Fernfachhochschule Schweiz
Zürich | Basel | Bern | Brig
Mitglied der SUPSI

Fernfachhochschule Schweiz
BSc Informatik

Hamming-Codes

Semesterarbeit im Modul LinAlg, des Studiengangs BSc Informatik

von

Tobias Gasche

Dozent:
Prof. Dr. Matthias Dehmer

4554 Etziken, 20. November 2024

Zusammenfassung

In der Informatik werden Daten in Form von Bits repräsentiert. Ein Bit ist eine Einheit, die den Zustand 0 oder den Zustand 1 annehmen kann.

Diese Arbeit beschreibt die Idee der Codierung, also der Transformation in Einsen und Nullen.

Hauptbestandteil ist aber die Theorie zu Fehlererkennung und Fehlerkorrektur anhand der sogenannten Hamming-Codes. Diese können feststellen, ob in einer Datenübertragung Fehler stattgefunden haben und können diese unter gewissen Umständen sogar selbständig korrigieren.

Im Anschluss an den Theorieteil wird in dieser Arbeit eine Python-Applikation vorgestellt, die Generator- und Prüfmatrizen für Hamming-Codes erstellt. Auf Basis dieser können mit den entsprechenden Methoden Codewörter erstellt und decodiert werden. Weiter kann mit einer Prüfmethode überprüft werden, ob ein gegebenes Codewort korrekt sei.

Inhaltsverzeichnis

1 Einleitung

In der Informatik werden Daten in Form von Bits dargestellt. Ein Bit kann den Zustand 1 oder den Zustand 0 annehmen. Es handelt sich also um ein binäres System.
Sämtliche bekannten Prozesse und Aktionen, sei es das Speichern dieser Arbeit, das Versenden einer E-Mail, die Tastenanschläge, um etwas zu schreiben, ein Mausklick, oder was auch immer als ganz normal und gegeben erscheint: Das alles sind Einsen und Nullen.

Im Folgenden wird meist von Datenübertragung gesprochen. Dabei sind sämtliche Prozesse gemeint, bei denen Daten gesendet und empfangen werden. So ist auch das Anschlagen einer Taste auf der Tastatur ein Senden von Daten, die vom Prozessor empfangen und verarbeitet werden, worauf nach weiteren Verarbeitungsschritten (Senden und Empfangen) der korrekte Buchstabe auf dem Bildschirm erscheint.

Es ist unschwer vorstellbar, dass bei so vielen Datenübertragungen Fehler passieren können.
Trotzdem ist nicht jeder Fehler als Fehler erkennbar. Dies ist den Fehlerkorrigierenden Codes geschuldet, die in dieser Arbeit am Beispiel von Hamming-Codes erläutert werden.

Die Hamming-Codes wurden von Richard Hamming im April 1950 veröffentlicht.
Hamming stellte fest, dass Computerberechnungen, wohlgemerkt zu dieser Zeit noch mit Lochkarten, immer mal wieder Fehler enthalten. [1]
Zwar existierten bereits fehlererkennende Codes, auf welche Mitarbeiter reagieren konnten. Dies befriedigte Hamming allerdings nicht, da, wie er in [1] zu bedenken gibt, die Computer auch über Nacht und am Wochenende in Betrieb sind.
Konnte niemand auf die Fehler reagieren führte dies dazu, dass Prozesse in falscher Art und Weise weitergeführt, oder sogar abgebrochen wurden.
Um dieses Problem zu lösen entwickelte Hamming die Fehlerkorrigierenden Codes.

Kapitel 2 dieser Arbeit beschreibt die Theoretischen Grundlagen solcher Codes, Kapitel 4 erläutert den zu dieser Arbeit gehörenden Python-Code, der Hamming-Codes erstellt und prüft.

2 Theorie

2.1 Alphabet

Ein Alphabet beschreibt die Menge aller zur Verfügung stehenden Zeichen. So besteht das Deutsche Alphabet ohne Umlaute und dem scharfen S aus den Buchstaben a-z in Gross- und Kleinschreibung. Sei S das Alphabet, so gilt:

$$S = \{a - z, A - Z\}$$

In der Computertechnologie besteht ein Alphabet in der Regel aus Einsen und Nullen. Es gilt:

$$S = \{0,1\}$$

Dabei bezeichnet S^* die Menge aller endlichen Zeichenketten, die aus S gebildet werden können. Ein Beispiel dazu folgt im folgenden Kapitel.

2.2 Wort

Ein Wort ist eine Kombination aus den Gegebenheiten des Alphabets. Sei $S = \{0,1\}$ und $m = 4$, so gilt:

$$S^m = \{0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,1111\}$$

Es gibt für das Alphabet S^4, für welches gilt $S = \{0,1\}$ also 16 Wörter.

2.3 Code

Ein Code C ist eine Auswahl, oder die Gesamtheit der Wörter, die mit einem gegebenen Alphabet dargestellt werden können.
Der Code $C = \{000,111\}$ ist gültig für S^3, $C = \{000,111,00101\}$ ist es aber nicht, da 00101 aus mehr als 3 «Buchstaben» besteht.

2.4 Abstand

Als Abstand wird die Differenz zweier Wörter aus einem Code bezeichnet.
In Binärcodes entspricht dieser Abstand der Anzahl Stellen, in welchen sich die Wörter unterscheiden.
Sei $C = \{0011,1100\}$ so unterscheiden sich die beiden Wörter in allen vier Positionen. Der Abstand $d(C)$ ist 4.
Wenn ein Code mehr als zwei Wörter beinhaltet, definiert sich der Abstand durch den kleinsten Abstand aus allen möglichen Kombinationen von Wörtern, unter der Berücksichtigung, dass ein Wort nicht mit sich selbst verglichen werden darf: [2]

$$d(C) = \min\{\delta(c, c') | c, c' \in C, c \neq c'\}$$

Für einen Code $C = \{0001,0101,1100,1001\}$ gilt demnach

$$\delta(0001,0101) = 1, \delta(0001,1100) = 3, \delta(0001,1001) = 1,$$
$$\delta(0101,1100) = 2, \delta(0101,1001) = 2, \delta(1100,1001) = 2$$

$$d(C) = \min(1,4,1,2,2,2) = 1$$

Um die Distanz zwischen zwei Wörtern zu finden, ohne Positionen vergleichen zu müssen, können die zwei Wörter addiert werden. Bei dieser Operation gilt zu beachten, dass $1 + 1 = 0$, ohne Übertrag auf die nächste Position.
Beispielsweise für den Abstand $\delta(0001,0101)$ gilt:

$$\begin{array}{rcccc} & 0 & 0 & 0 & 1 \\ + & 0 & 1 & 0 & 1 \\ \hline = & 0 & 1 & 0 & 0 \end{array}$$

2.5 Gewicht

Das Gewicht eines Wortes ist bestimmt durch die Anzahl Einsen in ihm. $\gamma(11001) = 3$.
Das Minimalgewicht eines Codes $g(C)$ ist definiert durch das kleinste Gewicht eines Wortes aus C, welches nicht 0 entspricht.

$$g(C) = \min\{\gamma(w) | \, w \in C, w \neq 0\}$$

2.6 Fehlererkennung

Sollte in einer Datenübertragung ein Fehler passieren, soll dieser festgestellt werden. EIN Fehler heisst konkret, dass genau ein Bit falsch übertragen wurde. Im Folgenden wird für falsch übertragene Bits das Vorkommen eines Bitflips genannt. Für die folgenden Erklärungen wird davon ausgegangen, dass maximal ein Bitflip stattfindet.

Gegeben sei der Code aus 2.4 und 0001 wird versendet. Flipt das Bit an zweiter Stelle wird 0101 empfangen. Dies ist ein gültiges Word in C, weshalb kein Fehler festgestellt werden kann. Wird allerdings 0101 versendet und beim Empfänger kommt 0111 an, so kann ein Fehler festgestellt werden. Dies aus der Tatsache, dass 0111 nicht Element von C ist. Für $\delta(w_1, w_2) = 1$ kann also ein Fehler nicht unbedingt festgestellt werden, für $\delta(w_1, w_2) = 2$ sei die Vermutung angestellt, dass dies funktioniere.

Um dieser Vermutung nachzugehen, stelle man sich den Abstand zwischen zwei Worten bildlich vor.
Der Verständlichkeit zuliebe sei $C = \{00,01,10,11\}$ und somit $d(C) = 1$. Es wird 01 gesendet und 11 empfangen (Abb. 1). Da der Abstand des Codes nur eins beträgt, ist unklar, dass dies eine fehlerhafte Übermittlung ist. Das empfangene Wort ist teil des Alphabets.

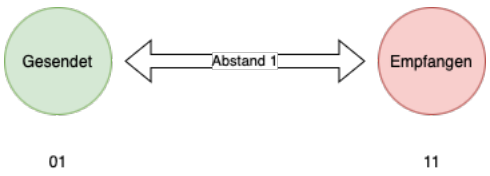

Abb. 1 - Abstand = 1

Sei $C = \{000,011\}$ und somit $d(C) = 2$. Wird 000 gesendet und 001 empfangen, so ist klar, dass ein Fehler vorliegt, da $\delta(000,001) = 1$ und $\delta(011,001) = 1$. Das Empfangen Wort befindet sich also zwischen den Möglichkeiten gemäss Alphabet.

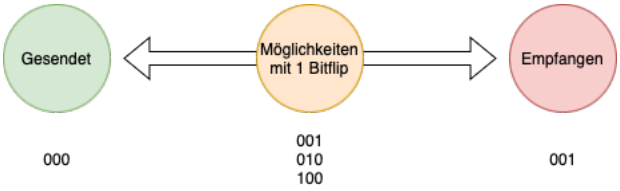

Abb. 2 - Abstand = 2

Ein Abstand der Grösse zwei reicht also aus, um genau einen Fehler zu erkennen.
An welcher Position der Bitflip stattfand ist allerdings nicht eindeutig feststellbar. Wäre im verwendeten Beispiel 010 beim Empfägner angekommen statt 001 würde ebenfalls ein Fehler erkannt. Ob aber ausgehend von 000 das letzte Bit flippte $000 \rightarrow 001$ oder ausgehend von 011 das zweite Bit $011 \rightarrow 001$, ist nicht eindeutig.

Um also einen Fehler erkennen zu können, braucht der Code Abstand 2.
Sei n die Anzahl Fehler, die erkannt werden können, so gilt allgemein

$$d(C) \geq n + 1$$

Ist $d(C) = 2$, gilt für n wie bereits gezeigt $2 \geq 1 + 1 \Rightarrow n = 1$.
Um entsprechend 2 Bitflips zu erkennen, bedarf es $d(C) \geq n + 1$ wobei $n = 2$, also $d(C) \geq 2 + 1$. Der Mindestabstand beträgt also 3. Es sei dabei festgehalten, dass bei einem Mindestabstand 3 in der Theorie zwei Bitflips festgestellt werden können, dies aber auch einem Bitflip entsprechen könnte. $C = \{00000, 11100\}$ mit zwei Bitflips in der Übertragung $00000 \rightarrow 11000$ könnte ebenso gut eine Übertragung mit einem Bitflip sein $11100 \rightarrow 11000$. Die Theorie besagt aber nicht, dass zwei Bitflips eindeutig identifiziert werden können, sondern lediglich, dass sie festgestellt werden können, was im Beispiel gegeben ist.

2.7 Fehlerkorrektur

Wird die Idee aus dem vorangegangenen Kapitel mit $C = \{00001,\ 11000, 10110\}$ und somit $d(C) = 3$ verfolgt, fällt auf, dass eindeutig festgestellt werden kann, welches das nächste bekannte Wort ist und somit, wo ein Bitflip stattfand.
Sei 00001 gesendet und 10001 empfangen, wird, wie bereits gesehen, ein Fehler erkannt. Um festzustellen, wo der Fehler stattfand, genügt es, die Distanzen zwischen den Worten zu ermitteln:

$$\delta(00001,10001) = 1$$
$$\delta(11000,10001) = 2$$
$$\delta(10110,10001) = 3$$

Das falsch übertragene Wort, ist demnach am nächsten an 00001, womit der Fehler korrigiert werden kann. Sollen zwei Fehler korrigiert werden können, bedarf es bereits einem Abstand der Grösse 5.
Sei $C = \{0000011,1111111, 1010100\}$ und die Übertragung mit zwei Bitflips sei $0000011 \rightarrow 0110011$, kann analog dem vorherigen Beispiel gerechnet werden:

$$\delta(0000011,0110011) = 2$$
$$\delta(1111111,0110011) = 3$$
$$\delta(1010100,0110011) = 4$$

Der geringste Abstand ist zwischen dem fehlerhaften Wort und 0000011, womit die zwei Fehler korrigiert werden können.
Allgemein gilt $d(C) \geq 2k + 1$, wobei k die Anzahl korrigierender Fehler darstellt.

Sollen also 3 Bitflips selbständig korrigiert werden können, bedarf es einem Code mit $d(C) \geq 2 * 3 + 1 = 7$ Mindestabstand.

3 Hamming Code

3.1 Linearer Code

Die möglichen Codes eines Alphabets S^n mit $S = \{0,1\}$ kann man sich als Unterräume des Vektorraums $\mathbb{Z}_2^n$ vorstellen. $\mathbb{Z}_2$ steht dabei für den Raum, der Modulo 2 entsteht, sprich nur Nullen und Einsen enthalten kann. $\mathbb{Z}^n$ definiert die Dimension des Vektorraums.
$C = \{000,111\}$ ist demnach ein Unterraum von $\mathbb{Z}_2^3$, da die Vektoren, die wir bisher Wörter genannt haben, aus drei Koordinaten bestehen, der Raum also dreidimensional ist, und aus den Ziffern 0 oder 1 bestehen.
Ein Code ist linear, wenn der Nullvektor enthalten ist und wenn die Summe zweier Codewörter wiederum ein Codewort ergibt. [2]
Am Beispiel $C = \{000,111\}$ geprüft:
000 ist enthalten, also Bedingung 1 erfüllt.

$$000 + 000 = 000$$
$$000 + 111 = 111$$
$$111 + 000 = 111$$
$$111 + 111 = 000$$

Sämtliche Additionen führen wieder zu einem Codewort. Somit Bedingung 2 ebenfalls erfüllt.
C ist also ein linearer Code und Unterraum von $\mathbb{Z}_2^3$. ■

3.2 Problematik

Mit der Theorie aus 2.7 wird klar, dass der Code mit zunehmender Wortgrösse entsprechend erweitert werden muss.
Soll ein Code $C = \{000,001,010,100,011,101,110,111\}$ so übertragen werden, dass Bitflips sämtlicher Stellen korrigiert werden können, bedarf es eines Mindestabstands der Grösse 7. Hier gilt $d(C) = 1$. Es müssten also alle Worte erweitert werden, bis $d(C) = 4$ gilt.
Dies kann durch ein Vervierfachen der Worte erreicht werden:

$$000 \rightarrow 000000000000$$
$$001 \rightarrow 001001001001$$
$$010 \rightarrow 010010010010$$
$$100 \rightarrow 100100100100$$
$$011 \rightarrow 011011011011$$
$$101 \rightarrow 101101101101$$
$$110 \rightarrow 110110110110$$
$$111 \rightarrow 111111111111$$

Sei nun aber $C = S^4, mit\ S = \{0,1\}$, also sämtliche Permutationen mit Nullen und Einsen auf vier Stellen ($0000, 0001, 0010$ usw.), müsste für die Korrektur der vier möglichen Bitflips ein Mindestabstand von $d(C) \geq 2 * 4 + 1 = 9$ gelten.
Um dies zu erreichen, müssten die Wörter mindestens verneunfacht werden, damit zwischen $0000 \rightarrow 000000000000000000000000000000000000$ und $0001 \rightarrow 000100010001000100010001000100010001$ ein Abstand der Grösse 9 entstünde. Die anderen Wörter und Distanzen seien hier ungeprüft, da bereits ersichtlich wird, dass dieses Vorgehen wenig effizient ist.

3.3 Parity Check

Hamming Codes überprüfen die gegebenen Wörter mit Überwachungs-Bits.
Diese speziellen Bits, die nicht zum Codewort selbst gehören, sind an strategischen Positionen im Wort eingebettet und überwachen bestimmte Positionen des Codewortes.
Diese strategischen Positionen sind alle Potenzen von 2.
Sei ein Wort aus S^4 gegeben $w = 1011$, so werden an Position $2^0, 2^1, 2^2$ Paritätsbits eingesetzt und das Hamming-Codewort lautet neu $101?1??$.
Um die Paritätsbits korrekt zu setzten, werden nach [3] zuerst die Positionen, an welchen sich eine Eins befindet, in Binärzahlen umgerechnet. Hier Position 7, 5 und 3. Entsprechend $111, 101, 011$. Diese 3 Binärzahlen werden addiert

$$\begin{array}{cccc} & 1 & 1 & 1 \\ + & 1 & 0 & 1 \\ + & 0 & 1 & 1 \\ \hline = & 0 & 0 & 1 \end{array}$$

Das Resultat entspricht den Paritätsbits und das Hamming-Codewort lautet 1010101.
Alternativ können die Paritätsbits auch mit einem Venn-Diagramm gefunden werden, wie Abb. 3 zeigt.

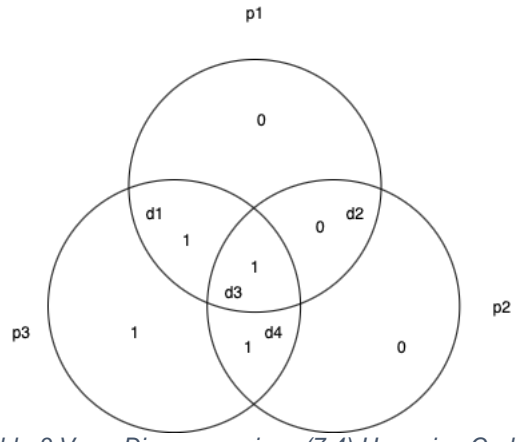

Abb. 3 Venn-Diagramm eines (7,4) Hamming Codes

Das Datenwort 1011 wird in die Bits $d1 - d4$ aufgeteilt und in die Schnittmengen der Paritätsbits $p1 - p3$ geschrieben. An diesem Beispiel wird die Idee der Parität einfacher sichtbar. $p1$ ist für die Bits $1,1,0$ zuständig, was einer geraden Anzahl Einsen entspricht. Somit muss $p1 = 0$ sein. $p3$ hingegen ist zuständig für die Bits $1,1,1$, weshalb $p3 = 1$ gelten muss, um eine gerade Anzahl Einsen zu erhalten.
Um 4 Datenbits zu prüfen, werden 3 Paritätsbits benötigt, was zu siebenstelligen Hamming-Codes führt.
Verallgemeinert lautet die Definition $2^k \geq m + k + 1$, wobei m der Anzahl Datenbits und k der Anzahl Paritätsbits entspricht. Sei n die Menge aller möglichen Positionen, folgt $m + k = n$ und daraus $2^m \leq \frac{2^n}{n+1}$. [1]
Dass für das Wort 1011 drei Paritätsbits verwendet werden, hätte demnach auch mit $2^k \geq 4 + k + 1$ berechnet werden können, womit klar gewesen wäre, dass wegen $2^3 = 8 \geq 4 + 3 + 1$, die Anzahl Paritätsbits gleich 3 sein muss. Eine andere Definition, die ebenfalls gefunden wird, ist $2^m - 1 = n$, wobei m der Anzahl Paritätsbits entspricht! Daraus folgt für einen Code mit 3 Paritätsbits, dass die er $n = 2^3 - 1 = 7$ Zeichen lang ist und aus 3 Paritätsbits, sowie $n - m = 4$ Datenbits besteht. Diese Definition wird in der zu dieser Arbeit gehörenden Umsetzung in Python verwendet, da es in der Aufgabenstellung entsprechend gefordert wurde.

Die Zuständigkeit der Paritätsbits ist in diesem Beispiel absolut zufällig gewählt, erfüllt aber ihren Zweck. Im Abschnitt 4, welcher die Umsetzung in Python behandelt, wird eine weitere Möglichkeit vorgestellt, welche die Erweiterbarkeit auf längere Codes sicherstellt.

3.4 Generator-Matrix

Anstelle von Venn-Diagrammen und Additionen, können die die resultierenden Codewörter mit Generator-Matrizen generiert und, wie später erläutert wird, mit Parity-Check-Matrizen überprüft werden.

Die Generator-Matrix G ist von der Form $G = (E_m|A)$, wobei E_m die Einheitsmatrix des m-dimensionalen Raums und A die Prüfmatrix darstellen.
Es wird weiterhin das Wort 1011 als Beispiel verwendet. Ebenfalls gilt weiterhin $p1 = d1 \oplus d2 \oplus d3, p2 = d2 \oplus d3 \oplus d4, p3 = d1 \oplus d3 \oplus d4$.

$$E_k = \begin{pmatrix} d1 & d2 & d3 & d4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Um die Prüfmatrix A zu bilden, wird zeilenweise die Definition der Paritätsbits aus E_k übertragen. $p1$ wird beispielhaft mit den Berechnungen notiert:

$$A = \begin{pmatrix} p1 & p2 & p3 \\ 1 \oplus 0 \oplus 0 = 1 & 0 & 1 \\ 0 \oplus 1 \oplus 0 = 1 & 1 & 0 \\ 0 \oplus 0 \oplus 1 = 1 & 1 & 1 \\ 0 \oplus 0 \oplus 0 = 0 & 1 & 1 \end{pmatrix}$$

Daraus resultiert die Generatormatrix

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

Mit dieser Matrix kann jedes Wort aus dem linearen Code C aus S^4 mit $S = \{0,1\}$ in einen entsprechenden Hamming Code transcodiert werden.

$$(1011) \cdot G = (1011001)$$

Das Wort 1011001 entspricht dem Codewort 1011 und den drei Paritätsbits $0,0,1$. Ob also eine der zwei Methoden aus 3.3, oder die Generator-Matrix verwendet werden hat auf das Resultat keinen Einfluss. Alle führen zum selben Ergebnis. Werden auch in der Generator-Matrix die Paritätsbits an den Positionen $2^0, 2^1, 2^2 \ldots 2^n$ angeordnet, entsteht die Matrix G' und $(1011) \cdot G'$ führt zu exakt demselben Hamming-Codewort wie in 3.3 berechnet.

$$G = \begin{pmatrix} d1 & d2 & d3 & p1 & d4 & p2 & p3 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

$$(1011) \cdot G' = (1010101)$$

3.5 Prüfmatrix

Wird das in 3.4 berechnete Hamming-Codewort versendet, will der Empfänger selbstverständlich dessen Korrektheit prüfen.
Während die Generator-Matrix in der Form $G = (E_k|A)$ vorliegt, ist es bei der Prüfmatrix $H = (A^T|E_{n-m})$. A^T bedeutet, dass die Matrix transponiert wird, also aus Zeilen Spalten

werden. Für die gegebene Matrix $A = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$ heisst das $A^T = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$.

E_{n-m} ist die Einheitsmatrix des $n-m$-Dimensionalen Raums.
Im gegebenen Beispiel ist $n = 7$, $m = 4$. Gesucht ist die Einheitsmatrix des dreidimensionalen Raums $E_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

Die resultierende Prüfmatrix H lautet daher:

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

Ein Wort w ist genau dann ein Codewort aus C, wenn $H \cdot w^t = \{0\}$ gilt. [2]
Am Hamming-Codewort aus 3.4 demonstriert bedeutet dies

$$\begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 0 \cdot 1 + 1 \cdot 0 + 0 \cdot 0 + 0 \cdot 1 = 0 \\ 0 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 0 + 0 \cdot 1 = 0 \\ 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 1 = 0 \end{pmatrix}$$

Womit 1011001 ein gültiger Hamming-Code eines Wortes aus C ist.

3.6 Fehlerkorrektur

Wäre in der Übertragung ein Bitflip passiert und anstelle 1011001 wäre 1111001 übertragen worden, lautete das Resultat der Überprüfung

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

Da das Resultat ungleich 0 ist, ist bereits klar, dass ein Fehler in der Übertragung stattfand. Was zudem spannend und äusserst hilfreich ist, ist die Tatsache, dass das Resultat 110 als Dezimalzahl 6 lautet, was der Position des geflippten Bits entspricht, wenn von rechts nach links gezählt wird und die Position ganz rechts Position 1 ist.

Etwas einfacher ist der Umstand, dass $\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$ dem Matrix-Vektor an zweiter Stelle entspricht und der Bitflip an zweiter Stelle im Codewort geschah. Dieser Ansatz ist zum

Programmieren sinnvoller, da lediglich eine Gleichheitsprüfung über alle Spaltenvektoren vorgenommen werden muss.

4 Umsetzung in Python

Der gesamte Code befindet sich im Anhang und im öffentlichen Repository auf Github (https://github.com/tobias-hu/ffhs_linalg). Einige Stellen seien hier trotzdem erwähnt und erläutert.

Sämtliche Überprüfungen und Matrizen sind davon abhängig, wie lange der gegebene Code ist.
Daher verfügt die Klasse *HammingCode* über einen eher grossen Konstruktor, der die Konstanten

- *self.n* (Länge des Codewortes)
- *self.m* (Anzahl Paritätsbits)
- *self.k* (Anzahl Datenbits)
- *self.E* (Einheitsmatrix)
- *self.databitPositions* (Positionen der Datenbits im Codewort)
- *self.paritybitPositions* (Positionen der Paritätsbits im Codewort)
- *self.A* (Paritätsmatrix)
- *self.G* (Generatormatrix)
- *self.P* (Checkmatrix)

befüllt.

```
def __init__(self, m):
    # Codeword length
    self.n = 2 ** m - 1
    # number of parity bits
    self.m = m
    # number of data bits
    self.k = self.n - self.m
    # EinheitsMatrix
    self.E = _getEinheitsMatrix(self.k)
    self.databitPositions, self.paritybitPositions = self._getPari-
tybitAndDatabitPositions()
    self.parityResponsibility = _getParityBitResponsibility(self.databit-
Positions, self.paritybitPositions)
    # ParityMatrix
    self.A = self._getParityMatrix(self.parityResponsibility)
    # GeneratorMatrix
    self.G = self.get_generator_matrix()
    # Checkmatrix
    self.P = self.get_check_matrix()
```

Die Generatormatrix ist von der Form $G = (E_k|A)$. Da E_k und A in direkter Abhängigkeit zur Codelänge stehen, sind sie bereits generiert. Die *get_generator_matrix()* Methode braucht die beiden Matrizen nur noch zusammenzufügen.

```
def get_generator_matrix(self):
    """
    returns the generatormatrix
    this is fully calculated by the given length self.m which is defined
when the object is created
    :return:
    """
    G = np.vstack([self.E, self.A])
    self.G = np.array(G).transpose()
    return self.G
```

Wie bereits in 3.3 erwähnt, kan die Zuständigkeit der Paritätsbits rein zufällig gwählt werden. Bei einem (7,4)-Hamming-Code ist dies mit einem Venn-Diagramm rasch erledigt, die einzige Schwierigkeit ist, nicht zweimal dasselbe Datenbit in eine Schnittmenge einzutragen. Diese Hürde wird als sehr tief betrachtet.
Wird der Code allerdings länger, oder soll eine Paritätsmatrix programmiert werden, so soll dies ohne Zufall, dafür mit Logik geschehen. Diesem Umstand war sich auch Hamming bewusst, weshalb er in [1] folgende Idee festhielt:
Ein Paritätsbit steht immer an einer Stelle 2^n. Also 1, 2, 4, 8, 16 usw.
Binär dargestellt befinden sich die Paritätsbits an $0001, 0010, 0100, 1000$ usw.
Jedes Paritätsbit überwacht diejenigen Positionen, die ebenfalls eine 1 an der Position haben, an welcher das Paritätsbit eine 1 hat.
Tabelle 1 erläutert die Idee.

Paritätsbit Nummer	Position	Position Binär	Verantwortlich für (Binär)	Verantwortlich für (Dezimal)
1	1	0001	0001, 0011, 0101, 0111, 1001, 1011,	1, 3, 5, 7, 9, 11
2	2	0010	0010, 0011, 0110, 0111, 1010, 1011,	2, 3, 6, 7, 10, 11
3	4	0100	0100, 0101, 0110, 0111, 1100, 1101	4, 5, 6, 7, 12, 13
4	8	1000	1000, 1001, 1010, 1011, 1100, 1101	8, 9, 10, 11, 12, 13

Tabelle 1 Paritätsbits Verantwortlichkeiten

Um die Positionen der Paritätsbits zu berechnen, wird die Methode *_getParitybitAndDatabitPositions()* verwendet.
Diese iteriert über die Länge der Codewörter (*self.n*) und prüft für jeden Index, ob er eine Potenz von zwei ist.
Wenn ja, wird die binäre Darstellung der Position zum Array *paritybitPositions* hinzugefügt, ansonsten wird die binäre Darstellung zum Array *databitPositions* hinzugefügt.

```
def _getParitybitAndDatabitPositions(self):
    """
    Returns the positions of the parity- and the databits corresponding
to the codelength
    :return:
    """
    databitPositions = []
    paritybitPositions = []
    for i in range(0, self.n):
        if (i + 1) & i == 0:
            """position of a paritybit"""
            paritybitPositions.append(np.binary_repr(i + 1, len(np.bi-
nary_repr(self.n))))
        else:
            """position of a databit"""
            databitPositions.append(np.binary_repr(i + 1, len(np.bi-
nary_repr(self.n))))
    return databitPositions, paritybitPositions
```

Mit diesen beiden Arrays können anschliessend mit der Methode *_getParityBitResponsibility(databitPositions, paritybitPositions)* die Verantwortlichkeiten analog Tabelle 1 berechnet werden.

```
def _getParityBitResponsibility(databitPositions, paritybitPositions):
    """
    Returns the responsibility of paritybits for the given paritybit- and
databitpositions
    Paritybit at positon 001 is responsible for all databits at position
xxx1
    Paritybit at position 010 is responsible for all databits at position
xx1x
    etc.
    :param databitPositions:
    :param paritybitPositions:
    :return:
    """
    parityResponsibility = []
    for p in paritybitPositions:
        # get the index where 1 stands
        responsibleForIndex = p.index('1')

        # get indexes of databits with 1 in responsibleindex
        responsibleForDatabits = []
        i = 0
        for d in databitPositions:
            if d[responsibleForIndex] == '1':
                responsibleForDatabits.append(i)

            i += 1
        parityResponsibility.append(responsibleForDatabits)
    return parityResponsibility
```

Die Paritätsmatrix berechnet sich anschliessend aus den zu überwachenden Datenbits, wie in 3.4 beschrieben.

```
def _getParityMatrix(self, parityResponsibility):
    """
    Returns the Paritymatrix.
    This is the part where it matters which paritybit is responsible for
which databits
    :param parityResponsibility:
    :return:
    """
    A = []
    for element in parityResponsibility:
        vector = [0 for _ in range(len(self.E[0]))]
        for index in element:
            vector = np.array((np.array(vector) + self.E[index]) % 2)

        A.append(vector)
    self.A = np.array(A)
    return self.A
```

Die Methode *_resultOfCheckmatrix(codeword)* überprüft das gegebene Codewort mit der Checkmatrix und gibt den resultierenden Vektor zurück.
In 3.5 wurde erklärt, dass dieser **0** sein müsste. Ist er es nicht, kann gemäss 3.6 festgestellt werden, wo ein Fehler vorliegt.

Dies geschieht mit der Methode *getCorrection(codeword)*

```
def getCorrection(self, codeword):
    """
    Returns the correction of the given codeword
    if the codeword is valid there is nothing to correct and the given
codeword is returned
    :param codeword:
    :return:
    """
    codeword = self._formatCodeword(codeword)
    result = self._resultOfCheckmatrix(codeword)
    match = np.all(self.P.transpose() == result, axis=1)
    errorposition = np.where(match)[0].squeeze()

    # if there is no error, return codeword
    if errorposition.size == 0:
        return codeword

    codeword[errorposition] = (codeword[errorposition] + 1) % 2
    return codeword
```

5 Diskussion

Hamming-Codes sind ein guter Einstieg, um Fehlerkorrigierende Codes zu verstehen.
In der praktischen Anwendung gilt es jeweils zu entscheiden, wieviel «Risiko» man eingehen möchte.
Ein klassischer Hamming-Code (es gibt auch erweiterte Hamming-Codes) hat einen minimalabstand von 3. Das heisst, er kann genau einen Fehler korrigieren.
Wie in dieser Arbeit geschildert, oder auch im angehängten Python-Code selbst auszuprobieren, definiert sich die Codelänge durch $2^m - 1$, wobei m der Anzahl Paritätsbits entspricht. Spricht man von einem (7,4)-Hamming Code, bedeutet dies eine Codewortlänge von 7 Bits und eine Menge von 4 Datenbits. Es bleiben entsprechend 3 Paritätsbits.
Die nächstgrössere Codewortlänge wäre $2^4 - 1 = 15$, mit 4 Paritätsbits und 11 Datenbits.

Code	Codewortlänge	Datenbits	Paritätsbits
(7,4)	7	4	3
(15,11)	15	11	4
(31,26)	31	26	5
(63,57)	63	57	6

Tabelle 2 Codewortlängen

Tabelle 2 zeigt vier verschiedene Hamming-Codes. Es fällt auf, dass die Schritte der Paritätsbits im Vergleich zu den überprüften Datenbits klein sind. So können mit dem Sprung von 3 auf 4 Paritätsbits 7 Datenbits mehr überprüft werden, mit 5 Paritätsbits 22 Datenbits mehr.

Das einleitend erwähnte «Risiko» besteht nun darin abzuwägen, wie effizient die Übertragung sein soll, da sämtliche Daten in Blöcke der Grösse der Anzahl Datenbits geteilt werden müssen, dann das Codewort berechnet werden muss und der Empfänger das Codewort prüfen, gegebenenfalls korrigieren, dann zurückrechnen und die unterteilten Daten wieder zusammensetzen muss. Diese Tatsache lässt darauf schliessen, dass möglichst grosse Blöcke übertragen werden sollten. Dagegen spricht, dass der Hamming-Code genau einen Fehler korrigiert. Die Wahrscheinlichkeit, dass in einem Codewort der Länge 63 mehr als ein Fehler vorkommt, ist signifikant höher, als dass in einem Codewort der Länge 7 mehr als ein Fehler vorkommt.

6 Anhang

```
"""
Created on 13.11.2024
Author: Tobias Gasche
Description: HammingCode Class. Provides methods to generate generator-
matrix and check-matrix for
    Hamming-Codes. Also provides a method to check if a codeword is valid
"""

import numpy as np


def _getParityBitResponsibility(databitPositions, paritybitPositions):
    """
    Returns the responsibility of paritybits for the given paritybit- and
databitpositions
    Paritybit at positon 001 is responsible for all databits at position
xxx1
    Paritybit at position 010 is responsible for all databits at position
xx1x
    etc.
    :param databitPositions:
    :param paritybitPositions:
    :return:
    """
    parityResponsibility = []
    for p in paritybitPositions:
        # get the index where 1 stands
        responsibleForIndex = p.index('1')

        # get indexes of databits with 1 in responsibleindex
        responsibleForDatabits = []
        i = 0
        for d in databitPositions:
            if d[responsibleForIndex] == '1':
                responsibleForDatabits.append(i)

            i += 1
        parityResponsibility.append(responsibleForDatabits)
    return parityResponsibility


def _getEinheitsMatrix(length):
    """
    returns the Einheitsmatrix for the given length
    :param length:
    :return:
    """
    E = []
    for i in range(0, length):
        inner = []
        for j in range(0, length):
            if j == i:
                inner.append(1)
            else:
                inner.append(0)
        E.append(inner)
    return np.array(E)
```

```
class HammingCode:
    """
    Provides Methods to gnerate
    generator matrix
    checkmatrix
    paritymatrix
    check codewords
    encode words
    decode codewords
    """
    def __init__(self, m):
        # Codeword length
        self.n = 2 ** m - 1
        # number of parity bits
        self.m = m
        # number of data bits
        self.k = self.n - self.m
        # EinheitsMatrix
        self.E = _getEinheitsMatrix(self.k)
        self.databitPositions, self.paritybitPositions = self._getPari-
tybitAndDatabitPositions()
        self.parityResponsibility = _getParityBitResponsibility(self.dat-
abitPositions, self.paritybitPositions)
        # ParityMatrix
        self.A = self._getParityMatrix(self.parityResponsibility)
        # GeneratorMatrix
        self.G = self.get_generator_matrix()
        # Checkmatrix
        self.P = self.get_check_matrix()

    def get_generator_matrix(self):
        """
        returns the generatormatrix
        this is fully calculated by the given length self.m which is de-
fined when the object is created
        :return:
        """
        G = np.vstack([self.E, self.A])
        self.G = np.array(G).transpose()
        return self.G

    def _getParityMatrix(self, parityResponsibility):
        """
        Returns the Paritymatrix.
        This is the part where it matters which paritybit is responsible
for which databits
        :param parityResponsibility:
        :return:
        """
        A = []
        for element in parityResponsibility:
            vector = [0 for _ in range(len(self.E[0]))]
            for index in element:
                vector = np.array((np.array(vector) + self.E[index]) % 2)

            A.append(vector)
        self.A = np.array(A)
        return self.A
```

```
    def get_check_matrix(self):
        """
        Returns the checkmatrix
        Checkmatrix is (A.transpose | E_n-k)
        """
        At = self.A.transpose()
        Enk = _getEinheitsMatrix(self.n - self.k)
        Enk = Enk.transpose()

        self.P = np.vstack([At, Enk]).transpose()
        return self.P

    def encode(self, word):
        """
        Encodes a given word with the generatormatrix created for the
given codelength
        :param word:
        :return:
        """
        if self._check_word(word):
            wordVector = [int(i) for i in str(word)]
            wordVector = np.array([wordVector])
            result = np.dot(wordVector, self.G) % 2
            return result

    def decode(self, codeword):
        """
        Decodes a given codeword
        Checks whether the codeword is valid. if it is not, the codeword
is first corrected
        :param codeword:
        :return:
        """
        # first -> check codeword
        if not self.check_codeword(codeword):
            codeword = self.getCorrection(codeword)

        codeword = self._formatCodeword(codeword)
        codewordVector = [int(i) for i in codeword]
        return codewordVector[:4]

    def check_codeword(self, codeword):
        """
        Checks if the given codeword is valid
        :param codeword:
        :return:
        """
        codeword = self._formatCodeword(codeword)

        if len(codeword) != self.n:
            raise Exception("Codeword to short")

        result = self._resultOfCheckmatrix(codeword)

        return np.array_equal(result, [0, 0, 0])
```

```
    def getCorrection(self, codeword):
        """
        Returns the correction of the given codeword
        if the codeword is valid there is nothing to correct and the
given codeword is returned
        :param codeword:
        :return:
        """
        codeword = self._formatCodeword(codeword)
        result = self._resultOfCheckmatrix(codeword)
        match = np.all(self.P.transpose() == result, axis=1)
        errorposition = np.where(match)[0].squeeze()

        # if there is no error, return codeword
        if errorposition.size == 0:
            return codeword

        codeword[errorposition] = (codeword[errorposition] + 1) % 2
        return codeword

    def _getParitybitAndDatabitPositions(self):
        """
        Returns the positions of the parity- and the databits correspond-
ing to the codelength
        :return:
        """
        databitPositions = []
        paritybitPositions = []
        for i in range(0, self.n):
            if (i + 1) & i == 0:
                """position of a paritybit"""
                paritybitPositions.append(np.binary_repr(i + 1,
len(np.binary_repr(self.n))))
            else:
                """position of a databit"""
                databitPositions.append(np.binary_repr(i + 1, len(np.bi-
nary_repr(self.n))))
        return databitPositions, paritybitPositions

    def _check_word(self, word):
        """
        Checks if the word is valid
        Returns true or raises exceptions
        :param word:
        :return:
        """
        if len(str(word)) != self.k:
            raise Exception("Code word has to be {} but was {}.".for-
mat(self.k, len(str(word))))

            # check characters
        for char in str(word):
            if char != '0' and char != '1':
                raise Exception("{} detected. Word has to be binary. Only
use 0 and 1 characters.".format(char))

        return True
```

```
    def bitFlip(self, codeword, position):
        """
        Little helper function to flip a bit in a certain position
        returns the given codeword with a flipped bit at the given posi-
tion
        :param codeword:
        :param position:
        :return:
        """
        if position > len(codeword[0]):
            raise Exception("C'mon... You can't flip more than you
have!")
        if position > 0:
            position = position - 1
        if position < 0:
            raise Exception("Position can't be negative.")
        if position == 0:
            print("Position can't be 0. We don't look at it as program-
mers would do. I took position 1")
        codeword[0][position] = (codeword[0][position] + 1) % 2
        return codeword

    def _resultOfCheckmatrix(self, codeword):
        """
        Multiplies the checkmatrix with the codewordvector
        Returns the resulting vector
        :param codeword:
        :return:
        """
        codeWordVector = np.array([codeword]).transpose()
        result = np.dot(self.P, codeWordVector) % 2
        return np.squeeze(result)

    def _formatCodeword(self, codeword):
        """
        Helperfunction to get the codeword as list wheather it was a
ndarray or an integer
        :param codeword:
        :return:
        """
        if isinstance(codeword, np.ndarray):
            if len(codeword) != 1:
                raise Exception("Codeword can only have one vector")
            codeword = codeword[0].tolist()

        if isinstance(codeword, int):
            codeword = [int(i) for i in str(codeword)]

        codeword = [int(i) for i in codeword]

        return codeword
```

Literatur- und Quellenverzeichnis

[1] R. W. Hamming, "Error Detection and Error Correcting Codes," *The Bell System Technical Journal,* vol. XXIX No.2, 1950.

[2] R. Socher, Mathematik für Informatiker, Fachbuchverlag Leipzig im Carl-Hanser-Verlag, 2011.

[3] F. Dalwigk, "YouTube - Hamming-Code berechnen," 25 März 2019. [Online]. Available: https://www.youtube.com/watch?v=nEGeRkhLoTk&t=167s. [Accessed 11 November 2024].

Bildverzeichnis

Tabellenverzeichnis