPS}-Ebene auffassen. Der Quotient $n_{\mathrm{LPS}}/(n_{\mathrm{LPS}} + n_{\mathrm{MPS}})$ ist dabei die Maximum-Likelihood-Schätzung der Wahrscheinlichkeit p_{LPS} [Bro89, Kap. 5.2.2.2], während der Abstand vom Ursprung in etwa angibt, wie sicher diese Schätzung ist. Bei der Erstellung der Zustandstabelle für den JBIG-Coder wurde genaugenommen statt der obigen einfachen Maximum-Likelihood-Schätzung die etwas präzisere Bayes'sche Schätzung

$$\hat{p}_{\mathrm{LPS}} = \frac{n_{\mathrm{LPS}} + \delta}{n_{\mathrm{LPS}} + \delta + n_{\mathrm{MPS}} + \delta}$$

verwendet, wobei $\delta = 0.45$ ein empirisch ermittelter Parameter ist, der von der Verteilung der als Zufallsvariable betrachteten zu schätzenden Wahrscheinlichkeit p_{LPS} abhängt.

Da sich der Schätzalgorithmus an eine geänderte Wahrscheinlichkeit p_{LPS} (beispielsweise hervorgerufen durch einen Wechsel der Schriftart in einem Dokument) auch nach langer Zeit noch anpassen können muß, wird n_{LPS} nach oben hin beschränkt. Wenn n_{LPS} über eine obere Grenze steigt, so werden n_{LPS} und n_{MPS} durch den gleichen Faktor dividiert, so daß die Anzahl der in die Schätzung einfließenden Symbole nicht beliebig wachsen kann. Eine größere obere Schranke für n_{LPS} erlaubt eine präzisere Schätzung, macht diese aber auch träger bei Änderungen. Die obere Grenze für n_{LPS} liegt abhängig von n_{MPS} im Bereich 1 bis 11, so daß Wahrscheinlichkeiten nahe 0.5 eher präzise geschätzt werden, während sehr kleine Wahrscheinlichkeiten sich sehr kurzfristig ändern können.

Sinn der Vereinfachung des Schätzalgorithmus auf einen Automaten, dessen 113 Zustände Repräsentanten für Punkte der n_{LPS}-n_{MPS}-Ebene sind, ist es, den Schätzalgorithmus sehr einfach und effizient auch in Hardware implementierbar zu gestalten. Ein Ansatz für die Konzeption eines derartigen Automaten wäre es, bei jedem ankommenden Symbol auf einen neuen Zustand überzugehen. Der Automat soll jedoch Werte von p_{LPS} bis hinunter zu 0.00002 darstellen können und der erste Repräsentant dieser Wahrscheinlichkeit in der

n_{LPS}-n_{MPS}-Ebene liegt bei den Koordinaten $(0, 22498)$. Daher würde ein Automat, der bei jeder Ankunft eines Symbols nur um die Strecke 1 sich in der n_{LPS}-n_{MPS}-Ebene bewegen kann, viele zehntausend Zustände benötigen, was riesige Zustandstabellen erforderlich machen und sicher keine einfache Hardwareimplementation erlauben würde.

Daher finden Zustandsübergänge nur bei Renormalisierungen statt. Bei Eintreffen eines LPS findet stets eine Renormalisierung statt, aber da n_{LPS} nicht über 11 wachsen kann ist in diese Richtung auch keine große Anzahl von Zuständen möglich. Je kleiner p_{LPS}, also je größer n_{MPS} wird, desto geringer wird die Wahrscheinlichkeit, daß ein weiteres ankommendes MPS eine Renormalisierung auslösen wird, da `LSZ` immer weiter sinkt. Diese Wahrscheinlichkeit beträgt

$$p(A \cdot (1 - p_{\mathrm{LPS}}) < \mathrm{0x8000}) = p(A < \mathrm{0x8000}/(1 - p_{\mathrm{LPS}})).$$

Wegen $p(A = t) = k/t$ läßt sich die Verteilungsfunktion von $p(A)$ abschätzen mit

$$p(A < x) = \int_{\mathrm{0x8000}}^{x} p(A = t)\, dt = \log_2 \frac{x}{\mathrm{0x8000}}$$

für $x \in [\mathrm{0x8000}, \mathrm{0x10000}]$. Damit beträgt die Wahrscheinlichkeit, daß beim Codieren eines MPS eine Renormalisierung stattfindet

$$p(A \cdot (1 - p_{\mathrm{LPS}}) < \mathrm{0x8000}) = p(A < \mathrm{0x8000}/(1 - p_{\mathrm{LPS}})) = \log_2 \frac{1}{1 - p_{\mathrm{LPS}}}.$$

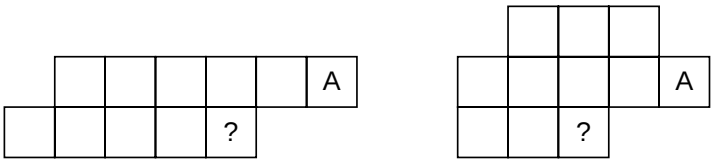
Es muß beim Renormalisieren nach Eintreffen des wahrscheinlicheren Symbols in der n_{LPS}-n_{MPS}-Ebene ein Schritt der Größe $-log_2 p_{\mathrm{MPS}}$ in n_{MPS}-Richtung ausgeführt werden, da im Mittel etwa soviele MPS seit dem letzten Zustandswechsel eingetroffen sind. Da nun weiter vom Ursprung entfernt die n_{LPS}-n_{MPS}-Ebene anläßlich der Renormalisierungen nur noch in größeren Schritten vom Schätzautomaten durchquert wird, reichen beispielsweise ganze 13 Zwischenzustände und Renormalisationen aus, um von $(0, 0)$ nach $(0, 22498)$ zu gelangen. Auf diese Weise läßt sich die n_{LPS}-n_{MPS}-Ebene im geforderten Bereich mit nur 113 Automatenzuständen repräsentativ abdecken.

Der Schätzautomat ist in [ITU93a] als Tabelle mit 113 Zuständen angegeben. Zu jedem Zustand ist ein `LSZ`-Wert für die Codierung, sowie die beiden möglichen Nachfolgezustände, die abhängig davon eintreten, ob die Renormalisierung durch das wahrscheinlichere (MPS) oder weniger wahrscheinlichere (LPS) Symbol ausgelöst wurde. Die Zustände repräsentieren alle nur den Fall $n_{\mathrm{LPS}} \leq n_{\mathrm{MPS}}$. Die Zustandsübergänge die durch ein LPS ausgelöst wurden und auf der n_{LPS}-n_{MPS}-Ebene die Winkelhalbierende in den Bereich $n_{\mathrm{LPS}} > n_{\mathrm{MPS}}$ hinein überschreiten würden sind besonders markiert, so daß in diesem Fall einfach das MPS und das LPS vertauscht werden und wieder $n_{\mathrm{LPS}} \leq n_{\mathrm{MPS}}$ gilt.

3.3 Kontextmuster

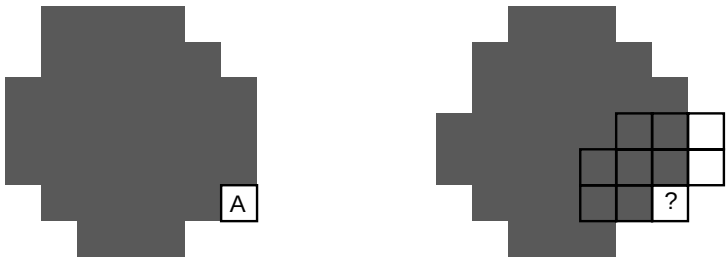
Der aktuelle Schätzautomatenzustand und ob das derzeitige MPS den Wert 0 oder 1 hat merkt sich der Encoder für jeden einzelnen möglichen Kontext. In der Auflösungsstufe 0

existieren $2^{10} = 1024$ mögliche Kontexte. Zur Bestimmung des aktuellen Kontexts eines Bildpunkts stehen die beiden folgenden Kontextmuster zur Verfügung:



Das 2-zeilige Kontextmuster erlaubt eine etwas effizientere Softwareimplementation, dafür schafft die 3-zeilige Variante ein etwa 5% besseres Kompressionsergebnis. Das Bit `LRLTWO` ist im Kopf des JBIG Datenstroms auf eins gesetzt, wenn sich der Encoder für die 2-zeilige Version entschieden hat.

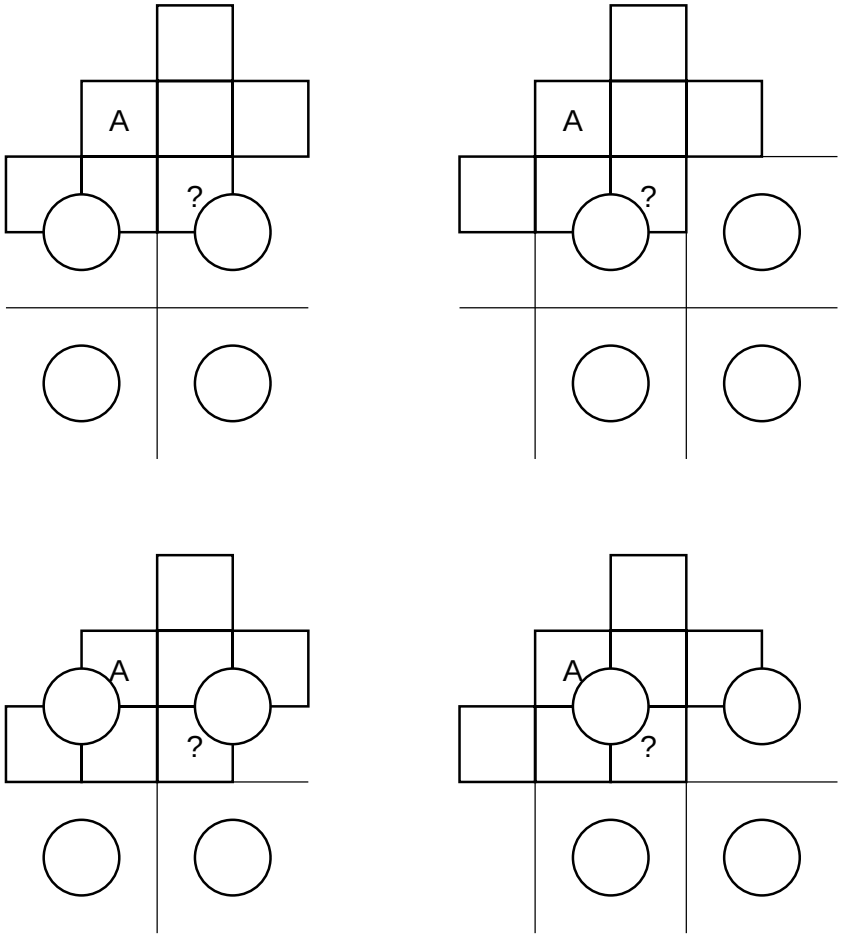
In der obigen Darstellung ist das mit „?“ gekennzeichnete Element des Kontextmusters der als nächstes zu codierende Bildpunkt, für dessen Schätzung der Kontext ermittelt wird. Das mit „A“ gekennzeichnete adaptive Element ist frei beweglich und kann innerhalb eines bis zu 255 Bildpunkte breiten und 256 Bildpunkte hohen Fensters links und oberhalb des zu codierenden Bildpunkts verschoben werden. Besonders bei gerasterten Grauflächen kann ein einzelner Referenzpunkt, dessen Abstand zum zu codierenden Punkt genau der Periodenlänge des Rastermusters entspricht, wesentlich mehr zu einer guten Schätzung beitragen als die unmittelbaren Nachbarn aus dem Kontext. Die folgende Darstellung zeigt einen Ausschnitt aus einem Zeitungsphoto in dem Graustufen mit einem Raster aus kleinen Kreisen dargestellt werden, sowie den dort auf einen Bildpunkt angesetzten und dem Rastermuster angepaßten Kontext:



Der Kontext des zu codierenden Bildpunkts und der des Punkts direkt darüber unterscheiden sich nicht, wenn nur das normale 3-zeilige Standardkontextmuster benutzt wird. Erst durch das adaptive Kontextelement hat der Coder eine Möglichkeit den Unterschied zwischen diesen beiden Fällen zu erkennen. Der Standard sieht auch vor, daß das bewegliche Kontextmuster-Element nach oben verschoben werden kann, jedoch wird von Decodern nur verlangt, daß sie mit adaptiven Elementen auf der untersten Kontextzeile bis zu 16 Schritte vom zu codierenden Bildpunkt entfernt umgehen können müssen, was für übliche Rastermuster und Auflösungen völlig ausreicht. Der Encoder überwacht ständig für alle in Frage kommenden Positionen des beweglichen Elements die Korrelation zum zu codierenden Punkt. Wenn diese gewisse Schwellwerte im Vergleich zur Korrelation der aktuellen Position an anderer Stelle überschreitet, so wird das Element bewegt und dies dem Decoder in einem `ATMOVE`-Segment das den SDE-Daten vorangestellt wird mitgeteilt.

Bei den höheren Auflösungsstufen wird ein anderes Kontextmuster eingesetzt, daß neben sechs Elementen aus der aktuellen Stufe auch noch vier Nachbarpixel der dem Decoder

bereits bekannten nächst niedrigeren Stufe beinhaltet. Da ein Bildpunkt der niedrigeren Auflösung vier Bildpunkte der höheren Stufe überdeckt, gibt es insgesamt vier Phasenlagen des neuen Bildpunkts relativ zur vorangegangenen Auflösungsstufe. Zwischen diesen vier Lagen wird unterschieden und es ergeben sich dadurch insgesamt $4 \cdot 2^{10} = 4096$ verschiedene Kontexte. In den vier Phasenlagen sehen die Kontextmuster wie folgt aus, wobei die Kreise den Bildpunkten der niedrigeren Stufe entsprechen, die jeweils die Fläche von vier höherauflösenden Punkten bedecken:



3.4 Auflösungsreduktion

Bevor ein JBIG-Encoder mit der progressiven Codierung von mehreren Auflösungsstufen beginnen kann, müssen zunächst einmal die Versionen niedrigerer Auflösung der Eingangsbilder erzeugt werden. Da die niedrigeren Auflösungsstufen jeweils nur die halbe Größe also ein viertel der Bildpunkte haben, wäre ein denkbarer einfacher Algorithmus zur Auflösungsreduktion, einfach jede zweite Zeile und Spalte eines Bilds zu entfernen. Dieses Trivialverfahren hat jedoch eine Reihe von Nachteilen:

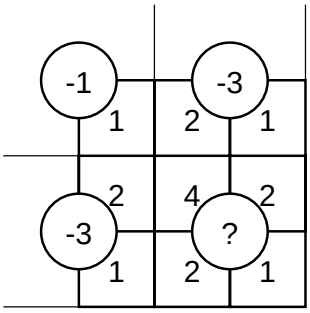
– Das Abtasttheorem nach NYQUIST wird verletzt. Nur wenn in der fouriertransformierten Darstellung eines abzutastenden Signals keine Frequenzen größer der halben Abtastfrequenz vorliegen, kann das abgetastete Signal vollständig rekonstruiert werden. Durch die Abtastung werden Frequenzanteile oberhalb der NYQUIST-Grenze in

den Bereich zwischen 0 und der halben Abtastfrequenz als Alias-Frequenzen abgebildet und erscheinen dort als Bildmerkmale, die im Ursprungssignal nicht vorhanden waren und daher stören. Vor einer Abtastung sollten daher die Frequenzanteile über der halben Abtastfrequenz mit einem Tiefpaß-Filter stark abgeschwächt werden, damit diese Anteile keine störenden Aliasing-Effekte verursachen können.

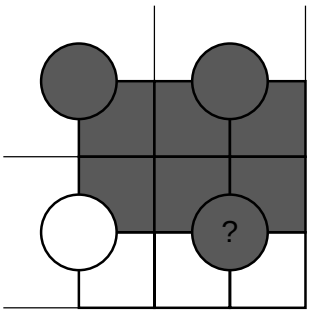
– Bei Auflösungen, die zur guten Darstellung des Dokumenteninhalts gerade noch ausreichen (also etwa 75–100 dpi) werden die senkrechten und waagrechten Striche der Buchstaben nur einen Bildpunkt breit dargestellt. Wenn einfach jede zweite Zeile und Spalte gestrichen wird, so werden damit viele dieser dünnen Striche ausgelöscht, was Text und Liniengraphiken völlig unkenntlich machen kann.

– Die Helligkeit von Rastermustern kann sich deutlich ändern. Wenn beispielsweise ein Grauwert durch stets abwechselnde schwarze und weiße Punkte dargestellt wird, so würde das Entfernen jeder zweiten Spalte alle schwarzen Punkte entfernen und die verbliebene Fläche wäre plötzlich völlig weiß.

All diese Probleme lassen sich letztendlich auf die Verletzung des Abtasttheorems und die dadurch entstehenden Alias-Effekte zurückführen. Daher bietet sich als Lösung ein zweidimensionaler Tiefpaßfilter mit nicht zu steiler Flanke an, so daß keine Überschwingeffekte auftreten. Im JBIG-Standard wird ein linearer rekursiver IIR (*infinite impulse response*) Filter vorgeschlagen, der nicht nur ein 3×3-Fenster aus der höherauflösenden Ebene, sondern auch noch drei Nachbarwerte aus dem bereits gefilterten Signal der niedrigerauflösenden Stufe berücksichtigt. Diese 12 Bildpunkte mit ihren möglichen Werten 0 und 1 werden mit den folgenden Filterkoeffizienten multipliziert, wobei „?“ den zu bestimmenden neuen Bildpunkt darstellt:

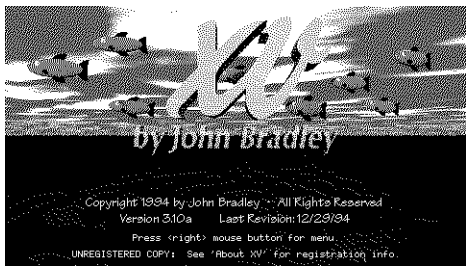


Sobald die Summe der Koeffizienten, deren entsprechender Bildpunkt den Wert 1 hatte, 5 oder größer ist wird der neue Punkt auf 1 gesetzt, sonst auf 0. Durch diesen Filter werden bereits weitgehend gute Ergebnisse erzielt, jedoch läßt sich dieser Algorithmus noch weiter verbessern. In einigen Fällen verschluckt oder öffnet der Filter noch Linien, und läßt gelegentlich Kanten etwas ausfransen. Ein Beispiel ist die Reaktion des Filters auf das folgende Muster:



Hier wird eine glatte horizontale Kante durch den Filter mit Unebenheiten versehen. Das obige Beispiel ist eine von 132 Ausnahmen von der Filterregel, die im JBIG-Auflösungsreduktionsverfahren eingesetzt werden um Zick-Zack Muster an geraden Kanten zu vermeiden. Darüber hinaus wurden 420 Ausnahmen zum Erhalt von Linien, 10 Ausnahmen, die die Ergebnisse bei Übergängen zwischen Rastermustern und 12 Ausnahmen, die die Behandlung von sehr dunklen oder sehr hellen Rastermustern mit wenigen Einzelpunkten verbessern gefunden. Da die $2^{12} = 4096$ Antworten des Filters in Implementationen ohnehin in einer Tabelle abgelegt sind, verursachen die Ausnahmen von der Filterregel keinen zusätzlichen Bearbeitungsaufwand im Encoder.

Als Beispiel der Wirkungsweise des JBIG-Auflösungsreduktionsalgorithmus dient die folgende Abbildung (eine gerasterte bi-level Version des Logos der `XV` Bildverarbeitungssoftware von John Bradley):



Die folgende linke um den Faktor zwei auflösungsreduzierte Version entstand durch einfaches Entfernen jeder zweiten Spalte und Zeile aus dem Originalbild, während die rechte Version mit Hilfe des eben beschriebenen Verfahrens erzeugt wurde:

Während im linken Bild durch Aliasing-Effekte die Rasterung in eine völlig willkürliche Struktur übergeht bleiben die Grauschattierungen beim JBIG-Verfahren weitgehend erhalten und werden durch ein neues Rastermuster repräsentiert. Auch wenn die für die Lesbarkeit der Schrift erforderliche Auflösung bereits unterschritten wurde, so lassen sich doch

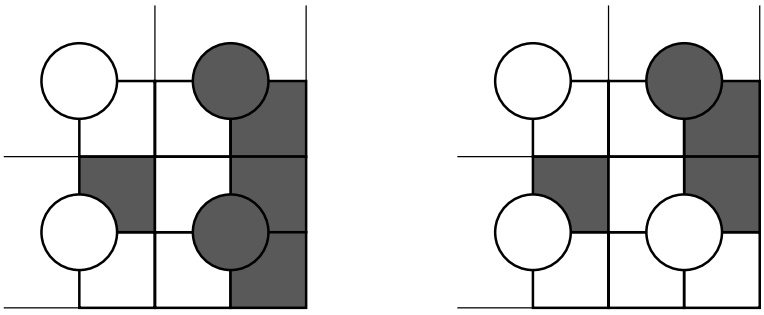
zahlreiche Buchstaben bei Anwendung des JBIG-Verfahrens immer noch deutlich besser erkennen als im linken Bild.

3.5 Deterministische Vorhersage

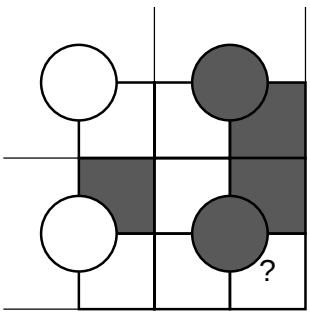
Bei Auflösungsstufen größer 0, bei denen in den Kontext Bildpunkte von zwei Auflösungsstufen einfließen, hilft die Information aus der im Decoder bereits vorhandenen niedrigerauflösenden Version mit, die Entropie der neuen Bildpunkte zu verringern. In bestimmten Situationen kann aber bereits durch Kenntnis des Auflösungsreduktionsalgorithmus ohne jede Informationsübertragung durch den arithmetischen Encoder der Inhalt des nächsten Bildpunkts bestimmt werden. Diese deterministische Vorhersage von Bildpunkten anhand der vorangegangenen Bildpunkte der gleichen und der nächst niedrigeren Auflösungsstufe wird im JBIG-Verfahren durchgeführt. Vorhersagbare Bildpunkte werden vor dem arithmetischen Encoder aus der Folge von zu codierenden Symbolen entfernt und hinter dem arithmetischen Decoder wieder eingefügt.

Wird als Auflösungsalgorithmus beispielsweise nur jede Zeile und Spalte mit einer ungeraden Nummer entfernt, so haben im höherauflösenden Bild alle Punkte mit gerader Zeilen- und Spaltennummer den gleichen Wert wie der entsprechende Bildpunkt des niedrigerauflösenden Bilds und damit müßte etwa 1/4 der Bildpunkte in höherauflösenden Schichten nicht codiert werden.

Bei dem wesentlich komplexeren in JBIG eingesetzten Auflösungsreduktionsalgorithmus ist die deterministische Vorhersage nicht ganz so einfach, aber dennoch praktikabel. In den beiden folgenden Tabelleneinträgen des Auflösungsreduktionsalgorithmus führt eine Änderung im Bildpunkt unten rechts in der höherauflösenden Schicht auch im rechten unteren Bildpunkt der niedrigerauflösenden Schicht zu einer Änderung:



Daraus läßt sich folgern, daß in der Situation



in welcher der mit „?“ gekennzeichnete Bildpunkt als nächstes decodiert werden soll dieser nur schwarz sein kann, denn wäre er weiß, so hätte der Auflösungsreduktionsalgorithmus im JBIG-Encoder auch den entsprechenden Bildpunkt der niedrigeren Auflösungsstufe auf weiß gesetzt.

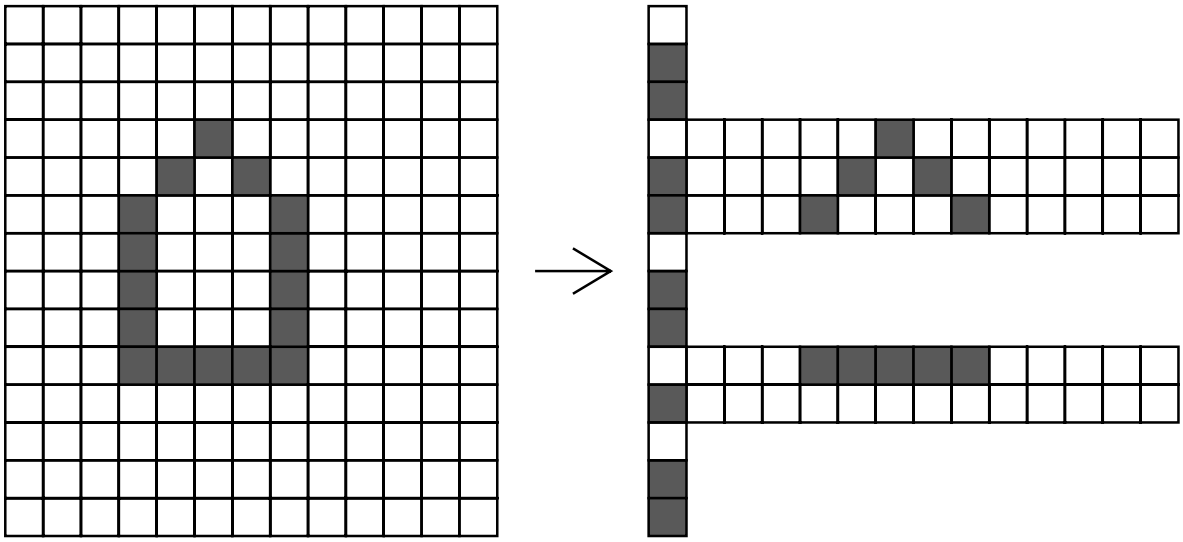
Der deterministische Vorhersage-Algorithmus besteht aus einer Tabelle in der für alle Situationen eingetragen ist, ob der nächste Bildpunkt mit 1 oder mit 0 vorhergesagt werden kann oder ob er codiert werden muß. Abhängig von der jeweiligen Lage des zu codierenden Bildpunkts relativ zum entsprechenden Punkt der niedrigeren Auflösungsstufe wird dabei ein Kontext mit 8, 9, 11 oder 12 Nachbarpunkten eingesetzt, weshalb diese Tabelle $2^8+2^9+2^{11}+2^{12} = 6912$ Einträge hat. Der Decoder kennt als Voreinstellung die zum JBIG-Auflösungsreduktionsalgorithmus passende Vorhersagetabelle, sie muß daher nicht mit den Daten übertragen werden. Wird jedoch im Encoder ein anderer Algorithmus zur Erzeugung der Auflösungsstufen eingesetzt, so kann dem JBIG-Datenstrom eine dazu passende Vorhersagetabelle beigelegt werden. Diese ist 1728 Bytes lang, da für jeden Tabelleneintrag zwei Bits belegt werden.

3.6 Typische Vorhersage

Ein weiterer Algorithmus entfernt im Encoder noch vor der deterministischen Vorhersage Bildpunkte, die sich effizient anderweitig codieren lassen und fügt sie im Decoder an entsprechender Stelle wieder ein. Dieses Verfahren der „typischen Vorhersage“ kann insbesondere die Bearbeitungszeit eines JBIG-Systems deutlich beschleunigen, da es in Dokumenten mit großen homogenen Bildflächen einen großen Teil der Bildpunkte mit wesentlich weniger Rechenaufwand als der arithmetische Coder bearbeiten kann. Die typische Vorhersage wird sowohl in der niedrigsten Auflösungsstufe 0 als auch in höheren differentiellen Schichten eingesetzt, jedoch handelt es sich dabei um zwei völlig unterschiedliche Algorithmen.

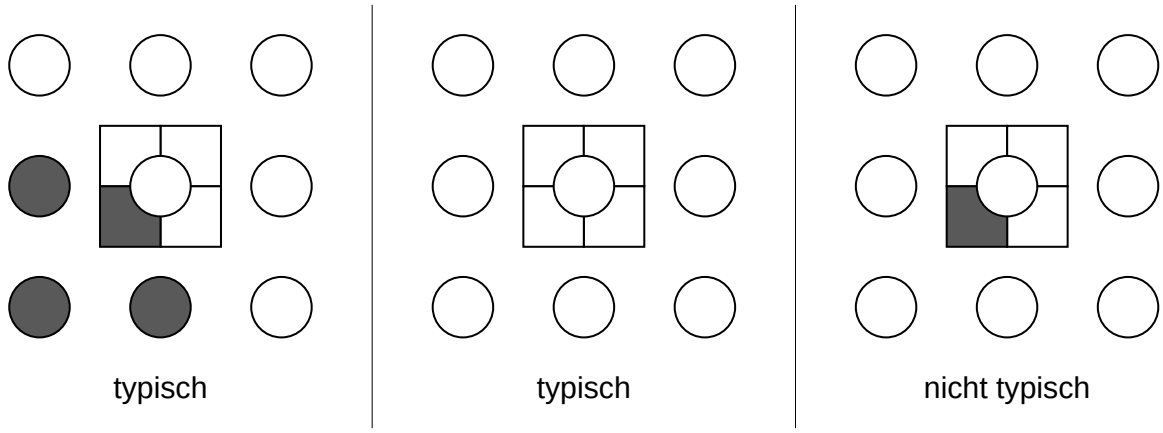
In Schicht 0 dient die typische Vorhersage dazu, aufeinanderfolgende identische Zeilen nur einmal zu codieren. Gerade bei kurzen Geschäftsbriefen ist oft fast die Hälfte der Bildzeilen völlig weiß. Um Zeilen kennzeichnen zu können, die identisch mit der Vorgängerzeile sind, wird vor dem Beginn jeder Bildzeile ein zusätzliches Symbol an den arithmetischen Encoder geschickt. Dieser Pseudobildpunkt entspricht keinem Punkt des Bilds und der zu seiner Codierung eingesetzte Kontext ist fest vorgegeben. Eine Bildzeile wird in Schicht 0 als „typisch“ bezeichnet, wenn ihr Inhalt identisch mit der unmittelbar darüberliegenden Zeile ist. Die Zeile über der ersten Zeile ist per Definition weiß. Die Bildpunkte von typischen Zeilen werden nicht an den arithmetischen Encoder gegeben, denn der Decoder kann sie durch Kopieren der letzten Zeile wiederherstellen.

Die Pseudopixel kennzeichnen nicht direkt, ob eine Zeile typisch ist oder nicht. Für jede Zeile bestimmt der Encoder zunächst ein Bit das auf 1 gesetzt wird, wenn die betreffende Zeile nicht mit der vorangegangenen Zeile übereinstimmt. Für die Zeile über der ersten Zeile wird dieses Bit auf 1 gesetzt. Nun wird, falls sich dieses Bit in einer Zeile vom entsprechenden Bit der vorangegangenen Zeile unterscheidet, der Pseudopixel auf weiß gesetzt, andernfalls auf schwarz. Die folgende Abbildung verdeutlicht, wie die typische Vorhersage identische Zeilen eliminiert und diese Information in einer zusätzlichen Spalte von Pseudopixeln mit überträgt:

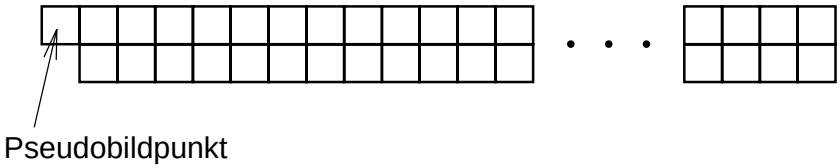


Empirische Untersuchungen haben gezeigt, daß eine Quelle, die aussagt, ob eine Zeile identisch zu ihrer Vorgängerzeile ist, eine höhere Entropie hat als eine Quelle, die nur den Übergang zwischen typischen und nicht-typischen Zeilen angibt. Dies ist darauf zurückzuführen, daß identische Zeilen in der Regel jeweils mehrfach auftreten. Daher werden bei JBIG nur die Übergänge zwischen Bereichen mit identischen bzw. verschiedenen Zeilen als Symbole unterschieden und übertragen. Als Kontext wird für den Pseudobildpunkt ein Bitmuster eingesetzt, das normalerweise selten auftritt und in der Regel für Weiß eine geringere Wahrscheinlichkeit ergibt. Die Wahrscheinlichkeit für das Pseudopixel wird über den normalen Kontextmechanismus ermittelt, um bei Hardwarerealisierungen keinen zusätzlichen Aufwand im arithmetischen Coder zu verursachen.

Völlig anders arbeitet dagegen die typische Vorhersage in differentiellen Auflösungsstufen, bei denen dem Algorithmus als Informationsquelle auch die nächst niedrigere Auflösungsstufe zur Verfügung steht. In der Regel ist zu erwarten, daß falls in der niedrigeren Auflösungsstufe ein Bildpunkt die gleiche Farbe hat wie alle seine acht direkten Nachbarpunkte, dann auch alle vier diesem Bildpunkt entsprechenden Punkte der höheren Schicht diese Farbe haben. Bildpunkte, die eine Ausnahme dieser Regelbeobachtung bilden werden als nicht-typisch bezeichnet. Ein Bildpunkt ist typisch, wenn aus der Tatsache, daß er und seine acht Nachbarn die gleiche Farbe haben folgt, daß auch die ihm entsprechenden vier Punkte der höheren Auflösungsschicht die gleiche Farbe haben. Die folgende Abbildung zeigt zwei typische und einen nicht-typischen Bildpunkt mit seinen acht Nachbarn und die entsprechenden vier hochauflösenden Punkte:



Eine ganze Bildzeile der niedrigeren Auflösungsstufe wird als typisch bezeichnet, wenn alle Bildpunkte in ihr typisch sind. Zu jeder Bildzeile niedrigerer Auflösung gehören zwei höherauflösende Bildzeilen. Vor dem ersten Bildpunkt dieses höherauflösenden Zeilenpaars wird ein Pseudopixel mit festgelegtem Kontext eingefügt:



Dieser Pseudopixel wird weiß codiert, wenn die dem Zeilenpaar entsprechende Zeile niedrigerer Auflösung typisch ist, ansonsten schwarz. Wenn eine Zeile typisch ist, dann werden im höherauflösenden Zeilenpaar diejenigen Bildpunkt-Vierergruppen nicht an den arithmetischen Encoder übergeben, die einem niedrigauflösenden Bildpunkt entsprechen welcher die gleiche Farbe wie seine acht Nachbarn hat. Da der Decoder durch den Pseudobildpunkt weiß, daß alle Punkte der diesem Zeilenpaar entsprechenden niedrigerauflösenden Zeile typisch sind, kann er selbst erkennen, wann durch die Farbgleichheit der Nachbarn die vier höherauflösenden Bildpunkte ebenfalls diese Farbe haben müssen. Dadurch lassen sich bei progressiver Codierung große Flächen homogener Farbe bereits durch die niedrigen Auflösungsstufen weitgehend codieren. Bei nicht-typischen Zeilen mit schwarzem Pseudopixel tritt die typische Vorhersage nicht in Kraft und es werden alle Bildpunkte des Zeilenpaars an den arithmetischen Encoder übergeben.

3.7 Struktur einer JBIG bi-level Bildeinheit

Der JBIG-Standard [ITU93a] spezifiziert neben dem Kompressionsverfahren auch die exakte Syntax eines Datenstroms. Dieser wird als bi-level Bildeinheit (*bi-level image entity, BIE*) bezeichnet und enthält neben den Ergebnisdaten des arithmetischen Encoders auch Begleitdaten, die zur Verarbeitung im Decoder wichtig sind. Eine BIE kann ein komplettes komprimiertes Bild enthalten, jedoch können auch die einzelnen Auflösungsstufen über mehrere BIEs verteilt werden. Ein typisches Szenario dafür wäre eine Dokumentendatenbank, deren Datensätze zwei Felder für eine Bildschirmversion und eine hochauflösende

Laserdruckerversion eines Textes enthalten. Das Dokument könnte in drei Auflösungsstufen zu 75, 150 und 300 dpi codiert werden. Auflösungsschicht 0 würde als BIE im ersten Feld abgelegt werden, und die Schichten 1 und 2 zusammen als eine BIE im 2. Feld.

Jede bi-level Bildeinheit beginnt mit einem Kopf in Form einer 20 Byte langen festgelegten Datenstruktur. Diese enthält die folgenden Angaben:

- Die Nummer D_L der niedrigsten in dieser BIE abgelegten Auflösungsstufe ($0 \leq D_L \leq 255$).
- Die Nummer D der höchsten in dieser BIE abgelegten Auflösungsstufe ($D_L \leq D \leq 255$).
- Die Breite X_D und Höhe Y_D der höchsten in diesem Bild vorhandenen Auflösungsstufe ($1 \leq X_D, Y_D \leq 2^{32} - 1$).
- Die Anzahl P der in diesem Bild vorhandenen Bitebenen ($1 \leq P \leq 255$).
- Die Höhe L_0 eines Streifens in der Auflösungsstufe 0 ($1 \leq L_0 \leq 2^{32} - 1$).
- Der maximale Versatz M_X (horizontal) und M_Y (vertikal) des frei verschiebbaren Kontextmuster-Elements ($0 \leq M_X \leq 127, 0 \leq M_Y \leq 255$).
- Die vier Bits `HITOLO`, `SEQ`, `ILEAVE` und `SMID`, die die Anordnung der SDEs innerhalb der bi-level Bildeinheit angeben.
- Die fünf Bits `LRLTWO`, `VLENGTH`, `TPDON`, `TPBON` und `DPON`, mit denen der Encoder mitteilt, ob die 2-zeilige Variante des Kontextmusters für Schicht 0, das `NEWLEN`-Segment, die typische Vorhersage in differentiellen Schichten, die typische Vorhersage in Schicht 0 sowie die deterministische Vorhersage eingesetzt wurden.
- Die beiden Bits `DPPRIV` und `DPLAST`, die angeben, ob statt der Standardtabelle für die deterministische Vorhersage eine eigene Tabelle eingesetzt werden soll, und falls ja, ob diese mitgeliefert wird oder aus der vorangegangenen BIE übernommen werden muß.

Auf den 20 Byte langen Kopf folgt falls die Bedingung $\texttt{DPON} \wedge \texttt{DPPRIV} \wedge \neg\texttt{DPLAST}$ erfüllt ist die 1728 Byte lange Tabelle für die deterministische Vorhersage. Der Rest der BIE besteht aus $(D - D_L + 1) \cdot P \cdot S$ *stripe data entities (SDEs)* von denen jeder einen Streifen aus einer Auflösungsstufe und einer Bitebene codiert enthält (wobei $S = \lceil Y_0/L_0 \rceil$ die Anzahl der Streifen ist). Zwischen den SDEs können sich weitere Segmente (*floating marker segments*) befinden. Jedes dieser Segmente beginnt zur Unterscheidung von SDE-Bytes mit 0xff gefolgt von einem Kenn-Byte, das den Typ des Segments angibt. Diese Segmente enthalten Informationen, die nicht vom arithmetischen Encoder codiert werden und die eventuell noch nicht vor Beginn der Kompression zur Verfügung stehen und daher nicht mit in den Kopf aufgenommen werden konnten. Folgende Segmenttypen existieren:

- `ABORT`: bricht die eine BIE ab.
- `ATMOVE`: teilt dem Decoder mit, in welchen Zeilen in der folgenden SDE das frei bewegliche Kontextelement wohin verschoben wird.
- `COMMENT`: enthält beliebige Anwenderdaten, die der Decoder ignoriert.
- `NEWLEN`: teilt dem Decoder ein neues Y_D mit, das kürzer ist als das ursprünglich im Kopf angegebene. Dies kann beispielsweise bei in Faxgeräten üblichen Scannern

eingesetzt werden, die schon während das Dokument noch eingezogen wird mit der Übertragung von Daten beginnen und die genaue Länge des eingezogenen Papiers erst kurz vor Ende der Übertragung feststellen können.

- `RESERVE`: reserviert für Encoder, die nur einen Teil des JBIG-Algorithmus in Hardware implementieren und in diesem Segment Hinweise an die weiterverarbeitende Software übergeben müssen.
- `SDNORM`: markiert das Ende einer SDE.
- `SDRST`: markiert das Ende einer SDE und initialisiert vor der nächsten SDE im Decoder die Modell-Statistiken sowie andere Parameter (sollte normalerweise nicht eingesetzt werden).
- `STUFF`: ersetzt einen innerhalb einer SDE vorkommenden 0xff-Wert, damit dieser nicht mit dem Anfang eines neuen Segments verwechselt wird (*byte stuffing*).

Die SDEs bestehen aus den Ausgaben des arithmetischen Encoders, wobei 0x00 Bytes am Ende entfernt werden dürfen. Abgeschlossen wird jede SDE durch ein unmittelbar darauffolgendes `SDNORM`- oder `SDRST`-Segment. Der Decoder weiß anhand der im Kopf angegebenen Bildgröße, wieviele Symbole in einer SDE zu erwarten sind und hängt gegebenenfalls selbstständig 0x00-Bytes an, falls zum Decodieren mehr Werte benötigt werden.

Um den JBIG-Standard so universell wie möglich anwendbar zu machen, wurden in die BIE nur die Begleitinformationen mit aufgenommen, die unbedingt zum Decodieren erforderlich sind. In der Regel wird eine BIE alleine nicht bereits als eigenständiges Dateiformat eingesetzt werden können, sondern muß mit einer zusätzlichen Kopfdatenstruktur versehen in ein anderweitig spezifiziertes Dateiformat eingebettet werden. Von einem komfortablen Dateiformat würde man je nach Anwendung beispielsweise die folgenden Eigenschaften fordern, die durch die in der BIE enthaltenen Informationen nicht abgedeckt sind:

- Einige charakteristische Anfangsbytes (*magic code*), die es erlauben, den Dateityp zu identifizieren bzw. verifizieren.
- Angaben über die Bedeutung der verschiedenen Bitebenen. Der JBIG-Standard gibt nur an, daß bei einer einzelnen Bitebene das Symbol 0 der Hintergrundfarbe (in der Regel weiß) und das Symbol 1 der Vordergrundfarbe (in der Regel schwarz) entsprechen soll. Mit mehreren Bitebenen könnten z.B. verschiedene Farben oder Graustufen benutzt werden, jedoch muß anderweitig festgelegt und übertragen werden, welche Farbe welchem Bitmuster entspricht.
- Angaben über die Auflösung und Größe des Originalbilds.
- Textuelle bibliographische Angaben wie Name des Autors, Titel des Dokuments und Erfassungsdatum um diese in Auswahlmenüs entsprechend anbieten zu können.
- Ein Inhaltsverzeichnis und ein Index mit Stichwörtern um eine textuelle Suche zu ermöglichen. Dieser Index könnte automatisch mit OCR-Algorithmen erstellt worden sein, da bei Volltextsuchen die Fehlerrate von Zeichenerkennungsverfahren nicht so sehr ins Gewicht fällt wie bei der Archivierung des kompletten Texts.
- Eine Prüfsumme, digitale Unterschrift oder bei Langzeitarchivierung eventuell sogar Vorwärtsfehlerkorrekturinformation.

Der JBIG-Standard beschreibt nur einen komplexen und vielseitig einsetzbaren Baustein einer für ein Dokumentenverwaltungssystem geeigneten Datenstruktur, kein Universal-Bilddateiformat.

4 Implementation des JBIG-Verfahrens

Zu den Anforderungen, welche an die im Rahmen dieser Arbeit erstellte wiederverwendbare Implementation des JBIG-Algorithmus gestellt wurden gehört neben voller Kompatibilität zum Standard und hoher Ausführungsgeschwindigkeit auch die Einsetzbarkeit in interaktiven Systemen und die schonende Nutzung des Hauptspeichers. Die Benutzerschnittstelle wurde so gestaltet, daß der Anwendungsprogrammierer nicht mit den internen Strukturen des JBIG-Datenstroms vertraut sein muß. Die Implementation erfolgte streng kompatibel in Standard C [ISO90] und sollte daher auf einer sehr großen Zahl von Systemen ohne Modifikationen einsetzbar sein, sofern genügend Hauptspeicher für die zum Teil umfangreichen Eingangs- und Ausgangsdaten vorhanden ist. Falls der GNU `gcc` Compiler eingesetzt wird, so werden automatisch einige nur auf diesem Compiler verfügbare Optimierungen aktiviert (*inline functions*). Getestet wurde die JBIG-Bibliothek unter verschiedenen UNIX Systemen (Linux 1.2, SunOS 4.1.3, HP-UX) mit den jeweiligen C Compilern. Ferner wurde berichtet, daß auch Tests auf Apple Macintosh Rechnern und unter Microsoft Windows 3.1 erfolgreich waren. Zur Qualitätssicherung und um Kompatibilitätstests zu vereinfachen wurden alle durchgeführten Tests in Form des `test_codec` Hilfsprogramms wiederholbar gestaltet, was sich insbesondere beim Test und bei der Fehlersuche auf Architekturen die dem Entwickler nicht selbst zur Verfügung standen (z.B. Apple Macintosh) als sehr vorteilhaft erwies.

Die Implementation steht auf dem Internet zur Verfügung mittels *anonymous ftp* vom Rechner `ftp.uni-erlangen.de` im Unterverzeichnis `pub/doc/ISO/JBIG/`. Die Datei `jbig-kit-`$x.y$`.tar.gz` dort enthält den Quellcode zusammen mit Dokumentation und Hilfsprogrammen.

4.1 Gestaltung der Schnittstelle

Die Implementation besteht aus den Dateien `jbig.c`, `jbig.h` sowie `jbig_tables.c`. Zusammengebunden ergeben diese Dateien auf UNIX-Systemen eine `libjbig.a` Bibliothek, welche nur die für den Benutzer wichtigen Funktionen in den Namensraum des Anwendungsprogramms exportiert. Da die Programmiersprache C über keine Möglichkeiten zur Strukturierung des Namensraums der deklarierten Bezeichner kennt (wie etwa das `namespace`-Konstrukt in C++ oder `uses` in Ada) beginnen alle in `jbig.h` deklarierten Symbole mit `jbg_` oder `JBG_` um Kollisionen mit Namen aus anderen Modulen eines Anwendungsprogramms zu vermeiden. Die Sprache C wurde für die Implementation ausgewählt, da in dieser Sprache die meisten möglichen Anwendungen, in denen die JBIG-Bibliothek eingesetzt werden kann geschrieben wurden. Da heute alle verfügbaren C Compiler dem ANSI/ISO C Standard entsprechen wurde auf ältere Sprachdialekte keine Rücksicht genommen. Obwohl C keine Sprachmittel zur objekt-orientierten Strukturierung von Programmsystemen vorsieht, ist sowohl die Benutzerschnittstelle der JBIG-Bibliothek als auch die Schnittstelle

zwischen dem arithmetischen Coder und dem restlichen Algorithmus klassenbasiert gestaltet.

Die vier verwendeten Klassen repräsentieren den JBIG Encoder und Decoder, sowie den arithmetischen Encoder und Decoder. Diese Klassen werden durch die Strukturen `struct jbg_enc_state` und `struct jbg_dec_state` sowie durch `struct jbg_arenc_state` und `struct jbg_ardec_state` repräsentiert. Zu diesen Klassen gehören jeweils die Konstruktoren `jbg_enc_init()` und `jbg_dec_init()` sowie `arith_enc_init()` und `arith_dec_init()`.

Alle exportierten Funktionen arbeiten auf Variablen vom Typ `struct jbg_enc_state` bzw. `struct jbg_dec_state`, in denen der gesamte Zustand des Encoders und Decoders enthalten ist. Da keine globalen oder als `static` deklarierte Variablen eingesetzt werden (außer zu Testzwecken), sind alle Funktionen vollständig reentrant, d.h. eine Funktion kann jederzeit unterbrochen und von einem zweiten Aktivitätsträger betreten werden. Dadurch eignet sich diese Implementation insbesondere für Programmierumgebungen mit mehreren Aktivitätsträgern im gleichen Adressraum (*multithreading*), wie sie bei komfortablen interaktiven Systemen häufig benutzt werden. Die einzigen Funktionen aus der C Standardbibliothek, auf die zugegriffen wird sind `malloc()`, `realloc()` und `free()` zur Speicherverwaltung. Da diese Funktionen auf vielen Systemen nicht reentrant gestaltet sind, wurden ihre Aufrufe in den Funktionen `checked_malloc()`, `checked_realloc()` und `checked_free()` zusammengefaßt, so daß an dieser Stelle bei Bedarf entsprechende Synchronisationsmechanismen wie etwa eine binäre Semaphore leicht eingebaut werden können.

Den Konstruktoren werden als Parameter jeweils nur die unbedingt notwendigen Angaben übergeben. Beim Encoder sind das die zu codierenden Bilddaten, die Bildgröße sowie die Anzahl der Bitebenen. Ferner wird dem Encoder eine Rückruf-Funktion zur Übergabe der Ergebnis-Daten mitgeteilt. Die Daten des erzeugten JBIG-Datenstroms (BIE) werden wenn nicht unbedingt notwendig nicht zwischengepuffert, sondern sobald verfügbar in verschieden großen Blöcken mit Hilfe der Rückruf-Funktion an die Anwendung übergeben, die auf diese Weise noch während der Kompression mit dem Abspeichern in einer Datei oder der Übertragung auf einer Netzverbindung beginnen kann. Dieser Weg der Datenrückgabe spart Zeit und unnötigen Pufferspeicherplatz. Neben der Rückruf-Funktion wird dem Encoder auch ein Zeigerparameter übergeben, den dieser jeweils an die Rückruf-Funktion weitergibt. Auch diese Möglichkeit unterstützt den Einsatz der Bibliothek mit mehreren Aktivitätsträgern, da so eine einzige Rückruf-Funktion sehr einfach zwischen verschiedenen gleichzeitig ablaufenden Codiervorgängen unterscheiden und die Daten jeweils an die richtige Stelle weiterleiten kann. Der Konstruktor für den Decoder erfordert keine weiteren Parameter.

Andere Angaben und Optionen werden vom Konstruktor mit sinnvollen Voreinstellungen versehen. Mit weiteren Funktionsaufrufen können diese Werte vor Beginn des Codiervorgangs eingestellt und verändert werden. Dazu gehören beim Encoder die Anzahl der Auflösungsschichten, die entweder direkt mit `jbg_enc_layers()` oder durch Angabe einer Maximalgröße für Schicht 0 mit `jbg_enc_lrlmax()` bestimmt werden kann. Beim Einstellen der Anzahl der Auflösungsstufen ermitteln die entsprechenden Funktionen wiederum geeignete Werte für davon abhängige Parameter wie die Anzahl der Streifen in die das Bild unterteilt wird. Dabei werden vom Standard über den möglichen Wertebereich der Parameter hinaus empfohlenen Grenzen zur Sicherstellung der Kompatibilität automatisch berücksichtigt. Mit `jbg_enc_options()` können weitere Parameter wie der maximale Verschiebungsbereich des frei beweglichen Kontextmusterelements sowie die Streifenhöhe

direkt bestimmt werden und `jbg_enc_lrange()` erlaubt es, bi-level Bildeinheiten zu erzeugen, die nicht alle Auflösungsschichten enthalten. Der eigentliche Codiervorgang wird mit `jbg_enc_out()` angestoßen. Mit einem weiteren Aufruf von `jbg_enc_lrange()` und `jbg_enc_out()` können noch verbleibende Auflösungsschichten ausgegeben werden, ohne daß unnötige Doppelberechnungen ausgeführt werden müssen. Abschließend kann mit dem Destruktor `jbg_enc_free()` der restliche belegte Speicherplatz freigegeben werden, den der Encoder eventuell benötigt hat, um SDEs intern umzuordnen.

Nach der Initialisierung können dem Decoder mit der Funktion `jbg_dec_in()` beliebig aufgeteilte Blöcke einer bi-level Bildeinheit übergeben werden. Der Rückgabewert verrät, ob die BIE bereits vollständig erhalten wurde, ob noch weitere Daten notwendig sind, oder ob ein Fehler aufgetreten ist. Wenn noch weitere Daten benötigt werden, so werden vom Decoder stets alle bis dahin übergebenen Daten übernommen und müssen nicht von der Anwendung gepuffert und später nocheinmal übergeben werden. Nur wenn über das Ende einer BIE hinaus mehr Daten als notwendig geliefert wurden, dann kann sich der Anwender im Parameter `cnt` mitteilen lassen, wie viele Bytes der übergebenen Daten wirklich gelesen wurde. Der Decoder kann das Ende einer BIE selbstständig erkennen und mitteilen. Sollte ein Fehler aufgetreten sein, so gibt der Rückgabewert die Ursache dafür an. Damit das Anwendungsprogramm den Fehlergrund einfach an den Benutzer mitteilen kann, läßt sich der Fehlercode mit der Funktion `jbg_strerror()` in einen lesbaren Fehlertext übersetzen, wobei die Sprache und der gewünschte Zeichensatz angegeben werden können. Derzeit sind Fehlermeldungen nur auf Englisch (ASCII) und Deutsch (ISO 8859-1 und UTF-8) verfügbar, weitere Sprachen und Zeichensätze lassen sich bei Bedarf sehr leicht integrieren. Vor Beginn der Decodierung kann mit `jbg_dec_maxsize()` angegeben werden, welche Größe des Ergebnisbilds bereits zu groß wäre, so daß der Decoder bei einer ohnehin beschränkten Darstellungsfläche, beispielsweise auf einem Bildschirm, falls mehrere Auflösungsschichten vorliegen rechtzeitig den Decodiervorgang unterbricht. Durch einen erneuten Aufruf von `jbg_dec_in()` können anschließend auch die restlichen Auflösungsebenen decodiert werden, falls sich beispielsweise der Benutzer für einen Laserdruckerabzug des am Bildschirm betrachteten Dokuments entscheidet, ohne daß das Anwendungsprogramm dazu nocheinmal die Daten der bereits bearbeiteten Schichten übergeben muß. Mit den Funktionen `jbg_dec_getwidth()`, `jbg_dec_getheight()`, `jbg_dec_getplanes()`, `jbg_dec_getsize()` und schließlich `jbg_dec_getimage()` können von Decoder die empfangenen Daten und ihre Parameter übernommen werden. Am Ende können mit dem Destruktor `jbg_dec_free()` die von Decoder belegten Speicherbereiche wieder freigegeben werden.

Sowohl der Encoder als auch der Decoder benutzen das gleiche Datenformat für die Bilddaten. In einem Byte werden jeweils acht unmittelbar nebeneinanderliegende Bildpunkte einer Bitebene abgelegt. Im höchstwertigsten Bit steckt der sich in der Achtergruppe am weitesten links befindliche Pixel. Zeilen sind jeweils eine ganze Zahl von Bytes lang, wozu gegebenenfalls am rechten Bildrand bis zu sieben Bildpunkte mit dem Wert 0 angefügt werden müssen. Eine Bitebene von x Bildpunkten Höhe und y Bildpunkten Breite benötigt daher $y\lceil x/8\rceil$ Bytes Speicherplatz. Übergeben wird ein Feld mit Zeigern auf die ersten Bytes der einzelnen Bitebenen.

Eine wesentlich detailliertere und aktuelle Beschreibung der Schnittstelle in englischer Sprache, weitere Nutzungshinweise und praktische Anwendungsbeispiele befinden sich in der Datei `jbig.doc`.

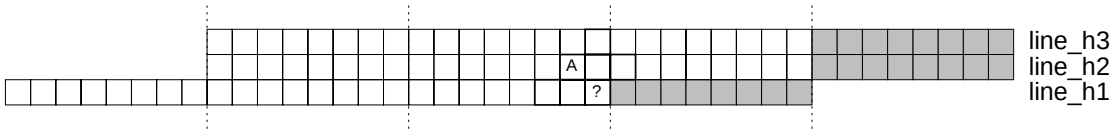
4.2 Realisierung des Encoders

Die Funktion `jbg_enc_out()` prüft zunächst die eingestellten Parameter, erzeugt die Kopfdatenstruktur der BIE und übergibt diese der Rückruf-Funktion. Anschließend durchläuft sie in drei generischen Schleifen alle auszugebenden SDEs. Welche Schleife welche der drei Variablen für Bitebene, Streifen und Auflösungsschicht bestimmt, wird von drei Parameter Bits festgelegt. Das vierte `HITOLO`-Bit bestimmt, ob die Schleife über die Auflösungsschichten aufsteigend oder fallend durchlaufen wird. Das dreidimensionale Feld `sde` dient dazu, eventuell zwischenzuspeichernde SDEs aufzunehmen. Die Einträge in diesem Feld sind entweder Zeiger auf eine doppeltverkettete Listenstruktur vom Typ `struct jbg_buf`, oder aber haben einen der beiden Werte `SDE_DONE` bzw. `SDE_TODO`, mit denen gekennzeichnet wird, ob die entsprechende Streifeneinheit bereits ausgegeben und der Puffer geräumt wurde, bzw. ob die SDE noch nicht errechnet wurde.

In der innersten der drei Schleifen wird die Funktion `output_sde()` aufgerufen. Ihre Aufgabe ist es, die entsprechende SDE, sowie alle zuvor zu erzeugenden SDEs zu codieren und anschließend die gewünschte SDE an die Rückruf-Funktion zu übergeben. Neben dem Speicher, der die Originalbilddaten enthält benutzt der Encoder nur noch einen zweiten Speicher mit einem viertel der Größe des ersten für die jeweilig andere Auflösungsschicht. Falls mehr als eine Schicht codiert werden soll, müssen immer zwei aufeinanderfolgende Schichten sich im Speicher befinden, da das Kontextmuster in differentiellen Schichten ebenso wie die deterministische und typische Vorhersage auf Bildpunkte von beiden Auflösungen zurückgreift. Da nur zwei Bildspeicher benutzt werden sollen, müssen bevor eine SDE aus einer Schicht d codiert werden kann alle Schichten von $d+1$ bis D vollständig codiert sein. Denn in den beiden Bildspeichern müssen für die Codierung der Schicht d die beiden Schichten $d-1$ und d vorhanden sein. Höherauflösende Versionen sind zu diesem Zeitpunkt bereits überschrieben worden. Also kann es (bei `HITOLO` $= 0$) vorkommen, daß der Auftrag an `output_sde()` zunächst die Codierung und Zwischenspeicherung einer Reihe von anderen SDEs auslöst, bevor die Auflösungsreduktionen durchgeführt werden können, nach denen schließlich erst die gewünschte Schicht codiert werden kann. Um den verwendeten Speicherplatz zu minimieren werden nur komprimierte SDEs und keine unkomprimierten Bilddaten zwischengepuffert. Dies verkompliziert zwar etwas die in `output_sde()` implementierte Ablaufsteuerung, stellt aber sicher, daß sich die eingesetzte Speichermenge für Puffer am möglichen Minimum bewegt.

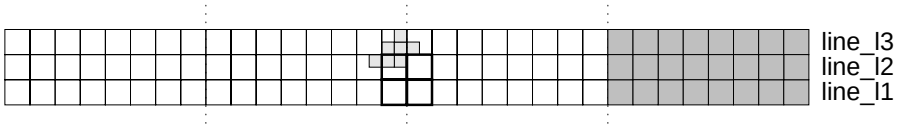
Die Funktion `output_sde()` stellt in den Bildpuffern die passenden Auflösungsstufen bereit und ruft die Funktion `encode_sde()` auf, in der die eigentliche Kompression durch deterministische und typische Vorhersage sowie Ermittlung des Kontexts und Übergabe der Symbolfolge an den arithmetischen Coder stattfindet. Der für die Effizienz der Implementation entscheidende Bereich ist die innerste Schleife über alle Punkte einer Bildzeile. In diesem Bereich müssen sowohl die deterministische Vorhersage, als auch die Ermittlung des Kontexts des aktuellen als nächstes zu codierenden Bildpunkts schnellen Zugriff auf die Nachbarbildpunkte haben.

Dazu wurde die folgende Lösung gewählt. In insgesamt sechs 32 Bit großen Registern werden die Bildpunkte der unmittelbaren Umgebung gehalten. Die folgende Abbildung zeigt die Lage der Bits in den drei Registern für die höherauflösenden Nachbarpunkte relativ zum Kontextmuster für die differentiellen Schichten:



Um diese Werte zu aktualisieren, ist pro Bildpunkt in den drei Registern `line_h1`, `line_h2` und `line_h3` nur jeweils eine Linksschiebeoperation notwendig und alle acht Bildpunkte muß in die in der Abbildung grau unterlegten Bits ein Byte aus den Ausgangsbilddaten übernommen werden. Die Bits, in welche die Daten geschrieben werden, enthalten nicht immer gültige Daten, da die Schiebeoperation die rechts entstehenden Lücken mit 0 füllt und alle acht Schritte enthalten diese Bereiche nur Nullen. Daher dürfen diese Bits nicht Teil des Kontextmusters werden. Die Lage des Kontextmusters innerhalb dieser Register wurde so gewählt, daß eine Initialisierung besonders einfach ist. Wenn das Kontextmuster links oder rechts über das Bild hinausragt, so schreibt der JBIG-Standard vor, daß Null-Bits einzusetzten sind. Zu Beginn jeder Zeile werden die beiden nicht grau unterlegten Bytes rechts des mit „?“ markierten zu codierenden Pixels in `line_h2` und `line_h3` mit den ersten beiden Bytes der beiden Zeilen über der aktuellen Zeile gefüllt. Bei jeder Spaltennummer, die durch 8 teilbar ist (also auch bei Spalte 0), werden die grauen Bereiche gefüllt und vor der Codierung jedes Bildpunkts werden alle drei Register um ein Bit nach links verschoben. So gelangt beim ersten Durchlauf das höchstwertigste Bit des ersten Bytes der aktuellen Zeile bereits in das „?“-Bit. In `line_h1` schließt links an den Bereich der die neuen Daten aufnimmt sofort der erste für die Codierung relevante Bildpunkt an. Dadurch werden in diesem Register sehr viele Bits links des aktuellen Bildpunkts gespeichert, die alle vom frei beweglichen Kontextmusterelement genutzt werden können, das sich bei dieser Implementation um bis zu 23 Punkte weit nach links bewegen kann. Der Standard empfiehlt allerdings nicht mehr als 16 Punkte horizontalen Abstand zu benuzten und keine vertikale Verschiebung einzusetzen, um die Kompatibilität mit allen Minimalimplementationen zu gewährleisten. In `line_h2` wird ein Bildpunkt rechts der aktuellen Position benötigt, so daß es sich empfiehlt das ganze Register nach rechts zu verschieben. Um die Initialisierung einfach zu gestalten und damit die Abfrage, wann Bytes nachgeladen werden müssen, in der innersten Schleife nur einmal durchgeführt werden muß, wurde dieses Register gleich um ein Byte verschoben. Das Register `line_h3` ist ebenfalls verschoben, da es beim 3-zeiligen Kontextmuster für Schicht 0 ein Kontextmusterelement enthält.

Nach einem ähnlichen Verfahren werden auch in den drei Registern `line_l1`, `line_l2` und `line_l3` die Nachbarpunkte der niedrigeren Auflösungsschicht gehalten. Die folgende Abbildung zeigt die Lage der vier Elemente des Kontextmusters in diesen Registern sowie zum Vergleich die Lage der entsprechenden höherauflösenden Kontextelemente:



Die gezeigte Situation ist nur eine von vier möglichen Phasenlagen der höherauflösenden Elemente relativ zu den Punkten in diesen Registern. Die drei Register `line_l1`, `line_l2` und `line_l3` werden nur bei jedem zweiten codierten Bildpunkt um eine Position nach links verschoben und nur bei jedem 16. wird ein neues Byte in die grau unterlegten Bereiche

geladen. Für die Ermittlung des Kontexts sind nur die Register `line_l1` und `line_l2` notwendig, aber für die deterministische und typische Vorhersage wird auch die niedriger-auflösende Zeile darüber in `line_l3` benötigt.

Darüber hinaus ist in der innersten Schleife noch etwas Aufwand für die Sonderbehandlung an den Bildrändern notwendig. Zur Ermittlung der passenden Einträge in der Tabelle für die deterministische Vorhersage werden nur die entsprechenden Bits aus den 6 Registern durch bitweises „und“ ausmaskiert, nicht-überlappend zusammengeschoben und durch bitweises „oder“ in eine einzige Zahl verwandelt. Beim Laden der Tabelle für die deterministische Vorhersage wird diese mit der Funktion `jbg_dppriv2int()` so umgeordnet, daß mit der so ermittelten Indexzahl direkt auf einen einzelnen Tabelleneintrag zugegriffen werden kann. Dieser gibt an, ob mit der arithmetischen Codierung dieses Bildpunkts fortzufahren ist oder ob die deterministische Vorhersage im Decoder den Bildpunkt selbst bestimmen kann. Auch die Nummer des Kontexts wird auf die gleiche Art aus den Registern gewonnen und dient im arithmetischen Encoder als Index in eine Tabelle in welcher der aktuelle Schätzautomatenzustand für diesen Kontext enthalten ist sowie welches Bit gerade dem wahrscheinlicheren Symbol entspricht.

Falls durch $M_X > 0$ die Adaption des Kontextmusters zugelassen ist, wird in einem Feld `c` für alle möglichen Positionen des beweglichen Kontextelements ständig mitprotokolliert, wie groß die Korrelation zum Wert des zu codierenden Bildpunkts ist. Für jeden Streifen werden, wie in Anhang C von [ITU93a] vorgeschlagen 2048 Bildpunkte abgewartet und anschließend wird zu Beginn der nächsten Zeile anhand von verschiedenen Schwellwerten entschieden, ob das Element bewegt werden soll. In diesem Fall wird nach Vollendung des Streifens ein `ATMOVE`-Segment den codierten SDE-Bytes vorangestellt, das dem Decoder mitteilt, in welcher Zeile das Kontextelement seine Position wohin wechselt. Um einen Kompatibilitätstest mit einer Referenzimplementation und deren im Standard beschriebenen Beispielergebnissen zuzulassen, wurde auch die Option implementiert, die Bewegung des Kontextelements auf die erste Zeile des nächsten Streifens zu verschieben, da andere Implementationen eventuell keine Möglichkeit haben, nachträglich noch ein `ATMOVE`-Segment vor eine SDE zu stellen.

Für die typische Vorhersage wird jeweils vor jedem zweiten Zeilendurchlauf in einer eigenen Zeilenschleife getestet, ob es sich um eine typische Zeile handelt und es wird der arithmetische Encoder zur Ausgabe des entsprechenden Pseudobildpunkts aufgerufen. Die Registervariablen werden dabei wie zuvor beschrieben eingesetzt. Um im Falle einer typischen Zeile nicht vor der eigentlichen Codierung nocheinmal testen zu müssen, ob der nächste Pixel unter einem typischen Pixel niedrigerer Auflösung liegt, werden die Ergebnisse in einem Feld `tp` für ein Zeilenpaar gespeichert.

Die Implementation des arithmetischen Encoders wurde in eine eigene Funktion `arith_encode()` ausgelagert, da sie auch an anderer Stelle zur Codierung der Pseudobildpunkte für die typische Vorhersage benötigt wird und da die inneren Schleifen getrennt für die Schicht 0 und die differentiellen Schichten realisiert sind. Wird zum Übersetzen allerdings der GNU `gcc` Compiler eingesetzt, so wird in den inneren Schleifen der arithmetische Encoder nicht als Unterprogramm aufgerufen, sondern textuell eingesetzt (*inline function*), was eine Geschwindigkeitssteigerung von etwa 2 % erzielt. Die Implementation des arithmetischen Encoders selbst entspricht weitgehend den entsprechenden Flußdiagrammen in [ITU93a]. Dieser Algorithmus wurde nur geringfügig optimiert indem der Schätzautomatenzustand und die Information welches Bit das derzeit wahrscheinlichere Symbol ist so in

einem Byte untergebracht wurden, daß die Anzahl der notwendigen Assemblerinstruktionen etwas reduziert wurde. Ferner wurde ein Mechanismus implementiert, der am Ende des Codierergebnisses auftauchende 0x00-Werte entfernt.

4.3 Realisierung des Decoders

Das Hauptmerkmal der Decoder-Implementation ist, daß Daten die dem Decoder mit `jbg_dec_in()` übergeben werden sofort verarbeitet und so weit wie möglich in die Datenbereiche für das Ergebnisbild übernommen werden. Daher ist es auf der Basis dieser Implementation möglich, Anwendungen zu entwickeln, die noch während die Daten für ein Bild eintreffen alle durch bereits vorhandene Daten festgelegten Bildpunkte anzeigen. Diese Eigenschaft ist insbesondere bei Zugriffen auf Bilddaten über sehr langsame Netzwerkverbindungen wichtig. Bei Zugriffssoftware auf das *World Wide Web* auf dem Internet zählt beispielsweise als wichtiges Qualitätskriterium, ob Bilder bereits während der zum Teil sehr lange dauernden Übertragung schon teilweise sichtbar sind.

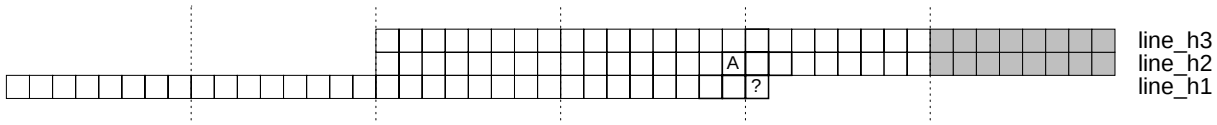
Die von den Flußdiagrammen des JBIG-Standards abweichende Struktur des arithmetischen Decoders ist dadurch begründet, daß der Decoder in der Lage sein muß, alle verfügbaren Bytes aufzulesen, auch wenn die vorliegende Information noch nicht zur Decodierung des nächsten Symbols ausreicht. Dem arithmetischen Decoder wird eine eventuell unvollständige Bytefolge übergeben, wobei die Variablen `pscd_ptr` und `pscd_end` auf Anfang und Ende dieser Folge zeigen. Nach einem Aufruf von `arith_decode()` kann sich der arithmetische Decoder in einem der folgenden Zustände befinden:

- `JBG_OK`: Es wurde ein Symbol erfolgreich decodiert und es sind vermutlich noch weitere Symbole in der restlichen vorhandenen Bytefolge enthalten.
- `JBG_READY`: Das Ende der codierten Bytefolge wurde gefunden, es sind keine weiteren Bytes mehr notwendig, aber es können noch beliebig viele weitere Symbole decodiert werden, da der Decoder fehlende Bytes selbst mit 0x00 auffüllt. Die Funktion, die `arith_decode()` aufruft um das nächste Symbol abzufragen, muß selbst wissen, wann das letzte Symbol erreicht wurde. Der Zeiger `pscd_ptr` zeigt auf das nächste noch nicht verarbeitete Byte 0xff, welches schon zum nachfolgenden Segment gehört.
- `JBG_MORE`: Der arithmetische Decoder hat alle sich zwischen `pscd_ptr` und `pscd_end` befindlichen Bytes aufgebraucht. Es konnte kein neues Symbol decodiert werden, sondern es müssen erst weitere Bytes übergeben werden.
- `JBG_MARKER`: Wie bei `JBG_MORE` kann kein Symbol mehr mangels Eingabebytes decodiert werden, jedoch war dieses Mal das letzte Byte ein 0xff-Wert. Dies könnte falls das nächste neue Byte den Wert 0x00 hat wegen des *byte stuffing* Verfahrens eine normale Ausgabe des Encoders sein. Wenn aber ein anderes Byte folgt, so ist dies der Beginn eines neuen Segments und damit das Ende der Ausgaben des arithmetischen Encoders. In diesem Fall würde der arithmetische Decoder in den Zustand `JBG_READY` übergehen. Da in einem Fall das letzte Byte dem arithmetischen Decoder gehört, im anderen Fall aber nicht, aber noch nicht klar ist, welcher Fall eingetreten ist, wird der 0xff-Wert nicht übernommen und muß in jedem Fall nocheinmal mit den neuen Daten übergeben werden.

Im Gegensatz zum Implementationsvorschlag für den arithmetischen Decoder in [ITU93a] wurde die Renormalisation vor den eigentlichen Decodierschritt verlegt, da nur die erfolgreiche Renormalisation sicherstellen kann, daß ein weiteres Symbol extrahiert werden kann. Da ansonsten, falls die Decodierung erst später als noch nicht möglich erkannt werden würde, bereits geänderte Werte restauriert werden müßten, spart diese Vorgehensweise weitere Abfragen in der zeitkritischen innersten Schleife ein. In diesem Zusammenhang mußte auch der Initialisierungsmechanismus des Decoders modifiziert werden, da durch die gewünschte Schnittstelle des Decoders die ersten Bytes des Datenstroms noch nicht zum Zeitpunkt des Konstruktoraufrufs zur Verfügung stehen. Es gelang durch Einführung der Variable `startup`, die nur im Falle einer notwendigen Renormalisierung überprüft wird, diese Semantik effizient zu realisieren. Eine weitere Besonderheit der durchgeführten Implementation des arithmetischen Decoders ist, daß die Behandlung der 0xff-Werte (*byte stuffing*) in der Renormalisierungsroutine und nicht in einem eigenen Schleifendurchlauf außerhalb des Decoders durchgeführt wird, was ebenfalls die ohnehin große Zustandsvielfalt in anderen Teilen des Decoders reduziert.

Die prinzipielle Vorgehensweise zur Ermittlung des Kontexts und bei differentiellen Schichten auch der deterministischen Vorhersage ist ähnlich wie bei der Implementation des Encoders. Diese Komponenten befinden sich in `decode_pscd()`. Die Abkürzung PSCD (engl. *protected stripe coded data*) bezeichnet im Standard die Ausgabe des arithmetischen Encoders (SCD, *stripe coded data*), bei der die 0xff-Werte mit einem nachfolgenden 0x00-Wert vor Verwechslung mit dem Anfang eines neuen Segments geschützt wurden.

Da bei jedem Aufruf des arithmetischen Decoders sich herausstellen könnte, daß die Prozedur nicht fortgesetzt werden kann, da weitere Eingabebytes fehlen, dürfen vor den Aufrufen des arithmetischen Decoders keine Zustandsänderungen durchgeführt werden, die sonst eventuell rückgängig gemacht werden müßten. Daher ist das Layout der Register die die Nachbarbildpunkte enthalten so ausgelegt, daß erst nach dem erfolgreichen Decodieren eines Bildpunkts eine Schiebeoperation erforderlich ist:



Da der Inhalt von `line_h1` durch das Decodierergebnis beim Linksschieben Bit für Bit eingetragen wird, sind Bits aus `line_h1` nur links des zu decodierenden Bildpunkts notwendig. Daher kann `line_h1` um 16 Punkte gegenüber `line_h2` und `line_h1` verschoben sein und dadurch ist es möglich, im Decoder das bewegliche Kontextmusterelement um bis zu 32 Bildpunkte nach links zu verschieben. Die Lage der Bildpunkte in den Registern `line_l1`, `line_l2` und `line_l3` entspricht bis auf einen Bildpunkt (da hier erst nach dem Decodieren nach links geschoben wird) der im Encoder.

Die Register werden als Teil des Decoderzustands in `struct jbg_dec_state` mitgesichert. Die Schleifen in `decode_pscd()` sind alle ohne Initialisierung der Laufvariablen realisiert und so angelegt, daß jederzeit wieder mit dem vorangegangenen Wert der Variablen bei einem neuen Aufruf fortgefahren werden kann. Die Schleifenvariablen werden erst jeweils nach Ende der Schleife auf den Startwert zurückgesetzt.

Die Funktion `jbg_dec_in()` nimmt vom Anwendungsprogramm beliebige Bruchstücke der BIE entgegen. Sie sammelt die ersten 20 Bytes, bis der Kopf vollständig gelesen werden

kann, anschließend wird falls vorhanden eine private Tabelle zur deterministischen Vorhersage in einen Puffer eingelesen und in das effizientere interne Darstellungsformat umgewandelt. Wenn dieser Teil erledigt ist werden die übergebenen Daten, je nachdem, an welcher Stelle und in welchem Zustand sich der Decoder gerade befindet, entweder an `decode_pscd()` übergeben oder in einem kleinen Puffer angesammelt, bis ein Segment (z.B. vom Typ `ATMOVE` oder `NEWLEN`) komplett eingetroffen ist und bearbeitet werden kann. Der Aufbau von `jbg_dec_in()` ist ebenso wie schon bei `decode_pscd()` und `arith_decode()` von der Vorgabe geprägt, daß an jeder Stelle der übergebene Datenstrom zuende sein kann und die Prozedur sofort verlassen werden muß. Damit dürfen keine nicht-wiederholbaren Aktionen begonnen worden sein, bevor nicht alle dazu notwendigen Bytes empfangen worden sind und jedes einzelne bereits gelesene Byte muß im Gesamtzustand des Decoders festgehalten werden, da es vom Anwendungsprogramm nicht nocheinmal übergeben wird.

Dieser Aufwand ist notwendig um es Anwendungen zu erlauben, nach jedem einzelnen angekommenen Byte bereits die darin enthaltenen Daten anzuzeigen. Darüber hinaus erlaubt es diese Vorgehensweise bei der Implementation des Decoders dem Anwendungsprogramm, mehrere Bilder gleichzeitig zu laden und zu decodieren, ohne daß mehrere Aktivitätsträger vom Betriebssystem angefordert werden müssen. Bei vielen existierenden Betriebssystemen sind leider nicht mehrere Aktivitätsträger (*threads*) im gleichen Adressraum verfügbar und getrennte Prozesse für einzelne Bilder verursachen einen hohen Programmieraufwand zur Interprozeß-Kommunikation. Darüber hinaus erschwert sich bei mehreren Prozessen auch die Kommunikation mit den oft nicht für *multithreading* ausgelegten graphischen Benutzeroberflächen. Daher ist der eingeschlagene Weg, wenn auch für den Entwickler der JBIG-Bibliothek aufwendig, für den Anwendungsprogrammierer eine bequeme Möglichkeit mit einem einzigen Aktivitätsträger komfortable und effiziente Benutzerschnittstellen zu gestalten.

5 Bewertung des JBIG-Verfahrens

Im folgenden sind einige Angaben über die mit JBIG auf einer Reihe von Beispielbildern erreichbaren Kompressionsverhältnisse zusammengestellt. Dadurch soll interessierten Anwendern die Möglichkeit gegeben werden, die mit JBIG erzielbare Speichernutzung und Zugriffsgeschwindigkeit abzuschätzen, und das Verfahren mit anderen Alternativen zu vergleichen.

Der erste Satz von Testbildern mit den Dateinamen `ccitt1` bis `ccitt8` wurde bereits von den Entwicklern des Gruppe 3 Fax-Algorithmus zur Bewertung und Optimierung eingesetzt. Er dient auch heute noch als ein wichtiger Maßstab für die Leistungsfähigkeit von bi-level Kompressionsverfahren. Die offiziellen Testdokumente wurden nur auf Papier veröffentlicht, weshalb verschiedene Forscher mit unterschiedlichen selbstdigitalisierten Pixeldateien arbeiten. Die verwendeten Dateien stammen vom Internet Server `ftp.funet.fi` und sind vermutlich die verbreitetste Variante. Sie wurden mit einer Auflösung von 8 Pixel/mm (etwa 200 dpi) wie beim Fax-System üblich abgetastet und haben eine Größe von 1728×2376 Punkten. Eine unkomprimierte Rohdatei im PBM-Format mit 8 Bildpunkten pro Byte und einem kleinen 13 Byte langen Dateikopf ist 513 229 Byte lang. Die Testseiten bestehen aus einem kurzen Geschäftsbrief, einer handgezeichneten elektronischen Schaltung, einem Formular, einer eng bedruckten Buchseite, einer Buchseite mit Text sowie Formeln und Diagrammen, einer Buchseite mit einem Graphen aus Meßwerten, einer eng mit japanischen Schriftzeichen bedruckten Seite sowie einer handgeschriebenen Notiz.

Die folgende Tabelle gibt für alle acht Bilder die Länge der Ergebnisdatei, den Kompressionsfaktor sowie die auf einem mit 66 MHz getakteten Intel 486/DX2 PC unter Linux 1.2 gemessene Ausführungszeit an. Alle diese Angaben werden außer für das JBIG-Verfahren zum Vergleich auch für das Gruppe 3 Fax-Verfahren sowie für den GNU `gzip` *deflate* Algorithmus angewendet auf eine PBM-Datei angegeben. Bei der JBIG-Messung wurde nur eine Auflösungsschicht codiert, bei `gzip` (Version 1.2.4) wurden keine algorithmischen Optionen aktiviert und für die Gruppe 3 Fax Messung wurde das Programm `pbmtog3` von Paul Haeberli aus der `netpbm`-Software verwendet. Bei den Zeitmessungen befand sich die Ausgangsdatei bereits im Hauptspeicher und wurde nicht abgespeichert, so daß Massenspeicherzugriffe keinen Einfluß hatten.

	JBIG			Gruppe 3 Fax			`gzip`		
Datei	Bytes	Faktor	Zeit [s]	Bytes	Faktor	Zeit [s]	Bytes	Faktor	Zeit [s]
`ccitt1`	14761	34.8	3.23	37425	13.7	2.69	30991	16.6	1.45
`ccitt2`	8591	59.7	5.51	34368	15.0	2.64	27792	18.5	1.57
`ccitt3`	22052	23.3	5.53	65035	7.9	2.90	47733	10.8	1.74
`ccitt4`	54369	9.4	5.27	108076	4.7	3.30	103031	5.0	2.64
`ccitt5`	25917	19.8	5.72	68318	7.5	2.94	56480	9.1	2.02
`ccitt6`	12611	40.7	5.25	51172	10.0	2.80	31879	16.1	1.65
`ccitt7`	56327	9.1	5.78	106422	4.8	3.28	114717	4.5	5.05
`ccitt8`	14310	35.9	5.61	62802	8.2	2.86	44097	11.6	2.31
Mittel	26117	19.7	5.24	66702	7.7	2.93	57090	9.0	2.30

Das Gruppe 4 ISDN-Fax Verfahren erzielt auf diesen Daten einen Kompressionsfaktor von 15.5, der in Hardware implementierte ABIC Algorithmus erreicht 18.8 [Arp88] und der Q-Coder aus [Pen88b] den Faktor 19.0.

Der Encoder der im Rahmen dieser Arbeit durchgeführten JBIG-Implementation benötigt um einen im Mittel 26117 Byte langen Datenstrom zu erstellen 5.24 Sekunden auf einem PC der heute mittleren Leistungsklasse. Das entspricht einer Datenrate von 40 kbit/s, liegt also etwas über der mit Telefonmodems zur Verfügung stehenden Kapazität. Mit etwas leistungsfähigeren Prozessoren wie sie in etwa einem Jahr bereits auch in der unteren Leistungsklasse von Personal Computern zu erwarten sind wird ein ISDN B-Kanal mit 64 kbit/s vollständig ausgenutzt. Für preisgünstigere ISDN-Fax-Anwendungen wird bei den derzeitigen Kosten für entsprechend leistungsfähige Prozessoren eher eine Hardwareimplementation des JBIG-Verfahrens in Frage kommen. Der JBIG-Decoder hatte bei diesen Tests eine dem Encoder vergleichbare Ausführungszeit von im Mittel 5.07 s.

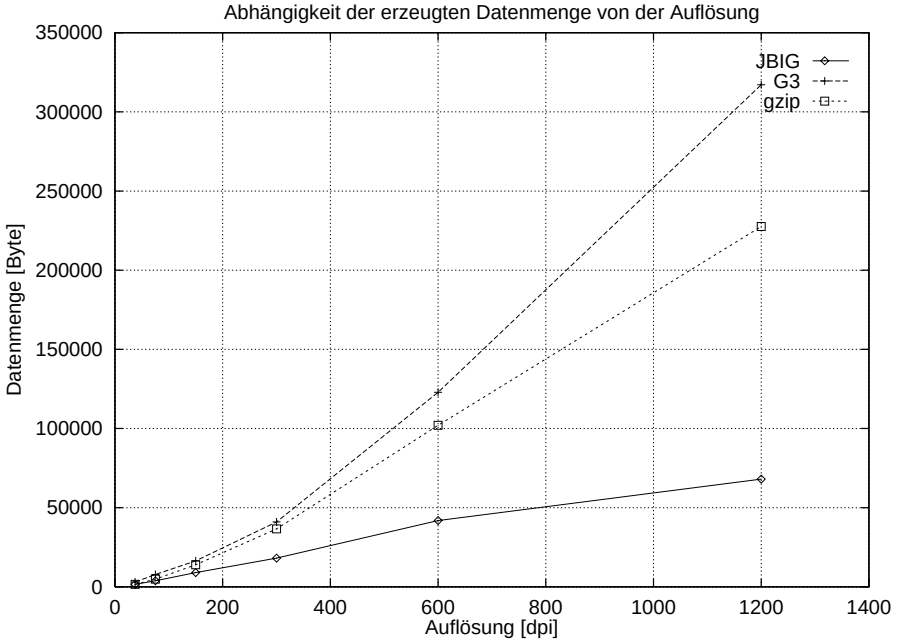
In der obigen Messung wurde nur eine einzige Schicht codiert, wie bei den Vergleichsverfahren. In der folgenden Tabelle werden die Bilder in vier Auflösungsschichten codiert, so daß die kleinste 216×297 Bildpunkte groß ist und damit leicht auf jedem Bildschirm dargestellt werden kann. Dadurch reduziert sich der Kompressionsfaktor etwas und die Rechenzeit steigt an.

	progressives JBIG		
Datei	Bytes	Faktor	Zeit [s]
`ccitt1`	16830	30.5	8.79
`ccitt2`	8958	57.3	8.43
`ccitt3`	23642	21.7	10.33
`ccitt4`	58748	8.7	11.78
`ccitt5`	28092	18.3	10.21
`ccitt6`	13503	38.0	9.29
`ccitt7`	60640	8.5	11.74
`ccitt8`	15135	33.9	9.24
Mittel	28193	18.2	9.98

Der Decoder benötigt für die Auflösungsschicht 0 (216×297 Pixel) im Mittel 124 ms, für die ersten beiden Schichten (432×594) 542 ms, für die ersten drei Schichten (864×1188) 1.96 s und für alle vier Schichten (1728×2376) 7.04 s. Auf der Referenzhardware ist der Decoder somit in der Lage, Bilder in Bildschirmfenstergröße in Sekundenbruchteilen zu bearbeiten und die Geschwindigkeit für die volle Auflösung liegt mit etwa 8 Seiten pro Minute im Bereich der Ausgabegeschwindigkeit von Laserdruckern.

Die in den vorangegangenen Messungen benutzten Testdaten wurden mit der im Fax-Verkehr üblichen Auflösung von 8 Punkte/mm (etwa 200 dpi) aufgenommen. Für eine qualitativ hochwertige Darstellung ist jedoch die 1.5 bis 3-fache Auflösung (300 oder 600 dpi) notwendig, da erst dann für das Auge die Bildpunkte kaum mehr wahrnehmbar werden. Wie die folgenden Diagramme zeigen, hängt der Kompressionsfaktor deutlich von der Auflösung ab. Die Testdaten bestehen aus einigen Absätzen Text ohne Zeichnungen, Formeln oder viel weißem Freiraum aus einem Buch. Dieser für die Archivierung von meist engbedrucktem wissenschaftlichen Schriftgut typische Testdatensatz wurde mit 150, 300, 600

und 1200 dpi Auflösung digitalisiert. Darüber hinaus wurden 75 und 37.5 dpi Versionen mit dem JBIG-Auflösungsreduktionsverfahren erzeugt. Dabei ergaben sich für den gleichen Bildausschnitt in unterschiedlichen Auflösungen folgende Längen der komprimierten Ergebnisdaten:

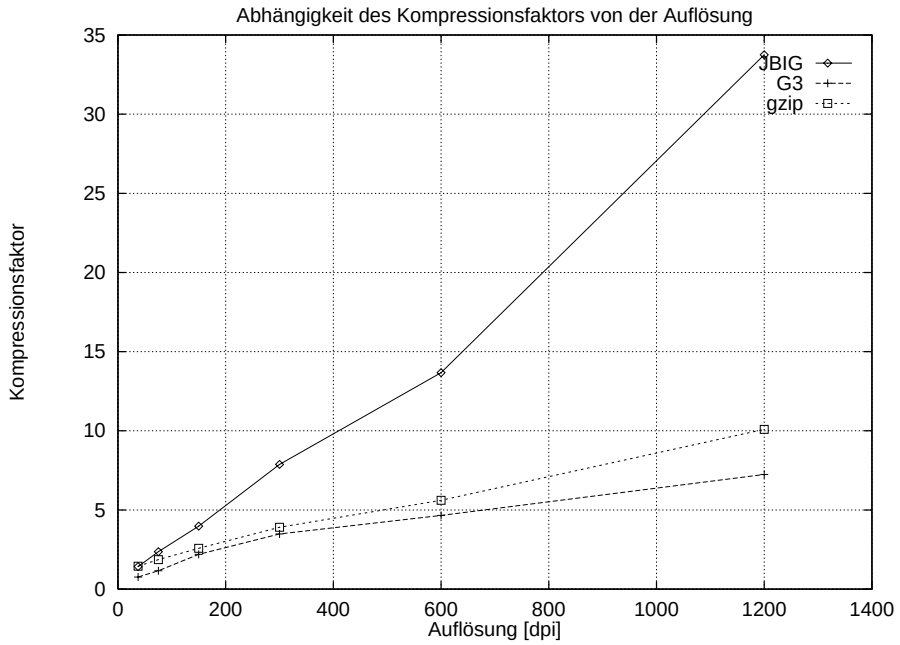


Während die Länge der Rohdaten $O(r^2)$ mit der Auflösung r steigt, wächst die Länge beim Gruppe 3 Algorithmus nicht mehr quadratisch, aber immer noch deutlich stärker als linear an. Nicht wesentlich stärker als linear wächst die Länge des `gzip`-Ergebnisses an und bei sehr geringen Auflösungen ist `gzip` sogar das beste Verfahren. Letzteres ist darauf zurückzuführen, daß bei JBIG während die Schätzung der Wahrscheinlichkeiten noch läuft bereits Daten ausgegeben werden, wohingegen `gzip` jeweils zuerst für einen ganzen Datenblock die Statistiken ermittelt und anschließend erst der Huffman-Encoder in Aktion tritt. Bei größeren Auflösungen ist das JBIG-Verfahren jedoch deutlich überlegen und das Anwachsen der Länge seines Datenstroms kann scheinbar gut mit $O(r)$ beschrieben werden.

Es ist schwer eine fundierte Begründung für dieses Verhalten der Algorithmen anzugeben, da das Anwachsen der anfallenden Datenmenge eng mit der Natur der zu komprimierenden Bilder verknüpft ist. Werden beispielsweise nicht-komprimierbare Rauschbilder in denen jeder Bildpunkt die Entropie 1 aufweist an die Algorithmen gegeben, so wachsen die Längen bei allen Verfahren mit $O(r^2)$. Der Gruppe 3 Algorithmus betrachtet die einzelnen Zeilen unabhängig voneinander und da die Zeilenzahl linear mit der Auflösung steigt wachsen seine Datenlängen schon mindestens proportional zu r. Ähnlich kann bei `gzip` argumentiert werden, wo aber gelegentlich auch auf Daten aus der vorangegangenen Zeile zurückgegriffen werden kann. JBIG dagegen kann sehr gut große weiße und schwarze Flächen codieren und das Hauptdatenvolumen dürfte an den Rändern dieser Flächen auftreten. Die Länge des Rands eines nicht-fraktalen Objekts wie etwa einem Buchstaben wächst jedoch nur proportional mit der Größe des Objekts, weshalb ein in etwa linearer Anstieg bei nicht sehr verrauschten Vorlagen plausibel erscheint.

Noch deutlicher wird die Überlegenheit des JBIG-Verfahrens bei einer Darstellung in der

statt der Länge der Ergebnisdaten der Kompressionsfaktor, also der Quotient aus Länge der Roh- und Ergebnisdaten, über der Auflösung aufgetragen ist:



Diese Darstellung demonstriert deutlich, daß der vergleichsweise große Implementations- und Rechenaufwand des JBIG-Verfahrens sich in erster Linie für hochauflösende Dokumentenverarbeitung über 200 dpi lohnt, während zur Codierung von sehr kleinen piktogrammartigen Bildschirmdarstellungen anderen Verfahren wie `gzip` und dem darauf beruhenden PNG-Graphikdateiformat der Vorzug zu geben ist.

Die folgenden Testbeispiele spiegeln die zu erwartenden Kompressionsfaktoren bei der Archivierung von wissenschaftlichem Schriftgut besser wieder als die 8 CCITT Fax-Testseiten. Es handelt sich dabei durchweg um die bei wissenschaftlichen Veröffentlichungen und technischen Texten üblichen dichtbedruckten Buch- und Zeitschriftenseiten mit 300 dpi Auflösung. Im einzelnen handelt es sich bei den Testbeispielen um eine Doppelseite aus einem Mathematikbuch [Bro89] (`sci1`), drei aufeinanderfolgende Seiten aus einer Monographie (`sci2`–`sci4`), eine auf das A4-Format verkleinerte Doppelseite aus einem ISO-Standard (`sci5`), eine Seite aus den *Communications of the ACM* (`sci6`) und die erste Seite von [Ziv77] (`sci7`):

	JBIG			Gruppe 3 Fax			`gzip`		
Datei	Bytes	Faktor	Zeit [s]	Bytes	Faktor	Zeit [s]	Bytes	Faktor	Zeit [s]
`sci1`	48369	18.5	10.64	104134	8.6	4.99	104131	8.6	3.72
`sci2`	38869	13.6	5.18	94903	5.6	3.24	82785	6.4	2.72
`sci3`	41751	12.7	4.70	101281	5.2	3.29	88059	6.0	2.73
`sci4`	36939	14.3	5.66	86097	6.2	3.16	75733	7.0	2.60
`sci5`	60789	17.5	12.75	141674	7.5	6.17	127766	8.3	4.33
`sci6`	93370	9.7	10.44	186196	4.9	5.79	179806	5.1	5.35
`sci7`	115396	7.9	11.76	221058	4.1	6.07	217050	4.2	6.11
Mittel	62212	12.3	8.73	133620	5.7	4.67	125047	6.1	3.94

Die JBIG-Ergebnisse der drei mit dem gleichen Zeichensatz gedruckten Seiten in `sci2`, `sci3` und `sci4` sind zusammen 117 559 Byte lang. Werden die drei Bilddateien zu einem einzigen großen Bild aneinandergehängt, so ist die entstehende BIE mit 117089 nur geringfügig kürzer, obwohl bei der Codierung der ersten Bildpunkte von `sci3` und `sci4` die Statistiken des Coders bereits an den Zeichensatz angepaßt sind. Dies demonstriert, daß bei hochauflösenden Bildern die Verluste durch die anfangs unrichtige Abschätzungen der Wahrscheinlichkeiten für die zu codierenden Symbole sehr gering sind. Eine Codierung von mehreren Textseiten in einem Zug oder die Übernahme der Statistiken der vorangegangenen Seite um Einschwingvorgänge in den ersten Zeilen zu vermeiden lohnt sich nicht, zumal dadurch der wahlfreie Zugriff auf einzelne Seiten erschwert wird.

Bemerkenswert erscheint, daß der Kompressionsfaktor auf computererzeugten Pixeldateien deutlich besser ist als auf mit Scannern durch Abtastung von Papiervorlagen gewonnen Bildern. Eine Seite dieser Arbeit mit 300 dpi dargestellt komprimiert um den Faktor 2.0 besser, wenn sie direkt vom Druckertreiber als Graphikdatei erstellt worden ist verglichen mit einer Datei die vom Laserdrucker ausgegeben und anschließend mit einem Scanner wieder eingelesen wurde. Beim G3-Verfahren ist der Unterschied nur ein Faktor 1.2 und bei `gzip` immerhin noch 1.7. Im vom Druckertreiber erzeugten bi-level Bild sind alle gleichen Buchstaben völlig identisch geformt und es tauchen keine Störflecken auf. Das G3-Verfahren kann nur das Fehlen von Störflecken in der computererzeugten Datei ausnutzen, während die anderen adaptiven Verfahren auch die bessere Vorhersagbarkeit der Daten anwenden können.

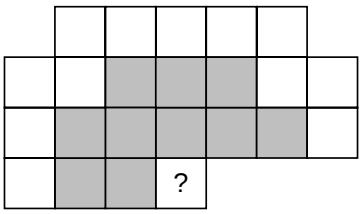
Am Ende dieser Leistungsbetrachtungen noch ein kurzer Blick auf eine bi-level Kompressionsanwendung, die sich nicht mit *textual images* befaßt: Ein mit 600 dpi abgetasteter mit Stempelfarbe auf Papier abgerollter Fingerabruck läßt sich mit JBIG auf etwa 15 kbyte verdichten, während das G3-Verfahren 27 kbyte und `gzip` 25 kbyte benötigen.

6 Ausblick auf modernere Verfahren

Zum Abschluß werden noch einige neuere Ideen zur Kompression von bi-level Bildern vorgestellt, die veröffentlicht wurden, nachdem der JBIG-Standard weitgehend fertiggestellt war.

Der Kompressionserfolg hängt ganz wesentlich davon ab, wie gut sich mit Hilfe des Kontexts der nächste Bildpunkt vorhersagen läßt. Eine denkbare Verbesserung besteht daher darin, den Kontext einfach um weitere Elemente zu vergrößern, so daß die Statistik besser zwischen verschiedenen Situationen unterscheiden kann. Jedoch steigt die Anzahl der Kontexte exponentiell mit der Anzahl der Elemente. Dies hat zur Folge, daß einzelne Vertreter jedes Kontexts wesentlich seltener auftreten und dadurch die Wahrscheinlichkeitsschätzungen wiederum ungenauer werden. Die in JBIG gewählte Kontextgröße von 10 Elementen stellt daher einen Kompromiß dar zwischen der Menge an Information, die in die Statistik einfließt, und der Anzahl der Beispiele auf denen die Schätzung beruht.

Ein in [Mof91] beschriebener Vorschlag, diesen widersprüchlichen Anforderungen an die Kontextgröße zu entgehen, ist die Idee der zweistufigen Kontexte. Es werden zwei verschieden große Kontextmuster eingesetzt, wobei das größere das kleinere enthält. Das von MOFFAT beschriebene 10/22 Kontextmuster hat die folgende Form:



Der innere (hier grau schattierte) Kern aus 10 Elementen stimmt mit dem 3-zeiligen JBIG-Kontextmuster für Schicht 0 überein. Zunächst wird nur mit diesem Muster gearbeitet. Wenn jedoch mit diesem Muster ein Kontext hinreichend oft aufgetreten ist, so daß die Schätzung der Wahrscheinlichkeit eine gewisse Präzision erreicht hat, so wird dieser Kontext ersetzt durch $2^{22-10} = 4096$ neue große Kontexte mit 22 Elementen, die auf den inneren 10 Elementen den gleichen Inhalt haben wie der entfernte kleine Kontext. Diese großen Kontexte übernehmen die geschätzte Wahrscheinlichkeit des kleinen. Es werden nicht auf einen Schlag 2^{10} mögliche kleine Kontexte durch 2^{22} große ersetzt, sondern es wird zu einem Zeitpunkt nur jeweils ein einziger Kontext mit 10 Elementen, der hinreichend oft aufgetreten ist, ersetzt durch 4096 große zu ihm passende. Auf diese Weise werden die großen Kontexte nur in den Situationen eingesetzt, in denen genügend Fälle für eine gute Schätzung auftreten und sie erhalten von Anfang an eine gute Wahrscheinlichkeitsvorgabe. Auf die acht CCITT Fax-Testbilder angewendet ergibt das Verfahren von MOFFAT beispielsweise einen Kompressionsfaktor von 21.4 (zum Vergleich JBIG nur 19.7) und durch weitere Verbesserungen des Verfahrens wurde sogar ein Faktor 22.3 erzielt [Ign95].

Ein wesentlich aufwendigeres Verfahren wird in [Wit92] vorgestellt. Ähnlich wie bei der *Optical Character Recognition* werden ganze Zeichen im Text erkannt und codiert. Zusammenhängende Mengen aus schwarzen Punkten werden erfaßt und in eine Zeichenbibliothek eingetragen. Ähnliche Zeichen werden zu einem einzigen Zeichen zusammengefaßt, das durch Mittelung gebildet wird. Zeichen die nur ein einziges Mal in dieser Zeichenbibliothek

auftauchen werden wieder entfernt. Anschließend wird das Dokument in drei Teilen abgespeichert: Die Zeichenbibliothek, eine Liste der Orte im Bild an denen Zeichen die in die Bibliothek aufgenommen wurden aufgetreten sind, sowie ein Restfehlerbild. Der Decoder kann anhand der Liste und der Bibliothek die gefundenen Zeichen wieder im Bildspeicher an die richtigen Stellen setzten. Wenn das so gewonnene Bild mit einer bildpunktweisen XOR-Operation mit dem Originalbild verknüpft wird, so entsteht das Restfehlerbild. Da der Encoder diese Operation durchführt, kann der Decoder auf das mit der Zeichenbibliothek rekonstruierte Bild mit einer weiteren XOR-Operation das Restfehlerbild anwenden und erhält als Ergebnis das vollständig erhaltene Originalbild.

Im Unterschied zu OCR-Algorithmen wird bei diesem Verfahren nicht versucht, aus der Folge der gefundenen Zeichen eine Textdatei zu erstellen. Informationen über den verwendeten Zeichensatz, die genaue relative Lage der Zeichen zueinander (wie sie zum Verständnis von mathematischen Formeln wichtig ist), Druckbildstörungen, usw. bleiben im Gegensatz zu OCR vollständig erhalten. Dadurch daß die einzelnen Zeichen durch sehr ähnliche mittlere Repräsentanten ersetzt wurden, enthält das Restfehlerbild nur dünne Linien entlang der Ränder der Zeichen, die das bei jedem Zeichen individuelle Quantisierungsrauschen an den Kanten repräsentieren. Außerdem sind im Restfehlerbild noch die Zeichen enthalten, die nur einmal aufgetreten sind und daher nicht in die Zeichenbibliothek aufgenommen wurden. Die Liste der Symbole wird mit einem normalen Textkompressionsalgorithmus verdichtet, die Koordinaten der Symbole untereinander werden abhängig von der Symbolnummer arithmetisch codiert und die Zeichenbibliothek sowie das Restfehlerbild werden mit einem JBIG-ähnlichen Verfahren mit MOFFAT's 10/22 Kontextmuster komprimiert.

Leider hat sich gezeigt, daß das Restfehlerbild sich kaum besser komprimieren läßt als das Originalbild. Ein besseres Ergebnis ist möglich, wenn bei der Codierung des Restfehlerbilds im Kontext auch Nachbarbildpunkte aus dem rekonstruierten Bild zur Verfügung stehen. Im veröffentlichten Beispiel war die Kompression dann um den Faktor 1.4 besser als MOFFAT's Verfahren. Aber auch dann macht das Volumen des Restfehlerbilds etwa 90 % des komprimierten Datenstroms aus.

Wenn in der Bibliothek auch nur einmal auftauchende Zeichen mit aufgenommen werden würden, dann bestünde das Restfehlerbild nur noch aus dem Quantisierungsrauschen an den Kanten der Buchstaben und Linien sowie aus kleinen Störflecken, die zu klein waren um in die Bibliothek aufgenommen zu werden. Da das Restfehlerbild daher in der Regel nur noch aus für den Benutzer irrelevanter Information besteht, könnte es auch einfach weggelassen werden und es ergäbe sich ein verlustbehaftetes Kompressionsverfahren mit wesentlich besseren Kompressionsverhältnissen. An der Entwicklung und Optimierung derartiger verlustbehafteter bi-level Kompressionsalgorithmen wird derzeit gearbeitet. Sie bieten sehr hohe Kompressionsverhältnisse je nach gewünschtem Qualitätsfaktor und erhalten dennoch viel mehr typographische Information als OCR-Verfahren.

Ein Standard wie JBIG kann nur einen Stand der Technik zu einem bestimmten Zeitpunkt festhalten und repräsentiert daher in der Regel nie das leistungsfähigste derzeit bekannte Verfahren. Mit Eigenschaften wie der einfachen VLSI-Realisierbarkeit, der Möglichkeit zur progressiven Codierung, der Implementierbarkeit mit minimalem Speicheraufwand in Faxgeräten und dem größenordnungsmäßig gleichen Rechenaufwand für Kompression und Dekompression erfüllt das JBIG-Verfahren viele für den praktischen Einsatz relevante Randbedingungen, die bei veröffentlichten Verfahren mit etwas besseren Kompressionsergebnissen oft noch nicht gegeben sind.

Glossar

BIE *Bi-level image entity.* Der in [ITU93a] definierte Datenstrom, der das komprimierte Bild enthält. Eine BIE besteht aus einem 20 Byte langen Kopf, einer optionalen Tabelle für die deterministische Vorhersage, sowie aus einer Folge von SDEs und Segmenten.

bi-level Ein *bi-level* Bild besteht aus Bildpunkten, die nur einen von zwei Werten annehmen können, in der Regel *schwarz* und *weiß*.

dpi *Dots per inch.* Auflösung gemessen in Bildpunkten pro 25.4 mm.

JBIG *Joint Bi-level Image Experts Group* ist der informelle Name des Normungsgremiums, das den auch als ISO 11544 veröffentlichten Standard [ITU93a] entwickelt hat. Der offizielle Name dieser Arbeitsgruppe heißt aufgrund eines Vorschlags der schwarz-weiß gepunkteten Delegierten vom Bürokraten-Planeten Faksimilus „ISO/IEC JTC1/SC29/WG1 | CCITT SGVIII.“

Kontext Kombination der Werte von Nachbarpunkten eines Bildpunkts. Die Auswahl der Nachbarpunkte, die in den Kontext eingehen wird als Kontextmuster (engl. *model template*) bezeichnet.

LPS *Less probable symbol.* Siehe auch MPS.

MPS *More probable symbol.* Das entsprechend der aktuellen Wahrscheinlichkeitsschätzung für einen Kontext wahrscheinlichere Symbol unter den beiden Alternativen 0 und 1 (bzw. Hintergrund- und Vordergrundfarbe).

OCR *Optical character recognition.* Ein Verfahren der Mustererkennung, das versucht, eine Bilddatei eines Textes durch Erkennen der einzelnen Zeichen und Wörter in eine Textdatei zu verwandeln.

PSCD *Protected stripe coded data.* Entsteht aus SCD, indem nach jedem 0xff-Byte ein 0x00-Byte eingefügt wird, damit dieses vom Decoder vom Beginn eines beweglichen Segments unterschieden werden kann.

SCD *Stripe coded data.* Die vom arithmetischen Encoder ausgegebenen Bytes für einen Streifen, wobei gegebenenfalls 0x00-Werte am Ende der SCD entfernt worden sind.

SDE Eine SDE (*stripe data entity)* enthält die codierten Informationen für einen Bildstreifen in einer Auflösungsschicht und einer Bitebene. Eine SDE besteht aus PSCD-Bytes gefolgt von entweder einem `SDNORM`- oder `SDRST`-Segment.

Literatur

[Abr63] Abrahamson, N. *Information Theory and Coding*. McGraw-Hill, New York, 1963.

[Arp88] Arps, R. B., et al. *A multi-purpose VLSI chip for adaptive data compression of bilevel images*. IBM Journal of Research and Development, November 1988, Vol. 32, No. 6, pp. 775ff.

[Bro89] Bronstein, I. N., Semendjajew, K. A. *Taschenbuch der Mathematik*. 24. Auflage, Verlag Harri Deutsch, Thun, 1989.

[Den95] Denning, P. J., Rous, B. *The ACM Electronic Publishing Plan*. Communications of the ACM, April 1995, Vol. 38, No. 4, pp. 97–109.

[Gai94] Gailly, Jean-loup. *GNU gzip 1.2.4*. Source Code CD-ROM, Free Software Foundation, Cambridge, December 1994.

[Gir93] Girod, B. *Bildkommunikation*. Skriptum zur Vorlesung, Lehrstuhl für Nachrichtentechnik, Universität Erlangen-Nürnberg, Erlangen, 1993.

[Ing95] Inglis, Stuart `<singlis@borg.cs.waikato.ac.nz>`. Persönliche Mitteilung. University of Waikato, Neuseeland, Juni 1995.

[ISO90] *Programming languages – C*. International Standard ISO 9899:1990, International Organization for Standardization, Geneva, 1990.

[ISO95] *Technical Corrigendum 1 for ISO/IEC 11544 | ITU-T T.82*, ISO/IEC JTC1/SC29/WG1 N165, International Organization for Standardization, Geneva, March 1995.

[ITU93a] *Information Technology – Coded Representation of Picture and Audio Information – Progressive Bi-Level Image Compression*. ITU-T Recommendation T.82, International Telecommunication Union, Geneva, March 1993.

[ITU93b] *Terminal Equipments and Protocols for Telematic Services – Standardization of Group 3 Facsimile Apparatus for Document Transmission*. ITU-T Recommendation T.4, International Telecommunication Union, Geneva, March 1993.

[Mof91] Moffat, A. *Two level context based compression of binary images*. In James A. Storer, J. H. Reif (ed.), Proceedings Data Compression Conference 1991, pp. 382–391, IEEE Computer Society Press, Los Alamitos, 1991.

[Pen88a] Pennebacker, W. B., et al. *An overview of the basic principles of the Q-Coder adaptive binary arithmetic coder*. IBM Journal of Research and Development, November 1988, Vol. 32, No. 6, pp. 717–726.

[Pen88b] Pennebacker, W. B., Mitchell, J. L. *Probability estimation for the Q-Coder*. IBM Journal of Research and Development, November 1988, Vol. 32, No. 6, pp. 737–752.

[Rei94] Reinitzer, H. *Papierzerfall – Kulturzerfall? Über die Probleme der Bewahrung des 'geistigen Erbes'*. Bibliotheksdienst, 28. Jg. (1994), Heft 12, S. 1911–1925.

[Sha48] Shannon, C. E. *A mathematical theory of communication*. Bell System Technical Journal, 1948, Vol. 27, pp. 379–423 and 623–656.

[Ste94] Steinbrink, B. *Gibt es ein Mindesthaltbarkeitsdatum für CDs?*. c't Magazin für Computertechnik, Verlag Heinz Heise, Hannover, April 1994, S. 62–66.

[Top74] Topsøe, F. *Informationstheorie*. Teubner-Verlag, Stuttgart, 1974.

[Wel84] Welch, Terry A. *A Technique for High-Performance Data Compression*. IEEE Computer, Vol. 17, No. 6, June 1984, pp. 8–19.

[Wit87] Witten, I., Neal, R. and Cleary, J. *Arithmetic Coding for Data Compression*. Communications of the ACM, June 1987, Vol. 30, No. 6, pp. 520–538.

[Wit92] Witten, I. H., et al. *Textual Image Compression*. In James A. Storer, Martin Cohn (ed.), Proceedings Data Compression Conference 1992, pp. 42–51, IEEE Computer Society Press, Los Alamitos, 1992.

[Ziv77] Ziv, J., Lempel, A. *A Universal Algorithm for Sequential Data Compression*. IEEE Transactions on Information Theory, May 1977, Vol. 23, No. 3, pp. 337–343.