Visualisierung von verlustlosen Kompressionsalgorithmen

Studienarbeit

von

Oliver Schmid

aus

Forchtenberg

vorgelegt am

Lehrstuhl für Praktische Informatik IV

Prof. Dr. W. Effelsberg

Fakultät für Mathematik und Informatik

Universität Mannheim

Mai 2002

Betreuer: Dipl.Wirtsch.-Inf. Nikolai K. Scheele

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Gliederung 2

2 Verlustlose Kompressionsalgorithmen **3**
2.1 Grundlagen 3
2.2 Huffman-Code 5
2.2.1 Präfixfreiheit des Codes 5
2.2.2 Shannon-Fano-Kodierung 6
2.2.3 Huffman-Kodierung 7
2.3 Lempel-Ziv-Kodierung 11
2.3.1 LZ77-Algorithmus 12
2.3.2 LZSS-Algorithmus und adaptierte Verfahren 12
2.3.3 LZ78-Algorithmus 13
2.3.4 LZW-Algorithmus 13
2.3.5 LZC-Algorithmus 19
2.4 Arithmetische Kodierung 21
2.4.1 Minimalitätseigenschaft 21
2.4.2 Entwicklung der Arithmetischen Kodierung 22
2.4.3 Statische Arithmetische Kodierung 22
2.4.4 Dynamische Arithmetische Kodierung 26

3 Das Applet **29**
3.1 Technische Realisierung 30
3.2 Darstellung der Kompressionsalgorithmen 31
3.2.1 Huffman-Code 31
3.2.2 Lempel-Ziv-Welch-Algorithmus 32
3.2.3 Arithmetische Kodierung 33

4 Zusammenfassung **35**

Kapitel 1

Einleitung

1.1 Motivation

Im Zeitalter der Telekommunikation und globalen Vernetzung müssen immer mehr Daten zwischen Rechnern übertragen werden. Einerseits wird versucht, die Fehlertoleranz in der Datenfernübertragung durch Erzeugen künstlicher Redundanzen zu erhöhen, andererseits wären die Grenzen der Übertragungskapazitäten wesentlich schneller erreicht oder bereits überschritten, wenn die Datenmengen nicht durch effiziente Kompressionsalgorithmen teilweise erheblich reduziert werden könnten.

Mittlerweile wurde eine Vielzahl von Verfahren zur Datenkomprimierung entwickelt, die prinzipiell in zwei Gruppen unterteilt werden können: Bei der verlustbehafteten Kompression ("`lossy compression`") geht ein Teil der Information verloren, das heißt, daß sich die dekodierten Daten von den Originaldaten unterscheiden. Dieser Informationsverlust kann in Kauf genommen werden, wenn es sich bei den zu komprimierenden Daten um Standbilder, Audio- oder Videodateien handelt, da hier die physiologischen und wahrnehmungspsychologischen Eigenschaften der Sinnesorgane, also der Augen und der Ohren, ausgenutzt werden können. So ist es für die menschliche Wahrnehmung charakteristisch, die Bereiche eines Frequenzspektrums unterschiedlich gut zu erfassen. Diese qualitative Wahrnehmungslücke wird bei der Kompression berücksichtigt.

Dafür können wesentlich höhere Kompressionsraten erzielt werden als bei der verlustlosen Kompression ("`lossless compression`"). Bei diesen Verfahren kann das Original jederzeit ohne Qualitätsverlust aus den komprimierten Daten reproduziert werden. Diese Techniken finden in der Regel dann Anwendung, wenn der Verlust einzelner Bits die Qualität des Originals wahrnehmbar beeinflussen würde, beispielsweise bei der Kompression von Texten und Tabellen [1].

Drei Vertreter der verlustlosen Kompression werden in dieser Arbeit vorgestellt; insbesondere wird ihre Vorgehensweise bei der Kodierung und der Dekodierung der Daten schrittweise erklärt.

Eine weitere Methode zur verlustlosen Datenkomprimierung ist die Lauflängenkodierung (''run-length encoding''). Hier werden die Wiederholungen eines Zeichens im Text durch einen Zähler und das Zeichen ersetzt. Diese Vorgehensweise kann zum Beispiel bei der Faxübertragung genutzt werden. Betrachtet man die zu übertragenden Bilder als bitonal, so setzt sich das Bild aus schwarzen und weißen Pixeln zusammen. Da jedes Pixel sich nun mit einem Bit darstellen läßt, erhält man dadurch in der Regel größere Bitfolgen, sogenannte ''runs'', die sich effizient zusammenfassen lassen. Diese Methode entspricht somit der Kompression einer Binärdatei, im Folgenden soll jedoch nicht weiter auf sie eingegangen werden.

1.2 Gliederung

In Abschnitt 2.2 wird auf die Entwicklung des **Huffman**-Codes eingegangen. Hierbei handelt es sich um ein statisches Kompressionsverfahren, das durch Aufteilen der Häufigkeitstabelle einen Binärbaum entwickelt, welcher als Grundlage dieser Kodierungstechnik dient.

Die wichtigsten Vertreter der **Lempel-Ziv**-Familie werden in Abschnitt 2.3 vorgestellt. Sie gehören zur großen Gruppe der wörterbuchbasierten Kompressionsalgorithmen, bei denen die zu komprimierenden Zeichenfolgen in einem Wörterbuch verwaltet werden. Die Zeichenfolgen des zu komprimierenden Eingabestroms werden im Wörterbuch nachgeschlagen. Ist dort bereits ein entsprechender Eintrag vorhanden, so erfolgt die Substitution im Ausgabestrom durch eine Referenz, ansonsten wird ein neuer Eintrag generiert.

Auch bei der **Arithmetischen Kodierung** werden die zu verschlüsselnden Zeichen unter Verwendung ihrer Wahrscheinlichkeit kodiert. Dabei erfolgt eine Abbildung der Zeichenkette auf eine Gleitkommazahl aus dem Intervall $[0, 1)$, welches korrespondierend zu den Wahrscheinlichkeiten aufgeteilt wird. Eine nähere Betrachtung dieser Vorgehensweise erfolgt in Abschnitt 2.4.

Kapitel 2

Verlustlose Kompressionsalgorithmen

2.1 Grundlagen

Das prinzipielle Ziel aller Kompressionsalgorithmen ist die Ersetzung redundanter Daten im Eingabestrom durch kürzere Zeichenfolgen. Der Begriff Redundanz ist sehr weitreichend, so daß hier die wichtigsten Arten von Redundanz kurz erläutert werden. Dabei kann vorweggenommen werden, daß es kein universelles Kompressionsverfahren gibt, das für alle Arten von Redundanz optimal wäre. Es gibt jedoch eine Reihe spezieller Algorithmen, die auf eine bestimmte Art oder Kombination von Redundanz hin optimiert wurden [13].

Zeichenverteilung

Die Verteilung der Redundanz bezogen auf das Alphabet hängt vom Datensatz ab und kann daher stark variieren. In sprachlichen Texten kommen Vokale, insbesondere der Buchstabe *"e"* sehr häufig vor. Dabei ist die Sprache des Textes ebenfalls von Bedeutung, vergleicht man beispielsweise die Häufigkeit des Ypsilons in deutschen und englischen Texten. Ein vergleichbares Phänomen tritt bei der Musterwiederholung (siehe unten) auf. Tabellenkalkulationen hingegen enthalten verhältnismäßig viele Zahlen. Die Verteilung der Zeichen eines Alphabets ist daher meistens nicht zufällig, sondern hängt stark vom Ursprung der Daten ab. In Abschnitt 2.2 wird gezeigt, wie die Anwendung des Huffman-Codes auf diese Art der Redundanz abgestimmt ist.

Wiederholung von Einzelzeichen

Zeichenketten mit gleichen, aufeinanderfolgenden Zeichen stellen eine Art von Redundanz dar, wie sie bei Bildern, insbesondere aus CAD-Anwendungen oder bei Faxen relativ häufig vorkommt. Hier ist die Lauflängenkodierung als besonders effiziente Komprimierungsstrategie anzusehen.

Musterwiederholung

Muster, d*ie* sich w*ie*derholen, sind eine Form der Redundanz, w*ie* man s*ie* häufig in Textdate*ie*n vorfindet. So tritt das Muster *"ie"* im vorherigen Satz fünfmal auf. Die Häufigkeit von Mustern hängt sehr stark von der Art und der Sprache des Textes ab. Das Wort *"die"* kommt in deutschen Texten erheblicher häufiger vor als in englischen. Diese Art von Redundanz ist mit der Zeichenverteilung (siehe oben) verwandt und als eine Erweiterung von ihr zu verstehen. Wie aus Abschnitt 2.3 hervorgehen wird, eignen sich die verschiedenen Vertreter der Lempel-Ziv-Familie für die Substitution solcher häufig auftretender Muster am besten.

Ortsredundanz

Die Häufigkeit des Erscheinens eines bestimmten Zeichens oder einer Zeichenkette kann auch von der relativen Lage in einer Datei abhängig sein, das heißt, Redundanz muß nicht immer gleichmäßig über den gesamten Datensatz verteilt sein. Vielmehr gibt es beispielsweise in Bildern Bereiche, in denen andere Farben und Muster auftreten als andernorts. Der Kompressionsalgorithmus muß in der Lage sein, sich während der Kodierung an die auftretenden Daten anzupassen, da sonst die Effizienz erheblich vermindert wird. Sind hingegen Wiederholungen einzelner Farbwerte auf bestimmte Bildbereiche beschränkt, so können diese von speziell darauf angepaßten Verfahren besonders gut erfaßt werden.

Daher wurden verschiedene Lempel-Ziv-Varianten entwickelt, die auf bestimmte Arten von Daten und deren Redundanz hin optimiert wurden. Es gibt jedoch bisher noch kein Verfahren, das jede Art von Daten gut komprimieren kann.

2.2 Huffman-Code

Die üblichen Zeichen-Codes verwenden für jedes Zeichen die gleiche Anzahl an Bits zur Darstellung. Abhängig von der Sprache, beispielsweise deutsch oder englisch, werden die verschiedenen Zeichen jedoch mit unterschiedlicher Häufigkeit verwendet. Die Darstellung häufig auftretender Buchstaben mit einer kürzeren Bitfolge stellt also einen ersten Ansatz zur Verringerung von Redundanz mittels Datenkompression dar [10].

Das Modell der Shannonschen Informationsquelle ist ein einfaches Modell einer Nachricht, welches sich darauf beschränkt, anzunehmen, daß jedes Zeichen des festgelegten Alphabets mit einer vom Kontext unabhängigen Wahrscheinlichkeit auftritt. In der Realität ist dies jedoch nur selten der Fall, so sind in der menschlichen Sprache bestimme Buchstabenkombinationen häufiger als andere, man denke beispielsweise an die deutsche Sprache, in der auf ein *"q"* immer ein *"u"* folgt. Daher ignorieren Algorithmen, die auf der Annahme einer Shannonschen Informationsquelle basieren, die strukturellen Eigenschaften der Nachricht und somit auch zwei der darin enthaltenen Redundanzarten, nämlich die Wiederholung von Einzelzeichen und von Mustern.

2.2.1 Präfixfreiheit des Codes

Die Idee der Kodierung von Zeichen mit variabler Länge ist keinesfalls eine Erfindung des zwanzigsten Jahrhunderts, sondern lag bereits dem zwischen 1832 und 1837 von `Samuel F. B. Morse` entwickelten Morsealphabet zugrunde. Dabei entspricht die Kodierung von Zeichen durch lange und kurze Symbole der Darstellung von Zeichen als Bitfolgen. Zur Kodierung von Nachrichten wird jedoch noch die Pause zwischen zwei Zeichen als drittes Symbol benötigt, um eine eindeutige Rekonstruktion zu ermöglichen [19].

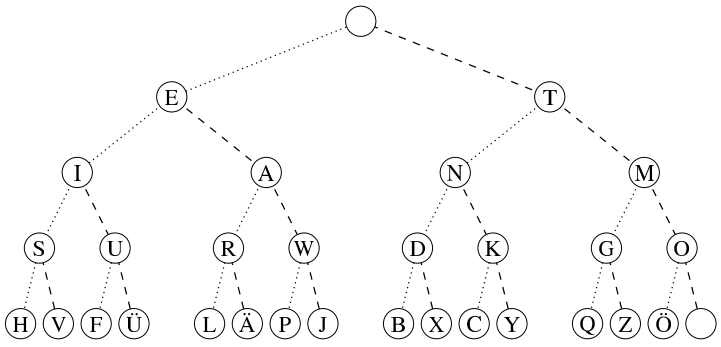

Abbildung 2.1: **Baumstruktur des Morsealphabets**

Der Grund für diese expliziten Begrenzer ist, daß einige Codes des Morsealphabets gleichzeitig den Anfang anderer Codes bilden. In Abbildung 2.1 ist das

Morsealphabet als Binärbaum dargestellt, in dem (fast) alle Knoten außer der Wurzel mit einem Buchstaben belegt sind. Dabei steht der Linksabstieg für ein kurzes und der Rechtsabstieg für ein langes Zeichen.

Hier wird die Notwendigkeit der Pause zwischen den Buchstaben deutlich, da der Code eines Knoten auch der Beginn des Codes eines im Teilbaum darunter stehenden Buchstaben sein könnte. So lassen sich sechs aufeinanderfolgende Punkte ohne Pause beispielsweise als *"EIS"* oder als *"SIE"* interpretieren.

Während bei Blockcodes durch die vorgegebene Länge das Ende eines jeden Codes eindeutig ist, benötigt man bei Codes mit variabler Länge weitergehende Einschränkungen. Um einen Code eindeutig dekodierbar zu konstruieren, kann man als Voraussetzung eine zwar nicht unbedingt notwendige, aber hinreichende Forderung formulieren: Kein Code darf Präfix eines anderen sein, das heißt, kein Zeichen darf so kodiert werden, daß sein Code der Beginn eines anderen Codes ist. Im Modell des Morsealphabets stellt beispielsweise das E einen Präfix für die Buchstaben I und S dar. In Binärbäumen ist die geforderte Präfixfreiheit immer dann gegeben, wenn in den inneren Knoten keine Eintragungen vorgenommen werden sondern nur in den Blättern des Baumes.

2.2.2 Shannon-Fano-Kodierung

`Claude E. Shannon` entwickelte bei den *"Bell Laboratories"* Ende der 40er Jahre fast zeitgleich mit `Robert M. Fano` am *"Massachusetts Instutute of Technology" (MIT)* einen ersten Ansatz zur Konstruktion eines solchen Baumes zur Kodierung mit bekannten Wahrscheinlichkeiten. Dabei wurden alle vorkommenden Zeichen mit der Häufigkeit ihres Auftretens (oder der entsprechenden Wahrscheinlichkeit) in einer Liste nach ihren Häufigkeiten sortiert [19].

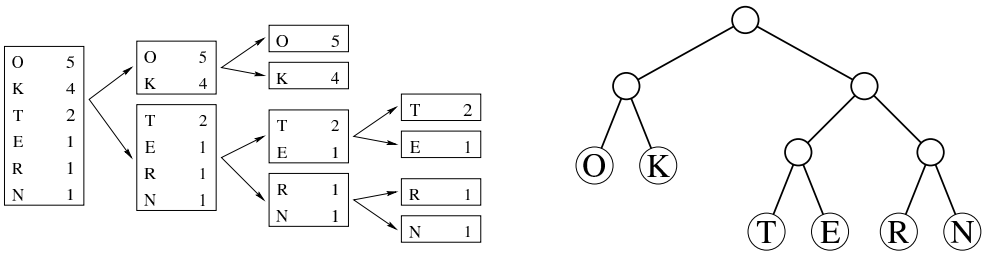

Abbildung 2.2: **Aufteilen der Häufigkeitstabelle (Shannon-Fano)**

Die erstellte Liste wird dann so geteilt, daß die Summen der Häufigkeiten der Zeichen in beiden Hälften möglichst gleich groß sind. Mit den so entstandenen Listen wird nun so lange nach diesem Prinzip verfahren, bis jede Liste nur noch einen Eintrag enthält. Die Vorgehensweise ist in Abbildung 2.2 dargestellt und entspricht der Konstruktion eines Baumes von der Wurzel aus. Das für dieses Beispiel gewählte Wort *"Rokokokokotten"* ließe sich mit dieser Kodierung durch

33 Bits darstellen, bei einer Darstellung mittels eines Blockcodes, dem drei Bit pro Zeichen zugrunde gelegt werden, hätte das Wort eine Länge von 42 Bits.

2.2.3 Huffman-Kodierung

Im Jahre 1952 veröffentlichte `David A. Huffman` ein Verfahren, das unter der Annahme einer Shannonschen Quelle nachweisbar einen präfixfreien Code liefert und für den die Optimalität im Sinne der minimalen Zieldatenlänge bewiesen wurde [17]. Während bei der Shannon-Fano-Kodierung der Baum top-down konstruiert wird, entwickelt Huffman den Baum bottom-up [14].

Die Darstellung des Huffman-Codes erfolgt in dieser Arbeit als Trie [12]. Das Wort "Trie" wird üblicherweise wie das englische Wort *"try"* gesprochen, um es vom *"tree"* unterscheiden zu können. Es hat seinen Ursprung im Wort *"retrieval"*, das auf die Verwendung von Tries als Suchstruktur hinweist [11].

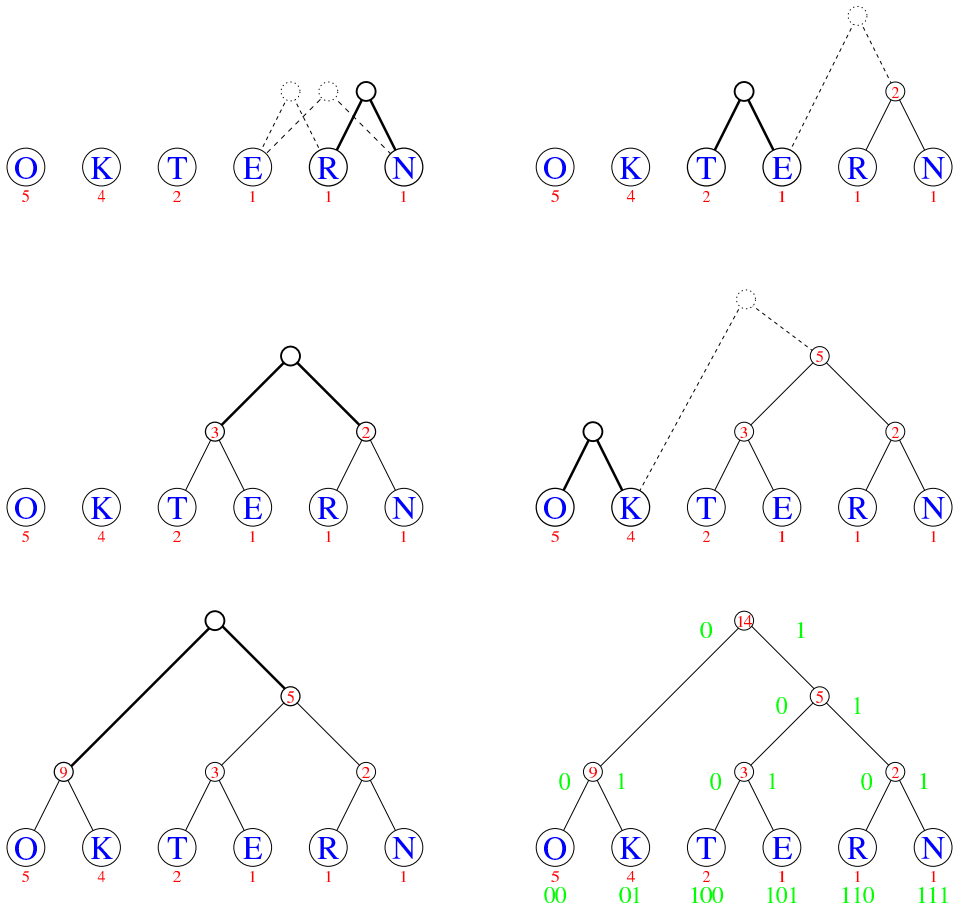

Abbildung 2.3: **Schrittweiser Aufbau eines Huffman-Tries**

Die Vorgehensweise von D. Huffman ist in Abbildung 2.3 Schritt für Schritt dargestellt. Zuerst beginnt man mit noch nicht miteinander verbundenen Knoten,

in denen die Buchstaben mit ihrer jeweiligen Häufigkeit oder Wahrscheinlichkeit eingetragen sind. Jetzt werden immer die beiden Knoten mit der geringsten Häufigkeit ausgewählt, die keinen Vater haben. Auf diese Weise wird ein neuer Knoten erzeugt, welcher die beiden gewählten Knoten zusammenfaßt und als Wert die aufsummierte Häufigkeit seiner Söhne zugewiesen bekommt.

Diese Methode wird nun so lange wiederholt, bis nur noch ein Knoten keinen Vater besitzt, dieser Knoten ist dann die Wurzel des Baumes. Das heißt, daß bei jedem Schritt die Anzahl der vaterlosen Knoten um eins reduziert wird, bis zuletzt ein zusammenhängender Baum entstanden ist.

Abbildung 2.3 zeigt außerdem, daß es bei dieser Vorgehensweise mehrere Lösungen gibt. Besitzen mehrere Knoten die gleiche Auftrittshäufigkeit, so ist nicht eindeutig festgelegt, welche Knoten auf welche Weise zusammengefaßt werden müssen. Die Alternativen bei der Zusammenfassung zweier Knoten wurden in der Abbildung durch gestrichelte Linien dargestellt. Dabei haben verschiedene Lösungsbäume zwar unterschiedliche Codes, sie kodieren aber alle den zu komprimierenden Text mit der gleichen Länge. Die Bäume in Abbildung 2.4 stellen zwei alternative Lösungsmöglichkeiten für das gleiche Problem dar.

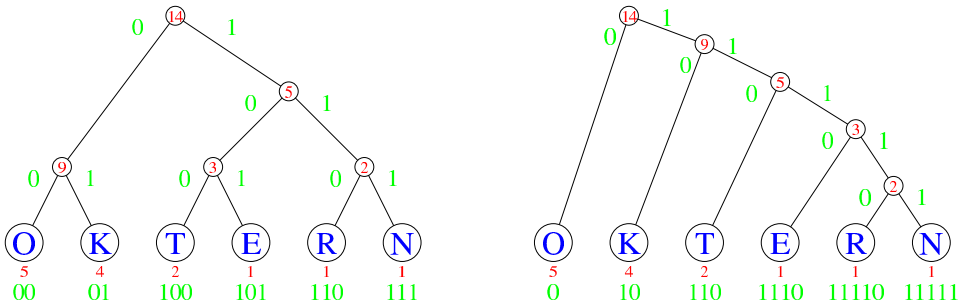

Abbildung 2.4: **Alternative Lösungen bei gleichen Häufigkeiten**

Die Erzeugung eines Huffman-Tries sieht in der Notation in Pseudo-Code wie folgt aus:

Schrittweise Erzeugung des Huffman-Tries

1. Bestimme die Auftrittshäufigkeiten der Zeichen.

2. Trage die Zeichen und ihre Häufigkeiten in die Blätter ein.

3. Suche die beiden vaterlosen Knoten mit der geringsten Häufigkeit.

4. Erzeuge für diese beiden Knoten einen Vaterknoten mit der aufsummierten Häufigkeit seiner Söhne.

5. Weise dem linken Pfad eine `0` und dem rechten Pfad eine `1` zu.

6. Wenn es noch mehr als einen vaterlosen Knoten gibt, gehe zu Schritt 3.

Die Kodierung mit dem Huffman-Trie

Bei der Kodierung eines Zeichens wird der Baum von unten nach oben durchlaufen. Das heißt, daß das Blatt mit dem zu verschlüsselnden Zeichen als Startpunkt gewählt wird und der Baum von dort aus bis zur Wurzel durchquert wird. In Abbildung 2.5 wird dies anhand des Buchstaben *"E"* verdeutlicht. Der Code setzt sich dabei aus den Bits auf dem Weg von der Wurzel zum entsprechenden Blattknoten zusammen.

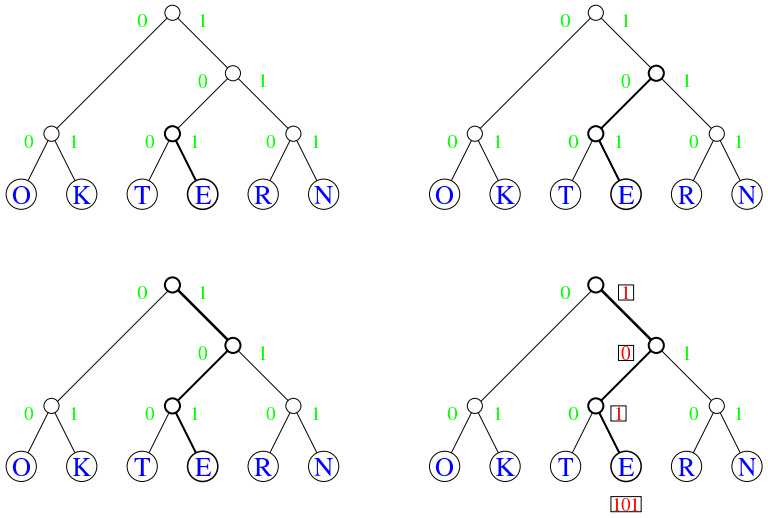

Abbildung 2.5: **Kodieren des Buchstaben E (bottom-up)**

Die Dekodierung mit dem Huffman-Trie

In Abbildung 2.6 ist dargestellt, wie die Bitfolge `101` dekodiert wird, indem an jedem Knoten je nach Wert des entsprechenden Bits der Weg zum linken bzw. rechten Sohn gewählt wird. Im auf diese Weise erreichten Blattknoten steht das zum Code gehörende Zeichen.

Nachteile der Huffman-Kodierung

Einer der Nachteile ist zunächst derjenige, den die Huffman-Kodierung mit allen Codes gemeinsam hat, die Zeichen als Bitfolgen unterschiedlicher Länge darstellen: Die Umsetzung erfolgt durch Bitoperationen, die auf den meisten Systemen relativ umständlich sind.

Desweiteren sind die Auftrittswahrscheinlichkeiten in den meisten Fällen nicht a priori bekannt, so daß stattdessen von geschätzten Werten ausgegangen wird, um dann einen suboptimalen Code zu konstruieren. Alternativ können vor dem eigentlichen Kodiervorgang die Häufigkeiten der einzelnen Zeichen gezählt werden. Sind die Häufigkeiten bekannt, so müssen diese auch bei der Dekodierung

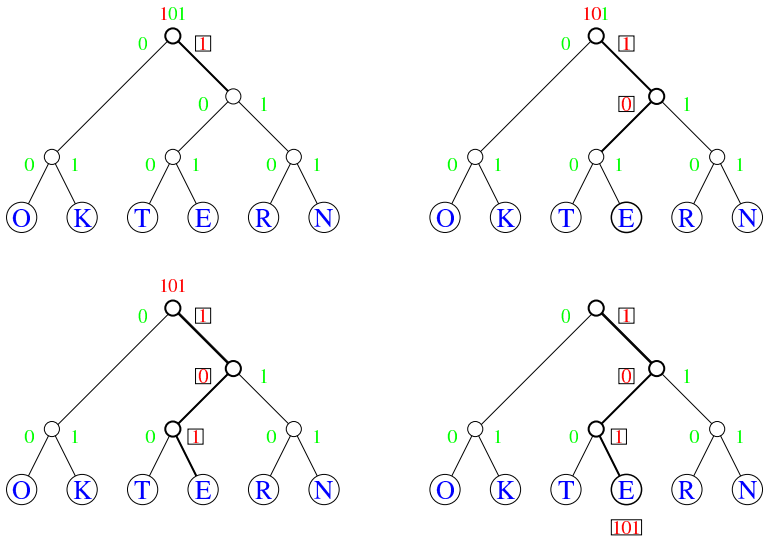

Abbildung 2.6: **Dekodieren der Bitfolge** `101` **(top-down)**

zur Verfügung stehen, um den komprimierten Text fehlerfrei rekonstruieren zu können.

Wie das Beispiel gezeigt hat, gibt es oft mehr als nur eine optimale Lösung für gegebene Häufigkeiten. Daher muß die kodierte Nachricht durch den verwendeten Huffman-Trie ergänzt werden, was diese nur unwesentlich verlängert, wenn man ein kleines Eingabealphabet wie beispielsweise den ASCII-Zeichensatz mit nur 256 Zeichen zugrunde legt. In diesem Fall ist der zusätzliche Speicheraufwand verschwindend gering, wenn man ihn mit den Dateigrößen vergleicht, die eine Komprimierung rechtfertigen. Möchte man die Kontextabhängigkeit zumindest teilweise nutzen und nicht jedes einzelne Zeichen, sondern Diagramme über dem Eingabealphabet betrachten, so wären das bereits 65536 Zeichen.

Ergebnisse in der Praxis

Im obigen Beispiel wird das Wort *"Rokokokokotten"* mit 33 Bits dargestellt, was gegenüber der normalen ASCII-Darstellung eine Einsparung von 79 Bits, also mehr als 70 Prozent bedeutet. Die Kompressionsrate sinkt im Normalfall mit der Größe des Alphabets, das in diesem Beispiel ja nur aus sechs Zeichen bestand. Normalerweise erreicht man bei Texten eine Kompressionsrate von ungefähr 50 Prozent [1].

2.3 Lempel-Ziv-Kodierung

Wie bereits eingangs erwähnt, sind die verschiedenen Mitglieder der Lempel-Ziv-Familie auf einem Wörterbuch basierende Kodierungstechniken. Abgesehen von der Verlustlosigkeit sind alle Vertreter der LZ-Algorithmen durch weitere grundlegende Eigenschaften gekennzeichnet [15].

Im Gegensatz zu den statischen Verfahren, bei denen vor der Kodierung das Wörterbuch festgelegt und dieses weder beim Kodieren noch beim Dekodieren verändert wird, handelt es sich bei der Lempel-Ziv-Kodierung um ein *dynamisches Verfahren*. Das heißt, das Wörterbuch wird sowohl während der Kodierung aus den zu verschlüsselnden Daten als auch während der Dekodierung implizit aus den verschlüsselten Daten neu generiert. Dies hat zur Folge, daß zur Dekompression keine zusätzlichen Informationen über die Daten benötigt werden, da diese Informationen während des Dekodiervorganges gewonnen werden. Hier liegt der Hauptunterschied gegenüber anderen verlustlosen Verfahren, da ja beispielsweise beim Huffman-Code und, wie in Abschnitt 2.4 beschrieben wird, bei der Arithmetischen Kodierung die Kodierungstabelle mit übertragen werden muß.

Ein weiterer Vorteil der Lempel-Ziv-Algorithmen ist, daß sie sich auf Grund dieser Eigenschaften sehr gut dazu eignen, für den Benutzer transparent implementiert zu werden, wie es zum Beispiel beim Modem-Übertragungsprotokoll v42.bis der Fall ist. Dabei handelt es sich konkret um den Lempel-Ziv-Welch-Algorithmus (siehe Abschnitt 2.3.4), der unter anderem auch in verschiedenen Grafikformaten wie GIF und TIFF sowie in Dateien im PostScript Level 2-Format als Grundlage für den dort verwendeten LZC (siehe Abschnitt 2.3.5) dient.

Prinzipiell erfolgt die Datenkompression bei allen Lempel-Ziv-Verfahren, indem redundante Zeichenketten durch kürzere Codes ersetzt werden. Die Art und Weise, wie Redundanz bei der Kompression erkannt und wie das Wörterbuch im Einzelfall erzeugt und verwaltet wird, ist das wichtigste Unterscheidungsmerkmal zwischen den einzelnen Lempel-Ziv-Derivaten [20].

Die Lempel-Ziv-Welch-Kodierung (kurz: *LZW*) ist das mit Abstand bekannteste dieser Verfahren, so daß sie häufig fälschlicherweise als *die* Lempel-Ziv-Kodierung bezeichnet wird. Auf ihr liegt in diesem Abschnitt auch das Hauptaugenmerk, zunächst erfolgt jedoch noch die Aufteilung der einzelnen Lempel-Ziv-Algorithmen in zwei Hauptgrupppen [16].

Die erste Gruppe untersucht die zu komprimierende Datenmenge auf Wiederholungen von Zeichenketten. Wird eine Zeichenkette erneut gefunden, wird sie durch einen Zeiger auf ihr letztes Auftreten ersetzt. Das Wörterbuch wird in diesem Fall also durch die bereits verarbeitete Datenfolge repräsentiert (`implicit dictionary`). Zu dieser Gruppe gehören die Varianten des LZ77-Algorithmus in den Abschnitten 2.3.1 und 2.3.2.

In der zweiten Gruppe wird während des Kompressionsvorganges ein Wörterbuch aus Zeichenfolgen erzeugt, die in den Quelldaten auftreten. Eine Zeichenfolge, die im Wörterbuch eingetragen ist, wird dann durch den Index ihres Eintrages

ersetzt. Dieses Wörterbuch wird, wie bereits erwähnt, während der Dekodierung dynamisch generiert. Die in den Abschnitten 2.3.3 bis 2.3.5 beschriebenen Varianten des LZ78-Algorithmus, insbesondere der LZW-Algorithmus, gehören dieser Gruppe an.

2.3.1 LZ77-Algorithmus

Dieses Verfahren wurde 1977 von `Abraham Lempel` und `Jacob Ziv` vorgestellt [2]. Dabei wird ein Datenfenster der Größe n (Zeichen) auf den Eingabedatenstrom gelegt, anstatt die gesamten zu verschlüsselnden Daten nach einem Muster zu durchsuchen. Es wird also nur das Datenfenster auf Übereinstimmungen untersucht, um den Rechenaufwand zu minimieren.

Der Algorithmus durchsucht das Datenfenster nach der längsten Zeichenkette, die mit der an der aktuellen Kodierungsposition beginnenden Zeichenkette übereinstimmt. Wird eine Übereinstimmung gefunden, so wird diese durch einen Zeiger auf die Position der Übereinstimmung ersetzt. Bei diesem Zeiger handelt es sich im Gegensatz zum normalen Gebrauch eines Zeigers um das Tupel (*Position*; *Länge*). Hat der Algorithmus keine Übereinstimmung gefunden, so gibt er den Nullzeiger $(0, 0)$ aus.

Zu den bekannteren Implementierungen des LZ77-Algorithmus gehören das UNIX-Programm `compress` sowie die Packprogramme `arj`, `lha`, `pkzip`, `zip` und `zoo`. Das Verfahren ist außerdem Bestandteil vieler Textverarbeitungsprogramme.

2.3.2 LZSS-Algorithmus und adaptierte Verfahren

Der 1982 von `James A. Storer` und `Thomas G. Szymanski` veröffentlichte LZSS-Algorithmus ist eine Weiterentwicklung des LZ77 [4]. Eine erneute Verbesserung erhält man, indem man dessen Ergebnis mit Hilfe anderer Kompressionsverfahren weiter verdichtet. Als Beispiele seien hierfür zwei adaptierte Verfahren genannt:

Da die in der eben vorgestellten Methode erläuterten Zeiger, also Tupel der Form (*Position*, *Länge*), mit einer Länge von 3 wesentlich häufiger vorkommen als mit einer Länge von beispielsweise 23, können unter Verwendung der Arithmetischen Kodierung (siehe Abschnitt 2.4) die häufig vorkommenden Zeichen mit einer geringeren Anzahl von Bits dargestellt werden als die seltener vorkommenden. Bei diesem Verfahren handelt es sich um LZSS mit adaptierter Arithmetischer Kodierung.

Ersetzt man nun die Arithmetische Kodierung durch den Huffman-Code (siehe Abschnitt 2.2), erhält man das LZH-Verfahren, das beispielsweise im Archivierungsprogramm LHArc Verwendung findet. Bei beiden Verfahren erreicht man ungefähr die gleiche Kompressionsrate, die Variante mit Huffman-Code ist jedoch erheblich schneller.

2.3.3 LZ78-Algorithmus

Ein Jahr nach der Veröffentlichung des LZ77 publizierten A. Lempel und J. Ziv dieses Verfahren [3]. Es wurde zur Grundlage für die Algorithmen der zweiten Gruppe. Bei diesem substituierenden Verfahren wird der Text nicht durch Zeiger ersetzt, indem die Zeichenketten durch Zeiger auf eine Position im Datenfenster referenziert werden, sondern es wird sowohl während des Kodier- als auch des Dekodiervorganges ein Wörterbuch (dynamisch) aufgebaut. Es erfolgt also eine Substitution durch eindeutige Codes, welche den Index auf das Wörterbuch darstellen und somit die Zeichenketten dort referenzieren.

Die prinzipielle Vorgehensweise der LZ78-Komprimierung wird bei den Ausführungen zum LZW in Abschnitt 2.3.4 erläutert. Der wesentliche Unterschied zwischen den beiden Algorithmen besteht darin, daß bei der Datenkompression mit LZW nur noch Codes und keine einzelnen Zeichen mehr ausgegeben werden.

Im Gegensatz zu LZW ist hier das Wörterbuch zu Beginn des Kodierungsvorganges noch leer und wird erst im Laufe des Vorgangs aufgebaut. Dabei wird beim zeichenweisen Einlesen der Eingabedaten versucht, eine möglichst lange Zeichenfolge im Wörterbuch zu finden. Ist eine Zeichenkette (noch) nicht im Wörterbuch verzeichnet, so wird sie an die nächste freie Stelle eingetragen und der Index des zuletzt gefundenen Eintrags ausgegeben. Wird hingegen ein Eintrag gefunden, so wird das nächste Zeichen eingelesen und erneut gesucht.

2.3.4 LZW-Algorithmus

Dieses Verfahren ist eine Weiterentwicklung des LZ78-Algorithmus, die im Jahre 1984 von `Terry A. Welch` veröffentlicht wurde [5]. Wie bereits erwähnt ist LZW das bekannteste der Lempel-Ziv-Verfahren und ist, wohl auch auf Grund seiner Verbreitung, das variantenreichste unter den LZ-Verfahren. Da diese Variante des LZ78 von T. A. Welch zur Hardwareimplementierung entwickelt wurde, mußte es mehreren Ansprüchen genügen.

Vor allem mußte es sowohl bei der Kompression als auch bei der Dekompression hohe Arbeitsgeschwindigkeiten erreichen. Desweiteren sollte es unabhängig von der im Dateneingabestrom auftretenden Art der Redundanz möglichst hohe Kompressionsraten erreichen. Außerdem ist als weiterer Vorzug anzumerken, daß der LZW-Algorithmus auf Grund der Tatsache, daß er seine komprimierten Daten als Bytes und nicht als Wörter ablegt, von der Byteanordung der verschiedenen Plattformen unabhängig ist.

Die Standardimplementierung erstellt ein Wörterbuch von bis zu 4096 Einträgen mit einer festen Codelänge von 12 Bit, welche notwendig ist, um die gesamte Tabelle addressieren zu können. Dieses Wörterbuch enthält im Gegensatz zum leeren Wörterbuch beim LZ78 zu Beginn die 256 Abbildungen der einzelnen Bytes als Einträge.

Die Kodierung mit LZW

Beim Einlesen des ersten Zeichens ist der Präfix leer. Das erste Zeichen muß der Definition nach im Wörterbuch eingetragen sein. Nun wird das Zeichen im Präfix gespeichtert und das nächste Zeichen wird eingelesen. Da die aus dem Präfix und dem Zeichen zusammengesetzte Zeichenkette nicht im Wörterbuch vorhanden ist, wird sie an der ersten freien Stelle eingetragen und der Code für den Präfix wird ausgegeben.

Solange der Kompressor noch Zeichen aus dem Eingabedatenstrom lesen kann, verfährt er immer auf diese Weise, wobei sich ihm mit jedem neu eingelesenen Zeichen zwei Alternativen stellen:

Entweder ist die aus dem Präfix und dem neuen Zeichen zusammengesetzte Zeichenkette bereits im Wörterbuch eingetragen. Dann wird einfach das nächste Zeichen eingelesen.

Oder die Zeichenkette hat noch keinen Eintrag im Wörterbuch. In diesem Fall wird der Code für den Präfix ausgegeben, also der dazugehörige Index aus dem Wörterbuch. Ausserdem wird die nicht gefundene Zeichenkette, also der Präfix mit dem angehängten Zeichen, im Wörterbuch eingetragen und dieses Zeichen wird zum neuen Präfix. Zum besseren Verständnis ist der Algorithmus hier noch einmal in Pseudo-Code notiert:

Der Kodierungs-Algorithmus in Einzelschritten

1. Zu Beginn enthält das *Wörterbuch* für jedes im *Eingabedatenstrom* vorkommende Zeichen einen Eintrag, der **Präfix** ist leer.

2. **Zeichen** := nächstes Zeichen aus dem *Eingabedatenstrom*

3. Ist **"Präfix + Zeichen"** im *Wörterbuch*?

JA: **Präfix** := **"Präfix + Zeichen"**

NEIN:

1. Gib den *Code* für **Präfix** aus.
2. Trage **"Präfix + Zeichen"** in das *Wörterbuch* ein.
3. **Präfix** := **Zeichen**

4. Ist das Ende des *Eingabedatenstrom*es erreicht?

NEIN: Gehe zu Schritt 2

JA: Wenn **Präfix** nicht leer ist, gib seinen korrespondierenden *Code* aus.

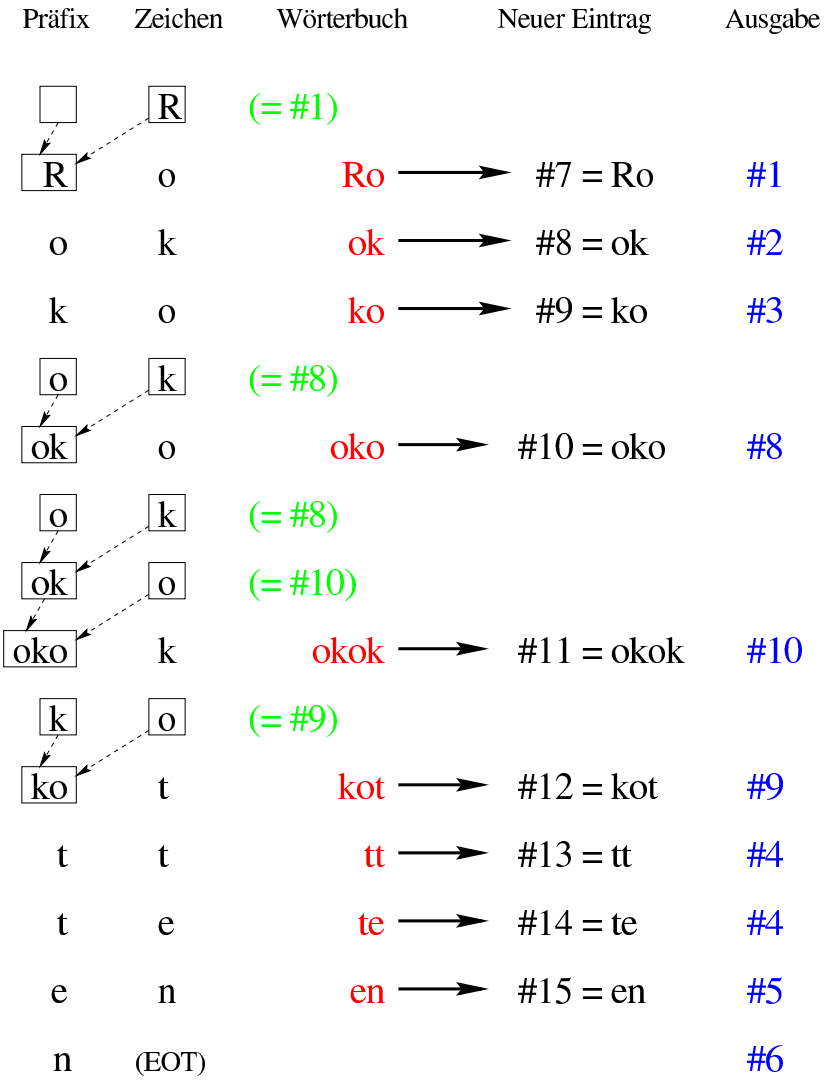

Abbildung 2.7: **Kodierung des Wortes** *"Rokokokokotten"*

Beispiel für den Kodierungsvorgang

Das Beispiel in Abbildung 2.7 soll die Vorgehensweise der LZW-Kodierung noch einmal veranschaulichen. Wie bereits bei der Huffman-Kodierung wird auch in diesem Beispiel das Wort *"Rokokokokotten"* komprimiert. Aus Sicht des LZW-Verfahrens wurde dieses Wort auf Grund seiner häufigen Silbenwiederholung gewählt. Das periodische Auftreten der Buchstabenkombination *"ko"* (beziehungsweise *"ok"*) steigert in diesem Beispiel die Kompressionsrate innerhalb eines Wortes.

Die grünen Einträge im Wörterbuch symbolisieren die erfolgreiche Suche nach der aus Präfix und letztem eingelesenen Zeichen zusammengesetzten Zeichenkette. Ein roter Eintrag hingegen kennzeichnet das Fehlen der gesuchten Buchsta-

benkombination im Wörterbuch, welches durch den folgenden Eintrag behoben wird. Die blauen Einträge in der letzten Spalte stellen die Ausgabe der zuletzt gefundenen Zeichenfolge dar. Am Ende des Eingabedatenstroms wird der Präfix, der zu diesem Zeitpunkt ja bereits im Wörterbuch eingetragen sein muß, ausgegeben. Das Wörterbuch zu Beginn der Kodierung ist in Abbildung 2.8 dargestellt.

#1 = R #2 = o #3 = k #4 = t #5 = e #6 = n

Abbildung 2.8: **Wörterbuch zu Beginn der (De-)Kodierung**

Die Dekodierung mit LZW

Wie bei der Kodierung muß per Definition auch hier für den ersten Code ein Eintrag im Wörterbuch vorhanden sein, der gleich ausgegeben werden kann. Der alte Code wird nun vor dem Einlesen des nächsten Codes immer gespeichert, um während des Dekodiervorganges jederzeit darauf zugreifen zu können. Der vorherige Code wird bei der Dekodierung zur Erstellung des Wörterbuches benötigt, da der Dekodierer dem Kodierer gewissermassen um einen Eintrag hinterherhinkt.

Solange der Eingabedatenstrom noch nicht leer ist, wird auch bei der Dekodierung immer nach der gleichen Methode verfahren, nach dem Einlesen eines neuen Codes bieten sich immer zwei Möglichkeiten, man muß also auch hier eine Fallunterscheidung vornehmen:

Im ersten Fall ist der Code bereits im Wörterbuch eingetragen. Dann kann die Zeichenfolge, welche der Index symbolisiert, einfach aus dem Wörterbuch ausgelesen und ausgegeben werden. An den alten Code wird nun das erste Zeichen des neuen Codes angehängt, diese zusammengesetzte Zeichenfolge kann nun in das Wörterbuch eingetragen werden.

Im zweiten Fall ist noch kein Eintrag für den Code im Wörterbuch vorhanden. Dann tritt der sogenannte "KΩK"-Fall ein, auf den am Ende der Ausführungen zum LZW-Algorithmus noch genauer eingegangen wird. Dort wird dann auch erläutert, warum der im Wörterbuch "fehlende" Code aus dem alten Code und dessen erstem Zeichen zusammengesetzt wird. Der auf diese Weise generierte Code wird nun in das Wörterbuch aufgenommen und ausgegeben. Dieser Fall tritt immer dann auf, wenn sich bestimmte Zeichen und Zeichenkombinationen periodisch häufen. Die Vorgehensweise des Algorithmus bei der Dekodierung soll durch die Notation in Pseudo-Code auf der nächsten Seite noch einmal übersichtlicher dargestellt werden.

Der Dekodierungs-Algorithmus in Einzelschritten

1. Zu Beginn enthält das *Wörterbuch* für jedes im *Eingabedatenstrom* vorkommende Zeichen einen Eintrag.

2. **Code** := erster *Code* aus dem *Eingabedatenstrom* (immer ein Zeichen)

3. Gib **Code** aus.

4. Merke **Code** in **alterCode**

5. **Code** := nächster *Code* aus dem *Eingabedatenstrom*

6. Ist **"Code"** im *Wörterbuch*?

JA:

1. Gib **Code** aus.
2. **Präfix** := **alterCode**
3. **Zeichen** := erstes Zeichen von **Code**
4. Trage **"Präfix + Zeichen"** in das *Wörterbuch* ein.

NEIN:

1. **Präfix** := **alterCode**
2. **Zeichen** := erstes Zeichen von **alterCode**
3. Trage **"Präfix + Zeichen"** (= **Code**) in das *Wörterbuch* ein
UND gib es aus.

7. Wenn das Ende des *Eingabedatenstrom*es noch nicht erreicht ist, gehe zu Schritt 4.

Beispiel für den Dekodierungsvorgang

Um die Arbeitsweise der LZW-Dekodierung an einem Beispiel zu visualisieren, wird nun das im vorigen Beispiel komprimierte Wort *"Rokokokokotten"* wieder dekomprimiert. Dabei reichen die Einträge der Einzelzeichen im Wörterbuch völlig aus, um aus dem kodierten `#1 #2 #3 #8 #10 #9 #4 #4 #5 #6` das ursprüngliche Wort fehlerfrei zu rekonstruieren.

Die Entschlüsselung der bereits im Wörterbuch eingetragenen Indizes ist in der Abbildung grün unterlegt. Besondere Aufmerksamkeit dagegen verdient die Rekonstruktion des Codes `#10`, dessen einzelne Schritte rot dargestellt wurden. In der letzten Spalte sind die ausgegebenen Zeichen(ketten) in blau eingetragen.

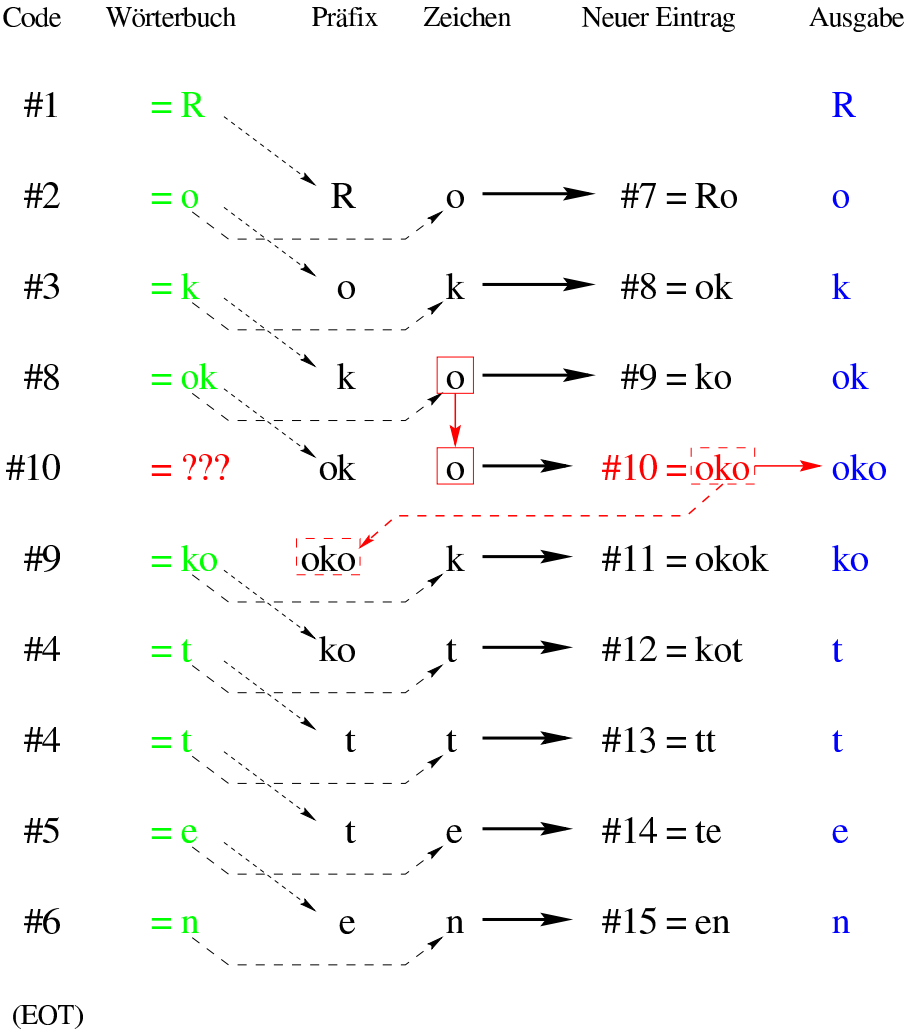

Abbildung 2.9: **Dekodierung des Wortes** *"Rokokokokotten"*

Fall KΩK

Der Fall, daß der LZW-Dekodierer für den nächsten Code des Eingabedatenstroms keinen Eintrag im Wörterbuch vorfindet, wird in verschiedenen Quellen als "KΩK"- oder "ZwZ"-Problematik bezeichnet [13]. Da im Gegensatz zum LZ78 keine Zeichen mehr explizit ausgegeben werden, mußte die LZW-Kodierung so konzipiert werden, daß der Dekodierer immer in der Lage sein muß, das Wörterbuch korrekt zu rekonstruieren. Der Dekodierer hinkt jedoch bei der Erstellung des Wörterbuches dem Kodierer immer um einen Eintrag hinterher.

Für den Fall, daß für einen Code kein Eintrag im Wörterbuch vorhanden ist, wurde die Vereinbarung getroffen, daß es sich bei diesem Code um eine Zeichenkette der Form KΩK handelt. Die Zeichenkette setzt sich also aus dem alten Code, das heißt einem beliebigen Zeichen K, einer beliebigen, also auch beliebig

langen Zeichenkette Ω, und noch einmal dem ersten Zeichen des alten Codes, also K, zusammen.

Die Funktionsweise dieser Vereinbarung ist in Abbildung 2.9 rot dargestellt. Anstatt den neuen Eintrag in das Wörterbuch aus dem alten Code und dem ersten Zeichen des *neuen* Codes zusammenzusetzen, wird hier die Zeichenfolge aus dem alten Code und dem ersten Zeichen des *alten* Codes zusammengesetzt.

2.3.5 LZC-Algorithmus

Der LZC-Algorithmus ist eine Variante des LZW. Er unterscheidet sich von seinem Vorbild darin, daß das Wörterbuch und die Codelänge keine fest vorgegebene Größe haben. Dadurch können die beiden Werte bei Bedarf bis zu einer durch den Benutzer bestimmbaren Grenze wachsen.

Während eine Standardimplementierung des LZW für die Verwaltung der 4096 Wörterbucheinträge eine notwendige Codelänge von 12 Bit besitzt, um die gesamte Tabelle addressieren zu können, beginnt das LZC-Verfahren mit einer Codelänge von 9 Bit, da zu Beginn noch keine großen Indizes benötigt werden. Während der Erstellung des Wörterbuches steigt diese Größe bei Bedarf dynamisch auf bis zu 12 Bit.

Diese Implementierung basiert auf einem Vorschlag von T. A. Welch [5] und unterscheidet sich vom ursprünglichen LZW auch in der Flexibilität des Wörterbuches. Ist das Wörterbuch voll, so kann es als ineffizient erkannt, gelöscht und dann unter Berücksichtigung örtlicher Redundanz neu aufgebaut werden.

Der LZC-Algorithmus ist somit eine Verfeinerung des ursprünglichen LZW und findet Verwendung im Unix-Tool `compress` sowie im wesentlichen bei der Komprimierung des Bilddateiformates GIF. Gegenüber dem herkömmlichen LZW mit fester Codelänge erreicht der LZC-Algorithmus eine Verbesserung der Kompressionsrate um ungefähr 6 bis 7 Prozent.

Ergebnisse in der Praxis

Das Wort *"Rokokokokotten"* kann im oben verwendeten Beispiel anstatt mit ursprünglich 14 Zeichen mit 10 Codes dargestellt werden. Effizienter wird der Algorithmus bei einem erneuten Kodieren des Wortes, das sich bereits im zweiten Schritt mit 6 Codes darstellen läßt und bei der sechsten Wiederholung mit nur noch 3 Codes. Diese Ersparnis wird durch die Häufigkeit der Silben *"ko"* bzw. *"ok"* ermöglicht. Unter normalen Umständen können Texte mit den unterschiedlichen Lempel-Ziv-Varianten auf bis zu 30 Prozent ihrer Ursprungsgröße komprimiert werden [1].

Patente und Lizenzen für LZW

Abschließend bleibt anzumerken, daß der LZW-Algorithmus keine frei verfügbare Software ist. In den letzten Jahren kursierten daher auch mehrfach widersprüchliche Gerüchte über zu entrichtende Lizenzgebühren für die Verwendung von Bilddateien des GIF- oder TIFF-Dateiformates. Für erhebliche Entrüstung sorgte die Bekanntmachung, daß die Firmen *"CompuServe Information Service"* und *"Unisys Corporation"* einen Lizenzvertrag zur Nutzung des LZW geschlossen hätten, insbesondere als das Gerücht aufkam, Unisys verlange von nun an Lizenzgebühren für alle GIF-Bilddateien, die sich auf Webseiten befinden [24].

Tatsache ist, daß in den meisten Fällen bei der Verwendung von GIF-Dateien keine Verletzung des Patents vorliegt. Gegenstand dieses Patents ist nicht das GIF-Dateiformat als solches, sondern das LZW-Verfahren, das in diesem Fall zur Datenkompression verwendet wird. Auf eben dieses Verfahren (*"High speed date compression and decompression apparatus and method"*) wurde das Patent Nr. US-A- 4,558,302 am 10. Dezember 1985 in den USA erteilt [26]. Das Patent wurde ursprünglich ausgestellt auf die *"Sperry Corporation"*, ging jedoch durch die Übernahme durch Unisys in deren Besitz über [22].

In Europa wurde ebenfalls ein Patent erteilt, obwohl nach Artikel 52 Absatz 2c des Europäischen Patentübereinkommens (*EPÜ*) "Programme für Datenverarbeitungsanlagen", also gewissermaßen Software, von den patentfähigen Erfindungen ausgeschlossen sind [23]. Vermutlich wurde dies durch die Darstellung des Verfahrens als Teil einer Maschine und somit als Hardware ermöglicht.

Eine weitere Kuriosität bezüglich der Patente für den LZW-Algorithmus ist die Erteilung des Patents sowohl an Sperry als auch an IBM. Diese legten ihren ersten Antrag auf ein Patent 19 Tage vor Sperry vor, und ihre Ausführungen bezüglich des Verfahrens waren etwas allgemeiner gehalten. Auf die Fortführung eben dieses Antrags wurde IBM 1989 ein Patent erteilt, das inhaltlich der Funktionsweise des LZW-Verfahrens entspricht [25]. Damit wurde das Patent auf den LZW-Algorithmus also letztendlich doppelt vergeben [22].

Unisys verlangt inzwischen Lizenzgebühren für Software, die auf das LZW-Verfahren zurückgreift, um beispielsweise GIF-Bilder zu erzeugen. Von Betreibern von Webservern werden ebenfalls Lizenzgebühren verlangt, falls sich auf deren Sites Bilder befinden, die möglicherweise mit nicht lizenzierten Programmen erstellt wurden. Unter diesem Druck wurde auch die Suche nach einem alternativen Dateiformat für Bilder begonnen. Auf Grund der Verbreitung des GIF-Formates ist jedoch davon auszugehen, daß dieses nicht so schnell ersetzt werden wird.

Bemerkenswert in diesem Zusammenhang ist zu guter letzt, daß das amerikanische Patent der Firma Unisys auf LZW im Jahre 2003 auslaufen wird.

2.4 Arithmetische Kodierung

Der in Abschnitt 2.2 vorgestellte Huffman-Code stellt die zu kodierenden Zeichen durch eine ganzzahlige Anzahl von Bits dar und ist somit nicht ganz optimal [9].

2.4.1 Minimalitätseigenschaft

Die Eigenschaft, daß ein Code minimal ist, läßt sich nach Shannon wie folgt umformulieren: Die Differenz der durchschnittlichen Codewortlänge

$$D(S) = \sum_{i=1}^{n} l(x)p(x)$$

einer Sequenz S und der durchschnittlichen Entropie

$$H(S) = \sum_{i=1}^{n} -ld(p(x))p(x)$$

entspricht der Redundanz der Kodierung und sollte möglichst gering sein. Dabei ist x eines der n Zeichen des Alphabets, $p(x)$ seine Auftrittswahrscheinlichkeit und $l(x)$ die Länge des dazugehörigen Codes. Da die Länge der Codes ganzzahlig ist, dies für den Informationsgehalt jedoch nicht gelten muß, findet man den Idealfall, daß der Huffman-Code redundanzfrei arbeitet, also $D = H$ gilt, nur selten vor. Daher gilt für den Huffman-Code

$$H(S) \leq D(S) \leq H(S) + 1$$

Der Informationsgehalt

$$I(x) = -ld(p(x))$$

eines Zeichens ist aber nur dann ganzzahlig, wenn die Wahrscheinlichkeit eine Potenz von 2 ist. Tritt beispielsweise ein Zeichen mit einer Wahrscheinlichkeit von 90 Prozent auf, so beträgt sein Informationsgehalt $I(x) = -ld(0,9) \approx 0,15$, das heißt, das Zeichen könnte mit 0,15 Bit kodiert werden, muß aber im Huffman-Code trotzdem mit mindestens einem Bit dargestellt werden.

2.4.2 Entwicklung der Arithmetischen Kodierung

Die Einschränkung, Zeichen durch eine ganzzahlige Anzahl von Bits darzustellen, gilt nicht für die Arithmetische Kodierung. Sie erreicht daher höhere Kompressionsraten als der Huffman-Code [18]. Obwohl das Verfahren vom Prinzip her nicht sehr komplex ist, wurde es erst Ende der 70er Jahre bekannt [21].

Bereits im Jahre 1960 gab es erste Ansätze für diese Kodierung, die jedoch an dem Problem scheiterten, daß die Rechner noch nicht die nötige Genauigkeit der Arithmetik besaßen, die mit der Länge der Nachricht stetig erhöht werden muß. Bei seiner Implementierung von 1976 war es `Jorma Rissanen` auch noch nicht möglich, die Speichernutzung effizient zu gestalten [6]. Erst drei Jahre später beschrieb er zusammen mit `Glen G. Langdon Jr.` einen Algorithmus zur Arithmetischen Kodierung, der noch heute so genutzt wird [7][8].

2.4.3 Statische Arithmetische Kodierung

Kodierungsalgorithmus

Der erste Schritt der Arithmetischen Kodierung ist die Zuordnung der Zeichen zu halboffenen, paarweise disjunkten Teilintervallen aus dem Intervall $[0; 1)$, wobei die Verhältnisse der jeweiligen Intervallgrößen gleich den Verhältnissen der Wahrscheinlichkeiten sein müssen.

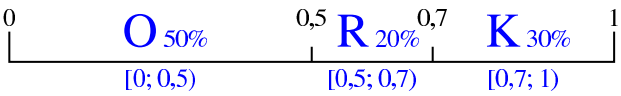

Abbildung 2.10: **Einteilung des Intervalls** $[0, 1)$ **in Teilintervalle**

In Abbildung 2.10 wurde die Aufteilung der Intervalle für das Beispielwort *"Rokoko"* vorgenommen. Aus Gründen der Übersichtlichkeit wurde für die Beispiele der Arithmetischen Kodierung nicht das gleiche Wort wie bei den Beispielen der Abschnitte 2.2.3 und 2.3.4 verwendet.

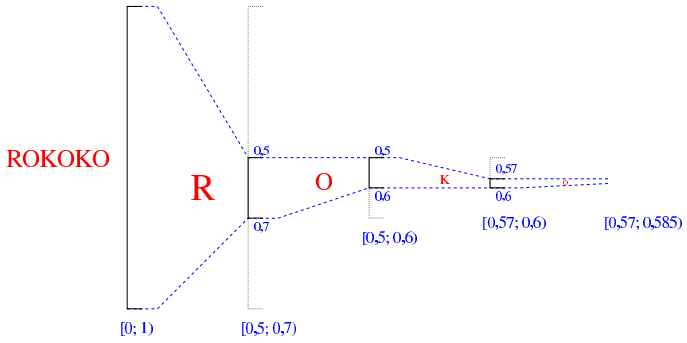

Abbildung 2.11: **Schrittweise Intervallschachtelung**

Die Gleitkommazahl, welche die zu kodierende Nachricht darstellt, wird nun durch sequentielle Intervallschachtelung bestimmt (siehe Abbildung 2.11). Dabei empfiehlt sich die in Abbildung 2.12 verwendete Darstellung aus Gründen der Übersichtlichkeit.

Zuerst wird ein Zeichen eingelesen und das dazugehörige Teilintervall bestimmt. Dieses Teilintervall dient nun als Grundlage für das nächste Zeichen, indem die Zeichenwahrscheinlichkeiten darauf abgebildet werden. Dieser Vorgang kann nun bis zum Einlesen des Endesymbols wiederholt werden. Alternativ kann auch nach einer bestimmten Anzahl von eingelesenen Zeichen der Code abgespeichert und der Vorgang erneut auf dem Intervall $[0; 1)$ begonnen werden.

Zuletzt wird eine Gleitkommazahl als Repräsentant aus dem letzten Teilintervall gewählt. Dabei empfiehlt es sich, eine Zahl zu wählen, die mit möglichst wenig Bits darstellbar ist. Hier ist die schrittweise Entwicklung im Pseudo-Code dargestellt.

Der Kodierungsalgorithmus in Einzelschritten

1. Unterteile das Intervall $[0; 1)$ $\forall$ x entsprechend $p(x)$.

2. Untergrenze := 0, Obergrenze := 1

3. Lies nächstes Zeichen x ein.

4. Größe := Obergrenze - Untergrenze

5. Obergrenze := Obergrenze - Größe $\bullet$ $obererWert(x)$

6. Untergrenze := Untergrenze + Größe $\bullet$ $untererWert(x)$

7. Falls Endesymbol bzw. vorgegebene Anzahl an Zeichen nicht erreicht ist, gehe zu Schritt 3.

8. Bestimme Zahl zwischen Ober- und Untergrenze mit möglichst wenigen Bits.

Beispiel für den Kodierungsvorgang

In Abbildung 2.12 wird die Vorgehensweise anhand des Wortes *"Rokoko"* visualisiert. Das Lösungsintervall wird nach jedem Schritt kleiner, so daß am Ende die Gleitkommazahl im Intervall $[0,5805; 0,58275)$ liegen muß. Dabei stellen die in blau eingezeichneten gestrichelten Linien die Unterteilung des neuen Intervalls als Projektion des alten Teilintervalls dar.

Als Lösung wählt man nun die Zahl aus dem Intervall, die sich mit möglichst wenig Zweierpotenzen darstellen läßt. In diesem Fall ist das die Zahl $0,58165625$, welche, wie in der Abbildung zu sehen ist, in Binärdarstellung acht Nachkommastellen besitzt.

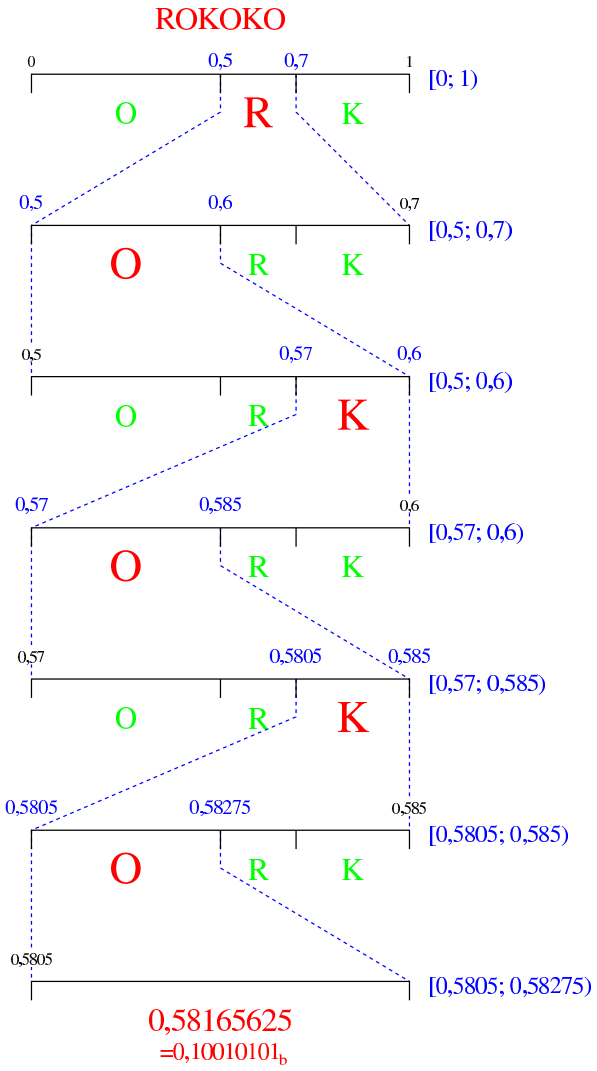

Abbildung 2.12: **Schrittweise Kodierung des Wortes** *"Rokoko"*

Dekodierungsalgorithmus

Die Dekodierung verläuft nach dem gleichen Prinzip. Durch die Bestimmung des ersten Teilintervalls, in dem die Zahl liegt, wird das erste Zeichen dekodiert. Dieser Vorgang wiederholt sich nun so lange, bis das Endesymbol dekodiert wird oder eine bestimmte Anzahl von Zeichen rekonstruiert wurde.

Dem Dekodierer muß, wie bereits aus dem Huffman-Code bekannt, die aus den Wahrscheinlichkeiten erstellte Zeichenverteilung auf die einzelnen Intervalle mitgeteilt werden. Außerdem benötigt er zur fehlerfreien Rekonstruktion des Codes die maximale Anzahl zu kodierender Zeichen, da er sonst nach Erreichen des letzten Zeichens des Eingabedatenstroms weiterarbeiten würde.

Beispiel für den Dekodierungsvorgang

Die Darstellung des Dekodierungsvorganges ist in Abbildung 2.13 erfolgt. Dabei wird in jedem Schritt überprüft, in welchem Teilintervall die Zahl $0,58165625$ liegt. Die Zahl in Klammern dient als Hinweis, nach wievielen Iterationsschritten das Ende des Codes erreicht sein wird. Nach sechs Schritten ist das Wort *"Rokoko"* wieder rekonstruiert.

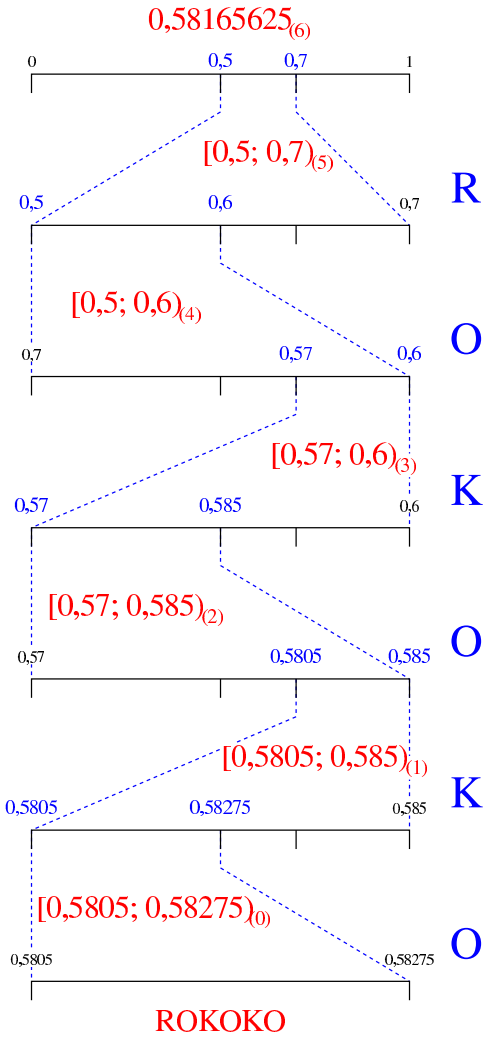

Abbildung 2.13: **Schrittweise Dekodierung der Zahl 0,58165625**

Die Dekodierung unterscheidet sich lediglich im ersten und dritten Schritt von der Kodierung, der achte Schritt wird bei der Dekodierung nicht benötigt. Die Notation im Pseudo-Code erfolgt auf den nächsten Seite.

Der Dekodierungsalgorithmus in Einzelschritten

1. Entnehme Intervalle $\forall\ x$ entsprechend $p(x)$ aus Code.

2. Untergrenze := 0, Obergrenze := 1

3. Bestimme nächstes Zeichen x durch Wahl des Intervalls.

4. Größe := Obergrenze - Untergrenze

5. Obergrenze := Obergrenze - Größe $\bullet$ $oberer Wert(x)$

6. Untergrenze := Untergrenze + Größe $\bullet$ $unterer Wert(x)$

7. Falls Endesymbol bzw. vorgegebene Anzahl an Zeichen nicht erreicht ist, gehe zu Schritt 3.

2.4.4 Dynamische Arithmetische Kodierung

Die Dynamische Arithmetische Kodierung setzt bei einer der Schwächen des eben erläuterten Verfahrens an: Da sich bei der statischen Methode die Auftrittswahrscheinlichkeiten während eines Kodiervorganges nicht ändern, kann der Algorithmus auftretenden Ortsredundanzen nicht entgegenwirken.

Das dynamische Verfahren hingegen reagiert auf wechselnde Häufigkeiten flexibel, indem es regelmäßig die Intervalle an die Auftrittshäufigkeiten anpaßt. Nach der Übertragung einer vorher bestimmten Anzahl von Blöcken werden dann die neuen zu erwartenden Auftrittswahrscheinlichkeiten anhand einer Formel berechnet, die aus Gründen des Rechenaufwandes meistens recht einfach gehalten wird. So bietet es sich an, die neue Wahrscheinlichkeit

$$p_{alt}(x) \quad = \quad \frac{p_{alt}(x) + p_{aktuell}(x)}{2}$$

für ein Zeichen x beispielsweise als Mittel aus der Summe der alten Wahrscheinlichkeit p_{alt} und zuletzt aufgetretenen Wahrscheinlichkeit $p_{aktuell}$ zu berechnen. Man kann durch die Entwicklung komplexerer Formeln auch durchaus sehr realistische Wahrscheinlichkeiten erhalten.

Für die im obigen Beispiel verwendeten Auftrittshäufigkeiten würden sich dann unter Anwendung dieser Formel zum Beispiel nach einem Block, in denen die Buchstaben *"K"* und *"R"* gleich oft vorkommen und der Buchstabe *"O"* überhaupt nicht, die folgenden aktualisierten Wahrscheinlichkeiten ergeben:

$$\begin{aligned} p_{neu}(O) &= \tfrac{0,5+0}{2} &= 0,25 \\ p_{neu}(R) &= \tfrac{0,2+0,5}{2} &= 0,35 \\ p_{neu}(K) &= \tfrac{0,3+0,5}{2} &= 0,4 \end{aligned}$$

Somit kann sich die Dynamische Arithmetische Kodierung den aktuellen Häufigkeiten flexibel anpassen und bei geschickter Wahl der Aktualisierungsformel entsprechend höhere Kompressionsraten erzielen.

Nachteile der Arithmetischen Kodierung

Auf den ersten Blick wirkt dieses Verfahren durchaus kompliziert. Trotzdem benötigt es keine aufwendigen Gleitkommaoperationen, sondern kommt mit einfachen Mitteln aus. Auch muß während der Kodierung nicht mit langen Zahlen gerechnet werden, da Stellen, die sich nicht mehr ändern, bereits ausgegeben werden können.

Auf Grund der Tatsache, daß die Arithmetische Kodierung etwas langsamer ist als die Huffman-Kodierung, erhält diese trotz der weniger effizienten Komprimierung meistens den Vorzug. Dies liegt hauptsächlich daran, daß die Arithmetische Kodierung sehr fehleranfällig ist. Durch einen einzelnen Bitfehler wird eine komprimierte Nachricht bereits unbrauchbar gemacht, da sie ab diesem Bit die Gleitkommazahl auf ein anderes Intervall abbildet und dadurch auch bei den folgenden Zeichen die Intervalle nicht mehr korrekt rekonstruiert werden können.

Ein weiteres Problem teilt die Arithmetische Kodierung mit dem Huffman-Code: Da die exakten Auftrittswahrscheinlichkeiten für die einzelnen Zeichen meistens nicht a priori bekannt sind, kann die theoretisch mögliche maximale Effizienz in der Praxis nicht erreicht werden.

Kapitel 3

Das Applet

Das programmierte Applet dient zur Veranschaulichung der in dieser Arbeit vorgestellten verlustlosen Kodierungsalgorithmen. Auf die Lauflängenkodierung, die intuitiv am einfachsten vorstellbar ist und daher auch nur in der Einleitung Erwähnung fand, wurde bei der Implementierung verzichtet. Zur Auswahl stehen daher die Huffman-Kodierung, die in Abschnitt 2.2 behandelt wurde, der in Abschnitt 2.3 vorgestellte Lempel-Ziv-Welch-Algorithmus und die Arithmetische Kodierung, auf die in Abschnitt 2.4 eingegangen wurde.

Der Benutzer hat nach dem Start des Applets die Möglichkeit, mittels *Radio-Buttons* die gewünschte Kodierung auszuwählen. Außerdem kann er den zu kodierenden Text selbst eingeben. Für diesen Eingabetext wurde die Einschränkung festgelegt, daß er eine Länge von mindestens drei und höchstens achtzehn Zeichen hat. Für eine Zeichenkette von bis zu zwei Zeichen Länge wäre der didaktische Nutzen der grafischen Darstellung der Algorithmen als eher gering anzusehen.

Die obere Schranke für die Länge der Eingabe wurde aus Gründen der Übersichtlichkeit festgelegt. Während bei der LZW-Kodierung lediglich mehr Zeilen für die Visualisierung der Vorgehensweise dargestellt werden müßten, würde die Darstellung der anderen beiden Algorithmen bei einer zu großen Anzahl unterschiedlicher Buchstaben erhebliche Qualitätseinbußen erleiden. Insbesondere die Intervalle bei der Arithmetischen Kodierung würden in diesem Fall zu klein und damit zu unübersichtlich werden.

Mit Ausnahme der LZW-Kodierung wird außerdem überprüft, ob die Zeichenkette aus mindestens zwei verschiedenen Zeichen besteht. Während der Huffman-Trie nur einen Blattknoten besäße und bei der Arithmetischen Kodierung nur ein Intervall vorläge, macht die Kodierung einer Sequenz gleicher Buchstaben mit dem Lempel-Ziv-Welch-Algorithmus durchaus Sinn.

3.1 Technische Realisierung

Die Programmierung der Applets erfolgte in der Programmiersprache *Java*^{TM}, es wurde für den Einsatz von *Swing*-Komponenten die Version 1.3.1 gewählt. Für die Rechner, auf denen die Applets betrachtet werden sollen, wurde eine Auflösung von $800x600$ Pixel vorausgesetzt.

Abbildung 3.1: **Das Start-Applet**

Beim Start-Applet handelt es sich um ein `JPanel`, Dieses setzt sich, wie in Abbildung 3.1 zu sehen ist, aus einem Textfeld vom Typ `JTextfield`, einem `JButton` und einem `ButtonGroup`-Objekt zusammen, welches wiederum eine Gruppe von drei `JRadioButtons` beinhaltet.

Nach dem Drücken des Start-Buttons wird überprüft, ob der Eingabetext den an ihn gestellten Anforderungen genügt. Dazu gehört einerseits eine Länge von drei bis achtzehn Zeichen. Andererseits darf sich die Eingabe nur aus Buchstaben zusammensetzen. Diese zweite Einschränkung soll lediglich die Eingabe von Satzzeichen und *"White-Space"* verhindern, da diese Zeichen in der Darstellung zu klein erscheinen würden.

Im nächsten Schritt erfolgt die Darstellung des gewählten Kodierungsalgorithmus in einem `JFrame`, für welches ein eigener `Thread` gestartet wird. Es ist daher möglich, mehrere Simulationen parallel zu starten, um zum Beispiel die Vorgehensweise der verschiedenen Verfahren besser vergleichen zu können.

Das im `JFrame` erzeugte `Graphics2D`-Objekt greift dabei auf ein `BufferedImage` zu, in welchem der jeweilige `Thread` zeichnet. Durch das Überschreiben der `paint()`-Methode werden nun bei jedem Aufruf der Methode `repaint()` die Veränderungen dargestellt, die im `BufferedImage` vorgenommen wurden.

3.2 Darstellung der Kompressionsalgorithmen

3.2.1 Huffman-Code

Bei der Kodierung mittels Huffman-Trie wird zuerst die schrittweise Konstruktion des Baumes dargestellt. Da die Verwaltung der Struktur des Baumes in Form von linearen Feldern erfolgt, werden diese vor der grafischen Darstellung so sortiert, daß die Reihenfolge der Einträge in den Feldern der Reihenfolge der Blätter im Baum von links nach rechts entspricht, mit den dazugehörigen Vätern wird analog verfahren.

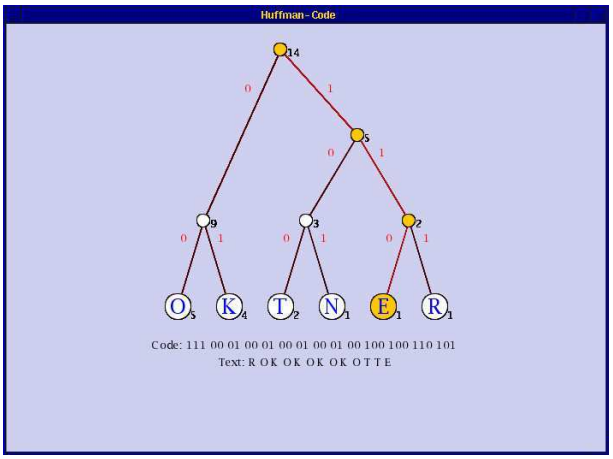

Abbildung 3.2: **Darstellung des Huffman-Codes**

Aus Abbildung 3.2 geht hervor, wie die in den Kreisen eingetragenen Buchstaben mit ihrer jeweiligen Auftrittshäufigkeit unten rechts vom Kreis versehen sind. Die Blattknoten mit der niedrigsten Häufigkeit werden zu einem Teilbaum zusammengefaßt, und der Vater der beiden verbundenen Knoten wird dann wie die Blätter mit den kumulierten Häufigkeiten seiner Söhne versehen. Auf diese Weise werden die Knoten schrittweise zu einem Baum zusammengefaßt, dessen Pfade dann mit dem entsprechenden Bit versehen werden. An den linken Pfaden werden die Nullen eingetragen und an den rechten Pfaden die Einsen.

Im zweiten Schritt der Simulation wird die Kodierung des eingegebenen Wortes gezeigt. Der Knoten des zu verschlüsselnden Buchstaben wird orange unterlegt, danach wird der Weg zur Wurzel gekennzeichnet, indem die Pfade rot und die besuchten Knoten ebenfalls orange dargestellt werden. Die dadurch entstandene Bitfolge wird dann unterhalb des Baumes an die Zeile mit den kodierten Buchstaben angefügt.

Mit der Dekodierung der zuvor entstandenen Bitfolge folgt der letzte Schritt. Die Wurzel wird orange dargestellt, von dort aus wird in jedem Knoten entsprechend der oben beschriebenen Bits verzweigt, bis ein Blattknoten erreicht ist. Dessen Inhalt wird in die Zeile, in welcher der dekodierte Huffman-Code steht, eingetragen. Bei der Dekodierung erfolgt die Darstellung der besuchten Knoten und Pfade auf die gleiche Weise wie bei der Kodierung.

3.2.2 Lempel-Ziv-Welch-Algorithmus

Um die Darstellung der Vorgehensweise des Lempel-Ziv-Welch-Verfahrens nicht zu klein und somit übersichtlicher zu gestalten, wurde die Simulation in zwei Phasen unterteilt. Zwischen diesen beiden Phasen wird nach einer Pause das Bild gelöscht, um für beide Phasen die gesamte Fläche des Fensters zur Verfügung stellen zu können.

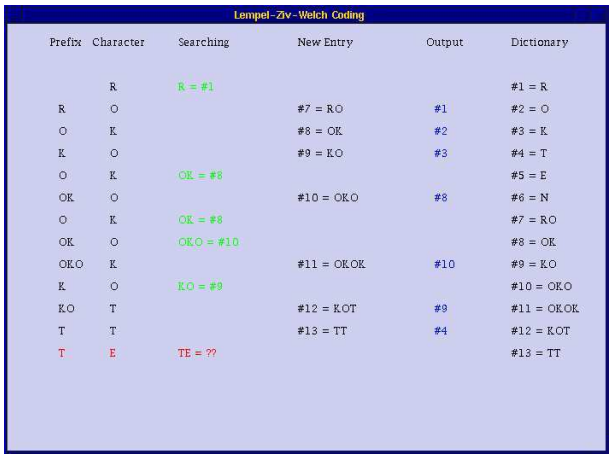

Abbildung 3.3: **Kodierung des Lempel-Ziv-Welch-Algorithmus**

In der ersten Phase erfolgt die Kodierung des Wortes. Die im Eingabetext enthaltenen Buchstaben werden zuerst ins Wörterbuch eingetragen, der Einfachheit halber in der Reihenfolge ihres ersten Erscheinens. Die Kodierung läuft nach dem in Abschnitt 2.3.4 beschriebenen Schema ab und wird hier in Abbildung 3.3 so dargestellt wie bereits im Beispiel der Abbildung 2.7. Im Wörterbuch vorhandene Einträge werden aus Kontrastgründen türkis anstatt grün dargestellt, noch nicht eingetragene Zeichenfolgen hingegen rot. Die verschlüsselte Ausgabe wird durch ihre blaue Farbe hervorgehoben.

Die Dekodierung des komprimierten Textes in Phase zwei entspricht der in Abschnitt 2.3.4 erläuterten Vorgehensweise. Auch hier entsprechen sich der Aufbau der Abbildung 2.3.4 und der in Abbildung 3.4 dargestellten Implementierung. Ist der Eintrag zum zu entschlüsselnden Index bereits im Wörterbuch eingetragen, so wird dies türkis gekennzeichnet, ansonsten wird der Vorgang zur Erstellung

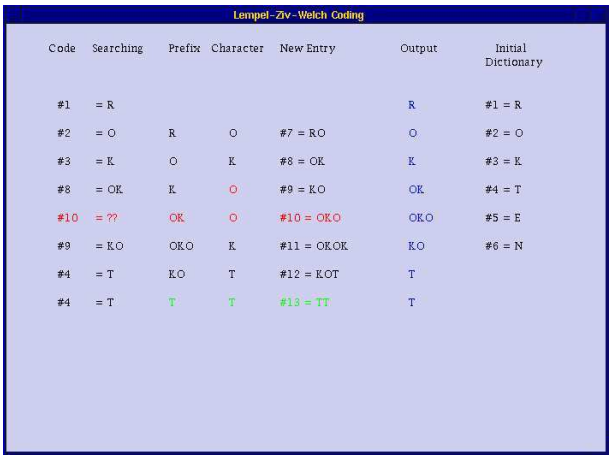

Abbildung 3.4: **Dekodierung des Lempel-Ziv-Welch-Algorithmus**

des noch nicht eingetragenen Indexes rot hervorgehoben. Wie schon bei der Kodierung wird die endgültige Ausgabe blau geschrieben.

3.2.3 Arithmetische Kodierung

Auch für die Arithmetische Kodierung gilt die Überlegung, daß eine Darstellung in zwei Phasen der Übersichtlichkeit und somit dem besseren Verständnis dient. Dementsprechend wurde für die Arithmetische Kodierung der gleiche zeitliche Aufbau wie für den LZW-Algorithmus gewählt. Wie bereits bei der Umsetzung der Lempel-Ziv-Welch-Kompression entspricht auch hier die implementierte Grafik den Abbildungen 2.12 und 2.13 aus Abschnitt 2.4.3.

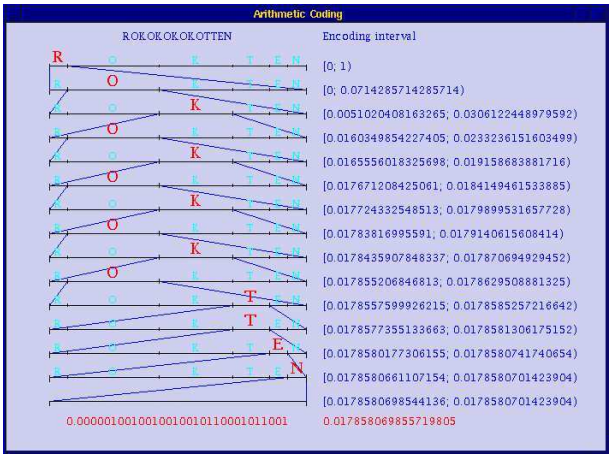

Abbildung 3.5: **Darstellung der Arithmetischen Kodierung**

In Abbildung 3.5 ist die Darstellung des Kodiervorganges zu sehen. Der Eingabetext ist oben rot dargestellt, darunter befinden sich die in ihre Teilintervalle

unterteilten Kodierungsintervalle. Der gewählte rote Buchstabe hebt sich dabei von den anderen, türkisen Buchstaben ab, die Grenzen des Intervalls werden blau neben das entsprechende Intervall eingetragen. Im letzten Schritt dieser Phase wird diejenige Zahl ausgegeben, die im zuletzt berechneten Intervall mit der kleinsten möglichen Anzahl an Bits dargestellt werden kann.

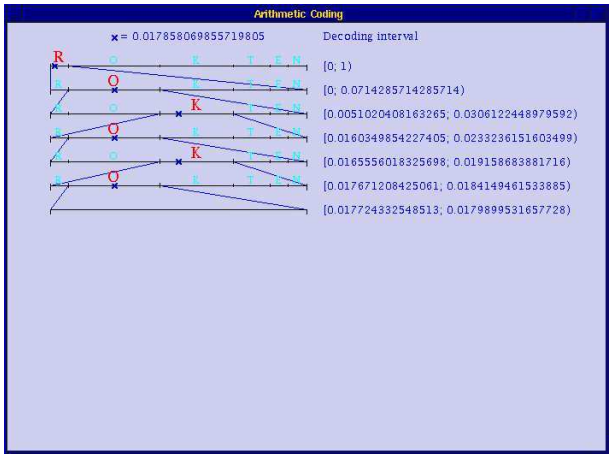

Abbildung 3.6: **Darstellung der Arithmetischen Dekodierung**

Bei der Dekodierung wurde, wie aus Abbildung 3.6 hervorgeht, auf die gleiche Weise verfahren wie bei der Kodierung. Die oben dargestellte kodierte Zahl wird mit einem Kreuz im aktuellen Intervall eingezeichnet, um zu verdeutlichen, wie die Dekodierung mittels Intervallschachtelung von statten geht. Am Ende erfolgt dann die Ausgabe des entschlüsselten Textes.

Kapitel 4

Zusammenfassung

Verlustlose Kompressionsalgorithmen kommen immer dann zum Einsatz, wenn der Verlust von Informationen bei der Rekonstruktion die Qualität der Daten beeinträchtigen würde. Ein ideales Verfahren zur Komprimierung beliebiger Daten gibt es allerdings nicht.

Vielmehr hat jeder der hier vorgestellten Algorithmen Stärken und Schwächen, die durch geschicktes Kombinieren an die jeweiligen Anforderungen angepaßt werden können. In Abschnitt 2.3.2 wurden bereits verschiedene adaptierte Verfahren vorgestellt, in denen sich die Kombination eines Lempel-Ziv-Algorithmus mit der Arithmetischen Kodierung bzw. dem Huffman-Code die Vorzüge beider Verfahren zu Nutze macht.

Da sowohl der Huffman-Code als auch die Arithmetische Kodierung nur der durch Zeichenhäufigkeit entstehenden Redundanz entgegenwirken, ist die Kombination mit einem Verfahren zur Ersetzung redundanter Zeichen- und Musterwiederholungen durchaus effizient und daher empfehlenswert. Auf Grund seiner höheren Geschwindigkeit, aber auch wegen seiner geringeren Fehleranfälligkeit erhält der Huffman-Code dabei oft den Vorzug vor der Arithmetischen Kodierung, obwohl diese, wie aus Abschnitt 2.4.1 hervorgeht, eine höhere Kompressionsrate erzielt.

Während das Prinzip des Huffman-Codes als *bottom-up*-entwickelter Baum eine Verbesserung des von Shannon und Fano entwickelten *top-down*-Ansatzes ist, erfuhr die Arithmetische Kodierung keine größere Veränderungen. Die ursprünglichen Algorithmen von Lempel und Ziv hingegen wurden über die Jahre hinweg mehrfach modifiziert.

Letzten Endes wurden in den vergangenen 25 Jahren, also seit Veröffentlichung des ersten Lempel-Ziv-Verfahrens, viele Kombinationen von Kompressionsverfahren entwickelt, die jeweils auf ganz bestimmte Eigenschaften der zu komprimierenden Daten zugeschnitten worden sind. Der Stein der Weisen wurde aber auch hier noch nicht gefunden.

Literaturverzeichnis

[1] Effelsberg, W.: *Multimediatechnik - Grundlagen der Kompressionsverfahren*; Vorlesungsskript, Kapitel 2, Abschnitt 1, (2a-)1-38

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

[3] Ziv, J.; Lempel, A.: *Compression of Individual Sequences via Variable-Rate Coding*; IEEE Transactions on Information Theory, Vol. 24 No. 5, 530-536; September 1978

[4] Storer, J. A.; Szymanski, T. G.: *Data compression via textural substitution*; Journal of the ACM, Vol. 29 No. 4, 928-951; October 1982

[5] Welch, T. A.: *A Technique for High Performance Data Compression*; IEEE Computer, Vol. 17 No. 6, 8-19; June 1984

[6] Rissanen, J.: *Generalized Kraft Inequality and Arithmetic Coding*; IBM Journal of Research and Development, Vol. 20 No. 3, 198-203; May 1976

[7] Rissanen, J.; Langdon, G. G. Jr.: *Arithmetic Coding*; IBM Journal of Research and Development, Vol. 23 No. 2, 149-162; March 1979

[8] Langdon, G. G. Jr.: *An Introduction to Arithmetic Coding*; IBM Journal of Research and Development, Vol. 28 No. 2, 135-149; March 1984

[9] Schirmer, O.: *Verlustfreie Datenkompression - Überblick und Kategorisierung*; Diplomarbeit, Technische Universität Berlin; November 1994
scara.com/∼schirmer/dipl.ps.gz

[10] Nelson, M.; Gailly, J.-L.: *The Data Compression Book*; M&T Books; 2000

[11] Ottmann, T.; Widmayer, P.: *Algorithmen und Datenstrukturen*; Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford; 4. Auflage; 2002

[12] Sedgewick, R.: *Algorithmen in C*; Addison-Wesley, München; 1. Auflage; 1993

[13] Tomczyk, M.: *Lempel-Ziv-Kodierung*; Proseminar Redundanz; Universität Karlsruhe; SS 1995
www.uni-karlsruhe.de/∼un55/lz.html

[14] Davis, C.; Grimm, N.; Trottier, N.; Chamma, P.: *Topic #22: Huffman Trees*; McGill University: School of Computer Science; Data Structures and Algorithms; Class Notes for 308-251B; Winter 1997
www.cs.mcgill.ca/∼cs251/OldCourses/1997/topic22/

[15] Khan, H.; Bejerman, P.; Lam, P.: *Topic #23: Lempel-Ziv Compression*; McGill University: School of Computer Science; Data Structures and Algorithms; Class Notes for 308-251B; Winter 1997
www.cs.mcgill.ca/∼cs251/OldCourses/1997/topic23/

[16] Hraboswki, M.: *Lempel Ziv*; Proseminar Redundanz; Universität Karlsruhe; WS 1998/99
goethe.ira.uka.de/seminare/redundanz/vortrag05/

[17] Decker, T.: *Huffman*; Proseminar Redundanz; Universität Karlsruhe; WS 1998/99
goethe.ira.uka.de/seminare/redundanz/vortrag06/

[18] Hirsch, D.: *Arithmetische Kodierung*; Proseminar Datenkompression; Rheinisch Westfälische Technische Hochschule Aachen; SS 2000
www.daniel-hirsch.de/studium/download/ArithmetischeKodierung.ps

[19] Katz, B.: *Der Huffmancode*; Proseminar Redundanz, Fehlertoleranz und Kompression; Universität Karlsruhe; WS 2001/02
goethe.ira.uka.de/seminare/rftk/huffman/

[20] Jäger, M.-A.: *Die Ziv-Lempel-Kompession*; Proseminar Redundanz, Fehlertoleranz und Kompression; Universität Karlsruhe; WS 2001/02
goethe.ira.uka.de/seminare/rftk/zivlempel

[21] Bodden, E.; Clasen, M.; Kneis, J.: *Arithmetische Kodierung*; Proseminar Datenkompression; Rheinisch Westfälische Technische Hochschule Aachen; WS 2001/02
www-users.rwth-aachen.de/eric.bodden/ac/ac.pdf

[22] *Beispiel eines Softwarepatents: Der LZW-Algorithmus*
www-pu.informatik.uni-tuebingen.de/iug/archiv/SoSe01/open_source_grupe2/LZW-Algorithmus.html

[23] *Artikel 52 EPÜ (Europäisches Patentübereinkommen): "Patentfähige Erfindungen"*
www.european-patent-office.org/legal/epc/d/ar52.html

[24] *Patentschutz für das Datenkompressionsverfahren in GIF/TIFF-Dateien*
www.dfn.de/service/ra/checkliste/GIF-Patent.html

[25] *United States Patent 4,814,746: Data compression method*
http://164.195.100.11/netacgi/nph-Parser?Sect1=PTO1 &Sect2=HITOFF &d=PALL &p=1 &u=/netahtml/srchnum.htm &r=1 &f=G &l=50 &s1='4814746'.WKU. &OS=PN/4814746 &RS=PN/4814746

[26] *United States Patent 4,558,302: High speed data compression and decompression apparatus and method*
http://164.195.100.11/netacgi/nph-Parser?Sect1=PTO2 &Sect2=HITOFF &p=1 &u=/netahtml/search-bool.html &r=1 &f=G &l=50 &co1=AND &d=ft85 &s1='4558302'.WKU. &OS=PN/4558302 &RS=PN/4558302

[27] *Computer Science Bibliography Universität Trier*
www.informatik.uni-trier.de/~ley/db/index.html