Codierungstheorie

Semesterarbeit Teil II

Fernfachhochschule Schweiz

Semesterarbeit

im Studiengang BSc Informatik

von

Arber Osmani

Bern, 13. November 2021

Eingereicht bei: Prof. Dr. Jörg Osterrieder

Inhaltsverzeichnis

1 Einleitung

In manchen Bereichen der elektronischen Datenübertragung genügt es nicht eine fehlerhafte Übertragung bloss zu erkennen, sondern es bedarf der Möglichkeit die genaue Lokation des Fehlers zu kennen und zu korrigieren. Unter fehlerhafter Übertragung verstehen wir, dass die gesendete Folge von Bits (0 und 1) beim Empfänger fehlerhaft ankommt. Beispielsweise kann es passieren, dass einzelne Bits umkippen können (aus einer 1 wird eine 0 und umgekehrt).

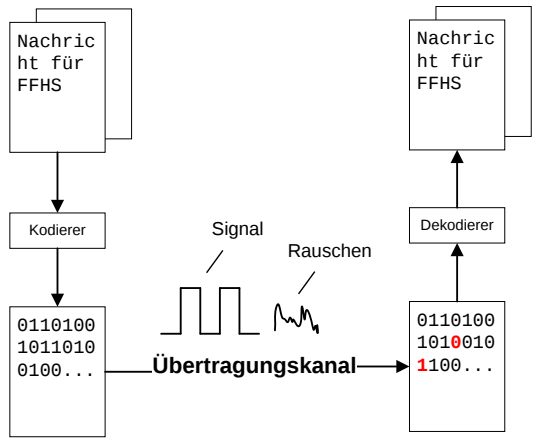

Abbildung 1.1: Übertragung einer Nachricht

Bei einem Telefongespräch ist es beispielsweise dem Empfänger möglich, den Sender um das Wiederholen der Nachricht zu bitten. Bei beschädigten Datenstücken auf einer CD, sieht das anders aus. Der Empfänger soll in der Lage sein, die Fehler zu erkennen und korrigieren können.

Die Lösung des Problems beruht intuitiver Weise auf *Wiederholung* der Nachricht – also die Nachricht irgendwie redundant zu versenden, oder um auf das Beispiel mit der CD zurückzukommen, redundant auf der CD zu schreiben [1].

Fehlerkorrigierende Codes basieren auf dem Konzept des Paritätbits. In den theoretischen Grundlagen besprechen wir daher zunächst im Abschnitt 2.1 die Bedeutung des Paritätsbits. Anschliessend erarbeiten wir im Abschnitt 2.2 die Grundlagen für den Hamming-Code. Um die Fehlerkorrektur möglichst durchführen zu können, besprechen wir im Abschnitt 2.3 was es mit den Prüfbitstellen auf sich hat.

Im Kapitel 3 gehen wir auf den Python-Code ein und besprechen die gemäss Anforderungen erstellen, Funktionen `get_generator_matrix`, `get_check_matrix`, `encode`, `decode` und `check`, der Klasse `HammingCode`.

2 Theoretische Grundlagen

2.1 Paritätsbit

Die nachfolgenden fehlerkorrigierenden Codes die wir anschauen werden, beruhen auf dem Prinzip des Paritätbits. Um die Bedeutung dieses Bit zu verstehen, schauen wir ein Beispiel aus dem Zeichensatz ASCII[1] an.

Der Computer arbeitet mit Binärziffern, das entspricht dem Zahlenraum $\mathbb{Z}_2$. Der ASCII Zeichensatz ist eine 7-Bit-Zeichenkodierung die alle druckbaren und nicht druckbaren Zeichen für die englische Sprache umfasst. Das entspricht also $2^7 = 128$ Zeichen. Da die meisten Rechner mit 8-Bit Paketen arbeiten, wurde in der Anfangszeit von ASCII gerne das achte Bit als Paritätsbit (Prüfbit) verwendet.

Das Schriftzeichen `A` im ASCII, hat die Nummerierung 65_{10}. Im $\mathbb{Z}_2^7$ entspricht das der Bitfolge $m = (1\ 0\ 0\ 0\ 0\ 0\ 1)^T$. Durch hinzufügen eines Paritätsbit erhalten wir $m' = (1\,0\,0\,0\,0\,0\,1\ 0)^T$.

Das Paritätsbit ist so gestaltet, dass die Anzahl Einsen gerade ist (einschliesslich des Paritätsbit selbst). Im Beispiel ist die Anzahl Einsen gerade, daher ist das Paritätsbit 0. Wenn auf der Empfängerseite die Anzahl Einsen nicht gerade ist, kann der Empfänger davon ausgehen, dass bei der Übertragung etwas schiefgegangen ist.

Für die Berechnung stellen wir uns das mit einer Matrix vor. $m \in \mathbb{Z}_2^7$ entspricht der zu übertragenen Nutzlast, von der wir eigentlich alle sieben Bits direkt übernehmen möchten und das achte Bit berechnen wollen. Dafür multiplizieren wir m mit der Matrix A. Die Matrix A ist so konstruiert, dass die ersten sieben Zeilen einer Einheitsmatrix entsprechen und die letzte Zeile nur aus Einsen besteht. Das hat genau den gewünschten Effekt, den wir mit dem Paritätsbit erhalten wollen.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Die Multiplikation $A \cdot m^T$ entspricht dann dem gewünschten Ergebnis m'.

Der Empfänger kann die empfangene Nachricht m' mit $(1\ 1\ 1\ 1\ 1\ 1\ 1\ 1)^T$ multiplizieren und erwartet das Ergebnis 0, andernfalls kann von einem Übertragungsfehler ausgegangen werden.

Wie einfach zu erkennen ist, ist diese Methode sehr limitiert. Fehler können nur erkannt werden, wenn höchstens ein Bit oder besser gesagt, eine ungerade Anzahl Bits, gestört wird. Darüberhinaus kann nicht gesagt werden, welches Bit gestört wurde. Es hat also keinen fehlerkorrigierenden Charakter.

[1] `https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange`

2.2 Hamming-Code

Der Hamming-Code wurde im Jahre 1950 von Richard Hamming[2] publiziert. Der Code basiert auch auf Paritätbits und ist ein fehlerkorrigierender Code. Ebenfalls basiert der Hamming-Code, wie auch die meisten anderen fehlerkorrigierenden Codes, auf den Prinzipien der linearen Codes. Wodurch alle Vorzüge der linearen Algebra zum Zuge kommen.

Es wird darauf aufgebaut, dass ein Vektor als ein Untervektorraum eines Vektorraumes mit einer sogenannten Generatormatrix multipliziert wird. Dadurch kann man ein *Wort* in eine höhere Dimension befördern und sogleich eine Redundanz schaffen. Denn durch ein dekodieren, also der Multiplikation mit einer Dekodierungsmatrix (Kontrollmatrix), kann auf einfache Weise die Integrität festgestellt werden und zugleich auch korrigiert werden.

Am besten schauen wir uns dazu ein Beispiel an. Wir wollen vier Datenbits $d_1, \ldots, d_4$ übertragen. Dazu kommen drei weitere Prüfbits $p_1, \ldots, p_3$ hinzu.

Abbildung 2.1 zeigt eine gute graphische Darstellung der Idee vom Hamming-Code. Die einzelnen Prüfbits p_n überwachen jeweils eine unterschiedliche Menge von Datenbits d_n. Wenn nun beispielsweise das Datenbit d_2 fehlerhaft ist, werden die Prüfbits $p_2 \in B$ und $p_3 \in C$ nicht mit der Erwartung übereinstimmen, während $p_1 \in A$ schon. Die Menge $B \cap C \setminus A = \{d_2\}$ enthält somit das fehlerhafte Bit [2].

Ähnlich wie bei der Kreuzparität, möchten wir das Datenwort $m = (d_1 \ d_2 \ d_3 \ d_4)^T$ mit einer sogenannten Generatormatrix G multiplizieren, so dass die 3 Prüfbits im resultierenden Vektor

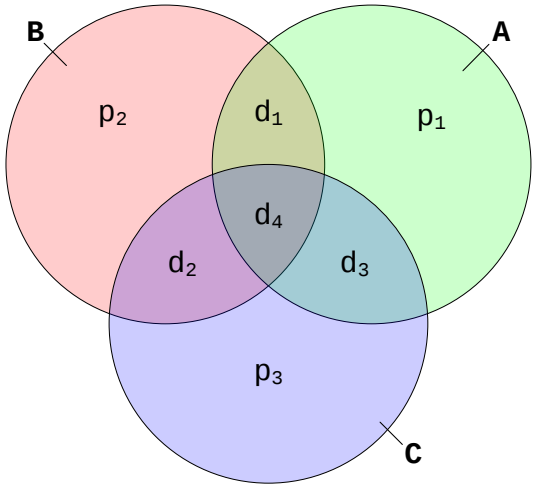

Abbildung 2.1: (7,4)-Hamming-Code

m' *integriert* sind. Auf der Empfängerseite soll es dann möglich sein mithilfe der Kontrollmatrix H, m' auf Fehler zu überprüfen und ggf. zu korrigieren.

$$m' = G \cdot m = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix}$$

Die Generatormatrix ist so gestaltet, dass die ersten vier Zeilen wieder einer Einheitsmatrix entsprechen. Die Datenbits werden also 1:1 in m' übernommen. Die letzten 3 Zeilen entsprechen den Prüfbits und sollen jeweils für eine bestimmte Teilmenge der Datenbits, das Paritätbit berechnen. Die Farben korrespondieren mit denen der Abbildung 2.1, so dass erkenntlich ist, welches Prüfbit für welche Teilmenge zuständig ist.

Auf der Empfängerseite könnte man nun einfach

[2] https://de.wikipedia.org/wiki/Richard_Hamming

die ersten vier Elemente (Datenbits) entnehmen und damit arbeiten. Zunächst aber soll mithilfe der drei letzten Elemente von m', den Prüfbits, auf Fehler überprüft werden. Dafür rechnen wir $H \cdot m'$.

$$H \cdot m' = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix}$$

Im Falle das keine Fehler vorhanden sind, ist das Resultat dieser Multiplikation $c_0 = (0\ 0\ 0)^T$. Ziehen wir unser Beispiel ein wenig weiter und nehmen an, das Resultat wäre

$$c_1 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

und fragen uns wie das zu interpretieren ist. Bei genauerer Betrachtung der Kontrollmatrix H, sehen wir dass alle binären Kombination von 2^3, abzüglich der 0 (die den fehlerfreien Fall repräsentiert), vorhanden sind. Das sind also $2^3 - 1 = 7$ Spalten, die wir von links nach rechts als $d_1, \ldots, d_4, p_1, \ldots, p3$ bezeichnen können.

Unser Ergebnis c_1 entspricht genau der zweiten Spalte, womit wir sagen können, dass das Datenbit d_2 falsch sein muss.

In unserem Beispiel haben wir ein $(7, 4)$-Hamming-Code verwendet. Diese Notation gibt an, wieviele Nachrichtenbits N und wieviele Datenbits n gesendet wurden – (N, n)-Code. Die Anzahl der Datenbits n, also der eigentlichen Nutzlast, berechnet sich aus $N - k$. Für die Berechnung von n überlegen wir uns, wie viele mögliche *Fehlerkombinationen* wir mit k Paritätsbits darstellen können; die Antwort ist ziemlich einfach, es sind 2^k Möglichkeiten, wobei jedoch eine Kombination dafür steht, dass es keinen Fehler gab – also $2^k - 1$. Das gibt uns aber N, also müssen wir noch die Paritätsbits k abziehen.

$$n = 2^k - k - 1$$

Mit nachfolgender Tabelle soll das Verhältnis zwischen Daten- und Paritätsbits bei zunehmenden n gezeigt werden. Es sei noch gesagt, dass die Anzahl Datenbits n nur bestimmte Längen haben kann, die sich aus der obigen Formel ergeben – man spricht auch von einem *Blockcode*.

n	k	Verhältnis k/N	Notation
1	2	0.66	$(3, 1)$
4	3	0.43	$(7, 4)$
11	4	0.26	$(15, 11)$
26	5	0.16	$(31, 26)$
57	6	0.10	$(63, 57)$
120	7	0.05	$(127, 120)$

Das Verhältnis wird also zunehmend besser bzw. der Overhead ist bei grosser Anzahl Datenbits unwesentlich. Sollten wir also alle zu versendeten Datenbits mit der minimalen Anzahl Paritätsbits ausstatten, um so das Verhältnis k/N so optimal wie möglich zu halten? Nein, denn es gilt weiterhin, dass maximal nur ein Bit fehlerhaft vorhanden sein darf, damit noch eine Fehlerkorrektur möglich ist. Die Entscheidung, welcher Hamming-Code gewählt werden soll, hängt von der Frage ab, mit wievielen Fehlern in der realen Anwendung zu rechnen ist. In der Praxis ist der Hamming-Code $(63, 57)$ nicht untypisch [3].

[3] `https://de.wikipedia.org/wiki/Hamming-Code`

2.3 Prüfbitstellen

Unser Beispiel hatte noch eine Unschönheit als es darum ging, die Position des fehlerhaften Datenbits d_2 aus dem Ergebnis c_1 zu ermitteln. Wir hatten einfach gesagt, dass der Vektor c_1 der zweiten Spalte der Kontrollmatrix H entspricht und wir aufgrund der Reihenfolge auf d_2 rückschliessen können. Es wäre aber viel elleganter, wenn man durch geschickte Anordnung der Prüfbits, die Position aus dem Ergebnis direkt auslesen könnte – das ist tatsächlich auch machbar.

Dafür halten wir uns nochmal den $(7, 4)$-Hamming-Code vor Augen wo wir insgesamt sieben Bit übertragen haben. Wenn wir die einzelnen Bitpositionen mit binärer Darstellung beschriften (von 1 bis 7), erhalten wir 001, 010, 011, 100, 101, 110, 111. Die Positionen die einer Zweierpotenz entsprechen (2^n) haben zwangsläufig ein einziges Einserbit. An diesen Positionen kommen nun unsere Paritätbits (p_n) und an allen anderen Positionen (mit mehr als einem Einserbit), schreiben wir die Datenbits (d_n).

n-tes Parirätbit	Position	Position in Binär
1	$2^0 = 1$	00000 1
2	$2^1 = 2$	0000 1 0
3	$2^2 = 4$	000 1 00
4	$2^3 = 8$	00 1 000
5	$2^4 = 16$	0 1 0000
6	$2^5 = 32$	1 00000
n	$2^n = N$	a_{n+1} $a_n \dots a_0$

Die einzelnen Paritätbits *kümmern* sich ja immer um eine bestimmte Menge an Datenbits. Jetzt ist die Frage, wie berechnet wird, um welche Datenbits sich p_1, p_2 usw. sich kümmern? Das Paritätbit an der Position 2^n kümmert sich um alle Datenbits, deren Position in binärer Schreibweise, an der Stelle $n + 1$ (Spalte der Bitfolge) eine Eins haben bzw. dort wo $p_n \wedge p_n$ gilt.

Die nachfolgende Tabelle veranschaulicht die Anordnung nochmal.

Bit Position		1	2	3	4	5	6	7
Codiertes Bit		p_1	p_2	d_1	p_3	d_2	d_3	d_4
Zuständigkeit des Paritätbits	p_1	•		•		•		•
	p_2		•	•			•	•
	p_3				•	•	•	•

Die Kontrollmatrix H ist bei dieser Anordnung der Prüfbits, ganz einfach anzugeben.

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Die farblich markierten Positionen entsprechen den Prüfbits, während die Positionen 3, 5, 6 und 7, den Datenbits entsprechen. Für welche Datenbits d_n beispielsweise das erste Prüfbit p_1 zuständig ist, entspricht wieder der Menge $p_1 \wedge d_n$. Für p_1 ist das (1 1 0 1). In der Generatormatrix entspricht diese Zeile dann der Position 2^0.

$$G = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In der berechneten Nachricht m' sind entsprechen dann nicht mehr die ersten vier Bits den Datenbits wie im Abschnitt 2.2 sondern nicht ebenfalls entsprechend verteilt.

Nach diesen Schema und den erarbeiteten theoretischen Grundlagen, sind wir nun in der Lage, den Hamming-Code in Python zu implementieren.

3 Python-Code

3.1 Implementierungsidee

Nachfolgend einige Implementierungsdetails zu den Klassenmethoden der Klasse `HammingCode`.

`gen_generator_matrix` produziert eine Matrix nach dem im Abschnitt 2.3 beschriebenen Verfahren. Die zurückgegebene Matrix hat die Grösse entsprechend der Instanziierung der `HammingCode` Instanz.

```
hc = HammingCode(k)
G = hc.get_generator_matrix()
```

`G` ist in diesem Fall eine $(k^2 - k - 1 \times k)$ Matrix.

`get_check_matrix` erzeugt die passende Kontrollmatrix. Nach Implementation für das Setzen der Prüfbitstellen wie im Abschnitt 2.3 beschrieben, ist die Kontrollmatrix sehr einfach gestaltet.

`encode` und `decode` sind für das kodieren bzw. dekodieren von einem Wort zuständig. Ersteres verwendet die Matrix G, letzteres die Matrix H. Bei der Dekodierung werden Einbitfehler zu korrigieren versucht.

```
hc = HammingCode(k)
m == hc.decode(hc.encode(m))  # True
```

`check` prüft ob ein gegebenes Codewort korrekt ist oder nicht. Der Rückgabewert ist daher `True` oder `False`. Die Prüfung besteht darin, das gegebene Codewort mit der Kontrollmatrix zu multiplizieren und das Ergebnis gegen die Aussage „ist Nullvektor“überprüfen. Die Antwort darauf entspricht dem Rückgabewert.

Die Klasse ist im Python Modul `sema.teil2.HammingCode` definiert.

Über die Aufgabenstellung hinaus, sind noch einige Utility Funktionen entwickelt, um das versenden von kodierten Nachrichten über einen nicht rauschfreien Kanal zu simulieren. All diese Funktionen sind in der Funktion `sema.teil2.utils.transmit` verpackt.

3.2 Programmcode

Nachfolgend die Signaturen der Methoden und Funktionen.

3.2.1 `sema.teil2.HammingCode`

```
* get_generator_matrix()
    -> linalg.Matrix

* get_check_matrix() -> linalg.Matrix

* encode(word: ARRAY_LIKE[1])
    -> linalg.Vector

* decode(codeword: ARRAY_LIKE)
    -> linalg.Vector

* decode(codeword: ARRAY_LIKE) -> bool
```

[1] `ARRAY_LIKE = typing.Union[typing.List[float], np.ndarray, linalg.Vector]`

3.2.2 sema.teil2.utils

```
* encode(s: str) -> list[list[int]]

* decode(bytes: list[list[int]] -> str

* channel(list[list[int]],
          errorsPerByte: int=0,
          errorProbability: float=1)
    -> list[list[int]]

* transmit(message: str,
           errorsPerByte: int=0,
           errorProbability: float=1)
    -> str
```

3.3 Anwendung

3.3.1 sema.teil2.HammingCode

```
# HammingCode Instanz
hc = HammingCode(4)

# Zu versendende Nachricht ‘m‘
m = [1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1]

# Kodierte Nachricht ‘m‘ -> m_1
m_1 = hc.encode(m)

# Fehlerhafte Nachricht ‘m‘ -> m_1_e
m_1_e = [m_1[0] + 1 % 2, *[*m_1][1:]]

# Dekodierte Nachricht ‘m_1‘ -> m_d
m_d = hc.decode(m_1)

# Dekodierte Nachricht
# ‘m_1_err‘ -> m_d_1
m_d_1 = hc.decode(m_1_e)

# Vergleich von ‘m_d‘ und ‘m_d_1‘
m_d == m_d_1  # True
```

3.3.2 sema.teil2.utils.transmit

```
# Zu versendende Nachricht
msg = 'Fernfachhochschule Schweiz'

# Versenden der Nachricht mit einer
# Fehlerrate von 0 Fehler / Byte
transmit(msg, 0) == msg  # True

# Versenden der Nachricht mit einer
# Fehlerrate von 1 Fehler / Byte
# und einer Fehlerwahrscheinlichkeit
# von 1 (=100%)
transmit(msg, 1, 1) == msg  # True

# Versenden der Nachricht mit einer
# Fehlerrate von 2 Fehler / Byte
# und einer Fehlerwahrscheinlichkeit
# von 1 (=100%)
transmit(msg, 2, 1) == msg  # False

# Versenden der Nachricht mit einer
# Fehlerrate von 2 Fehler / Byte
# und einer Fehlerwahrscheinlichkeit
# von 0.4 (=40%)
# Resultat: FernfĠchhochchule Sfhwŧ9z
transmit(msg, 2, 0.4)
```

Diese Simulation zeigt, dass der Hamming Code perfekt funktioniert, wenn maximal ein Einbitfehler vorliegt. Die Blockgrösse der Wörter, muss also abhängig von der Fehlerhaftigkeit des vorliegenden Übertragungskanals, gewählt werden. Sind Burstfehler[2] zu erwarten, also blockweise Störungen mehrerer Bits, könnte es Vorteilhaft die zu übertragenen Bits so anzuordnen, das die zu einem Block gehörigen Bits, über unterschiedliche Blocks gelegt werden. Das macht das Kodieren und Dekodieren jedoch komplizierter – an der Arbeitsweise vom Hamming Code ändert sich dabei jedoch nichts.

[2] https://de.wikipedia.org/wiki/Burstfehler

4 Konklusion

Der Hamming-Code ist sehr einfach zu implementieren. Mit der richtigen Anordnung der Prüfbits in der Generatormatrix, ist eine fehlerhafte Position in Codewörter, sofort erkennbar. Dabei wird auf Redundanz in besonderer Art und Weise gesetzt. Durch geschicktes anbringen von Paritätsbits und Anwendung von Methoden aus der linearen Algebra, können auf erstaunliche Art und Weise, Übertragungsfehler erkannt und korrigiert werden. Welcher Hamming-Code gewählt wird und wie die zu übertragenen Bits angeordnet werden, ist abhängig von der erwarteten Fehlerrate des Übertragungskanal und der erwarteten Fehlerarten, zu bestimmten. Die Erhöhung der Coderate führt zwar verhältnismässig zu einer geringeren Redundanz, erhöht aber gleichzeitig das Risiko, Fehler zu übersehen [2].

Der Hamming-Abstand h vom Hamming-Code, beträgt 3. Es können also maximal $(h-1) = 2$ Bitfehler erkannt und maximal $\lfloor (h-1)/2 \rfloor = 1$ Bitfehler korrigiert werden. Der erweiterte Hamming-Code nutzt ein zusätzliches Paritätsbits für den gesamten Codeblock. Dadurch wird ein Hamming von 4 erreicht, wodurch 2 Einbitfehler erkannt werden können. Korrigiert können aber weiterhin nur Einbitfehler werden.

Hamming-Code wird typischerweise in ECC-Speicherchips eingesetzt [2]. Es existieren auch fortgeschrittenere Codes, wie der Reed-Solomon-Code[1], der auch besser für Burstfehler geeignet ist.

[1] `https://de.wikipedia.org/wiki/Reed-Solomon-Code`

Literaturverzeichnis

[1] R. Socher. *Mathematik für Informatiker: mit Anwendungen in der Computergrafik und Codierungstheorie*. Fachbuchverl. Leipzig im Carl-Hanser-Verlag, 2011. ISBN 9783446422544. URL `https://books.google.ch/books?id=7_5mygAACAAJ`.

[2] Edmund Weitz. *Konkrete Mathematik (nicht nur) für Informatiker*. Springer Fachmedien Wiesbaden, 2018. doi: 10.1007/978-3-658-21565-1. URL `https://doi.org/10.1007/978-3-658-21565-1`.