Computer Visualisierungen ausgewählter Themen der Kodierungstheorie

Michael Seidel

30. Januar 2011

Computer Visualisierung ausgewählter Themen der Kodierungstheorie

Schriftliche Hausarbeit im Rahmen der
Ersten Staatsprüfung für das Lehramt
an Gymnasien und Gesamtschulen

dem Landesprüfungsamt
für Erste Staatsprüfungen
für Lehrämter an Schulen
- Geschäftsstelle Siegen -

vorgelegt von:

Michael Seidel

Siegen, 30.01.2011

Gutachter:

Prof. Dr. Nils Peter Skoruppa

Algebra und Zahlentheorie
Universität Siegen

Inhaltsverzeichnis

1 Einleitung **2**

2 Grundlagen und Notationen **5**

3 ISBN-Code **9**

3.1 Definition und Eigenschaften 10

3.2 Visualisierung 12

3.3 Programmierung 15

4 Lineare Codes **19**

4.1 Definitionen und Eigenschaften 20

4.1.1 Hamming-Code (H_7) 24

4.1.2 Erweiterter Hamming-Code (H_8) 25

4.1.3 Binärer Golay-Code (G_{23}) 26

4.1.4 Erweiterter Binärer Golay-Code (G_{24}) 27

4.2 Visualisierung 28

4.3 Programmierung 34

5 Fazit und Ausblick **46**

6 Literatur **48**

7 Anlagen **50**

1 Einleitung

Kodiert werden Informationen seit der Mensch in der Lage ist, zu kommunizieren. Sei es das einfache bestätigende Nicken als Antwort auf eine Frage, oder auch komplexe Gebärdensprache. Bedeutungen werden in Signale umgewandelt und übertragen. Der Empfänger[1] interpretiert die Signale und versteht so wieder die Bedeutung. Dieses Konzept ist schon lange in Gebrauch.

Erstaunlich ist nur, dass kaum jemand Notiz davon nimmt. Mit manchen Themen setzt man sich scheinbar erst auseinander, wenn sie Probleme aufwerfen. Kommunikation existiert und funktioniert. Warum soll man sich also damit aufhalten, die Eigenschaften im Detail zu analysieren?

Interessant geworden sind diese Details erst, als man technischen Geräten beibringen wollte, miteinander zu kommunizieren. Anfangs musste jeder hierbei auftretende Fehler von Menschen erkannt werden. Eine Korrektur wurde oft nur dadurch erreicht, dass man einen Vorgang solange wiederholt hat, bis man keinen Fehler mehr erkennen konnte.

Die Datenmengen sind seit der Zeit geradezu explodiert, so dass eine manuelle Überprüfung überhaupt nicht mehr möglich ist. Auch kann man es sich zeitlich nicht mehr erlauben, Arbeitsschritte mehrfach durchzuführen. Also verarbeitet man die Daten nicht mehr *roh*, sondern benutzt Kodierungen, die ein Gerät in die Lage versetzen, Fehler eigenständig zu erkennen, oder zu korrigieren.

[1] Allgemeine Personenbezeichnungen gelten immer für die männliche und weibliche Form.

Heute findet man Kodierungen überall in der Technik. Sei es Telekommunikation, digitales Fernsehen oder Weltraummissionen. Selbst Datenträger wie Festplatten oder CDs benutzen intern Fehler-korrigierende Kodes.

Die Mathematik bringt immer bessere und effizientere Kodierungen hervor. Dabei bedient sie sich einer vergleichsweise hohen Anzahl von Teilgebieten, so dass eine breitgefächerte mathematische Bildung notwendig ist, um Kodierungen zu verstehen. Die Erklärungen sind oft sehr abstrakt gehalten, was das Verständnis zusätzlich erschwert.

Und das ist genau der Punkt, an dem die hiermit vorliegende Arbeit ansetzt. Bestehende und in der Lehre verbreitete Kodierungen werden in einem ersten Schritt formal erklärt, um anschließend in einem zweiten Schritt mit Hilfe von Visualisierungen greifbar und erfahrbar gemacht zu werden.

Technisch realisiert wird das Vorhaben mit dem Softwarepaket *Sage*, welches bereits in der Lehre an der Universität Siegen Verwendung findet. Die Bedienung erfolgt über ein Web-Interface. *Sage* arbeitet mit der Programmiersprache *Python*, so dass sich auch komplexere Programme komfortabel umsetzen lassen.

Mit dem Kapitel *Grundlagen und Notationen* wird der Grundstein für die weiteren Betrachtungen gelegt. Erläutert werden basale Eigenschaften von Kodierungen sowie verschiedene Notationen, die im Verlauf der Arbeit benutzt werden. Alle verwendeten Definitionen und Notationen entsprechen in der Kodierungstheorie weit verbreiteten Konventionen und lassen sich daher in vielen Quellen finden. Die Zusammenstellung orientiert sich an den Bedürfnissen der nachfolgenden Programmierung und erhebt keinerlei Anspruch auf Vollständigkeit.

Nachdem die Grundlagen beschrieben worden sind, folgen konkrete Kodierungen. Die erste Kodierung ist die *ISBN*. Sie ist genau so prominent wie unbekannt. Jeder weiß, wo eine *ISBN* zu finden ist, aber kaum jemand ist sich der tatsächlichen Eigenschaften bewusst. Landläufig hält man sie für eine fortlaufende Nummer, die sich nur dadurch auszeichnet, zentralisiert und international einheitlich für die Kennzeichnung von Büchern vergeben zu werden.

Anschließend folgt ein umfangreicheres Programm, das den Vergleich verschiedener linearer Codes erlaubt. Konkret werden H_7 und G_{23}, sowie deren Erweiterungen thematisiert. Was es genau damit auf sich hat, werden wir im angesprochenen Kapitel erarbeiten.

Als Abschluss der Arbeit wird eine Nachbetrachtung formuliert, bei der sowohl Stärken als auch Schwächen sowie Potentiale für zukünftige Erweiterungen ausführlich beschrieben werden.

2 Grundlagen und Notationen

Eine Nachricht wird von einem Sender zu einem Empfänger durch einen Kanal gesendet. Die Nachricht besteht aus Wörtern, die über einem endlichen Alphabet A gebildet werden. Elemente von A heißen *Symbole*. A^* bezeichnet die Menge aller Wörter über einem Alphabet A und A^n die Menge aller Wörter über dem Alphabet A mit der Länge n.

Innerhalb des Datenkanals benutzt man Worte, die über einem endlichen Alphabet Σ gebildet werden. Die Mengen Σ^* und Σ^n sind analog zu A^* und A^n definiert. Die Bezeichnungen orientieren sich an den Ausführungen von Matthes (vergleiche [RM03, S.198]).

Definition 2.1. ([RM03, S. 198]) *Eine injektive Abbildung*

$$C\colon A \to \Sigma^*$$

heißt Kodierung.

Nach Matthes (Siehe [RM03, S. 198]) genügt es, die Abbildung für Symbole zu definieren, da diese aneinandergereiht die Nachricht ergeben. $C(A)$ heißt *Code*, die Elemente des Codes *Codewörter*. Haben alle Codewörter die Länge n, handelt es sich um einen **Blockcode** der Länge n.

Ein Empfänger muss empfangene Wörter decodieren, um wieder die Nachricht zu erhalten. Matthes beschreibt in diesem Zusammenhang (Vergleiche [RM03, S. 201]), dass Worte, die durch die Übertragung verfälscht worden sind, zunächst

durch eine passende Fehlerkorrektur wieder in ein Codewort überführt werden. Jedes Wort wird durch das nächst gelegene Codewort ersetzt. Diese Vorgehensweise nennt er **maximum-likelihood-Methode**. Da die Kodierung injektiv ist, lässt sich von einem Codewort eindeutig auf ein Symbol schließen und so die Nachricht erhalten.

Wir benötigen für die maximum-likelihood-Methode einen Abstand zwischen Wörtern, um entscheiden zu können, welches das nächst gelegene Codewort ist. Genau wie Matthes ([RM03, S.201]), wählen wir dafür den Hamming-Abstand.

Definition 2.2. ([RM03, S.210]) *Seien $a, b \in \Sigma^n$, wobei $a = a_1 \dots a_n$ sowie $b = b_1 \dots b_n$ ist. Dann heißt*

$$d(a, b) := \#\{i | a_i \neq b_i\}$$

Hamming-Abstand von a und b.

Satz 2.3. ([RM03, S. 211]) *Der Hamming-Abstand ist eine Metrik.*

Beweis: ([RM03, S. 211]) Zu zeigen sind die drei Metrikaxiome:

1. $d(a, b) = 0$ genau dann, wenn $a = b$ ist.

2. $d(a, b) = d(b, a)$

3. Es gilt die Dreiecksungleichung. Das heißt, für drei Worte a, b und c ist $d(a, c) \leq d(a, b) + d(b, c)$.

Zu 1.: Zwei Worte sind genau dann gleich, wenn sie in jedem Buchstaben übereinstimmen.

Zu 2.: Nach Definition 2.2 ist $d(a, b) = \#\{i | a_i \neq b_i\}$. Außerdem ist $a \neq b$ genau dann, wenn $b \neq a$ ist. Damit ist also $\#\{i | a_i \neq b_i\} = \#\{i | b_i \neq a_i\}$. Weil auch $\#\{i | b_i \neq a_i\} = d(b, a)$ ist, gilt insgesamt, dass $d(a, b) = d(b, a)$ ist.

Zu 3.: Das Wort a unterscheide sich von b an k Stellen, das Wort b seinerseits von c an l Stellen. Das Wort a kann sich dann von c an höchstens $k + l$ Stellen unterscheiden. □

Durch Kodierung lässt sich eine bestimmte Anzahl von Fehlern erkennen, ein Teil davon sogar korrigieren. Die Korrektur selbst kann jedoch auch fehlerhaft verlaufen, wie wir noch zeigen werden.

Definition 2.4. ([RM03, S. 211]) *Für einen Code* C *heißt*

$$d_C := \min\{\, d(a,b) \,|\, a, b \in C, a \neq b \,\}$$

der Minimalabstand von C*.*

Nach Matthes ([RM03, S. 211]) lassen sich zu jedem Codewort a ähnliche Worte finden, die sich an bis zu r Stellen von a unterscheiden. Sie bilden Kugeln um a mit dem Radius r, die wie folgt definiert sind:

$$K_r(a) := \{w \in \Sigma^n | d(w,a) \leq r\}$$

Der Radius sei so gewählt, dass zwar möglichst viele Elemente, aber keine weiteren Codeworte in der Kugel enthalten sind. Für diesen Fall gilt also $r = d_C - 1$, wobei d_C den *Minimalabstand* des Codes C bezeichnet.

Der Mindestabstand d_C gibt an, wie viele Veränderungen an einem Codewort mindestens nötig sind, um ein weiteres Codewort zu erhalten. Treten weniger als d_C viele Veränderungen auf, handelt es sich also entweder um das unveränderte Codewort, oder um kein Codewort mehr. Dies gilt sicher für bis zu $d_C - 1$ Fehler in einem Block. Für mehr Fehler ist nicht ausgeschlossen, dass es sich um ein anderes Codewort handelt. C heißt daher $d_C - 1$ Fehler-erkennend.

Für die Korrektur von Fehlern suchen wir zu einem empfangenen Wort das Codewort, welches den geringsten Hamming-Abstand hat. Dazu bilden wird in Anlehnung an Matthes ([RM03, S. 212]) um die Codeworte Kugeln mit Radius $\lfloor \frac{d_C - 1}{2} \rfloor$. Die Kugeln sind disjunkt, da ein Wort w, das in zwei verschiedenen Kugeln liegt, zu zwei verschiedenen Codeworten c_1 und c_2 einen Abstand $\leq \lfloor \frac{d_C - 1}{2} \rfloor$ haben müsste. Das kann jedoch nicht sein. Die Dreiecksungleichung für den Hamming-Abstand besagt, dass $d(c_1, c_2) \leq d(c_1, w) + d(w, c_2)$ ist. Wir betrachten ohne Beschränkung der Allgemeinheit den Fall, dass $d_C = d(c_1, c_2)$ und damit $d_C \leq d(c_1, w) + d(w, c_2)$ ist.

Gleichzeitig ist $d(c_1, w) \leq \lfloor \frac{d_C - 1}{2} \rfloor \leq \frac{d_C - 1}{2}$ und $d(w, c_2) \leq \lfloor \frac{d_C - 1}{2} \rfloor \leq \frac{d_C - 1}{2}$. Setzt man dies in die Dreiecksungleichung ein, ist damit insgesamt $d_C \leq d_C - 1$ und der Widerspruch gezeigt. Wählt man einen Radius $> \lfloor \frac{d_C - 1}{2} \rfloor$ gilt für die Dreiecksungleichung mindestens die Gleichheit. Damit ist der Radius $\lfloor \frac{d_C - 1}{2} \rfloor$ maximal, wenn die Kugeln disjunkt sein sollen.

Liegt ein Wort nun innerhalb einer dieser Kugeln, ist es damit per Definition auch dem zur Kugel gehörenden Codewort am nächsten gelegen und folglich dadurch zu ersetzen. Durch diese Methode lassen sich bis zu $\lfloor \frac{d_C - 1}{2} \rfloor$ Fehler korrigieren. C heißt entsprechend $\lfloor \frac{d_C - 1}{2} \rfloor$-*Fehler-korrigierend.*

Treten mehr als $\lfloor \frac{d_C - 1}{2} \rfloor$ Fehler auf, kann einer von zwei Fällen auftreten. Entweder verschiebt sich das empfangene Wort in die Kugel eines anderen Codeworts, oder es liegt außerhalb von allen Kugeln. Im ersten Fall wird es einem falschen Codewort zugeordnet. Die Korrektur erzeugt somit einen Fehler. Der zweite Fall lässt sich nicht entscheiden. Es hängt von der jeweiligen Implementierung des Decoders ab, wie mit dem Problem umgegangen wird.

Definition 2.5. ([RM03, S. 213]) *Ein Blockcode mit ungeradem Mindestabstand* $d_C = 2e + 1$*, bei dem die Vereinigung paarweise disjunkter Kugeln* $\bigcup_{a \in C} K_e(a)$ *eine Überdeckung von* Σ^n *ist, heißt* perfekt.

Bei einem perfekten Code kann jedes empfangene Wort eindeutig einem Codewort zugeordnet werden. Damit kann der zweite der zuvor beschriebenen Fälle nicht auftreten, weil es keine Worte außerhalb der Kugeln gibt. Auf diese Eigenschaft werden wir später nochmal zurückkommen.

3 ISBN-Code

ISBN steht für *International Standard Book Number*. Es gibt verschiedene Versionen. Wir betrachten hier die bis 2006 übliche mit 10 Zeichen. Die Prüfsumme ergibt sich aus einer gewichteten Quersumme der ersten neun Ziffern.

Die *ISBN* ist eines der am häufigsten benutzten Beispiele für einen Prüfziffer-Code. Sie dient vor allem der Fehlererkennung. Eine Korrektur ist im Allgemeinen nicht vorgesehen. Dennoch ist es interessant, die einzelnen Möglichkeiten der *ISBN* zu erleben. Die ältere Version hat den Vorteil, dass sich die einzelnen Ziffern in der Berechnung leichter wiederfinden lassen, was eine Visualisierung begünstigt. Das ist der Grund für die Verwendung der veralteten Version.

3.1 Definition und Eigenschaften

Definition 3.1. ISBN-10 ([DIN ISO 2108])

$$C := \{(a_1, \ldots, a_{10}) | \sum_{i=1}^{9} i \cdot a_i \equiv a_{10} \mod 11\}; \qquad a_i \in \{0, 1, \ldots, 9, X\}$$

a_1 bis a_9 enthalten Informationen zu dem betreffenden Buch. a_{10} ist eine Prüfziffer, die wie in der Definition angegeben berechnet wird. Sollte für a_{10} der Wert 10 berechnet werden, wird dieser durch ein X dargestellt.

Korollar 3.2. ([SH08]) *C hat den Mindestabstand zwei.*

Beweis: ([SH08]) Es lassen sich Elemente finden, so dass der Hamming-Abstand zweier Codeworte zwei beträgt. Beispielsweise:

$$d((1, 1, \ldots, 1, 1),\ (2, 1, \ldots, 1, 2)) = 2$$

Der Mindestabstand beträgt also zwei oder weniger. Die Annahme, dass $d_C = 1$ ist, führt zum Widerspruch. Seien $a, b \in C$ mit $d(a, b) = 1$. Dann gibt es $i \cdot x \equiv 0 \mod 11$, wobei i die Gewichtung der abweichenden Ziffer und $x = |a_i - b_i|$ die Abweichung ist. Da per Definition $i > 0$ und $x \neq 0$ sind, muss gelten: $11 | i \cdot x$. Sowohl i als auch x sind per Definition beide kleiner als 11. Da 11 eine Primzahl ist, kann die Primfaktorzerlegung von $i \cdot x$ sie nicht als Primfaktor enthalten. Der Widerspruch ist gefunden und somit gezeigt, dass $d_C = 2$ ist. □

Der *ISBN-10*-Code ist $d_C - 1 = 1$ Fehler-erkennend und kann im Allgemeinen keine Fehler korrigieren, da $\lfloor \frac{d_C - 1}{2} \rfloor = 0$ ist. Maximal eine unbekannte Ziffer lässt sich mit der folgenden Kongruenz berechnen:

$$a_j \equiv \Big(- \sum_{\substack{i=1 \\ i \neq j}}^{10} i \cdot a_i \Big) \cdot j^{-1} \mod 11; \qquad j \in \{1, \ldots, 10\}$$

Satz 3.3. *Die Vertauschung zweier aufeinanderfolgender Ziffern einer* ISBN-*10 lässt sich anhand der Prüfziffer sicher erkennen.* ([SS06])

Beweis: ([SS06]) Sei $(c_1, \ldots, c_{10}) \in C$. Nach Definition gilt also:

$$1 \cdot c_1 + \ldots + 9 \cdot c_9 \equiv c_{10} \mod 11$$

Zu zeigen ist, dass sich durch Vertauschen zweier beliebiger aufeinanderfolgender Ziffern die Prüfziffer ändert. Es gelte $c_i \neq c_{i+1}$, da eine Vertauschung sonst sinnlos wäre. Die Kongruenz für die Prüfsumme mit vertauschten Ziffern lautet:

$$1 \cdot c_1 + \ldots + i \cdot c_{i+1} + (i+1) \cdot c_i + \ldots + 9 \cdot c_9 \equiv c_{10} \mod 11.$$

Durch ergänzen und abziehen von c_{i+1} lässt sich die Kongruenz umformen zu

$$1 \cdot c_1 + \ldots + 9 \cdot c_9 + c_i - c_{i+1} \equiv c_{10} \mod 11.$$

Nach Definition ist also

$$c_{10} + c_i - c_{i+1} \equiv c_{10} \mod 11.$$

Es bleibt zu zeigen, dass $c_i - c_{i+1} \not\equiv 0 \mod 11$ ist. c_i und c_{i+1} können Werte zwischen 0 und 9 annehmen und es gilt nach Voraussetzung $c_i \neq c_{i+1}$. Die Differenz kann also weder 0 noch ein Vielfaches von 11 ergeben. Die Prüfziffer ist in jedem Fall unterschiedlich und somit der Fehler sicher erkennbar. □

3.2 Visualisierung

Das primäre Ziel ist es, speziell die Korrekturmöglichkeiten der *ISBN* erfahrbar zu machen. Anders als die sonstigen Informationen etwa über den Verlag oder den Buchtitel hängen diese mit der verwendeten Prüfziffer zusammen, sind also im Zusammenhang mit Kodierungen von Interesse.

Wie bereits im letzten Abschnitt beschriebenen, erkennt man anhand der Prüfziffer, wenn eine Ziffer falsch ist oder zwei aufeinanderfolgende Ziffern vertauscht sind. Darüber hinaus lässt sich eine unbekannte Ziffer berechnen. Zu Beginn der Visualisierung werden diese Eigenschaften neben der Definition dem Benutzer mitgeteilt. Dabei wird die in der Definition enthaltene Kongruenz

$$\sum_{i=1}^{9} i \cdot a_i \equiv a_{10} \mod 11$$

zugunsten der Lesbarkeit abgewandelt. Zunächst bringt man sie auf die Form:

$$-1 \cdot a_{10} + \sum_{i=1}^{9} i \cdot a_i \equiv 0 \mod 11,$$

das heißt

$$10 \cdot a_{10} + \sum_{i=1}^{9} i \cdot a_i \equiv 0 \mod 11,$$

und damit

$$\sum_{i=1}^{10} i \cdot a_i \equiv 0 \mod 11.$$

Der Vorteil an dieser Schreibweise ist, dass der Benutzer direkt sehen kann, ob es sich um eine gültige *ISBN* handelt, ohne wirklich rechnen zu müssen. Wenn die gewichtete Quersumme kongruent $0 \mod 11$ ist, handelt es sich um eine gültige *ISBN*. Bei einem Ergebnis nicht kongruent $0 \mod 11$ ist die *ISBN* ungültig.

Damit der Benutzer erfahren kann, welche Korrekturmöglichkeiten die *ISBN* bietet, kann er eine solche frei eingeben. Die Inhalte der *ISBN* sind dabei nicht von Bedeutung. Es wird also nicht darauf eingegangen, welches Land oder welcher Verlag ein Buch mit einem bestimmten Titel herausgegeben hat. Ausgaben beschränken sich rein auf die Korrekturleistung.

Mit seinen Eingaben kann der Benutzer vier verschiedene Zustände erreichen. Den Ersten durch Eingabe einer korrekten *ISBN*. Der Einfachheit halber wird eine solche *ISBN* bereits beim Start vorgegeben. Abbildung 3.1 zeigt den Startbildschirm des Programms. Die gezeigte Nummer kann der Benutzer abändern, um die verschiedenen Eigenschaften zu testen.

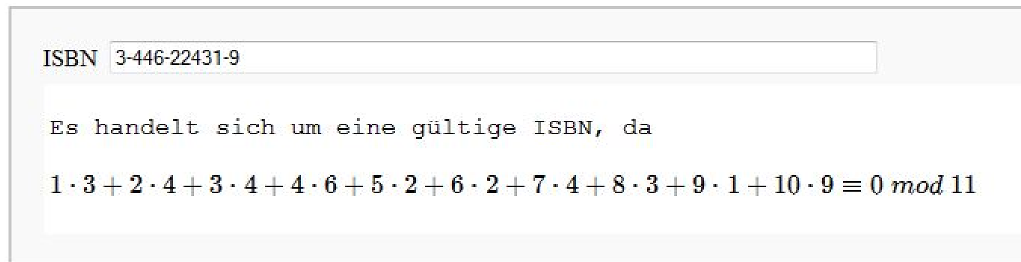

Abbildung 3.1: Gültige ISBN

Der zweite Zustand des Programms wird erreicht, wenn es sich bereits dem Format nach nicht um eine gültige *ISBN* handeln kann (Abbildung 3.2). Die Eingabe muss abzüglich der Bindestriche zehn Zeichen lang sein und darf nur aus zulässigen Zeichen bestehen. Dies sind Ziffern 0 bis 9, sowie die Sonderzeichen X und $*$. Groß- oder Kleinschreibung wird bei Buchstaben ignoriert. Die Fehlerausgabe erfolgt bewusst nicht differenziert, da ein zu starkes Eingehen auf Formalitäten von der eigentlichen Thematik, der Prüfziffer, ablenkt.

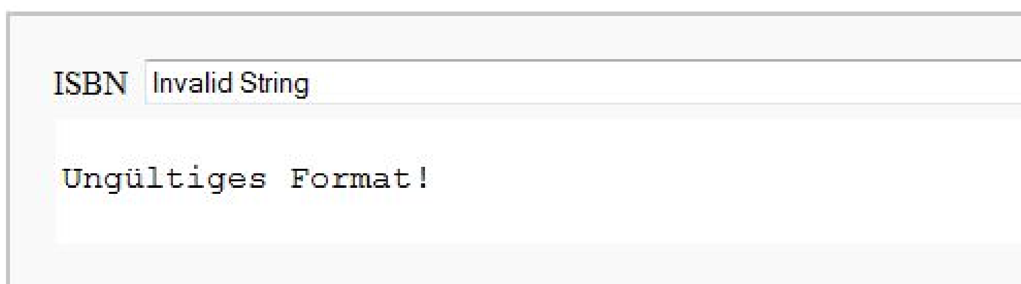

Abbildung 3.2: Ungültiges Format

Stimmt zwar das Format, nicht jedoch die enthaltenen Werte, erzeugt man dadurch den dritten Zustand des Programms. Farblich abgesetzt wird kenntlich gemacht, dass ein Fehler vorliegt (Abbildung 3.3). Abweichend von der Definition ergibt die gewichtete Quersumme der Ziffern einen Wert $x \not\equiv 0 \mod 11$. Die farbliche Kennzeichnung vereinfacht es dem Benutzer, den Fehler wahrzunehmen, da sich Zustand eins und drei auf den ersten Blick sehr ähnlich sehen.

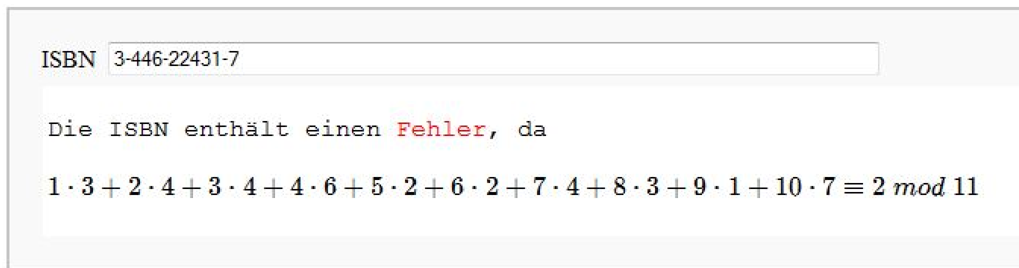

Abbildung 3.3: Prüfsumme fehlerhaft

Weil man bei der Erkennung von Fehlern nicht zwischen der Vertauschung zweier benachbarter Ziffern oder einer einzelnen falschen Ziffer differenzieren kann, werden beide Fehler zusammen dargestellt.

Mit den drei bisherigen Zuständen, *kein Fehler*, *Format-Fehler* und *inhaltlicher Fehler*, lässt sich die Funktionalität der *ISBN* bereits überprüfen. Als potentielle Service-Leistung wird die Berechnung einer unbekannten Ziffer hinzugefügt. Wie in der Bedienungsanleitung angegeben, darf genau eine Ziffer durch einen Stern ($*$) ersetzt werden. Zwei oder mehr Sterne gelten als Format-Fehler. Bestätigt man die Eingabe, berechnet das Programm die unbekannte Ziffer.

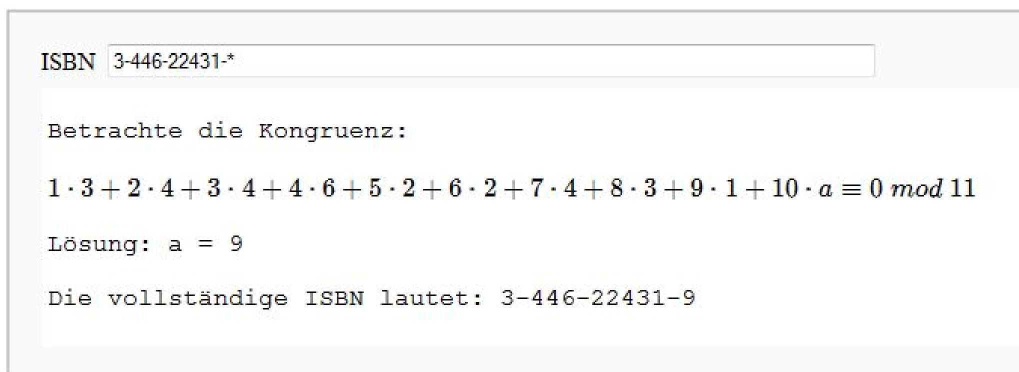

Abbildung 3.4: Berechnung einer unbekannten Stelle

In Abbildung 3.4 sieht man eine *ISBN*, deren letzte Ziffer durch einen Stern ($*$) ersetzt worden ist. Anschließend wird der Ansatz für die Berechnung sowie das Ergebnis ausgegeben.

3.3 Programmierung

In diesem Abschnitt wird zunächst der Aufbau der Benutzeroberfläche sowie der grobe Ablauf des Programms beschrieben. Anschließend erfolgt eine Erläuterung der einzelnen Funktionen anhand eines skizzierten Quellcodes.

Wie bereits auf den Screenshots zu sehen ist, besteht die Benutzeroberfläche aus einer Eingabezeile sowie einem Textfeld für die Ausgabe von Ergebnissen. Die Eingabe wird erst nach Drücken der *ENTER*-Taste ausgewertet.

Listing 3.1 zeigt den Programm-Code in skizzierter Form. Das heißt, nicht gut druckbare Inhalte sind durch Kommentare ersetzt. Ursprünglich im Code enthaltene Kommentare fehlen.

Die interaktive Funktion `validate_isbn` (Listing 3.1, Zeile 50) steuert den Ablauf des Programms. Wenn es sich um einen korrekt formatierten String handelt, erfolgt anschließend die inhaltliche Auswertung. Enthält er einen Stern, wird die so als unbekannt gekennzeichnete Ziffer berechnet. Rechenansatz und Lösung werden ausgegeben.

Ist kein Stern enthalten, muss die *ISBN* inhaltlich bewertet werden. Die Zahlenwerte werden anhand der Definition auf Richtigkeit hin überprüft. Der jeweils zutreffende Fall wird vom Programm ausgegeben.

Das Programm durchläuft bei jeder Eingabe die gleiche Kette von Fallunterscheidungen. Betrachten wir im Folgenden die einzelnen Funktionen, welche im Zuge dessen aufgerufen werden können.

Die Funktionen `check_format` (Listing 3.1, Zeile 1) nimmt als Parameter einen *ISBN*-String auf. Die einzelnen Eigenschaften werden schrittweise abgeprüft. Sobald ein Fehler festgestellt wird, bricht die Überprüfung ab und gibt `false` zurück. Der String muss ohne Bindestriche exakt 10 Zeichen lang sein. Zulässige Zeichen sind: `'0'`, ... , `'9'`,`'x'`,`'X'`,`'*'`. Er darf höchstens einen Stern ($*$) enthalten. Nur wenn kein Fehler aufgetreten ist, wird `true` zurückgegeben.

`printDef(isbn, result)` (Listing 3.1, Zeile 14) dient lediglich der Ausgabe. Durch String-Operationen wird aus den Parametern ein HTML-Output nach dem folgenden Muster erzeugt:

$$\underbrace{3-446-22431-9, \qquad result}$$

$$\rightsquigarrow$$

$$1\cdot3+2\cdot4+3\cdot4+4\cdot6+5\cdot2+6\cdot2+7\cdot4+8\cdot3+9\cdot1+10\cdot9 \equiv result \mod 11$$

Dabei wird auch die Funktion `getNumber` (Listing 3.1, Zeile 18) verwendet. Mit einfachen Fallunterscheidungen liefert sie für die Darstellung der Kongruenz geeignete Zeichen. Ein X wird dabei als 10 und ein Stern als a dargestellt. Die übrigen Zeichen werden durch ihren Integer-Wert ersetzt. Es ist hierbei zu beachten, dass keinerlei *exception-handling* stattfindet. Durch den Kontext ist allerdings gewährleistet, dass es sich um gültige Zeichen handeln muss.

Um die Prüfsumme aus einem *ISBN*-String zu berechnen, gibt es die Funktion `getCheckSum` (Listing 3.1, Zeile 26). Sie berechnet die gewichtete Quersumme des übergebenen Strings und gibt das Ergebnis mod 11 zurück. Hierbei wird auf die Funktion `getNumber` zurückgegriffen. Dadurch wird der Fall abgedeckt, dass die Prüfsumme als X dargestellt wird. Ein reiner typecast *char* $\rightarrow$ *int* schlägt in diesem Fall sonst fehl. Auch hier ist über den Kontext gesichert, dass das passende Format vorliegt und sich die Prüfsumme somit berechnen lässt.

Die Funktion `getUnknownValue` dient zur Berechnung einer unbekannten Ziffer. Sei j der Index des Sterns im von Bindestrichen befreiten *ISBN*-String. Es wird die gewichtete Quersumme der bekannten Ziffern berechnet und damit die bereits beschriebene Kongruenz gelöst:

$$a_j \equiv \Big(-\sum_{\substack{i=1\\ i\neq j}}^{10} i\cdot a_i\Big)\cdot j^{-1} \mod 11; \qquad j\in\{1,\ldots,10\}$$

Sollte das Ergebnis dem Wert 10 entsprechen, erfolgt die Rückgabe von X anstelle des Zahlenwertes.

Listing 3.1: ISBN (Skizzierte Fassung)

```
1  def check_format(isbn):
2      isbn = isbn.replace('-','')
3      if len(isbn) <> 10:
4          return false
5      for i in range(9):
6          if isbn[i] not in # gültige Zeichen:
7              return false
8      if isbn[9] not in # gültige Zeichen:
9          return false
10     if isbn.count('*') > 1:
11         return false
12     return true
13
14 def printDef (isbn, result):
15     # Schreibt die ISBN-Definition
16     # mit dem übergebenen Ergebnis
17
18 def getNumber(ziffer):
19     if ziffer == 'X' or ziffer == 'x':
20         return 10
21     elif ziffer == '*':
22         return 'a'
23     else:
24         return int(ziffer)
25
26 def getCheckSum(isbn):
27     isbn_number = isbn.replace('-','')
28     result = 0
29     for i in range(10):
30         result += (i+1)*getNumber(isbn_number[i])
31     return result % 11
32
33
```

```
34 def getUnknownValue (isbn):
35     isbn = isbn.replace('-','')
36 
37     index = isbn.find('*')
38     result = 0
39     numbers = range(10)
40     numbers.remove(index)
41     for i in numbers:
42         result += (i+1) * int(isbn[i])
43     value = Mod(-result,11) / (index+1)
44     if value == 10:
45         return 'X'
46     else:
47         return value
48 
49 @interact
50 def validate_isbn(isbn=input_box(...)):
51 
52     if check_format(isbn) == False:
53         html('Ungültiges Format!')
54     else:
55         if isbn.count('*') == 1:
56             value = getUnknownValue (isbn)
57             html('Betrachte die Kongruenz: \n')
58             printDef (isbn,'0')
59             html('\nLösung: a = ' + str(value) )
60             # Ausgabe der vollständigen ISBN
61         else:
62             if getCheckSum(isbn) == 0:
63                 # Ausgabe: Gültige ISBN (mit Begründung)
64             else:
65                 # Ausgabe: Ungültige ISBN (mit Begründung)
```

4 Lineare Codes

In diesem Kapitel geht es um die Darstellung linearer Codes. Konkret um den *Hamming-Code* (H_7), den *Binary Goley Code* (G_{23}), sowie die jeweiligen Erweiterungen H_8 und G_{24}. Ein Grund, dass gerade diese Codes ausgewählt werden, ist unter anderem ihr hoher Bekanntheitsgrad.

Die Grundidee ist ein Leistungsvergleich von H_7 und G_{23}. Es bietet sich an, die jeweiligen Erweiterungen H_8 und G_{24} in die Betrachtungen miteinzubeziehen. So können Lernende zusätzlich erfahren, wie eine Übertragung durch Erweiterung des verwendeten Codes beeinflusst wird.

Zunächst definieren wir lineare Codes und beschreiben deren Eigenschaften. Beispiele ergeben sich im Zuge der anschließenden Visualisierung und Programmierung, so dass sie vorher entfallen.

4.1 Definitionen und Eigenschaften

Definition 4.1. ([SH08]) *Ein linearer Code ist ein Untervektorraum von $\mathbb{F}_q^n$, wobei $\mathbb{F}_q$ ein endlicher Körper mit $q = p^a$ (p ist eine Primzahl) Elementen ist.*

Ein linearer Code mit Länge n und Dimension k wird notiert als $[n,k]_q - Code$ beziehungsweise als $[n,k,d]_q - Code$, wenn der Mindestabstand $d_C = d$ ebenfalls angegeben werden soll.

Korollar 4.2. ([SH08]) *Ein $[n,k,d]_q$-Code ist genau dann perfekt (siehe Definition 2.5), wenn seine Parameter die folgende Gleichung erfüllen:*

$$q^k \sum_{i=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{n}{i} \cdot (q-1)^i = q^n$$

Beweis: ([SH08]) Wir haben bereits vorher gezeigt, dass

$$\bigcup_{c \in C} K_{\lfloor \frac{d-1}{2} \rfloor}(c)$$

disjunkt ist. Als Menge von Elementen aus $\mathbb{F}_q^n$ ist auch

$$\bigcup_{c \in C} K_{\lfloor \frac{d-1}{2} \rfloor}(c) \subset \mathbb{F}_q^n.$$

Zu zeigen bleibt also, dass beide Mengen genau dann gleich mächtig sind, wenn sie die Gleichung erfüllen. Da die Vereinigung disjunkt ist, gilt

$$|\bigcup_{c \in C} K_{\lfloor \frac{d-1}{2} \rfloor}(c)| = |C| \cdot |K_{\lfloor \frac{d-1}{2} \rfloor}(c)|.$$

Wir wissen, dass $|C| = q^k$ und damit gilt:

$$|C| \cdot |K_{\lfloor \frac{d-1}{2} \rfloor}(c)| = q^k \cdot |K_{\lfloor \frac{d-1}{2} \rfloor}(c)|.$$

Jede Kugel enthält alle Elemente, die sich an bis zu $\lfloor \frac{d_C-1}{2} \rfloor$ Stellen vom zugehörigen Codewort unterscheiden. Dafür gibt es $\binom{n}{i}$ mögliche Kombinationen von abweichenden Stellen, wobei i die Anzahl der abweichenden Stellen ist. Für jede Stelle gibt es $q-1$ abweichende Möglichkeiten, also $\binom{n}{i} \cdot (q-1)^i$ Elemente, die sich an i Stellen von c unterscheiden.

Insgesamt ist also

$$|K_{\lfloor\frac{d-1}{2}\rfloor}(c)| = \sum_{i=0}^{\lfloor\frac{d-1}{2}\rfloor} \binom{n}{i} \cdot (q-1)^i$$

und damit dann

$$|\bigcup_{c\in C} K_{\lfloor\frac{d-1}{2}\rfloor}(c)| = q^k \cdot \sum_{i=0}^{\lfloor\frac{d-1}{2}\rfloor} \binom{n}{i} \cdot (q-1)^i.$$

Auf der anderen Seite ist $|\mathbb{F}_q^n| = q^n$. Es gilt also

$$|\bigcup_{c\in C} K_{\lfloor\frac{d-1}{2}\rfloor}(c)| = |\mathbb{F}_q^n| \quad \text{genau dann, wenn} \quad q^k \sum_{i=0}^{\lfloor\frac{d-1}{2}\rfloor} \binom{n}{i} \cdot (q-1)^i = q^n,$$

womit die Behauptung gezeigt wäre. □

Da ein Datenkanal in der heutigen Zeit normalerweise binär arbeitet, wählen wir für die weiteren Betrachtungen als Kanal-Alphabet $\Sigma = \mathbb{F}_2$.

Notation:

$\mathbb{F}_q^n$	:	Menge der Zeilenvektoren
$\mathbb{F}_q^{1\times r}$	:	Menge der Spaltenvektoren
$\mathbb{F}_q^{n\times r}$	:	Menge der $n \times r$ Matrix

Für die folgenden Definitionen sei $C \subseteq \mathbb{F}_2^n$ ein linearer Code und $k = \log_2 |C|$.

Definition 4.3. ([SH08]) *Eine* Erzeugermatrix *von C ist eine $k \times n$-Matrix G, so dass $C = \{a \cdot G | a \in \mathbb{F}_2^k\} = \mathbb{F}_2^k \cdot G$*

Definition 4.4. ([SH08]) *Eine* Kontrollmatrix *von C ist eine $(n-k) \times n$-Matrix H, so dass $C = \{a \in \mathbb{F}_2^n | H \cdot a^t = 0\}$*

Definition 4.5. ([RM03, S. 223]) *Das* Gewicht *$\omega(c)$ eines Codewortes $c \in C$ ist die Anzahl der von 0 verschiedenen Komponenten von c.*

Definition 4.6. ([SH08]) *Für einen linearen Code $C \subset \mathbb{F}_2^n$ heißt*

$$\omega_C(x) = \sum_{c\in C} x^{\omega(c)}$$

der Gewichtszähler.

Satz 4.7. ([SH08]) *Für eine abelsche Gruppe $\mathcal{A}$ gilt $d(x+t, y+t) = d(x, y)$ für alle $x, y, t \in \mathcal{A}^n$.*

Beweis: ([SH08]) Nach Definition 2.2 ist $d(x+t, y+t) = \#\{i|x_i + t \neq y_i + t\}$. Weiter ist $\#\{i|x_i + t \neq y_i + t\} = \#\{i|x_i \neq y_i\}$ und $\#\{i|x_i \neq y_i\} = d(x, y)$. Insgesamt ist also $d(x+t, y+t) = d(x, y)$. □

Satz 4.8. ([SH08]) *Ist $\mathcal{A}$ eine abelsche Gruppe und $\mathcal{C} \subset \mathcal{A}^n$ eine Untergruppe, so gilt*

$$d_C = \omega_C, mit\ \omega_C := \min_{\substack{x \in C \\ x \neq 0}}(\omega(x)).$$

Beweis: ([SH08]) Wir betrachten $\{d(x, y) \mid x, y \in \mathcal{C}, x \neq y\}$. Nach Satz 4.7 ist

$$\{d(x, y) \mid x, y \in \mathcal{C}, x \neq y\} = \{d(x-y, y-y) \mid x, y \in \mathcal{C}, x \neq y\}$$

und damit auch

$$\{d(x, y) \mid x, y \in \mathcal{C}, x \neq y\} = \{d(x-y, 0) \mid x, y \in \mathcal{C}, x \neq y\}.$$

$d(x-y, 0)$ entspricht genau der Anzahl von 0 verschiedener Komponenten von $x - y$, also dem Gewicht (Siehe Definition 4.5) der Differenz und damit ist

$$\{d(x, y) \mid x, y \in \mathcal{C}, x \neq y\} = \{\omega(z) \mid z \in \mathcal{C}, z \neq 0\}.$$

Sind die Mengen gleich, so haben sie auch das gleiche Minimum. Damit gilt also, dass

$$d_C = \min_{\substack{x,y \in \mathcal{C} \\ x \neq y}} d(x, y) = \min_{\substack{z \in \mathcal{C} \\ z \neq 0}} \omega(z) = \omega_{\mathcal{C}}$$

ist. □

Definition 4.9. ([SH08]) *Sei $C \subset \mathbb{F}_2^n$ ein Code. Dann heißt*

$$\hat{C} := \{(c, a) \in \mathbb{F}_2^{n+1},\ c \in C,\ a \in \mathbb{F}_2 | \sum_{i=1}^{n+1} c_i \equiv 0 \mod 2\} \subset \mathbb{F}_2^{n+1}$$

Erweiterung von C.

Satz 4.10. *([SH08]) Für geraden Minimalabstand* d_C *ist* $d_{\hat{C}} = d_C$ *und für ungeraden Minimalabstand* d_C *ist* $d_{\hat{C}} = d_C + 1$.

Beweis: ([SH08]) Zu einem $c \in C$ kann entweder eine 0 oder 1 hinzugefügt werden, damit $(c, 0)$ oder $(c, 1)$ ein Element von $\hat{C}$ ist. Der Minimalabstand von $\hat{C}$ kann dadurch nicht kleiner als d_C und auch nicht größer als $d_C + 1$ werden.

Betrachten wir zunächst den Fall, dass der Minimalabstand von C gerade ist. Sei $c \in C$ mit $\omega(c) = d_C$. Dann ist $\omega((c, 0)) \equiv 0 \mod 2$ und $\omega((c, 1)) \equiv 1 \mod 2$. In einem erweiterten Code haben jedoch alle Codewörter per Definition die Eigenschaft $\omega(\hat{c}) \equiv 0 \mod 2$, so dass nur $(c, 0)$ in $\hat{C}$ liegen kann. Die Erweiterung eines Codes mit geradem Minimalabstand hat folglich den gleichen Minimalabstand wie der ursprüngliche Code, also gilt $d_{\hat{C}} = d_C$.

Betrachten wir nun den Fall, dass der Minimalabstand von C ungerade ist. Sei $c \in C$ mit $\omega(c) = d_C$. Dann ist $\omega((c, 0)) \equiv 1 \mod 2$ und $\omega((c, 1)) \equiv 0 \mod 2$. Also kann nur $(c, 1)$ in $\hat{C}$ liegen. Damit wird das Gewicht und somit der Minimalabstand um 1 erhöht, so dass gilt $d_{\hat{C}} = d_C + 1$. □

4.1.1 Hamming-Code (H_7)

Der [7,4] Hamming-Code, auch H_7 genannt, wurde 1950 von Richard Hamming erstmals publiziert. Er ist nur einer von vielen *Hamming-Codes*, da dieser Begriff eine ganze Klasse von Codes bezeichnet. Dennoch ist er *der* Hamming-Code. Bedingt durch seine überschaubare Komplexität wird er in der Lehre gerne als Beispiel für einen linearen Code benutzt.

Definition 4.11. ([RH50]) $H_7 := \{x \in \mathbb{F}_2^4 | x \cdot G\}$ *ist der [7,4]-Hamming-Code,*

wobei $G := \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$ *eine Erzeugermatrix ist* ([Sage]).

Satz 4.12. ([RH50]) H_7 *hat den Mindestabstand* $d_{H_7} = 3$.

Beweis. ([SH08])

$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$ ist eine Kontrollmatrix von H_7 ([Sage]).

Die Spaltenvektoren a_j von A sind paarweise linear unabhängig. Wir finden beispielsweise ein Wort $w = (1, 1, 0, 0, 0, 0, 1)$, das ein Gewicht von 3 hat und für das gilt: $A \cdot (1, 1, 0, 0, 0, 0, 1)^t = 0$. Da A eine Kontrollmatrix von H_7 ist, ist $w \in H_7$. Damit ist $d_{H_7} \leq 3$. Bleibt zu zeigen, dass $d_{H_7} \not< 3$ sein kann.

Angenommen es ist $d_{H_7} = 1$. Dann gibt es ein $c \in H_7$ mit $\omega(c) = 1$, so dass $A \cdot c^t = 0$. Es sind aber alle Spaltenvektoren von A ungleich dem Nullvektor. Damit ist $d_{H_7} \neq 1$.

Angenommen es ist $d_{H_7} = 2$. Dann gibt es ein $c \in H_7$ mit $\omega(c) = 2$, so dass $A \cdot c^t = 0$. Der Nullvektor lässt sich jedoch nicht durch eine Summe zweier Spalten von A erzeugen. Damit ist $d_{H_7} \neq 2$. □

Aus $d_{H_7} = 3$ folgt, dass H_7 ein $d_{H_7} - 1 = 2$ Fehler erkennender, und $\lfloor \frac{d_{H_7}-1}{2} \rfloor = 1$ Fehler-korrigierender Code ist.

Nach Korollar 4.2 ist H_7 perfekt, da die Gleichung

$$\begin{aligned} 2^4 \cdot \sum_{i=0}^{1} \binom{7}{i} \cdot (2-1)^i &= 2^4 \cdot \left(\binom{7}{0} + \binom{7}{1} \right) \\ &= 2^4 \cdot (1+7) \\ &= 2^4 \cdot 2^3 \\ &= 2^7 \end{aligned}$$

erfüllt ist.

4.1.2 Erweiterter Hamming-Code (H_8)

Die Erweiterung von H_7 folgt direkt aus Definition 4.9.

Definition 4.13. $H_8 := \{x \in \mathbb{F}_2^4 | x \cdot G\}$ *ist der [8,4]-Hamming-Code,*

wobei $G := \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$ *eine Erzeugermatrix ist* ([Sage]).

Nach Satz 4.10 ist $d_{H_8} = d_{H_7} + 1 = 4$, da d_{H_7} ungerade.

Aus $d_{H_8} = 4$ folgt, dass H_8 ein $d_{H_8} - 1 = 3$ Fehler erkennender, und $\lfloor \frac{d_{H_8}-1}{2} \rfloor = 1$ Fehler-korrigierender Code ist. Außerdem ist er nicht perfekt, da gemäß Definition 2.5 nur Codes mit ungeradem Mindestabstand perfekt sind.

4.1.3 Binärer Golay-Code (G_{23})

Golay hat diesen Code erstmals 1949 vorgestellt. Dafür benötigte er weniger als eine Seite. Dies ist erstaunlich, wenn man sieht, wie vielschichtig und komplex der Code ist. Neben vielen theoretischen Betrachtungen unter den verschiedensten Gesichtspunkten gibt es auch berühmte praktische Anwendungen für G_{23}. Um 1980 hat die NASA bei den ersten beiden *Voyager*-Missionen den Code eingesetzt, um Bilder von Jupiter und Saturn zurück zur Erde zu schicken.

Definition 4.14. ([MG49]) $G_{23} := \{x \in \mathbb{F}_2^{12} | x \cdot G\}$, *wobei* $G :=$

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

eine Erzeugermatrix ist ([Sage]).

Für die Bestimmung des Mindestabstands betrachten wir den Gewichtszähler von G_{23}, den wir mit Hilfe von *Sage* ([Sage]) ermitteln:

$$\omega_{23}(x) = 1 + 253x^7 + 506x^8 + 1288x^{11} + 1288x^{12} + 506x^{15} + 253x^{16} + x^{23}$$

Nach Satz 4.8 ist somit $d_{G_{23}} = 7$. Also ist G_{23} ein $[23, 12, 7]-Code$, $d_{G_{23}} - 1 = 6$ Fehler-erkennend und $\lfloor \frac{d_{G_{23}}-1}{2} \rfloor = 3$ Fehler-korrigierend.

Die Gleichung

$$
\begin{aligned}
2^{12} \cdot \sum_{i=0}^{3} \binom{23}{i} \cdot (2-1)^i &= 2^{12} \left(\binom{23}{0} + \binom{23}{1} + \binom{23}{2} + \binom{23}{3} \right) \\
&= 2^{12} \cdot (1 + 23 + 253 + 1771) \\
&= 2^{12} \cdot 2048 \\
&= 2^{12} \cdot 2^{11} \\
&= 2^{23}
\end{aligned}
$$

ist erfüllt und somit G_{23} nach Korollar 4.2 perfekt.

4.1.4 Erweiterter Binärer Golay-Code (G_{24})

Die Erweiterung von G_{23} folgt direkt aus Definition 4.9.

Definition 4.15. $G_{24} := \{x \in \mathbb{F}_2^{12} | x \cdot G\}$ *ist ein* $[24, 12, 8] - Code$*, wobei* $G :=$

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

eine Erzeugermatrix ist ([Sage]).

G_{24} ist $d_{G_{24}} - 1 = 7$ Fehler-erkennend und $\lfloor \frac{d_{G_{24}}-1}{2} \rfloor = 3$ Fehler-korrigierend und als Code mit geradem Mindestabstand nicht perfekt.

4.2 Visualisierung

Die Codes werden in der Mathematik normalerweise rein theoretisch behandelt. Ohne Unterstützung eines Computers lassen sich Kodierungen auch nur selten praktisch anwenden. Für einen Leistungsvergleich ist jedoch gerade diese Praxis wichtig. Es muss praktisch erfahrbar werden, was eine Kodierung leistet.

Genutzt werden die Codes beispielsweise bei der Übertragung von Bilddaten. Also ist eine naheliegende Lösung, den Ablauf einer solchen Übertragung schrittweise darzustellen, und so die Auswirkungen der Kodierung erkennbar zu machen. Dieser Ansatz birgt jedoch verschiedene Probleme.

Ändert man an einem Bild nur wenige Pixel, ist dies mit bloßem Auge nur schwer festzustellen. Wenn das Display des Benutzers verschmutzt ist oder Pixelfehler aufweist, kann man nicht eindeutig entscheiden, ob der Fehler nur durch die simulierte Übertragung oder durch äußere Einflüsse entstanden ist. Die Wahrnehmung des Menschen kann hier nicht differenziert genug arbeiten. Hebt man die Häufigkeit von Fehlern an, kommt es zu einer diffusen Wahrnehmung. Man erkennt wesentlich leichter, dass Fehler vorhanden sind, jedoch ist ein einzelner Fehler und seine Korrektur kaum noch nachzuvollziehen.

Auch die technische Umsetzung wirft Probleme auf. Die Entwicklungsumgebung stellt sich bei näheren Untersuchungen als nicht ausreichend performant heraus, um Datenmengen in der Größenordnung zu verarbeiten. Kodierungen finden normalerweise weitgehend auf Hardwareebene statt. Hier reden wir jedoch von einer vielschichtigen Software-Architektur, die einen deutlich spürbaren Verwaltungsapparat beinhaltet. Man erreicht schnell Laufzeiten im höheren Minuten-Bereich, was nicht als akzeptabel angesehen werden kann.

Eine Lösung ist die Auslagerung bestimmter Programmteile in eine Blackbox. Die Blackbox lässt sich von *Sage* ansteuern und somit ist die Anbindung an das Webinterface geschaffen. Vorteil ist eine wesentlich höhere Performance. Die Nachteile dieses Ansatzes sind jedoch immens.

Der Entwicklungsaufwand ist um ein Vielfaches höher - man bedenke, dass dann die mathematischen Aspekte neu zu lösen sind - und der Benutzer verliert den direkten Zugriff auf den Programmcode. Genau das ist aber eines der wesentlichen Kriterien, die das Programm erfüllen muss. Ohne direkten Zugriff auf den Programmcode ist ein Hinterfragen und Untersuchen der gezeigten Effekte faktisch unmöglich. Also ist der Ansatz zu verwerfen.

Damit sind die wesentlichen Probleme der Entwicklung charakterisiert. Um sie zu lösen, werden ein paar Änderungen vorgenommen. Pixel sind offenbar zu klein, um als Einheit für die Darstellung dienen zu können. Wir werden folglich Pixel symbolisch zeigen, und sie durch große Punkte ersetzen. Damit ist die Grundlage geschaffen, Fehler und Korrekturen erkennbar zu machen.

Die Punkte sind nicht mehr dafür geeignet, ein Bild im herkömmlichen Sinne darzustellen. Darum erstellen wir eine rechteckige Anordnung von fünf Punkten in der Höhe und sechs Punkten in der Breite als Ersatz. Das Feld lässt sich mehrfach nebeneinander anzeigen, was für die Darstellung eines Ablaufs von entscheidender Wichtigkeit ist.

Ein weiterer positiver Effekt dieser Maßnahme ist, dass wesentlich weniger Daten verarbeitet werden müssen. Es gibt per Definition weit weniger Punkte als Pixel. Abhängig vom Datenformat benötigt ein Punkt genau so viele Daten wie ein Pixel. Geht man davon aus, dass man ein Bild von 320×240 Pixeln durch 30 Punkte ersetzt, reduziert man entsprechend die Datenmenge grob um den Faktor 2500, was sich sehr positiv auf die Laufzeit auswirkt.

Betrachten wir die Bilddaten genauer. Sie bestehen grob gesagt aus Farben und Positionen für die einzelnen Pixel. Je nach Bildformat sind die Informationen anders angelegt. Verfälschungen können in der Regel in beiden Teilen auftreten. Veränderungen an der Position der Punkte haben zur Folge, dass eine eindeutige Zuordnung für den Betrachter verloren geht. Bei einer abschnittsweisen Darstellung der Übertragung ist für den Benutzer nicht erkennbar, welcher Punkt korrigiert worden ist, wenn sich Farbe und Position ändern.

Darüber hinaus bedeuten zufällige Verschiebungen eine fehlende Kontrollierbarkeit des Layouts. Der Platz auf dem Display ist begrenzt, so dass es zu Überlappungen oder unerwünschten Scrollbalken kommen kann. Eingrenzen kann man die Verschiebungen nicht, da man so die Daten verfälscht. Und gerade um die soll es bei der Simulation gehen. Wie man sieht, kann man bei der Verfälschung von Positionsinformationen viel verlieren, aber effektiv nichts gewinnen. Wir verzichten also darauf und beschränken uns auf die Verfälschung der Farbwerte.

Farben kann der Mensch recht gut unterscheiden. Vor allem, wenn sie direkt vergleichbar nebeneinander zu sehen sind. Die fünf Zeilen von Punkten innerhalb unseres Bildes bekommen aus diesem Grund spaltenweise einheitliche Farben, so dass auch geringe Abweichungen leicht erkennbar sind.

Für die konkrete Auswahl der Farben ist es notwendig, den Aufbau von Farbwerten innerhalb von Grafiken zu kennen. Eine von mehreren Varianten, Farbwerte zu speichern, besteht aus einem Zahlentupel mit drei Werten für die Farbanteile von *Rot*, *Grün* und *Blau*, kurz auch *RGB-Wert* genannt. Jeder der drei Werte belegt 8 Bit Speicher, also benötigt man insgesamt 24 Bit pro Bildpunkt.

Wir nutzen für eine leichtere Erkennbarkeit von Fehlern eine besondere Eigenschaft der *RGB-Werte* aus. Graustufen besitzen drei exakt gleiche Farbanteile. Die Höhe der Farbanteile bestimmt die jeweilige Helligkeit. Es ist unwahrscheinlich, dass eine Verfälschung bei allen drei Farbanteilen genau eines Bildpunktes in gleicher Weise auftritt. Konsequenz daraus ist, dass ein Punkt unkontrolliert bunt wird, sobald es zu einer Verfälschung kommt. Grau und bunt lassen sich sehr leicht unterscheiden, und damit auch die Fehler identifizieren.

Damit sind die Beschaffenheiten des Bildes ausreichend charakterisiert. Offen ist noch die Darstellung der Übertragung. Eine Übertragung beginnt, wie bereits in einem vorherigen Kapitel erwähnt, mit einer ursprünglichen Nachricht, sozusagen dem Urbild. In diesem Fall besteht es aus den 5×6 Punkten, die mit spaltenweise unterschiedlichen Graustufen dargestellt werden (Abb. 4.1).

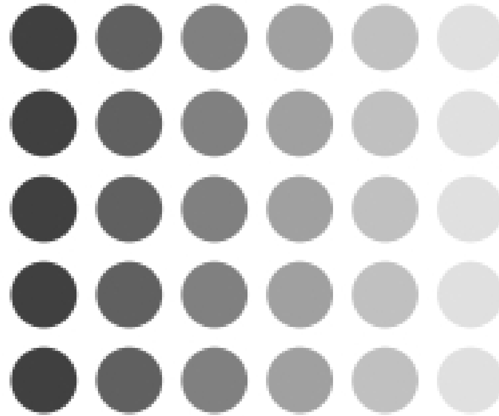

Abbildung 4.1: Urbild

Insgesamt gibt es vier verschiedene Zustände, in denen sich das Bild befindet:

1. Urbild
2. Codiertes Bild
3. Empfangenes Bild
4. Decodiertes Bild

Das codierte Bild wird dadurch hergestellt, dass die einzelnen Datenblöcke des Urbilds codiert werden. So vergrößert sich abhängig von der gewählten Kodierung die Datenmenge. Für eine direkte Darstellung ist es nicht mehr geeignet. Die Code-Anteile sind nicht als Bild darstellbar, da sie nicht dem Dateiformat, beziehungsweise in diesem Fall der Anzahl der Farbwerte entsprechen. Reduziert man das Bild um die Zusatzinformationen der Kodierung, entsteht wieder das Urbild. Da eine doppelte Darstellung des Urbilds nicht sinnvoll ist, existiert das codierte Bild nur für interne Verarbeitungszwecke.

Das *empfangene Bild* wird aus dem *codierten Bild* generiert. Dazu werden Verfälschungen an den Bilddaten erzeugt, was die Übertragung durch einen Kanal simuliert. Für eine Darstellung der Fehler entnimmt man dem Bild die entsprechenden Daten und zeigt die enthaltenen Farbwerte an. Es entsteht ein buntes Bild, das die Fehler gut sichtbar macht.

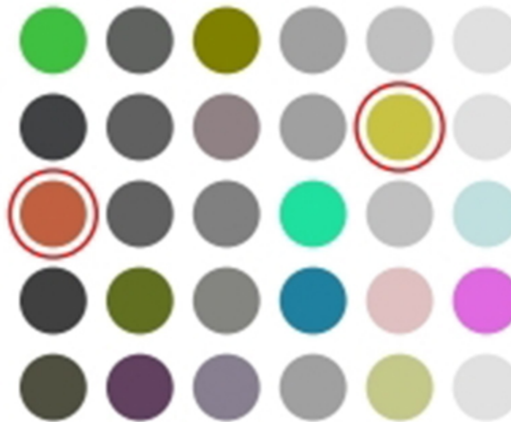

Abbildung 4.2: Empfangenes Bild

Abbildung 4.2 zeigt ein solches Bild. Enthalten die Daten für einen Punkt mehr Fehler als der Code erkennen kann, wird dies durch einen roten Kreis kenntlich gemacht. Auf diese Weise wird eine Differenzierung der Fehler bezüglich ihrer Erkennbarkeit erreicht. Alle nicht markierten Fehler werden vom Code erkannt.

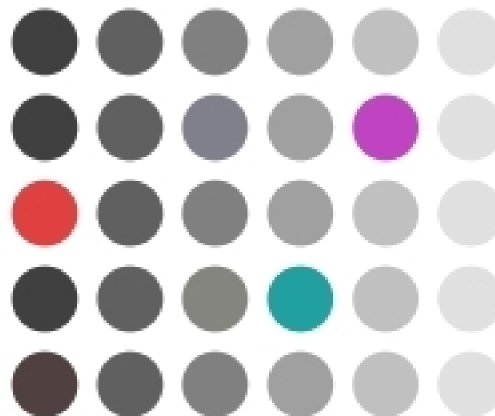

Abbildung 4.3: Decodiertes Bild

Das decodierte Bild zeigt die Fehler an genau den Stellen, wo man sie bereits anhand der Kreise aus dem empfangenen Bild heraus erahnen konnte. Wie man jedoch am dritten verfälschten Punkt sieht, enthalten auch Punkte ohne Markierung Fehler, die nicht korrigiert werden können. Dadurch wird deutlich, dass man mehr Fehler erkennen als korrigieren kann.

Damit sind die vier Zustände gezeigt. Als Schnittstelle zum Benutzer gibt es vier Knöpfe, je einen pro Code, sowie einen Schieberegler für die Fehlerrate. Statistiken werden für jede Simulation in einer Tabelle angezeigt. Die Statistiken beinhalten Informationen über die Anzahl fehlerhafter Bits, der als fehlerhaft erkannten Bits und die nach der Korrektur noch vorhandenen fehlerhaften Bits. Jeder Wert ist mit einem Prozentwert versehen, was eine Deutung erleichtert.

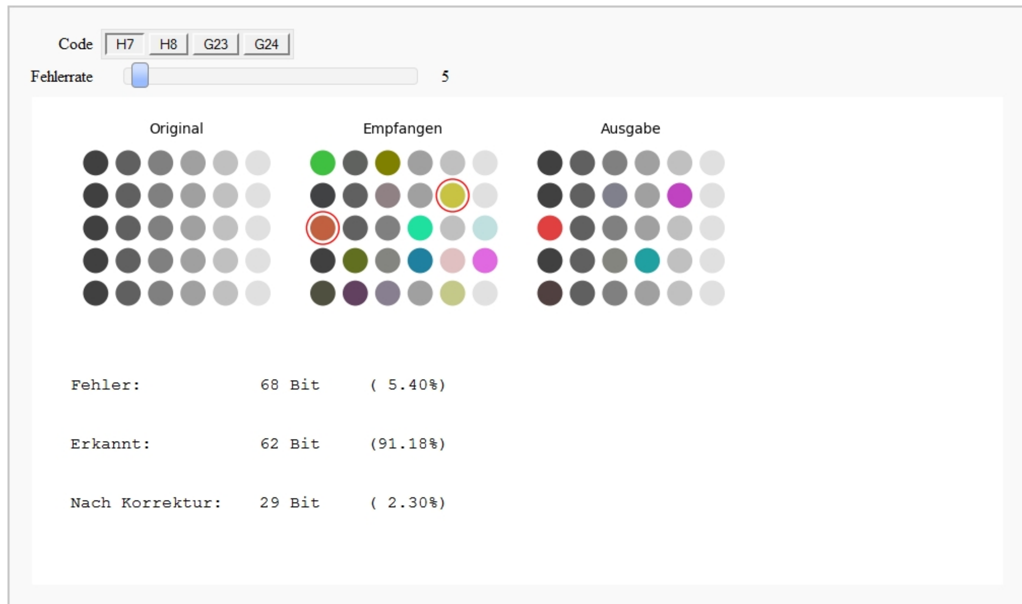

Abbildung 4.4: Übersicht

Realistische Fehlerraten sind Werte unterhalb von 5%. Erkannte Fehler können durch eine Wiederholung der Übertragung behoben werden. Jeder rote Kreis im Bild bedeutet also mit hoher Wahrscheinlichkeit einen Datenverlust.

Durch Erweiterung der hier behandelten Codes verbessert sich nicht die Korrekturleistung, sondern nur die Fehlererkennung. Bei einer Fehlerrate von 10% kann man zwischen H_7 und H_8 die Unterschiede gut erkennen. H_8 erkennt ungefähr 10% mehr Fehler als H_7. Ähnliches kann man für G_{23} und G_{24} feststellen. Jedoch ist hier eine Fehlerrate von 15% geeigneter. Dies macht auch die Unterschiede zwischen Hamming-Code und Goley-Code deutlich.

Die Korrektur von Fehlern funktioniert nur unterhalb von 5% Fehlerrate. Höhere Fehlerraten führen schnell zu Korrekturfehlern. Erkennen kann man diese daran, dass nach der Korrektur mehr Bits fehlerhaft sind als vorher.

4.3 Programmierung

Realisiert man das Programm objektorientiert, sieht das Hauptprogramm sehr schlicht aus. Die wichtigen Details passieren im Hintergrund und sind konzeptbedingt nicht direkt einzusehen. Wie bereits erwähnt, soll es für einen Benutzer möglich sein, den Ablauf des Programms im Detail nachzuvollziehen. Ohne die Objektorientierung und damit verbundene Polymorphie zu kennen, wird ihm dies jedoch nur schwerlich gelingen. Deshalb ist das Programm in prozeduralem Stil geschrieben. Ein der Polymorphie ähnlicher Effekt wird durch Einsatz von Referenzen erreicht, um den Code nicht unnötig in die Länge zu ziehen.

Die Beschreibung lässt sich in die drei wesentlichen Bereiche *Datenstrukturen*, *Programmablauf* und *Darstellung* untergliedern. Beginnen wir mit den Datenstrukturen. Zeilenangaben beziehen sich jeweils auf Listing 4.1.

Die ersten 67 Zeilen des Programms dienen vor allem dem Aufbau der Datenstrukturen. Zu Beginn werden die Codes definiert. Jeder Code bekommt eine eigene Liste von Vektoren. Außerdem wird je eine Erzeugermatrix als globale Variable angelegt. *Sage* berechnet die Erzeugermatrix offenbar bei jedem Zugriff auf die Methode `gen_mat` neu, so dass es sich im Hinblick auf die Laufzeit lohnt, jeweils eine fertige Version zu speichern.

Die Vektorlisten sind zunächst leer initialisiert, werden aber dann mit zwei *for*-Schleifen (Zeilen 18-24 und 26-31) befüllt. Ziel ist es, alle Codewörter in den Listen abzulegen. Dafür benötigt man also alle Elemente von $\mathbb{F}_2^k$, wobei k den Wert 4 für Hamming Codes und 12 für Golay Codes hat. Elemente von $\mathbb{F}_2^k$ bestehen aus der Binärdarstellung der Werte 0 bis 2^k.

In den Zeilen 19 bis 22 werden solche Vektoren aufgebaut. i nimmt Werte von 0 bis $2^{12} - 1$, also genau den Wertebereich von 12 Bit, der Dimension der Binären Goley Codes, an. *Sage* bietet für Binärwerte leider keine Möglichkeit, sie durch Ergänzung führender Nullen auf eine fest vorgegebene Länge zu formatieren.

Ohne führende Nullen enthält ein Vektor jedoch nicht genug Daten. Wir rechnen die Dezimalwerte also eigenständig in einen Binärwert vorgeschriebener Länge um. Dazu wird der Wert von i bitweise um z Stellen nach rechts verschoben. So kommt immer das jeweils nächste Bit der Zahl auf die Einerstelle zu liegen. Bitweises UND mit dem Wert 1 gibt also Auskunft darüber, ob die jeweilige Ziffer eine 0 oder 1 ist. Die Ergebnisse werden als String zwischengespeichert aus dem direkt anschließend ein Vektor erzeugt wird.

Dieser Vektor lässt sich mit der Erzeugermatrix multiplizieren, so dass ein Codewort entsteht. Die Hamming Codes werden in den Zeilen 26 bis 31 analog initialisiert. Die Codewörter werden in den Listen gespeichert, so dass man insgesamt die Bildmengen der Kodierungen in Listenform erhält.

Als nächstes werden die eigentlichen Bilddaten angelegt. Die Zeilen 38 bis 47 definieren jeweils mehrdimensionale Arrays für die einzelnen Kodierungen sowie passende Matrizen. Anstatt also die Daten, wie es bei einer wirklichen Übertragung vorkommt, in einem Bitstream zu bearbeiten, werden hier fest definierte Datenblöcke verwendet. Eine Matrix entspricht jeweils genau einem Farbwert mit 24 Bit. Diese lassen sich in diesem konkreten Fall in 2×12 oder 6×4 Matrizen schreiben, so dass eine Zeile genau einem Block entspricht.

Im Original werden die Daten als *RGB*-String gespeichert, wie es unter anderem in HTML üblich ist. Das heißt nach einem einleitenden # folgen insgesamt sechs hexadezimale Ziffern für einen Farbwert. Die Werte werden in Abhängigkeit von der jeweiligen Spalte des Feldes initialisiert, um den Farbverlauf zu gestalten.

Der *RGB*-String wird zeichenweise in einen Binärstring umgewandelt (Zeilen 54-56), wobei das führende # übersprungen wird. Anschließend werden die Matrizen mit den Binärdaten befüllt. *Sage* indiziert die einzelnen Zellen einer Matrix fortlaufend, so dass Zeilensprünge automatisch passieren. Das Feld der Original-Matrizen stellt das im letzten Unterkapitel angesprochene *Urbild* dar. Durch Multiplikation der Matrizen mit den jeweiligen Erzeugermatrizen der Codes ergeben sich entsprechend die codierten Matrizen, also das *codierte Bild*.

Die bisherigen Daten werden im weiteren Verlauf des Programms nicht mehr geändert. Daher erfolgen die Berechnungen nur einmal zu Beginn. Besonders im Zusammenhang mit G_{24} spürt man den Rechenaufwand erheblich.

Kommen wir nun zur Beschreibung des Ablaufs. Dazu beginnen wir mit der interaktiven Funktion `visualisierung` in Zeile 151. Der Selector `codeID` wird ausgewertet und eine Reihe von Variablen entsprechend der gewählten Kodierung mit Werten versehen (Zeilen 156-183). Dabei werden Referenzen auf die zuvor definierten Datenstrukturen so gesetzt, dass das Programm sich im weiteren Verlauf ähnlich verhält, wie es bei einer rein objektorientierten Programmierung und damit verbundener Polymorphie der Fall wäre.

Das Programm durchläuft das Feld der Punkte (Zeile 185-208). Für jeden Punkt wird zunächst eine verfälschte Matrix von der Funktion `corrupt` (Zeile 74) aus der codierten Matrix sowie der eingestellten Fehlerrate erzeugt. Die Fehlerrate liegt, wie für den binären symmetrischen Kanal definiert, zwischen 0$\%$ und 49$\%$.

Dadurch, dass die Matrizen Elemente aus $\mathbb{F}_2$ haben, lassen sich die Bits durch Addition von 1 an den jeweiligen Stellen toggeln. `corrupt` addiert zur übergebenen Matrix eine Matrix gleicher Größe, die mit der am Schieberegler eingestellten Wahrscheinlichkeit aus Einsen, sonst Nullen, besteht. Die von *Sage* zur Verfügung gestellte Funktion für die Erstellung von Zufallsmatrizen liefert leider keine zufriedenstellenden Ergebnisse, so dass dafür die eigene Funktion `getRandomBit` (Zeile 68) benutzt wird.

In den Zeilen 191 bis 198 wird die verfälschte Matrix mit dem Original verglichen. Das ist natürlich in der Realität nicht möglich, aber hier für Anschauungszwecke praktisch. Abweichende Bits werden gezählt. Die Auswertung geschieht wortweise, so dass anhand des Mindestabstandes entschieden werden kann, ob der Code den Fehler von sich aus erkennt. Worte, bei denen die Fehleranzahl $\geq d_C$ ist, gelten ohne weitere Differenzierung pauschal als nicht erkennbar. Gezählt werden die Fehler, welche als *erkennbar* klassifiziert werden. Auf nichterkennbare Fehler wird lediglich im Zuge der Darstellung eingegangen.

Anschließend wird versucht, die verursachten Fehler zu beseitigen. Dazu wird die Funktion `correct` unter anderen mit der verfälschten Matrix als Parameter aufgerufen. Im ersten Schritt werden also für die empfangenen Worte ähnliche Codeworte gesucht. Die Zeilenvektoren der Matrix, also die empfangenen Worte, werden jeweils mit der Kontrollmatrix multipliziert. Ist das Ergebnis der Nullvektor, handelt es sich bereits um ein Codewort. Für Ergebnisse ungleich Null wird die Funktion `getCorrectedElement` aufgerufen.

Diese Funktion durchläuft die Liste aller Codewörter und berechnet den jeweiligen Mindestabstand zum empfangenen Wort. Ist dieser $\leq \lfloor \frac{d-1}{2} \rfloor$ wird das aktuelle Codewort zurückgegeben. Für perfekte Codes ist diese Abbruchbedingung bereits ausreichend, da es garantiert zu jedem Wort ein ähnliches Codewort gibt. H_8 und G_{24} sind jedoch nicht perfekt, so dass dafür zusätzliche Maßnahmen notwendig sind. Die Implementierung sieht hier vor, dass das letzte Wort mit dem kleinsten Abstand als zugehöriges Codewort genommen wird, wenn sich nach einem kompletten Durchlauf der Liste kein passendes Codewort ergeben hat.

Korrigierte Elemente werden zunächst lokal in einer Liste zwischengespeichert, für den weiteren Gebrauch vor der Rückgabe aber in eine Matrix umgewandelt. Diese Matrix wird mit dem Original verglichen und so die nicht- oder falschkorrigierten Fehler gezählt.

Die Fehlerkorrektur benötigt mit Abstand die meisten Ressourcen. Für jedes Wort wird eine lineare Suche durchgeführt, die für Golay-Codes im Durchschnitt nach 2048 Vergleichen ein Ergebnis liefert. Es ist möglich hier eine künstliche Beschleunigung vorzunehmen, indem man Vergleiche zunächst mit den Codeworten durchführt, die aus den fest vorgegebenen Daten entstehen.

Auf die Beschleunigung wird verzichtet, weil die Rechenzeit auch ein Teil der Simulation ist. Der Benutzer bekommt einen Eindruck davon, dass für komplexere Codes mehr Rechenzeit benötigt wird.

Als letzter Punkt bleibt noch die Darstellung. Das Original-Bild sowie die Beschriftungen der Bilder allgemein lassen sich vorberechnet als Grundgerüst festlegen. Im Quellcode findet man diese Struktur unter `baseGraphix`.

Abgesehen davon befindet sich die komplette Darstellung innerhalb der Funktion `visualisierung`. Die einzelnen Elemente folgen jeweils direkt im Anschluss an die Berechnungen der Matrizen sowie deren Auswertungen. Da die Matrizen noch codiert sind, ist eine direkte Verwendung als Farbwert nicht möglich.

Die Funktion `matrixToColor` stellt, wie der Name schon sagt, aus einer Matrix einen Farbwert her. Dabei spielt die Validität der Daten keine Rolle. Abhängig von der verwendeten Kodierung werden aus den Matrizen genau die Bits herausgelöst, die den Einheitsvektoren in den jeweiligen Erzeugermatrizen (Siehe Definitionen) entsprechen. Je 4 Bits werden dabei wieder zu einer hexadezimalen Ziffer. Code-Bits werden dabei ignoriert. Die Aussagekraft der Visualisierung leidet darunter nicht. Die Funktion kann sowohl für verfälschte als auch für korrigierte Matrizen verwendet werden.

Der rote Kreis, als Signal für eine nicht mehr zwingend erkennbare Fehleranzahl, wird an den entsprechenden Koordinaten hinzugefügt, wenn die Fehleranzahl für ein Wort zu hoch wird. Die Verwendung der Koordinaten des Punktes lässt keine Differenzierung der einzelnen Worte innerhalb eines Farbwertes zu.

Alle ermittelten Statistiken werden abschließend in einer Tabelle ausgegeben. Beim Anteil der erkannten Fehler ist zu beachten, dass die Anzahl der vorhandenen Fehler auch Null sein kann. Die Division ist nicht möglich und muss daher abgefangen werden.

Listing 4.1: Bildübertragung (Skizzierte Fassung)

```
rows    = 5
columns = 6
H7  = HammingCode(3,GF(2))
H8  = H7.extended_code()
G23 = BinaryGolayCode()
G24 = ExtendedBinaryGolayCode()

vecListH7  = []
vecListH8  = []
vecListG23 = []
vecListG24 = []

genMatH7   =  H7.gen_mat()
genMatH8   =  H8.gen_mat()
genMatG23 = G23.gen_mat()
genMatG24 = G24.gen_mat()

for i in range(4096):
    binString = "".join( [str(( i >> z ) & 1 )
        for z in range(11, -1, -1)] )
    v = vector(GF(2),[binString[i]
        for i in range(12)])
    vecListG23.append ( v * genMatG23 )
    vecListG24.append ( v * genMatG24 )

for i in range(16):
    binString = "".join( [str(( i >> z ) & 1 )
        for z in range(3, -1, -1)] )
    v = vector(GF(2),[binString[j] for j in range(4)])
    vecListH7.append ( v * genMatH7 )
    vecListH8.append ( v * genMatH8 )


```

```
baseGraphix  = text("Original" , ( 2.5, 5), rgbcolor='#000000')
baseGraphix += text("Empfangen", ( 9.5, 5), rgbcolor='#000000')
baseGraphix += text("Ausgabe"  , (16.5, 5), rgbcolor='#000000')

original      = [[[] for i in range(columns)] for j in range(rows)]
originalMat04 = [[[] for i in range(columns)] for j in range(rows)]
originalMat12 = [[[] for i in range(columns)] for j in range(rows)]

M04 = MatrixSpace ( GF(2), 6, 4 )
encodedMatH7  = [[[] for i in range(columns)] for j in range(rows)]
encodedMatH8  = [[[] for i in range(columns)] for j in range(rows)]
M12 = MatrixSpace ( GF(2), 2,12 )
encodedMatG23 = [[[] for i in range(columns)] for j in range(rows)]
encodedMatG24 = [[[] for i in range(columns)] for j in range(rows)]

for i in range (rows):
    for j in range(columns):
        original[i][j] = '#%02x%02x%02x'
            % ( (j+2)*32, (j+2)*32, (j+2)*32 )
        binString = ''
        for c in original[i][j][1:]:
            binString += "".join( [str( (int(c,16) >> z) & 1 )
                for z in range(3, -1, -1)] )
        originalMat04[i][j] = M04([binString[k]
            for k in range(len(binString))])
        originalMat12[i][j] = M12([binString[k]
            for k in range(len(binString))])
        baseGraphix +=
            point([j,i], rgbcolor = original[i][j], pointsize=250)

        encodedMatH7 [i][j] = originalMat04[i][j] * genMatH7
        encodedMatH8 [i][j] = originalMat04[i][j] * genMatH8
        encodedMatG23[i][j] = originalMat12[i][j] * genMatG23
        encodedMatG24[i][j] = originalMat12[i][j] * genMatG24
```

```
68 def getRandomBit( errorRate ):
69     if randint(1,100) <= errorRate:
70         return 1
71     else:
72         return 0
73
74 def corrupt ( A, errorRate ):
75     M = MatrixSpace ( GF(2), len(A.rows()), len(A.columns()) )
76     size = len(A.rows()) * len(A.columns())
77     B = M([getRandomBit(errorRate) for i in range(size)])
78     return A+B
79
80 def matrixToColor ( matrix, codeID ):
81
82     if codeID == 'H7' or codeID == 'H8':
83         colorString = '#'
84         for v in matrix.rows():
85             tempString = ''
86             for j in [0,1,2,4]:
87                 tempString += str(v[j])
88             colorString += hex(int(tempString,2))[2:]
89         return colorString
90
91     if codeID == 'G23' or codeID == 'G24':
92         colorString = '#'
93         for v in matrix.rows():
94             for j in range(3):
95                 tempString = ''
96                 for k in range(4):
97                     tempString += str(v[4*j + k])
98                 colorString += hex(int(tempString,2))[2:]
99         return colorString
100
101     return ''
```

```
102 def getCorrectedElement( v, vecList, codeMinDist ):
103     minVec  = v
104     minDist = 25
105     for w in vecList:
106         dist = hamming_weight(v-w)
107         if dist <= floor((codeMinDist-1)/2):
108             return w
109         if dist <= minDist:
110             minVec  = w
111             minDist = dist
112 
113     return minVec
114 
115 def correct( A, codeID, minDist, checkMat, checkRowsCount ):
116 
117     lenARows = 6
118     lenACols = 0
119 
120     if codeID == 'H7':
121         vecList = vecListH7
122         lenARows = 6
123         lenACols = 7
124     if codeID == 'H8':
125         vecList = vecListH8
126         lenARows = 6
127         lenACols = 8
128     if codeID == 'G23':
129         vecList = vecListG23
130         lenARows = 2
131         lenACols = 23
132     if codeID == 'G24':
133         vecList = vecListG24
134         lenARows = 2
135         lenACols = 24
```

```
136     tempList = []
137 
138     for v in A.rows():
139         if checkMat*v==vector([0 for j in range(checkRowsCount)]):
140             tempList.append ( v )
141         else:
142             w = getCorrectedElement( v, vecList, minDist )
143             tempList.append ( w )
144     M = MatrixSpace ( GF(2), lenARows, lenACols )
145     B = M(tempList)
146 
147     return B
148 
149 
150 @interact
151 def visualisierung( # selector, slider ):
152     g = baseGraphix
153     countRawErrors       = 0
154     countIDedErrors      = 0
155     countFinalErrors     = 0
156     if codeID == 'H7':
157         C = H7
158         encodedMat = encodedMatH7
159         checkMat = H7.check_mat()
160         checkRowsCount = 3
161         minDist  = 3
162         numberOfTotalBits = 1260
163     if codeID == 'H8':
164         C = H8
165         encodedMat = encodedMatH8
166         checkMat = H8.check_mat()
167         checkRowsCount = len(checkMat.rows())
168         minDist  = 4
169         numberOfTotalBits = 1440
```

```
170     if codeID == 'G23':
171         C = G23
172         encodedMat = encodedMatG23
173         checkMat = G23.check_mat()
174         checkRowsCount = 11
175         minDist  = 7
176         numberOfTotalBits = 1380
177     if codeID == 'G24':
178         C = G24
179         encodedMat = encodedMatG24
180         checkMat = G24.check_mat()
181         checkRowsCount = 12
182         minDist  = 8
183         numberOfTotalBits = 1440
184
185     for i in range (rows):
186         for j in range(columns):
187             corruptedMat = corrupt ( encodedMat[i][j], errorRate )
188             color = matrixToColor ( corruptedMat, codeID )
189             g += point([j+1*columns+1,i], #color , size )
190
191             errorMat = encodedMat[i][j] - corruptedMat
192             for word in errorMat.rows():
193                 localRawError = hamming_weight ( word )
194                 countRawErrors += localRawError
195                 if localRawError >= minDist:
196                     g += circle([j+1*columns+1,i], #... )
197                 if localRawError < minDist:
198                     countIDedErrors += localRawError
199
200             correctedMat = correct ( corruptedMat, #... )
201             color = matrixToColor ( correctedMat, codeID )
202             g += point([j+2*columns+2,i], #color , size )
203
```

```
204                 errorMat = encodedMat[i][j] − correctedMat
205                 for word in errorMat.rows():
206                     countFinalErrors += hamming_weight ( word )
207
208         g.show( #... )
209
210         rawErrorRate    = (countRawErrors    / numberOfTotalBits ) * 100
211         IDedErrorRate   = 0
212         if countRawErrors != 0:
213             IDedErrorRate = ( countIDedErrors / countRawErrors ) * 100
214         finalErrorRate = (countFinalErrors / numberOfTotalBits ) * 100
215
216         html('<table border="0">')
217         html('<tr>')
218         html('    <td> Fehler: </td>')
219         html('    <td>' + str(countRawErrors) + ' Bit </td>')
220         html('    <td> ('+ "%5.2f%%" % rawErrorRate +')</td>')
221         html('</tr>')
222         html('<tr>')
223         html('    <td> Erkannt: </td>')
224         html('    <td>' + str(countIDedErrors) + ' Bit </td>')
225         html('    <td> ('+ "%5.2f%%" % IDedErrorRate +')</td>')
226         html('</tr>')
227         html('<tr>')
228         html('    <td> Nach Korrektur: </td>')
229         html('    <td>' + str(countFinalErrors) + ' Bit</td>')
230         html('    <td> ('+ "%5.2f%%" % finalErrorRate +')</td>')
231         html('</tr>')
232         html('</table>')
```

5 Fazit und Ausblick

Visualisierungen zu ausgewählten Themen der Kodierungstheorie besitzen ein breites Spektrum an beteiligten Wissensgebieten. Zu nennen sind an erster Stelle Mathematik, Informatik und Didaktik, wobei jeder dieser Bereiche sich weiter aufgliedert. Wie bereits in der Einleitung erwähnt, erfordert das Verständnis der Kodierungstheorie eine breitgefächerte mathematische Bildung. Zahlentheorie, Stochastik und Algebra sind nur eine Auswahl der involvierten Themenbereiche. Auch die Informatik lässt sich nicht auf einen Bereich einschränken. Von Operationen auf Bit-Ebene über Aufbau und Verhalten von Algorithmen bis hin zu graphischen Darstellungen in einem Webbrowser ist alles enthalten.

Die Didaktik koordiniert den zielgerichteten Einsatz der breiten Wissensbasis. Durch die verschiedenen Visualisierungen lässt sich erkennen, dass es dafür nicht den einzig und allein richtigen Weg gibt. Die *ISBN* ist im Grunde eine genau so abstrakte Nummer wie ein beliebiges Codewort von H_7. Visualisiert werden beide jedoch komplett unterschiedlich.

Die *ISBN* wird rein als Nummer dargestellt. Enthaltene Informationen über das Land, den Verlag und die fortlaufende Nummer, oder gar der Titel des Buches selbst werden bewusst verschwiegen. Es scheint paradox, dass eine abstraktere Sichtweise hier zu einer besseren Erkennbarkeit führt, wo doch eigentlich der Alltagsbezug das allgemeine Credo ist. Hier geht es jedoch eben nicht um das, was der Alltag zeigt, sondern um das, was er zeigen könnte. Blendet man den Alltag auf beiden Seiten aus, wird folglich die Differenz deutlicher erkennbar.

Der Vergleich der linearen Codes geht einen völlig anderen Weg. Weniger die zugrundeliegende Theorie, sondern vielmehr ein Erleben der praktischen Umsetzung ist hier von Bedeutung. Die Mathematik beschreibt die Tätigkeiten rein formal, ohne sie jedoch wirklich auszuführen, oder überhaupt ausführen zu können. Bestenfalls einzelne Worte werden als Beispiel (de-)codiert und damit das Konzept klar gemacht.

Wirkliche Praxiserfahrung lässt sich so jedoch nicht gewinnen. Darum ist es wichtig, mit der Visualisierung genau diese Erfahrungen möglich zu machen. Nur so bekommt man eine Antwort auf die Fragen: Wie viele Daten kommen noch fehlerfrei an, wenn im Kanal 5% der Bits verfälscht und ein Hamming-Code benutzt wird? Ist das Ergebnis mit dem Golay-Code besser?

So vielfältig die beteiligten Themen sind, so zahlreich sind denkbare Fortsetzungen und Erweiterungen. Eine interessante Idee ist beispielsweise eine Übersicht über die Leistungsfähigkeit von Codes im Laufe der Jahrzehnte. Hamming hat seinen Code 1950 veröffentlicht und eingesetzt. Der Golay-Code wurde um 1980 verwendet. Eine Verbesserung lässt sich erkennen. Doch wie leistungsfähig sind Codes auf dem aktuellen Stand der Technik?

Im Bereich der Informatik stellen sich Fragen vor allem bezüglich der Algorithmen und Datenstrukturen. Welche Leistungssteigerungen erreicht man durch geschicktere Such-Algorithmen im Zusammenhang mit der Decodierung? Je komplexer ein System ist, desto mehr Funktionen bietet es, die man nicht braucht. Rechtfertigt also der Performance-Gewinn den Verzicht auf Funktionalität bei der Verwendung selbst definierter Datentypen?

6 Literatur

[RM03] Matthes, Roland: *Algebra, Kryptologie und Kodierungstheorie*, Carl Hanser Verlag München Wien, 2003

[SS06] Alexander Souza, Universität Freiburg, Angelika Steger, ETH Zürich, Artikel zum 13. Algorithmus, Informatikjahr 2006

[SH08] Shuichi Hayashida, Universität Siegen, Skript zur Vorlesung "Kodierungstheorie", Stand 2008

[RH50] Richard W. Hamming: Error Detection and Error Correction Codes. The Bell System Technical Journal, Vol. XXVI 2, 1950, Seite 147-160.

[MG49] Golay, Marcel J. E.: "Notes on Digital Coding", Proc. IRE 37, Seite 657, 1949.

[CS48] C. E. Shannon, „A mathematical theory of communication“, Bell System Technical Journal, Vol. 27, Seiten 379-423 und 623-656, Juli und Oktober, 1948.

[Sage] William Stein, *Sage* Computer Algebra System, University of Washington, 2005

Ich versichere, dass ich die schriftliche Hausarbeit - einschließlich beigefügter Zeichnungen, Kartenskizzen und Darstellungen - selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

7 Anlagen

CD mit folgenden Inhalten:

- ISBN - Visualisierung als *Sage*-Worksheet (.sws)
- ISBN - Visualisierung als Textdatei (.txt)
- Bildübertragung - als *Sage*-Worksheet (.sws)
- Bildübertragung - als Textdatei (.txt)
- Dieses Dokument im PDF-Format (.pdf)