

**Dr. Jürg M. Stettbacher**

Neugutstrasse 54  
CH-8600 Dübendorf

---

Telefon: +41 43 299 57 23

E-Mail: [dsp@stettbacher.ch](mailto:dsp@stettbacher.ch)

# **Quellencodierung**

## **Grundlagen der Datenkompression**

Version 1.01  
2018-03-12

Zusammenfassung: Ausgehend von den Grundzügen der Informationstheorie werden verschiedene Verfahren der Quellencodierung behandelt. Dabei geht es um die Kompression von Daten.

# Inhaltsverzeichnis

<b>1</b>	<b>Zweck</b>	<b>3</b>
<b>2</b>	<b>Einleitung</b>	<b>3</b>
<b>3</b>	<b>Quellencodierungstheorem</b>	<b>5</b>
3.1	Quellencode . . . . .	5
3.2	Codewortlänge . . . . .	6
3.3	Redundanz . . . . .	8
3.4	Theorem zur Quellencodierung . . . . .	9
<b>4</b>	<b>Laufängencodierung</b>	<b>10</b>
<b>5</b>	<b>Huffman Codes</b>	<b>14</b>
5.1	Einführendes Beispiel . . . . .	14
5.2	Verfahren . . . . .	15
5.3	Anwendungen . . . . .	16
<b>6</b>	<b>Lempel Ziv Codes</b>	<b>20</b>
6.1	LZ77 . . . . .	21
6.2	LZ78 . . . . .	25
6.3	LZW . . . . .	27
<b>7</b>	<b>Arithmetische Codes</b>	<b>30</b>
<b>8</b>	<b>JPEG</b>	<b>30</b>

Früher produzierten wir Automobile in Massen,  
heute Information.

*John Naisbitt*

# 1 Zweck

Ziel dieses Dokuments ist es,

- Die Absicht der *Quellencodierung* aufzueigen,
- mit Hilfe der *Informationstheorie* ihre Grenzen abzustecken,
- sowie verschiedene Verfahren der Quellencodierung anhand von Beispielen<sup>1</sup> als exemplarische Vertreter vorzustellen.

# 2 Einleitung

Als Überblick betrachten wir die folgende bekannte Darstellung (siehe Abbildung 1). Eine Datenquelle<sup>2</sup> produziert einen Datenstrom. Wir nehmen vorläufig an, dass die einzelnen Datenwerte einen zufälligen Charakter haben. Diese Daten sollen nun übertragen werden. Aber der Datenkanal ist nicht perfekt, nicht sicher und die Übertragung ist nicht kostenlos. Das heisst, es entstehen Übertragungsfehler und Kosten. Dem Aspekt des Datenschutzes wollen wir an dieser Stelle nicht weiter nachgehen. Um unnötige Kosten und Übertragungsfehler zu verhindern werden die Daten der Quelle zuerst aufbereitet, bevor sie übertragen werden:

---

<sup>1</sup> In diesem Dokument sind Beispiele zur Kennzeichnung von den Symbolen ▼ und ▲ umschlossen.

<sup>2</sup> Ein Beispiel für eine Datenquellen könnte etwa eine Textdatei sein, wenn sie ausgelesen wird, oder das Mikrofon in einem Telefon, das eine Abfolge von digitalisierten Schalldruckwerten liefert, oder ein physikalischer Sensor, der über ein Netzwerk periodisch Wetterdaten verschickt.

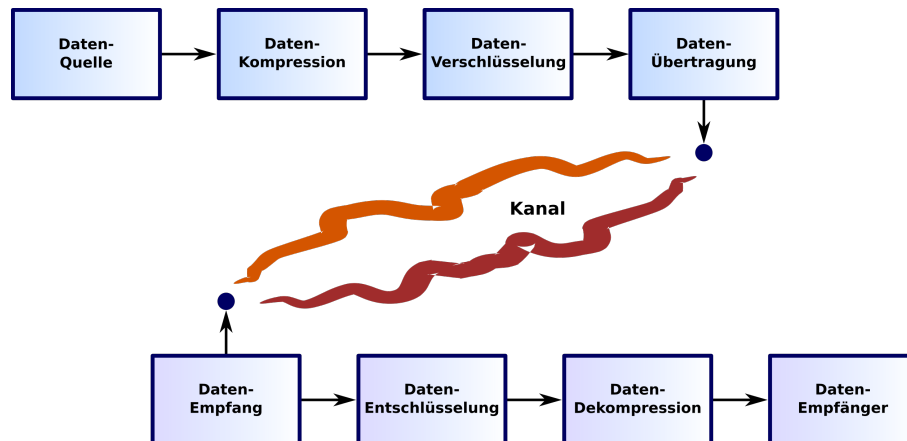


Abbildung 1: Schema eines allgemeinen Kommunikationssystems.

- Um Kosten zu sparen werden sie komprimiert.
- Zur Verhinderung von Übertragungsfehlern verwendet man einen speziellen Fehlerschutzcode.

Im Folgenden betrachten wir das Thema der Datenkompression und in dem Kontext stellen sich sofort einige Fragen, zum Beispiel diese:

Wie oft muss man eine Textdatei durch das Zip-Programm<sup>3</sup> schicken bis die Datei nur noch 1 Bit gross ist? Haben Sie sich diese Frage auch schon gestellt? Die Informationstheorie gibt darauf eine sehr einfache Antwort. Sie sagt nämlich, dass das in der Regel<sup>4</sup> gar nicht möglich ist.

Das sogenannte Quellencodierungstheorem nennt eine klare Grenze der Komprimierbarkeit und ist somit von zentraler Bedeutung wenn es zum Beispiel um die Beurteilung von Kompressionsverfahren geht. Im Folgenden wird das Theorem behandelt, bevor wir damit beginnen, einige bekannte Quellencodierungsverfahren zu erläutern. Bei den behandelten Beispielen verzichten wir auf die Betrachtung aller Details und beschränken uns auf die hauptsächlichen Prinzipien.

---

<sup>3</sup> Zip ist eine Familie von Datei-Kompressionsprogrammen. Unter Linux heisst das Programm einfach *zip*. Unter Windows heissen die bekanntesten Versionen PKZIP, WinZip, 7-Zip, etc.

<sup>4</sup> Mit *in der Regel* sei die Aussage vorerst auf eine normale Datei beschränkt, die zum Beispiel einen gewöhnlichen Satz in deutscher Sprache enthält.

### 3 Quellencodierungstheorem

Aus der Informationstheorie kennen wir die folgenden zentralen Begriffe:

- Information (in Bit)
- Entropie (in Bit/Symbol)

Zur Erinnerung: Die Information, resp. der Informationsgehalt  $I(x_n)$  bezieht sich auf ein Symbol  $x_n$ , das von einer Quelle  $Q$  erzeugt wird. Die Entropie  $H(Q)$  ist der Erwartungswert<sup>5</sup> der Information  $I(x_n)$  aller Symbole  $x_n$  der Quelle  $Q$  und bezieht sich damit auf die Quelle selbst. Wir sagen auch: Die Entropie ist die mittlere Information der Quelle pro Symbol.

Für die weiteren Betrachtungen benötigen wir nun einige neue Begriffe.

#### 3.1 Quellencode

Wir nehmen an, dass die Symbole  $x_n$  der Quelle  $Q$  schon in der Form von irgendwelchen binären Mustern daher kommen. Das ist zwar nicht zwingend, aber in einer Welt von elektronischen Informationsquellen typisch. Wir wollen zwei alltägliche Beispiele dazu ansehen (die aber gerade nicht aus der elektronischen Welt stammen).

▼ Die Quelle *Münzwurf* liefert drei Symbole  $x_n$  mit  $n = 0 \dots 2$ . Dabei steht  $x_0$  für *Kopf*,  $x_1$  für *Zahl* und  $x_2$  wird verwendet für den Fall, wo die Münze auf dem Rand stehen bleibt oder vom Tisch fällt. Die Ereignisse werden von der Quelle willkürlich gemäss der folgenden Tabelle gemeldet:

Symbol	Code
$x_0$	$\underline{c}_0 = (10)$
$x_1$	$\underline{c}_1 = (110)$
$x_2$	$\underline{c}_2 = (1110)$

---

<sup>5</sup> Der Erwartungswert ist ein gewichteter Mittelwert. Das heisst im konkreten Fall der Entropie, dass der Informationsgehalt von jedem Symbol bei der Mittelwertbildung mit seiner Auftretenswahrscheinlichkeit gewichtet wird. Seltene Symbole tragen folglich wenig, häufige Symbole tragen viel zur Entropie bei.

Beachte, dass die Codeworte unterschiedlich lang sind, was absolut zulässig ist. Werden die Codeworte allerdings seriell als Bitstrom übertragen, was oft der Fall ist, so muss darauf geachtet werden, dass kein Codewort eine Vorsilbe eines anderen Codewortes ist. Wäre zum Beispiel bei einem anderen Code das Codewort  $q_0 = (101)$  und das Codewort  $q_1 = (1011)$ , so könnte der Empfänger nach Empfangen der ersten drei Bits nicht entscheiden, ob er nun  $q_0$  bekommen hat, oder den Anfang von  $q_1$ . Man nennt Codes, bei denen kein Codewort eine Vorsilbe eines anderen Codewortes ist, *präfixfreie Codes*. ▲

▼ Wir können das Werfen eines Würfels als Datenquelle betrachten, die Zufallszahlen  $y_m$  zwischen eins und sechs liefert. Mit je drei Bits können allen Symbole dargestellt werden:

Symbol	Code
$y_1 = 1$	$c_1 = (001)$
$y_2 = 2$	$c_2 = (010)$
...	...
$y_6 = 6$	$c_6 = (110)$

Wir haben also die Symbole  $y_m$  mit  $m = 1 \dots 6$  und  $m$  entspricht gerade auch der betreffenden Augenzahl des Würfels. Zudem ist jedem Symbol  $x_m$  ein Codewort  $c_m$  zugeordnet, das aus dem Binärwert der Augenzahl besteht. Beachte, dass die Codeworte (000) und (111) im vorliegenden Beispiel nicht definiert sind und daher nie auftreten dürfen. ▲

## 3.2 Codewortlänge

Wir gehen weiterhin davon aus, dass wir es mit binären Codes zu tun haben. Die Codewortlänge  $\ell_n$  bezieht sich auf das Codewort  $c_n$  eines bestimmten Symbols  $x_n$  und gibt an, aus wievielen Bits das Codewort besteht. Wir nehmen die Beispiele von oben nochmals auf.

▼ Die Codeworte  $c_n$  mit  $n = 0 \dots 2$  beim Münzwurf haben die folgenden Längen:

Symbol	Code	Codewortlänge
$x_0$	$c_0 = (10)$	$\ell_0 = 2 \text{ Bit}$
$x_1$	$c_1 = (110)$	$\ell_1 = 3 \text{ Bit}$
$x_2$	$c_2 = (1110)$	$\ell_2 = 4 \text{ Bit}$

▲

▼ Die Codeworte der Würfel-Quelle mit den Ereignissen  $y_m$  mit  $m = 1 \dots 6$  haben alle eine Länge von drei Bit. Also ist in diesem Fall  $\ell_m = 3$  Bit für alle Codeworte  $c_m$ . ▲

Zum Schluss wollen wir die mittlere Codewortlänge  $L$  einer Quelle angeben, welche die Symbole  $x_n$  mit  $n = 0 \dots N - 1$  liefert. Dabei müssen wir berücksichtigen, dass die Codeworte  $c_n$  allenfalls unterschiedlich oft auftreten. Das tun man, indem man  $L$  als Erwartungswert der Codewortlängen  $\ell_n$  bildet:

$$L = \sum_{n=0}^{N-1} P(x_n) \cdot \ell_n \quad (\text{Bit/Symbol}) \quad (1)$$

Beachte, dass wir die Einheit *Bit/Symbol* verwenden um anzuzeigen, dass es sich bei dieser Grösse um einen Mittelwert pro Symbol handelt. Wir setzen unsere beiden Beispiele fort:

▼ Für den Münzwurf müssen wir uns zuerst die Wahrscheinlichkeiten  $P(x_n)$  beschaffen. Zu diesem Zweck werfen wir die Münze sehr oft (zum Beispiel 10'000 mal) und zählen aus, wie häufig jedes Ereignis auftritt. Wenn wir dann noch durch die gesamte Anzahl Würfe dividieren, erhalten wir eine Approximation von  $P(x_n)$ .

Symbol	Code	Codewortlänge	Wahrscheinlichkeit
$x_0$	$c_0 = (10)$	$\ell_0 = 2$ Bit	$P(x_0) = 0.45$
$x_1$	$c_1 = (110)$	$\ell_1 = 3$ Bit	$P(x_1) = 0.47$
$x_2$	$c_2 = (1110)$	$\ell_2 = 4$ Bit	$P(x_2) = 0.08$

Beachte, dass die Summe aller Wahrscheinlichkeiten eins sein muss. Wir können jetzt die mittlere Codewortlänge ausrechnen:

$$L = P(x_0) \cdot \ell_0 + P(x_1) \cdot \ell_1 + P(x_2) \cdot \ell_2 = 2.65 \text{ Bit/Symbol}$$

Würde man also die Quelle sehr lange beobachten und schliesslich die mittlere Länge über alle gesehenen Codeworte bilden, so würde das selbe Resultat von 2.65 Bit/Symbol heraus kommen. ▲

▼ Beim Würfelexperiment waren von Anfang an alle Codeworte gleich lang, nämlich 3 Bit. Unabhängig von den Wahrscheinlichkeiten wird demnach auch die mittlere Codewortlänge  $L = 3$  Bit/Symbol. ▲

### 3.3 Redundanz

Genau genommen gibt es zwei Definitionen der Redundanz:

- Die Redundanz einer Quelle<sup>6</sup>.
- Die Redundanz eines Codes.

Da wir nur Quellen betrachten, die bereits einen Code ausgeben, interessieren wir uns im Folgenden nur für die Redundanz von Codes. Die Definition dieser Redundanz  $R$  lautet:

$$R = L - H \quad (\text{Bit/Symbol}) \quad (2)$$

Die Redundanz ist also die Differenz von mittlerer Codewortlänge  $L$  und Entropie  $H$  einer Quelle.

Zur Erinnerung: Die Entropie  $H$  ist die mittlere Information pro Symbol einer Quelle. Wir geben sie in Bit/Symbol an, genauso wie die mittlere Codewortlänge. Die Entropie sagt uns also, wieviele Bits pro Codewort (minimal) notwendig sind, um die Information der Quelle zu codieren.

Die mittlere Codewortlänge gibt an, wieviele Bits tatsächlich pro Codewort vorhanden sind. Ist also  $R > 0$ , so umfasst der Code mehr Bits als notwendig sind, um die Information wiederzugeben. Ist  $R < 0$ , so reicht die Anzahl Bits pro Codewort nicht aus, um die Information darzustellen. Die Information der Quelle lässt sich demnach mit einem derartigen Code nicht transportieren. Ein Teil der Information geht zwangsläufig verloren.

Wir möchten nochmals die Beispiele von oben fortsetzen:

▼ Für den *Münzwurf* haben wir oben bereits die Wahrscheinlichkeiten der Symbole  $x_n$  mit  $n = 0 \dots 2$  beziffert.

Symbol	Code	Codewortlänge	Wahrscheinlichkeit
$x_0$	$\underline{c}_0 = (10)$	$\ell_0 = 2 \text{ Bit}$	$P(x_0) = 0.45$
$x_1$	$\underline{c}_1 = (110)$	$\ell_1 = 3 \text{ Bit}$	$P(x_1) = 0.47$
$x_2$	$\underline{c}_2 = (1110)$	$\ell_2 = 4 \text{ Bit}$	$P(x_2) = 0.08$

Damit folgt sofort die Entropie  $H$ :

$$H = P(x_0) \cdot \log_2 \frac{1}{P(x_0)} + P(x_1) \cdot \log_2 \frac{1}{P(x_1)} + P(x_2) \cdot \log_2 \frac{1}{P(x_2)} = 1.32 \text{ Bit/Symbol}$$

<sup>6</sup> Die Redundanz  $R$  einer Quelle ist definiert als  $R_Q = \log_2(N) - H$ , wenn  $N$  Anzahl verschiedener Symbole der Quelle ist, und  $H$  die Entropie der Quelle. In unserem Kontext hat diese Form der Redundanz jedoch keine Bedeutung.



Die mittlere Codewortlänge  $L$  haben wir oben schon berechnet:

$$L = 2.65 \text{ Bit/Symbol}$$

Damit folgt die Redundanz  $R$ :

$$R = L - H = 1.33 \text{ Bit/Symbol}$$

Man erkennt, dass bei diesem Code rund die Hälfte aller Bits keine informationstragende Funktion haben. ▲

▼ Beim Würfeln hat jede der sechs Augenzahlen dieselbe Auftretenswahrscheinlichkeit, nämlich  $P(y_m) = 1/6$  mit  $m = 1 \dots 6$ . Damit folgt die Entropie  $H$ :

$$H = \sum_{m=1}^6 P(y_m) \cdot \log_2 \frac{1}{P(y_m)} = \log_2(6) = 2.59 \text{ Bit/Symbol}$$

Mit der oben schon gefundenen mittlere Codewortlänge  $L = 3 \text{ Bit/Symbol}$  folgt die Redundanz:

$$R = L - H = 0.41 \text{ Bit/Symbol}$$

▲

### 3.4 Theorem zur Quellencodierung

Das Theorem betrachtet eine Quelle, resp. einen Code und geht aus von der Redundanz  $R$ :

Solange die Redundanz  $R$  eines Codes grösser als null ist, kann verlustfrei komprimiert werden. Falls  $R \leq 0$ , so kann nur noch verlustbehaftet komprimiert werden.

Ausgedrückt durch die mittlere Codewortlänge  $L$  und die Entropie  $H$ : Man kann verlustfrei komprimieren, solange  $L > H$  ist. Im Idealfall der verlustfreien Kompression wird demnach  $L = H$ . Im letzteren Fall lässt sich nur noch verlustbehaftet weiter komprimieren.

Das Ziel der Quellencodierung besteht normalerweise darin, die Redundanz mit vertretbarem Aufwand zu entfernen oder mindestens zu reduzieren. Zu diesem Zweck wird der vorliegende Code (resp. die betreffenden Symbole) in einen neuen Code übersetzt, der bezüglich Redundanz bessere Eigenschaften hat. Die folgenden Kapitel behandeln entsprechende Verfahren.

# 4 Lauflängencodierung

Die Lauflängencodierung (englisch *Run Length Encoding*, RLE) ist eine lose Sammlung von ähnlichen Methoden zur Komprimierung von Sequenzen (englisch *Runs*) identischer Symbole. Die zentrale Idee ist diese:

Ketten von identischen Zeichen werden zusammengefasst.

Es werden im folgenden zwei typische Anwendungen gezeigt.

**Strings von Zeichen** Betrachten wir beispielsweise den folgenden String, der nur aus binär codierten Grossbuchstaben besteht:

...TERRRRRRRRRMAUIIIIIIIIIIIIIIIIIIIWQCSSSSSSSSSSL...

Man erkennt sofort, dass sich immer wieder einzelne Symbole mehrfach wiederholen. Diese Wiederholungen könnte man zusammenfassen, zum Beispiel so:

...TE (9xR) MAU (17xI) WQC (10xS) L...

Offensichtlich ist der String jetzt kürzer. Wir haben jeden Run zur Markierung in Klammern gesetzt und in der Klammer angegeben, wie oft welches Zeichen vorkommt. Wir nennen das auch einen *Token*. Der Empfänger erkennt mit Hilfe der Klammer den Run und kann ihn wieder expandieren. Aber es gibt ein Problem: Wir verwenden nun Klammerzeichen um die Token zu markieren, aber Klammern kommen im ursprünglichen Zeichensatz gar nicht vor. Nebenbei sei noch erwähnt, dass das kleine *x* im ursprünglichen Zeichensatz genauso wenig vorkommt. Statt den Klammern ist also ein anderer Marker gesucht, einer, der im Zeichensatz vorkommt, aber selten gebraucht wird. Wir wollen nun annehmen, dass die Analyse einer längeren Sequenz ergibt, dass *A* das seltenste Zeichen ist. Damit erhält man:

...TEA09RMA01AUA17IWQCA10SL...

Zur Verbesserung der Sichtbarkeit wurden die zusammengefassten Runs unterstrichen. Betrachten wir den ersten Token A09R, so ist *A* der Marker, der einen Run einleitet. *09* symbolisiert den Zähler. Beachte: Ziffern kommen im Zeichensatz ebenfalls nicht vor. Aber wir können den Zähler ohne Weiteres binär mit einer fixen Breite einsetzen, zum Beispiel mit 6 Bit. Damit könnten Runs bis zur Länge 63 codiert werden. Das kleine *x* wurde fallen gelassen. Statt dessen folgt gleich das Zeichen, das im

Run wiederholt wird. Der Empfänger muss das Format des Tokens selbstverständlich kennen. Dann aber kann er nach jedem Marker einfach die nächsten 6 Bit lesen, als Zahl, resp. Zähler interpretieren, dann das nächste Zeichen lesen und entsprechend dem Zähler wiederholen.

Wir wollen noch den zweiten Token A01A ansehen, denn dieser kam im Vorschlag mit Klammern noch gar nicht vor. Im ursprünglichen String steht an dieser Stelle allein der Buchstabe A, der uns nun als Marker dient. Wenn der Empfänger das A liest, wird er es als Marker verstehen und die folgenden Bits entsprechend decodieren. Aus diesem Grund müssen wir zwingend bei jedem Auftreten des Markers im ursprünglichen Text, ihn durch einen Run der Länge eins ersetzen. Das wurde hier gemacht.

Zusammenfassend lautet die Anleitung für das eben beschriebene Verfahren so:

- Definiere vorgängig einen Marker, resp. ein gültiges Zeichen, das in den zu verarbeitenden Texten selten vorkommt.
- Definiere vorgängig eine Zählerbreite (in Bits), so dass Runs der typischen Länge damit erfasst werden können.
- Ersetze im Urtext jeden Run durch einen Token. Ist der Run länger, als was der Token abbilden kann, so bilde mehrere Token hintereinander.
- Ersetze im Urtext alle verbleibenden Marker-Zeichen durch einen Token mit der Run-Länge eins.

Der Decoder sucht lediglich nach Markern und expandiert die betreffenden Token.

**Bilder (serialisiert)** In einfachen Bildern mit Flächen derselben Farbe lässt sich ein ähnliches Verfahren anwenden. Als Beispiel dient Abbildung 2. Jeder Pixel besteht aus einem Bit und kann nur schwarz (0) oder weiss (1) sein. Das Bild ist 24 Pixel breit und 11 Pixel hoch. Total umfasst das Bild also 264 Pixel an je ein Bit, also 33 Byte, wenn das Bild unkomprimiert gespeichert wird.

Mit Hilfe der Lauflängencodierung lassen sich nun Pixel identischer Farbe zusammenfassen. Zu diesem Zweck wird das Bild zuerst serialisiert. Das heisst, wir denken uns die einzelnen Zeilen alle aneinander aufgereiht. Selbstverständlich wäre das mit Spalten genauso möglich, aber wir wählen hier die zeilenweise Serialisierung.

Beginnend mit schwarzen Pixeln wird nun im serialisierten Bild gezählt, wieviele solche Pixel auftreten bis zum ersten weissen Pixel. Dann werden die weissen Pixel gezählt, bis wieder ein schwarzer auftritt, und so weiter. Wir erhalten die folgende Zahlenreihe (die Klammern, Kommas und Abstände dienen nur der Darstellung und würden im binären String selbstverständlich nicht vorkommen):

( 65, 1, 23, 2, 8, 17, 7, 18, 6, 17, 21, 2, 22, 1, 54 )

Die Summe all dieser Zahlen muss wieder 264 ergeben, die Gesamtzahl der Pixel im Bild. Da aus der Zahlenreihe die Dimension des Bildes (24 x 11 Pixel) nicht mehr ersichtlich ist, muss dies der Zahlenreihe noch voran gestellt sein. Somit erhalten wir als komprimierte Variante des Bildes diesen Zahlenvektor:

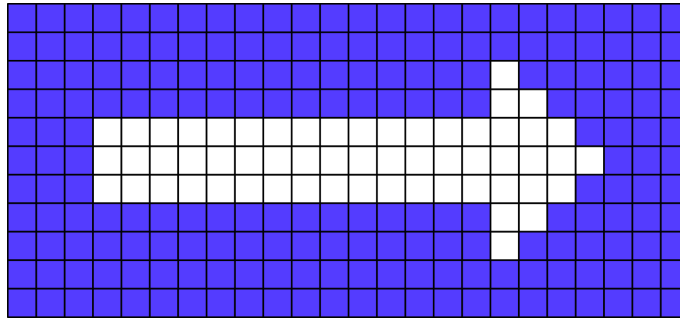


Abbildung 2: Bild mit einem Bit pro Pixel.

( 24, 11, 65, 1, 23, 2, 8, 17, 7, 18, 6, 17, 21, 2, 22, 1, 54 )

Offen ist noch die Frage, nach der Anzahl Bits, mit der jede Zahl dargestellt wird. Selbstverständlich müssen wir jede Zahl in gleicher Weise darstellen. Ein konservativer Vorschlag wäre 9 Bit pro Zahl. Damit kann man bis 511 zählen. Würde das Bild also nur aus schwarzen Pixeln bestehen, so könnte das gesamte Bild mit nur einer Zahl (plus Dimension) dargestellt werden. Unser Vektor besteht aus 17 Zahlen, total also aus 153 Bit. Damit wird das Bild nicht ganz auf die halbe Grösse komprimiert.

Da Bilder mit nur einer Farbe nicht allzu spannend sind und in der Praxis vermutlich nicht allzu oft auftreten, könnte man sich auch darauf einigen, beispielsweise nur 6 Bit pro Pixel vorzusehen. Damit kann man Runs bis zur Länge 63 zählen. Damit lässt sich allerdings die dritte Zahl in unserem Vektor nicht mehr darstellen. Darum muss man die 65 ersetzen durch 63, 0, 2. Das bedeutet, dass zuerst 63 schwarze Pixel kommen, dann null weisse und nochmals 2 schwarze, total also 65 schwarze Pixel nacheinander. Wir erhalten den neuen Zahlenvektor:

( 24, 11, 63, 0, 2, 1, 23, 2, 8, 17, 7, 18, 6, 17, 21, 2, 22, 1, 54 )

Es resultieren folglich 19 Werte an je 6 Bit, total also 114 Bit, was eine Verbesserung gegenüber dem ersten Vorschlag ist. Wir geben noch die Kompressionsrate<sup>7</sup>  $R$  an als Quotient von komprimierten Bits durch originale Bits:

$$R = \frac{114}{264} \approx 0.43$$

Zusammenfassend lässt sich das Verfahren so beschreiben:

- Definiere vorgängig die Bitbreite des Pixelzählers<sup>8</sup>.

---

<sup>7</sup> Die Kompressionsrate wird zwar oft mit dem Symbol  $R$  bezeichnet, wie die Redundanz. Die beiden Grössen haben aber nichts miteinander zu tun und dürfen nicht verwechselt werden.

- Definiere vorgängig die Pixelfarbe mit der begonnen wird.
- Serialisierung des Bildes.
- Falls der erste Pixel nicht die vordefinierte Farbe hat, setze die erste Pixelzahl auf null.
- Zähle jeweils alle Pixel gleicher Farbe bis zum nächsten Farbwechsel.
- Sollte eine Pixelzahl grösser werden als das Maximum des Zählers, so teile die Zahl auf, wie oben gezeigt.

**Bilder (zeilenweise)** Statt die Zeilen eines Bildes zuerst aneinander zu reihen, um dann zusätzlich die Bilddimension angeben zu müssen, könnte man das Bild auch zeilenweise komprimieren. Das lohnt sich vor allem bei grösseren Bildern, bedingt aber, dass ein bestimmter Zahlwert (in der Regel der Maximalwert) reserviert wird, um anzuzeigen, wann eine Zeile fertig ist. Im Übrigen wäre das Verfahren identisch mit dem zuvor beschriebenen.

Als Beispiel wird das Bild in Abbildung 2 nochmals aber zeilenweise komprimiert. Dabei sei der Pixelzähler 5 Bit breit und wir beginnen wieder mit schwarzen Pixeln. Mit 5 Bit können wir 0 bis 31 Pixel zählen, wobei wir den Wert 31 für das Zeilenende reservieren. Damit folgt der Zahlenvektor (Bildzeilen getrennt):

( 24, 31,  
24, 31,  
17, 1, 6, 31,  
17, 2, 5, 31,  
3, 17, 4, 31,  
3, 18, 3, 31  
3, 17, 4, 31,  
17, 2, 5, 31,  
17, 1, 6, 31,  
24, 31,  
24, 31 )

Der Vektor besteht nun aus 36 Einträgen mit total 180 Bit. Beachte, dass die Bilddimension nicht mehr angegeben werden muss, da jede Zeile genau 24 Pixel (Bildbreite) umfasst und sich zusammen mit der Anzahl Zeilen (Anzahl des Wertes 31) die Bilddimension automatisch ergibt. Die Kompressionsrate  $R$  beträgt:

$$R = \frac{180}{264} \approx 0.68$$

---

<sup>8</sup> Die Bitbreite sollte so gewählt werden, dass damit auch die Bilddimension wiedergegeben werden kann. Alternativ kann für die Bilddimension (die ersten beiden Werte im Vektor) eine separate Bitbreite definiert werden.

# 5 Huffman Codes

Idee:

Häufige Symbole erhalten kurze Codes.  
Seltene Symbole erhalten lange Codes.

Huffmans<sup>9</sup> Methode zählt zu den statistischen Kompressionsverfahren. Voraussetzung für die Bildung eines derartigen Codes ist die a priori Kenntnis der Symbol-Wahrscheinlichkeiten  $P(x_n)$ . Nach dem Huffmanverfahren erzeugte Codes sind automatisch präfixfrei. Was das bedeutet wurde weiter oben in Kapitel 3.1 schon erläutert.

Unter allen präfixfreien Codes sind Huffman Codes optimal, das heisst, man kann keinen besseren präfixfreien Code finden. Allerdings stimmt das nur dann, wenn die vorausgesetzten Wahrscheinlichkeiten  $P(x_n)$  auch tatsächlich zutreffen. Wird also beispielsweise ein Huffman Code für die buchstabenweise Kompression von deutschem Text entwickelt, so erzielt er mit englischem Text keine optimalen Resultate, da sich in den beiden Sprachen die Buchstabenhäufigkeiten<sup>10</sup> unterscheiden.

## 5.1 Einführendes Beispiel

Eine Quelle  $Q$  liefere vier verschiedene Symbole, nämlich  $A$ ,  $B$ ,  $C$  und  $D$ . Die Quelle verwendet den folgenden, naheliegend Binärcode für die Symbole:

Symbol	Nummerncode
$A$	(00)
$B$	(01)
$C$	(10)
$D$	(11)

---

<sup>9</sup> David Albert Huffman (1925 bis 1999), amerikanischer Pionier der Computerwissenschaften.

<sup>10</sup> Man erkennt das leicht, wenn man sich zum Beispiel in Erinnerung ruft, dass das Zeichen  $y$  in der deutschen Sprache sehr selten auftritt, in der englischen dagegen recht häufig.

Es wurden die Symbole also einfach binär nummeriert. Daher nennen wir einen solchen Code auch einen *Nummerncode*. Im Folgenden werden wir zeigen, dass dies oft nicht die beste Lösung ist. Betrachten wir nun die Wahrscheinlichkeiten der Symbole:

$$P(A) = 0.60 \quad P(B) = 0.30 \quad P(C) = 0.05 \quad P(D) = 0.05$$

Damit folgt die Entropie  $H(Q)$  der Quelle  $Q$ :

$$H(Q) = \sum_{x=A}^D P(x) \cdot \log_2 \frac{1}{P(x)} = 1.40 \text{ Bit/Symbol}$$

Da der oben vorgeschlagene Nummerncode die mittlere Codewortlänge  $L_N = 2$  Bit hat, erhalten wir eine Redundanz von  $R_N = 0.60$  Bit. Das Quellencodierungstheorem besagt, dass man einen Code finden könne, der ohne Informationsverlust die Symbole A bis D mit durchschnittlich  $L_{opt} = H(Q) = 1.40$  Bit/Symbol codiert.  $L_{opt}$  bezeichnet dabei die optimale mittlere Codewortlänge. Die Frage ist: Wie findet man einen solchen Code?

Huffmans Idee ist es nun, unterschiedlich lange Codewort zu verwenden. Je häufiger ein Symbol auftritt, umso kürzer soll sein Codewort sein. In unserem Beispiel resultiert nach Huffman der folgende Code (wir werden anhand von anderen Beispielen gleich zeigen, wie man darauf kommt):

Symbol	Huffman Code
A	(1)
B	(01)
C	(001)
D	(000)

Man erhält nun die mittlere Codewortlänge  $L_H$  des Huffman Codes:

$$L_H = \sum_{x=A}^D P(x) \cdot \ell(x) = 1.50 \text{ Bit/Symbol}$$

Das bedeutet, wir erreichen die perfekte Lösung  $L_{opt} = H(Q) = 1.40$  Bit/Symbol nicht ganz, sind aber schon recht nahe.

## 5.2 Verfahren

Einen Huffman Code findet man mit dem folgenden grafischen Verfahren. Es handelt sich dabei um einen sogenannten Huffman-Baum, der oben Blätter hat, darunter Äste und schliessliche zuunterst einen Stamm<sup>11</sup>.

---

<sup>11</sup> Selbstverständlich kann man den Baum auch liegend zeichnen. In der Regel sind die Blätter dann links und der Stamm rechts.

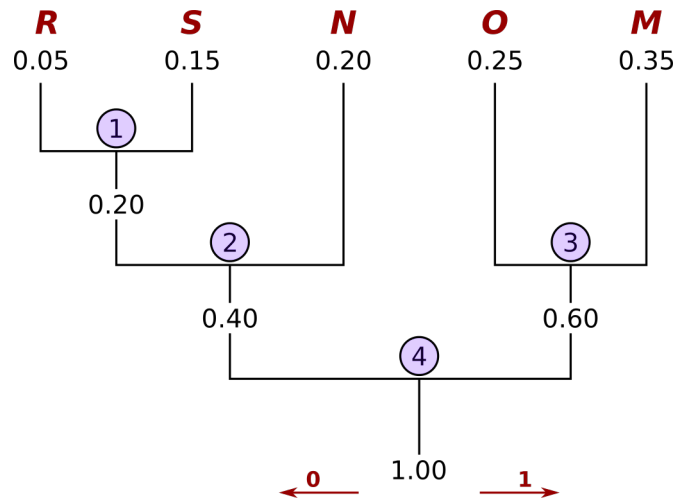


Abbildung 3: Huffman-Baum für die Quelle  $\Psi$ .

1. Ordne alle Symbole nach aufsteigenden Auftretenswahrscheinlichkeiten auf einer Zeile. Dies sind die Blätter des Huffman-Baums.  
Gibt es Symbole mit gleichen Wahrscheinlichkeiten, so spielt die Reihenfolge unter ihnen keine Rolle.
2. Notiere unter jedes Blatt seine Wahrscheinlichkeit.
3. Schliesse die beiden Blätter mit der kleinsten Wahrscheinlichkeit an einer gemeinsamen Astgabel an. Ordne dem Ast die Summe der Wahrscheinlichkeiten der beiden Blätter zu.  
Gibt es mehrere mögliche Kombination von Blättern mit den kleinsten Wahrscheinlichkeiten, so spielt es keine Rolle, welche man davon auswählt.
4. Wiederhole Punkt 3 mit Blättern und Ästen so lange, bis nur noch der Stamm des Baums übrig bleibt.
5. Nun wird festgelegt, ob bei jeder Astgabel der linke Zweig eine 0 oder eine 1 erhält. Der rechte Zweig erhält dann das Komplement.
6. Nun werden auf dem Pfad vom Stamm zu jedem Blatt die Nullen und Einsen ausgelesen und von links nach rechts nebeneinander geschrieben. Dies sind die Huffman-Codeworte.

## 5.3 Anwendungen

Anhand von Beispielen werden einige Anwendungen erläutert.



▼ Gegeben sei eine Quelle  $\Psi$ , welche die Symbole  $M$ ,  $N$ ,  $O$ ,  $R$  und  $S$  produziert. Die folgenden Wahrscheinlichkeiten sind gegeben:

$$P(M) = 0.35 \quad P(N) = 0.20 \quad P(O) = 0.25 \quad P(R) = 0.05 \quad P(S) = 0.15$$

Wir überprüfen noch kurz, ob nichts fehlt:

$$P(M) + P(N) + P(O) + P(R) + P(S) = 1.00$$

Es ist also alles in Ordnung. Nun wenden wir das oben beschriebene Huffman-Verfahren an, siehe Abbildung 3. Oben sind die nach Wahrscheinlichkeiten sortierten Blätter aufgetragen. Die Zahlen im Kreis geben die Reihenfolge an, in der die Äste gezeichnet werden. In jedem neuen Ast ist seine Wahrscheinlichkeit eingetragen, der Stamm hat folglich die Summe aller Wahrscheinlichkeiten, also eins. Beim Stamm sind mit den horizontalen Pfeilen noch die Ausleserichtungen angegeben. Suchen wir also beispielsweise das Codewort für das Symbol  $N$ , so beginnen wir mit Auslesen unten und fahren zuerst den Stamm hoch. Dann, beim Punkt Nr. 4, fahren wir nach links. Dies ergibt eine Null im Codewort. Dann fahren wir den Ast weiter nach oben. Beim Punkt Nr. 2 geht es nach rechts, was eine Eins im Codewort ergibt. Auf diesem Weg erreicht man schliesslich das Blatt  $N$ . Das Codewort dieses Blattes ist also (01). In analoger Weise findet man alle Codeworte mit den betreffenden Codewortlängen.

Symbol	Huffman Code	Codewortlänge
$M$	(11)	$\ell(M) = 2 \text{ Bit}$
$O$	(10)	$\ell(O) = 2 \text{ Bit}$
$N$	(01)	$\ell(N) = 2 \text{ Bit}$
$S$	(001)	$\ell(S) = 3 \text{ Bit}$
$R$	(000)	$\ell(R) = 3 \text{ Bit}$

Nun soll überprüft werden, wie erfolgreich dieser Code ist: Auf Grund der Wahrscheinlichkeiten folgt die Entropie  $H(\Psi) = 2.12 \text{ Bit/Symbol}$ . Die mittlere Codewortlänge resultiert zu  $L = 2.20 \text{ Bit/Symbol}$ . Es verbleibt also eine Redundanz von  $R = 0.08 \text{ Bit/Symbol}$ , was weniger als 4 % der mittleren Codewortlänge entspricht. Das Potenzial für Verbesserungen ist folglich nicht sehr gross. ▲

▼ Eine Quelle  $\Omega$  liefert statistisch unabhängige Symbole  $X$ ,  $Y$  und  $Z$  mit den Wahrscheinlichkeiten:

$$P(X) = 0.80 \quad P(Y) = 0.10 \quad P(Z) = 0.10$$

Den Huffman-Baum zeichnen wir diesmal horizontal, siehe Abbildung 4. Die Analyse liefert die folgenden Resultate, wobei wir anschliessend gleich sehen werden, warum die Grössen hier mit dem Index 1 versehen sind:

$$\begin{aligned} H_1(\Omega) &= 0.922 \text{ Bit/Symbol} \\ L_1 &= 1.200 \text{ Bit/Symbol} \\ R_1 &= 0.278 \text{ Bit/Symbol} \end{aligned}$$

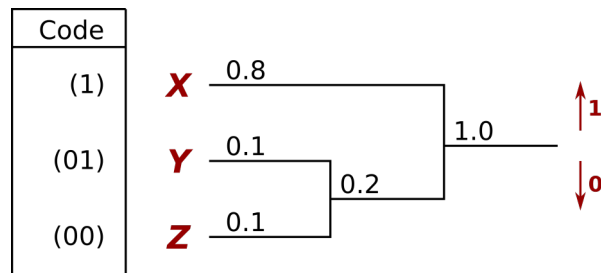


Abbildung 4: Huffman-Baum für die Quelle  $\Omega$ .

Fazit: Nahezu ein Viertel des Codes ist immer noch redundant. Das ist kein wirklich befriedigendes Resultat. Es fragt sich nur, Wie man das verbessern könnte, denn mit den drei Symbolen und ihren Wahrscheinlichkeiten ist kein anderer Huffman-Code möglich.

Das Rezept dafür ist das Folgende: Man bildet alle möglichen Doppelsymbole, das heisst, man fasst immer zwei aufeinander folgende Symbole der Quelle zusammen und codiert sie als Paar. Auf diese Weise vervielfacht sich die Anzahl Symbole, was zu einer feineren Granularität des Huffman-Codes führt und schliesslich eine bessere Annäherung an den Wert der Entropie zulässt. Falls das Resultat immer noch nicht zufriedenstellend sein sollte, so kann man Dreifachsymbole oder noch grössere Symbol-Gruppen bilden. Sind die betreffenden Symbole statistisch unabhängig voneinander, so lassen sich die Wahrscheinlichkeiten der Mehrfachsymbole (Verbundwahrscheinlichkeiten) als Produkt der Wahrscheinlichkeiten der involvierten Einzelsymbole berechnen. Besteht eine statistische Abhängigkeit zwischen Symbolen, so müssen die Verbundwahrscheinlichkeiten anderweitig ermittelt werden, zum Beispiel experimentell.

Wir wollen nun alle möglichen Doppelsymbole der Quelle  $\Omega$  mit den dazu gehörigen Wahrscheinlichkeiten auflisten. Von den drei Symbolen  $X, Y, Z$  kann jedes Symbol an erster Stelle und an zweiter Stelle auftreten. Folglich gibt es  $3 \cdot 3 = 9$  mögliche Doppelsymbole.

$$\begin{array}{lll}
 XX: & P(XX) = 0.64 & YX: & P(YX) = 0.08 & ZX: & P(ZX) = 0.08 \\
 XY: & P(XY) = 0.08 & YY: & P(YY) = 0.01 & ZY: & P(ZY) = 0.01 \\
 XZ: & P(XZ) = 0.08 & YZ: & P(YZ) = 0.01 & ZZ: & P(ZZ) = 0.01
 \end{array}$$

Damit können wir nun einen neuen Huffman-Baum zeichnen, siehe Abbildung 5. In der Abbildung sind als didaktische Hilfe wieder die Entstehungsschritte des Baumes eingezeichnet (Zahlen in Kreisen). Im Normalfall kann man diese Nummern weglassen. Man erkennt sofort, dass der Huffman-Baum jetzt sehr viel grösser ist als jener mit Einfachsymbolen. Die Analyse liefert folgende Resultate:

$$\begin{array}{ll}
 H_2(\Omega) & = 1.844 \text{ Bit} / 2 \text{ Symbole} \\
 L_2 & = 1.920 \text{ Bit} / 2 \text{ Symbole} \\
 R_2 & = 0.076 \text{ Bit} / 2 \text{ Symbole}
 \end{array}$$

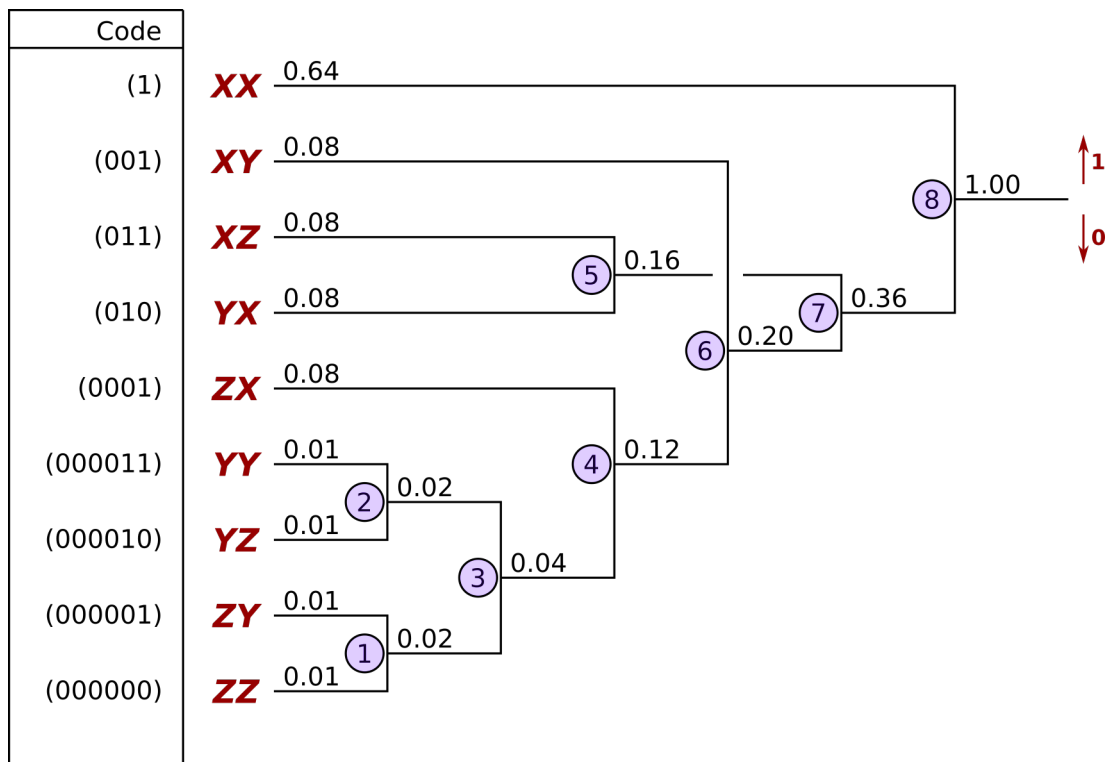


Abbildung 5: Huffman-Baum für Doppelsymbole der Quelle  $\Omega$ .

Beachte: Alle diese Grössen beziehen sich jetzt auf Doppelsymbole. Das wird mit dem Index 2 angezeigt. Ferner ist die Einheit dieser Grössen jetzt nicht mehr Bit pro Quellensymbol, sondern Bit pro *zwei* Symbole. Wollen wir also diese Resultate mit dem Huffman-Code von Einfachsymbolen vergleichen, so muss man diesem Umstand Rechnung tragen, und zwar so:

$$\begin{array}{lll} L_1 = 1.200 \text{ Bit / 1 Symbol} & \Longleftrightarrow & \frac{L_2}{2} = 0.960 \text{ Bit / 1 Symbol} \\ R_1 = 0.278 \text{ Bit / 1 Symbol} & \Longleftrightarrow & \frac{R_2}{2} = 0.038 \text{ Bit / 1 Symbol} \end{array}$$

Auf den ersten Blick mag erstaunen, dass mit dem zweiten Code die mittlere Codewortlänge  $L_2/2$  weniger als ein Bit pro Symbol beträgt. Das ist darum möglich, weil jedes Codewort (das kürzeste davon ist nur ein Bit lang) zwei Quellensymbole transportiert. Die Redundanz ist nun nur noch knapp 4 % der mittleren Codewortlänge. ▲

## 6 Lempel Ziv Codes

Die von Lempel<sup>12</sup> und Ziv<sup>13</sup> begründete Familie der sogenannten LZ-Codierungsverfahren basiert immer auf einer Art von Wörterbuch. Dabei liegt kein von Anfang an gegebenes Wörterbuch<sup>14</sup> vor, sondern das Wörterbuch wird während dem Kompressionsverfahren erst aufgebaut. Aus diesem Grund arbeiten diese Verfahren nicht sofort effizient. Bei sehr grossen Datensätzen ist die Kompression jedoch (nahezu) optimal. Man bezeichnet die Verfahren daher als *asymptotisch optimal*.

Die zentrale Idee der LZ-Verfahren ist diese:

Jede Sequenz von Quellensymbolen wird durch einen Index in ein Wörterbuch ersetzt.

Dazu ist zu sagen, dass die Indizes eines LZ-Verfahrens immer gleiche Länge haben, wogegen die Länge der Symbol-Sequenzen im Verlauf des Prozesses ständig zunimmt.

Die LZ-Verfahren arbeiten typischerweise Byte-basiert. Aus diesem Grund werden wir sie anhand von ASCII-Texten illustrieren. Beachte aber, dass die Methoden nicht auf Texte oder Bytes beschränkt sind.

---

<sup>12</sup> Abraham Lempel (geboren 1936), israelischer Informatiker.

<sup>13</sup> Jacob Ziv (geboren 1931), israelischer Informationstheoretiker.

<sup>14</sup> Es gibt andere Verfahren, die mit einem gegebenen Wörterbuch arbeiten. Allerdings sind dann Vorkehrungen notwendig für den Fall, dass ein zu komprimierendes Wort im Wörterbuch nicht enthalten ist.

### 6.1 LZ77

Das LZ77-Verfahren verwendet ein sogenanntes *Sliding Window*. Das heisst, der zu komprimierende String wandert durch einen Buffer, der in diesem Fall in zwei Bereiche aufgeteilt ist, nämlich in den Vorschau-Buffer und den Such-Buffer.

Die folgenden Erläuterungen beziehen sich auf Abbildung 6. Auf der ersten Zeile (1) sind Such- und Vorschau-Buffer noch leer. Das ist der Zustand unmittelbar, bevor der Prozess startet. Rechts ist der Textstring dargestellt, der nun komprimiert werden soll. Er wird während dem Prozess von rechts nach links (Pfeilrichtung) durch die beiden Buffer fliessen.

Vorerst fliesst der Text von rechts in den Vorschau-Buffer (Zeile 2 in Abbildung 6), bis zur Grenze zwischen Vorschau- und Such-Buffer. Es sind jetzt fünf Zeichen im Vorschau-Buffer. Alle anderen sind für das Verfahren noch nicht sichtbar. Nun wird geprüft, ob es eine Übereinstimmung zwischen Such- und Vorschau-Buffer gibt, beginnend mit jenem Zeichen, das im Vorschau-Buffer ganz links steht. Da der Such-Buffer noch leer ist, gibt es gegenwärtig keine Übereinstimmung. Schliesslich wird ein Token gebildet, der dies beschreibt. Jeder Token hat die folgende Gestalt:

Token: (Offset, Länge, Zeichen)

Dabei bezeichnet der *Offset* die Distanz zwischen dem ersten Zeichen links im Vorschau-Buffer und dem Ort der besten Übereinstimmung im Such-Buffer. Die *Länge* gibt an, wieviele Zeichen übereinstimmen. Und das *Zeichen* ist ein zusätzliches Zeichen, das im Token mit codiert wird. Dieses zusätzliche Zeichen ist nötig, damit das Verfahren nicht stecken bleibt, wenn es keine Übereinstimmung gibt.

Der erste Token wird also (0, 0, A). Beachte, dass die Klammern, Kommas und Abstände nur darstellerischen Charakter haben. Die beiden Nullen bedeuten, dass es keine Übereinstimmung gab. Der Token enthält also lediglich das erste Symbol A. Damit ist der erste Schritt abgeschlossen. Der Token wird an den Empfänger geschickt.

Nun wird der Zeichenstring derart weiter nach links geschoben, dass die bereits verarbeiteten Zeichen im Such-Buffer zu liegen kommen. Das ist im Moment nur das Symbol A, siehe die Zeile 3 in Abbildung 6. Es wird erneut geprüft, ob es zwischen Such-Buffer und der linken Seite des Vorschau-Buffers eine Übereinstimmung gibt. Das ist aber wiederum nicht der Fall, das Zeichen M kommt im Such-Buffer noch nicht vor. Es wird also der Token (0, 0, M) erzeugt, der lediglich das Symbol M enthält. Der Token wird an den Empfänger geschickt, während das Zeichen M vom Vorschau- in den Such-Buffer geschoben wird.

Wir befinden uns anschliessend auf Zeile 4 der Abbildung. Es wird wieder geprüft, ob es im Such-Buffer eine Übereinstimmung mit dem Inhalt des Vorschau-Buffers gibt. Und tatsächlich, jetzt findet sich eine Übereinstimmung: Ab der zweiten Stelle des Such-Buffers (Offset 2) stimmen zwei Zeichen mit dem linken Ende des Vorschau-Buffers überein. Wir beschreiben also diese Übereinstimmung in den Token und geben ein weiteres Zeichen mit: (2, 2, M). Dieser Token umfasst also die drei Zeichen A M M, die jetzt in den Such-Buffer übertragen werden.

Auf Zeile 5 sind gleich drei Übereinstimmungen auffindbar, nämlich bei den Offsets 1, 2 und 4. Wir wählen selbstverständlich die längste Übereinstimmung beim Offset 4. Hier finden wir eine Übereinstimmungslänge von 2 Zeichen (M A). Der betreffende Token wird (4, 2, A). So gelangen wir zur

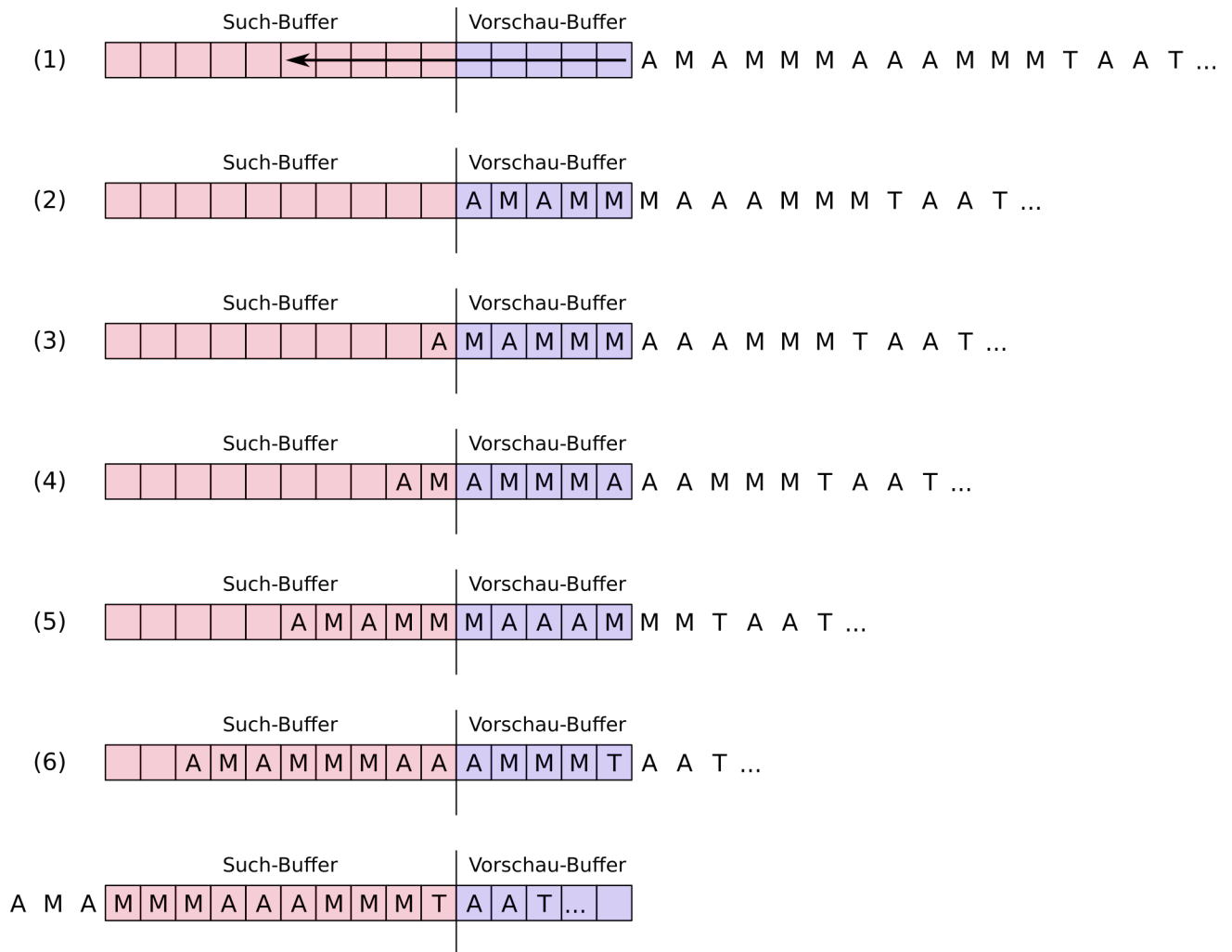


Abbildung 6: Beispiel eines LZ77 Encoder-Laufs.

Zeile 6. Wieder gibt es verschiedene Übereinstimmungen. Die beste findet sich bei Offset 6. Sie hat eine Länge von 4 Zeichen. Damit ergibt sich der Token (6, 4, T), der gerade den gesamten Vorschau-Buffer umfasst. Beachte, dass eine längere Übereinstimmung als 4 Zeichen im vorliegenden Fall nicht zulässig ist, da wir sonst kein zusätzliches Zeichen angeben können.

Werden nun die Zeichen weiter nach links geschoben, so fallen die ersten drei Zeichen aus dem Such-Buffer raus und stehen dem Verfahren nicht mehr zur Verfügung. Zusammenfassend ergeben sich die folgenden Tokens für den bisher verarbeiteten String von 13 Zeichen:

(0, 0, A) (0, 0, M) (2, 2, M) (4, 2, A) (6, 4, T)

Jetzt stellt sich natürlich die Frage, ob wir schon in der Lage waren, mit diesen Token den Zeichenstring zu komprimieren. Dazu müssen wir uns zuerst Rechenschaft über die Zeichen- und Tokengrößen ablegen. Nehmen wir an, es handle sich um Zeichen, die je 1 Byte belegen<sup>15</sup>. Die ursprünglichen 13 Zeichen belegen demnach total  $13 \cdot 8 = 104$  Bit. Im Token kommt ebenfalls ein derartiges Zeichen vor. Zusätzlich muss der Offset gross genug sein, damit der jede Position (1 bis 10) im Such-Buffer bezeichnen kann. Hierfür sind 4 Bit nötig. Die Längenangabe muss den gesamten Vorschau-Buffer abdecken können, ohne das letzte Zeichen, das bestenfalls die Funktion eines zusätzlichen Zeichens haben kann. Man muss also bis 4 zählen können, wozu 3 Bit nötig sind. Ein Token besteht demnach aus total  $4 + 3 + 8 = 15$  Bit. Die fünf Token umfassen gesamthaft  $5 \cdot 15 = 75$  Bit. Da dies weniger als die 104 Bit des Originalstrings sind, haben wir eine Kompression erzielt. Die Kompressionsrate  $R$  ist:

$$R = \frac{75}{104} = 0.72$$

Wir wollen noch erwähnen, dass die Buffergrößen im vorliegenden Beispiel nicht sehr günstig gewählt wurden. Mit den 3 Bit des Längenzählers könnte man bis 7 zählen. Folglich könnte der Vorschau-Buffer maximal 8 Zeichen lang sein, was eine längere maximale Übereinstimmung ermöglichen würde, ohne dass deswegen der Token grösser wäre. Zudem liesse sich mit den 4 Bit des Offset-Zählers ein Such-Buffer bis zu einer Länge von 15 Zeichen abdecken, was wiederum bei gleicher Tokengrösse eine potenziell bessere Kompression ergäbe. Schliesslich sei noch ergänzt, dass in der Realität sowohl Such-, wie Vorschau-Buffer deutlich grösser sind. Typische Grössen wären zum Beispiel 4095 Zeichen im Such-Buffer und 32 oder 64 Zeichen im Vorschau-Buffer.

▼ Die Tokengrösse in einem LZ77-Verfahren mit 4095 Zeichen an je 8 Bit im Such-Buffer und 64 Zeichen im Vorschau-Buffer ist:

$$\lceil \log_2 64 \rceil + \lceil \log_2 4095 \rceil + 8 = 26 \text{ Bit}$$

Dabei bezeichnet der Operator  $\lceil \cdot \rceil$  das Aufrunden auf die nächste ganze Zahl. ▲

---

<sup>15</sup> Das wäre beispielsweise bei Unicode mit UTF8 so, oder bei den ASCII-basierten ISO 8859 Zeichensätzen.

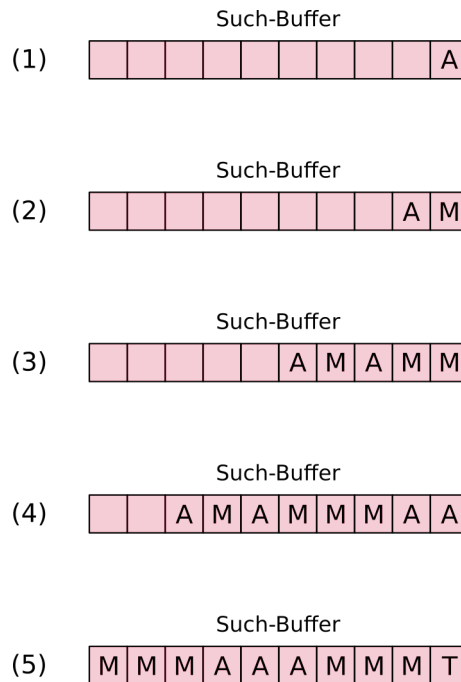


Abbildung 7: Beispiel eines LZ77 Decoder-Laufs.

▼ Als Beispiel sei noch der Decoder betrachtet. Er empfängt der Reihe nach, von links nach rechts, die folgenden Token:

(0, 0, A) (0, 0, M) (2, 2, M) (4, 2, A) (6, 4, T)

Der Decoder benötigt nur den Such-Buffer (siehe Abbildung 7), in welchen er den String decodiert und gleichzeitig ausgibt. Am Anfang ist der Such-Buffer leer. Wenn der erste Token (0, 0, A) ankommt, so wird er interpretiert: Der Token enthält keine Referenz in den Such-Buffer. Er enthält lediglich das Zeichen A. Es wird also das Zeichen A rechts in den Such-Buffer geschoben (1. Zeile in Abbildung. 7), genauso wie es der Encoder gemacht hatte, und gleichzeitig wird das A am Ausgang ausgegeben.

Mit dem zweiten Token geschieht genau dasselbe. Da er keine Referenz in den Such-Buffer enthält, wird bloss der Buchstabe M ausgegeben und in den Such-Buffer gefüttert (Zeile 2). Der dritte Token referenziert nun auf den Such-Buffer, nämlich auf Position 2. Die Übereinstimmungslänge ist zwei Zeichen, und entspricht gerade dem gesamten belegten Bereich im Such-Buffer. Es werden also die beiden Zeichen A und M aus dem Such-Buffer kopiert und zusammen mit dem zusätzlichen Zeichen M in den Such-Buffer geschoben und parallel dazu ausgegeben. Im Such-Buffer steht nun A M A M M (Zeile 3 in der Abbildung) .

Im nächsten Token steht wieder eine Referenz in den Such-Buffer, diesmal auf Position 4, mit einer Länge von zwei Zeichen, nämlich M A. Diese beiden Zeichen mit dem zusätzlichen A werden in den



Such-Buffer geschoben (Zeile 4). Schliesslich dekodieren wir den 4. Token mit einer Referenz auf die Zeichen A M M M. Werden diese Zeichen zusammen mit dem zusätzlichen Zeichen T in den Such-Buffer gefüllt, so fallen auf der linken Seite des Buffers die ersten drei Zeichen (A M A) raus und stehen dem Verfahren nicht mehr zur Verfügung.

Der gesamte bisher decodierte Output ist folglich A M A M M M A A A M M M T . Er stimmt - wie erhofft - mit dem Anfang des ursprünglichen Texts überein. ▲

## 6.2 LZ78

Zur Illustration des LZ78-Verfahrens sei dieselbe Nachricht wie oben verwendet:

A M A M M M A A A M M M T A A T ...

Es wird nun aber nicht ein Sliding Window verwendet, sondern das Verfahren baut im Verlauf des Prozesses ein eigenes Wörterbuch auf. Dabei werden Token gebildet, die beschreiben, wie der jeweils letzte Eintrag im Wörterbuch gebildet wurde. Dies ist wichtig, denn der Empfänger wird aus den Token sein eigenes Wörterbuch aufbauen, das natürlich identisch mit jenem des Senders sein muss. Beide Wörterbücher, jenes beim Sender und jenes beim Empfänger, sind am Anfang leer und sehen so aus:

Index	String
0	-

Der Index nummeriert die Einträge des Wörterbuchs. Der String ist der eigentlichen Eintrag. Im leeren Wörterbuch gibt es nur einen Eintrag mit Index 0, der einen leeren String (-) enthält.

Das Verfahren funktioniert so, dass die Nachricht immer von links nach rechts gelesen wird, bis der gelesene Teilstring im Wörterbuch nicht mehr vorhanden ist. In dem Moment, wo der Teilstring nicht mehr im Wörterbuch vorkommt, besteht er aus einem Teil, der noch im Wörterbuch vorkommt plus einem zusätzlichen Zeichen. Der Token, der nun gebildet wird, beschreibt genau dies: Er beschreibt, welcher bestehende Eintrag im Wörterbuch mit welchem zusätzlichen Zeichen erweitert wird, um den neuen Eintrag zu bilden. Der Token besteht also aus zwei Elementen:

Token: (Index, Zeichen)

Der *Index* referenziert den bestehenden Eintrag im Wörterbuch, das *Zeichen* gibt an, in welcher Weise der bestehende Eintrag erweitert wird. Beachte wieder, dass Klammern, Komma und Abstand nur der Darstellung dienen, aber nicht wirklich Teil des Tokens sind.

Beginnt man also die Nachricht von links her zu lesen, so stoppt man bereits beim ersten Buchstaben A, denn dieser ist im Wörterbuch noch nicht verzeichnet. Man bildet also den ersten Token (0, A). Der neue Eintrag beruht also auf einem leeren String, der mit einem A ergänzt wird. Dann lesen wir

weiter und finden den Buchstaben M, der im Wörterbuch ebenfalls noch nicht vorkommt. Analog zu vorher wird der zweite Token (0, M) gebildet. Dann lesen wir beim dritten Buchstaben weiter und finden A M. Das A alleine ist nun im Wörterbuch schon enthalten, die Buchstabengruppe A M aber nicht. Damit bilden wir nun einen dritten Token (1, M). Er besagt, dass der Wörterbucheintrag 1 (String A) mit einem M ergänzt wird. Lesen wir abermals beim nächsten Zeichen weiter, so finden wir zuerst ein einfaches M, das im Wörterbuch schon vorkommt, dann ein Doppel-M, das noch nicht vorkommt. Wir bilden also einen neuen Wörterbucheintrag M M. Setzen wir den Prozess bis ans Ende der vorliegenden Nachricht fort, so erhalten wir das folgende Wörterbuch und die entsprechenden Token, die vom Encoder verschickt werden.

Index	String	Token
0	-	
1	A	(0, A)
2	M	(0, M)
3	AM	(1, M)
4	MM	(2, M)
5	AA	(1, A)
6	AMM	(3, M)
7	MT	(2, T)
8	AAT	(5, T)
9	...	...

Man erkennt unschwer, dass die Strings in den Einträgen je länger desto breiter werden. Das heisst, je länger das Verfahren läuft, umso mehr Zeichen werden in einem Token codiert. Bei sehr langen Nachrichten ergibt sich nach einer Weile eine optimale Kompression. Im vorliegenden Fall besteht der ursprüngliche Text aus 16 Zeichen an je 8 Bit, total also  $16 \cdot 8 = 128$  Bit. Der entsprechende LZ78-Output besteht aus 8 Token. Nehmen wir an, das Wörterbuch sei für 31 Einträge ausgelegt, so benötigt ein Token 5 Bit für den Index plus 8 Bit für das zusätzliche Zeichen. Ein Token wäre dann 13 Bit gross, alle 8 Token zusammen wären  $13 \cdot 8 = 104$  Bit. Damit folgt die Kompressionsrate  $R$  zu:

$$R = \frac{104}{128} = 0.81$$

Beachte allerdings, dass das Verfahren in der Praxis natürlich für grössere Nachrichten ausgelegt ist und darum auch das Wörterbuch entsprechend gross geplant wird. Ein übliches Wörterbuch ist darum zum Beispiel für 1 Mio. Einträge vorgesehen. Der entsprechende Index im Token beansprucht dann 20 Bit.

Es stellt sich in diesem Zusammenhang zudem die Frage, was man tut, wenn das Wörterbuch dennoch voll wird. Die Antwort ist simpel: Normalerweise wird man das Wörterbuch einfach löschen und mit einem leeren Wörterbuch weiterfahren.

▼ Der LZ78-Decoder sieht nur die Token:

(0, A) (0, M) (1, M) (2, M) (1, A) (3, M) (2, T) (5, T) ...

Daraus wird er seine eigene Arbeitskopie des Wörterbuches aufbauen und gleichzeitig den Output, also die decodierte Nachricht bilden. Zuerst empfängt er den Token (0, A). Dieser Token besagt, dass ein neuer Wörbucheintrag gebildet wird, dessen String sich zusammensetzt aus dem Leerstring (Index 0) und dem Zeichen A. An der Wörterbuch-Position 1 steht also der String A, der gleichzeitig der Output ist.

Jetzt folgt der Token (0, M), der in analoger Weise den zweiten Wörterbuch-Eintrag M bildet. Der dritte Token (1, M) verbindet den Eintrag mit Index 1 mit dem Zeichen M. Daraus entsteht der neue Eintrag A M, der mit Index 3 im Wörterbuch verzeichnet wird. Das Verfahren setzt sich so fort, bis das Empfänger-Wörterbuch die Gestalt der vorhergehenden Tabelle annimmt. Gleichzeitig hat der Empfänger die Ausgangsmeldung A M A M M M A A A M M M T A A T ... rekonstruiert. ▲

### 6.3 LZW

Das LZW-Verfahren<sup>16</sup> ist eine Weiterentwicklung von LZ78 mit dem Ziel, den Token zu verkleinern. Der LZW-Token besteht in der Folge nur noch aus einem Wörterbuch-Index. Das Verfahren ist etwas umständlicher als LZ78, aber in der Regel leicht effizienter. Zur Illustration verwenden wir nochmals dieselbe Nachricht.

A M A M M M A A A M M M T A A T ...

Das Wörterbuch von LZW ist vorinitialisiert mit allen Zeichencodes von 0 bis 255 (einschliesslich den ASCII-Zeichen im Bereich 0 bis 127). Das sieht dann so aus:

Index	String
...	...
65	A
...	...
77	M
...	...
84	T
...	...
255	?

---

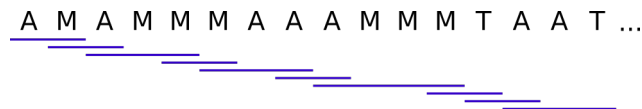
<sup>16</sup> Der Buchstabe W bezieht sich auf Terry A. Welch (1939 bis 1988), einen US-Informatiker. Er entwickelte und patentierte das LZW-Verfahren 1983, ausgehend von den Arbeiten von Lempel und Ziv.

Es wird nun wie bei LZ78 die Nachricht von links nach rechts gelesen, bis der gelesene Teilstring im Wörterbuch nicht mehr vorkommt. In dem Moment, wo der Teilstring nicht mehr im Wörterbuch enthalten ist, besteht er aus einem Teil, der noch im Wörterbuch vorkommt und einem zusätzlichen Zeichen. Im Gegensatz zum LZ78 enthält der Token nun aber nur den Index des schon bestehenden Eintrags im Wörterbuch, nicht aber das neue Zeichen:

Token: (Index)

In der folgenden Darstellung sind die ersten gelesenen Zeichen A M mit einer Linie unterstrichen. Für diesen Abschnitt A M wird nun ein Wörterbucheintrag mit Index 256 erstellt. Es wird aber nur der Token (65) verschickt, der den Wörterbucheintrag ohne das letzte Zeichen beschreibt.

A M A M M M A A A M M M T A A T ...



Jetzt wird aber nicht beim Folgezeichen weitergelesen, sondern es wird mit jenem letzten, bereits gelesenen Zeichen fortgefahren, das vom Token noch nicht wiedergegeben wird. Im vorliegenden Fall ist es das M. Man liest also M A und, da dieses Doppel-Symbol im Wörterbuch noch nicht verzeichnet ist, bildet man einen neuen Eintrag mit Index 257. Verschickt wird der Token (77), der nur das M beschreibt aber noch nicht das A. Man liest also mit dem A weiter, findet, dass A M im Wörterbuch schon vorkommt, und liest daher weiter bis zu A M M. Der Wörterbucheintrag 258 wird demnach A M M. Verschickt wird der Token (256). Fahren wir entsprechend fort, so finden wir das Wörterbuch und die Token gemäss der folgenden Tabelle.

Index	String	Token	Index	String	Token
...	...		258	AMM	(256)
65	A		259	MM	(77)
...	...		260	MAA	(257)
77	M		261	AA	(65)
...	...		262	AMMM	(258)
84	T		263	MT	(77)
...	...		264	TA	(84)
255	?		265	AAT	(261)
256	AM	(65)	...	...	...
257	MA	(77)			

Es fällt auf, dass das LZW-Verfahren etwas mehr Token erzeugt, als die LZ78-Methode, aber die Token sind - wie beabsichtigt - kürzer als bei der LZ78-Methode. Nehmen wir wiederum an, dass das Wörterbuch klein sei, also zum Beispiel maximal 511 Einträge aufnehmen kann (inkl. den schon initialisierten Einzelsymbolen), so ist ein Token 9 Bit gross. Die 10 Token benötigen demnach  $10 \cdot 9 =$

90 Bit. Der entsprechende Originaltext ist 16 Zeichen lang, das letzte Zeichen wurde aber noch nicht übertragen. Damit erhalten wir den Kompressionsfaktor  $R$ :

$$R = \frac{9 \cdot 10}{15 \cdot 8} = 0.75$$

Im vorliegenden Fall komprimiert das LZW-Verfahren also tatsächlich etwas besser als der LZ78-Kompressor. Es sei aber auch an dieser Stelle nochmals darauf hingewiesen, dass in praktischen Anwendungen typischerweise ein viel grösseres Wörterbuch verwendet wird, zum Beispiel eines, das eine Mio. Einträge aufnehmen kann.

▼ Es sei im Folgenden gezeigt, wie der Decoder arbeitet. Verwendet werden die Token von oben. Der Decoder startet mit einem initialisierten Wörterbuch, das die Einträge mit den Indizes von null bis 255 enthält. Jetzt erhält der Decoder den ersten Token (65). Damit bildet er einen Wörterbucheintrag. Er besteht aus einem Doppel-Symbol, wobei aber der zweite Buchstabe noch nicht bekannt ist. In der Tabelle unten ist das fehlende Zeichen mit dem Symbol  $\square$  dargestellt. Mehr kann der Empfänger vorläufig nicht tun. Der Output des Decoders ist das Zeichen A, entsprechend dem Token.

Token	Index	String	Output
	...	...	
	65	A	
	...	...	
	77	M	
	...	...	
	84	T	
	...	...	
	255	?	
(65)	256	A $\square$	A

Beim Empfang des zweiten Tokens (77) erzeugt der Empfänger wieder einen Wörterbucheintrag, nämlich wie folgt (der Übersichtlichkeit halber sind nur noch die neuen Wörterbucheinträge dargestellt):

Token	Index	String	Output
(65)	256	A $\square$	A
(77)	257	M $\square$	M

Wiederum fehlt noch das letzte Zeichen des neuen Eintrags. Nun aber ist das erste Zeichen M des neuen Eintrags das fehlende Zeichen des voraus gegangenen Wörterbucheintrags mit dem Index 256. Damit folgt:

Token	Index	String	Output
(65)	256	AM	A
(77)	257	M□	M

Man sieht also, dass jeder Wörtebucheintrag immer erst beim Eintreffen des nächsten Tokens fertiggestellt werden kann. Das so entstehende Wörterbuch ist natürlich wieder identisch mit dem Wörterbuch des Senders. Tritt der nächste Token (265) ein, so folgt:

Token	Index	String	Output
(65)	256	AM	A
(77)	257	MA	M
(256)	258	AM□	AM

Wir wollen das Ganze nochmals zusammenfassen: Auf der Empfängerseite ergibt jeder Token einen neuen aber noch unvollständigen Wörtebucheintrag. Es fehlt jeweils das letzte Zeichen des Eintrags, das wir mit □ markieren. Erst wenn der folgende Token eintrifft, kann das fehlende Zeichen eingesetzt werden. Wir ersetzen dann □ mit dem ersten Zeichen jenes Wörtebucheintrags, auf den sich der betreffende Folgetoken bezieht. ▲

## 7 Arithmetische Codes

(Werden nicht behandelt)

## 8 JPEG

(ToDo)

Krieg ist 90 % Information.

*Angeblich von Napoleon Bonaparte*