



DIPLOMARBEIT

Fehlerkorrektur im Slack Space unter Verwendung von Reed-Solomon Codes

Ausgeführt am Institut für

Diskrete Mathematik und Geometrie
der Technischen Universität Wien

unter der Anleitung von

ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerhard Dorfer

durch

Adrian van Oyen
Michael-Bernhard-Gasse 4/1
1120 Wien

Datum

Unterschrift (Student)

Abstract

Kurzfassung deutsch

Mit den gängigen Methoden Daten zu speichern kann normalerweise nicht der gesamte zur Verfügung stehende Speicherplatz genutzt werden. Ungenutzte Bereiche die dennoch als beschrieben markiert sind, der so genannte Slack Space, kann und soll verwendet werden um Daten zu speichern. Doch damit ergeben sich Probleme, die den Einsatz von Fehlerkorrigierenden Codes notwendig machen. Reed-Solomon (RS) Codes scheinen sich hier besonders gut zu eignen, vor allem da Dank des Guruswami-Sudan (GS) Decodieralgorithmus eine Möglichkeit gegeben ist jenseits der Minimaldistanz zu korrigieren.

Daher werden im Folgenden zuerst zwei Codierungs- und die dazu gehörenden Decodierungsmethoden (der Berlekamp-Massey (BM) und der GS Algorithmus) für RS Codes untersucht. Die Vorteile der systematischen Codierung mit Generatorpolynom sowie die Decodierung mittels GS Algorithmus werden herausgearbeitet und ein Algorithmus beschrieben um mit Generatorpolynom und GS Algorithmus speichereffizient arbeiten zu können. Vor allem die Parameter des GS Algorithmus werden sehr genau analysiert, da dieser auf Grund seines größeren Korrekturradius eine Reduktion des Speicherbedarfs ermöglicht. Mittels gängiger Methoden der Codierungstheorie, wie z.B. der Verkettung von Codes, können Daten beliebiger Größe gespeichert werden sofern genug Platz vorhanden ist. Analysiert und verglichen werden zum Abschluss drei verschiedene Varianten um Daten im Slack Space zu speichern.

Abstract english

When using conventional methods for saving data on electronic devices it is usually not possible to occupy the entire allocated disk space. Empty clusters that are nevertheless marked as allocated, the so called slack space, can and should be used for saving data. But using those clusters leads to problems which makes the use of error-correcting codes necessary. Reed-Solomon (RS) codes seem to fit very well, especially because of the Guruswami-Sudan (GS) decoding algorithm which offers a very good possibility for decoding beyond the minimum distance.

In this diploma thesis we will first discuss two different methods of coding and their corresponding decoding algorithms (the Berlekamp-Massey (BM) and the GS algorithm) regarding RS codes. The advantages of systematic coding with generator polynomials and decoding using the GS algorithm will be reviewed and a memory-efficient algorithm using the generator polynomials and the GS algorithm will be described. The parameters of the GS decoding algorithm will be analysed in detail, because of its ability to often correct beyond the $\frac{d}{2}$ limit, which allows memory-efficient saving of data. Using conventional methods of coding theory (e.g. code concatenation), data of any size can be saved as long as the slack space is big enough. In conclusion we will analyse and compare three different methods for saving data in slack space.

Vorwort

Als ich im Jahr 2011 zum ersten Mal auf das Thema meiner Diplomarbeit aufmerksam wurde, ahnte ich zwar bereits, dass es nicht leicht sein wird diese in ein bis zwei Semestern zu schreiben, doch das Thema klang zu spannend und interessant als dass ich es wieder fallen lassen wollte. Die Verbindung von mathematischem Wissen aus der Codierungstheorie mit der praktischen Anwendung in der Informatik hatte einen zu großen Reiz. Das Finalisieren dieser Diplomarbeit hat schlussendlich deutlich länger gebraucht als ich ursprünglich gedacht und geplant hatte und ich war auf die Hilfe vieler Personen angewiesen.

Ich möchte mich daher für die Geduld und Unterstützung bei all jenen Menschen bedanken, die mich auf diesem Weg begleitet haben und mir halfen diese Diplomarbeit zu beenden. Ein großer Dank gilt natürlich meinem Betreuer Gerhard Dorfer (E104 - Institut für Diskrete Mathematik und Geometrie, TU Wien), der mir ausführlich in vielen Treffen Feedback zu meiner Arbeit gegeben hat und sich viel Zeit für fachliche Diskussionen genommen hat. Ebenso wichtig waren für mich die Rückmeldungen von Martin Mulazzani (SBA Research Wien) um das theoretische Mathematik-Wissen etwas näher an die Praxis heran zu führen. Außerdem gilt mein Dank Teresa Matiassek, die sich nicht nur Zeit genommen hat meine Arbeit zu lektorieren, sondern mich auch motiviert hat meinen Zeitplan einzuhalten.

Zu guter Letzt möchte ich mich noch bei meiner Freundin Elisabeth Neumann bedanken. Ohne ihre Unterstützung in all den Jahren weiß ich nicht, wie ich diese Arbeit hätte beenden können.

Danke.

Wien, November 2014 Adrian van Oyen

Inhaltsverzeichnis

1	Einleitung	6
2	Grundlagen	8
2.1	Begriffsbestimmungen	8
2.2	Mathematische Definitionen und Grundlagen	8
2.3	Slack Space	15
3	Vorbereitung GS Algorithmus	17
3.1	Ordnung bivariater Polynome	18
3.2	Nullstellen bivariater Polynome	21
3.3	Interpolation bivariater Polynome	23
3.3.1	Kötters Interpolationsalgorithmus	25
3.4	Faktorisierung bivariater Polynome	32
3.4.1	Roth-Ruckensteins Faktorisierungsalgorithmus	34
4	Reed-Solomon Codes	41
4.1	Codieren durch Auswertung bei α^i	41
4.2	Codieren durch Multiplikation mit Generatorpolynom	43
5	Decodierung von RS Codes	46
5.1	Decodierung mittels GS Algorithmus	46
5.1.1	Decodierung von Auslöschungen	47
5.2	Decodierung mittels BM Algorithmus	48
5.2.1	Forneys Algorithmus	57
5.2.2	Decodierung von Auslöschungen	59
5.3	Chien Suche	62
6	Eigenschaften	63
6.1	Eigenschaften GS Algorithmus	63
6.2	Eigenschaften BM Algorithmus	72
7	Modifizierung von Codes	72
7.1	Systematische Codierung	73
7.2	Verkürzen von Codes	74
7.3	Verketten von RS Codes	77
8	Fehlerkorrektur im Slack Space	79
8.1	Variante 1	80
8.2	Variante 2	85
8.3	Variante 3	94
8.4	Schlussfolgerung	96
9	Quellenverzeichnis	100
A	Additionstabelle für GF(16)	101

B	Matlab Programme	102
B.1	Grundrechnungsarten in Galoisfeldern	102
B.1.1	Logarithmen- und Potenzlisten	102
B.1.2	Addition und Subtraktion	102
B.1.3	Multiplikation	103
B.1.4	Division	103
B.1.5	Potenzieren	104
B.2	Arbeiten mit Arrays	104
B.2.1	Multiplikation von 2 Polynomen	104
B.2.2	Multiplikation (Skalar*(bivariatem) Polynom)	104
B.2.3	Addition von 2 Matrizen	105
B.3	Codierungsalgorithmen	105
B.3.1	Codieren durch Auswertung bei α^i	105
B.3.2	Codieren durch Multiplikation mit Generatorpolynom	106
B.4	Guruswami-Sudan Algorithmus	107
B.4.1	Kötters Interpolationsalgorithmus	107
B.4.2	Roth-Ruckensteins Faktorisierungsalgorithmus	109
B.5	Berlekamp-Massey Algorithmus	111

1 Einleitung

Die Idee zu dieser Diplomarbeit kam von Edgar Weippl (Research Director der SBA Research in Wien). Der Wunsch war es, Daten wie z.B. Passwörter, Bilder, Programme oder sonstige Dateien, effizient im Slack Space (eine genaue Erklärung folgt in Kapitel 2.3) zu verstecken und diese wiederherstellen zu können. Die Schwierigkeit besteht darin, dass der Slack Space ein Bereich auf Speichermedien ist, der laut Computer (genauer laut Dateisystem) als „verwendet“ markiert ist, jedoch in Wirklichkeit keine Daten beinhaltet. Sollten die Dateien, denen der Slack Space zugeordnet ist, vergrößert, oder gelöscht und an gleicher Stelle neue Dateien abgelegt werden, werden unsere im Slack Space versteckten Daten überschrieben. Deswegen müssen unsere Daten redundant gespeichert und die ausgelesenen Daten auf Richtigkeit geprüft werden, wobei davon auszugehen ist, dass zumindest ein Teil der ursprünglichen Daten verloren gegangen ist. Eine Fehlerkorrektur, die die fehlerhaften Stellen findet und ausbessert, ist daher notwendig.

Algorithmen, die diesen Anforderungen entsprechen, gibt es bereits in der Codierungstheorie. Hier liegt der Fokus zwar auf Nachrichten, die nach deren Übertragung wieder rekonstruiert werden sollen, allerdings entspricht diese Problematik der des Speicherns von Daten im Slack Space. In der Codierungstheorie wird ein Nachrichtewort mittels Codierungsalgorithmus zu einem Codewort. Dieses wird einem Empfänger geschickt, was normalerweise nicht fehlerfrei passiert. Der Empfänger muss nun mit Hilfe eines Decodierungsalgorithmus das wahrscheinlichste ursprüngliche Nachrichtewort wieder herstellen. Im Slack Space lässt sich diese Vorgehensweise analog verwenden! Wir codieren ein Datenwort zu einem Codewort und speichern es ab. Wenn wir dieses Codewort wieder auslesen wollen, kann es sein, dass Teile davon überschrieben wurden, wodurch es nun fehlerhaft ist. Wir benötigen demnach einen Decodierungsalgorithmus, der das ursprüngliche Datenwort liefert. Der Übertragungsfehler aus der Codierungstheorie entspricht demnach dem Datenverlust im Slack Space, wenn Cluster überschrieben werden, in denen wir das Codewort (teilweise) gespeichert haben.

Um byteweise arbeiten zu können, ist es sinnvoll, Codes mit fixer Länge (Blockcodes) zu verwenden. Diese haben zwar den Nachteil, dass größere Datenmengen zuerst unterteilt werden müssen, allerdings gibt es mit dem Reed-Solomon Code (siehe Kapitel 4) ein Codierungsverfahren, das sehr gut geeignet ist, um unseren Ansprüchen gerecht zu werden, was man bereits in Kapitel 6 erkennt. Die Verwendung des RS Code zur Codierung auf der CD diente hier als Leitfaden, da es sich hierbei (vor allem in Kombination mit Produktcodes) um eine sehr effiziente Methode zur Fehlerkorrektur handelt.

Der in [11] beschriebene Algorithmus zur Decodierung von Fehlern bei RS Codes jenseits der Minimaldistanz war ein weiteres Argument für die Verwendung dieses Codes. Zwar kann die Laufzeit des Algorithmus mitunter sehr lang werden, allerdings steht im Vordergrund der Datenspeicherung im Slack Space, dass dieser, wenn man ihn nicht gerade künstlich erzeugen möchte, eher klein ausfällt und Platz Mangelware ist. Daher werden Algorithmen, die auf eine Reduktion des Speicherbedarfs abzielen, bevorzugt.

Durch Verwendung des komplexen Guruswami-Sudan Algorithmus, auf dem ein

großer Teil dieser Arbeit beruht, ist diese Diplomarbeit so aufgebaut, dass nach den Grundlagen in Kapitel 2 die Herleitung des GS Algorithmus folgt. In Kapitel 3, das sich stark an [11] orientiert, werden alle benötigten Teile des Algorithmus beschrieben, hergeleitet und bewiesen. Das ist notwendig, da der GS Algorithmus eigentlich die Zusammensetzung zweier anderer Algorithmen ist.

Die Kapitel 4 und 5 widmen sich dann zur Gänze RS Codes. Zuerst betrachten wir zwei verschiedene Möglichkeiten, um einen RS Code zu erhalten, da es zwei wesentlich verschiedene Codierungsverfahren gibt, die uns dennoch denselben Code liefern. Danach werden zwei Decodierungsverfahren vorgestellt. Zum einen der GS Algorithmus, dessen Grundlagen wir bereits in Kapitel 3 kennen gelernt haben und der auf dem ersten Codierungsverfahren beruht. Zum anderen wird der Vollständigkeit und Vergleichbarkeit halber der Berlekamp-Massey Algorithmus vorgestellt, der auf dem zweiten Codierungsverfahren beruht. Außerdem werden Möglichkeiten vorgestellt, um die Algorithmen zu verbessern.

In Kapitel 6 betrachten wir dann sowohl den GS als auch den BM Algorithmus etwas genauer, um mit deren Eigenschaften vertraut zu werden. Besonders ersterer ist sehr interessant, da er einen frei wählbaren Parameter m hat, der für die Fehlerkorrektur wichtig ist, allerdings auch andere Faktoren, wie z.B. die Laufzeit, massiv beeinflusst. Eine gute Abstimmung ist hier besonders wichtig. Bei der anschließend durchgeführten Analyse des BM Algorithmus fallen dann rasch Vor- und Nachteile der beiden Verfahren auf. Die so aufgezeigten Stärken und Schwächen liefern die Grundlagen für weitere Überlegungen.

In Kapitel 7 werden Möglichkeiten vorgestellt, um die RS Codes zu modifizieren. Sinn und Zweck ist es, die Codes den Gegebenheiten im Slack Space anzupassen. Zum einen können durch die Modifizierungen Fehlerfortpflanzungen verringert und Korrektoreigenschaften verbessert werden, zum anderen können unnötige Redundanzen verhindert werden, indem man z.B. RS Codes verkürzt, falls nur eine geringe Datenmenge zu speichern ist.

Zum Schluss, in Kapitel 8, wird das Wissen über den Slack Space mit den vorgestellten Codierungsverfahren kombiniert. Es werden drei unterschiedliche Varianten wie Daten gespeichert werden können vorgestellt, und miteinander verglichen. Dabei kommen die Modifizierungen aus Kapitel 7 zur Geltung.

2 Grundlagen

2.1 Begriffsbestimmungen

Wie in der Einleitung bereits beschrieben, betrachten wir Bereiche auf denen wir Daten speichern wollen. Da der Reed-Solomon Code ein linearer Blockcode ist (siehe dazu 11 und 12), müssen wir die Daten in Blöcke unterteilen. Wir bezeichnen, analog zum Nachrichtenwort über dem Alphabet A :

Definition 1. Ein Datenwort der Länge k ist eine Folge von Buchstaben $(b_i)_{i=1}^k$, wobei $k \in \mathbb{N}$ und $b_i \in A$ ist.

Für die Größenbezeichnungen werden die von der *Internationalen Elektronischen Kommission* (kurz IEC) eingeführten Begriffe KiB, MiB, GiB, ... anstatt der häufig sehr ungenau verwendeten Einheiten kB, MB, GB, ... verwendet. Nachdem im Normalfall Speichergrößen immer in Potenzen von 2 vorkommen, ist es sinnvoll dies bei ihrer Bezeichnung zu berücksichtigen:

Definition 2. Mit KiB (Kibibyte) bezeichnen wir eine Speichergröße von $2^{10} = 1.024$ Byte, mit MiB (Mebibyte) bezeichnen wir eine Speichergröße von $2^{20} = 1.048.576$ Byte, mit GiB (Gibibyte) bezeichnen wir eine Speichergröße von $2^{30} = 1.073.741.824$ Byte, usw.

Diese Bezeichnung ist genauer, als die bisher nur an die Größenordnungen angepassten Vorsilben¹.

Auf Grund des sehr beschränkt vorhandenen Slack Space (siehe Kapitel 2.3), ist es besonders wichtig eine geringe Redundanz zu haben. Im Unterschied zur Codierungstheorie, in der die Informationsrate R betrachtet wird, betrachten wir die Redundanz R (siehe Kapitel 8):

Definition 3. Als Redundanz R bezeichnen wir das Verhältnis

$$R = \frac{\text{Anzahl hinzugefügter Stellen}}{\text{Länge des Codeworts}}.$$

Ziel dieser Arbeit ist es ein Codierungsverfahren mit möglichst geringer Redundanz zu finden, das dennoch möglichst sicher ist. Die Redundanz ist gleichzeitig der prozentuale Anteil an Kontrollstellen die ein Codewort hat, ein Wert der auch häufig betrachtet wird.

2.2 Mathematische Definitionen und Grundlagen

In diesem Kapitel wird mathematisches Grundwissen aus der Codierungstheorie, bzw. allgemein der Mathematik zusammengefasst. Dieses lässt sich auch in [4] und [2] nachlesen. Detailliertere Verweise erfolgen in diesem Kapitel nur sporadisch.

Wie bereits beschrieben, wollen wir Datenwörtern der Länge k Codewörter der Länge n zuordnen, wobei beide Wörter über einem Alphabet A definiert sind. Codewörter sind demnach endliche Folgen $\chi = (\chi_i)_{i=1}^n$ mit $\chi_i \in A$. Die Menge aller

¹Bezeichnungen wie 8 MB RAM sind zum Beispiel nicht korrekt, da es sich in Wirklichkeit zumeist um $8 \cdot 2^{20} = 8.388.608$ Byte, also eigentlich rund 8,3 MB, oder aber exakt 8 MiB handelt.

möglichen Wörter $\{\beta_1\beta_2\dots\beta_n \mid \beta_i \in A\}$ wird Sequenzraum genannt. Sollte das Alphabet A zusätzlich ein Körper sein, so kann man die Wörter des Sequenzraums auch als Vektoren $(\beta_1, \beta_2, \dots, \beta_n)$ des arithmetischen Vektorraums A^n auffassen. Um Verwechslungen zu vermeiden und Argumenten leichter folgen zu können bezeichnen wir beliebige Wörter des Sequenzraums mit β und γ , während Codewörter (die zwar auch Elemente des Sequenzraums sind, aber zusätzlich Elemente des Codes) mit χ und ζ bezeichnet werden.

Bemerkung 1. Da Codewörter Elemente des Sequenzraums sind, ist die Menge aller Codewörter C eine Teilmenge des Sequenzraums. Es ist also $C \subseteq A^n$.

Definition 4. Als Hammingdistanz bezeichnen wir die Anzahl unterschiedlicher Stellen zweier Wörter $(\beta_i)_{i=1}^n$ und $(\gamma_i)_{i=1}^n$ des Sequenzraums A^n :

$$d((\beta_i)_{i=1}^n, (\gamma_i)_{i=1}^n) = |\{i \in \{1, 2, \dots, n\} \mid \beta_i \neq \gamma_i\}|$$

Diese Definition können wir gleich verwenden, um die sogenannte Minimaldistanz eines Codes zu bestimmen:

Definition 5. Als Minimaldistanz $d(C)$ eines Codes C bezeichnen wir den kleinsten Abstand zwischen zwei unterschiedlichen Codewörtern des Codes:

$$d(C) = \min \{d((\chi_i)_{i=1}^n, (\zeta_i)_{i=1}^n) \mid (\chi_i)_{i=1}^n, (\zeta_i)_{i=1}^n \in C, (\chi_i)_{i=1}^n \neq (\zeta_i)_{i=1}^n\}$$

Da eigentlich immer klar ist auf welchen Code man sich bezieht, wird die Minimaldistanz häufig auch nur mit d anstatt $d(C)$ bezeichnet.

Eine weitere nützliche Eigenschaft der Hammingdistanz ist die Erfüllung der Bedingungen für eine Metrik (wie man sich für $(\beta_i)_{i=1}^n, (\gamma_i)_{i=1}^n, (\delta_i)_{i=1}^n \in A^n$ schnell überlegt):

- $d((\beta_i)_{i=1}^n, (\gamma_i)_{i=1}^n) = 0 \Leftrightarrow (\beta_i)_{i=1}^n = (\gamma_i)_{i=1}^n$
- $d((\beta_i)_{i=1}^n, (\gamma_i)_{i=1}^n) = d((\gamma_i)_{i=1}^n, (\beta_i)_{i=1}^n)$
- $d((\beta_i)_{i=1}^n, (\delta_i)_{i=1}^n) \leq d((\beta_i)_{i=1}^n, (\gamma_i)_{i=1}^n) + d((\gamma_i)_{i=1}^n, (\delta_i)_{i=1}^n)$

Diese Eigenschaft brauchen wir, wenn wir uns Gedanken über die Fehlerkorrektur machen.

Betrachten wir dazu zuerst die Kugelumgebungen in A^n , die wir mit Hilfe der Hammingdistanz definieren können:

$$K_t((\beta_i)_{i=1}^n) := \{(\gamma_i)_{i=1}^n \in A^n \mid d((\beta_i)_{i=1}^n, (\gamma_i)_{i=1}^n) \leq t\}$$

Definition 6. Ein Code C ist ein t -fehlerkorrigierender Code, wenn

$$K_t((\chi_i)_{i=1}^n) \cap K_t((\zeta_i)_{i=1}^n) = \emptyset \quad \text{für alle } (\chi_i)_{i=1}^n, (\zeta_i)_{i=1}^n \in C, (\chi_i)_{i=1}^n \neq (\zeta_i)_{i=1}^n$$

ist.

Sollten also bis zu t Buchstaben eines Codeworts $(\chi_i)_{i=1}^n$ falsch sein, so befindet sich das entstandene Wort $(\beta_i)_{i=1}^n$ dennoch in der Kugelumgebung des ursprünglichen Codeworts $(\chi_i)_{i=1}^n$ und in keiner anderen Kugelumgebung. Anders ausgedrückt: In der Umgebung $K_t((\beta_i)_{i=1}^n)$ befindet sich genau ein Codewort, nämlich $(\chi_i)_{i=1}^n$, das

daher aus dem verfälschten Wort $(\beta_i)_{i=1}^n$ eindeutig rekonstruiert werden kann.

Doch wie können wir mit Hilfe dieser Information eine mögliche Decodierungsstrategie entwickeln? Ansätze gibt es mehrere:

Gehen wir davon aus, dass wir $(\beta_i)_{i=1}^n \in A^n$ ausgelesen haben und bezeichnen wir die Codewörter eines Codes C mit $(\chi_i)_{i=1}^n$ und $(\zeta_i)_{i=1}^n$. Dann decodieren wir $(\beta_i)_{i=1}^n$ jeweils zu $(\chi_i)_{i=1}^n \in C$, wenn für alle $(\zeta_i)_{i=1}^n \in C$ mit $(\zeta_i)_{i=1}^n \neq (\chi_i)_{i=1}^n$ gilt:

$$\text{ME } P((\chi_i)_{i=1}^n \text{ gesendet} \mid (\beta_i)_{i=1}^n \text{ empfangen}) \geq P((\zeta_i)_{i=1}^n \text{ gesendet} \mid (\beta_i)_{i=1}^n \text{ empfangen})$$

$$\text{ML } P((\beta_i)_{i=1}^n \text{ empfangen} \mid (\chi_i)_{i=1}^n \text{ gesendet}) \geq P((\beta_i)_{i=1}^n \text{ empfangen} \mid (\zeta_i)_{i=1}^n \text{ gesendet})$$

$$\text{NN } d((\beta_i)_{i=1}^n, (\chi_i)_{i=1}^n) \leq d((\beta_i)_{i=1}^n, (\zeta_i)_{i=1}^n)$$

Um bei jeder Strategie auf jeden Fall nur ein Codewort zu erhalten, ergänzt man alle um folgende Regel:

Sollten mehrere Codewörter $(\chi_i)_{i=1}^n \in C$ eine Eigenschaft erfüllen, so wähle man ein beliebiges aus.

Im ersten Fall, der *Minimal Error* Strategie, müssten die Wahrscheinlichkeiten für die einzelnen Codewörter $(\chi_i)_{i=1}^n \in C$ bekannt sein, was häufig nicht der Fall ist. Daher ist diese Methode nicht zu verwenden. Die zweite Variante, auch bekannt als die *Maximum Likelihood* Strategie, stimmt bei symmetrischen Kanälen mit der dritten Variante, bekannt unter dem Namen *Nearest Neighbour Decodierung*, überein, wie man z.B. in [4, Seite 10] nachlesen kann. Wir sehen also, dass die Verwendung der Hammingdistanz bei der Decodierung uns das wahrscheinlichste² Codewort liefert.

Ein sehr wichtiges Zusammenspiel zwischen der Minimaldistanz d und der Fehlerkorrektur t eines Codes liefert uns folgender Satz:

Satz 1. (ohne Beweis) Ein Code C ist genau dann t -fehlerkorrigierend, wenn $d(C) \geq 2t + 1$ ist.

Aus dieser Ungleichung erhalten wir durch Umformen, dass die größtmögliche Fehlerkorrektur

$$t = \left\lfloor \frac{d(C) - 1}{2} \right\rfloor$$

beliebige Stellen umfasst. Doch was passiert, wenn mehr als t Fehler auftreten? Hier können wir leider wenig machen. Folgender Satz gibt uns jedoch Auskunft über die Fehlererkennung:

Satz 2. (ohne Beweis) Ein Code C mit Minimaldistanz d kann bis zu $d - 1$ Fehler erkennen. Bei Fehlern an d Stellen kann er jedoch versagen.

Wie wir in Satz 1 gesehen haben, hängen die Parameter t und d eines Codes von einander ab. Natürlich gelten auch für die weiteren Größen, wie die Datenwortlänge k , die Codewortlänge n und die Alphabetgröße q gewisse Zusammenhänge, die jetzt aufgezeigt werden, bevor im Anschluss genauer auf gewisse Voraussetzungen eingegangen wird.

²Hier ist das wahrscheinlichste Codewort bezüglich der Maximum Likelihood Decodierung gemeint.

Satz 3. (ohne Beweis) Angenommen wir haben einen linearen $[n, k]$ -Code C mit der Minimaldistanz d gegeben. Dann gilt für die Parameter die sogenannte *Singleton-Schranke*:

$$k + d \leq n + 1$$

Häufig findet man diese auch in der Form $n - k \geq d - 1$.

Definition 7. Ein Code für den die Gleichheit gilt, also $k + d = n + 1$ ist, wird Maximum Distance Separable-Code (kurz MDS-Code) genannt.

Wie wir später sehen werden, sind Reed-Solomon Codes MDS-Codes. Sie eignen sich daher besonders gut, wenn man die Redundanz möglichst gering halten möchte.

Satz 4. (ohne Beweis) Angenommen wir haben einen $[n, k]$ -Blockcode C über dem Alphabet A mit $|A| = q$ Buchstaben, der bis zu t Fehler korrigiert. Dann gilt für die Parameter die sogenannte *Hamming-Schranke*:

$$n - k \geq \log_q \sum_{i=0}^t \binom{n}{i} (q - 1)^i$$

Definition 8. Ein Code für den in der obigen Ungleichung die Gleichheit gilt, also $n - k = \log_q \sum_{i=0}^t \binom{n}{i} (q - 1)^i$ ist, wird t -perfekter Code genannt.

Bemerkung 2. Codes die t -perfekt sind können bis zu t Fehler korrekt decodieren. Sollten jedoch mehr Fehler auftreten wird auf jeden Fall falsch decodiert, da sich das ausgelesene Wort $(\beta_i)_{i=1}^n$ in der Kugelumgebung eines anderen Codeworts befindet. Im Fall eines t -perfekten Codes können wir den Sequenzraum disjunkt in Kugeln $K_t((\chi_i)_{i=1}^n)$ zerlegen:

$$\bigcup_{(\chi_i)_{i=1}^n \in C} K_t((\chi_i)_{i=1}^n) = A^n$$

Reed-Solomon Codes C sind nicht t -perfekt mit $t = \lfloor \frac{d(C)-1}{2} \rfloor$. D.h. es gibt Kugeln $K_t((\beta_i)_{i=1}^n)$ mit $(\beta_i)_{i=1}^n \in A^n$, in denen sich kein Codewort befindet. Daher ist es sinnvoll Kugeln mit größeren Radien zu betrachten, um so eventuell doch noch ein Codewort zu finden, dass zu einem ausgelesenen Wort passen könnte. Diese Vorgehensweise entspricht dem Guruswami-Sudan Algorithmus, den wir im Verlauf dieser Arbeit kennenlernen werden.

Gehen wir nun einigen Begriffen auf den Grund, die bereits gefallen sind:

Definition 9. Als einen $[n, k]$ -Blockcode bezeichnen wir eine injektive Codierungsfunktion f_C , die Datenwörter der Länge k auf Codewörter der Länge $n \geq k$ abbildet:

$$f_C = \begin{cases} A^k \longrightarrow A^n \\ b_1 b_2 \dots b_k \mapsto \chi_1 \chi_2 \dots \chi_n \end{cases}$$

Daten die aus mehr, oder weniger als k Buchstaben bestehen werden entweder in Blöcke der Länge k unterteilt, oder auf einen Block der Länge k verlängert.

Reed-Solomon Codes sind Blockcodes. Datenwörter die kürzer als k sind werden mit Nullen aufgefüllt.

Der Begriff „Code“ wird in der Codierungstheorie mehrdeutig verwendet. Selten, aber doch, kann damit die Codierungsfunktion gemeint sein. Im Normalfall bezeichnet man damit aber die Menge aller Codewörter, die durch die Codierungsfunktion bestimmt wird:

$$\text{Code} = \{f_C(b_1 b_2 \dots b_k) \mid b_1 b_2 \dots b_k \in A^k\}$$

Bemerkung 3. Da Reed-Solomon Codes MDS-Codes sind, also $d = n - k + 1$ ist, und wir nur diese betrachten, werden Codes nur als $[n, k]$ -Codes bezeichnet anstatt sie korrekterweise $[n, k, d]$ -Codes zu nennen.

Bevor wir uns mit linearen Codes und in weiterer Folge Polynomcodes beschäftigen, wiederholen wir noch kurz Wissen über endliche Körper bzw. Mengen:

Die Struktur $(\mathbb{Z}_m, +, \cdot)$ ist ein Ring, bzw. ein Körper, wenn m prim ist.

Die für diese Aussage notwendigen Eigenschaften lassen sich leicht nachrechnen:

- $(\mathbb{Z}_m, +)$ ist eine kommutative Gruppe (abgeschlossen, assoziativ, kommutativ, neutrales Element, inverses Element).
- (\mathbb{Z}_m, \cdot) ist eine kommutative Halbgruppe mit Einselement (abgeschlossen, assoziativ, kommutativ, neutrales Element).
- Es gilt das Distributivgesetz.

Um zu zeigen, dass es sich um einen Körper handelt muss zusätzlich gelten:

- In $(\mathbb{Z}_m \setminus \{0\}, \cdot)$ haben alle Elemente ein inverses Element.

Bemerkung 4. Von jetzt an sei A immer ein endlicher Körper. Diese Einschränkung ist wichtig, damit wir eine Struktur für die Menge der Datenwörter und Codewörter erhalten. Der Sequenzraum A^n wird so nämlich zu einem Vektorraum, wenn wir die Vektorraumoperationen wie gewohnt komponentenweise definieren.

Definition 10. Wir bezeichnen einen Code als linearen Code, oder Linearcode, wenn dieser ein Unterraum von A^n ist:

$$C \leq A^n$$

Es ist also C abgeschlossen bezüglich Addition und skalarer Multiplikation.

Aus den Grundvorlesungen ist bekannt, dass sich alle Elemente eines Vektorraums als Linearkombination eines Erzeugendensystems darstellen lassen. Ein linear unabhängiges Erzeugendensystem heißt Basis, wobei die kanonische Basis die bekannteste ist. Die Anzahl linear unabhängiger Basisvektoren bezeichnet man auch als Dimension, wobei die Dimension eines Linearcodes wohldefiniert ist. Um einen Linearcode zu erhalten fordern wir von einer Codierungsfunktion zusätzlich zur Injektivität die Linearität:

$$\begin{aligned} f_C((b_i)_{i=1}^k + (c_i)_{i=1}^k) &= f_C((b_i)_{i=1}^k) + f_C((c_i)_{i=1}^k) \\ f_C(a(b_i)_{i=1}^k) &= a f_C((b_i)_{i=1}^k) \end{aligned}$$

Mit dieser Forderung ist der Code C immer ein Unterraum des Sequenzraums A^n . Diese Eigenschaften der Linearität und Injektivität lassen sich bei Reed-Solomon Codes leicht nachrechnen (siehe Kapitel 4).

Auf Grund der Linearität existieren für jeden linearen Code eine Generatormatrix $\mathcal{G} \in A^{k \times n}$, sowie eine Kontrollmatrix $\mathcal{H} \in A^{(n-k) \times n}$. Mit Hilfe dieser Matrizen lassen sich nicht nur Aussagen über Eigenschaften von Linearcodes (wie z.B. die Singleton-Schranke, Dualcodes, oder die Minimaldistanz) treffen, sondern auch Methoden zur Codierung und Decodierung finden.

Die Codierung erfolgt dabei relativ leicht durch Multiplikation eines Datenworts $(b_i)_{i=1}^k \in A^k$ mit der Generatormatrix \mathcal{G} : $f_C((b_i)_{i=1}^k) = (b_i)_{i=1}^k \cdot \mathcal{G}$. Die Decodierung erfolgt durch Multiplikation des ausgelesenen Worts $(\beta_i)_{i=1}^n \in A^n$ mit der transponierten Kontrollmatrix \mathcal{H}^T . Auf diese Weise berechnet man sich das so genannte Syndrom $s_{\mathcal{H}}((\beta_i)_{i=1}^n) = (\beta_i)_{i=1}^n \cdot \mathcal{H}^T$. Dieses Syndrom ist für alle Wörter aus der selben Nebenklasse des Codes³ gleich.

Unter Verwendung eines Standardkorrekturschemas ist dann die Decodierung eines ausgelesenen Worts möglich. Allerdings können wir nur für t -perfekte Codes dieses Schema eindeutig aufbauen. Sollte unser Code also nicht t -perfekt sein, gibt es Fehler die wir nicht eindeutig decodieren können. Dieser Nachteil in Kombination mit der großen Anzahl an Nebenklassen, die wir bei Reed-Solomon Codes haben (es gibt q^{n-k} verschiedene Nebenklassen mit jeweils q^k Wörtern des Sequenzraums) zeigt uns, dass Standardkorrekturschemas für unsere Zwecke nicht verwendbar sind.

Betrachten wir daher eine weitere wichtige Eigenschaft, die Reed-Solomon Codes (wie sich leicht nachrechnen lässt) auch erfüllen:

Definition 11. Ein Code wird als zyklisch bezeichnet, falls durch zyklisches Vertauschen der Buchstaben eines Codeworts wieder ein Codewort entsteht. Für zyklische Codes können wir also folgern:

$$\beta_1\beta_2 \dots \beta_n \in C \Rightarrow \beta_n\beta_1 \dots \beta_{n-1} \in C$$

Durch diese weitere Eigenschaft haben wir die Möglichkeit unsere Codewörter als Polynome über dem Alphabet A aufzufassen. Viele Argumente in dieser Arbeit bauen auf den Eigenschaften von Polynomcodes auf.

Doch betrachten wir zuerst die Identifizierung von $(\beta_i)_{i=1}^n \in C$ mit dem Polynom $\beta(x) = \beta_1 + \beta_2x + \dots + \beta_nx^{n-1}$:

Durch die Identifizierung ist es uns gelungen eine multiplikative Struktur auf dem ehemaligen Vektorraum $(A^n, +, (a \cdot)_{a \in A})$ zu definieren, da wir im Polynomring $(A[x], +, \cdot)$, den wir jetzt haben, die Polynome ja wie gewohnt multiplizieren können⁴. Jedoch kann der Grad des Produkts größer als $n - 1$ sein, weshalb wir kein Element des Codes mehr haben. Um das zu vermeiden, fassen wir die Elemente des Codes als Polynome aus $A[x]/(x^n - 1)$ auf. D.h. es handelt sich um Elemente des Faktorrings nach dem von

³Nachdem der Code Unterraum des Sequenzraums ist, können wir diesen in Partitionen, so genannte Nebenräume, $(\beta_i)_{i=1}^n + C$ aufteilen.

⁴Zur Erinnerung: Das Alphabet A ist ein Körper.

$x^n - 1$ erzeugten Hauptideal. Mit anderen Worten gilt in $A[x]/(x^n - 1)$: $x^n \equiv 1$, $x^{n+1} \equiv x$,
 \dots

Man sieht sofort, dass Multiplikation von x mit $\beta(x)$ der zyklischen Vertauschung um eine Stelle entspricht. Es gilt daher allgemein für einen zyklischen Code C :

$$\forall \beta(x) \in C, \forall p(x) \in A[x] : \beta(x) \cdot p(x) \pmod{(x^n - 1)} \in C$$

Genauso wie die Multiplikation, ist auch die Division in dem Polynomring $A[x]$ möglich. Diese erfolgt für zwei Polynom $a(x), b(x)$ wie aus der Schule bekannt. Es gibt also eindeutige Polynome $q(x)$ und $r(x)$, die die Gleichung

$$a(x) = q(x)b(x) + r(x)$$

mit $r(x) = 0$, oder $\deg r(x) < \deg b(x)$ erfüllen⁵. Unser Polynomring $A[x]$ ist demnach ein Euklidischer Ring.

Ein wichtiger Satz für zyklische Codes lautet (vgl. [4, Seite 27]):

Satz 5. (ohne Beweis) Sei C ein zyklischer Code der Länge n über dem Körper A . Dann gilt:

- C entspricht in eindeutiger Weise einem Ideal in $A[x]/(x^n - 1)$.
- Es existiert genau ein normiertes Polynom $g(x) \neq 0$ minimalen Grades in C . Dieses Polynom heißt Generatorpolynom von C .
- Das Generatorpolynom $g(x)$ ist Teiler von $x^n - 1$, d.h. $\exists h(x) \in A[x] : g(x) \cdot h(x) = x^n - 1$. $h(x)$ heißt Kontrollpolynom von C .
- Jedes Codewort $\chi(x)$ kann eindeutig in der Form $\chi(x) = f(x) \cdot g(x)$ geschrieben werden, wobei $\deg f(x) < n - \deg g(x)$ gilt, d.h. $g(x)$ teilt jedes Codepolynom und umgekehrt kann man $g(x)$ mit beliebigem Polynom mit Grad $< (n - \deg g(x))$ multiplizieren und erhält wieder ein Codepolynom.

Definition 12. Ein nicht-konstantes Polynom wird als irreduzibel bezeichnet, wenn es nur triviale Teiler hat:

$$p(x) \in A[x] \text{ ist irreduzibel} \Leftrightarrow (p(x) = a(x) \cdot b(x) \Rightarrow a(x) \text{ oder } b(x) \text{ konstant})$$

Bemerkung 5. Nachdem wir wissen, dass der Polynomring $A[x]$ über dem Körper A ein Euklidischer Ring ist, ist dieser auch ein ZPE-Ring. Die Faktorisierung jedes Polynoms in irreduzible Polynome ist demnach im wesentlichen (bis auf die Reihenfolge der Faktoren) eindeutig möglich.

Lemma 1. (ohne Beweis) Für die Zerlegung von Polynomen $p(x) \in A[x]$ und Elemente $a \in A$ gilt:

$$(x - a) \mid p(x) \Leftrightarrow p(a) = 0$$

Wir wollen nun das Alphabet A vergrößern. Dazu betrachten wir endliche Körper etwas genauer. In Grundvorlesungen [6, Seite 115-121] wurden die Galoisfelder bereits vorgestellt. Wir wissen daher:

⁵Mit $\deg b(x)$ bezeichnen wir den Grad des Polynoms $b(x)$.

Satz 6. (ohne Beweis) Zu jeder Primzahlpotenz $q = p^{\bar{n}}$ mit $\bar{n} \in \mathbb{N}$ gibt es bis auf Isomorphie genau einen endlichen Körper mit q Elementen und alle endlichen Körper sind von dieser Gestalt.

Diese Körper werden häufig auch mit \mathbb{F}_q , oder $GF(q) = GF(p^{\bar{n}})$ bezeichnet. Es gilt sogar:

$$x^q - x = \prod_{a \in GF(q)} (x - a)$$

Das Galoisfeld $GF(q)$ besteht also aus allen Nullstellen des Polynoms $x^q - x$.

Konstruiert wird das Galoisfeld für $\bar{n} > 1$ indem man mit Polynomen aus $\mathbb{Z}_p[x]$ modulo einem irreduziblen Polynom $p(x)$ vom Grad \bar{n} rechnet. Es lässt sich auch zeigen, dass es für jeden Grad \bar{n} mindestens ein irreduzibles Polynom gibt.

Für die Addition, bzw. Subtraktion fasst man $GF(q)$ als Vektorraum über \mathbb{Z}_p auf. Das 0-Polynom entspricht dem Nullvektor, das 1-Polynom entspricht dem Vektor $(0, 0, \dots, 1)$ u.s.w. bis zu dem Polynom $x^{\bar{n}-1}$ das dem Vektor $(1, 0, \dots, 0)$ entspricht. Sämtliche Polynome modulo $p(x)$ sind somit als Linearkombination dieser Vektoren darstellbar. Die Addition und Subtraktion von Elementen aus $GF(q)$ entspricht daher der vektoriellen Addition und Subtraktion in \mathbb{Z}_p .

Die Multiplikation, bzw. Division in $GF(q)$ lässt sich mit Hilfe des irreduziblen Polynoms erledigen. Bei genauerer Betrachtung lässt sich sogar feststellen, dass sich alle Elemente eines Galoisfelds (außer der Null) als Potenzen eines primitiven Elements darstellen lassen:

Satz 7. (ohne Beweis) Jeder endliche Körper besitzt ein primitives Element.

Wir können also auch jedes Element des Galoisfelds $GF(q)$ als Potenz eines primitiven Elements α darstellen, wobei wir wissen, dass $\alpha^{q-1} = 1$ ist. D.h. für jedes $a \in GF(p^{\bar{n}})$ gilt:

$$a = \alpha^m \text{ für ein } m \in \{0, 1, \dots, p^{\bar{n}} - 2\}, \text{ oder } a = 0.$$

Wie wir in Kapitel 4 sehen werden, sind Reed-Solomon Codes lineare $[q-1, k]$ -Blockcodes über dem Alphabet $A = GF(q)$. Wie für jeden Linearcode können wir auch für diese eine Generator- sowie eine Kontrollmatrix angeben, jedoch gibt es geeignetere Methoden wie wir im Laufe dieser Arbeit sehen werden, um zu codieren, bzw. zu decodieren.

2.3 Slack Space

Dateien werden auf Datenträgern nicht wahllos aneinander gefügt, sondern in sogenannten Clustern und Sektoren, die aus mehreren Bytes bestehen, abgespeichert. Die Größe dieser Cluster und Sektoren hängt von der Formatierung des Datenträgers und dem gewählten Dateisystem ab. Zusätzlich gibt es Einschränkungen, die von der Größe des Datenträgers und des benutzten Dateisystems abhängen. Betrachten wir zum Bei-

spiel ganz grob das Dateisystem NTFS [15] ⁶:

Dieses unterteilt standardmäßig eine Festplatte, die zwischen 2GiB und 2TiB groß ist, in Cluster die jeweils 4.096 Bytes haben und aus 8 Sektoren (1 Sektor hat demnach 512 Bytes) bestehen. Die Anzahl der Cluster ist natürlich abhängig von der Größe des Datenspeichers. In der sogenannten Master File Table (kurz MFT) werden Informationen zu Daten gespeichert. Diese Informationen beinhalten unter anderem den Speicherort und die Länge der Datei. Ist die Datei klein genug, also kleiner als 1 KiB = 1.024 Bytes, dann wird die Datei direkt in der MFT gespeichert. Alle größeren Dateien werden entsprechend einer festgelegten Speicherstrategie (teilweise) auf einem separaten Teil des Datenträgers abgelegt, wobei in der MFT die Cluster aufgelistet sind in denen man die zu der Datei gehörenden Informationen findet. Jeder Cluster wird dabei höchstens einer Datei zugeordnet. Nachdem Dateigrößen aber nicht notwendigerweise immer ein Vielfaches der Clustergröße sind, bleiben im letzten zu einer Datei gehörendem Cluster zwangsläufig einige Sektoren frei. D.h. dass Speicher zwar für eine Datei vorgesehen ist, diese ihn jedoch nicht benötigt und somit Speicherplatz „verschwendet“ wird.

Durch diese Vorgehensweise beim Speichern ergeben sich natürlich Einschränkungen was die Anzahl an Dateien sowie die maximale Größe einer Datei betrifft, die man speichern möchte. Wir wollen diese Eigenschaften jedoch nicht näher betrachten, sondern konzentrieren uns auf die nicht beschriebenen Bytes in den Clustern:

Definition 13. Wenn eine Datei kein Vielfaches der Clustergröße ist, dann werden beim Speichern, im letzten der Datei zugeordneten Cluster, Bytes für diese Datei reserviert die von der Datei nicht benötigt werden, aber auch für keine anderen Informationen vorgesehen sind. Diese Bytes bezeichnen wir als Slack Space der Datei [3, Ch.8. Metadata Category].

Als Slack Space bezeichnen wir die Summe des Slack Space aller Dateien ⁷.

Der Slack Space kann künstlich vergrößert werden, indem man bei der Formatierung der Festplatte bereits auf eine große Clustergröße achtet und viele kleine Dateien anlegt. Neben dieser Art der bewussten Bildung von Slack Space, bei der wir natürlich das Überschreiben der künstlich angelegten Dateien verhindern werden, um unsere versteckten Daten zu schützen, interessieren wir uns für Bereiche im Speicher, die möglichst große Stabilität aufweisen, sich also möglichst nicht verändern. Wie man sich vorstellen kann, ist das Hinterlegen von Dateien im Slack Space mit einer großen Unsicherheit verbunden, da beim Löschen (=Freigeben der Cluster) oder Erweitern der eigentlichen Datei der dazu gehörende Slack Space überschrieben werden könnte und so die gesamte Information verloren wäre.

Der Slack Space des Betriebssystems ist hier von besonderem Interesse, da sich dieser im Normalfall nur beim Einspielen von Updates und Servicepacks ändert. Wie man in [12] nachlesen kann, bleiben bei Windows Betriebssystemen durchschnittlich 78% des ursprünglichen Slack Space erhalten.

⁶Ausführliche Informationen zu NTFS findet man auch in [3, Ch.11. NTFS Concepts].

⁷Dieses zu betrachten ist aber unpraktisch (man denke z.B. an temporäre Dateien), weshalb man sich auf Slack Space von Dateien mit speziellen Gemeinsamkeiten beschränkt.

Betrachten wir noch ein erstes Modell zur Berechnung des potentiell vorhandenen Slack Space einer Datei im Dateisystem NTFS [12, Seite 191]. Gehen wir davon aus, dass k die Größe des Clusters ist, also die Anzahl an Sektoren, aus denen er besteht, und s die Anzahl an Bytes pro Sektor. Außerdem nehmen wir an, dass die Datei größer als 1 KiB ist und somit nicht in der MFT gespeichert wird. Zur Vereinfachung berechnen wir uns außerdem nur die Anzahl freier Bytes in freien Sektoren:

Wir sehen, dass wir unterscheiden müssen, ob die Datei größer als 4 KiB ist oder kleiner. Das liegt daran, dass (nachdem die Datei ja größer als 1 KiB ist) in erstem Fall das Ende der Datei bereits im ersten Sektor eines Clusters liegen kann, während im zweiten Fall die Datei frühestens im dritten Sektor (bei standardmäßiger Wahl eines Sektors mit 512 Byte) zu Ende ist. Die zu erwartende Größe des Slack Space einer Datei die größer als 4 KiB ist (was eigentlich auf die meisten Dateien zutrifft) $S_{s,k}$ berechnen wir dann durch:

$$S_{s,k} = s \sum_{x=1}^{k-1} x \frac{1}{k} = s \frac{k-1}{2}$$

Wobei wir mit $\sum_{x=1}^{k-1} x \frac{1}{k}$ die zu erwartenden freien Sektoren im letzten Cluster einer Datei berechnen unter der Voraussetzung, dass sich das Ende der Datei auf alle Sektoren eines Clusters gleich verteilt und der erste Sektor sicher beschrieben wird. Demnach wird der zweite bis k -te Sektor des Clusters jeweils mit einer Wahrscheinlichkeit von $\frac{1}{k}$ beschrieben. Bei unserer standardmäßigen Formatierung der Festplatte erhalten wir demnach durchschnittlich $512 \cdot \frac{8-1}{2} = 1.792$ freie Bytes pro Datei. Bei über 40.000 Dateien die das Betriebssystem Windows 7 erstellt, bzw. über 55.000 bei Windows 8 [12, Seite 190] ergibt sich ein dementsprechend großer Slack Space.

Für den Fall, dass die Datei zwischen 1 KiB und 4 KiB groß ist, können wir dieselben Überlegungen anstellen. Wir sehen, dass wir nur k durch $k-2$ ersetzen müssen, um uns den Slack Space von Dateien dieser Größe zu berechnen, da die ersten zwei Cluster ja mit Sicherheit beschrieben wurden und das Ende der Datei irgendwo zwischen dem dritten und k ten Sektor liegen muss. Eine derartige Datei würde uns also im Schnitt nur $512 \cdot \frac{8-3}{2} = 1.280$ freie Bytes liefern.

3 Vorbereitung GS Algorithmus

Bevor wir uns der Decodierung von Reed-Solomon Codes mittels des Guruswami-Sudan Algorithmus (kurz GS Algorithmus) widmen, müssen wir uns mit bivariaten Polynomen, also Polynomen in 2 Variablen beschäftigen:

Definition 14. Als bivariates Polynom bezeichnen wir ein Polynom, bestehend aus zwei Variablen (z.B. x und y) mit Koeffizienten $a_{i,j} \in A$, wobei nur endlich viele Koeffizienten $a_{i,j} \neq 0$ sind:

$$Q(x, y) = \sum_{i,j \geq 0} a_{i,j} x^i y^j = \sum_{(i,j) \in I} a_{i,j} x^i y^j \text{ mit } I \subset \mathbb{N}^2 \text{ endlich.}$$

Wir interessieren uns für bivariate Polynome mit Koeffizienten aus Galoisfeldern $a_{i,j} \in GF(q)$:

$$Q(x, y) = \sum_{(i,j) \in I} a_{i,j} x^i y^j$$

Grund dafür diese Polynome zu betrachten, ist der Aufbau des GS Algorithmus (siehe Kapitel 5.1), sowie die Verwendung von Reed-Solomon Codes (siehe Kapitel 4). Ohne jetzt schon zu detailliert auf beide Punkte einzugehen sei erklärt:

Der GS Algorithmus ist die Kombination eines Interpolations- und Faktorisierungsalgorithmus, wobei im Zuge der Interpolation ein bivariates Polynom entstehen soll, das gewisse Eigenschaften hat, damit die Faktorisierung eben dieses Polynoms einen wesentlichen Beitrag zur Decodierung liefert. Das Arbeiten in einem endlichen Körper $\mathbb{K} = GF(q)$ liegt an dem RS Code, der über eben diesen Körpern erklärt ist.

3.1 Ordnung bivariater Polynome

Um diese bivariaten Polynome besser betrachten zu können, ist es hilfreich sie zu ordnen. Das heißt wir brauchen eine Ordnung der Monome

$$\mathbb{M}[x, y] = \{x^i y^j : i, j \geq 0\}$$

Wie im Fall von „normalen“ Polynomen benötigen wir hierzu den Grad des Polynoms. Wir betrachten also nur die Exponenten von x und y . Auf diesen Tupeln aus \mathbb{N}^2 wollen wir jetzt eine Ordnung definieren, die folgende drei Eigenschaften hat (siehe auch [11, Seite 7]):

1. Wenn $i_1 \leq i_2$ und $j_1 \leq j_2$, dann soll auch $(i_1, j_1) \leq (i_2, j_2)$
2. Alle Monome sollen miteinander verglichen werden können. D.h. entweder $(i_1, j_1) \leq (i_2, j_2)$, oder $(i_2, j_2) \leq (i_1, j_1)$
3. Wenn $(i_1, j_1) < (i_2, j_2)$, dann gilt für alle $(i_3, j_3) \in \mathbb{N}^2$, dass $(i_1, j_1) + (i_3, j_3) < (i_2, j_2) + (i_3, j_3)$

Um diese Forderungen zu erfüllen, benötigen wir den gewichteten Grad eines Monoms, den wir mit Hilfe des Paares $\mathbf{w} = (u, v)$ berechnen ($u, v \in \mathbb{N}$). Die Berechnung des gewichteten Grades ist gegeben durch:

$$\deg_{\mathbf{w}}(x^i y^j) = i \cdot u + j \cdot v$$

Ein erstes ordnen der Monome aus $\mathbb{M}[x, y]$ ist somit bereits möglich:

$$\deg_{\mathbf{w}}(x^{i_1} y^{j_1}) \leq \deg_{\mathbf{w}}(x^{i_2} y^{j_2}) \Rightarrow x^{i_1} y^{j_1} \leq x^{i_2} y^{j_2} \text{ bzw. } (i_1, j_1) \leq (i_2, j_2)$$

Somit müssen wir uns nur noch überlegen, wie die Monome $x^i y^j$ bei gleichem gewichteten Grad geordnet werden sollen ([11, Seite 8]):

Definition 15. Die **w**-lex Ordnung ist definiert als:

$$x^{i_1} y^{j_1} < x^{i_2} y^{j_2} \Leftrightarrow \deg_{\mathbf{w}}(x^{i_1} y^{j_1}) < \deg_{\mathbf{w}}(x^{i_2} y^{j_2}) \text{ oder } \deg_{\mathbf{w}}(x^{i_1} y^{j_1}) = \deg_{\mathbf{w}}(x^{i_2} y^{j_2}) \text{ und } i_1 < i_2$$

Definition 16. Die **w**-revlex Ordnung ist genauso definiert, wie die **w**-lex Ordnung bis auf den Fall der Gleichheit des gewichteten Grades, für den festgelegt wird, dass $x^{i_1} y^{j_1} < x^{i_2} y^{j_2} \Leftrightarrow j_1 < j_2$.

Beispiel 1. Eine $(1, 6)$ -revlex Ordnung für $x^i y^j$ würde dann wie folgt aussehen:

i=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...
j=0	0	1	2	3	4	5	6	8	10	12	14	16	18	21	...
1	7	9	11	13	15	17	19	22	...						
2	20	23	...												
⋮	⋮														

Bemerkung 6. Wenn wir von jetzt an von einer Ordnung bei bivariaten Polynomen sprechen, dann beziehen wir uns immer auf die revlex-Ordnung. Diese Festlegung ist für den später beschriebenen GS Algorithmus wichtig und spielt in den beiden zu diesem gehörenden Teilalgorithmen eine Rolle, doch dazu später mehr.

Bemerkung 7. Mit Hilfe der Ordnung ist es uns möglich die Monome aus $\mathbb{M}[x, y]$ zu reihen. Es gibt also eine Bijektion zwischen den Potenzen von x und y aus \mathbb{N}^2 und den natürlichen Zahlen (siehe dazu auch Beispiel 1; z.B. ist $x^5 y^1$ das 17te Monom, also $(5, 1)$ das 17te Tupel aus \mathbb{N}^2). Wir bezeichnen die geordneten Monome mit $\phi_j(x, y)$, wobei $j \in \mathbb{N}$. Nachdem bivariate Polynome endliche Linearkombinationen von Monomen sind, können wir diese daher schreiben als:

$$Q(x, y) = \sum_{j=0}^J a_j \phi_j(x, y) \text{ mit } a_j \neq 0$$

Definition 17. Das Monom $\phi_J(x, y)$ wird dabei als das führende Monom von $Q(x, y)$ bezeichnet ([11, Seite 8]). Wir schreiben auch $\text{LM}(Q(x, y)) := \phi_J(x, y)$.

$\phi_J(x, y)$ ist bezüglich jeder Ordnung das J -te Monom in $\mathbb{M}[x, y]$. Diesen Index J bezeichnen wir als Rang des Polynoms, $\text{Rang}(Q(x, y)) := J$.

Der Vergleich von Polynomen, der nun bezüglich einer gegebenen Ordnung möglich ist, erfolgt über die führenden Monome von $Q_1(x, y) = \sum_{j=0}^{J_1} a_{j_1} \phi_{j_1}(x, y)$ und $Q_2(x, y) = \sum_{j=0}^{J_2} a_{j_2} \phi_{j_2}(x, y)$, wobei nur die Eigenschaften der Transitivität und Reflexivität, jedoch nicht die der Totalität und Antisymmetrie erfüllt sind. Bei der gewichteten Ordnung handelt es sich demnach um eine Quasiordnung. Es ist also $Q_1(x, y) < Q_2(x, y)$ genau dann, wenn $\text{LM}(Q_1(x, y)) = \phi_{j_1}(x, y) < \phi_{j_2}(x, y) = \text{LM}(Q_2(x, y))$, bzw. wenn der $\text{Rang}(Q_1(x, y)) < \text{Rang}(Q_2(x, y))$ ist.

Wie wir im weiteren Verlauf sehen werden, interessiert es uns auch zu wissen, welche maximalen y -Potenzen, bzw. x -Potenzen es gibt, die kleiner als ein gegebenes Monom bezüglich einer $(1, v)$ -Ordnung sind. Nehmen wir also an wir haben eine $(1, v)$ -revlex Ordnung und das Monom $\phi_C(x, y)$. Uns interessiert jetzt was die größtmöglichen x - und y -Potenzen sind, deren Indices kleiner als die des Monoms sind. Dazu müssen wir wissen, das wievielte Monom y^L bzw. x^K ist.

Definition 18. Als $\text{Rang}_v(x^K)$ (nach [11, Seite 9]) bezeichnen wir den Rang von x^K , bezüglich einer $(1, v)$ -revlex Ordnung. Das ist die Position, die dieses Monom in der gegebenen Ordnung hat. Als $\text{Rang}_v(y^L)$ bezeichnen wir die Position von y^L .

Mit anderen Worten gibt $\text{Rang}_v(x^K)$, bzw. $\text{Rang}_v(y^L)$ an, der wievielte Summand x^K bzw. y^L in der mit der $(1, v)$ -revlex Ordnung geordneten Reihe $Q(x, y) = \sum_{(i,j) \in \mathbb{N}^2} x^i y^j$ ist.

Beispiel 2. Für unsere $(1, 6)$ -revlex Ordnung bedeutet das, dass $\text{Rang}_6(x^{13}) = 21$, oder $\text{Rang}_6(y^1) = 7$ ist.

Auf Grund der Definition von $\text{Rang}_v(\cdot)$ und der $(1, v)$ -revlex Ordnung wissen wir, dass x^K das erste Monom vom Grad K und y^L das letzte Monom vom Grad $v \cdot L$ ist. Mit anderen Worten ([11, Seite 10]):

$$\begin{aligned}\text{Rang}_v(x^K) &= |\{(i, j) \in \mathbb{N}^2 : i + v \cdot j < K\}| \\ \text{Rang}_v(y^L) &= |\{(i, j) \in \mathbb{N}^2 : i + v \cdot j \leq L \cdot v\}| - 1\end{aligned}$$

Mit diesen Gleichungen können wir nun berechnen an der wievielten Stelle sich x^K und y^L in einer $(1, v)$ -revlex Ordnung befinden:

Satz 8. (siehe auch [11, Seite 11]) Sei $K \geq 0$ und $L \geq 0$, sowie $r = K \bmod v$. Dann lässt sich der Rang, bzw. die Position der Monome x^K und y^L wie folgt berechnen:

$$\begin{aligned}\text{Rang}_v(x^K) &= \frac{K^2}{2v} + \frac{K}{2} + \frac{r(v-r)}{2v} \\ \text{Rang}_v(y^L) &= \frac{vL^2}{2} + \frac{(v+2)L}{2}\end{aligned}$$

Beweis. Wir beweisen zuerst die Gleichung für $\text{Rang}_v(y^L)$: Dazu bedienen wir uns folgender Rekursion:

$$\begin{aligned}\text{Rang}_v(y^L) &= |\{(i, j) \in \mathbb{N}^2 : i + v \cdot j < L \cdot v\}| - 1 \\ &= (|\{(i, j) \in \mathbb{N}^2 : i + v \cdot j < (L-1) \cdot v\}| - 1) + \\ &\quad |\{(i, j) \in \mathbb{N}^2 : (L-1) \cdot v + 1 \leq i + v \cdot j \leq L \cdot v\}| \\ &= \text{Rang}_v(y^{L-1}) + v \cdot L + 1 = \text{Rang}_v(y^{L-2}) + v \cdot (L-1) + 1 + v \cdot L + 1 = \dots \\ &= v \cdot \sum_{j=1}^L j + L = \frac{v(L+1)L}{2} + \frac{2L}{2} = \\ &= \frac{vL^2}{2} + \frac{(v+2)L}{2}\end{aligned}$$

Um die Gleichung für $\text{Rang}_v(x^K)$ zu beweisen, bedienen wir uns ebenfalls der zu diesem Index gehörenden Menge $\{(i, j) \in \mathbb{N}^2 : i + v \cdot j < K\}$. Außerdem sei $r = K \bmod v$:

$$\begin{aligned}\text{Rang}_v(x^K) &= |\{(i, j) \in \mathbb{N}^2 : i + v \cdot j < K\}| \\ &= K + (K-v) + (K-2v) + \dots + (K - \frac{K-r}{v} \cdot v) \\ &= \sum_{j=0}^{\frac{K-r}{v}} (K - j \cdot v) \\ &= K \cdot (\frac{K-r}{v} + 1) - v \cdot \frac{(\frac{K-r}{v}) \cdot (\frac{K-r}{v} + 1)}{2} \\ &= \frac{2K^2 - 2Kr + 2Kv - K^2 + 2Kr - r^2 - Kv + rv}{2v} \\ &= \frac{K^2}{2v} + \frac{K}{2} + \frac{r(v-r)}{2v}\end{aligned}$$

□

Betrachten wir noch einmal die Frage nach den maximalen x - und y -Potenzen, die kleiner als ein gegebenes Monom $\phi_C(x, y)$ sind:

Beispiel 3. Nehmen wir jetzt an, dass wir von unserer $(1, 6)$ -revlex Ordnung das 22te Monom, also x^7y^1 , gegeben haben. Wir suchen nun die größtmöglichen x^K und y^L , deren Ränge trotzdem noch kleiner, bzw. maximal gleich 22 sind:

$$\begin{aligned} \text{Rang}_6(x^K) &\leq 22 \\ \frac{K^2}{12} + \frac{K}{2} + \frac{r(6-r)}{12} &\leq 22 \\ &\vdots \\ K &\leq 13 \end{aligned}$$

Also befinden sich alle x^K mit $K \leq 13$ vor dem 22-ten Monom unserer $(1, 6)$ -revlex Ordnung bzw. ist $x^{13} \leq x^7y^1$ bezüglich unserer Ordnung. Für die y^L lässt sich der Exponent analog berechnen:

$$\begin{aligned} \text{Rang}_6(y^L) &\leq 22 \\ \frac{vL^2}{2} + \frac{(v+2)L}{2} &\leq 22 \\ &\vdots \\ L &\leq 2 \end{aligned}$$

Es ist also auch $y^2 \leq x^7y^1$ bezüglich unserer $(1, 6)$ -revlex Ordnung.

3.2 Nullstellen bivariater Polynome

Als nächstes wollen wir uns überlegen, wie Nullstellen und mehrfache Nullstellen von Polynomen in zwei Variablen behandelt werden können. Als Nullstelle bezeichnen wir ein Paar $(\lambda, \mu) \in GF(q)^2$, für das das Polynom $Q(x, y) \in GF(q)[x, y]$ den Wert 0 annimmt, also $Q(\lambda, \mu) = 0$. Von größerem Interesse für uns sind die mehrfachen Nullstellen:

Definition 19. Wir sagen $Q(x, y) = \sum_{(i,j) \in I} a_{i,j}x^i y^j$ mit $I \subset \mathbb{N}^2$ endlich, hat eine m -fache Nullstelle bei $(0, 0)$ und bezeichnen das als

$$\text{ord}(Q : 0, 0) := m$$

wenn $a_{i,j} = 0$ für alle $(i, j) : i + j < m$ ist ([11, Seite 13]). $Q(x, y)$ besitzt also keine Monome mit $\deg_{(1,1)}(x, y) < m$. Analog besitzt $Q(x, y)$ eine m -fache Nullstelle bei $(\lambda, \mu) \in GF(q)^2$, also

$$\text{ord}(Q : \lambda, \mu) = m,$$

wenn $Q(x + \lambda, y + \mu)$ eine m -fache Nullstelle bei $(0, 0)$ hat.

Um $\text{ord}(Q : \lambda, \mu)$ berechnen zu können, müssen wir also das Polynom $Q(x + \lambda, y + \mu)$ in x und y darstellen. Dies können wir mit Hilfe des Binomischen Lehrsatzes wie folgt realisieren:

Satz 9. Wenn $Q(x, y) = \sum_{(i,j) \in \mathcal{I}} a_{i,j} x^i y^j$ ist, dann lässt sich für jedes $(\lambda, \mu) \in GF(q)^2$ das Polynom $Q(x + \lambda, y + \mu)$ darstellen als

$$\sum_{(r,s) \in \mathcal{I}} Q_{r,s}(\lambda, \mu) x^r y^s,$$

wobei

$$Q_{r,s}(\lambda, \mu) = \sum_{\substack{i \geq r, j \geq s, \\ (i,j) \in \mathcal{I}}} \binom{i}{r} \binom{j}{s} a_{i,j} \lambda^{i-r} \mu^{j-s}$$

ist ([11, Seite 11]). Wir verwenden im Folgenden auch die Bezeichnung $D_{r,s}Q(\lambda, \mu)$ anstatt $Q_{r,s}(\lambda, \mu)$.

Beweis. Mit Hilfe des Binomischen Lehrsatzes lässt sich dieser Zusammenhang sehr leicht zeigen:

$$\begin{aligned} Q(x + \lambda, y + \mu) &= \sum_{(i,j) \in \mathcal{I}} a_{i,j} (x + \lambda)^i (y + \mu)^j \\ &= \sum_{(i,j) \in \mathcal{I}} a_{i,j} \left(\sum_{r=0}^i \binom{i}{r} x^r \lambda^{i-r} \right) \left(\sum_{s=0}^j \binom{j}{s} y^s \mu^{j-s} \right) \\ &= \sum_{(r,s) \in \mathcal{I}} x^r y^s \left(\sum_{\substack{i \geq r, j \geq s, \\ (i,j) \in \mathcal{I}}} \binom{i}{r} \binom{j}{s} a_{i,j} \lambda^{i-r} \mu^{j-s} \right) \\ &= \sum_{(r,s) \in \mathcal{I}} Q_{r,s}(\lambda, \mu) x^r y^s \end{aligned}$$

□

Aus diesem Satz folgen zwei für uns interessante Aussagen (siehe auch [11, Seite 15]). Beide sind nützlich für Kötters Interpolationsalgorithmus (Kapitel 3.3.1) und daher wichtig für den GS Algorithmus.

Korollar 1. Da die $Q_{r,s}(\lambda, \mu)$ die Koeffizienten von $Q(x, y)$ sind, hat dieses Polynom genau dann eine m -fache Nullstelle bei (λ, μ) , wenn $Q_{r,s}(\lambda, \mu) = 0$ für alle r, s mit $0 \leq r + s < m$.

Beweis. Die Richtigkeit dieser Aussage folgt unmittelbar aus Definition 19 und Satz 9.

□

Korollar 2. Wenn $\tilde{Q}(x, y) = xQ(x, y)$, lassen sich die Koeffizienten von $\tilde{Q}(x, y)$ mit Hilfe der Koeffizienten von $Q(x, y)$ berechnen. Es gilt:

$$\tilde{Q}_{r,s}(\lambda, \mu) = Q_{r-1,s}(\lambda, \mu) + \lambda Q_{r,s}(\lambda, \mu) \text{ mit } Q_{-1,s}(\lambda, \mu) = 0.$$

Beweis. Mit Hilfe von Definition 19 und Satzes 9 können wir zeigen:

$$\begin{aligned}
\tilde{Q}(x + \lambda, y + \mu) &= (x + \lambda)Q(x + \lambda, y + \mu) \\
&= (x + \lambda) \sum_{(r,s) \in \mathcal{I}} Q_{r,s}(\lambda, \mu) x^r y^s \\
&= \sum_{(r,s) \in \mathcal{I}} Q_{r,s}(\lambda, \mu) x^{r+1} y^s + \sum_{(r,s) \in \mathcal{I}} \lambda Q_{r,s}(\lambda, \mu) x^r y^s \\
&= \sum_{(r,s) \in \mathcal{I}} (Q_{r-1,s}(\lambda, \mu) + \lambda Q_{r,s}(\lambda, \mu)) x^r y^s
\end{aligned}$$

□

Analoges lässt sich für den Fall $\tilde{Q}(x, y) = yQ(x, y)$ zeigen, der uns allerdings nicht weiter interessiert.

3.3 Interpolation bivariater Polynome

Wie schon zu Beginn von Kapitel 3 umrissen, besteht der GS Algorithmus sowohl aus einem Interpolations- als auch einem Faktorisierungsteil. In diesem Abschnitt wird näher auf die Interpolation eingegangen. Dazu müssen wir uns überlegen, wie man ein bivariates Polynom mit m-fachen Nullstellen herstellt. Dass es so ein Polynom geben muss, sagt uns folgender Satz:

Satz 10. ([11, Seite 16]) Angenommen wir haben bei $(\lambda_i, \mu_i)_{i=1}^n$ Nullstellen mit Vielfachheiten $m(\lambda_i, \mu_i)_{i=1}^n$ und eine Ordnung $\phi_0 < \phi_1 < \dots$ gegeben, dann existiert ein Interpolationspolynom $Q(x, y)$ ungleich dem Nullpolynom wie folgt:

$$Q(x, y) = \sum_{i=0}^C a_i \phi_i(x, y)$$

mit

$$C = \sum_{i=1}^n \binom{m(\lambda_i, \mu_i) + 1}{2}$$

Beweis. Dank der Ordnung können wir jedes Polynom $\sum_{(r,s) \in \mathcal{I}} a_{r,s} x^r y^s$ umordnen zu $\sum_{j=0}^J a_j \phi_j$. Dieses Polynom soll nun Nullstellen der Vielfachheit $m(\lambda_i, \mu_i)_{i=1}^n$ bei $(\lambda_i, \mu_i)_{i=1}^n$ besitzen. Wegen Satz 9 erhalten wir insgesamt $\sum_{i=1}^n \binom{m(\lambda_i, \mu_i) + 1}{2} = C$ lineare homogene Gleichungen für die Koeffizienten a_i . Wenn wir nun ein Polynom mit den gewünschten Nullstellen, ungleich dem Nullpolynom, haben wollen, benötigen wir $C + 1$ Koeffizienten. Wir haben also mehr Koeffizienten als lineare homogene Gleichungen. D.h. es muss ein Polynom wie im Satz beschrieben geben. □

Nachdem wir nun wissen, dass es ein Interpolationspolynom geben muss, überlegen wir uns wie man so ein Polynom am leichtesten berechnen kann. Dazu benötigen wir ein paar Zusatzüberlegungen: Unsere Aufgabe ist es, die Koeffizienten eines Polynoms mit gegebenen Nebenbedingungen

$$D_{r,s}Q(\lambda, \mu) = 0$$

zu berechnen, das bezüglich unserer Ordnung minimal ist. Nachdem wir die Paare $(r, s) \in \mathbb{N}^2$ mit Hilfe unserer Ordnung gereiht haben, können wir auch die Nebenbedingungen unnummerieren zu

$$D_i Q(\lambda, \mu) = 0 \text{ mit } i = 0, 1, \dots, \binom{m(\lambda, \mu) + 1}{2}$$

für alle Nullstellen $(\lambda, \mu) \in GF(q)$. Diese Nebenbedingungen sind lineare Abbildungen (was sich einfach nachrechnen lässt [11, Seite 20]). Sie erfüllen also für alle $a, b \in GF(q)$:

$$D_i(aQ_1 + bQ_2) = aD_iQ_1 + bD_iQ_2$$

Wir können daher für die Abbildung D_i einen Kern betrachten. Dieser ist:

$$K_i = \ker D_i = \{Q : D_iQ = 0\}$$

Zusätzlich betrachten wir folgende Bilinearform ([11, Seite 21]) bezüglich D_i :

Definition 20.

$$[Q_1, Q_2]_{D_i} := D_i(Q_1)Q_2 - D_i(Q_2)Q_1$$

Diese hat folgende Eigenschaft:

Lemma 2. Für alle $Q_1(x, y), Q_2(x, y) \in GF(q)[x, y]$ gilt $[Q_1, Q_2]_{D_i} \in \ker D$ und falls $Q_1 > Q_2$ ($Q_2 \notin \ker D$) ist $\text{Rang}([Q_1, Q_2]_{D_i}) = \text{Rang}(Q_1)$.

Beweis. Nachdem wir wissen, dass $D(\cdot)$ eine lineare Abbildung ist, ist folgende Umformung leicht nachvollziehbar:

$$D([Q_1, Q_2]_{D_i}) = D(D(Q_1)Q_2 - D(Q_2)Q_1) = D(Q_1)D(Q_2) - D(Q_2)D(Q_1) = 0$$

Womit folgt, dass $[Q_1, Q_2]_{D_i} \in \ker D$. Dass der Rang von $[Q_1, Q_2]_{D_i}$ gleich dem von Q_1 ist, falls $Q_1 > Q_2$ ($Q_2 \notin \ker D$), sieht man aus Definition der Bilinearform:

$$D_i(Q_1) \sum_{j=0}^{J_2} a_{2,j} \phi_j(x, y) - D_i(Q_2) \sum_{j=0}^{J_1} a_{1,j} \phi_j(x, y) = \sum_{j=0}^{J_1} a_j \phi_j(x, y)$$

Mit $a_{J_1} = a_{1,1} \neq 0$, da $Q_1 > Q_2$. □

Bemerkung 8. Um möglichst effizient ein Interpolationspolynom $Q(x, y)$ zu berechnen, dass für vorgegebene $(\lambda_i, \mu_i)_{i=1}^n$ eine Nullstelle der Vielfachheit $m(\lambda_i, \mu_i)_{i=1}^n$ hat, legen wir zuerst einmal zur Vereinfachung fest, dass alle $m(\lambda_i, \mu_i)_{i=1}^n = m$ sind⁸. Außerdem, um Korollar 2 auszunutzen, reihen wir die (r, s) entsprechend einer $(m-1, 1)$ -lex Ordnung: $(0, 0), (0, 1), \dots, (0, m-1), (1, 0), \dots, (1, m-2), \dots, (m-1, 0)$.

Weiters benötigen wir:

⁸Wir werden später sehen, dass diese Vielfachheit eine wichtige Rolle spielt. Je größer diese ist, desto wichtiger ist das Paar (λ, μ) für den GS Algorithmus. Nachdem wir jedoch nicht wissen, welche Paare korrekt und welche falsch sind (dazu mehr in Kapitel 8), ist es am sinnvollsten allen die gleiche Wichtigkeit einzuräumen und m , unabhängig vom Paar (λ, μ) , immer denselben Wert zu geben.

Definition 21. Wir definieren L_0 als

$$L_0 := \max\{L : \text{Rang}_v(y^L) \leq C\}$$

mit $C = n \binom{m+1}{2}$, dem Rang des Polynoms $Q(x, y)$, und einem noch unbekanntem v das wir abhängig von der noch folgenden Faktorisierung wählen müssen (vgl. Satz 12).

Dann können wir das Interpolationspolynom auch schreiben als:

$$Q(x, y) = \sum_{j=0}^{L_0} q_j(x)y^j \text{ mit } q_j(x) \in GF(q)[x]$$

Diese Überlegung in Kombination mit der Betrachtung der kumulativen Kerne $D_i = \overline{K_i}$ unserer Nebenbedingungen ermöglicht uns dann einen Algorithmus (Kapitel 3.3.1) zur Berechnung von $Q(x, y)$ zu formulieren und zu beweisen. Dazu betrachten wir die Kerne etwas genauer. Es sei

$$K_i = \{Q(x, y) \in GF(q)_{L_0}[x, y] : D_i(Q) = 0\}$$

mit $GF(q)_{L_0}[x, y] = \{Q(x, y) \in GF(q)[x, y] \text{ mit } \deg_{(0,1)} Q(x, y) \leq L_0\}$.

Wenn wir $\overline{K_0} = GF(q)_{L_0}[x, y]$ initialisieren, dann sind die weiteren kumulativen Kerne $\overline{K_i}, i = 1, \dots, C$ rekursiv definiert als:

$$\begin{aligned} \overline{K_i} &:= \overline{K_{i-1}} \cap K_i = \\ &= K_0 \cap K_1 \cap K_2 \cap \dots \cap K_i = \\ &= \{Q(x, y) \in GF(q)_{L_0}[x, y] : D_1(Q) = D_2(Q) = \dots = D_i(Q) = 0\} \end{aligned}$$

Unsere Aufgabe ist also, ein minimales Element aus $\overline{K_C}$ zu berechnen. Eine Lösung dieses Problems lieferte R. Kötter in [8] und [7]. Im Zuge seiner Arbeit entwickelte er folgenden Algorithmus:

3.3.1 Kötters Interpolationsalgorithmus

Kötters Algorithmus liefert uns das bezüglich einer gegebenen Ordnung kleinstmögliche bivariate Polynom mit m -fachen Nullstellen bei gegebenen Paaren $(\lambda_i, \mu_i)_{i=1}^n$. Also angenommen wir haben die Werte $(\beta_i)_{i=1}^n$ aus $GF(q)$ und ein primitives Element α des Galoisfelds, dann wollen wir, dass unsere Nullstellen die Paare $(\alpha^{i-1}, \beta_i)_{i=1}^n$ sind (warum die Paare genau so lauten, sehen wir in Kapitel 5.1). Um Algorithmus 1 verwenden zu können, müssen wir nur noch einen Wert für den Parameter m wählen, sowie die $(1, v)$ -revlex Ordnung festlegen. Das benötigte L_0 berechnet man sich mittels Definition 21.

Bevor wir zu dem Satz kommen, der uns bestätigt, dass wir mit Kötters Algorithmus das gewünschte Interpolationspolynom bekommen, definieren wir noch:

Definition 22.

$$S_j := \{Q(x, y) \in GF(q)_{L_0}[x, y] : \deg_{(0,1)} \text{LM}(Q(x, y)) = j\}$$

Algorithm 1 Kötters Interpolationsalgorithmus

Input: $(\beta_i)_{i=1}^n, m, (1, v)$ -revlex Ordnung
Output: $Q(x, y) = \min_j \{g_0, g_1, \dots, g_{L_0}\}$
Berechne L_0
Initialisiere $g_j \leftarrow y^j$ für $j = 0, 1, \dots, L_0$
for $i = 1$ bis n **do**
 for $(r, s) = (0, 0)$ bis $(m - 1, 0)$ mit $(m - 1, 1)$ -lex Ordnung **do**
 $\mathcal{J} \leftarrow \{j : \Delta_j = D_{r,s}g_j(\alpha^{i-1}, \beta_i) \neq 0\}$
 if $\mathcal{J} \neq \emptyset$ **then**
 $j_{\min} \leftarrow \arg \min \{g_j : j \in \mathcal{J}\}$
 $f \leftarrow g_{j_{\min}}$
 $\Delta \leftarrow \Delta_{j_{\min}}$
 end if
 for $j \in \mathcal{J}$ **do**
 if $j \neq j_{\min}$ **then**
 $g_j \leftarrow \Delta \cdot g_j - \Delta_j \cdot f$
 else if $j = j_{\min}$ **then**
 $g_j \leftarrow \Delta \cdot (x - \alpha^{i-1}) \cdot f$
 end if
 end for
 end for
end for

Außerdem benötigen wir noch einige Überlegungen:

Bemerkung 9. McEliece verwendet in [11] die Bezeichnung $g_{\rho,j}$ während wir diese Polynome in Algorithmus 1 mit g_j bezeichnen. Diese Notation erleichtert die Beweisführung für Kötters Interpolationsalgorithmus, wie wir später sehen werden. Doch woher kommt ρ ?

Verantwortlich dafür sind die beiden ineinander verschachtelten **for**-Schleifen, sowie der Wunsch den Iterationsschritt zu kennen, in dem wir uns befinden. Während die äußere Schleife für $i = 1, \dots, n$ läuft, startet die innere Schleife immer wieder bei $(r, s) = (0, 0)$ und endet bei $(m - 1, 0)$, wobei diese Tupel entsprechend einer $(m - 1, 1)$ -lex Ordnung (siehe Bemerkung 8) geordnet sind. Wenn wir diese $n \cdot \binom{m+1}{2}$ Schleifendurchläufe fortlaufend mit ρ nummerieren, haben wir eine Möglichkeit zu sagen in welchem Iterationsschritt ρ wir uns befinden. Das Bestimmen von i (und damit auch der Nullstellen (α^{i-1}, β_i)) und dem Tupel (r, s) ist für den Algorithmus jedoch zu umständlich, weshalb wir diesen trotzdem mit den beiden **for**-Schleifen realisieren.

Dank der Verwendung von $g_{\rho,j}$ wissen wir also in welchem Iterationsschritt wir uns befinden. Doch nach der Berechnung von $g_{\rho+1,j}$ aus $g_{\rho,j}$ wird zweites nicht mehr benötigt, daher verwenden wir in Algorithmus 1 nur g_j und überschreiben diese, um zu Gunsten von Speicherplatz darauf zu verzichten die Polynome je Iterationsschritt ρ als $g_{\rho,j}$ zu speichern.

Zusätzlich zur Verwendung von $g_{\rho,j}$ bezeichnen wir für den Beweis von Kötters Interpolationsalgorithmus die Nebenbedingung $D_{r,s}$ und den Kern des Iterationsschritts ρ mit D_ρ und K_ρ .

Bemerkung 10. Algorithmus 1 ist bereits für unsere Ansprüche adaptiert. Den noch unverfälschten Interpolationsalgorithmus von Kötter findet man in [11, Seite 23-24]. Da für den Beweis von Kötters Interpolationsalgorithmus die Berechnung der $g_{\rho,j}$ (siehe Bemerkung 9) leichter nachvollziehbar ist, wollen wir auch die iterative Definition aus [11] verwenden (wobei wir \mathcal{J} , j_{min} und f wie in Algorithmus 1 definiert übernehmen):

$$g_{\rho+1,j} = \begin{cases} g_{\rho,j} & \text{wenn } j \notin \mathcal{J}, \\ [f, g_{\rho,j}]_{D_{\rho+1}} & \text{wenn } j \in \mathcal{J} \text{ und } j \neq j_{min}, \\ [f, xf]_{D_{\rho+1}} & \text{wenn } j = j_{min}. \end{cases}$$

Satz 11. Für Kötters Interpolationsalgorithmus, Algorithmus 1 (allgemeine Formulierung in [11, Seite 23-24]), gilt, dass für alle $\rho = 0, 1, \dots, C = n \cdot \binom{m+1}{2}$ die $g_{\rho,j}$ folgende Eigenschaften haben:

$$g_{\rho,j} = \min\{g : g \in \overline{K_\rho} \cap S_j\} \quad j = 0, \dots, L_0$$

Bemerkung 11. Wie in Bemerkung 10 beschrieben erfolgt die Neuberechnung der $g_{\rho+1,j}$ eigentlich mit Hilfe der Nebenbedingungen $[\cdot, \cdot]_{D_{\rho+1}}$, sofern die $g_{\rho+1,j}$ nicht gleich bleiben (Fall 1). Diese Neuberechnung ist im zweiten Fall in Algorithmus 1 vermutlich nachvollziehbar (vgl. Definition 20), allerdings ist die Berechnung des dritten Falls ($j = j_{min}$) doch etwas speziell, weshalb wir diesen hier etwas genauer ausführen sollten⁹:

$$\begin{aligned} [f, xf]_{D_{\rho+1}} &= [f, xf]_{D_{r,s}} \Big|_{\substack{x=\alpha^{i-1} \\ y=\beta_i}} \\ &= D_{r,s}(f) \Big|_{\substack{x=\alpha^{i-1} \\ y=\beta_i}} xf - D_{r,s}(xf) \Big|_{\substack{x=\alpha^{i-1} \\ y=\beta_i}} f \\ &= D_{r,s}(f(\alpha^{i-1}, \beta_i)) \cdot x \cdot f(x, y) - D_{r,s}(x \cdot f(x, y)) \Big|_{\substack{x=\alpha^{i-1} \\ y=\beta_i}} \cdot f(x, y) \\ &= D_{r,s}(f(\alpha^{i-1}, \beta_i)) \cdot x \cdot f(x, y) - \alpha^{i-1} \cdot D_{r,s}(f(\alpha^{i-1}, \beta_i)) \cdot f(x, y) \\ &= D_{r,s}(f(\alpha^{i-1}, \beta_i)) f(x, y) (x - \alpha^{i-1}) \\ &= \Delta(x - \alpha^{i-1}) f \end{aligned}$$

Wobei wir wissen, dass $D_{r,s}(x \cdot f(x, y)) \Big|_{\substack{x=\alpha^{i-1} \\ y=\beta_i}} = \alpha^{i-1} \cdot D_{r,s}(f(\alpha^{i-1}, \beta_i))$ bei (α^{i-1}, β_i) ist auf Grund des Wachstums von (r, s) und Korollar 2 (siehe Bemerkung 8).

Bemerkung 12. Nach Satz 11 sind die $g_{C,j}$ minimale Polynome aus $\overline{K_C} \cap S_j$ für jedes $j = 0, \dots, L_0$ und wir brauchen uns nur noch das bezüglich der Ordnung kleinste Polynom aussuchen, um das gesuchte Interpolationspolynom zu erhalten. Im Fall von Algorithmus 1 reicht es daher, nach Terminierung des Algorithmus, das bezüglich der $(1, \nu)$ -revlex Ordnung kleinste Element aus $\{g_j \mid j = 0, 1, \dots, L_0\}$ zu bestimmen (siehe **Output** in Algorithmus 1).

Beweis von Satz 11. Wir wollen diesen Satz induktiv nach ρ beweisen. Dazu betrachten wir die rekursive Berechnung der Einträge von $g_{\rho+1,j}$ aus den $g_{\rho,j}$. Wir haben demnach 3 verschiedene Fälle zu betrachten:

⁹Wie wir wissen entspricht jeder Iterationsschritt ρ genau einer i -ten Nullstelle (α^{i-1}, β_i) und einem Tupel (r, s) aus \mathbb{N}^2 .

Zuerst sei jedoch darauf hingewiesen, dass für alle Elemente aus G_0 ¹⁰ die Behauptung des Satzes auf jeden Fall gilt, da die y^j minimal (bezüglich der $(1, \nu)$ -revlex Ordnung) in $\overline{K_0} \cap S_j$ sind. Gehen wir nun davon aus, dass die Aussage für G_ρ gilt und schließen wir auf $G_{\rho+1}$:

Fall 1: $j \notin \mathcal{J}$

In diesem Fall findet keine Neuberechnung statt (g_j wird ja nicht überschrieben). Es ist also (siehe Bemerkung 10):

$$g_{\rho+1,j} = g_{\rho,j}$$

Laut Induktionsvoraussetzung wissen wir, dass $g_{\rho,j} \in \overline{K_\rho} \cap S_j$ und, da $j \notin \mathcal{J}$, dass $D_{\rho+1}g_{\rho,j} = 0$ ist. Daher muss $g_{\rho,j} = g_{\rho+1,j}$ auch Element von $K_{\rho+1}$ sein, woraus folgt, dass $g_{\rho+1,j} \in K_{\rho+1} \cap \overline{K_\rho} \cap S_j = \overline{K_{\rho+1}} \cap S_j$ ist. Nachdem $g_{\rho,j}$ minimal in $\overline{K_\rho} \cap S_j$ ist, muss auch $g_{\rho+1,j} = g_{\rho,j}$ in der kleineren Menge $\overline{K_{\rho+1}} \cap S_j$ minimal sein.

Fall 2: $j \in \mathcal{J}$ und $j \neq j_{\min}$

In diesem Fall berechnen wir uns das neue Polynom g_j , bzw. $g_{\rho+1,j}$ mittels:

$$g_{\rho+1,j} = [f, g_{\rho,j}]_{D_{\rho+1}}$$

Nachdem, laut Induktionsvoraussetzung, $g_{\rho,j}$ und $f \in \overline{K_\rho}$ sind, muss auch deren Linearkombination dies sein. Außerdem wissen wir (siehe Lemma 2), dass $g_{\rho+1,j} \in K_{\rho+1}$ ist. Nachdem, laut Algorithmus, $f < g_{\rho,j}$ bezüglich der $(1, \nu)$ -revlex Ordnung ist, gilt (auch wieder wegen Lemma 2), dass $\text{Rang}(g_{\rho,j}) = \text{Rang}(g_{\rho+1,j})$, woraus folgt, dass $g_{\rho+1,j} \in \overline{K_{\rho+1}} \cap S_j$ sein muss. Genauso wie in Fall 1 gilt nun, nachdem $g_{\rho,j}$ bereits minimal war, dass auch $g_{\rho+1,j}$ minimal sein muss, da $\overline{K_{\rho+1}} \subseteq \overline{K_\rho}$ ist.

Fall 3: $j = j_{\min}$

In diesem Fall berechnen wir uns das neue Polynom g_j , bzw. $g_{\rho+1,j}$ mittels:

$$g_{\rho+1,j} = [f, xf]_{D_{\rho+1}}$$

Wegen Korollar 2, der Reihung der Tupel (r, s) und der Induktionsvoraussetzung, wissen wir, dass $D_\rho(xf) = D_k(f) + \lambda D_\rho(f)$ für ein $k < \rho$, oder aber (falls $r = 0$ war), dass $D_\rho(xf) = \alpha D_\rho(f)$ ist. In beiden Fällen können wir aus $f \in \overline{K_\rho}$ folgern, dass $D_\rho(xf) = 0$ ist, weshalb auch $xf \in \overline{K_\rho}$ ist. Somit muss $g_{\rho+1,j}$, das laut Lemma 2 in $K_{\rho+1}$ liegt und eine Linearkombination zweier Funktionen aus $\overline{K_\rho}$ ist, in $\overline{K_{\rho+1}}$ liegen.

Außerdem ist $f = g_{\rho,j} \in S_j$, weshalb auch $xf \in S_j$ ist. Da $xf > f$ ist, können wir (erneut wegen Lemma 2) folgern, dass $\text{Rang}(g_{\rho+1,j}) = \text{Rang}(xf) = \text{Rang}(xg_{\rho,j})$ ist, womit klar ist, dass $g_{\rho+1,j} \in S_j$ sein muss, da $g_{\rho,j} \in S_j$ ist und nach Multiplikation mit x der $\deg_{(0,1)}(g_{\rho+1,j}) = \deg_{(0,1)}(g_{\rho,j}) = j$ ist.

Bleibt nur noch zu zeigen, dass $g_{\rho+1,j}$ auch minimal in $\overline{K_{\rho+1}} \cap S_j$ ist:

Nehmen wir an es gibt ein $h \in \overline{K_{\rho+1}} \cap S_j$ mit $h < g_{\rho+1,j}$, das unsere Bedingung $D_{\rho+1}(h) = 0$ erfüllt. Dann ist dieses h natürlich auch aus $\overline{K_\rho}$, das ja eine Obermenge von $\overline{K_{\rho+1}}$ ist. Da jedoch f minimal in $\overline{K_\rho} \cap S_j$ ist, muss $f \leq h$ sein. Berücksichtigen

¹⁰Zur leichteren Lesbarkeit definieren wir $G_\rho := \{g_{\rho,j} \mid j = 0, 1, \dots, L_0\}$.

wir jetzt, dass es kein Polynom aus S_j geben kann, dessen Rang zwischen dem von $f = g_{\rho,j}$ und $g_{\rho+1,j} = [xf, f]_{D_{\rho+1}}$ liegt, folgt $\text{Rang}(h) = \text{Rang}(f)$. Durch Normieren (von f zu f^* und h zu h^*) und subtrahieren von f und h erhalten wir ein Polynom $f' = h^* - f^*$ das sowohl $f' < h$, als auch $f' < f$ erfüllt und, da es eine Linearkombination zweier Elemente aus $\overline{K_\rho}$ ist, selbst auch in dieser Menge liegt. Wenn wir uns jetzt in Erinnerung rufen, dass f das kleinste Polynom $\in \overline{K_\rho} \setminus \overline{K_{\rho+1}}$ war, dann sehen wir, dass wir mit f' ein Polynom konstruiert haben, das zum einen kleiner als f ist ($f' < f$), zum anderen $\in \overline{K_\rho}$ sowie wegen $D_{\rho+1}(f') = D_{\rho+1}(h) - D_{\rho+1}(f) = 0 - D_{\rho+1}(f) \neq 0$ (da ja $j_{\min} \in \mathcal{J}$ und $f = g_{\min}$) nicht aus $\overline{K_{\rho+1}}$ ist.

Somit hätten wir einen Widerspruch zur Minimalität von f , was aber Induktionsvoraussetzung war (die $g_{\rho,j}$ sind ja die kleinsten Polynome $\in S_j$, die $D_\rho g_{\rho,j} = 0$ erfüllen und f war minimal bezüglich aller $g_{\rho,j}$ mit $j \in \mathcal{J}$), weshalb $g_{\rho+1,j}$ minimal in $\overline{K_{\rho+1}} \cap S_j$ sein muss. \square

Beispiel 4. Bevor wir uns einem Beispiel zu Kötters Interpolationsalgorithmus widmen, müssen wir noch festlegen, welches Galoisfeld wir verwenden:

Zwar ist $GF(256)$ der interessantere Körper wenn es um die praktische Anwendung geht, doch würde das Beispiel unüberschaubar werden. Wir betrachten daher das kleinere Galoisfeld $GF(16)$ mit dem primitiven Element α und dem irreduziblen Polynom $x^4 + x + 1$. Für die Multiplikation bzw. die Addition gilt daher: $\alpha^{15} = \alpha^0 = 1$ und $\alpha^4 = \alpha + 1$.

Gehen wir vorläufig¹¹ davon aus, dass wir die Werte

$$(\beta_i)_{i=1}^{15} = (\alpha^{10}, \alpha^4, \alpha^{11}, \alpha^3, \alpha^2, \alpha^{10}, \alpha^8, \alpha^2, \alpha^1, \alpha^2, \alpha^6, \alpha^2, \alpha^3, \alpha^{11}, \alpha^{12})$$

haben und wir ein Interpolationspolynom mit 4-fachen Nullstellen bei $(\alpha^{i-1}, \beta_i)_{i=1}^{15}$ haben wollen. Die Ordnung legen wir mit einer (1, 6)-revlex Ordnung fest¹². Betrachten wir die ersten Rechenschritte von Algorithmus 1:

Zuerst werden L_0 berechnet und die g_j für $j = 0, 1, \dots, L_0$ initialisiert¹³:

$$L_0 := \max \left\{ L : \text{Rang}_6(y^L) \leq C \right\} \text{ mit } C = n \cdot \binom{m+1}{2} = 15 \cdot \binom{5}{2} = 150$$

$$L_0 = 6$$

$$G_0 = \{1, y, y^2, y^3, y^4, y^5, y^6\}$$

Für jede Nullstelle $(\alpha^{i-1}, \beta_i)_{i=1}^{15}$ wird jetzt schrittweise überprüft, ob diese eine 4-fache Nullstelle für jedes g_j , $j = 0, 1, \dots, L_0$, ist. Falls nicht werden die g_j , die diese Bedingung nicht erfüllen, erweitert, sodass der Rang möglichst nicht oder in nur geringem Ausmaß wächst und die Bedingung für eine 4-fache Nullstelle erfüllt ist:

Betrachten wir das erste Paar (α^0, α^{10}) . Wir müssen nun für alle $(r, s) = (0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (3, 0)$ und für alle Polynome g_j die Bedingung $D_{r,s} g_j(\alpha^0, \alpha^{10})$ berechnen:

¹¹Wie wir auf diese Werte kommen, sehen wir in Beispiel 7.

¹²Die Wahl von m bzw. v liegen an dem gewählten RS Code, sowie der gewünschten Fehlerkorrektur.

¹³Wie schon im Beweis von Satz 11 verwenden wir die Bezeichnung $G_\rho := \{g_{\rho,j} \mid j = 0, 1, \dots, L_0\}$.

$$\begin{aligned}\Delta_0 &= D_{0,0}g_0(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{0}{0} \cdot a_{0,0} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{0-0} = \alpha^0 \\ \Delta_1 &= D_{0,0}g_1(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{1}{0} \cdot a_{0,0} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{1-0} = \alpha^{10} \\ \Delta_2 &= D_{0,0}g_2(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{2}{0} \cdot a_{0,0} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{2-0} = \alpha^{20} = \alpha^5 \\ \Delta_3 &= D_{0,0}g_3(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{3}{0} \cdot a_{0,0} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{3-0} = \alpha^{30} = \alpha^0 \\ \Delta_4 &= D_{0,0}g_4(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{4}{0} \cdot a_{0,0} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{4-0} = \alpha^{40} = \alpha^{10} \\ \Delta_5 &= D_{0,0}g_5(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{5}{0} \cdot a_{0,0} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{5-0} = \alpha^{50} = \alpha^5 \\ \Delta_6 &= D_{0,0}g_6(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{6}{0} \cdot a_{0,0} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{6-0} = \alpha^{60} = \alpha^0\end{aligned}$$

Nachdem kein einziger Wert Null ist, ist $\mathcal{J} = \{0, 1, \dots, 6\}$ und wir müssen alle Polynome ändern. Dazu wird zuerst festgelegt, dass

$$\begin{aligned}j_{min} &= 0, \text{ da } g_0 \text{ bezüglich der } (1, 6)\text{-revlex Ordnung das kleinste Polynom aus} \\ &G_0 \text{ ist,} \\ f &= g_0 = 1 \text{ und} \\ \Delta &= \Delta_0 = \alpha^0 \text{ ist.}\end{aligned}$$

Alle Polynome bis auf g_0 werden nun, wie im Algorithmus beschrieben, durch $\Delta \cdot g_j - \Delta_j \cdot f$ ersetzt¹⁴:

$$\begin{aligned}g_1 &= \Delta \cdot g_1 - \Delta_1 \cdot f = \alpha^0 \cdot y - \alpha^{10} \cdot 1 = y + \alpha^{10} \\ g_2 &= \Delta \cdot g_2 - \Delta_2 \cdot f = \alpha^0 \cdot y^2 - \alpha^5 \cdot 1 = y^2 + \alpha^5 \\ g_3 &= \Delta \cdot g_3 - \Delta_3 \cdot f = \alpha^0 \cdot y^3 - \alpha^0 \cdot 1 = y^3 + \alpha^0 \\ g_4 &= \Delta \cdot g_4 - \Delta_4 \cdot f = \alpha^0 \cdot y^4 - \alpha^{10} \cdot 1 = y^4 + \alpha^{10} \\ g_5 &= \Delta \cdot g_5 - \Delta_5 \cdot f = \alpha^0 \cdot y^5 - \alpha^5 \cdot 1 = y^5 + \alpha^5 \\ g_6 &= \Delta \cdot g_6 - \Delta_6 \cdot f = \alpha^0 \cdot y^6 - \alpha^0 \cdot 1 = y^6 + \alpha^0\end{aligned}$$

g_0 ersetzen wir durch:

$$g_0 = \Delta \cdot (x - \alpha^0) \cdot f = \alpha^0 \cdot (x - \alpha^0) \cdot 1 = x + \alpha^0$$

Wir erhalten also:

$$G_1 = \{x + 1, y + \alpha^{10}, y^2 + \alpha^5, y^3 + 1, y^4 + \alpha^{10}, y^5 + \alpha^5, y^6 + 1\}$$

Durch diese Festlegung von G_1 hat sich nur der Rang von g_0 vergrößert und dieser auch nur so gering wie möglich. Damit wir jedoch eine 4-fache Nullstelle bei (α^0, α^{10})

¹⁴Da wir in $GF(2^4)$ rechnen, sind alle Elemente additiv selbst invers und wir können daher die Subtraktion durch die Addition ersetzen.

haben, müssen die Δ_j für alle Paare (r, s) Null sein. Betrachten wir noch das zweite Paar $(r, s) = (0, 1)$ ¹⁵:

$$\begin{aligned}\Delta_0 &= D_{0,1}g_0(\alpha^0, \alpha^{10}) = 0 \text{ (leere Summe)} \\ \Delta_1 &= D_{0,1}g_1(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{1}{1} \cdot a_{0,1} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{1-1} = \alpha^0 \\ \Delta_2 &= D_{0,1}g_2(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{2}{1} \cdot a_{0,2} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{2-1} = \alpha^{20} = 0 \\ \Delta_3 &= D_{0,1}g_3(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{3}{1} \cdot a_{0,3} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{3-1} = \alpha^{30} = \alpha^5 \\ \Delta_4 &= D_{0,1}g_4(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{4}{1} \cdot a_{0,4} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{4-1} = \alpha^{40} = 0 \\ \Delta_5 &= D_{0,1}g_5(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{5}{1} \cdot a_{0,5} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{5-1} = \alpha^{50} = \alpha^{10} \\ \Delta_6 &= D_{0,1}g_6(\alpha^0, \alpha^{10}) = \binom{0}{0} \cdot \binom{6}{1} \cdot a_{0,6} \cdot (\alpha^0)^{0-0} \cdot (\alpha^{10})^{6-1} = \alpha^{60} = 0\end{aligned}$$

Wie wir sehen, ist hier $\mathcal{J} = \{1, 3, 5\}$. Wir müssen also lediglich diese drei Polynome korrigieren. Dazu legen wir wieder fest:

$$\begin{aligned}j_{min} &= 1 \\ f &= g_1 = y + \alpha^{10} \\ \Delta &= \Delta_1 = \alpha^0\end{aligned}$$

Zur Bestimmung von G_2 müssen wir jetzt lediglich g_1, g_3 und g_5 neu berechnen:

$$\begin{aligned}g_3 &= \Delta \cdot g_3 - \Delta_3 \cdot f = \alpha^0 \cdot (y^3 + 1) - \alpha^5 \cdot (y + \alpha^{10}) = y^3 + \alpha^5 y \\ g_5 &= \Delta \cdot g_5 - \Delta_5 \cdot f = \alpha^0 \cdot (y^5 + \alpha^5) - \alpha^{10} \cdot (y + \alpha^{10}) = y^5 + \alpha^{10} y \\ g_1 &= \Delta \cdot (x - \alpha^0) \cdot f = \alpha^0 \cdot (x - \alpha^0) \cdot (y + \alpha^{10}) = xy + y + \alpha^{10} x + \alpha^{10}\end{aligned}$$

Wir erhalten:

$$G_2 = \{x + 1, xy + y + \alpha^{10}x + \alpha^{10}, y^2 + \alpha^5, y^3 + \alpha^5 y, y^4 + \alpha^{10}, y^5 + \alpha^{10}y, y^6 + 1\}$$

Führen wir diese Berechnungen für die weiteren Paare (r, s) , sowie für alle Nullstellen $(\alpha^{i-1}, \beta_i)_{i=1}^{15}$ durch, erhalten wir schlussendlich G_{150} mit den Polynomen:

$$\begin{aligned}g_0 &= x^{59} + 1 \\ g_1 &= \alpha^3 + \alpha^9 x + \alpha^1 x^2 + \alpha^9 x^3 + \alpha^6 x^4 + \alpha^8 x^5 + \alpha^7 x^6 + \alpha^3 x^{15} + \alpha^9 x^{16} + \alpha^1 x^{17} \\ &\quad + \alpha^9 x^{18} + \alpha^6 x^{19} + \alpha^8 x^{20} + \alpha^7 x^{21} + \alpha^3 x^{30} + \alpha^9 x^{31} + \alpha^1 x^{32} + \alpha^9 x^{33} \\ &\quad + \alpha^6 x^{34} + \alpha^8 x^{35} + \alpha^7 x^{36} + \alpha^3 x^{45} + \alpha^9 x^{46} + \alpha^1 x^{47} + \alpha^9 x^{48} + \alpha^6 x^{49} \\ &\quad + \alpha^8 x^{50} + \alpha^7 x^{51} + y(\alpha^8 + \alpha^8 x^{15} + \alpha^8 x^{30} + \alpha^8 x^{45}) \\ g_2 &= \dots\end{aligned}$$

¹⁵Zu beachten ist, dass $\binom{\cdot}{\cdot}$ nur 0 oder 1 sein kann, da wir in $GF(2^4)$ rechnen und alle Elemente additiv selbst invers sind.

Alle diese Polynome haben eine 4-fache Nullstelle bei allen $(\alpha^{i-1}, \beta_i)_{i=1}^{15}$. Allerdings wollen wir das vom Rang her kleinstmögliche Polynom haben. Daher ist unser Interpolationspolynom:

$$Q(x, y) = \alpha^9 + \alpha^3 x^4 + \alpha^1 x^8 + \alpha^3 x^{12} + \alpha^6 x^{16} + \alpha^{14} x^{20} + \alpha^{10} x^{24} + \alpha^{14} y^4$$

Die Tatsache, dass bei allen $(\alpha^{i-1}, \beta_i)_{i=1}^{15}$ 4-fachen Nullstellen existieren, lässt sich leicht nachrechnen:

Betrachten wir dazu die erste Nullstelle (α^0, α^{10}) . Damit diese eine 4-fache Nullstelle ist, müssen die Koeffizienten aller Monome $x^r y^s$ von $Q(x + \alpha^0, y + \alpha^{10})$ deren Potenzen $r + s < 4$ erfüllen, Null sein:

$$\begin{aligned} Q(x + \alpha^0, y + \alpha^{10}) &= \alpha^9 + \alpha^3(x + \alpha^0)^4 + \alpha^1(x + \alpha^0)^8 + \alpha^3(x + \alpha^0)^{12} + \alpha^6(x + \alpha^0)^{16} \\ &\quad + \alpha^{14}(x + \alpha^0)^{20} + \alpha^{10}(x + \alpha^0)^{24} + \alpha^{14}(y + \alpha^{10})^4 \\ &= \alpha^9 + \alpha^3(x^4 + \alpha^0) + \alpha^1(x^8 + \alpha^0) + \alpha^3(x^{12} + \alpha^0 x^8 + \alpha^0 x^4 + \alpha^0) \\ &\quad + \alpha^6(x^{16} + \alpha^0) + \alpha^{14}(x^{20} + \alpha^0 x^{16} + \alpha^0 x^4 + \alpha^0) + \alpha^{10}(x^{24} + \alpha^0 x^{16} + \alpha^0 x^8 + \alpha^0) \\ &\quad + \alpha^{14}(y^4 + \alpha^{10}) \\ &= \alpha^9 + \alpha^3 + \alpha^1 + \alpha^3 + \alpha^6 + \alpha^{14} + \alpha^{10} + \alpha^9 + \\ &\quad \text{Terme mit Monomen } x^r y^s \text{ mit } r + s \geq 4 \\ &= 0 + \text{Terme höherer Ordnung} \end{aligned}$$

3.4 Faktorisierung bivariater Polynome

Wie bereits beschrieben, besteht der zweite Teil des GS Algorithmus aus der Faktorisierung eines bivariaten Polynoms. Wir wollen uns nun also überlegen, wie man ein Polynom in x und y bestmöglich faktorisieren kann. Dabei interessieren uns nur Faktoren der Form $y - f(x)$.

Satz 12. Faktorisierungstheorem ([11, Seite 17])

Gegeben seien $f(x) \in GF(q)[x]$ mit $\text{Grad} \leq v$ und $Q(x, y) \in GF(q)[x, y]$. Wenn

$$\sum_{\alpha \in GF(q)} \text{ord}(Q : \alpha, f(\alpha)) > \text{deg}_{(1,v)} Q(x, y)$$

ist, dann ist $y - f(x)$ ein Faktor von $Q(x, y)$.

Dieser Satz sagt also, dass für ein Polynom vom $\text{Grad} \leq v$ (für uns ist im späteren Verlauf $v = k - 1$ wegen der Länge k von Datenwörtern interessant) $y - f(x)$ ein Faktor von $Q(x, y)$ ist, falls die Summe der Vielfachheiten der Nullstellen größer als der $(1, v)$ -gewichtete Grad von $Q(x, y)$ ist. Jetzt sehen wir auch, warum bei der Konstruktion des Interpolationspolynoms dieses bezüglich einer $(1, v)$ -gewichteten Ordnung minimal sein sollte. So wird erreicht, dass die rechte Seite der Ungleichung einen minimalen Wert annimmt. Umgekehrt kann durch Selektion eines größeren Parameters m die linke Seite der Ungleichung vergrößert werden. Allerdings wird dadurch auch der Grad des Polynoms beeinflusst. Dazu später mehr.

Für den Beweis des Faktorisierungstheorems benötigen wir folgende drei Lemmata ([11, Seite 17-18]):

Lemma 3. Wenn $f(x) \in GF(q)[x]$ mit $\text{Grad} \leq v$, dann ist $\deg Q(x, f(x)) \leq \deg_{(1,v)} Q(x, y)$.

Beweis. Da für $f(x) = 0$ die Aussage sofort einzusehen ist, betrachten wir $f(x) \neq 0$:
Für $Q(x, y) = \sum_{(i,j) \in I} a_{i,j} x^i y^j$ sei $\deg_{(1,v)} Q(x, y) = r + v \cdot s$. Dann gilt für alle $(i, j) \in I$ mit $a_{i,j} \neq 0$ und da der Grad von $f(x) \leq v$ ist: $\deg(x^i f(x)^j) \leq i + v \cdot j \leq r + v \cdot s$. Folglich ist auch $\deg Q(x, f(x)) \leq r + v \cdot s = \deg_{(1,v)} Q(x, y)$. \square

Lemma 4. $Q(x, f(x)) = 0$ genau dann wenn $(y - f(x)) | Q(x, y)$

Beweis. Fassen wir $Q(x, y)$ auf als ein Polynom in y über $GF(q)[x]$, also $q_0(x) + q_1(x)y + q_2(x)y^2 + \dots$ mit $q_i(x) \in GF(q)[x]$ für alle $i = 1, 2, \dots$, dann erhalten wir bei Division durch $y - f(x)$:

$Q(x, y) : (y - f(x)) = Q_0(x, y)$ und ein Restpolynom $r(x)$. Also $Q(x, y) = Q_0(x, y)(y - f(x)) + r(x)$. Substitution von y durch $f(x)$ liefert:

$$Q(x, f(x)) = r(x)$$

Nun ist $Q(x, f(x)) = 0$ genau dann, wenn $r(x) = 0$, was äquivalent ist zu $(y - f(x)) | Q(x, y)$. \square

Lemma 5. Wenn $\text{ord}(Q : \lambda, \mu) = K$ und $f(\lambda) = \mu$, dann gilt:

$$(x - \lambda)^K | Q(x, f(x))$$

Beweis. $Q(x, y)$ lässt sich wegen Satz 9 darstellen als $Q(x, y) = \sum_{r,s} Q_{r,s}(\lambda, \mu)(x - \lambda)^r (y - \mu)^s$. Substitution von y durch $f(x)$ und μ durch $f(\lambda)$ liefert nun:

$$Q(x, f(x)) = \sum_{r,s} Q_{r,s}(\lambda, f(\lambda))(x - \lambda)^r (f(x) - f(\lambda))^s$$

Nachdem λ eine Nullstelle von $f(x) - f(\lambda)$ ist, gilt $(x - \lambda) | (f(x) - f(\lambda))$, woraus folgt, dass $(x - \lambda)^s | (f(x) - f(\lambda))^s$. Laut Voraussetzung ist $\text{ord}(Q : \lambda, \mu) = K$, was zur Folge hat, dass wenn $Q_{r,s}(\lambda, f(\lambda)) \neq 0$ ist, $r + s \geq K$ sein muss. Somit ist jeder Term in $Q(x, f(x))$ der ungleich 0 ist durch $(x - \lambda)^K$ teilbar, also $(x - \lambda)^K | Q(x, f(x))$. \square

Nun können wir das Faktorisierungstheorem (Satz 12) beweisen:

Beweis von Satz 12. Dazu betrachten wir:

$$Q(x, f(x)) = \sum_{i,j \geq 0} a_{i,j} x^i f(x)^j$$

Wegen Lemma 5 wissen wir:

$$\prod_{\lambda \in GF(q)} (x - \lambda)^{\text{ord}(Q : \lambda, f(\lambda))} | Q(x, f(x))$$

Nachdem der Grad von $\prod_{\lambda \in GF(q)} (x - \lambda)^{\text{ord}(Q : \lambda, f(\lambda))}$ gleich $\sum_{\lambda \in GF(q)} \text{ord}(Q : \lambda, f(\lambda))$ und dieser laut Voraussetzung größer als der $\deg_{(1,v)} Q(x, y)$, sowie nach Lemma 3 $\deg_{(1,v)} Q(x, y) \geq \deg Q(x, f(x))$ ist, muss $Q(x, f(x)) = 0$ sein. Aus dem Lemma 4 können wir nun folgern:

$$(y - f(x)) | Q(x, y)$$

\square

Wir wollen jetzt die $f(x)$ berechnen, sodass $y - f(x)$ ein Teiler von $Q(x, y)$ ist. Dazu gehen wir rekursiv vor. Zuerst definieren wir:

Definition 23. Wenn $x^m | Q(x, y)$ und $x^{m+1} \nmid Q(x, y)$, dann bezeichnen wir mit

$$\langle\langle Q(x, y) \rangle\rangle := \frac{Q(x, y)}{x^m}$$

Für unsere Rekursion legen wir fest:

$$Q_0(x, y) = \langle\langle Q(x, y) \rangle\rangle$$

und

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots + a_vx^v$$

ist das gesuchte Polynom, dessen Koeffizienten wir uns nacheinander berechnen können:

Lemma 6. Betrachten wir $Q_0(0, y) = q_0 + q_1y + q_2y^2 + \dots$ mit $q_i \in GF(q)$, dann ist a_0 eine Lösung y der Gleichung $Q_0(0, y) = 0$.

Beweis. Nachdem $Q(x, y) = x^m Q_0(x, y)$ ist, und $y - f(x)$ ein Teiler von $Q(x, y)$ ist, muss es auch $Q_0(x, y)$ teilen, weil $GF(q)[x, y]$ ein faktorieller Ring ist und $y - f(x)$ irreduzibel, bzw. prim ist. Somit kann man dieses schreiben als

$$Q_0(x, y) = (y - f(x))T_0(x, y)$$

für ein $T_0(x, y) \in GF(q)[x, y]$. Einsetzen von 0 für x zeigt nun, weil $T_0(0, y) \neq 0$ ist, dass $y = f(0) = a_0$ eine Lösung von $Q_0(0, y) = 0$ sein muss. \square

Um jetzt a_1, a_2, \dots zu erhalten, initialisieren wir zusätzlich $f_0(x) = f(x)$ und betrachten die für $j \geq 1$ definierten Polynomfolgen ([11, Seite 34]):

$$\begin{aligned} f_j(x) &:= (f_{j-1}(x) - f_{j-1}(0))/x = a_j + a_{j+1}x + \dots + a_vx^{v-j} \\ T_j(x, y) &:= Q_{j-1}(x, xy + a_{j-1}) \\ Q_j(x, y) &:= \langle\langle T_j(x, y) \rangle\rangle \end{aligned}$$

R. Roth und G. Ruckenstein haben in ihrer Arbeit [14] nachgewiesen, dass sich mit dieser Rekursion alle Polynome $f(x)$ berechnen lassen, für die $(y - f(x)) | Q(x, y)$ gilt.

3.4.1 Roth-Ruckensteins Faktorisierungsalgorithmus

Der Faktorisierungsalgorithmus von Roth-Ruckenstein [11, Seite 37] (kurz RR Algorithmus), Algorithmus 2, benötigt das zu faktorisierte Polynom $Q(x, y)$, sowie einen Parameter k und berechnet rekursiv alle möglichen Polynome $f(x)$ mit $\text{Grad} \leq k - 1$ (und daher k Koeffizienten), für die $(y - f(x)) | Q(x, y)$ gilt.

Im Zuge des Algorithmus werden alle Möglichkeiten für die Koeffizienten a_j , $j = 0, 1, \dots, k - 1$, von $f(x)$ überprüft. Dazu werden im ersten Iterationsschritt alle potentiellen a_0 berechnet. Im zweiten Iterationsschritt werden für das erste mögliche a_0 alle potentiellen a_1 berechnet. Für das erste a_0 und a_1 bestimmt der dritte Iterationsschritt

widerum alle Möglichkeiten für a_2 u.s.w.

Um im weiteren Verlauf die Polynome $f(x)$ bestimmen zu können, speichern wir alle Informationen in Arrays. Wir nummerieren dabei die potentiellen Koeffizienten fortlaufend mit $v = 1, 2, \dots$ und halten in $K(v)$ den Wert des Koeffizienten, in $\pi(v)$ die Nummern u seines Vorgängers und in $\delta(v)$ den Grad von x in $f(x)$, zu dem der aktuelle Koeffizient gehört, fest. Wir initialisieren außerdem $\pi(0) = \text{“nichts“}$ und $\delta(0) = -1$. Ersteres brauchen wir als Abbruchbedingung bei der Berechnung von $f_u(x)$, denn sollte die Bedingung $Q_u(x, 0) = 0$ für den aktuellen Koeffizienten $K(u)$ erfüllt sein, bestimmen wir das potentielle Polynom $f(x)$, das wir, da wir aktuell beim u -ten Koeffizienten sind, mit $f_u(x)$ bezeichnen, rekursiv mittels

$$f_u(x) = K(u)x^{\delta(u)} + K(\pi(u))x^{\delta(\pi(u))} + K(\pi(\pi(u)))x^{\delta(\pi(\pi(u)))} + \dots$$

Algorithm 2 Roth-Ruckensteins Faktorisierungsalgorithmus

Input: $Q(x, y)$, k

Output: Outputliste $\{f(x) : (y - f(x)) \mid Q(x, y) \text{ und } \deg f(x) \leq k - 1\}$

Initialisiere:

Outputliste $\leftarrow \emptyset$

$v = 0$

$\pi(v) \leftarrow \text{NIL}$

$\delta(v) \leftarrow -1$

$t \leftarrow 1$

$Q_0(x, y) \leftarrow \langle\langle Q(x, y) \rangle\rangle$

if $Q_0(x, 0) = 0$ **then**

Outputliste $\cup \{f(x) = 0\}$

$Q_0(x, y) \leftarrow \frac{Q_0(x, y)}{y^r}$ für maximales r

end if

Rufe $TS(v)$ auf % $TS \dots$ Tiefensuche

Funktion $TS(u)$

if $Q_u(x, 0) = 0$ **then**

Outputliste $\cup \{f_u(x)\}$

else if $\delta(u) < k - 1$ **then**

$N = \{\text{Nullstellen von } Q_u(0, y)\}$

for all $n \in N$ **do**

$v \leftarrow t$

$t \leftarrow t + 1$

$\pi(v) \leftarrow u$

$\delta(v) \leftarrow \delta(u) + 1$

$K(v) \leftarrow n$

$Q_v(x, y) = \langle\langle Q_u(x, xy + n) \rangle\rangle$

Rufe $TS(v)$ auf

end for

end if

Die Initialisierung $\delta(0) = -1$ verwenden wir um den zu den möglichen Koeffizienten gehörenden Grad zu berechnen. Im ersten Schritt werden für $v = 1$ die Arrays mit $K(1) = a_0$ (dem ersten der möglichen Koeffizienten für a_0), $\pi(1) = 0$ und $\delta(1) = 0$

befüllt.

Zusätzlich zur bereits erwähnten Bedingung $Q_u(x, 0) = 0$ wird die rekursive Berechnung weiterer Koeffizienten gestoppt, falls wir den Grad $\delta(\cdot) = k - 1$ erreichen, da $f(x)$ maximal Grad $k - 1$ haben darf. Statt dessen werden weitere Kombinationsmöglichkeiten der Koeffizienten betrachtet und damit auch neue potentielle Koeffizienten berechnet.

Um einzusehen, dass dieser Algorithmus funktioniert, müssen wir uns allerdings noch mit ein paar Aussagen beschäftigen. Der nächste Satz ([11, Seite 34]) liefert uns einen wichtigen Zusammenhang zwischen $f(x)$ und den $f_j(x)$ mit dessen Hilfe wir im Anschluss die Berechnung der a_j zeigen werden:

Satz 13. Betrachten wir eine Funktion $f(x) = a_0 + a_1x + \dots + a_vx^v \in GF(q)[x]$ und ein Polynom $Q(x, y) \in GF(q)[x, y]$, dann gilt für die eben definierten Polynome, dass für alle $j \geq 1$

$$(y - f(x)) \mid Q(x, y) \Leftrightarrow (y - f_j(x)) \mid Q_j(x, y)$$

Beweis. Für den Beweis zeigen wir die analoge Aussage, dass $(y - f_j(x)) \mid Q_j(x, y) \Leftrightarrow (y - f_{j-1}(x)) \mid Q_{j-1}(x, y)$

\Rightarrow : Um die erste Richtung zu beweisen nehmen wir an $(y - f_j(x)) \mid Q_j(x, y)$. Dann wissen wir auf Grund der Definition von $Q_j(x, y)$, dass $T_j(x, y) = x^m Q_j(x, y)$ für ein passendes m gilt. Also gilt $(y - f_j(x)) \mid Q_{j-1}(x, xy + a_{j-1}) (= T_j(x, y))$. Es muss also $Q_{j-1}(x, xy + a_{j-1}) = (y - f_j(x))U(x, y)$ für ein passendes $U(x, y)$ sein. Um auf die gewünschte Teilbarkeit schließen zu können, substituieren wir $y = \frac{y - a_{j-1}}{x}$:

$$\begin{aligned} Q_{j-1}(x, y) &= \left(\frac{y - a_{j-1}}{x} - f_j(x) \right) U\left(x, \frac{y - a_{j-1}}{x}\right) = \\ &= \frac{1}{x} (y - a_{j-1} - x f_j(x)) \frac{1}{x^M} V(x, y) = \\ &= \frac{1}{x} (y - f_{j-1}(x)) \frac{1}{x^M} V(x, y) \end{aligned}$$

Da es sich bei $U(x, y)$ um ein endliches Polynom handelt, gibt es eine maximale M für das $U\left(x, \frac{y - a_{j-1}}{x}\right) = \frac{1}{x^M} V(x, y)$ ist. Wenn wir diese Gleichung nun mit x^{M+1} multiplizieren, erhalten wir $x^{M+1} Q_{j-1}(x, y) = (y - f_{j-1}(x)) V(x, y)$ woraus $(y - f_{j-1}(x)) \mid Q_{j-1}(x, y)$ folgt, da $(y - f_{j-1}(x)) \nmid x^{M+1}$ und $y - f_{j-1}(x)$ prim ist.

\Leftarrow : Zum Beweis der Rückrichtung gehen wir von $(y - f_{j-1}(x)) \mid Q_{j-1}(x, y)$ aus. D.h., dass $Q_{j-1}(x, y) = (y - f_{j-1}(x))U(x, y)$ für ein passendes $U(x, y)$ gilt. Nachdem $Q_j(x, y) = \langle\langle T_j(x, y) \rangle\rangle = \langle\langle Q_{j-1}(x, xy + a_{j-1}) \rangle\rangle$ ist, können wir folgern:

$$\begin{aligned} Q_j(x, y) &= \langle\langle (xy + a_{j-1} - f_{j-1}(x))U(x, xy + a_{j-1}) \rangle\rangle \\ &= \langle\langle x(y - f_j(x))U(x, xy + a_{j-1}) \rangle\rangle \\ &= (y - f_j(x)) \langle\langle xU(x, xy + a_{j-1}) \rangle\rangle \end{aligned}$$

Das heißt also, dass $(y - f_j(x)) \mid Q_j(x, y)$ □

Mit der Äquivalenz in diesem Satz, sowie der Methode zur Berechnung von a_0 können wir nun alle Koeffizienten von $f(x)$ berechnen ([11, Seite 35]):

Korollar 3. Wenn $(y-f(x)) \mid Q(x, y)$, dann sind die Koeffizienten a_j von $f(x)$ Lösungen der Gleichungen $Q_j(0, y) = 0$ für $j = 0, \dots, v$.

Beweis. Wegen Satz 13 folgt aus $(y-f(x)) \mid Q(x, y)$, dass $(y-f_j(x)) \mid Q_j(x, y)$. Wenn wir hier nun $x = 0$ setzen, erhalten wir $(y-a_j) = (y-f_j(0)) \mid Q_j(0, y)$ woraus die Aussage des Korollar folgt. \square

Die Umkehrung dieser Aussage gilt natürlich nicht. Wir müssen also herausfinden, welche der durch Lösen der Gleichungen $Q_j(0, y) = 0$ gewonnenen a_j das gesuchte Polynom $f(x)$ ergeben. Folgendes Korollar hilft hierbei ([11, Seite 35]):

Korollar 4. Wenn $y \mid Q_{v+1}(x, y)$, also wenn $Q_{v+1}(x, 0) = 0$, dann gilt für $f(x) = a_0 + a_1x + \dots + a_vx^v$ (dessen Koeffizienten wie soeben beschrieben berechnet wurden), dass $(y-f(x)) \mid Q(x, y)$.

Beweis. Aus unserer Rekursion der $f_j(x)$ wissen wir, dass $f_{v+1}(x) = 0$ sein muss. D.h., dass die Aussage $y \mid Q_{v+1}(x, y)$ dasselbe ist wie $(y-f_{v+1}(x)) \mid Q_{v+1}(x, y)$. Wegen Satz 13 können wir nun folgern dass $(y-f(x)) \mid Q(x, y)$. \square

Somit können wir Polynome $f(x)$ iterativ berechnen, für die $(y-f(x)) \mid Q(x, y)$ gilt, vorausgesetzt, die Bedingung $\sum_{\alpha \in GF(q)} \text{ord}(Q : \alpha, f(\alpha)) > \deg_{(1,v)} Q(x, y)$ ist erfüllt. Der in Matlab geschriebene Algorithmus (siehe Anhang) berechnet alle diese Polynome.

Das Zusammenspiel von Algorithmus 1 und Algorithmus 2 sehen wir dann in Kapitel 5.1. Wie sich aber jetzt schon vermuten lässt, ist die Wahl eines möglichst kleinen bivariaten Polynoms $Q(x, y)$ im Interpolationsteil eine gute Voraussetzung (siehe Bedingung) für den Faktorisierungsteil. Den Zusammenhang zwischen den α^i und den β_i sehen wir dann in Kapitel 4.1.

Beispiel 5. Auch in diesem Beispiel verwenden wir, wie in Beispiel 4, das Galoisfeld $GF(16)$ mit dem primitiven Element α mit den Eigenschaften $\alpha^{15} = \alpha^0 = 1$ und $\alpha^4 = \alpha + 1$. Wir wollen mit Hilfe des Roth-Ruckenstein Algorithmus alle Faktoren $y-f(x)$ berechnen, die das Interpolationspolynom aus Beispiel 4 teilen und Grad ≤ 6 haben. Für Algorithmus 2 legen wir daher fest, dass $k = 7$ und

$$Q(x, y) = \alpha^9 + \alpha^3x^4 + \alpha^1x^8 + \alpha^3x^{12} + \alpha^6x^{16} + \alpha^{14}x^{20} + \alpha^{10}x^{24} + \alpha^{14}y^4$$

ist. Des weiteren werden initialisiert:

$$\begin{aligned} v &= 0 \\ \pi(0) &= \text{NIL} \\ \delta(0) &= -1 \\ t &= 1 \end{aligned}$$

Nachdem $Q(x, y)$ nicht durch x teilbar ist, ist $Q_0(x, y) = Q(x, y)$. Und da $Q_0(x, 0) \neq 0$ ist, starten wir gleich mit der Tiefensuche und rufen $TS(0)$ auf:

So lange $Q_u(x, 0) \neq 0$ und $\delta(u) < k - 1$ ist, wird die Funktion $TS(v)$ immer wieder aufgerufen:

$TS(u)$	$\delta(u)$	Nullstellen	v	t	$\pi(v)$	$\delta(v)$	$K(v)$	Aufruf
$TS(0)$	-1	$\{\alpha^{10}\}$	1	2	0	0	α^{10}	$TS(1)$
$TS(1)$	0	$\{\alpha^1\}$	2	3	1	1	α^1	$TS(2)$
$TS(2)$	1	$\{\alpha^8\}$	3	4	2	2	α^8	$TS(3)$
$TS(3)$	2	$\{\alpha^1\}$	4	5	3	3	α^1	$TS(4)$
$TS(4)$	3	$\{\alpha^{13}\}$	5	6	4	4	α^{13}	$TS(5)$
$TS(5)$	4	$\{\alpha^0\}$	6	7	5	5	α^0	$TS(6)$
$TS(6)$	5	$\{\alpha^{14}\}$	7	8	6	6	α^{14}	$TS(7)$
$TS(7)$	6							

Beim letzten Funktionsaufruf ($TS(7)$) ist es dann so weit und wir sehen, dass für

$$Q_7(x, y) = \alpha^{14}y^4$$

die Bedingung $Q_u(x, 0) = 0$ erfüllt ist¹⁶. Wir können daher unser (erstes und einziges) Polynom

$$f_u(x) = K(u)x^{\delta(u)} + K(\pi(u))x^{\delta(\pi(u))} + K(\pi(\pi(u)))x^{\delta(\pi(\pi(u)))} + \dots$$

für das $(y - f_u(x)) \mid Q(x, y)$ gilt, zusammenstellen:

$$\begin{aligned} f_7(x) &= K(7)x^{\delta(7)} + K(\pi(7))x^{\delta(\pi(7))} + K(\pi(\pi(7)))x^{\delta(\pi(\pi(7)))} + \dots \\ &= \alpha^{14}x^6 + K(6)x^{\delta(6)} + K(\pi(6))x^{\delta(\pi(6))} + \dots \\ &= \alpha^{14}x^6 + \alpha^0x^5 + K(5)x^5 + K(\pi(5))x^{\delta(\pi(5))} + \dots \\ &= \dots \\ &= \alpha^{14}x^6 + \alpha^0x^5 + \alpha^{13}x^4 + \alpha^1x^3 + \alpha^8x^2 + \alpha^1x^1 + \alpha^{10} \end{aligned}$$

Im letzten Schritt brechen wir ab, da wir mit $K(\pi(1))x^{\delta(\pi(1))}$ auf den Term $K(0)x^{\delta(0)} = \text{NIL } x^{-1}$ verweisen und dieser, da nicht existent, nicht mehr Teil des Polynoms ist.

Da dieses Beispiel keinerlei Sonderfälle beinhaltet, betrachten wir noch ein weiteres:

Beispiel 6. Wir wollen die gleiche Aufgabenstellung wie in den Beispielen 4 und 5 lösen, allerdings gehen wir dieses Mal von den Werten

$$(\beta)_{i=1}^{15} = (\alpha^{13}, \alpha^9, \alpha^7, \alpha^9, \alpha^2, \alpha^{10}, \alpha^8, \alpha^2, \alpha^0, \alpha^2, \alpha^6, \alpha^2, \alpha^3, \alpha^{11}, \alpha^{12})$$

aus. Das zu diesen Werten passende Interpolationspolynom liefert uns Kötters Interpo-

¹⁶Eine weitere Berechnung wäre bei diesem Funktionsaufruf nicht erfolgt, da $\delta(7) = 6 \geq 6 = k - 1$ ist. Der Algorithmus terminiert daher auf jeden Fall, sollte jedoch niemals $Q_u(x, 0) = 0$ sein, ist die Outputliste leer und es gibt kein Polynom $f(x)$ mit $\text{Grad} \leq k - 1$, für das $(y - f(x)) \mid Q(x, y)$ gilt.

lationsalgorithmus (siehe Beispiel 4), Algorithmus 1:

$$\begin{aligned}
Q(x, y) = & (\alpha^3 x^0 + \alpha^1 x^1 + \alpha^0 x^2 + \alpha^6 x^4 + \alpha^2 x^5 + \alpha^5 x^6 + \alpha^1 x^7 + \alpha^3 x^9 + \alpha^{13} x^{10} + \alpha^7 x^{11} \\
& + \alpha^{12} x^{12} + \alpha^5 x^{13} + \alpha^3 x^{14} + \alpha^5 x^{15} + \alpha^9 x^{16} + \alpha^9 x^{17} + \alpha^0 x^{18} + \alpha^1 x^{19} \\
& + \alpha^{14} x^{20} + \alpha^{14} x^{21} + \alpha^{13} x^{22} + \alpha^{12} x^{23} + \alpha^9 x^{24} + \alpha^1 x^{25} + \alpha^{13} x^{26} + \alpha^{14} x^{27} \\
& + \alpha^1 x^{29} + \alpha^{11} x^{30} + \alpha^6 x^{31} + \alpha^6 x^{32} + \alpha^6 x^{33} + \alpha^9 x^{34} + \alpha^{12} x^{35} + \alpha^{14} x^{36} \\
& + \alpha^8 x^{37} + \alpha^{12} x^{38} + \alpha^5 x^{39}) \\
& + y \cdot (\alpha^7 + \alpha^6 x^1 + \alpha^8 x^2 + \alpha^4 x^3 + \alpha^{10} x^4 + \alpha^3 x^5 + \alpha^2 x^6 + \alpha^{14} x^7 + \alpha^8 x^8 \\
& + \alpha^4 x^9 + \alpha^5 x^{10} + \alpha^{10} x^{11} + \alpha^{10} x^{12} + \alpha^1 x^{13} + \alpha^8 x^{14} + \alpha^1 x^{17} + \alpha^{10} x^{18} \\
& + \alpha^{13} x^{19} + \alpha^8 x^{20} + \alpha^6 x^{21} + \alpha^2 x^{22} + \alpha^{12} x^{23} + \alpha^{10} x^{24} + \alpha^3 x^{25} + \alpha^1 x^{26} \\
& + \alpha^9 x^{27} + \alpha^5 x^{28} + \alpha^9 x^{29} + \alpha^7 x^{30} + \alpha^7 x^{31} + \alpha^5 x^{32} + \alpha^{12} x^{33}) \\
& + y^2 \cdot (\alpha^2 + \alpha^{12} x^1 + \alpha^{13} x^2 + \alpha^2 x^3 + \alpha^{14} x^4 + \alpha^1 x^5 + \alpha^4 x^6 + \alpha^{14} x^7 + \alpha^{13} x^8 \\
& + \alpha^8 x^9 + \alpha^2 x^{10} + \alpha^{13} x^{11} + \alpha^2 x^{12} + \alpha^2 x^{13} + \alpha^3 x^{14} + \alpha^0 x^{15} + \alpha^5 x^{16} \\
& + \alpha^{14} x^{17} + \alpha^5 x^{18} + \alpha^{12} x^{19} + \alpha^7 x^{20} + \alpha^4 x^{21} + \alpha^3 x^{22} + \alpha^5 x^{23} + \alpha^{12} x^{24} \\
& + \alpha^{10} x^{25} + \alpha^9 x^{26} + \alpha^{11} x^{27}) \\
& + y^3 \cdot (\alpha^{12} + \alpha^7 x^1 + \alpha^2 x^2 + \alpha^4 x^3 + \alpha^4 x^4 + \alpha^{10} x^5 + \alpha^4 x^6 + \alpha^{12} x^{15} + \alpha^7 x^{16} \\
& + \alpha^2 x^{17} + \alpha^4 x^{18} + \alpha^4 x^{19} + \alpha^{10} x^{20} + \alpha^4 x^{21}) \\
& + y^4 \cdot (\alpha^0 + \alpha^9 x^1 + \alpha^{10} x^2 + \alpha^2 x^3 + \alpha^8 x^4 + \alpha^7 x^5 + \alpha^8 x^6 + \alpha^7 x^7 + \alpha^8 x^8 \\
& + \alpha^3 x^9 + \alpha^1 x^{10} + \alpha^{13} x^{11} + \alpha^4 x^{12} + \alpha^0 x^{13} + \alpha^{10} x^{14}) \\
& + y^5 \cdot (\alpha^{10} + \alpha^9 x^2 + \alpha^3 x^3 + \alpha^{10} x^4 + \alpha^{14} x^5 + \alpha^1 x^6 + \alpha^4 x^7 + \alpha^4 x^8) \\
& + y^6 \cdot (\alpha^{11} + \alpha^{13} x^1 + \alpha^7 x^2)
\end{aligned}$$

Für dieses wollen wir wieder mit dem Roth-Ruckenstein Faktorisierungsalgorithmus (siehe Beispiel 5), Algorithmus 2, alle Faktoren $y - f(x)$ mit $(y - f(x)) \mid Q(x, y)$ und $\deg(f(x)) \leq 6$ bestimmen. Dazu initialisieren wir erneut $v = 0$, $\pi(0) = \text{NIL}$, $\delta(0) = -1$ und $t = 1$ und starten die Tiefensuche:

$TS(u)$	$\delta(u)$	Nullstellen	v	t	$\pi(v)$	$\delta(v)$	$K(v)$	Aufruf
$TS(0)$	-1	$\{\alpha^2, \alpha^{10}, \alpha^{11}, \alpha^0\}$	1	2	0	0	α^2	$TS(1)$
$TS(1)$	0	$\{\alpha^7\}$	2	3	1	1	α^7	$TS(2)$
$TS(2)$	1	$\{\alpha^{13}\}$	3	4	2	2	α^{13}	$TS(3)$
$TS(3)$	2	$\{\alpha^{10}\}$	4	5	3	3	α^{10}	$TS(4)$
$TS(4)$	3	$\{\alpha^9\}$	5	6	4	4	α^9	$TS(5)$
$TS(5)$	4	$\{\alpha^6\}$	6	7	5	5	α^6	$TS(6)$
$TS(6)$	5	$\{\alpha^{11}\}$	7	8	6	6	α^{11}	$TS(7)$
$TS(7)$	6							

Bis auf den ersten Funktionsaufruf $TS(0)$ ähnelt diese Aufgabe sehr dem vorigen Beispiel. Auch jetzt, beim Aufruf $TS(7)$ ist die Bedingung $Q_7(x, 0) = 0$ erfüllt und wir erhalten unser (erstes) Polynom $f_7(x)$:

$$f_7(x) = \alpha^{11} x^6 + \alpha^6 x^5 + \alpha^9 x^4 + \alpha^{10} x^3 + \alpha^{13} x^2 + \alpha^7 x^1 + \alpha^2$$

Doch noch sind wir nicht fertig, da wir im Funktionsaufruf $TS(0)$ nur α^2 betrachtet haben. Es muss aber für alle Nullstellen $TS(v)$ aufgerufen werden. Daher setzen wir fort mit:

$TS(u)$	$\delta(u)$	Nullstellen	v	t	$\pi(v)$	$\delta(v)$	$K(v)$	Aufruf
$TS(0)$	-1	$\{\alpha^2, \alpha^{10}, \alpha^{11}, \alpha^0\}$	8	9	0	0	α^{10}	$TS(8)$
$TS(8)$	0	$\{\alpha^1\}$	9	10	8	1	α^1	$TS(9)$
$TS(9)$	1	$\{\alpha^8\}$	10	11	9	2	α^8	$TS(10)$
$TS(10)$	2	$\{\alpha^1\}$	11	12	10	3	α^1	$TS(11)$
$TS(11)$	3	$\{\alpha^{13}\}$	12	13	11	4	α^{13}	$TS(12)$
$TS(12)$	4	$\{\alpha^0\}$	13	14	12	5	α^0	$TS(13)$
$TS(13)$	5	$\{\alpha^{14}\}$	14	15	13	6	α^{14}	$TS(14)$
$TS(14)$	6							

Und erneut liefert uns die Bedingung $Q_{14}(x, 0) = 0$ ein weiteres Polynom:

$$f_{14}(x) = \alpha^{14}x^6 + \alpha^0x^5 + \alpha^{13}x^4 + \alpha^1x^3 + \alpha^8x^2 + \alpha^1x^1 + \alpha^{10}$$

Wir setzen fort mit der nächsten Nullstelle:

$TS(u)$	$\delta(u)$	Nullstellen	v	t	$\pi(v)$	$\delta(v)$	$K(v)$	Aufruf
$TS(0)$	-1	$\{\alpha^2, \alpha^{10}, \alpha^{11}, \alpha^0\}$	15	16	0	0	α^{11}	$TS(15)$
$TS(15)$	0	$\{\alpha^1\}$	16	17	15	1	α^1	$TS(16)$
$TS(16)$	1	$\{\alpha^9\}$	17	18	16	2	α^9	$TS(17)$
$TS(17)$	2	$\{\alpha^6\}$	18	19	17	3	α^6	$TS(18)$
$TS(18)$	3	$\{\alpha^9\}$	19	20	18	4	α^9	$TS(19)$
$TS(19)$	4	$\{\alpha^0\}$	20	21	19	5	α^0	$TS(20)$
$TS(20)$	5	$\{\alpha^6\}$	21	22	20	6	α^6	$TS(21)$
$TS(21)$	6							

Und auch hier liefert uns die Bedingung $Q_{21}(x, 0) = 0$ ein weiteres Polynom:

$$f_{21}(x) = \alpha^6x^6 + \alpha^0x^5 + \alpha^9x^4 + \alpha^6x^3 + \alpha^9x^2 + \alpha^1x^1 + \alpha^{11}$$

Betrachten wir nun die letzte Nullstelle α^0 :

$TS(u)$	$\delta(u)$	Nullstellen	v	t	$\pi(v)$	$\delta(v)$	$K(v)$	Aufruf
$TS(0)$	-1	$\{\alpha^2, \alpha^{10}, \alpha^{11}, \alpha^0\}$	22	23	0	0	α^0	$TS(22)$
$TS(22)$	0	$\{\alpha^9\}$	23	24	22	1	α^9	$TS(23)$
$TS(23)$	1	$\{\alpha^1\}$	24	25	23	2	α^1	$TS(24)$
$TS(24)$	2	$\{\alpha^5\}$	25	26	24	3	α^5	$TS(25)$
$TS(25)$	3	$\{\alpha^6\}$	26	27	25	4	α^6	$TS(26)$
$TS(26)$	4	$\{\alpha^1\}$	27	28	26	5	α^1	$TS(27)$
$TS(27)$	5	$\{\alpha^4\}$	28	29	27	6	α^4	$TS(28)$
$TS(28)$	6							

Erneut haben wir den Fall das $\delta(u) \geq 6$, weshalb wir wieder abbrechen müssen, doch dieses Mal liefert uns die vorherige Überprüfung von $Q_u(x, 0)$ nicht Null, weshalb

wir diesen letzten Zweig beenden ohne eine weitere Lösung $f_u(x)$ gefunden zu haben:

$$\begin{aligned} Q_{28}(x, 0) = & \alpha^6 + \alpha^5 x^1 + \alpha^7 x^2 + \alpha^8 x^3 + \alpha^{13} x^4 + \alpha^{10} x^5 + \alpha^6 x^6 + \alpha^8 x^7 + \alpha^{13} x^8 + \alpha^{14} x^9 \\ & + \alpha^4 x^{10} + \alpha^{13} x^{11} + \alpha^3 x^{12} + \alpha^{12} x^{13} + \alpha^6 x^{14} + \alpha^7 x^{15} + \alpha^{10} x^{17} + \alpha^0 x^{18} \\ & + \alpha^{12} x^{20} + \alpha^{11} x^{21} + \alpha^4 x^{22} + \alpha^5 x^{23} + \alpha^8 x^{24} + \alpha^8 x^{25} + \alpha^9 x^{26} + \alpha^4 x^{27} \\ & + \alpha^{14} x^{28} + \alpha^0 x^{29} + \alpha^7 x^{30} + \alpha^6 x^{31} + \alpha^8 x^{32} \\ & \neq 0 \end{aligned}$$

Algorithmus 2 liefert uns also drei verschiedene Polynome $f(x)$ mit $\text{Grad} \leq 6$ für die $(y - f(x)) \mid Q(x, y)$ gilt:

$$\begin{aligned} \text{Outputliste} = \{ & \alpha^{11} x^6 + \alpha^6 x^5 + \alpha^9 x^4 + \alpha^{10} x^3 + \alpha^{13} x^2 + \alpha^7 x^1 + \alpha^2, \\ & \alpha^{14} x^6 + \alpha^0 x^5 + \alpha^{13} x^4 + \alpha^1 x^3 + \alpha^8 x^2 + \alpha^1 x^1 + \alpha^{10}, \\ & \alpha^6 x^6 + \alpha^0 x^5 + \alpha^9 x^4 + \alpha^6 x^3 + \alpha^9 x^2 + \alpha^1 x^1 + \alpha^{11} \} \end{aligned}$$

4 Reed-Solomon Codes

Reed-Solomon Codes (kurz RS Codes) sind lineare Blockcodes über endlichen Körpern $\mathbb{K} = GF(q)$. Ein Datenwort bestehend aus $k \leq q - 1$ (sinnvollerweise ist $k < q - 1$) Buchstaben wird auf ein Codewort mit $n = q - 1$ Buchstaben ausgeweitet¹⁷. Hier sieht man bereits einen Nachteil: Da wir mit endlichen Körpern arbeiten, können wir größere Datenmengen nur dann codieren, wenn wir diese zuerst in Blöcke der Länge k unterteilen. Einen derartigen Block nennen wir ein Datenwort. Nehmen wir an, wir möchten ein Datenwort $(b_i)_{i=1}^k$ mit $b_i \in GF(q)$ codieren:

Für diese Aufgabe gibt es bei RS Codes zwei Möglichkeiten, die zwar zum gleichen Code¹⁸, aber auf Grund der verschiedenen Zugänge zu unterschiedlichen Decodierungsverfahren führen. Zuerst betrachten wir das „klassische“ Codierungsverfahren, das wir mit Hilfe des Guruswami-Sudan Algorithmus (dazu verwenden wir sowohl Kötters Algorithmus aus Kapitel 3.3.1, als auch den Roth-Ruckenstein Algorithmus aus Kapitel 3.4.1) decodieren können. Danach betrachten wir noch die Codierung mit Hilfe des Generatorpolynoms, die wir im Kapitel 7.1 noch adaptieren werden. Zur Decodierung bei dieser Methode verwenden wir den Berlekamp-Massey Algorithmus.

4.1 Codieren durch Auswertung bei α^i

Angenommen wir haben ein Datenwort $(b_i)_{i=1}^k$ mit k Buchstaben $b_i \in GF(q)$, einem zuvor festgelegten Alphabet. Wir definieren ein Polynom $f_b(x) := b_1 + b_2 x + b_3 x^2 + \dots + b_k x^{k-1}$, wobei die Koeffizienten die Buchstaben des Datenworts sind. Weiters wählen wir ein primitives Element α aus $GF(q)$.

Das Codewort erhalten wir, indem wir die Potenzen $\alpha^i, i = 0, 1, \dots, n-1 = q-2$, in das Polynom $f_b(x)$ einsetzen. Das heißt unser Codewort $\chi = f_b(\alpha^0) f_b(\alpha^1) \dots f_b(\alpha^{n-1})$

¹⁷wobei dieses n nicht zu verwechseln ist mit der Primzahlpotenz \bar{n} von $q = p^{\bar{n}}$

¹⁸Die beiden Methoden codieren dasselbe Datenwort $\in GF(q)^k$ zu zwei verschiedenen Codewörtern $\in GF(q)^{q-1}$, allerdings erhält man nach Codierung aller zur Verfügung stehenden Datenwörter in beiden Fällen die gleiche Menge an Codewörtern.

$= \chi_1 \chi_2 \dots \chi_n$ ist eigentlich ein n -Tupel von Auswertungen eines Polynoms vom Grad $k - 1$. Es ist also:

$$\chi_i = f_b(\alpha^{i-1}) \text{ für } i = 1, 2, \dots, n$$

Eine wichtige Eigenschaft des RS Codes ist:

Satz 14. RS Codes sind MDS-Codes, d.h. die Minimaldistanz ist $d = n - k + 1$. Es unterscheiden sich daher zwei Codewörter immer an mindestens $n - k + 1$ Stellen.

Beweis. Nehmen wir an es gibt zwei verschiedene Codewörter $(\chi_i)_{i=1}^n$ und $(\zeta_i)_{i=1}^n$ zu unterschiedlichen Datenwörtern $(b_i)_{i=1}^k$ und $(c_i)_{i=1}^k$, die uns die Polynome $f_b(\cdot)$ und $f_c(\cdot)$ liefern.

Würden sich die Codewörter $(\chi_i)_{i=1}^n$ und $(\zeta_i)_{i=1}^n$ an weniger als $n - k + 1$ Stellen unterscheiden, dann hieße das, dass $\chi_i = \zeta_i$ an mindestens k Stellen ist. Mit anderen Worten: Es muss $f_b(\alpha^i) = f_c(\alpha^i)$ für mindestens k verschiedene Potenzen von α gelten (die ja alle unterschiedlich sind, da α ein primitives Element ist). Die Differenz $f_b(x) - f_c(x)$ wäre dann ein Polynom mit Grad $\leq k - 1$ mit k Nullstellen. Daher müssen $f_b(x)$ und $f_c(x)$ übereinstimmen, weshalb $(b_i)_{i=1}^k = (c_i)_{i=1}^k$ und damit wiederum auch $(\chi_i)_{i=1}^n = (\zeta_i)_{i=1}^n$ für alle $i = 1, 2, \dots, n$ gelten muss. Daraus können wir folgern, dass die Minimaldistanz $d \geq n - k + 1$ ist und wegen der Singleton Schranke (Satz 3) $d = n - k + 1$ gilt. \square

Wenn $d = n - k + 1$ ist, folgt wiederum, dass $t = \lfloor \frac{n-k}{2} \rfloor$ Fehler sicher decodiert werden können.

Beispiel 7. Ein Beispiel für die „klassische“ Codierung

Gegeben sei das Datenwort $b = \alpha^{10} \alpha^1 \alpha^8 \alpha^1 \alpha^{13} \alpha^0 \alpha^{14}$ mit Buchstaben aus $GF(16)$ mit $\alpha^4 = \alpha + 1$. Sei α das primitive Element, dann erhalten wir das Codewort durch Einsetzen von α^{i-1} für $i = 1, 2, \dots, 15$ in $f_b(x) = \alpha^{10} + \alpha^1 x^1 + \alpha^8 x^2 + \dots + \alpha^{14} x^6$.

Wie wir bereits wissen, lassen sich die hier benötigten Rechnungsarten in Galoisfeldern mit $q = 2^n$ Elementen mit Hilfe der binären XOR-Operation und Potenz-/Logarithmen-Tabellen lösen:

Wir können uns die Einträge unseres Codeworts $(\chi_i)_{i=1}^{15} = \chi_1 \chi_2 \dots \chi_{15}$ also durch Einsetzen berechnen:

$$\begin{aligned} \chi_1 &= f_b(\alpha^0) = \alpha^{10} + \alpha^1 \cdot (\alpha^0)^1 + \alpha^8 \cdot (\alpha^0)^2 + \dots + \alpha^{14} \cdot (\alpha^0)^6 = \alpha^{10} \\ \chi_2 &= f_b(\alpha^1) = \alpha^{10} + \alpha^1 \cdot (\alpha^1)^1 + \alpha^8 \cdot (\alpha^1)^2 + \dots + \alpha^{14} \cdot (\alpha^1)^6 = \alpha^4 \\ &\vdots \\ \chi_{15} &= f_b(\alpha^{14}) = \alpha^{10} + \alpha^1 \cdot (\alpha^{14})^1 + \alpha^8 \cdot (\alpha^{14})^2 + \dots + \alpha^{14} \cdot (\alpha^{14})^6 = \alpha^{12} \end{aligned}$$

In Summe erhalten wir also unser Codewort

$$\chi = \alpha^{10} \alpha^4 \alpha^{11} \alpha^3 \alpha^2 \alpha^{10} \alpha^8 \alpha^2 \alpha^1 \alpha^2 \alpha^6 \alpha^2 \alpha^3 \alpha^{11} \alpha^{12},$$

das mit unserem ursprünglichen Datenwort nicht mehr viel Ähnlichkeit hat.

4.2 Codieren durch Multiplikation mit Generatorpolynom

Wie bereits erwähnt liefert uns diese Methode die gleiche Menge an Codewörtern wie die Methode aus Kapitel 4.1. Damit diese Aussage auch stimmt (siehe Satz 15), müssen wir das hier benötigte Generatorpolynom entsprechend wählen. Wir wählen es so, dass dieses das Produkt der ersten $n - k$ aufeinander folgenden Potenzen des primitiven Elements $\alpha \in GF(q)$ ist:

$$g(x) = \prod_{i=1}^{n-k} (x - \alpha^i)$$

Zur Codierung eines Datenworts $(b_i)_{i=1}^k$ fassen wir dieses wieder als Polynom $f_b(x) = b_1 + b_2x + \dots + b_kx^{k-1}$ auf und multiplizieren $f_b(x)$ mit $g(x)$. Die Koeffizienten werden dabei entsprechend der Rechenregeln in Galoisfeldern addiert und multipliziert. Wir erhalten so ein Polynom mit maximalem Grad $(n-k)+(k-1) = n-1$. Die n Koeffizienten dieses Produkts sind unser Codewort $\chi(x) = \sum_{i=1}^n \chi_i x^{i-1}$. Es gilt also:

$$\chi(x) = g(x) \cdot f_b(x)$$

Mit dieser Codierung können wir also, ebenso wie mit der klassischen Codierung, allen Datenwörtern der Länge k ein eindeutiges Codewort der Länge n zuordnen. Wir sehen jedoch, dass die mit Generatorpolynom generierten Codewörter $\chi(x) = g(x) \cdot f_b(x)$, auf Grund der Berechnungsart, keinerlei Ähnlichkeit mit den durch Auswerten generierten Codewörtern $\chi(x) = \sum_{i=0}^{n-1} f_b(\alpha^i)x^i$ für dasselbe Datenwort $(b_i)_{i=1}^k$ haben. Dennoch erhalten wir in beiden Fällen den gleichen Code.

Lemma 7. Ein Wort $w(x)$ ist genau dann ein Codewort der Codierung mit Generatorpolynom, wenn $w(\alpha^i) = 0$ für alle $i = 1, 2, \dots, n - k$ ist.

Beweis. Wenn $w(\alpha^i) = 0$, dann gilt $(x - \alpha^i) \mid w(x)$. Da das für alle $i = 1, 2, \dots, n - k$ gilt, muss auch das Produkt dieser Monome (die ja alle unterschiedlich sind) $w(x)$ teilen. Nachdem aber $g(x) = \prod_{i=1}^{n-k} (x - \alpha^i)$ ist, folgt dass $g(x) \mid w(x)$. Demnach ist $w(x)$ ein Codewort und es gibt ein Datenwort $(b_i)_{i=1}^k$ für das $w(x) = g(x) \cdot f_b(x)$ ist.

Sei umgekehrt $w(x)$ ein Codewort. Dann wissen wir, dass $w(x) = g(x) \cdot f_b(x)$ ist. Nachdem $g(\alpha^i) = 0$ für alle $i = 1, 2, \dots, n - k$ ist (auf Grund der Definition von $g(x)$), ist auch $w(\alpha^i) = 0$. \square

Zeigen wir nun, dass wir mit beiden Methoden, also der klassischen Codierung und der mit Generatorpolynom, denselben Code, also die gleiche Menge an Codewörtern erhalten [2, Seite 43]:

Wie wir sehen sind beide Mengen gleich mächtig, da sowohl im ersten Code, als auch im zweiten Code q^k Codewörter enthalten sind. Daher reicht es, zu zeigen, dass ein Code Teilmenge des anderen Codes ist, womit die Gleichheit gelten muss:

Satz 15. Beide hier vorgestellten Methoden zur Berechnung von Codewörtern liefern denselben Code, also die gleiche Menge C an Codewörtern aus $GF(q)^n$. Es ist also jedes durch Auswerten berechnete Codewort $\chi(x) = \sum_{i=1}^n \chi_i x^{i-1}$ mit

$$\chi_i = f_b(\alpha^{i-1}) \text{ für } i = 1, 2, \dots, n$$

durch

$$g(x) = \prod_{i=1}^{n-k} (x - \alpha^i)$$

teilbar.

Beweis. Um zu zeigen, dass beide Methoden den gleichen Code liefern, reicht es zu zeigen, dass $\chi(\alpha^l) = 0$ für $l = 1, 2, \dots, n - k$. In diesem Fall folgt nämlich $g(x) \mid \chi(x)$ wegen Korollar 7, womit $\chi(x)$ auch ein mit Generatorpolynom erzeugtes Codewort ist. Betrachten wir daher:

$$\chi(x) = \sum_{i=1}^n \chi_i x^{i-1} = \sum_{i=1}^n f_b(\alpha^{i-1}) x^{i-1} = \sum_{j=0}^{n-1} f_b(\alpha^j) x^j$$

Und berechnen wir $\chi(\alpha^l)$:

$$\begin{aligned} \chi(\alpha^l) &= \sum_{j=0}^{n-1} f_b(\alpha^j) (\alpha^l)^j \\ &= \sum_{j=0}^{n-1} \sum_{i=0}^{k-1} b_{i+1} (\alpha^j)^i \alpha^{l \cdot j} \\ &= \sum_{i=0}^{k-1} b_{i+1} \left(\sum_{j=0}^{n-1} \alpha^{j(l+i)} \right) \end{aligned}$$

Wir müssen nur noch zeigen, dass $\sum_{j=0}^{n-1} \alpha^{j(l+i)} = 0$ für jegliche feste Wahl von l und i ist. Nachdem $1 \leq l \leq n - k$ und $0 \leq i \leq k - 1$, sehen wir sofort, dass für die Summe gelten muss:

$$1 \leq l + i \leq n - k + k - 1 = n - 1$$

Wir können die Summe daher umformulieren zu

$$\sum_{j=0}^{n-1} \alpha^{j(l+i)} = \sum_{j=0}^{n-1} (\alpha^{l+i})^j = \sum_{j=0}^{n-1} \beta^j$$

für ein $\beta \in GF(q) \setminus \{0, 1\}$, da α ja ein primitives Element ist, weshalb $\alpha \neq 0$, $\alpha^n = \alpha^0 = 1$ und alle Potenzen dazwischen ein Element aus dem Galoisfeld sein müssen. Wenn wir diese Summe mit $(1 - \beta)$ multiplizieren und berücksichtigen, dass $\beta^n = \beta^0$ ist, sehen wir, dass:

$$\begin{aligned} (1 - \beta) \sum_{j=0}^{n-1} \beta^j &= \sum_{j=0}^{n-1} \beta^j - \sum_{j=0}^{n-1} \beta^{j+1} \\ &= \sum_{j=0}^{n-1} \beta^j - \sum_{j=1}^n \beta^j \\ &= \sum_{j=0}^{n-1} \beta^j - \sum_{j=0}^{n-1} \beta^j \\ &= 0 \end{aligned}$$

Somit muss entweder $(1 - \beta) = 0$ sein, was aber nicht möglich ist, da $GF(q)$ ein Körper

und $\beta \neq 1$ ist, oder $\sum_{j=0}^{n-1} \beta^j = 0$. Demnach ist

$$\begin{aligned}\chi(\alpha^l) &= \sum_{i=0}^{k-1} b_{i+1} \left(\sum_{j=0}^{n-1} \alpha^{j(l+i)} \right) \\ &= \sum_{i=0}^{k-1} b_{i+1} \left(\sum_{j=0}^{n-1} \beta^j \right) \\ &= \sum_{i=0}^{k-1} b_{i+1} \cdot 0 = 0\end{aligned}$$

für alle $l = 1, 2, \dots, n - k$ voraus, wegen Korollar 7, $g(x) \mid \chi(x)$ folgt. \square

Nachdem wir jetzt also wissen, dass es sich in beiden Fällen um dieselbe Menge an Codewörtern handelt, können wir natürlich sofort folgern, dass auch bei Verwendung der Methode mit Generatorpolynom die Minimaldistanz $d = n - k + 1$ ist und wir somit wieder $t = \lfloor \frac{n-k}{2} \rfloor$ Fehler korrekt decodieren können. Dies war auch der Hintergedanke bei der Definition von $g(x)$ als Produkt der ersten $n - k$ Potenzen von α .

Beispiel 8. Ein Beispiel für Codierung „mit Generatorpolynom“

Nehmen wir wieder das Datenwort $b = \alpha^{10}\alpha^1\alpha^8\alpha^1\alpha^{13}\alpha^0\alpha^{14}$ mit der Länge $k = 7$ und Buchstaben aus $GF(16)$ mit $\alpha^4 = \alpha + 1$. Sei α wieder das primitive Element, dann erhalten wir das Codewort durch Multiplikation des Datenworts $f_b(x) = \alpha^{10} + \alpha^1 x^1 + \alpha^8 x^2 + \dots + \alpha^{14} x^6$ mit dem Generatorpolynom

$$g(x) = \prod_{i=1}^{n-k} (x - \alpha^i)$$

Wir berechnen uns $g(x)$:

$$g(x) = \prod_{i=1}^8 (x - \alpha^i) = \alpha^6 + \alpha^{11}x + \alpha^5x^2 + \alpha^{13}x^3 + \alpha^2x^4 + \alpha^4x^5 + \alpha^2x^6 + \alpha^{14}x^7 + \alpha^0x^8$$

Damit erhalten wir unser Codewort durch Polynommultiplikation mit Koeffizienten aus $GF(16)$:

$$\begin{aligned}\chi(x) &= g(x) \cdot f_b(x) \\ &= \alpha^1 + \alpha^{10}x + \alpha^{10}x^2 + \alpha^0x^3 + \alpha^{10}x^4 + \alpha^{10}x^5 + \alpha^{10}x^6 + \alpha^9x^7 + \\ &\quad \alpha^3x^8 + \alpha^{11}x^9 + \alpha^2x^{10} + \alpha^0x^{11} + \alpha^5x^{12} + \alpha^6x^{13} + \alpha^{14}x^{14}\end{aligned}$$

Unser Codewort lautet also

$$\chi = \alpha^1\alpha^{10}\alpha^{10}\alpha^0\alpha^{10}\alpha^{10}\alpha^{10}\alpha^9\alpha^3\alpha^{11}\alpha^2\alpha^0\alpha^5\alpha^6\alpha^{14}.$$

Wie wir sehen hat dieses weder Ähnlichkeit mit dem Datenwort

$$b = \alpha^{10}\alpha^1\alpha^8\alpha^1\alpha^{13}\alpha^0\alpha^{14},$$

noch mit dem durch Auswerten berechneten Codewort

$$\chi = \alpha^{10}\alpha^4\alpha^{11}\alpha^3\alpha^2\alpha^{10}\alpha^8\alpha^2\alpha^1\alpha^2\alpha^6\alpha^2\alpha^3\alpha^{11}\alpha^{12}.$$

Wir werden im Kapitel 7.1 die systematische Codierung kennen lernen. Dabei handelt es sich um eine Adaption der Codierungsmethode mit Generatorpolynom, die uns das Auslesen des Datenworts erleichtern wird.

5 Decodierung von RS Codes

Wie wir gesehen haben, gibt es zwei verschiedene Methoden, um ein Codewort eines RS Codes zu erzeugen. Daher gibt es auch verschiedene Ansätze zur Decodierung, die vom jeweiligen Codierungsverfahren inspiriert sind. Im Falle eines fehlerfrei ausgelesenen Worts ist es natürlich am einfachsten, die Codierungsmethode zu invertieren, um so das Datenwort aus dem Codewort zu erhalten. Doch diese Methode funktioniert im Normalfall, d.h. wenn zumindest ein Fehler auftritt, nicht. Wir suchen demnach für beide Codierungsmethoden einen Algorithmus der uns möglichst effizient das wahrscheinlichste Codewort, bzw. Datenwort berechnet.

5.1 Decodierung mittels GS Algorithmus

Im Fall der klassischen Codierung können wir, vorausgesetzt das ausgelesene Wort $\chi_1\chi_2 \dots \chi_n$ ist fehlerfrei, also ein Codewort, das ursprüngliche Datenwort $b_1b_2 \dots b_k$ berechnen, indem wir uns die Codierung in Erinnerung rufen und versuchen die Koeffizienten von $f_b(x)$ zu berechnen. Dazu muss ein Gleichungssystem mit Gleichungen der Gestalt

$$b_1 + b_2\alpha^{(i-1)1} + \dots + b_k\alpha^{(i-1)(k-1)} = \chi_i$$

für alle $i = 1, 2, \dots, n$ gelöst werden. Da das ausgelesene Wort ein Codewort ist, genügen sogar beliebige k dieser n Gleichungen, um das Datenwort zu berechnen. Sollten jedoch gespeicherte Werte χ_i verloren gegangen oder geändert worden sein, ist es deutlich komplizierter, das Datenwort mit dieser Methode zu berechnen.

Betrachten wir daher die Decodierung des RS Codes mit Hilfe des Guruswami-Sudan Algorithmus (kurz GS Algorithmus). Es handelt sich hierbei um ein zweistufiges Problem, bestehend aus einem Interpolations- und einem Faktorisierungsteil. Gehen wir davon aus, dass wir ein Wort $(\beta_i)_{i=1}^n$ ausgelesen haben, das möglicherweise keinem Codewort entspricht, was bedeutet, dass es fehlerbehaftet ist. Außerdem wissen wir, dass das ursprüngliche Datenwort k Zeichen lang war. Unsere Aufgabe ist es nun, jenes Polynom $f_b(x)$ mit $\deg f_b(x) \leq k - 1$ zu finden, dass die Gleichungen

$$f_b(\alpha^{i-1}) = \beta_i \quad \text{für } i = 1, 2, \dots, n$$

an möglichst vielen Stellen erfüllt, da wir davon ausgehen, dass eine Veränderung an möglichst wenig Stellen stattgefunden hat. Je mehr Übereinstimmungen wir finden, um so größer ist die Wahrscheinlichkeit, dass es sich um unser gesuchtes Polynom und damit um unser ursprüngliches Datenwort handelt.

Bisher haben wir bereits in Kapitel 3.3.1 *Kötters Interpolationsalgorithmus* und in Kapitel 3.4.1 *Roth-Ruckensteins Faktorisierungsalgorithmus* kennen gelernt. Die Kombination beider Algorithmen liefert uns unter einer gewissen Voraussetzung ein bis mehrere mögliche Datenwörter:

Kötters Interpolationsalgorithmus benötigt das ausgelesene Wort $(\beta_i)_{i=1}^n$, einen frei wählbaren Parameter m und eine von der Datenwortlänge k abhängige $(1, k - 1)$ -revlex Ordnung und liefert uns anschließend ein bezüglich dieser Ordnung minimales Polynom $Q(x, y)$, dass eine m -fache Nullstelle bei allen Stellen $(\alpha^{i-1}, \beta_i)_{i=1}^n$ hat.

Der *RR Faktorisierungsalgorithmus* benötigt ein Polynom $Q(x, y)$ und die Datenwortlänge k und liefert uns alle Polynome $f(x)$ vom Grad $\leq k - 1$ (also mit k Koeffizienten), für die $(y - f(x)) \mid Q(x, y)$ gilt. Unter diesen Polynomen befindet sich auch unser ursprüngliches Datenwort, wenn $K_m m > \deg_{(1, k-1)} Q(x, y)$ ist, wobei K_m die benötigte Anzahl an korrekten β_i ist, d.h. $\beta_i = f_b(\alpha^{i-1})$ für ein Datenwort $(b_i)_{i=1}^k$, und abhängig von m variieren kann. Sollten also mehr als K_m Stellen korrekt sein, ist unser ursprüngliches Datenwort in der Liste der Lösungen des Faktorisierungsalgorithmus enthalten¹⁹.

Die angesprochene benötigte Voraussetzung ist also die Ungleichung:

$$K_m m > \deg_{(1, k-1)} Q(x, y)$$

Da jedoch mit wachsendem m auch der Grad von $Q(x, y)$ wächst, was wiederum einen größeren $\deg_{(1, k-1)} Q(x, y)$ zur Folge hat, gibt es eine minimale Anzahl an Stellen K_m , an denen das ausgelesene Wort mit einem Codewort übereinstimmen muss. In Kapitel 6.1 werden die Eigenschaften des GS Algorithmus, also die Anzahl benötigter korrekter Stellen K_m , die Anzahl zulässiger Fehler t_m sowie die maximale Anzahl an Polynomen L_m , die uns der Faktorisierungsalgorithmus liefert und die alle von m abhängig sind, genauer betrachtet.

5.1.1 Decodierung von Auslöschungen

Die soeben beschriebene Methode berechnet uns somit das wahrscheinlichste Datenwort, bei maximal t_m Fehlern, wobei wir in Kapitel 6.1 berechnen werden, wie groß dieser Wert ist. Doch was passiert, wenn wir wissen, dass an einer bestimmten Stelle im ausgelesenen Wort ein Fehler zu finden oder ein Buchstabe verloren gegangen ist? In diesem Fall sprechen wir von einer sogenannten Auslöschung. Diese wollen bzw. können wir natürlich nicht einfach übergehen. Wir müssen Auslöschungen demnach in unsere Decodierung einfließen lassen. Doch was bedeutet das für unseren Decodierungsalgorithmus? Wie können und müssen wir ihn adaptieren, um aus dieser Zusatzinformation Nutzen zu ziehen?

Wenn wir keine weiteren Zusatzüberlegungen anstellen wollen, können wir an Stellen bei denen Auslöschungen aufgetreten sind einen beliebigen Wert eintragen. Wir begehen auf diese Art zwar mit großer Wahrscheinlichkeit einen Fehler, allerdings können wir das Wort jetzt mit der bereits bekannten Methode decodieren. Doch mutwillig einen Fehler einzubauen ist nicht sinnvoll. Daher wollen wir uns überlegen, was zu tun ist, wenn Auslöschungen entstehen.

Auslöschungen können auf zwei Arten entstehen:

- Durch Unlesbarkeit von Buchstaben. Dies kann bei Beschädigung des Datenträgers passieren.
- Durch Weglassen eines Buchstaben aus dem ausgelesenen Wort, da wir entweder sicher sind, dass der Buchstabe falsch ist, oder die Vermutung dafür sehr groß ist.

¹⁹Mit anderen Worten: Sollten mindesten K_m der Buchstaben β_i korrekt sein, also noch immer die berechneten χ_i , dann ist das Datenwort mit dem wir das Codewort $(\chi_i)_{i=1}^n$ berechnet haben, Teil der Lösungsmenge des RR Algorithmus aus Kapitel 3.4.1

In beiden Fällen reduzieren wir unser ausgelesenes Wort um diese Stellen und lassen das komplette Paar (α^{i-1}, β_i) bei der Interpolation für $Q(x, y)$ wegfallen. Wie man sofort sieht, behalten dennoch alle Aussagen ihre Gültigkeit. Einzig die Anzahl korrigierbarer Fehler t_m , bzw. die Anzahl der benötigten korrekten Stellen K_m ändert sich. Bezeichnen wir mit ϵ die Anzahl an aufgetretenen Auslöschungen, dann wird aus unserem $[n, k]$ -Code mit Minimaldistanz d ein $[n - \epsilon, k]$ -Code mit Minimaldistanz $\geq d - \epsilon$ wie man leicht nachvollziehen kann. Nachdem jedoch die Länge des Codeworts mindestens die des Datenworts sein muss, erhalten wir eine Schranke für die Anzahl an Auslöschungen:

$$\epsilon \leq n - k.$$

Zentrale Forderung damit $(y - f(x)) \mid Q(x, y)$ gilt, ist immer noch, dass:

$$K_m m > \deg_{(1, k-1)} Q(x, y)$$

Jedoch haben wir im Interpolationsschritt weniger Paare (α^{i-1}, β_i) zur Verfügung gehabt, weshalb sich der Grad $\deg_{(1, k-1)} Q(x, y)$ reduziert. Da wir ein festes m haben (das wurde ja zur Decodierung ohne Auslöschungen gewählt), bedeutet das für uns, dass sich die Anzahl der benötigten korrekten Stellen K_m auch reduzieren kann. Durch Nachrechnen der Gleichungen aus dem Kapitel 6.1 ergibt sich für K_m in Abhängigkeit von $n - \epsilon$ folgende neue Gleichung:

$$\left| \sqrt{\frac{(n - \epsilon)v(m + 1)}{m}} - \frac{v}{2m} \right| + 1 \leq K_m(n - \epsilon) \leq \left| \sqrt{\frac{(n - \epsilon)v(m + 1)}{m} + \frac{v^2}{4m^2}} - \frac{v}{2m} \right| + 1$$

Womit für $t_m(n - \epsilon)$, da es sich hierbei ja um die Differenz $n - \epsilon - K_m(n - \epsilon)$ handelt, folgt:

$$n - \epsilon - \left| \sqrt{\frac{(n - \epsilon)v(m + 1)}{m} + \frac{v^2}{4m^2}} - \frac{v}{2m} \right| - 1 \leq$$

$$t_m(n - \epsilon) \leq n - \epsilon \left| \sqrt{\frac{(n - \epsilon)v(m + 1)}{m}} - \frac{v}{2m} \right| - 1$$

Für $m \rightarrow \infty$ sehen wir sofort, dass der Algorithmus jetzt nur noch bis zu $t_{GS}(n - \epsilon) = n - \epsilon - 1 - \lfloor \sqrt{(n - \epsilon)v} \rfloor$ Fehler an unbekanntenen Stellen decodieren kann.

Wie wir sehen, ist die Decodierung von Auslöschungen durch simples Weglassen der betroffenen Stellen im ausgelesenen Wort und der dazugehörigen Potenz α^{i-1} machbar und wir erhalten einen verbesserten Decodierungsalgorithmus.

5.2 Decodierung mittels BM Algorithmus

Der Berlekamp-Massey Algorithmus [1, Seite 180-184] (kurz BM-Algorithmus) hilft uns bei der Decodierung von ausgelesenen (fehlerhaften) Wörtern deren ursprüngliche Datenwörter mit Generatorpolynom codiert wurden. Betrachten wir zuerst ein korrekt ausgelesenes Codewort $\chi_1 \chi_2 \dots \chi_n$. Nachdem wir dieses durch Multiplikation mit dem Generatorpolynom erhalten haben, können wir das Datenwort berechnen, indem wir $\chi_1 \chi_2 \dots \chi_n$ als Polynom

$$\chi(x) = \chi_1 + \chi_2 x + \dots + \chi_n x^{n-1}$$

auffassen und es wieder durch unser Generatorpolynom dividieren. Es ist also

$$f_b(x) = \frac{\chi(x)}{g(x)}$$

unter der Voraussetzung, dass $\chi(x)$ ein Codewort ist.

Doch sollten Fehler auftreten, benötigen wir eine andere Methode, um zu decodieren. Diese wurde von Berlekamp vorgestellt und von Massey verfeinert [1]. Der BM-Algorithmus berechnet das Codewort, das einem ausgelesenen Wort entspricht, sofern nicht mehr als $t = \lfloor \frac{n-k}{2} \rfloor$ Fehler gemacht wurden. Wir benutzen für diesen Algorithmus die Eigenschaft, dass alle Codewörter, wenn wir sie wieder als Polynome auffassen, dieselben Nullstellen wie das Generatorpolynom $g(x)$ haben müssen, da sie ja Produkt eines Datenworts mit $g(x)$ sind. D.h. ein ausgelesenes Wort $\beta(x)$ ist ein Codewort, genau dann wenn

$$\beta(\alpha^i) = f_b(\alpha^i) \cdot g(\alpha^i) = f_b(\alpha^i) \cdot 0 = 0$$

für alle $i = 1, 2, \dots, n - k$ ist.

Definition 24. Diese $\beta(\alpha^i)$ bezeichnen wir als Syndrome s_i . Unser ausgelesenes Wort $\beta(x) = \sum_{i=1}^n \beta_i x^{i-1}$ liefert uns also die Syndrome

$$s_i := \beta(\alpha^i) \text{ für } i = 1, 2, \dots, 2t.$$

Diese s_i können für alle $i \in \mathbb{N}$ berechnet werden. Da jedoch maximal $t = \lfloor \frac{n-k}{2} \rfloor$ Fehler korrigiert werden können, reicht die Betrachtung der ersten $2t$ Syndrome wie wir im Folgenden sehen werden.

Gehen wir nun davon aus, dass wir ein Datenwort codiert haben zu:

$$\chi(x) = \sum_{i=1}^n \chi_i x^{i-1}$$

Beim Auslesen dieses Codeworts können Fehler e_i an verschiedenen Stellen auftreten. Wir haben daher ein Fehlerwort

$$e(x) = \sum_{i=1}^n e_i x^{i-1}$$

für das die meisten $e_i = 0$ sind, da wir von einer geringen Anzahl an Fehlern ausgehen. Das von uns ausgelesene Wort $\beta(x)$ können wir daher schreiben als die Summe von $\chi(x)$ und $e(x)$:

$$\beta(x) = \chi(x) + e(x) = \sum_{i=1}^n (\chi_i + e_i) x^{i-1}$$

Woraus wir sofort folgern können, dass

$$s_i = \chi(\alpha^i) + e(\alpha^i) = 0 + \sum_{j=0}^{n-1} e_{j+1} (\alpha^i)^j = \sum_{j=0}^{n-1} e_{j+1} \alpha^{i+j}$$

ist. Betrachten wir die Summe auf der rechten Seite etwas genauer, dann sehen wir, dass wir diese umschreiben können zu:

$$s_i = \sum_{j=0}^{n-1} e_{j+1} (\alpha^j)^i = \sum_{k=1}^e Y_k X_k^i \quad \text{mit } Y_k = e_{j_k} \text{ und } X_k = \alpha^{j_k-1}, k = 1, 2, \dots, e$$

wobei $j_k \in \{j \mid e_j \neq 0, j = 1, 2, \dots, n\}$

und $e = \left| \{j \mid e_j \neq 0, j = 1, 2, \dots, n\} \right|$.

Mit X_k werden die Fehlerorte und mit Y_k die Fehlerwerte bezeichnet. Damit das ursprüngliche Datenwort wiederhergestellt werden kann, muss $e \leq t$ sein. Die $X_k, k = 1, 2, \dots, e$, sind paarweise verschieden und aus $X_k = \alpha^{j_k-1}$ können wir folgern, dass an der j_k -ten Stelle in unserem ausgelesenen Wort der Fehler $Y_k = e_{j_k}$ passiert ist. Korrigiert wird, indem man den gemachten Fehler Y_k an der richtigen Stelle des Wortes $\beta(x)$ abzieht: Sei also $X_k = \alpha^{j_k-1}$, dann ist $\chi_{j_k} = \beta_{j_k} - Y_k$.

Um das zu einem ausgelesenen Wort gehörende Codewort zu bestimmen, haben Berlekamp und Massey nur noch mit Hilfe der Syndrome s_i die X_k und Y_k ²⁰ berechnen müssen, wobei man natürlich davon ausgeht, dass die Anzahl der gemachten Fehler e möglichst gering ist.

Um die X_k und Y_k zu berechnen, müssen wir uns zuerst eine Hilfsgleichung (die so genannte Schlüsselgleichung [1, Seite 179]) aufstellen und uns überlegen, wie wir diese lösen können.

Definition 25. Dazu definieren wir

$$\sigma(x) := \prod_{i=1}^e (1 - X_i x) = 1 + \sum_{j=1}^e \sigma_j x^j$$

und bezeichnen dieses Polynom als das Fehlerlokalisierungspolynom [1, Seite 178], da wir durch Berechnung der Nullstellen $x = X_i^{-1} = \alpha^{-j_i+1}$ von $\sigma(x)$ die Positionen $j_i, i = 1, 2, \dots, e$, an denen ein Fehler gemacht wurde, bestimmen können.

Die Berechnung der Nullstellen erfolgt mit Hilfe der Chien Suche (siehe Kapitel 5.3), da $\sigma(x)$ nur einfache Nullstellen hat, wenn höchstens t Fehler auftreten. Die so erhaltenen Nullstellen des Fehlerlokalisierungspolynoms müssen dann nur noch invertiert werden, um die X_k zu erhalten. Da wir jedoch nur die Syndrome s_i berechnen können, brauchen wir noch einen weiteren Zusammenhang zwischen diesen und dem Fehlerlokalisierungspolynom. Dazu betrachten wir die Potenzreihe der Syndrome $s(x) = \sum_{i=1}^{\infty} s_i x^i$, die wir umschreiben können zu:

$$s(x) = \sum_{j=1}^{\infty} s_j x^j = \sum_{j=1}^{\infty} \sum_{i=1}^e Y_i X_i^j x^j = \sum_{i=1}^e Y_i \frac{X_i x}{1 - X_i x}$$

Multiplikation dieser Gleichung mit $\sigma(x)$ und Addition von $\sigma(x)$ auf beiden Seiten liefert uns:

$$(1 + s(x)) \sigma(x) = \sigma(x) + \sum_{i=1}^e X_i Y_i x \prod_{j=1, j \neq i}^e (1 - X_j x)$$

²⁰Wir sehen, dass die Indizierung des ausgelesenen Wortes mit $\beta_0, \beta_1, \dots, \beta_{n-1}$ günstiger wäre, allerdings ist es in Matlab nicht möglich das 0-te Element eines Arrays anzusprechen, weshalb die Nummerierung bei 1 startet.

Bemerkung 13. Die Betrachtung der Potenzreihe ist möglich, da wir ja wissen, dass $\alpha^{q-1} = 1$ ist, weshalb man $s_i = \chi(\alpha^i)$ für alle $i \in \mathbb{N}$ berechnen kann. $(s_i)_{i=1}^\infty$ ist also eine periodische Folge mit einer maximalen Periodenlänge von $q - 1$.

Definition 26. Die rechte Seite dieser Gleichung bezeichnen wir als Fehlerwertepolynom [1, Seite 179]

$$\omega(x) := \sigma(x) + \sum_{i=1}^e X_i Y_i x \prod_{j=1, j \neq i}^e (1 - X_j x)$$

da wir mit dessen Hilfe den begangenen Fehler berechnen können, wie wir später sehen werden.

Nachdem wir $t = \lfloor \frac{n-k}{2} \rfloor$ kennen, wissen wir, dass höchstens die ersten $2t$ Koeffizienten von $s(x)$ für uns interessant sind, da ansonsten eine Fehlerkorrektur nicht möglich ist. Daher reduziert sich die obige Gleichung zur so genannten Schlüsselgleichung von Berlekamp [1, Seite 178]:

$$(1 + s(x)) \sigma(x) \equiv \omega(x) \pmod{x^{2t+1}}$$

Unsere Aufgabe ist es jetzt, für ein gegebenes Polynom $s(x) \pmod{x^{2t+1}}$ die Polynome $\sigma(x)$ und $\omega(x)$ zu berechnen, wobei beide (wie wir aus den Definitionen sehen) maximal den Grad e haben, was der Anzahl aufgetretener Fehler entspricht. Diese Anzahl sollte möglichst gering sein, weshalb auch der Grad von $\sigma(x)$ und $\omega(x)$ möglichst klein sein sollte. Sobald wir diese Polynome haben, werden wir uns der Berechnung von X_k und Y_k widmen.

Doch das Berechnen von zwei passenden Polynomen, die möglichst geringen Grad haben sollen, ist nicht so leicht, weshalb wir die soeben vorgestellte Schlüsselgleichung rekursiv umschreiben mit dem Ziel $\sigma(x)$ und $\omega(x)$ berechnen zu können [1, Seite 180-184]. Wenn wir die Polynome auf beiden Seiten auf den Grad x^{k+1} reduzieren, muss also gelten:

$$(1 + s(x)) \sigma(x)^{(k)} \equiv \omega(x)^{(k)} \pmod{x^{k+1}}$$

Wir versuchen nun für $k = 0, 1, 2, \dots, 2t$ Polynome $\sigma(x)^{(k)}$ und $\omega(x)^{(k)}$ zu finden, die diese Gleichungen erfüllen. Wenn wir für ein k bereits eine Lösung haben, versuchen wir diese natürlich auch gleich für $k + 1$ zu verwenden. So können wir den Grad beider Polynome möglichst gering halten. Sollten jedoch $\sigma(x)^{(k+1)}$ und $\omega(x)^{(k+1)}$ die Schlüsselgleichung nicht mehr erfüllen, müssen wir versuchen, diese Polynome möglichst behutsam zu verändern, sodass nur im allerletzten Fall ein Erhöhen der Potenz der zuvor berechneten Polynome notwendig ist.

Der Algorithmus, der diese Aufgabe erfüllt, ist bekannt als BM Algorithmus und wird am Ende dieses Kapitels zusammengefasst. Wir werden ihn durch immer genaueres Bestimmen von $\sigma(x)^{(k+1)}$ und $\omega(x)^{(k+1)}$ im Folgenden sukzessive herleiten. Dieser heuristische Ansatz entspricht der Vorgehensweise in [1, Seite 180-184]:

Gehen wir davon aus, dass wir Polynome $\sigma(x)^{(k)}$ und $\omega(x)^{(k)}$ gefunden haben, die die Schlüsselgleichung $\pmod{x^{k+1}}$ erfüllen. Im Allgemeinen können wir nicht erwarten, dass für diese auch

$$(1 + s(x)) \sigma(x)^{(k)} \equiv \omega(x)^{(k)} \pmod{x^{k+2}}$$

gilt. Es wird also durch Vergrößern der zu betrachtenden Potenz von x ein $\Delta^{(k)}$ geben, das wir schreiben können als:

$$(1 + s(x))\sigma(x)^{(k)} \equiv \omega(x)^{(k)} + \Delta^{(k)}x^{k+1} \pmod{x^{k+2}}$$

wobei dieses $\Delta^{(k)}$ der Koeffizient von x^{k+1} in dem Produkt $(1 + s(x))\sigma(x)^{(k)}$ ist. Sollte dieser Koeffizient $\Delta^{(k)} = 0$ sein, können wir

$$\begin{aligned}\sigma(x)^{(k+1)} &= \sigma(x)^{(k)} \text{ und} \\ \omega(x)^{(k+1)} &= \omega(x)^{(k)}\end{aligned}$$

setzen und die Schlüsselgleichung mit der nächsten Potenz von x betrachten. Doch wie müssen wir $\sigma(x)^{(k+1)}$ und $\omega(x)^{(k+1)}$ wählen, falls $\Delta^{(k)} \neq 0$ ist? Dazu betrachten wir zwei Hilfspolynome $\tau^{(k)}$ und $\gamma^{(k)}$, die Lösung der Hilfsgleichung

$$(1 + s(x))\tau(x)^{(k)} \equiv \gamma(x)^{(k)} + x^k \pmod{x^{k+1}}$$

sind. Auch von diesen Polynomen wollen wir verlangen, dass sie möglichst kleinen Grad haben. Man sieht dann sofort, dass für

$$\begin{aligned}\sigma(x)^{(k+1)} &= \sigma(x)^{(k)} - \Delta^{(k)}x\tau(x)^{(k)} \text{ und} \\ \omega(x)^{(k+1)} &= \omega(x)^{(k)} - \Delta^{(k)}x\gamma(x)^{(k)}\end{aligned} \tag{1}$$

die Schlüsselgleichung

$$\begin{aligned}(1 + s(x))\sigma(x)^{(k+1)} &= (1 + s(x))\left(\sigma(x)^{(k)} - \Delta^{(k)}x\tau(x)^{(k)}\right) \\ &= (1 + s(x))\sigma(x)^{(k)} - (1 + s(x))\Delta^{(k)}x\tau(x)^{(k)} \\ &\equiv \omega(x)^{(k)} + \Delta^{(k)}x^{k+1} - \left(\gamma(x)^{(k)} + x^k\right)\Delta^{(k)}x \\ &= \omega(x)^{(k)} + \Delta^{(k)}x^{k+1} - \gamma(x)^{(k)}\Delta^{(k)}x - \Delta^{(k)}x^{k+1} \\ &= \omega(x)^{(k)} - \gamma(x)^{(k)}\Delta^{(k)} \\ &= \omega(x)^{(k+1)} \pmod{x^{k+2}}\end{aligned}$$

erfüllt ist. Nachdem $\tau(x)^{(k)}$ und $\gamma(x)^{(k)}$ Summanden von $\sigma(x)^{(k+1)}$ und $\omega(x)^{(k+1)}$ sind, ist auch nachvollziehbar, warum wir fordern, dass diese möglichst geringen Grad haben sollen. Doch wie sind diese beiden Polynome zu wählen? Wir sehen, dass wir dafür zwei verschiedene Möglichkeiten haben: Entweder

$$\tau(x)^{(k+1)} = x\tau(x)^{(k)} \quad \text{und} \quad \gamma(x)^{(k+1)} = x\gamma(x)^{(k)} \tag{2}$$

oder

$$\tau(x)^{(k+1)} = \frac{\sigma(x)^{(k)}}{\Delta^{(k)}} \quad \text{und} \quad \gamma(x)^{(k+1)} = \frac{\omega(x)^{(k)}}{\Delta^{(k)}} \tag{3}$$

In beiden Fällen ist die Hilfsgleichung

$$(1 + s(x))\tau(x)^{(k+1)} \equiv \gamma(x)^{(k+1)} + x^{k+1} \pmod{x^{k+2}}$$

erfüllt, wie man durch einfaches Nachrechnen sieht. Wir können die Hilfspolynome demnach auf zwei verschiedene Arten wählen. Sollte $\Delta^{(k)} = 0$ sein, so ist offensichtlich, dass wir nur mit Variante (2) arbeiten können. Im Fall $\Delta^{(k)} \neq 0$ müssen wir die Entscheidung für $\tau(x)^{(k+1)}$, dessen Grad ja möglichst gering sein soll, vom Grad von $\tau(x)^{(k)}$ und dem von $\sigma(x)^{(k)}$ abhängig machen. Analoges gilt für $\gamma(x)^{(k+1)}$.

Bemerkung 14. Um nun nicht alle Überlegungen doppelt machen zu müssen, betrachten wir ab jetzt primär $\sigma(x)^{(k)}$ und $\tau(x)^{(k)}$, da wir sämtlich Argumente analog auf $\omega(x)^{(k)}$ und $\gamma(x)^{(k)}$ anwenden können.

Betrachten wir also die Grade von $\sigma(x)^{(k+1)}$, $\tau(x)^{(k+1)}$, $\omega(x)^{(k+1)}$ und $\gamma(x)^{(k+1)}$, die wir mit Hilfe der soeben vorgestellten Rekursionen für alle vier Polynome in folgende Fälle unterteilen können:

$$\begin{aligned} \deg \sigma(x)^{(k+1)} &= \begin{cases} \deg \sigma(x)^{(k)} & \text{wenn } \Delta^{(k)} = 0, \text{ oder } \deg \tau(x)^{(k)} + 1 < \deg \sigma(x)^{(k)} \\ 1 + \deg \tau(x)^{(k)} & \text{wenn } \Delta^{(k)} \neq 0 \text{ und } \deg \tau(x)^{(k)} + 1 > \deg \sigma(x)^{(k)} \end{cases} \\ \deg \tau(x)^{(k+1)} &= \begin{cases} 1 + \deg \tau(x)^{(k)} & \text{wenn wir (2) verwenden} \\ \deg \sigma(x)^{(k)} & \text{wenn wir (3) verwenden} \end{cases} \\ \deg \omega(x)^{(k+1)} &= \begin{cases} \deg \omega(x)^{(k)} & \text{wenn } \Delta^{(k)} = 0, \text{ oder } \deg \gamma(x)^{(k)} + 1 < \deg \omega(x)^{(k)} \\ 1 + \deg \gamma(x)^{(k)} & \text{wenn } \Delta^{(k)} \neq 0 \text{ und } \deg \gamma(x)^{(k)} + 1 > \deg \omega(x)^{(k)} \end{cases} \\ \deg \gamma(x)^{(k+1)} &= \begin{cases} 1 + \deg \gamma(x)^{(k)} & \text{wenn wir (2) verwenden} \\ \deg \omega(x)^{(k)} & \text{wenn wir (3) verwenden} \end{cases} \end{aligned}$$

Eine besondere Rolle haben die zwei noch fehlenden Fälle. Wenn nämlich

$$\begin{aligned} \Delta^{(k)} \neq 0 \text{ und } \deg \tau(x)^{(k)} + 1 &= \deg \sigma(x)^{(k)}, \text{ bzw.} \\ \Delta^{(k)} \neq 0 \text{ und } \deg \gamma(x)^{(k)} + 1 &= \deg \omega(x)^{(k)} \end{aligned}$$

ist, besteht sogar die Möglichkeit, dass sich der Grad von $\sigma(x)^{(k+1)}$ bzw. $\omega(x)^{(k+1)}$ verringert, wie wir in den Gleichungen (1) sehen. Eine derartige Reduktion kann jedoch nur dann stattfinden, wenn die führenden Koeffizienten gleich sind. Da wir unsere Entscheidung aber nicht von diesem Zufall abhängig machen wollen (siehe dazu auch [1, Seite 182]), betrachten wir eine Funktion $D(k)$, die eine obere Abschätzung der Grade von $\sigma(x)$, $\tau(x)$, ... liefert.

Diese, von der Iterationsstufe abhängige Funktion $D(k)$, von der wir die Entscheidung zwischen (2) und (3) abhängig machen wollen, definieren wir so, dass

$$\begin{aligned} \deg \sigma(x)^{(k)} &\leq D(k) \text{ und} \\ \deg \tau(x)^{(k)} &\leq k - D(k) \end{aligned}$$

ist. Auf Grund dieser Forderung, sowie der Wachstumseigenschaften des Grades von $\sigma(x)^{(k)}$, sehen wir, dass folgende rekursive Definition von $D(k)$ als sinnvoll zu betrachten ist:

Definition 27.

$$D(k+1) := \begin{cases} D(k) & \text{wenn } \Delta(k) = 0 \text{ oder } D(k) \geq \frac{k+1}{2} \\ k+1 - D(k) & \text{wenn } \Delta(k) \neq 0 \text{ und } D(k) < \frac{k+1}{2} \end{cases}$$

Es lässt sich schnell nachrechnen, dass $\deg \sigma(x)^{(k+1)} \leq D(k+1)$ ist, was wir auch analog für $\omega(x)^{(k+1)}$ folgern können. Um sicher zu stellen, dass auch $\deg \tau(x)^{(k+1)} \leq (k+1) - D(k+1)$ ist, bzw. das Gleiche für $\gamma(x)^{(k+1)}$ gilt, müssen wir für die Wahl

zwischen (2) und (3) die Bedingungen korrigieren auf (siehe auch [1, Seite 183]):

- (2) ist zu verwenden, wenn $\Delta(k) = 0$ oder $D(k) > \frac{k+1}{2}$
(3) ist zu verwenden, wenn $\Delta(k) \neq 0$ und $D(k) < \frac{k+1}{2}$

Wie wir sehen fehlt noch der Fall $\Delta(k) \neq 0$ und $D(k) = \frac{k+1}{2}$. Für diesen führen wir noch eine boolesche Funktion $B(k)$ ein, die wir mit $B(0) = 0$ initialisieren:

Definition 28.

$$B(k+1) := \begin{cases} B(k) & \text{wenn wir (2) verwenden} \\ 1 - B(k) & \text{wenn wir (3) verwenden} \end{cases}$$

Mit dieser Funktion und den Startwerten $\sigma(x)^{(0)} = 1$, $\tau(x)^{(0)} = 1$, $\omega(x)^{(0)} = 1$, $\gamma(x)^{(0)} = 0$ und $D(0) = 0$, können wir, wenn wir die Wahl zwischen (2) und (3) erweitern zu

- (2) ist zu verwenden, wenn $D(k) \neq 0$, $D(k) = \frac{k+1}{2}$ und $B(k) = 0$,
(3) ist zu verwenden, wenn $D(k) \neq 0$, $D(k) = \frac{k+1}{2}$ und $B(k) \neq 0$,

die Ungleichungen für $\deg \omega(x)^{(k)}$ und $\deg \gamma(x)^{(k)}$, und zwar nur für diese, modifizieren zu:

$$\begin{aligned} \deg \omega(x)^{(k)} &\leq D(k) - B(k) \text{ und} \\ \deg \gamma(x)^{(k)} &\leq k - D(k) - (1 - B(k)) \end{aligned}$$

Das ist möglich, da wir auf Grund der Initialisierung von $\gamma(x)^{(0)} = 0$, den Grad $\deg \gamma(x)^{(0)} = \deg 0 = -\infty$ haben. Damit ist eine Unterscheidung für alle vorkommenden Fälle möglich und wir können den Algorithmus 3 nun als Ganzes formulieren. Dieser ermöglicht uns die Polynome $\sigma(x)$ und $\omega(x)$ mit minimalem Grad zu finden, sodass die Schlüsselgleichung erfüllt ist.

Satz 16. Sollte sich das ausgelesene Wort $\beta(x)$ an nur $e \leq t = \lfloor \frac{n-k}{2} \rfloor$ Stellen von einem Codewort $\chi(x)$ unterscheiden, so liefert uns Algorithmus 3 das eindeutige Fehlerlokalisierungspolynom $\sigma(x)$ mit dessen Hilfe $\chi(x)$ berechnet werden kann.

Beweis. Im Zuge der Herleitung des BM Algorithmus haben wir gesehen, dass uns dieser das $\sigma(x)$ und $\omega(x)$ mit niedrigstem Grad liefert, sodass die Schlüsselgleichung von Berlekamp erfüllt ist. Zu zeigen ist allerdings noch, dass dieses $\sigma(x)$ unter der Voraussetzung $e \leq t$, unser eindeutiges Fehlerlokalisierungspolynom ist.

Dazu machen wir einen kleinen Umweg über die LFSR („linear feedback shift register“). Zuerst zeigen wir, dass unsere Syndromfolge $(s_1, s_2, \dots, s_{2t})$ von einem LFSR $(\sigma_0, \sigma_1, \dots, \sigma_e)$ der Länge e , bestehend aus den Koeffizienten von dem durch den BM Algorithmus berechneten $\sigma(x)$, erzeugt wird. Im Anschluss weisen wir nach, dass für den Fall $e \leq t$ dieses LFSR bis auf einen konstanten Faktor eindeutig, sowie das kürzestmögliche ist (siehe auch [4, Seite 52-54]). Nachdem uns der BM Algorithmus

Algorithm 3 Algorithmus zur Berechnung von $\sigma(x)$ und $\omega(x)$

Input: $s(x)$

Output: $\sigma(x) = \sigma(x)^{(2t)}$, $\omega(x) = \omega(x)^{(2t)}$

Wir initialisieren $\sigma^{(0)} = 1$, $\tau^{(0)} = 1$, $\omega^{(0)} = 1$, $\gamma^{(0)} = 0$, $D(0) = 0$ und $B(0) = 0$

for $k = 0$ bis $2t - 1$ **do**

$\Delta^{(k)} \leftarrow$ Koeffizient von x^{k+1} in $(1 + s(x))\sigma(x)^{(k)}$

$\sigma(x)^{(k+1)} \leftarrow \sigma(x)^{(k)} - \Delta^{(k)}x\tau(x)^{(k)}$

$\omega(x)^{(k+1)} \leftarrow \omega(x)^{(k)} - \Delta^{(k)}x\gamma(x)^{(k)}$

if $\Delta^{(k)} = 0$, oder $D(k) > \frac{k+1}{2}$, oder $\Delta^{(k)} \neq 0$ und $D(k) = \frac{k+1}{2}$ und $B(k) = 0$ **then**

$D(k+1) \leftarrow D(k)$

$B(k+1) \leftarrow B(k)$

$\tau(x)^{(k+1)} \leftarrow x\tau(x)^{(k)}$

$\gamma(x)^{(k+1)} \leftarrow x\gamma(x)^{(k)}$

else

$D(k+1) \leftarrow k + 1 - D(k)$

$B(k+1) \leftarrow 1 - B(k)$

$\tau(x)^{(k+1)} \leftarrow \frac{\sigma(x)^{(k)}}{\Delta^{(k)}}$

$\gamma(x)^{(k+1)} \leftarrow \frac{\omega(x)^{(k)}}{\Delta^{(k)}}$

end if

end for

das Polynom $\sigma(x)$ mit $\sigma_0 = 1$ liefert, muss dieses eindeutig sein.

Betrachten wir kurz die erste Aussage: Auf Grund der Definition von $\sigma(x)$ wissen wir bereits, dass

$$\sigma(x) = \prod_{i=1}^e (1 - X_i x) = 1 + \sigma_1 x + \dots + \sigma_e x^e$$

und e die Anzahl an Fehlern ist. Wenn wir hier nun eine Nullstelle X_k^{-1} mit $k \in \{1, 2, \dots, e\}$ einsetzen, dann erhalten wir die Gleichung:

$$1 + \sigma_1 \frac{1}{X_k} + \dots + \sigma_e \frac{1}{X_k^e} = 0$$

Diese können wir mit $Y_k X_k^{j+e}$ multiplizieren, wobei wir in Hinblick auf die gleich folgende Summation $j = 1, 2, \dots, 2t - e$ fordern. Da wir diese Gleichung für alle Nullstellen X_k aufstellen können, können wir diese auch gleich aufsummieren und erhalten:

$$\sum_{k=1}^e Y_k X_k^{j+e} + \sigma_1 \sum_{k=1}^e Y_k X_k^{j+e-1} + \dots + \sigma_e \sum_{k=1}^e Y_k X_k^j = 0$$

Und damit können wir schlussfolgern, dass

$$s_{j+e} + \sigma_1 s_{j+e-1} + \dots + \sigma_e s_j = 0$$

für alle $j = 1, 2, \dots, 2t - e$ ist, wodurch wir gezeigt haben, dass $(s_1, s_2, \dots, s_{2t})$ von einem LFSR der Länge e mit $\sigma_0 = 1$ erzeugt wird. Zusätzlich wissen wir, dass dieses LFSR aus den Koeffizienten von $\sigma(x)$, also $(1, \sigma_1, \dots, \sigma_e)$ besteht, wenn wir dieses wie

zu Beginn des Kapitels definiert wählen.

Dieses ist, wenn wir fordern, dass $e \leq t$ ist, bis auf einen konstanten Faktor eindeutig, wie man im zweiten Teil des Beweises in [4, Seite 54] nachlesen kann. Wenn wir jedoch zusätzlich fordern, dass $\sigma_0 = 1$ sein soll, dann sehen wir, dass das LFSR eindeutig sein muss.

Wie schon zu Beginn des Beweises erwähnt, liefert uns der BM Algorithmus dieses eindeutige $\sigma(x)$. Sofern $\deg(\sigma(x)) = e \leq t$ ist, muss es sich bei den Koeffizienten von $\sigma(x)$ also um ein eindeutiges LFSR handeln, das die Schlüsselgleichung erfüllt und Nullstellen bei allen Fehlerorten X_k hat. Demnach liefert uns der BM Algorithmus bei $e \leq t$ Fehlern unser gesuchtes Fehlerlokalisierungspolynom $\sigma(x)$. \square

Nachdem wir jetzt die Polynome $\sigma(x)$ und $\omega(x)$ haben, die die Schlüsselgleichung erfüllen, können wir uns überlegen, wie wir uns berechnen können, an welchen Stellen X_k Fehler aufgetreten sind und welche Fehler Y_k (siehe dazu auch [1, Seite 220]) begangen wurden. Diese Werte helfen uns dabei das vermeintliche Codewort $\chi(x)$ wiederherzustellen, wie wir im weiteren Verlauf dieses Kapitels sehen werden. Dazu betrachten wir die berechneten Polynome:

$$\begin{aligned}\sigma(x) &= \prod_{i=1}^e (1 - X_i x) = 1 + \sum_{j=1}^e \sigma_j x^j \\ \omega(x) &= \sigma(x) + \sum_{i=1}^e X_i Y_i x \prod_{j=1, j \neq i}^e (1 - X_j x)\end{aligned}$$

Wie wir sehen, hat $\sigma(x)$ nur einfache Nullstellen, weshalb wir mit Hilfe der Chien Suche (Kapitel 5.3) diese berechnen können. Auf Grund der Definition von $\sigma(x)$ können wir uns alle Fehlerorte berechnen, indem wir die Nullstellen invertieren. Mit anderen Worten ist $\sigma(X_k^{-1}) = 0$ für alle $k = 1, 2, \dots, e$ und zwar nur für diese. Damit sind die Stellen an denen ein Fehler aufgetreten ist schnell berechnet. Um nun herauszufinden, welche Fehler Y_k passiert sind, betrachten wir das Fehlerwertepolynom $\omega(x)$. Einsetzen von X_k^{-1} , also den Nullstellen von $\sigma(x)$, liefert uns:

$$\begin{aligned}\omega(X_k^{-1}) &= \sigma(X_k^{-1}) + \sum_{i=1}^e X_i Y_i X_k^{-1} \prod_{j=1, j \neq i}^e (1 - X_j X_k^{-1}) \\ &= 0 + X_k Y_k X_k^{-1} \prod_{j=1, j \neq k}^e (1 - X_j X_k^{-1}) \\ &= Y_k \prod_{j=1, j \neq k}^e (1 - X_j X_k^{-1})\end{aligned}$$

Wie wir sehen, fällt uns nicht nur $\sigma(X_k^{-1})$ weg, sondern auch alle bis auf einen Summanden, da es immer ein j gibt, für das $(1 - X_j X_k^{-1}) = 0$ falls $i \neq k$ ist. Wir können uns

jetzt Y_k berechnen, indem wir die obige Gleichung umformen zu:

$$\begin{aligned} Y_k &= \frac{\omega(X_k^{-1})}{\prod_{j=1, j \neq k}^e (1 - X_j X_k^{-1})} \\ &= \frac{X_k^e \omega(X_k^{-1})}{X_k^e \prod_{j=1, j \neq k}^e (1 - X_j X_k^{-1})} \\ &= \frac{\hat{\omega}(X_k)}{X_k \prod_{j=1, j \neq k}^e (X_k - X_j)}, \end{aligned}$$

wobei wir mit dem berechneten $\omega(x) = \sum_{i=0}^e \omega_i x^i$ das reziproke Polynom

$$\hat{\omega}(x) = \sum_{i=0}^e \omega_{e-i} x^i$$

bestimmen können.

Somit erhalten wir das Codewort $\chi(x)$, indem wir das ausgelesene Wort $\beta(x) = \sum_{i=1}^n \beta_i x^{i-1}$, durch Subtrahieren des Fehlers Y_k von β_{j_k} , wobei $X_k = \alpha^{j_k-1}$ ist, korrigieren:

$$\chi(x) = \beta(x) - \sum_{k=1}^e Y_k x^{j_k-1} \text{ mit } X_k = \alpha^{j_k-1}, k = 1, 2, \dots, e$$

Nachdem wir nun unser Codewort $\chi(x)$ berechnet haben, erhalten wir das Datenwort $(b_i)_{i=1}^k$, indem wir, wie zu Beginn dieses Kapitels beschrieben, $\chi(x)$ durch $g(x)$ dividieren und die Koeffizienten ablesen:

$$f_b(x) = \frac{\chi(x)}{g(x)}$$

Und:

$$f_b(x) = \sum_{i=1}^k b_i x^{i-1}$$

5.2.1 Forneys Algorithmus

Eine Vereinfachung der Fehlerwerteberechnung für die Y_k ist der Algorithmus von Forney [5]. Er reduziert die Anzahl an Rechenoperationen, die wir zur Bestimmung der Y_k benötigen. Dazu hat Forney die Schlüsselgleichung leicht modifiziert, weshalb wir diese noch einmal betrachten müssen.

An dem Fehlerlokalisierungspolynom

$$\sigma(x) = \prod_{i=1}^e (1 - X_i x) = 1 + \sum_{j=1}^e \sigma_j x^j$$

ändert sich nichts, allerdings verwenden wir eine leicht modifizierte Potenzreihe für die Syndrome $s_i = \beta(\alpha^i) = \sum_{j=1}^e Y_j X_j^i$:

$$\tilde{s}(x) = \sum_{i=1}^{\infty} s_i x^{i-1}$$

Es ist also $\tilde{s}(x) \cdot x = s(x)$. Wir betrachten, ähnlich wie zuvor, dieses Mal nur das Polynom $\tilde{s}(x) \pmod{x^{2t}}$ anstatt $s(x) \pmod{x^{2t+1}}$ und haben daher ein Polynom vom Grad $2t - 1$ und nicht mehr $2t$. Betrachten wir erneut die Herleitung der Schlüsselgleichung, dann sehen wir, dass sich diese wegen $\tilde{s}(x) \pmod{x^{2t}} = \frac{1}{x} s(x) \pmod{x^{2t+1}}$ ändert zu:

$$\begin{aligned}\tilde{s}(x) &= \sum_{i=1}^{\infty} s_i x^{i-1} \\ &= \sum_{i=1}^{\infty} \sum_{j=1}^e Y_j X_j^i x^{i-1} \\ &= \sum_{j=1}^e Y_j \sum_{i=1}^{\infty} X_j^i x^{i-1} \\ &= \sum_{j=1}^e Y_j X_j \sum_{i=1}^{\infty} (X_j x)^{i-1} \\ &= \sum_{j=1}^e Y_j X_j \frac{1}{1 - X_j x}\end{aligned}$$

Multiplikation dieser Gleichung mit $\sigma(x)$ liefert uns:

$$\tilde{s}(x)\sigma(x) = \sum_{j=1}^e Y_j X_j \frac{1}{1 - X_j x} \sigma(x) = \sum_{j=1}^e Y_j X_j \prod_{i=1, i \neq j}^e (1 - X_i x)$$

Diese rechte Seite bezeichnen wir nun mit $\tilde{\omega}(x)$.

Nachdem wir auch hier keine komplette Potenzreihe von $\tilde{s}(x)$ kennen, müssen wir für die Schlüsselgleichung erneut eine Reduktion vornehmen:

$$\tilde{s}(x)\sigma(x) = \tilde{\omega}(x) \pmod{x^{2t}}$$

Auch für diese Gleichung können wir wieder mit dem Algorithmus 3 die Polynome $\sigma(x)$ und $\tilde{\omega}(x)$ berechnen. Nachdem wir bei dieser Variante allerdings die Addition von $\sigma(x)$ auf beiden Seiten weggelassen haben, muss Algorithmus 3 leicht modifiziert werden. Zum einen ändern sich die Startwerte für $\tau^{(0)} = \frac{1}{s_1}$ und $\omega^{(0)} = s_1$, zum anderen läuft die FOR-Schleife nur bis $2t - 2$, da der Grad von $\tilde{s}(x)$ nur $2t - 1$ ist.

Die Berechnung der Nullstellen von $\sigma(x)$, also der Stellen an denen Fehler aufgetreten sind, kann genauso wie zuvor mit Hilfe der Chien Suche durchgeführt werden. Das Bestimmen der Y_k , also der gemachten Fehler, findet jetzt jedoch etwas anders statt. Dazu setzen wir X_k^{-1} in unser neues Fehlerwertepolynom $\tilde{\omega}(x)$ ein:

$$\begin{aligned}\tilde{\omega}(X_k^{-1}) &= \sum_{j=1}^e Y_j X_j \prod_{i=1, i \neq j}^e (1 - X_i X_k^{-1}) \\ &= Y_k X_k \prod_{i=1, i \neq k}^e (1 - X_i X_k^{-1})\end{aligned}$$

Auch hier sehen wir, dass alle bis auf einen Summanden wegfallen müssen. Umformen liefert uns dann:

$$Y_k = \frac{\tilde{\omega}(X_k^{-1})}{X_k \prod_{i=1, i \neq k}^e (1 - X_i X_k^{-1})}$$

Dieser Bruch erinnert uns sehr an die Berechnung von Y_k mit Hilfe des BM Algorithmus, doch wir werden den Nenner noch etwas modifizieren, denn aus $\sigma(x) = \prod_{i=1}^e (1 - X_i x)$ erhalten wir durch Differenzieren:

$$\sigma'(x) = \sum_{i=1}^e \left(-X_i \prod_{j=1, j \neq i}^e (1 - X_j x) \right)$$

Auswerten bei X_k^{-1} liefert uns dann

$$\sigma'(X_k^{-1}) = \sum_{i=1}^e \left(-X_i \prod_{j=1, j \neq i}^e (1 - X_j X_k^{-1}) \right) = -X_k \prod_{j=1, j \neq k}^e (1 - X_j X_k^{-1})$$

was wir leicht umformen können zu:

$$X_k \prod_{j=1, j \neq k}^e (1 - X_j X_k^{-1}) = -\sigma'(X_k^{-1})$$

Damit reduziert sich die Berechnung des Fehlers auf:

$$Y_k = \frac{\tilde{\omega}(X_k^{-1})}{-\sigma'(X_k^{-1})}$$

Eine ähnliche Substitution ist natürlich auch beim ursprünglichen BM Algorithmus möglich.

Der Vorteil bei dieser Substitution liegt in der Berechnung von

$$\sigma'(x) = (1 + \sum_{i=1}^e \sigma_i x^i)' = \sum_{i=1}^e i \sigma_i x^{i-1}.$$

Bei der Multiplikation mit i handelt es sich um eine i -fache Addition. Da jedoch in Galoisfeldern, in denen q eine Potenz von 2 ist, alle Elemente additiv selbst invers sind (deswegen können wir das Minus im Nenner von Y_k ignorieren), bleiben nach Differentiation nur die ungeraden Koeffizienten erhalten:

$$\sigma'(x) = \sum_{i=1}^{\lceil \frac{e}{2} \rceil} \sigma_{2i-1} x^{2i-2} = \sum_{i=0}^{\lfloor \frac{e-1}{2} \rfloor} \sigma_{2i+1} x^{2i}$$

Die Ersparnis liegt damit bei $\lfloor \frac{e+2}{2} \rfloor$ Additionen und Multiplikationen, wenn zum Auswerten der Polynome z.B. das Horner Schema verwendet wird.

Die restliche Vorgehensweise zum Korrigieren und Decodieren des ausgelesenen Worts erfolgt dann analog zum BM Algorithmus.

5.2.2 Decodierung von Auslöschungen

Wie wir bereits im Kapitel über Auslöschungen beim GS Algorithmus gesehen haben, gehören diese gesondert betrachtet. Die Zusatzinformation, dass an einer gewissen Stelle ein Buchstabe fehlt, ist wertvoll und kann in die Decodierung mit eingebaut

werden.

Nachdem uns der BM Algorithmus ein Codewort liefert, das wir noch durch $g(x)$ dividieren müssen, um das Datenwort zu erhalten, können wir hier leider nicht auf die Auslöschungen verzichten. Ein „Wegfallen lassen“ von Teilen des ausgelesenen Wortes, so wie wir es beim GS Algorithmus vorgenommen haben, ist also nicht durchführbar.

Gehen wir also davon aus, dass wir ϵ Stellen kennen an denen Auslöschungen statt gefunden haben. Wir setzen an diesen Stellen beliebige Buchstaben in $\beta(x)$ ein. Dadurch begehen wir zwar mit großer Wahrscheinlichkeit einen Fehler, allerdings ist es uns jetzt möglich, die bereits getätigten Überlegungen auf das Fehlerlokalisierungspolynom, von dem wir einen Teil der Nullstellen kennen, anzuwenden (siehe auch [1, Seite 229-231]):

$$\sigma(x) = \prod_{i=1}^{e+\epsilon} (1 - X_i x) = \lambda(x)\mu(x)$$

Wobei wir mit

$$\lambda(x) = \prod_{i=1}^e (1 - X_i x)$$

das Produkt der unbekanntenen Stellen an denen Fehler begangen wurden und mit

$$\mu(x) = \prod_{i=1}^{\epsilon} (1 - X_i x)$$

das bekannte Produkt der Auslöschungen an den Stellen X_i bezeichnen. Insgesamt haben wir also $e + \epsilon$ Fehler begangen, wobei wir von den ϵ Fehlern (eigentlich Auslöschungen) wissen an welchen Stellen sie auftreten. Die e Nullstellen von $\lambda(x)$ bezeichnen wir als echte Fehler, die ϵ Nullstellen von $\mu(x)$ sind die Auslöschungen.

Wir können nun die Syndrome s_i wieder genauso berechnen wie zuvor:

$$s_i = \beta(\alpha^i) = \sum_{j=0}^{n-1} e_{j+1} (\alpha^i)^j = \sum_{j=0}^{n-1} e_{j+1} (\alpha^j)^i = \sum_{k=1}^{e+\epsilon} Y_k X_k^i$$

Damit bleibt uns nur noch zu überlegen, wie sich die Kenntnis der Auslöschungen auf die Modifikation der Schlüsselgleichung auswirkt. Wir kennen ja bereits $\mu(x)$, das ein Teil von $\sigma(x)$ ist:

$$\begin{aligned} (1 + s(x)) \sigma(x) &= \omega(x) \pmod{x^{2t+1}} \\ (1 + s(x)) \lambda(x)\mu(x) &= \omega(x) \pmod{x^{2t+1}} \end{aligned}$$

Wie wir sehen, können wir auf der linken Seite der Gleichung das Produkt $(1 + s(x))\mu(x) \pmod{x^{2t+1}}$ berechnen. Dieses Produkt ist bekannt als das Polynom der Forney T's T_k (siehe [1, Seite 230]), die definiert sind als:

$$1 + \sum_{k=1}^{2t} T_k x^k := \left(1 + \sum_{i=1}^{2t} s_i x^i \right) \mu(x) \pmod{x^{2t+1}}$$

Wir können die Schlüsselgleichung also umschreiben zu

$$\left(1 + \sum_{k=1}^{2t} T_k x^k\right) \lambda(x) = \omega(x) \pmod{x^{2t+1}}$$

wobei $\deg \lambda = e$ und $\deg \omega = e + \epsilon$ ist.

Umformen dieser Gleichung liefert uns:

$$\begin{aligned} \left(1 + \sum_{k=1}^{\epsilon} T_k x^k + \sum_{k=\epsilon+1}^{2t} T_k x^k\right) \lambda(x) &= \omega(x) \pmod{x^{2t+1}} \\ \left(\sum_{k=\epsilon+1}^{2t} T_k x^k\right) \lambda(x) &= \omega(x) - \lambda(x) \left(1 + \sum_{k=1}^{\epsilon} T_k x^k\right) \pmod{x^{2t+1}} \end{aligned}$$

Daraus können wir nun schlussfolgern, dass, nachdem die linke Seite der Gleichung durch $x^{\epsilon+1}$ teilbar ist, auch die rechte Seite durch $x^{\epsilon+1}$ geteilt werden kann. Definieren wir nun

$$\eta(x) := \frac{\omega(x) - \left(1 + \sum_{k=1}^{\epsilon} T_k x^k\right) \lambda(x)}{x^{\epsilon}} + \lambda(x) = \sum_{i=0}^{\tau} \eta_i x^i,$$

dann sehen wir, dass wir die umgeformte Schlüsselgleichung weiter modifizieren können zu:

$$\begin{aligned} \left(\sum_{k=\epsilon+1}^{2t} T_k x^k\right) \lambda(x) &= x^{\epsilon} (\eta(x) - \lambda(x)) \pmod{x^{2t+1}} \\ \left(\sum_{k=\epsilon+1}^{2t} T_k x^{k-\epsilon}\right) \lambda(x) &= \eta(x) - \lambda(x) \pmod{x^{2t+1-\epsilon}} \\ \left(1 + \sum_{k=\epsilon+1}^{2t} T_k x^{k-\epsilon}\right) \lambda(x) &= \eta(x) \pmod{x^{2t+1-\epsilon}} \\ \left(1 + \sum_{k=1}^{2t-\epsilon} T_{k+\epsilon} x^k\right) \lambda(x) &= \eta(x) \pmod{x^{2t+1-\epsilon}} \end{aligned}$$

Nachdem wir wissen, dass $\deg \eta(x) \leq \deg \lambda(x) = e$ ist, sehen wir, dass wir diese neue Schlüsselgleichung mit dem Algorithmus 3 lösen können, sofern $e \leq \frac{2t-\epsilon}{2}$ ist. Wo bei $t = \lfloor \frac{n-k}{2} \rfloor$ die maximale Anzahl an korrigierbaren Fehlern bei 0 Auslöschungen ist.

Wir sehen also, dass wir im Fall von Auslöschungen zwar weniger echte Fehler e , jedoch insgesamt mehr fehlerbehaftete Stellen $e + \epsilon$ (Fehler und Auslöschungen) korrigieren können:

$$\begin{aligned} e &\leq \frac{2t - \epsilon}{2} \\ e + \frac{\epsilon}{2} &\leq t \end{aligned}$$

Eine Auslöschung wirkt sich demnach nur halb so stark auf die Korrektoreigenschaften aus wie ein echter Fehler. Für die beiden Parameter e und ϵ bedeutet das, dass

$$e \leq t - \frac{\epsilon}{2}, \text{ bzw.} \\ \epsilon \leq 2(t - e)$$

ist. Rufen wir uns in Erinnerung, dass die Minimaldistanz $d = n - k + 1$ ist, dann sehen wir, dass außerdem

$$2e + \epsilon < d, \text{ also } 2e + \epsilon \leq n - k$$

gelten muss.

Für den Fall, dass in Summe nicht zu viele Fehler und Auslöschungen geschehen sind, können wir nun $\lambda(x)$ und $\eta(x)$ berechnen. Mit Hilfe der Chien Suche können nun wieder die Nullstellen von $\lambda(x)$ berechnet werden. Wir kennen nun alle Stellen X_k , an denen Fehler im ausgelesenen Wort auftreten und müssen nur noch die Fehlerwerte Y_k berechnen. Dies geschieht indem wir uns mit Hilfe der Definition von $\eta(x)$ das Fehlerwertepolynom $\omega(x)$ ausdrücken:

$$\omega(x) = \eta(x)x^\epsilon + \left(1 + \sum_{k=1}^{\epsilon} T_k x^k - x^\epsilon\right) \lambda(x)$$

Das Bestimmen der Y_k kann nun mit den bereits beschriebenen Methoden (entweder wie im Kapitel 5.2, oder mit Forney's Algorithmus aus Kapitel 5.2.1) durchgeführt werden.

5.3 Chien Suche

Bei der bereits erwähnten Chien Suche handelt es sich um eine elegante Methode zur Berechnung der Nullstellen von $\sigma(x)$. Wie in [1, Kapitel 5.4] nachzulesen ist, leistet dieser Algorithmus über $GF(2)$ noch deutlich mehr. In diesem Fall kann die Fehlerkorrektur zeitgleich mit der Berechnung der Nullstellen von $\sigma(x)$ mit Hilfe von Schieberegistern realisiert werden. Ausführliche Informationen zu rückgekoppelten Schieberegistern findet man z.B. in [9, Kapitel 8].

In unserem Fall müssen wir zuerst die Fehlerorte berechnen, bevor wir die Fehler korrigieren können. Gehen wir davon aus, dass wir die Koeffizienten von $\sigma(x) = 1 + \sum_{j=1}^e \sigma_j x^j$ bereits berechnet haben und wir nun die Nullstellen bestimmen wollen. Da $\sigma(x) = \prod_{i=1}^e (1 - X_i x)$ ist, wissen wir, dass $\sigma(X_i^{-1}) = 0$ gleichbedeutend damit ist, dass an der Stelle j_i ein Fehler auftritt. Wenn also $\sigma(a) = 0$ ist, dann ist $a^{-1} = X_i$ einer der gesuchten Fehlerorte.

Da wir nicht wissen, an welcher Stelle die Fehler aufgetreten sind, müssen wir für jedes Element $a \in GF(q)$ den Wert $\sigma(a)$ berechnen. Die Chien Suche ist eine praktische Methode, um diese Auswertung von $\sigma(x)$ effizient zu gestalten. Nachdem wir wissen, dass sich alle Elemente eines Galoisfelds als Potenz eines primitiven Elements darstellen lassen, werten wir $\sigma(x)$ bei allen a^i für $i = 0, 1, \dots, q - 2$ aus. Dies hat, im binären Fall, den Vorteil, dass eine sofortige Fehlerkorrektur möglich ist, da ein Fehler an der i -ten Stelle bedeutet, dass das $n - i$ -te Bit umgedreht werden muss. Mehr

Informationen dazu findet man in [1, Kapitel 5.4]. Uns interessiert nur die Auswertung $\sigma(\alpha^i)$. Die Berechnung der Fehlerwerte Y_i erfolgt nach Überprüfung, ob $\sigma(x)$ überhaupt $\deg(\sigma(x))$ verschiedene Nullstellen hat.

Wir starten indem wir $\sigma(\alpha^0)$ berechnen, danach $\sigma(\alpha^1)$, $\sigma(\alpha^2)$, usw. Der Algorithmus bricht ab, wenn wir entweder $\deg(\sigma(x))$ Nullstellen berechnet haben, oder wir bei α^{n-1} angekommen sind. Dabei werden die Zwischenergebnisse der aktuellen Auswertung für den nächsten Auswertungsschritt gespeichert:

$$\begin{aligned}\sigma(\alpha^0) &= \sum_{j=0}^e \sigma_j = \sigma_0 + \sigma_1 + \dots + \sigma_e \\ \sigma(\alpha^1) &= \sum_{j=0}^e \sigma_j \alpha^j = \sigma_0 + \sigma_1 \alpha + \dots + \sigma_e \alpha^e = \\ &= \sigma_0 + \sigma_1 \cdot \alpha + \dots + \sigma_e \cdot \alpha^e \\ \sigma(\alpha^2) &= \sum_{j=0}^e \sigma_j \alpha^{2j} = \sigma_0 + \sigma_1 \alpha^2 + \dots + \sigma_e \alpha^{2e} = \\ &= \sigma_0 + \sigma_1 \alpha \cdot \alpha + \dots + \sigma_e \alpha^e \cdot \alpha^e \\ &\vdots\end{aligned}$$

Wie wir sehen, reicht es zur Berechnung der nächsten Auswertung, wenn wir uns die Summanden im i -ten Auswertungsschritt merken (in einem Vektor speichern) und diese dann mit dem Vektor $(1, \alpha, \alpha^2, \dots, \alpha^e)$ elementweise multiplizieren und anschließend die Werte addieren. Sollte $\sigma(\alpha^l) = 0$ sein, wissen wir dass α^{-l} ein Fehlerort ist.

6 Eigenschaften

Wie wir bei der Vorstellung des GS Algorithmus in Kapitel 5.1 gesehen haben, ist dieser abhängig von einem frei wählbaren Parameter m , der die Decodierungseigenschaften sowie die Laufzeit des Algorithmus massiv beeinflussen kann. Und auch der BM Algorithmus wirft noch ein paar Fragen auf. In diesem Kapitel betrachten wir diese Eigenschaften etwas genauer.

6.1 Eigenschaften GS Algorithmus

Zu überlegen ist noch, welchen Einfluss die Wahl von m auf die Decodierung von RS Codes hat. Hierzu betrachten wir drei verschiedene Parameter. Zum einen die Mindestanzahl korrekt übertragener Stellen K_m , damit das Datenwort $f_b(x)$ in der ermittelten Lösungsmenge liegt. Dann die maximale Anzahl an Fehlern t_m die gemacht werden dürfen. Und schließlich die maximale Anzahl an möglichen Lösungen L_m die uns der Faktorisierungsalgorithmus liefert.

Durch Vergrößern von m kann man nur bedingt die Korrektüreigenschaften verbessern. Wünschenswert ist es natürlich, dass die K_m und L_m möglichst klein sind, während t_m sehr groß sein soll. In manchen Fällen gibt es durch Vergrößern von m keine Änderung der Decodierungseigenschaften.

Wegen des Algorithmus 2 wissen wir, dass wir den für die Ordnung der Polynome benötigten Parameter v festlegen müssen mit $v = k - 1$.

Überlegen wir uns zuerst, wie viele korrekte Stellen wir brauchen, damit das gesuchte Datenwort Teil der Lösung ist. Wegen des Faktorisierungstheorems (Satz 12) ist das Datenwort im Output, wenn $\sum_{a \in GF(q)} \text{ord}(Q : a, f(a)) > \deg_{1,v} Q(x, y)$. Wir müssen uns also überlegen, wie viele Stellen K_m mindestens korrekt sein müssen, damit diese Bedingung für unser Datenwort erfüllt ist. Dazu definieren wir:

$$K := \left| \left\{ i \in \{1, 2, \dots, n\} : f(\alpha^{i-1}) = \beta_i \right\} \right|$$

Wir wissen, dass wir im Fall eines korrekt ausgelesenen Buchstaben, eine m -fache Nullstelle für das Paar (α^{i-1}, β_i) haben. Für unser gesuchtes K_m gilt also, dass $mK_m > \deg_{(1,v)} Q(x, y)$ (wobei $\deg_{(1,v)} Q(x, y) \leq \deg_{(1,v)} \phi_C$) sein muss, vorausgesetzt $Q(x, y)$ ist bezüglich der $(1, v)$ -revlex Ordnung geordnet. Diese Aussage ist gleichbedeutend zu $\text{Rang}_v(x^{mK_m}) > C = n \binom{m+1}{2}$, wie man schnell aus den Definitionen sieht:

Angenommen $mK_m > \deg_{(1,v)} \phi_C$, dann muss der Rang $_v$ von x^{mK_m} , dessen Grad ja $1 \cdot (mK_m) + 0 \cdot v$ ist, größer sein, als der Rang von ϕ_C , der in der $(1, v)$ -revlex Ordnung ja gleich C ist.

Umgekehrt, wenn $\text{Rang}_v(x^{mK_m}) > C$ wissen wir dass $\deg_{(1,v)} x^{mK_m} \geq \deg_{(1,v)} \phi_C$ sein muss. Da aber x^{mK_m} das erste Monom vom Grad mK_m ist, und ϕ_C vor diesem Monom liegt, muss $\deg_{(1,v)} \phi_C < \deg_{(1,v)} x^{mK_m} = mK_m$ sein.

Es gilt nun:

$$\begin{aligned} K_m &= \min \left\{ K : mK > \deg_{(1,v)} \phi_C \right\} \\ &= \min \left\{ K : \text{Rang}_v(x^{mK}) > C \right\} \end{aligned}$$

Wir suchen also das kleinste K (das wir dann mit K_m bezeichnen werden), sodass $0 < m < \dots < (K_m - 1)m \leq C < K_m m$. Wenn wir statt dieser Folge, die Folge von $\text{Rang}_v(x^j)$ für $j = 0, 1, \dots$ betrachten (also $0 < 1 < \dots < (K_m - 1)m \leq \dots \leq R \leq C < R + 1 \leq \dots \leq K_m m$), dann sehen wir, dass wir dieses Minimum schreiben können als:

$$K_m = \min \left\{ K : \text{Rang}_v(x^{mK}) > C \right\} = \left\lfloor \frac{R}{m} \right\rfloor + 1,$$

wobei $R = \max \left\{ K : \frac{K^2}{2v} + \frac{K}{2} + \frac{(v-r)r}{2v} \leq C \right\}$ mit $r = K \bmod v$, was wir aus Satz 8 bereits kennen. K_m lässt sich dann durch Lösen der Ungleichung für R abschätzen.

$$\frac{K^2}{2v} + \frac{K}{2} + \frac{(v-r)r}{2v} \leq C = n \binom{m+1}{2}$$

$$K^2 + vK + (v-r)r - nv(m+1)m \leq 0$$

Nachdem r von K abhängt und $0 \leq r \leq v - 1$ ist, schätzen wir $(v-r)r$ mit $\frac{v^2}{4}$ nach oben und mit 0 nach unten ab. So erhalten wir folgende Abschätzungen für R :

$$\sqrt{nv(m+1)m} - \frac{v}{2} \leq R \leq \sqrt{nv(m+1)m} + \frac{v^2}{4} - \frac{v}{2}$$

Damit ergibt sich für $K_m = \left\lfloor \frac{R}{m} \right\rfloor + 1$, dass

$$\left\lfloor \sqrt{\frac{nv(m+1)}{m}} - \frac{v}{2m} \right\rfloor + 1 \leq K_m \leq \left\lfloor \sqrt{\frac{nv(m+1)}{m} + \frac{v^2}{4m^2}} - \frac{v}{2m} \right\rfloor + 1$$

und daher gilt für die maximale Anzahl an zu machenden Fehlern $t_m = n - K_m$:

$$n - \left\lfloor \sqrt{\frac{nv(m+1)}{m} + \frac{v^2}{4m^2}} - \frac{v}{2m} \right\rfloor - 1 \leq t_m \leq n - \left\lfloor \sqrt{\frac{nv(m+1)}{m}} - \frac{v}{2m} \right\rfloor - 1$$

Wir sehen, dass wir für $m \rightarrow \infty$ einen Grenzwert für die Anzahl zu korrigierender Fehler bekommen, den wir mit t_{GS} bezeichnen wollen²¹:

$$t_{GS} = n - 1 - \left\lfloor \sqrt{vn} \right\rfloor = n - 1 - \left\lfloor \sqrt{(k-1)n} \right\rfloor$$

Diesen Wert können wir mit der aus der Minimaldistanz gewonnen Fehlerkorrektureigenschaft vergleichen. Für diese galt ja $t = \left\lfloor \frac{n-k}{2} \right\rfloor$.

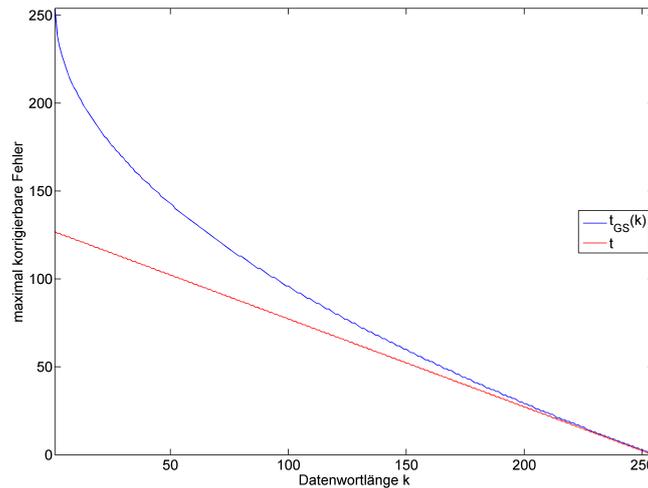


Abbildung 1: Vergleich korrigierbarer Fehler des GS Algorithmus (t_{GS}) und des BM Algorithmus ($t = \left\lfloor \frac{n-k}{2} \right\rfloor$) bei Codewortlänge $n = 255$.

Wie wir in Abbildung 1 sehen, ist bei konstanter Codewortlänge $n = 255$ die Fehlerkorrektureigenschaft des GS Algorithmus besser, je kürzer das Datenwort ist und je mehr Korrekturstellen wir haben. Für Datenwortlängen nahe der Codewortlänge ist fast kein Unterschied bemerkbar. Um jedoch bis zu t_{GS} ²² Fehler korrigieren zu können, muss der Parameter m , von dem die Anzahl korrekter Stellen K_m und daher auch $t_m = n - K_m$ abhängt, groß genug gewählt werden.

Doch warum reicht es m hinreichend groß zu wählen? Ausschlaggebend ist die Eigenschaft, dass die Folge der t_m monoton wachsend ist. Es gilt demnach

$$t_0 \leq t_1 \leq t_2 \leq \dots$$

²¹Für den GS Algorithmus muss $v = k - 1$ gewählt werden.

²²Bis zu t_{GS} Fehler können für $m \rightarrow \infty$ korrigiert werden.

bis ab einem Index m_0 gilt: $t_{m_0} = t_{m_0+1} = \dots = t_{GS}$. Bevor wir analysieren, wie groß dieses m_0 , abhängig von der Datenwortlänge k und der Codewortlänge n , sein muss um t_{GS} Fehler zu korrigieren, weisen wir noch nach, dass die Werte t_m monoton steigend sind und t_{GS} auch ab einem m_0 angenommen wird:

Anstatt der t_m betrachten wir die $K_m = n - t_m$. Die Aussage, dass die t_m monoton steigend sind, ist dann äquivalent dazu, dass die K_m monoton fallen. Wir weisen daher nach, dass

$$K_0 \geq K_1 \geq K_2 \geq \dots \geq K_{m_0} = K_{m_0+1} = \dots = K_{GS}$$

ist. Da K_m mittels $\text{Rang}_v(x^{mK})$ definiert ist, betrachten wir diese Funktion etwas genauer und formen sie um zu:

$$\begin{aligned} \text{Rang}_v(x^K) &= \frac{K^2}{2v} + \frac{K}{2} + \frac{(v-r)r}{2v} \\ &= v \cdot \frac{1}{2} \left(\left(\frac{K}{v} \right)^2 + \frac{K}{v} + \frac{r}{v} \cdot \left(1 - \frac{r}{v} \right) \right) \end{aligned}$$

Da $r = K \bmod (v)$ ist, lässt sich $\frac{r}{v}$ auch schreiben als $\left\{ \frac{K}{v} \right\}$, wenn man mit $\{x\} = x - \lfloor x \rfloor$ den gebrochenen Anteil bezeichnet. Wir können daher $\text{Rang}_v(x^K)$ weiter umformen zu

$$\begin{aligned} \text{Rang}_v(x^K) &= v \cdot \frac{1}{2} \left(\left(\frac{K}{v} \right)^2 + \frac{K}{v} + \left\{ \frac{K}{v} \right\} \cdot \left(1 - \left\{ \frac{K}{v} \right\} \right) \right) \\ &= v \cdot F\left(\frac{K}{v}\right) \end{aligned}$$

wobei wir mit $F(x)$ die Funktion

$$F(x) = \frac{1}{2} \cdot \left(x^2 + x + \{x\} \cdot (1 - \{x\}) \right)$$

bezeichnen. Mit Hilfe folgender Eigenschaften von $F(x)$ lässt sich schrittweise die Monotonie der K_m zeigen:

Lemma 8. Eigenschaften von $F(x)$ für $m \in \mathbb{N}_0$ und $x \geq 0$ (Beweis siehe [11, Anhang A]):

1. $F(x) \geq mx - \binom{m}{2}$ für alle $m \geq 1$
2. $F(x) \leq \frac{m+1}{2m} x^2$ für $x \geq m$
3. $F(mx) \leq \frac{m}{m+2} F((m+1)x)$ wenn $x \geq 1$
4. $\frac{1}{2}(x^2 + x) \leq F(x) \leq \frac{1}{2} \left(x + \frac{1}{2} \right)^2$ für alle $x > 0$

Um nachzuweisen, dass die Werte K_m monoton fallend sind und $K_{m_0} = K_{m_0+1} = \dots = K_{GS}$ ab einem hinreichend großen m_0 gilt, zeigen wir nun:

Satz 17. Die Werte K_m haben für $m \in \mathbb{N}_0$ und $K_{GS} = \lfloor \sqrt{vn} \rfloor + 1$ folgende Eigenschaften:

1. $K_0 \geq K_1$
2. $K_m \geq K_{GS}$ für alle $m \geq 1$

3. $K_m \geq K_{m+1}$ für alle $m \geq 1$
4. $K_m = K_{GS}$ für ein hinreichend großes m

Über die K_m ist bekannt ($v = k - 1$):

- $K_0 = n - t_0 = n - \lfloor \frac{n-k}{2} \rfloor = n - \lfloor \frac{n-v-1}{2} \rfloor = \lceil \frac{n+v+1}{2} \rceil$
- $K_m = \min \left\{ K : \text{Rang}_v(x^{mK}) > n \binom{m+1}{2} \right\}$ für $m \geq 1$

Beweis von Satz 17. Wir weisen Bedingung 1 der Monotonie mit Hilfe von Lemma 8, Eigenschaft 1 ($m = 2$), nach:

$$\begin{aligned}
 \text{Rang}_v(x^{K_0}) &= v \cdot F\left(\frac{K_0}{v}\right) \\
 &\stackrel{\text{Lemma 8.1}}{\geq} v \cdot \left(2 \frac{K_0}{v} - 1\right) \\
 &= 2K_0 - v \\
 &= 2 \left\lceil \frac{n+v+1}{2} \right\rceil - v \\
 &\geq n + 1
 \end{aligned}$$

Da $K_1 = \min \left\{ K : \text{Rang}_v(x^K) > n \right\}$ ist und für K_0 , wie eben gezeigt, $\text{Rang}_v(x^{K_0}) > n$ ist, ist entweder $K_1 = K_0$ oder $K_1 < K_0$.

Für Bedingung 2 der Monotonie wird Lemma 8, Eigenschaft 2 ($\frac{m(K_{GS}-1)}{v} \geq m$), verwendet. Wir zeigen $\text{Rang}_v(x^{m(K_{GS}-1)}) \leq n \binom{m+1}{2}$, was laut Definition der K_m unsere Bedingung 2 impliziert.

$$\begin{aligned}
 \text{Rang}_v(x^{m(K_{GS}-1)}) &= v \cdot F\left(\frac{m(K_{GS}-1)}{v}\right) \\
 &\stackrel{\text{Lemma 8.2}}{\leq} v \cdot \frac{m+1}{2m} m^2 \frac{(K_{GS}-1)^2}{v^2} \\
 &\leq \frac{(m+1)m}{2} \frac{(\sqrt{vn})^2}{v} \\
 &= n \binom{m+1}{2}
 \end{aligned}$$

Für Bedingung 3 der Monotonie wird Lemma 8, Eigenschaft 3 ($K_m \geq v + 1$), benötigt. Wir betrachten zuerst den Binomialkoeffizienten $n \binom{m+1}{2}$ aus der Definition von K_m :

$$\begin{aligned}
 n \binom{m+1}{2} &< \text{Rang}_v(x^{mK_m}) = v \cdot F\left(\frac{mK_m}{v}\right) \\
 &\stackrel{\text{Lemma 8.3}}{\leq} v \cdot \frac{m}{m+2} F\left((m+1) \frac{K_m}{v}\right) \\
 &= \frac{m}{m+2} \text{Rang}_v(x^{(m+1)K_m})
 \end{aligned}$$

Wir können daher folgern, dass

$$\text{Rang}_v(x^{(m+1)K_m}) > n \binom{m+1}{2} \frac{m+2}{m} = n \binom{m+2}{2}$$

ist, woraus geschlossen werden kann, dass K_m die für K_{m+1} benötigte Ungleichung $\text{Rang}_v(x^{(m+1)K_{m+1}}) > n \binom{m+2}{2}$ erfüllt, weshalb entweder $K_{m+1} = K_m$ oder $K_{m+1} < K_m$ ist.

Bleibt noch Bedingung 4. Wir wollen zeigen, dass für $m \geq m_0$ der Wert K_{GS} auch wirklich von den K_m angenommen wird. Mit anderen Worten benötigen wir ein hinreichend großes m sodass

$$\text{Rang}_v(x^{mK_{GS}}) > n \binom{m+1}{2}$$

ist. Dazu benötigen wir Lemma 8, Eigenschaft 4:

$$\begin{aligned} \text{Rang}_v(x^{mK_{GS}}) &= v \cdot F\left(\frac{mK_{GS}}{v}\right) \\ &\stackrel{\text{Lemma 8.4}}{>} \frac{m^2 K_{GS}^2}{2v} \\ &= n \frac{(m+1)m}{2} \frac{mK_{GS}^2}{(m+1)vn} \\ &= n \binom{m+1}{2} \frac{mK_{GS}^2}{(m+1)vn} \end{aligned}$$

Damit also $\text{Rang}_v(x^{mK_{GS}}) > n \binom{m+1}{2}$ ist, muss $\frac{mK_{GS}^2}{(m+1)vn} > 1$ sein. Nachdem m beliebig groß werden kann, sehen wir, dass auf Grund von

$$\frac{mK_{GS}^2}{(m+1)vn} > 1 \Leftrightarrow \frac{K_{GS}^2}{vn} > \frac{m+1}{m} = 1 + \frac{1}{m} \Leftrightarrow \left(\frac{K_{GS}^2}{vn} - 1\right)^{-1} < m$$

immer so ein m gefunden werden kann. Dieses ist unser m_0 ab dem $K_{m_0} = K_{m_0+1} = \dots = K_{GS}$ gilt.

Somit wäre gezeigt, dass K_m eine monoton fallende Folge mit Grenzwert K_{GS} ist, wobei dieser Grenzwert auch angenommen wird und daher die t_m monoton steigend sind und ebenfalls einen Grenzwert t_{GS} haben der auch ab einem m_0 angenommen wird. \square

Um nun die benötigte Größe des Parameters m_0 in Abhängigkeit von der Datenwortlänge k und des maximal korrigierbaren Fehlers t_{GS} zu bestimmen, berechnen wir

$$R = \max \left\{ K : \frac{K^2}{2 \cdot (k-1)} + \frac{K}{2} + \frac{((k-1)-r)r}{2 \cdot (k-1)} \leq n \binom{m+1}{2} \right\}$$

mit $r = K \bmod (k-1)$ für ein fixes m . Anschließend berechnen wir $K_m = \lfloor \frac{R}{m} \rfloor + 1$ und $t_m = n - K_m$ und vergleichen dieses mit t_{GS} . So lange $t_m < t_{GS}$ ist vergrößern wir m und führen die Berechnungen erneut durch. Auf Grund der gezeigten Monotonie der Folge t_m gibt es ein $m_0 \in \mathbb{N}$ sodass für alle $m \geq m_0$ gilt: $t_m = t_{GS}$. Wie wir in Abbildung 2

sehen, können diese m_0 sehr groß werden, was zu einer sehr langen Rechenzeit führen kann. Es erscheint daher eine Einschränkung der Datenwortlängen k je Codewortlänge n auf gewisse Bereiche sinnvoll, wenn man immer bis zu t_{GS} Fehler korrigieren können möchte.

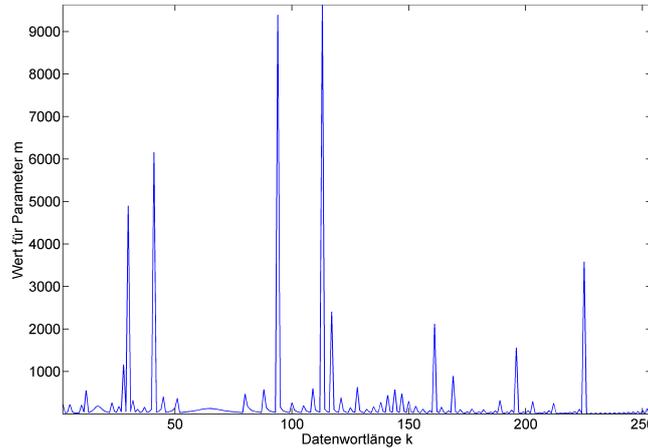


Abbildung 2: Benötigte Größe des Parameters m , um bis zu $t_{GS} = n - 1 - \lfloor \sqrt{(k-1)n} \rfloor$ Fehler bei Codewortlänge $n = 255$ zu korrigieren.

Betrachten wir noch den letzten interessanten Parameter: Die maximale Anzahl an möglichen Datenwörtern, die uns der Faktorisierungsalgorithmus liefert. Schließlich berechnet uns dieser Faktorisierungsalgorithmus nicht nur ein Datenwort, sondern gleich mehrere. Wie wir in den vorigen Kapiteln gesehen haben, gibt es für das Interpolationspolynom des GS Algorithmus einen maximalen Grad für die Potenzen von y : $\deg_{(0,1)} Q(x, y) \leq L_0$. Nachdem uns der Faktorisierungsalgorithmus jene $f(x)$ berechnet, für die $(y - f(x)) \mid Q(x, y)$ gilt, können es maximal so viele Polynome $f(x)$ sein, wie der Grad von $Q(x, y) \in (GF(q)[x])[y]$. Mit Hilfe der Funktion $\text{Rang}_v(y^L)$ lässt sich L_0 durch Lösen einer Ungleichung (analog zu vorhin) berechnen. Um zu verdeutlichen, dass die Anzahl an maximalen Lösungen L eigentlich nur von m abhängig ist (bei einer gegebenen RS Codierung sind ja die Codelänge n und die Datenwortlänge k vorgegeben), bezeichnen wir diese mit L_m . Gesucht ist also das maximale L mit $\text{Rang}_v(y^L) = \frac{vL^2}{2} + \frac{(v+2)L}{2} \leq \text{Rang}(Q(x, y))$:

$$\frac{vL^2}{2} + \frac{(v+2)L}{2} \leq C = n \binom{m+1}{2}$$

$$vL^2 + (v+2)L - n(m+1)m \leq 0$$

$$L^2 + \frac{v+2}{v}L - \frac{n(m+1)m}{v} \leq 0$$

Auch hier können wir wieder mit Hilfe der kleinen Lösungsformel folgern, dass die Nullstellen der linken Seiten lauten:

$$L_{1,2} = -\frac{v+2}{2v} \pm \sqrt{\left(\frac{v+2}{2v}\right)^2 + \frac{n(m+1)m}{v}}$$

Daher ist L_m gegeben durch

$$L_m = \left\lfloor \sqrt{\frac{n(m+1)m}{v} + \frac{(v+2)^2}{4v^2}} - \frac{v+2}{2v} \right\rfloor.$$

Wie wir sehen, strebt hier die Anzahl an Lösungen für $m \rightarrow \infty$ auch gegen unendlich. Ein willkürliches Vergrößern von m wirkt sich also negativ aus und kann dazu führen, dass eine sehr große Anzahl an Polynomen $f(x)$ durch den Faktorisierungsalgorithmus berechnet werden. Wie wir in Abbildung 3 sehen, bedeutet bereits eine Wahl von m , so dass $t_m = t_{GS}$ ist, eine teilweise sehr große Anzahl an potentiellen Datenwörtern.

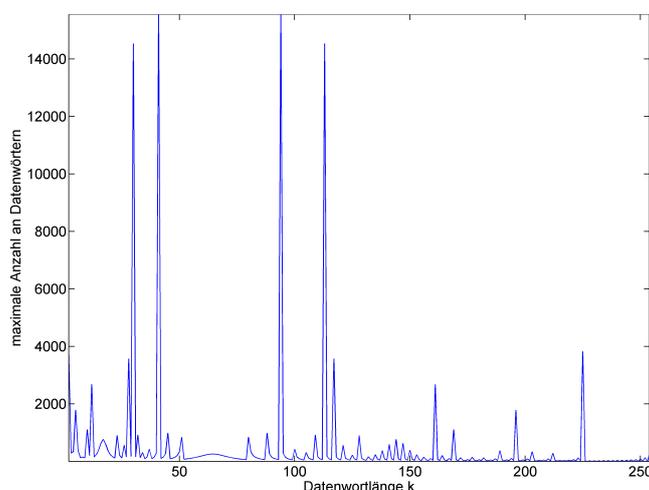


Abbildung 3: Maximale Anzahl an Datenwörtern die der GS Algorithmus liefert, wenn m so gewählt wird, dass $t_m = t_{GS}$ ist.

Wie wir gesehen haben, können wir mit Wahl der Datenwortlänge k nicht nur die Redundanz des Datenworts bestimmen, sondern auch die Qualität des GS Algorithmus beeinflussen. Natürlich ist es bei jeglicher Wahl von k immer möglich, sich auf ein m zu beschränken, so dass wir keine t_{GS} Fehler decodieren können. Der Rechenaufwand wird so verringert, allerdings verlieren wir auch die besonderen Korrektoreigenschaften die diesen Algorithmus ausmachen. In Abbildung 4 sehen wir eine Gegenüberstellung von Datenwortlängen, bei denen sowohl m , als auch L_m kleiner 50 sind. Die Codewortlänge n beträgt 255.

Ein weiterer Wert, der uns auch interessiert, ist die durchschnittliche Anzahl an Codewörtern, die wir in einer r -Kugelumgebung im Sequenzraum finden können. Im konkreten interessiert es uns natürlich zu wissen, mit wie vielen Datenwörtern wir im Durchschnitt mindestens rechnen müssen, wenn wir ein beliebiges ausgelesenes Wort $\beta(x)$ decodieren und m so wählen, dass bis zu t_{GS} Fehler korrigiert werden können. Wir wissen bereits, dass wir mit maximal L_m Datenwörtern rechnen müssen, doch ist dies natürlich nur der Worst-Case-Fall. Wir bezeichnen daher mit $\bar{L}(r)$ die durchschnittliche Anzahl an Codewörtern in einer r -Kugelumgebung in $GF(q)^n$ und halten fest:

$$\bar{L}(r) \leq \text{durchschnittliche Anzahl berechneter Datenwörter} \leq L_m$$

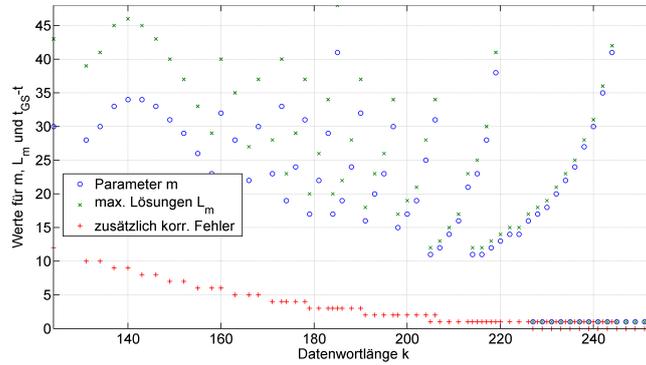


Abbildung 4: Übersicht über die Wahl der Datenwortlänge k und der von ihr abhängigen Werte m und L_m , sowie der zusätzlich korrigierbaren Fehler ($t_{GS} - t = t_{GS} - \lfloor \frac{n-k}{2} \rfloor$).

Wir wissen, dass es insgesamt q^k Codewörter in $GF(q)^n$ geben muss. Die Anzahl der Wörter in einer r -Kugelumgebung erhalten wir, indem wir uns überlegen, dass zwischen 0 und r Buchstaben abweichen dürfen, damit ein Wort noch in der Kugelumgebung ist. Das trifft auf insgesamt

$$K_r = \sum_{i=0}^r \binom{n}{i} (q-1)^i$$

Wörter zu. Betrachten wir nun alle q^n Kugelumgebungen mit Radius r , dann sehen wir, dass im Mittel jedes Codewort in K_r verschiedenen Kugeln vorkommt. Wir haben daher in Summe $q^k \cdot K_r$ Codewörter, wenn wir alle q^n Kugelumgebungen betrachten, was bedeutet, dass im Schnitt

$$\bar{L}(r) = \frac{q^k K_r}{q^n} = \frac{q^k}{q^n} \sum_{i=0}^r \binom{n}{i} (q-1)^i = q^{k-n} \sum_{i=0}^r \binom{n}{i} (q-1)^i$$

Codewörter je r -Kugelumgebung vorhanden sind.

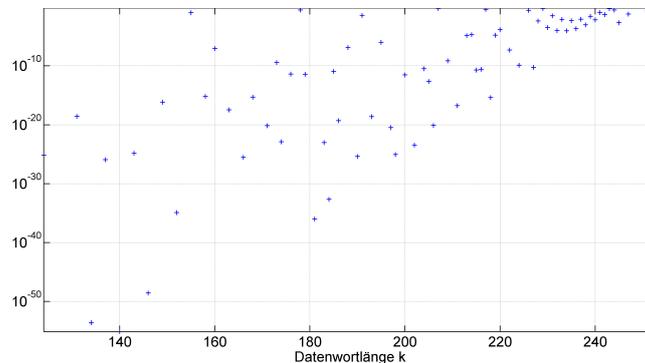


Abbildung 5: Durchschnittliche Anzahl an Codewörtern in einer t_{GS} -Kugelumgebung, wenn m und L_m kleiner 50 sind.

In Abbildung 5 sehen wir, mit wie vielen Codewörtern in einer t_{GS} -Kugelumgebung im Durchschnitt gerechnet werden muss. Berücksichtigt wurden dabei aber nur Datenwortlängen k , für die m und L_m kleiner 50 sind.

6.2 Eigenschaften BM Algorithmus

Wie wir bereits im Beweis gesehen haben, können wir mit Hilfe des BM Algorithmus bis zu $\lfloor \frac{n-k}{2} \rfloor$ Fehler korrekt decodieren. Der GS Algorithmus (wie in Kapitel 6.1 beschrieben) schafft es bei günstiger Wahl des Parameters m , mehr Fehler zu decodieren. Doch gelingt das auch dem BM Algorithmus?

Gehen wir davon aus, dass wir mehr als $t = \lfloor \frac{n-k}{2} \rfloor$ Fehler gemacht haben. Wie man in [1, Kapitel 10.5] nachlesen kann, können nun verschiedene Fälle eintreten. Es ist möglich, dass Fehlerorte und Fehlerwerte berechnet werden, die unser ausgelesenes Wort in das ursprüngliche oder ein falsches Codewort transformieren, oder uns das Zerlegen von $\sigma(x)$ in $\deg(\sigma(x))$ verschiedenen Faktoren der Form $(1 - X_i x)$ nicht möglich ist.

Ein Verändern von Parametern ist bei diesem Algorithmus nicht möglich, weshalb wir uns eigentlich nur darauf beschränken können, die ungünstigen Fälle im Vorhinein zu erkennen. Ein Korrigieren von falschen Polynomen $\sigma(x)$ und $\omega(x)$, bzw. $\gamma(x)$ und $\tau(x)$ ist, wie wir in [1, ab Seite 236] sehen, nur in manchen Fällen möglich.

Zwar sehen wir, dass sich durch Einfügen eines Zwischenschritts (in [1, Seite 233] als Zwischenschritt 10.1525 bezeichnet) eine große Anzahl an fehlerhaften $\sigma(x)$ und $\omega(x)$ bestimmen lässt und diese manchmal sogar korrigierbar sind, allerdings erscheint der Rechenaufwand für eine nur „mögliche“ Korrektur von $\sigma(x)$ und $\omega(x)$ zu groß, auch wenn man diesen Zwischenschritt eventuell vorzeitig abbrechen kann. Mit den notwendigen Vergleichen und zusätzlichen Berechnungen ist der Unterschied des Rechenaufwands zwischen diesem Zwischenschritt und der sofortigen Berechnung der Fehlerorte X_i zu minimal und die Korrektur von $\sigma(x)$ und $\omega(x)$ zu unwahrscheinlich, als dass es sich auszahlen würde. Die eindeutige Korrektur von Fehlern jenseits der Schranke von $t = \lfloor \frac{n-k}{2} \rfloor$ ist somit in günstigen Fällen möglich, kann jedoch nicht erwartet oder forciert werden.

7 Modifizierung von Codes

Die bis jetzt vorgestellten RS Codes können wir auf verschiedene Arten modifizieren, um sie besser an unsere Bedürfnisse anzupassen. Zum einen können wir mit Hilfe des systematischen Codierens das Ablesen eines Datenworts aus einem Codewort deutlich vereinfachen und zugleich die Fehlerfortpflanzung minimieren, zum anderen können Codes verkürzt werden. Dieses Verkürzen von Codes ist vor allem für das Verketteten von Codes sinnvoll, um keine zu großen Codewortblöcke zu generieren. Das Verketteten von Codes hilft uns wiederum, bei günstiger Verteilung der Fehler, welche mit hoher Wahrscheinlichkeit eintritt, bessere Korrektoreigenschaften zu erhalten.

7.1 Systematische Codierung

Betrachten wir erneut die Codierung mit Generatorpolynom. Zwei große Nachteile sind zum einen die Tatsache, dass das Bestimmen des Datenworts aus dem Codewort eine Polynomdivision erfordert. Zum anderen das Fortpflanzen von Fehlern durch die Polynomdivision, die bei der Bestimmung des Codeworts möglicherweise entstanden sind.

Eine Lösung für diese Probleme liefert das systematische Codieren, wodurch unser Datenwort Teil des Codeworts wird. Somit kann zum einen das Datenwort direkt aus dem Codewort abgelesen werden, zum anderen kann bei dieser Vorgangsweise sogar die Anzahl an begangenen Fehlern reduziert werden, sofern das ausgelesene Wort zu einem falschen Codewort decodiert wurde.

Definition 29. (siehe auch [4, Kapitel 1.2]) Eine Codierung heißt systematisch, genau dann, wenn für unser Datenwort $b_1 b_2 \dots b_k$ und das dazugehörige Codewort $\chi = \chi_1 \chi_2 \dots \chi_n$ gilt, dass entweder

$$\chi_1 = b_1, \chi_2 = b_2, \dots, \chi_k = b_k$$

oder

$$\chi_{n-k+1} = b_1, \chi_{n-k+2} = b_2, \dots, \chi_n = b_k \text{ ist.}$$

Betrachten wir wieder unser Generatorpolynom $g(x)$:

$$g(x) := \prod_{j=1}^{n-k} (x - \alpha^j)$$

Wir wollen den RS Code nun so gestalten, dass er systematisch ist. In unserem Fall suchen wir demnach eine Möglichkeit, dass unser Datenwort $f_b(x) = b_1 + b_2 x + \dots + b_k x^{k-1}$ Teil des Codeworts $\chi(x)$ wird. Betrachten wir dazu folgende Berechnung von Codewörtern $\chi(x)$:

$$\chi(x) = x^{n-k} f_b(x) - \left(x^{n-k} f_b(x) \pmod{g(x)} \right)$$

Wie wir bereits in Kapitel 4.2 in Korollar 7 gesehen haben, ist ein ausgelesenes Wort $\beta(x)$ genau dann ein Codewort, wenn $\beta(\alpha^i) = 0$ für alle $i = 1, 2, \dots, n - k$. Mit der soeben beschriebenen Berechnung der Codewörter $\chi(x)$ sehen wir, dass diese Bedingung erfüllt ist. Da wir uns in einem Euklidischen Ring befinden, können wir das Polynom $x^{n-k} f_b(x)$, wobei $\deg(f_b(x)) \leq k - 1$ ist, durch $g(x)$ teilen und es schreiben als: $x^{n-k} f_b(x) = g(x)f(x) + r(x)$ mit $\deg(f(x)) \leq k - 1$ und $\deg(r(x)) \leq n - k - 1$. Es gilt daher:

$$\begin{aligned} \chi(x) &= x^{n-k} f_b(x) - \left(x^{n-k} f_b(x) \pmod{g(x)} \right) \\ &= g(x)f(x) + r(x) - (g(x)f(x) + r(x) \pmod{g(x)}) \\ &= g(x)f(x) + r(x) - r(x) \\ &= g(x)f(x) \end{aligned}$$

Somit ist $\chi(\alpha^i) = g(\alpha^i) \cdot f(\alpha^i) = 0 \cdot f(\alpha^i) = 0$ für alle $i = 1, 2, \dots, n - k$ und damit $\chi(x)$ im von $g(x)$ erzeugten Code. Außerdem sehen wir, dass diese Codierungsmethode

systematisch ist, da die letzten k Einträge in $\chi(x)$ genau die Einträge des Datenworts $f_b(x)$ sind:

$$\chi(x) = r_1 + \dots + r_{n-k}x^{n-k-1} + b_1x^{n-k} + \dots + b_{k-1}x^{n-2} + b_kx^{n-1}$$

Wir müssen zur Decodierung zwar trotzdem noch die Berechnung der Syndrome s_i bzw. der Polynome $\sigma(x)$ und $\omega(x)$ (siehe Kapitel 5.2) durchführen, um herauszufinden, ob das ausgelesene Wort $\beta(x)$ ein Codewort ist, allerdings können wir nach der Decodierung (in dem Fall das $\beta(x)$ kein Codewort ist) sofort das Datenwort aus dem decodierten Wort ablesen, ohne durch $g(x)$ dividieren zu müssen. Außerdem erspart man sich beim BM Algorithmus das Berechnen der Fehlerwerte Y_k für alle Fehler X_k , die bei den ersten $n - k$ Stellen von $\beta(x)$ entstanden sind.

7.2 Verkürzen von Codes

Bis jetzt hatten wir, auf Grund der Wahl von $n = q - 1$, Codewörter der Länge $q - 1$ und eine fest vorgegebene Anzahl an korrigierbaren Fehlern, die nur durch Variation der Datenwortlänge k verändert werden konnte. Die Codewörter können durch dieses Vorgehen in manchen Fällen zu lang sein, weshalb wir z.B. eine zu große Redundanz und damit verbunden zu viel Speicherverlust haben. In diesem Kapitel betrachten wir daher Datenwortlängen k und Codewortlängen n mit $k < n = q - 1 - i$, wobei i eine natürliche Zahl mit $0 < i \leq q - k - 2$ ist. Durch diese Wahl werden die Codewörter und damit der ganze Code verkürzt. Doch was bedeutet es, einen Code zu verkürzen (siehe [4, Seite 38])?

Definition 30. Ein um i Stellen verkürzter $[n, k]$ -Code C_i mit $n = q - 1 - i$ bzw. $n + i = q - 1$ eines $[q - 1, k + i]$ -Codes C ist entweder

$$C_i = \{\chi_{i+1}\chi_{i+2} \dots \chi_{i+n} \mid \overbrace{0 \dots 0}^{i \text{ Stellen}} \chi_{i+1}\chi_{i+2} \dots \chi_{q-1} \in C\}$$

oder

$$C_i = \{\chi_1\chi_2 \dots \chi_n \mid \chi_1\chi_2 \dots \chi_n \overbrace{0 \dots 0}^{i \text{ Stellen}} \in C\}$$

Wir betrachten also nur Codewörter, deren erste oder letzte i Stellen Null sind. Diese haben, da sie auch Codewörter in C sind und dieser ein RS Code ist, wieder einen Minimalabstand von $d = q - 1 - k - i + 1^{23} = n - k + 1$. Doch bevor wir uns Gedanken über die Decodierung von verkürzten Codewörtern eines RS Codes machen, überlegen wir uns, wie wir einem beliebigen Datenwort der Länge k bestehend aus b_1, b_2, \dots, b_k Buchstaben aus $GF(q)$ ein Codewort der Länge $n > k$ zuordnen können. Dazu müssen die Codierungsverfahren aus den Kapiteln 4.1 und 4.2 wie folgt adaptiert werden:

Betrachten wir zuerst das Codierungsverfahren mit Generatorpolynom, so sehen wir, dass das Datenwort $(b_i)_{i=1}^k$ der Länge k in $GF(q)^{k+i}$ eingebettet werden kann, indem man es um $i = q - 1 - n$ Nullen verlängert:

$$b' = b_1b_2 \dots b_k \overbrace{0, \dots, 0}^{i \text{ Stellen}}$$

²³Wie bereits in Korollar 14 gezeigt.

Fassen wir dieses neue Datenwort als Polynom $f_b(x)$ auf und multiplizieren es nun mit dem Generatorpolynom

$$g(x) = \prod_{i=1}^{q-1-(k+i)} (x - \alpha^i)$$

des $[q-1, k+i]$ -Codes, so erhalten wir ein Codewort eines RS Codes mit Länge $q-1$, wobei wir jedoch sofort sehen, dass die letzten $i = q-1-n$ Stellen Null sind. Daher ist dieses Codewort in C_i enthalten. Das gleiche Codewort erhalten wir, wenn wir $f_b(x)$ gleich mit $g(x)$ multiplizieren:

$$f_b(x) \cdot g(x) = f_b(x) \cdot g(x)$$

Wir sehen also, dass bei vorgegebener Datenwortlänge k die Länge n des Codeworts im Bereich $k < n < q-1$ beliebig gewählt werden kann²⁴ und wir nach Multiplikation mit $g(x)$ verkürzte Codewörter eines RS Codes erhalten. Das Verfahren lässt sich auch mit Systematischem Codieren (Kapitel 7.1) verbinden, wie sich leicht nachrechnen lässt.

Ähnlich lässt sich auch im Fall der Auswertung bei α^i erklären, dass ein Datenwort der Länge k , das an n Potenzen von α ausgewertet wird, Teil eines verkürzten $[n, k]$ -RS Codes ist. Dazu betrachten wir erneut einen $[q-1, k+i]$ -RS Code und ordnen Datenwörtern der Länge k Datenwörter der Länge $k+i$ zu. Wir müssen jedoch sicherstellen, dass nach der Codierung die letzten $i = q-1-n$ Stellen der Codewörter Null sind, damit das Codewort in C_i enthalten ist. Gegeben seien wieder k, n und $q-1$ mit der Einschränkung $k < n < q-1$. Wir versuchen erneut $f_b(x)$ ein eindeutiges $f_b(x)$ zuzuordnen, dessen Codierung durch Auswertung in C_i enthalten ist. Dazu multiplizieren wir $f_b(x)$ mit

$$\prod_{j=n}^{q-2} (x - \alpha^j).$$

Dieses Produkt hat den Grad $q-1-n$, weshalb

$$f_b(x) = f_b(x) \cdot \prod_{j=n}^{q-2} (x - \alpha^j)$$

ein eindeutiges Polynom mit Grad höchstens $(q-1-n) + (k-1) = k+i-1$ ist und daher einem Datenwort der Länge $k+i$ entspricht. Wenn wir dieses an allen Potenzen von α aus, sehen wir, dass wir das Codewort

$$\chi = f_b(\alpha^0) f_b(\alpha^1) \dots f_b(\alpha^{n-1}) \overbrace{0 \dots 0}^{i \text{ Stellen}}$$

des $[q-1, k+i]$ -RS Codes erhalten, das wegen der letzten i Nullen ebenfalls in C_i enthalten ist. Nachdem auf Grund dieser Konstruktion das Auswerten der letzten i Stellen in dem Produkt $\prod_{j=n}^{q-2} (x - \alpha^j) \cdot f_b(x)$ überflüssig ist, reicht es die ersten n Stellen zu berechnen.

²⁴Durch die Kenntnis von k, n und $q-1$ ist es möglich sich $i = q-1-n$ und im Anschluss $k+i$ zu berechnen. So kann $g(x)$ bestimmt werden und durch Multiplikation von $f_b(x) \cdot g(x)$ können die Codewörter, die nun Länge $(k) + (q-1-(k+i)) = q-1-i = n$ haben berechnet werden.

Diese Berechnung erweist sich jedoch als aufwendig, da neben der Auswertung von $f_b(x)$ auch das Produkt $\prod_{j=n}^{q-2}(x - \alpha^j)$ mit seinen $q - 1 - n$ Subtraktionen und Multiplikationen jedes Mal neu zu berechnen ist. Eine Verbesserung des Rechenaufwands liefert uns eine zur Chien Suche analoge Überlegung:

$$\begin{aligned}
f_{b'}(\alpha^{i+1}) &= f_b(\alpha^{i+1}) \cdot \prod_{j=n}^{q-2} (\alpha^{i+1} - \alpha^j) \\
&= f_b(\alpha^{i+1}) \cdot \alpha^{q-1-n} \prod_{j=n}^{q-2} (\alpha^i - \alpha^{j-1}) \\
&= f_b(\alpha^{i+1}) \cdot \alpha^{q-1-n} \prod_{j=n-1}^{q-3} (\alpha^i - \alpha^j) \\
&= f_b(\alpha^{i+1}) \cdot \alpha^{q-1-n} \cdot \frac{\alpha^i - \alpha^{n-1}}{\alpha^i - \alpha^{q-2}} \cdot \prod_{j=n}^{q-2} (\alpha^i - \alpha^j)
\end{aligned}$$

Da es sich bei $\prod_{j=n}^{q-2}(\alpha^i - \alpha^j)$ um den zweiten Faktor das auszuwertende Datenpolynoms $f_{b'}(\alpha^i)$ handelt, reicht es für α^0 das Produkt $\prod_{j=n}^{q-2}(\alpha^0 - \alpha^j)$ zu berechnen und anschließend iterativ mit $\alpha^{q-1-n} \cdot \frac{\alpha^i - \alpha^{n-1}}{\alpha^i - \alpha^{q-2}}$ zu multiplizieren, um $\prod_{j=n}^{q-2}(\alpha^{i+1} - \alpha^j)$ zu erhalten. Die Anzahl an benötigten Rechenschritten um $\prod_{j=n}^{q-2}(x - \alpha^j)$ auszuwerten reduziert sich so auf 2 Subtraktionen und Multiplikationen sowie 1 Division.

Diese Methode erweist sich trotz dieser Vereinfachung in der Auswertung als rechenaufwändig, weshalb bei der Verkürzung von Codes die Variante mit Generatorpolynom zu bevorzugen ist.

Wir sehen also, dass wir bei einer gegebenen Datenwortlänge k , Codewörter der Länge $n < q - 1$ betrachten können. Dies ist insbesondere dann sinnvoll, wenn wir bei einer kurzen Datenwortlänge eine geringere Redundanz wünschen, oder bei der Verkettung von Codes (siehe Kapitel 7.3) die Datenblöcke nicht zu groß werden lassen wollen²⁵.

Bleibt noch zu klären, ob die Verkürzung von Codes eine Auswirkung auf die Decodierung und deren Eigenschaften hat. Betrachten wir zuerst die Decodierung, dann sehen wir, dass im Fall des GS Algorithmus das ausgelesene Wort $\beta(x)$ mit seinen n Koeffizienten ausreicht, um das ursprüngliche Datenwort $f_b(x)$ zu berechnen. Dazu müssen jedoch die β_i zuerst durch $\prod_{j=n}^{q-2}(\alpha^{i-1} - \alpha^j)$ dividiert werden. Für β_i die nicht verändert wurden bedeutet dies, dass

$$\bar{\beta}_i = \frac{\beta_i}{\prod_{j=n}^{q-2}(\alpha^{i-1} - \alpha^j)} = f_b(\alpha^{i-1})$$

ist. Unter der Voraussetzung das maximal $t_m = n - K_m$ Fehler gemacht wurden, ist das Datenwort in dem Output des GS Algorithmus enthalten.

Betrachten wir den Algorithmus im Detail, dann sehen wir, dass uns der Interpolationsalgorithmus ein Polynom $Q(x, y)$ liefert, das bei allen $(\alpha^{i-1}, \bar{\beta}_i)_{i=1}^n$ eine m -fache

²⁵Siehe auch das Beispiel zur Digitalisierung von Musik in [4, Seite 38].

Nullstelle hat, sowie minimal bezüglich $\deg_{(1,y)}$ ist. Wenn jetzt mindestens K_m Koeffizienten von $\tilde{\beta}(x)$ korrekt sind, dann heißt das auf Grund der Definition von K_m , dass

$$\sum_{\alpha \in GF(q)} \text{ord}(Q : \alpha, f(\alpha)) \geq mK_m > \deg_{(1,y)} Q(x, y)$$

ist. Daher muss der Faktorisierungsalgorithmus unter anderem das Datenwort $f_b(x)$ liefern.

Der zweite Decodierungsalgorithmus, der BM Algorithmus, berechnet mit Hilfe der Syndrome s_i das Fehlerlokalisierungspolynom sowie das Fehlerwertepolynom. Für beide ist die Verkürzung des Codes nicht relevant. Wenn wir wieder voraussetzen das maximal $\lfloor \frac{n-k}{2} \rfloor$ Fehler bei den $\beta(x)$ entstanden sind, dann liefert der Algorithmus wieder das ursprüngliche Codewort mit dessen Hilfe wir $f_b(x)$ berechnen können.

Was die Fehlerkorrektureigenschaften betrifft, so bleibt bei den soeben beschriebenen Verkürzungen von $[q-1, k+i]$ -RS Codes auf $[n, k]$ -Codes die Anzahl an Kontrollstellen und die Minimaldistanz gleich. Aus diesem Grund gibt es auch keine Änderung bei den Fehlerkorrektureigenschaften.

7.3 Verketteten von RS Codes

Eine Verbesserung der Decodierungsmöglichkeiten liefert uns ein Verketteten von RS Codes. Diese Methode wurde bereits im Bereich der Codierung auf der Compact Disc verwendet. Grundlegende Idee der Verkettung von Codes ist es Datenwörter zusammenzufassen indem man sie z.B. in Matrixform anschreibt und anschließend zeilen- und spaltenweise zu codieren. Durch diese Vorgehensweise können Bündelfehler die eine oder mehrere Zeilen oder Spalten betreffen dennoch korrigiert werden.

Wie man am Beispiel des Cross Interleave Reed-Solomon Code [13, ab Seite 76] (kurz CIRC) sieht, kann eine geschickte Kombination von RS Codes zu einer deutlichen Verbesserung von Korrektureigenschaften führen. In diesem Kapitel wird die Methode des CIRC kurz erläutert und aufgezeigt, welche Adaptionen im Falle der Nutzung dieser Methode für den Slack Space vorgenommen werden müssen. Unumgänglich wird eine Aufteilung eines Codeworts auf verschiedene Cluster des Speichermediums sein, um sicherzustellen, dass im Falle eines Datenverlusts (überschriebenen Clusters) trotzdem das ursprüngliche Daten, bzw. Codewort bestimmt werden kann.

Doch betrachten wir zuerst den für die CDs verwendeten CIRC. Dieser besteht sowohl aus Interleaving- als auch zwei Codierungsschritten. Während wir uns bereits mit dem RS Code beschäftigt haben, gehört das Interleaving noch erklärt:

Definition 31. Unter Interleaving [13, Seite 74] verstehen wir eine Verschachtelung von Codewörtern zu neuen Codewörtern, sodass sich Flächen- bzw. Bündelfehler bei den neuen Codewörtern nur als Einzelfehler auf verschiedene ursprüngliche Codewörter auswirken.

Der CIRC-Algorithmus ist im Detail in [13] nachzulesen. Kurz zusammengefasst kann man sagen, dass auf Grund von Fingerabdrücken, Staubkörnern, etc. Flächenfehler auftreten, die zu Verlust von Daten führen. Ein derartiger Verlust kann zum einen durch ein entsprechendes Codierungsverfahren korrigiert werden, zum anderen ist im Fall

von Audio-CDs eine Interpolation von Daten möglich. Diese zweite Variante ist bei uns natürlich nicht anwendbar, weshalb bei der Beschreibung des CIRC-Algorithmus auch auf Maßnahmen, die der Interpolation dienen, verzichtet wird.

Zu Beginn werden die Daten in Wörter bestehend aus 24 Byte-Symbolen unterteilt. Aus diesen Datenwörtern werden dann Codewörtern mit 28 Symbolen generiert. Wir haben also einen RS Code mit $n = 28$ und $k = 24$. Diese 28 Symbole werden anschließend zeilenweise in eine Datenmatrix bestehend aus 28 Zeilen und 28 Spalten geschrieben und spaltenweise erneut zu Codewörtern der Länge 32 codiert. Der zweite Code (ebenfalls ein RS Code) hat die Parameter $n = 32$ und $k = 28$ und ist - genauso wie der erste Code - eine Verkürzung des RS Codes über $GF(2^8)$ mit $n = 255$. Somit sind schlussendlich Codewörter bestehend aus 32 Symbolen bzw. eine Codewortmatrix aus $GF(q)^{28 \times 32}$ entstanden. Sollte eines dieser Codewörter nicht decodierbar sein, bedeutet das, dass 28 verschiedene Zeilen die nach dem ersten Codierungsschritt entstanden sind, jeweils ein nicht bestimmtes Symbol beinhalten. Der Verlust von 32 Symbolen und damit einem ganzen Codewort entpuppt sich so als Einzelfehler in 28 verschiedenen Codewörtern und kann hier unter der Voraussetzung, dass nicht zu viele weitere Fehler entstanden sind, problemlos korrigiert werden.

Während der Decodierungsalgorithmus bei der CD einen gewissen Zeithorizont nicht überschreiten darf, da ansonsten das Wiedergabegerät nichts abspielen könnte, haben wir mehr Zeit für die Decodierung. Dies ermöglicht uns natürlich aufwändigere Strategien zu wählen bzw. zu entwickeln, falls notwendig. Betrachten wir die „Superstrategie [13, Seite 84]“ von K. Pohlmann:

Dieser sieht vor den [32, 28]-RS Code zu verwenden, um ausschließlich Einzelfehler ohne weitere Überlegungen zu korrigieren obwohl er $\lfloor \frac{n-k}{2} \rfloor = 2$ Fehler korrigieren könnte, und Fehlerfreiheit festzustellen. Im Fall von 2 Fehlern werden diese zwar auch korrigiert, aber (genauso wie in allen übrigen Fällen, die kein 1-fach Fehler sind) werden alle 28 Symbole als fehlerhaft markiert.

Der zweite Code muss nun Fallunterscheidungen vornehmen und abhängig von Ort und Anzahl der errechneten Fehler sowie der markierten fehlerhaften Stellen entscheiden, wie zu decodieren ist. Sollte ein Fehlermuster vorhanden sein, dass nicht decodiert werden kann, muss interpoliert werden. Diese Option haben wir, wie bereits erwähnt, nicht. Wir müssen in diesem Fall entweder weitere Lösungsstrategien verfolgen, oder das Codewort ist für uns nicht decodierbar.

Eine für uns etwas praktischere Beschreibung des Produktcodes, der ja Teil des CIRC ist, findet man in [4, Seite 37]. Die Aufteilung auf die 28 verschiedenen Blöcke können wir also auch realisieren indem wir die Daten als Matrix $\in GF(2^8)^{24 \times 28}$ auffassen und nacheinander zuerst einen RS Code auf die 28 Spalten, die jeweils 24 Symbole beinhalten, und anschließend auf die 24+4 Zeilen mit jeweils 28 Symbolen anwenden. Die zusätzlichen 4 Zeilen ergeben sich aus dem ersten Codierungsschritt. Die Decodierung erfolgt dann in umgekehrter Reihenfolge und kann bei günstiger Verteilung der Fehler deutlich mehr Korrekturen vornehmen, als man erwarten würde. Daher ist das Interleaving ein wesentlicher Bestandteil des gesamten Codierungsalgorithmus.

8 Fehlerkorrektur im Slack Space

Wie wir im Grundlagenkapitel über Slack Space, Kapitel 2.3, bereits festgestellt haben, ist dieser eine Ansammlung von Bytes, bzw. Sektoren, aus verschiedenen Clustern auf einem Datenträger. Um längerfristig Informationen mit adäquater Redundanz auf diesem Datenträger speichern zu können, müssen wir wissen, mit welcher Stabilität des Slack Space wir rechnen können. Wie wir im Laufe dieser Arbeit gesehen haben, können wir bis zu $t_{GS} = n - 1 - \lfloor \sqrt{(k-1)n} \rfloor$ Fehler in einem einzelnen Codewort des RS Codes korrigieren. Die Verwendung von Produktcodes (ähnlich dem CIRC), Kapitel 7.3, ermöglicht uns im Allgemeinen noch eine Verbesserung der Korrektoreigenschaften. Allerdings nur bei guter Verteilung der Fehler und guter Lösungsstrategie.

Um möglichst fehlerfrei Daten wieder aus dem Slack Space entnehmen zu können, ist es wichtig, dass dieser nur in geringem Ausmaß geändert und überschrieben wird. Dies ist zum Beispiel bei Dateien des Betriebssystems der Fall. Diese werden im Prinzip nur beim Einspielen von Updates und Servicepacks geändert und liefern mehrere MB an Speicherplatz. Da wir hier bereits erste Zahlen betreffend der Sicherheit der Daten haben, beschränken wir uns auf Fehlerkorrektur von Daten, die im Slack Space des Betriebssystems gespeichert wurden. Auf Grund der durchschnittlichen Stabilität von 78% [12, Seite 191] erscheint die Verwendung eines Codierungsverfahrens mit einer Redundanz größer als 22% sinnvoll. Allerdings müssen wir beachten, dass, sollte die Redundanz sehr viel größer sein als von der Stabilität benötigt, wir zu viel Speicherplatz verschenken, während wir im umgekehrten Fall, bei zu geringer Redundanz, davon ausgehen müssen, dass Daten verloren gehen werden.

Gehen wir allgemein von einer Stabilität des Slack Space von $\mathcal{S}\%$ aus, dann können wir daraus für die Länge n eines Codeworts bzw. für das Verhältnis von k und n fordern, dass:

$$\frac{n-k}{n} \geq 1 - \mathcal{S} \Leftrightarrow \frac{k}{n} \leq \mathcal{S} \Leftrightarrow n \geq \frac{1}{\mathcal{S}}k$$

Mit dieser Abschätzung können wir uns nun überlegen, auf welche Art die Speicherung erfolgen soll. Wir betrachten 3 verschiedene Möglichkeiten mit unterschiedlichen Stärken und Schwächen. Hier eine kurze Auflistung der Varianten, die im Laufe dieses Kapitels noch genauer betrachtet werden:

- **Variante 1:** Die Daten werden in Datenwörter der Länge $k < n \leq q-1$ unterteilt, diese mittels $[n, k]$ -RS Code codiert und dann jedes Codewort byteweise, also jedes Symbol aus $GF(2^8)$, in einem eigenen Cluster des Slack Space gespeichert. Wird ein Cluster des Slack Space überschrieben geht dadurch nur 1 Buchstabe (Byte) eines Codeworts verloren. Der Vorteil dieser Variante ist, dass unter Verwendung des GS Algorithmus bis zu t_{GS} Fehler pro Codewort korrigiert werden können und im Unterschied zu den anderen Varianten nur Speicher belegt wird, der unbedingt notwendig ist. Nachteile sind natürlich nicht nur die Herausforderung für die Speicherverwaltung, sondern auch die Verteilung der Bytes auf die verschiedenen Cluster, da in ungünstigen Fällen mehr als t_{GS} Bytes verfälscht werden könnten. Eine detaillierte Analyse findet man in Kapitel 8.1.
- **Variante 2:** Eine zweite Variante wäre, das Wissen über Produktcodes aus Kapitel 7.3 zu nutzen, um die Daten in eine oder mehrere Datenmatrizen anstatt ein

oder mehrere Datenwörter zu unterteilen. Die Daten wären dann in einer oder mehreren Matrizen der Form $k_1 \cdot k_2$ zu bringen und diese zeilen- und spaltenweise mit RS Codes zu codieren. Diese Matrix speichern wir spaltenweise in Clustern.

Nachdem mit dieser Variante die Daten in größere Blöcke unterteilt werden, ist nicht nur der Aufwand für die Speicherverwaltung geringer, sondern auch die benötigte Redundanzen für die $[n_1, k_1]$ - und $[n_2, k_2]$ -RS Codes ist auf Grund der Fehlerkorrektureigenschaften von Produktcodes geringer. Ein Nachteil ist natürlich, dass Daten im Normalfall kein Vielfaches von $k_1 \cdot k_2$ Matrizen sind, weshalb wir die „letzte“ Datenmatrix mit irgendwelchen Symbolen auffüllen müssen, um eine komplette Datenmatrix zu erhalten. Eine detaillierte Analyse findet man in Kapitel 8.2.

- **Variante 3:** Eine weitere (zu Variante 2 analoge) Möglichkeit ist das Ausdehnen der Codewortmatrix aus Variante 2 auf den gesamten Slack Space, sofern dies das Alphabet zulässt. In unserem Fall, bei $GF(2^8)$, würde die größtmögliche Codewortmatrix $(q - 1)^2 = 65.025$ Bytes umfassen, eine eher kleine Datenmenge. Eine Erweiterung des Alphabets auf $GF(2^{16})$ indem man 2 Symbole zu einem zusammen fasst, wäre eine Möglichkeit die Datenmatrix auf bis zu ca. 4 GiB zu vergrößern. D.h. die Größe des Alphabets ist abhängig von der Größe des vorhandenen Slack Space.

Von Vorteil ist natürlich, dass sich die entstandenen Fehler deutlich besser abschätzen lassen. Der große Nachteil ist, dass durch diese Methode gerade bei geringen Datenmengen die im Slack Space gespeichert werden sollen, unnötig viel Codierungsarbeit vorgenommen werden muss. Außerdem muss für diese Variante zuerst die Größe des vorhandenen Slack Space berechnet werden. Details zu dieser Variante findet man in Kapitel 8.3.

8.1 Variante 1

Betrachten wir nun die erste Variante etwas genauer. Wir unterteilen die zu speichernden Daten in Datenwörter der Länge k . Diese werden dann durch systematisches Codieren mit Generatorpolynom (siehe Kapitel 7.1) auf $n \leq q - 1$ Stellen verlängert, sodass die Redundanz groß genug ist, um bei einer Stabilität von \mathcal{S} das Datenwort mit großer Sicherheit²⁶ wiederherstellen zu können.

Beispiel 9. Codierung eines Passworts:

Nehmen wir an, wir wollen ein 30stelliges Passwort im Slack Space speichern. Dann hat dieses, abhängig von der gewählten Zeichencodierung wie z.B. UTF-8, eine Länge von vermutlich etwas mehr als 30 Byte, da für die meisten Standardzeichen nur 1 Byte benötigt wird. Gehen wir davon aus, dass wir 35 Byte codieren wollen. Nachdem wir über dem Galoisfeld $GF(2^8)$ arbeiten, können wir das so entstandene Datenwort (eine Unterteilung des Datenworts auf 2 kürzere Wörter erscheint unnötig) auf eine Zeichenkette bestehend aus $n = 255$ Byte verlängern, was aber eine Redundanz von $\frac{255-35}{255} = 0,86 \dots \approx 86,3\%$ zur Folge hätte. Es ist daher sinnvoll, das Codewort auf eine Länge $n < 255$ (siehe dazu Kapitel 7.2) zu verkürzen, wobei wir jedoch berücksichtigen müssen, dass wir mit möglichst großer Sicherheit richtig decodieren wollen.

²⁶In diesem Kapitel berechnen wir die Wahrscheinlichkeiten, mit denen eine Wiederherstellung des Datenworts möglich ist.

Wie wir an diesem Beispiel sehen, ergeben sich für uns mehrere Fragen:

- Mit wie vielen Fehlern muss bei einem ausgelesenen Wort gerechnet werden?
- Ab welcher Länge des Datenworts muss dieses in zwei Datenwörter unterteilt werden?
- Wie groß muss n in Abhängigkeit von k gewählt werden, um bei einer gegebenen Stabilität S noch mit großer Sicherheit korrekt decodieren zu können?

Wir sehen, dass Frage 2 eigentlich durch Frage 3 mit beantwortet wird, da ein Codewort ja maximal Länge $q - 1$ haben darf.

Doch erklären wir zuerst noch wie die Decodierung zu erfolgen hat:

Wie wir in Kapitel 6.1 gesehen haben, können wir bis zu t_{GS} Fehler korrigieren, wenn wir den GS Algorithmus zur Decodierung verwenden, der jedoch auf der Codierung durch Auswertung beruht. Nachdem das Datenwort mit Hilfe des Generatorpolynoms systematisch codiert wurde, benötigen wir noch folgende Überlegung:

In Kapitel 7.2 wurde gezeigt, dass sowohl mit Auswertung, als auch mit Generatorpolynom Codewörter eines verkürzten RS Codes C_i erzeugt werden können. D.h. jedes mit Generatorpolynom erzeugte Codewort entspricht einem durch Auswertung erzeugten Codewort, wobei die codierten Datenwörter verschieden sind, da für die Auswertung $f_{b'}(x) = f_b(x) \cdot \prod_{j=n}^{q-2} (x - \alpha^j)$ berechnet werden muss.

Da wir, um den GS Algorithmus für die Decodierung zu verwenden, davon ausgehen, dass das ausgelesene Wort durch Auswertung bei α^i mit $i = 0, 1, \dots, n - 1$ entstanden ist, dividieren wir zuerst alle β_i durch $\prod_{j=n}^{q-2} (\alpha^{i-1} - \alpha^j)$ für $i = 1, 2, \dots, n$. Dadurch erhalten wir für alle unverfälschten Buchstaben β_i korrekt:

$$\bar{\beta}_i := \frac{\beta_i}{\prod_{j=n}^{q-2} (\alpha^{i-1} - \alpha^j)} = \frac{f_{b'}(\alpha^{i-1})}{\prod_{j=n}^{q-2} (\alpha^{i-1} - \alpha^j)} = \frac{f_b(\alpha^{i-1}) \prod_{j=n}^{q-2} (\alpha^{i-1} - \alpha^j)}{\prod_{j=n}^{q-2} (\alpha^{i-1} - \alpha^j)} = f_b(\alpha^{i-1})$$

Mit Hilfe des GS Algorithmus können wir aus den so berechneten $\bar{\beta}_1, \bar{\beta}_2, \dots, \bar{\beta}_n$ nun bis zu L_m Datenwörter der Länge k berechnen, die wir erneut bei den Potenzen von α^{i-1} auswerten und anschließend wieder mit $\prod_{j=n}^{q-2} (\alpha^{i-1} - \alpha^j)$ multiplizieren, um Codewörter zu erhalten die mit Generatorpolynom erzeugt wurden. Bei den so berechneten Codewörtern ist bei demjenigen bzw. denjenigen mit den meisten Übereinstimmungen mit dem ausgelesenen Wort die Wahrscheinlichkeit am größten, dass es sich um das ursprünglich gespeicherte Codewort handelt. Um das Datenwort $f_b(x)$ zu erhalten, müssen dann die ersten oder letzten k Buchstaben des Codeworts abgelesen werden (ursprünglich wurde ja mit Generatorpolynom systematisch codiert).

Wenn mehrere gleich wahrscheinliche Codewörter, und daher auch Datenwörter, berechnet worden sind, sind Überlegungen gefragt wie mit diesen umgegangen werden soll. Diese Überlegungen sind von den Anforderungen an die ursprünglich gespeicherten Daten, war es z.B. ein Passwort, ein Programm, oder eine Bilddatei, abhängig, da in manchen Fällen das ursprüngliche Datenwort benötigt wird, während in anderen Fällen ein annähernd richtiges Datenwort ausreichend ist. Eine Möglichkeit wäre es daher alle potentiellen Datenwörter auszugeben und z.B. einen Menschen entscheiden zu lassen, welches von ihnen am ehesten passt. Man könnte auch alle möglichen Datenwörter

$b_1 b_2 \dots b_k$ betrachten und für jeden einzelnen Buchstaben b_i eine Häufigkeitsanalyse durchführen. Jener Wert den b_i , für ein festes i , bei den meisten Datenwörtern annimmt wird ausgegeben. So erhält man ein Wort, das auf Grund der systematischen Codierung, an relativ vielen, womöglich allen, Stellen mit dem ursprünglich codierten Datenwort übereinstimmt.

Definitiv korrekt decodieren können wir, wenn maximal $t = \lfloor \frac{n-k}{2} \rfloor$ Fehler auftreten, da in diesem Fall das Codewort, das uns der Decodierungsalgorithmus liefert, die meisten Übereinstimmungen mit dem ausgelesenen Wort hat. Sollten mehr als t , jedoch maximal $t_{GS} = n - 1 - \lfloor \sqrt{(k-1)n} \rfloor$ Fehler auftreten, dann befindet sich das richtige Codewort bzw. Datenwort zwar in unserer Lösungsmenge, allerdings kann es sein, dass andere Codewörter mehr Ähnlichkeiten mit dem ausgelesenen Wort aufweisen und wir daher ein falsches Codewort und damit ein falsches Datenwort wählen. Wir sollten außerdem beachten, dass wie in Kapitel 6.1 bereits gezeigt, sowohl die Parameter m , als auch L_m bei ungünstiger Wahl von k zu einer sehr langen Laufzeit führen können.

Versuchen wir nun Antworten auf unsere zuvor gestellten Fragen zu finden: Die erste Frage galt der Anzahl an Fehlern mit denen wir rechnen müssen. Dazu sei erwähnt, dass es bei dieser Variante unmöglich ist, Auslöschungen zu erkennen im Unterschied zu den Varianten 2 und 3 wie wir später sehen werden.

Da ein Fehler gleichbedeutend mit der Überschreibung eines Clusters ist und wir die Modifizierung von Dateien z.B. bei Updates als unabhängig von einander annehmen, ist die Wahrscheinlichkeit, dass ein Cluster nicht verändert wurde und damit das Byte unseres Codeworts erhalten blieb gleich der Stabilität S des gesamten Slack Space. Gehen wir davon aus, dass ein Cluster entweder überschrieben wurde, oder unangetastet geblieben ist und sich ein Codewort über n Cluster erstreckt, dann sehen wir, dass die Wahrscheinlichkeit, dass ein ausgelesenes Wort ein Codewort ist binomial verteilt ist.

Wenn die Zufallsvariable X die Anzahl an Fehlern eines Codeworts beschreibt, dann kann die binomial verteilte Wahrscheinlichkeit $P(X \leq t_{GS})$, mit der Wahrscheinlichkeit $p = 1 - S$, durch eine Normalverteilung angenähert werden, falls die Standardabweichung

$$\hat{\sigma} = \sqrt{(1-S)Sn} > 3$$

ist, was die Berechnung deutlich erleichtert. Sollte diese Ungleichung nicht erfüllt sein, ist die Approximation der Binomialverteilung durch die Normalverteilung nicht gut genug. Dennoch werden wir auf Grund der Vereinfachung bei den Berechnungen der Wahrscheinlichkeiten $P(X \leq t_{GS})$ für alle auftretenden Stabilitäten S und Codewortlängen n die Annäherung verwenden. Im Falle einer Normalverteilung können wir erwarten, dass $\mathbb{E} = \lceil (1-S)n \rceil$ Bytes eines Codeworts fehlerhaft sein werden. Ausschließlich $\mathbb{E} = \lceil (1-S)n \rceil$ Fehler zu erwarten ist jedoch zu wenig, da so nur in höchstens der Hälfte der Fälle das ursprüngliche Datenwort wieder hergestellt werden kann. Wir betrachten daher außerdem die Wahrscheinlichkeiten für zusätzliche $\hat{\sigma}$ bzw. $2\hat{\sigma}$ Fehler:

$$\begin{aligned} P(X \leq \mathbb{E}) &= 0,5 \\ P(X \leq \mathbb{E} + \hat{\sigma}) &= 0,8413 \\ P(X \leq \mathbb{E} + 2\hat{\sigma}) &= 0,9772 \end{aligned} \tag{4}$$

Natürlich können auch jegliche andere Wahrscheinlichkeiten bzw. Intervalle betrachtet werden. Wir beschränken uns auf diese drei Fälle, da sie ausreichend erscheinen. Die Betrachtung der Intervalle $[0, \mathbb{E}]$, $[0, \mathbb{E} + \hat{\sigma}]$ und $[0, \mathbb{E} + 2\hat{\sigma}]$ sowie der dazugehörigen Wahrscheinlichkeiten ist wichtig, da wir nicht wissen, ob wir stabile Cluster beschrieben haben und wir daher davon ausgehen müssen, dass durchaus mehr als $\lceil (1 - S)n \rceil$ Fehler auftreten, wodurch nicht mehr korrekt decodiert werden könnte, sollten wir die Codewortlänge n nur so gewählt haben, dass $\mathbb{E} = \lceil (1 - S)n \rceil$ viele Fehler korrigiert werden können.

Wie man in [12] nachlesen kann, verlieren wir durch das Einspielen von Servicepacks deutlich mehr Stabilität im Slack Space, als nur beim Durchführen von Updates. Da die Wahrscheinlichkeiten jedoch für verschiedenen Betriebssysteme sehr ähnlich sind, betrachten wir exemplarisch das Betriebssystem *Windows 7 Professional* und dessen *SPI*, für das wir eine Stabilität von $S = 73,7\%$ (Einspielen von einem Servicepack sowie 156 Updates), bzw. $S = 93,4\%$ (Einspielen von 106 Updates) haben.

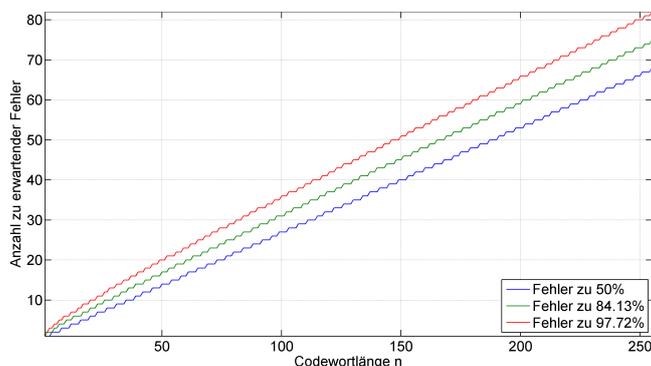


Abbildung 6: Anzahl an Fehlern bei Stabilität von $S = 73,7\%$ mit Wahrscheinlichkeiten von 50%, 84,13% und 97,72%

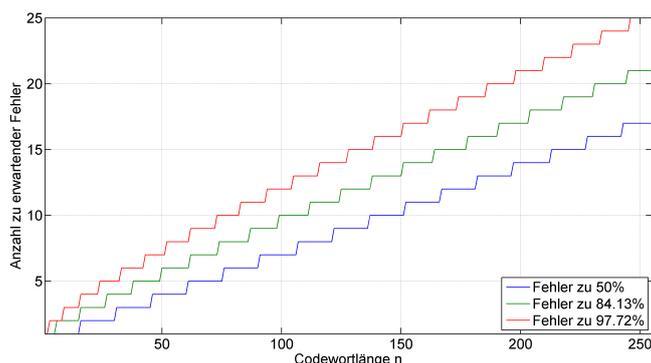


Abbildung 7: Anzahl an Fehlern bei Stabilität von $S = 93,4\%$ mit Wahrscheinlichkeiten von 50%, 84,13% und 97,72%

Die Anzahl an Fehlern \mathbb{E} , bzw. $\mathbb{E} + \hat{\sigma}$ und $\mathbb{E} + 2\hat{\sigma}$ lässt sich nun sofort in Abhängigkeit von n berechnen. Für jede der beiden Stabilitäten betrachten wir in den Abbildungen 6

und 7 die von n abhängige Anzahl an Fehlern mit denen wir mit den Wahrscheinlichkeiten von 50%, 84,13% und 97,72% rechnen müssen.

Um die zweite Frage zu beantworten, wenden wir uns gleich der dritten Frage zu, denn wenn wir wissen, wie die Länge des Datenworts und die des Codeworts zusammenhängen, können wir sofort den Rückschluss ziehen, ab wann wir das Datenwort teilen müssen. Schließlich gilt für das Codewort eine maximale Länge von 255 Buchstaben.

Wie groß ist nun n zu wählen, wenn wir die Datenwortlänge k haben? Dazu überlegen wir uns, dass wir abhängig von unseren Decodierungseigenschaften erwarten, dass

$$\mathbb{E} = \lceil (1 - S)n \rceil \leq \left\lfloor \frac{n - k}{2} \right\rfloor$$

bzw.

$$\mathbb{E} = \lceil (1 - S)n \rceil \leq n - 1 - \left\lfloor \sqrt{(k - 1)n} \right\rfloor$$

gelten soll. Im ersten Fall erwarten wir uns also, dass wir weniger Fehler haben als zur sicheren Decodierung notwendig, im zweiten Fall dürfen maximal t_{GS} Fehler auftreten, um korrekt zu decodieren. Da die Wahrscheinlichkeit das bis zu $\mathbb{E} = \lceil (1 - S)n \rceil$ Fehler auftreten jedoch nur 50% ist (siehe 4 auf Seite 82), betrachten wir ebenfalls folgende Ungleichungen:

Wenn

$$\begin{aligned} \lceil (1 - S)n + \widehat{\sigma} \rceil &\leq \left\lfloor \frac{n - k}{2} \right\rfloor \text{ und} \\ \lceil (1 - S)n + \widehat{\sigma} \rceil &\leq n - 1 - \left\lfloor \sqrt{(k - 1)n} \right\rfloor \end{aligned}$$

ist, ist mit mindestens 84,13%iger Wahrscheinlichkeit, bzw. wenn

$$\begin{aligned} \lceil (1 - S)n + 2 \cdot \widehat{\sigma} \rceil &\leq \left\lfloor \frac{n - k}{2} \right\rfloor \text{ und} \\ \lceil (1 - S)n + 2 \cdot \widehat{\sigma} \rceil &\leq n - 1 - \left\lfloor \sqrt{(k - 1)n} \right\rfloor \end{aligned}$$

ist, ist mit mindestens 97,72%iger Wahrscheinlichkeit das ursprüngliche Datenwort im Output unseres Decodieralgorithmus, da bis zu $\lceil \mathbb{E} + \widehat{\sigma} \rceil$ bzw. $\lceil \mathbb{E} + 2\widehat{\sigma} \rceil$ Fehler mit den Wahrscheinlichkeiten 84,13%, bzw. 97,72% auftreten.

In den Abbildungen 8 und 9 sehen wir, dass sich diese Variante auf Grund der maximalen Codewortlänge von 255 Buchstaben eher nur bei einer hohen Stabilität und bei Daten die nur aus wenigen Hundert Byte bestehen bezahlt macht. Die Redundanz ist ansonsten doch sehr hoch bzw. der Aufwand zum Speichern der Codewörter wird sehr groß, da jeder Buchstabe in einem eigenen Cluster gespeichert werden sollte.

Wie wir in Abbildung 8 erkennen, brauchen wir eine Redundanz von mindestens 50% (also $n \geq 2 \cdot k$), um halbwegs sicher decodieren zu können, wobei es auch stark auf den Korrekturwunsch (also bis zu t oder t_{GS} Fehler) ankommt. Außerdem sind nur Datenwörter mit einer Länge von maximal 90 bis 140 Buchstaben codierbar. Abbildung 9

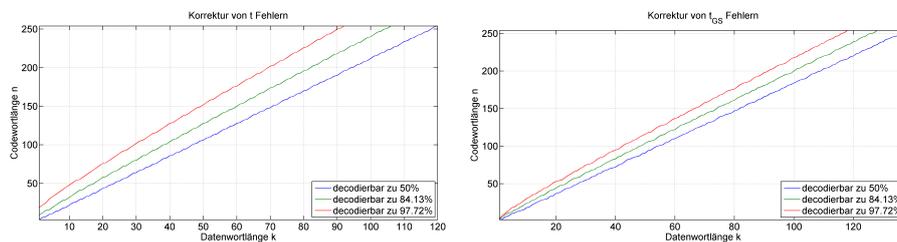


Abbildung 8: Benötigte Codewortlänge bei Stabilität von 73,7%, um zu gegebener Wahrscheinlichkeit t bzw. t_{GS} Fehler zu korrigieren

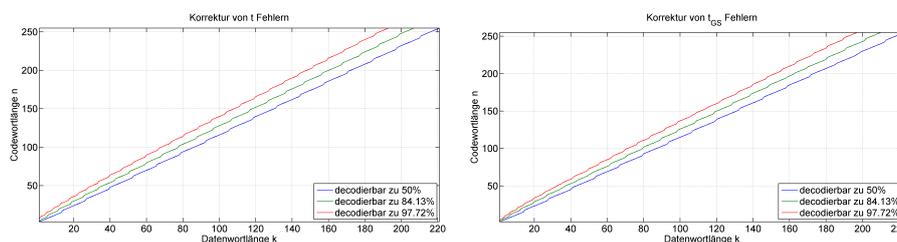


Abbildung 9: Benötigte Codewortlänge bei Stabilität von 93,4%, um zu gegebener Wahrscheinlichkeit t bzw. t_{GS} Fehler zu korrigieren

lässt uns erkennen, dass sich bei relativ hoher Stabilität nicht nur die Unterschiede der benötigten Codewortlängen stark reduzieren, sondern auch noch sehr lange Datenwörter mit großer Wahrscheinlichkeit richtig decodiert werden können, ohne diese unterteilen zu müssen oder eine besonders große Redundanz zu erfordern.

Eine Überlegung zum Schluss wäre noch, ob wir nicht gleich mehrere Bytes pro Cluster speichern sollten. Nehmen wir an, dass wir l Bytes pro Cluster speichern wollen, dann sehen wir, dass wir in diesem Fall nur mehr mit

$$l \cdot \left\lceil S \cdot \frac{n}{l} \right\rceil \leq \lfloor S \cdot n \rfloor$$

korrekten Bytes rechnen können. Da wir auf diese Weise aber nur noch mehr Buchstaben pro Codewort verlieren würden, anstatt irgendetwas dazu zu gewinnen ist diese Methode generell abzulehnen, außer es herrscht Gleichheit in obiger Ungleichung.

8.2 Variante 2

Bei dieser Variante lassen wir unser Wissen über das Verketteten von Codes, Kapitel 7.3 (bzw. Produktcodes [4, Seite 37]) und die Fehlerkorrektur bei RS Codes mit Auslöschungen, Kapitel 5.1.1, einfließen. Wie zuvor werden wir mit Überlegungen zur Codierung bzw. Decodierung beginnen und uns der Frage stellen, wie groß die Datenwort- bzw. Codewortlänge für beide Codes sein sollte.

Betrachten wir zuerst die Codierung. Wir haben Buchstaben a_{ij} , die wir in einen $k_2 \times k_1$ Datenblock unterteilen. Dieser wird zuerst zeilenweise auf n_1 und anschließend spaltenweise auf n_2 Stellen verlängert. Beide Codierungen erfolgen auf die gleiche Art wie in Variante 1 durch systematische Codierung mit Generatorpolynom. Diese

Codematrix speichern wir nun spaltenweise in verschiedenen Clustern:

$$\begin{array}{ccccccc}
 a_{11} & a_{12} & \dots & a_{1k_1} & \psi_{1(k_1+1)} & \dots & \psi_{1n_1} \\
 a_{21} & a_{22} & \dots & a_{2k_1} & \psi_{2(k_1+1)} & \dots & \psi_{2n_1} \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 a_{k_2 1} & a_{k_2 2} & \dots & a_{k_2 k_1} & \psi_{k_2(k_1+1)} & \dots & \psi_{k_2 n_1} \\
 \chi_{(k_2+1)1} & \chi_{(k_2+1)2} & \dots & \chi_{(k_2+1)k_1} & \chi_{(k_2+1)(k_1+1)} & \dots & \chi_{(k_2+1)n_1} \\
 \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 \chi_{n_2 1} & \chi_{n_2 2} & \dots & \chi_{n_2 k_1} & \chi_{n_2(k_1+1)} & \dots & \chi_{n_2 n_1} \\
 \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 \text{Cluster}_1 & \text{Cluster}_2 & \dots & \text{Cluster}_{k_1} & \text{Cluster}_{k_1+1} & \dots & \text{Cluster}_{n_1}
 \end{array}$$

Durch dieses Codierungsverfahren ergeben sich verschiedene Ansprüche an die beiden Codierungsschritte:

Durch Überschreiben eines Clusters geht eine ganze Spalte verloren. Daher muss der $[n_2, k_2]$ -Code kaum Fehler korrigieren, sondern sicherstellen, dass ein überschriebenes Codewort nicht irrtümlich für ein echtes gehalten wird. Auf diese Weise können Auslöschungen in den Codewörtern des $[n_1, k_1]$ -Codes erkannt werden. Wie viele Spalten verloren gehen ist, analog zur ersten Variante, abhängig von der Stabilität \mathcal{S} . Wenn der $[n_2, k_2]$ -Code gut genug ist, kann man vermuten, dass die meisten überschriebenen Spalten erkannt werden, wodurch der $[n_1, k_1]$ -Code eine geringere Redundanz benötigt, da die meisten „Fehler“ der ersten Variante (siehe Kapitel 8.1) als Auslöschungen erkannt werden und für Auslöschungen gilt, dass diese nur $e \leq n - k$ erfüllen müssen im Vergleich zu den Fehlern an unbekanntenen Stellen für die $e \leq \lfloor \frac{n-k}{2} \rfloor$ bei BM Decodierung oder $e \leq n - 1 - \lfloor \sqrt{n(k-1)} \rfloor$ bei GS Decodierung gilt. n_1 muss demnach nicht so groß gewählt werden wie für Variante 1.

Die Decodierung erfolgt in umgekehrter Reihenfolge zur Codierung. Spalten- und zeilenweise werden die ausgelesenen Wörter genauso decodiert wie in Variante 1 (Kapitel 8.1). Also durch Bijektion, GS Algorithmus, Auswertung, erneute Bijektion, Wahl des wahrscheinlichsten Codeworts und Abschneiden wird bestimmt, um welches Datenwort es sich gehandelt haben muss. Zuerst werden die Datenwörter des $[n_2, k_2]$ -Codes, also die Spalten wiederhergestellt. Wobei wir entweder 0 oder maximal einen Fehler korrigieren, da wir davon ausgehen, dass das Schreiben und Auslesen der Daten ohne Informationsverlust erfolgt und somit nur ein Datenverlust durch Überschreiben stattfindet. In diesem Fall ist jedoch die gesamte Spalte korrumpiert, womit die Korrektureigenschaften vom $[n_2, k_2]$ -Code nutzlos sind. Nach diesem ersten Schritt gibt es ϵ Spalten, die als ausgelöscht markiert sind. Nun werden zeilenweise die Datenwörter wiederhergestellt, wobei wir den Decodierungsalgorithmus inklusive Auslöschungen (Kapitel 5.1.1) verwenden. Auf Grund der Wahl des zweiten Codes muss der $[n_1, k_1]$ -Code nur sehr wenige echte Fehler korrigieren, da diese nur dann auftreten, falls der zweite Code einen überschriebenen Cluster als korrekt wahrnimmt. Somit muss n_1 nur etwas größer als $k_1 + \epsilon$ sein, wenn ϵ die Anzahl der Auslöschungen ist.

Bemerkung 15. In Kapitel 7.3 wurde das Interleaving vorgestellt. Also die Verschachtelung von Codewörtern, damit sich Bündelfehler (überschriebene Cluster) auf mehrere verschiedene Codewörter auswirken. In Variante 2 wird diese Methode dahingehend verwendet, dass wir die Spalten der Codematrix in verschiedenen Clustern speichern

und somit der Ausfall eines ganzen Clusters den Verlust von nur einem Buchstaben pro Codewort aus dem zeilenweise verwendeten $[n_1, k_1]$ -Code entspricht. Eine weitere Verschachtlung der Codematrizen, z.B. nehmen wir die erste Zeile der ersten Matrix, die zweite Zeile der zweiten Matrix, usw., ist natürlich auch möglich, damit sich überschriebene Cluster nur als Einzelfehler auf die verschiedenen Codematrizen auswirken. Diese Vorgehensweise erscheint aber zu kompliziert, bzw. umständlich, da nicht klar ist, welche Menge an Daten gespeichert werden müssen (und ob sich daher ein derartiger Aufwand auszahlt). Ein Interleaving der Codematrizen wird daher nicht weiter untersucht.

Es ergeben sich erneut drei Fragen:

- Wie groß muss n_2 sein, damit wir mit möglichst großer Wahrscheinlichkeit alle überschriebenen Cluster erkennen?
- Wie groß muss n_1 sein, damit wir bei gegebener Stabilität die gesamten Daten wiederherstellen können?
- Wie groß ist die Redundanz?

Frage 3 zeigt uns bereits den Zusammenhang zwischen den ersten beiden Fragen und der Wahl von n_1 und n_2 . Mit 100%iger Wahrscheinlichkeit lässt sich nicht immer decodieren. Allerdings: Je größer n_2 wird, desto sicherer können wir sein, dass wir keinen Fehler sondern nur Auslöschungen beim zeilenweisen Decodieren berücksichtigen müssen. Wenn wir dann n_1 so wählen, dass wir dennoch eine minimale Anzahl an Fehlern korrigieren können, erhöht sich die Redundanz dermaßen, dass viel Speicherplatz verloren geht.

Beantworten wir zunächst einmal Frage 1 so gut es geht: Wie wir in Kapitel 6.1 gesehen haben, können wir uns die durchschnittliche Anzahl an Codewörtern in einer r -Kugelumgebung berechnen durch:

$$\bar{L}(r) = q^{k_2 - n_2} \sum_{i=0}^r \binom{n_2}{i} (q-1)^i$$

Wenn wir nun davon ausgehen, dass $r \leq \lfloor \frac{n_2 - k_2}{2} \rfloor$ ist, dann entspricht $\bar{L}(r)$ der Wahrscheinlichkeit p_r , dass sich ein zufälliges Wort der Länge n_2 in einer r -Kugelumgebung eines Codeworts befindet. Gehen wir davon aus, dass wir nur 0, oder einen Fehler korrigieren wollen (mehr macht wie gesagt nicht viel Sinn), dann muss $r = 0$ (und daher $n_2 \geq k_2$) oder $r = 1$ (und daher $n_2 \geq k_2 + 2$) sein. Damit sind die Wahrscheinlichkeiten p_0 und p_1 mit denen ein beliebiges n_2 Symbole langes Wort ein Codewort des $[n_2, k_2]$ -Codes ist:

$$p_0 = q^{k_2 - n_2} = \frac{1}{q^{n_2 - k_2}}$$

$$p_1 = q^{k_2 - n_2} (1 + n_2 (q - 1))$$

Nachdem wir in $GF(2^8)$ rechnen ist $q = 256$, wodurch wir sofort sehen (siehe Abbildung 10), dass für den Fall der Korrektur von 0 Fehlern bereits die Wahl von $n_2 = k_2 + 2$ zu $p_0 = 0,000015 \dots$ führt. Diese Wahrscheinlichkeit ist so gering, dass es sehr unwahrscheinlich ist, dass ein überschriebener Cluster wieder ein Codewort des

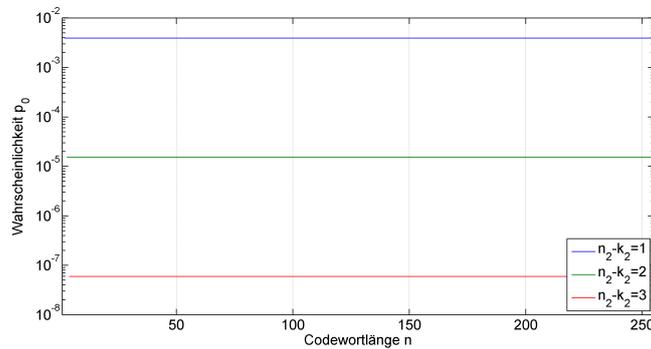


Abbildung 10: Wahrscheinlichkeit, dass $[n_2, k_2]$ -Code für $r = 0$ und konstante Differenz $n_2 - k_2$ versagt.

$[n_2, k_2]$ -Codes liefert. Selbst wenn wir für den $[n_1, k_1]$ -Code das größtmögliche $n_1 = 255$ wählen, erwarten wir nur $n_1 \cdot p_0 = 0,0038 \dots$ Fehler pro $[n_1, k_1]$ -Codewort. Demnach können wir davon ausgehen, dass es reicht, wenn der erste Code alle zu erwartenden Auslöschungen (diese sind ja noch vorhanden und von der Stabilität abhängig) zuzüglich eines Fehlers korrigiert.

Sollten wir zusätzlich innerhalb eines Clusters bei einem $[n_2, k_2]$ -Codewort die Korrektur eines Fehlers vornehmen wollen, vergrößert sich die Anzahl der Wörter, die irrtümlich für Codewörter gehalten werden können abhängig von n_2 linear wie wir der Berechnung von p_1 entnehmen, wenn wir wie zuvor $n_2 - k_2$ konstant lassen. Wir sehen, dass bei Wahl von $n_2 = k_2 + 2$ (siehe Abbildung 11) für sehr große n_2 , wie zB $n_2 = 255$ die Wahrscheinlichkeit $p_1 = 0,9922 \dots$ ist, womit eigentlich fast jeder überschriebene Cluster ein neues Codewort liefert. Durch Vergrößern der Redundanz auf $n_2 = k_2 + 4$ reduziert sich p_1 wieder auf einen Wert mit ähnlicher Größenordnung wie der von p_0 , nämlich $p_1 = 0,000015 \dots$ für maximales n_2 bzw. sogar nur auf $p_1 = 0,00000029 \dots$ für $n_2 = 5$.

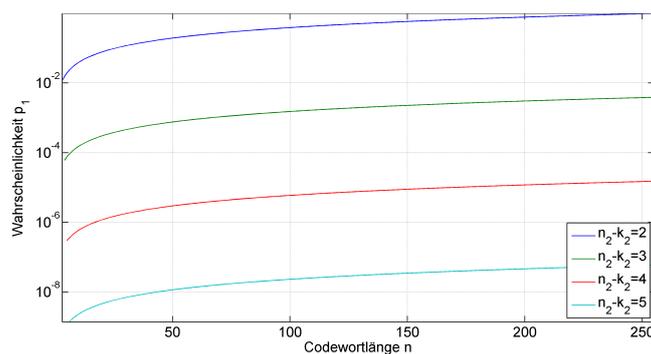


Abbildung 11: Wahrscheinlichkeit, dass $[n_2, k_2]$ -Code für $r = 1$ und konstante Differenz $n_2 - k_2$ versagt.

Wir sehen also, dass wir bei Korrektur von 0 Fehlern unabhängig von der Länge des Datenworts dieses nur um 2 Stellen verlängern müssen, um relativ sicher zu erkennen,

ob das Codewort überschrieben wurde. Sollten wir mit einem Fehler beim Auslesen rechnen, dann ist es sinnvoll, das Datenwort um 4 Stellen zu verlängern. In beiden Fällen ist die Wahrscheinlichkeit einen überschriebenen Cluster zu übersehen geringer als $1,53 \cdot 10^{-5}$. Die Redundanz $\frac{n_2-k_2}{n_2}$ nimmt Dank konstanter Differenz $n_2 - k_2$ für größere n_2 ab, weshalb es sinnvoll ist, Datenmatrizen mit vielen Zeilen auf diese Art zu codieren.

Natürlich kann man durch Vergrößern der Differenz $n_2 - k_2$, also durch Hinzufügen weiterer Bytes, die Wahrscheinlichkeit einen überschriebenen Cluster zu übersehen noch weiter reduzieren, allerdings lässt sich leicht berechnen, dass für die hier gewählten Größen die Sicherheit schon sehr gut ist. Bezeichnen wir mit X die Anzahl überschriebener Cluster pro $(n_2 \times n_1)$ -Codematrix, die wir fälschlicherweise für $[n_2, k_2]$ -Codewörter halten, dann interessiert uns die Wahrscheinlichkeit $P(X > 0)$:

$$\begin{aligned} P(X > 0) &= 1 - P(X = 0) \\ &= 1 - \binom{n_1}{0} p_i^0 (1 - p_i)^{n_1-0} \\ &= 1 - (1 - p_i)^{n_1} \end{aligned}$$

Der ungünstigste Fall für uns wäre es, wenn n_1 sehr groß wird, da dadurch die Wahrscheinlichkeit sich zu irren auch größer werden würde. Nehmen wir also an $n_1 = 255$ und setzen wir $p_i = 1,53 \cdot 10^{-5}$, damit wir mit größtmöglicher Wahrscheinlichkeit einen überschriebenen Cluster übersehen. Dann sehen wir, dass die Irrtumswahrscheinlichkeit $P(X > 0) = 3,89 \dots \cdot 10^{-3}$ ist. Wir können uns also zu 99,6% sicher sein, dass alle Auslöschungen im $[n_1, k_1]$ -Code erkannt werden. Ein Vergrößern der Redundanz $n_2 - k_2$ ist daher nicht notwendig und verbraucht nur mehr Speicherplatz.

Bemerkung 16. Wir wählen von jetzt an den $[n_2, k_2]$ -Code so, dass wir einen Fehler und zwei Auslöschungen erkennen können, also $n_2 - k_2 = 4$ (siehe Abbildung 11). Wir können daher mit einer Wahrscheinlichkeit von mindestens 99,6% (abhängig von der Codewortlänge n_1) alle Auslöschungen im $[n_1, k_1]$ -Code erkennen.

Widmen wir uns nun der zweiten Frage. Wir haben soeben gesehen, dass es bei entsprechender Wahl von n_2 als relativ unwahrscheinlich anzusehen ist, dass ein überschriebener Cluster nicht erkannt wird. Die Anzahl an Auslöschungen ϵ bei den $[n_1, k_1]$ -Codewörtern entspricht daher der Anzahl an Fehlern (manipulierten Clustern) aus Variante 1 (Kapitel 8.1). Mit t_{C_1} bezeichnen wir die Anzahl Fehler an unbekanntem Stellen, die unser $[n_1, k_1]$ -Code korrigieren können soll.

Abhängig von der gewünschten Sicherheit kann man davon ausgehen, dass man zusätzlich nur 1 Fehler korrigieren können möchte, also $t_{C_1} = 1$ ist. Nachdem wir aus den Kapiteln 5.1.1 und 5.2.2 wissen, dass der GS Algorithmus bis zu $t_{GS} = n - \epsilon - 1 - \lfloor \sqrt{(n - \epsilon)(k - 1)} \rfloor$ Fehler korrigieren kann bzw. bis zu $t = \lfloor \lfloor \frac{n-k}{2} \rfloor - \frac{\epsilon}{2} \rfloor$ Fehler auf jeden Fall korrekt decodiert werden, wir aber nur t_{C_1} Fehler korrigieren wollen, können wir mit diesen Gleichungen die benötigte Länge für n_1 berechnen.

Um sicher zu decodieren ist n_1 das kleinste n für das

$$t_{C_1} \leq \left\lfloor \frac{n - k_1}{2} \right\rfloor - \frac{\epsilon}{2}$$

gilt bzw. um die besseren Decodierungseigenschaften des GS Algorithmus auszunutzen, muss

$$t_{C_1} \leq n - \epsilon - 1 - \left\lfloor \sqrt{(n - \epsilon)(k_1 - 1)} \right\rfloor$$

sein. Wobei die erwartete Anzahl an Auslöschungen ϵ gleich der Anzahl der erwarteten Fehler $\mathbb{E} = \lceil (1 - S)n \rceil$ aus Variante 1 ist. Abhängig von der gewünschten Wahrscheinlichkeit für die richtige Korrektur (50%, 84,13% oder 97,72%, siehe Seite 82) betrachten wir diese Anzahl noch zuzüglich der einfachen oder doppelten Standardabweichung $\widehat{\sigma} = \sqrt{(1 - S)Sn}$. Um jetzt nicht zu viele Fälle behandeln zu müssen, reduzieren wir unsere Untersuchungen auf den Fall $t_{C_1} = 1$. Zur Steigerung der Sicherheit kann man natürlich $t_{C_1} > 1$ wählen, was jedoch den Speicherplatzbedarf erhöht und zu einer höheren Redundanz führt.

In den Abbildungen 12 und 13 sehen wir die benötigte Codewortlänge n_1 abhängig von der Länge k_1 des Datenworts. Um diese Variante mit Variante 1 vergleichen zu können, werden dieselben Unterscheidungen wie bei Variante 1 getroffen. Dabei fällt auf, dass vor allem bei geringerer Stabilität bei weitem längere Datenwörter codiert werden können. Grund dafür ist natürlich, dass wir fast ausschließlich Auslöschungen (genauer $t - t_{C_1}$ bzw. $t_{GS} - t_{C_1}$ Auslöschungen, wobei $t_{C_1} = 1$ ist) und keine Fehler an unbekannter Stelle korrigieren müssen. Außerdem ist der Unterschied zwischen den drei Wahrscheinlichkeiten für richtige Korrektur bei hoher Stabilität noch geringer. Wir können also durch nur geringes Vergrößern der Redundanz die Wahrscheinlichkeit richtig zu decodieren stark erhöhen.

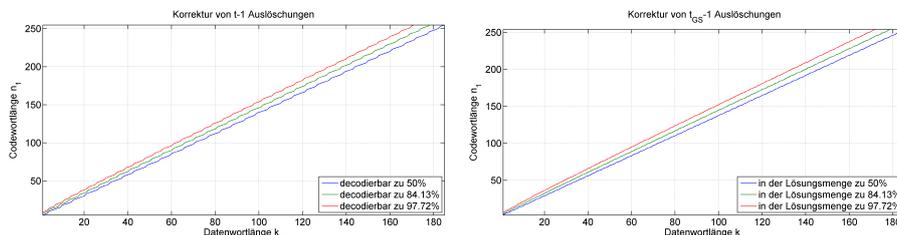


Abbildung 12: Die benötigte Codewortlänge n_1 bei Stabilität von 73,7%, um zu gegebener Wahrscheinlichkeit $t - 1$ bzw. $t_{GS} - 1$ Auslöschungen zu korrigieren.

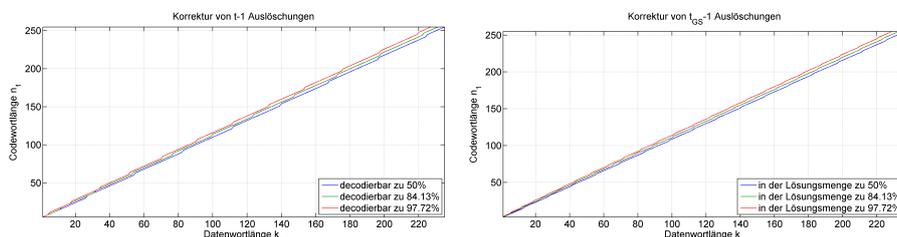


Abbildung 13: Benötigte Codewortlänge bei Stabilität von 93,4%, um zu gegebener Wahrscheinlichkeit $t - 1$ bzw. $t_{GS} - 1$ Auslöschungen zu korrigieren.

Ein wesentlicher Aspekt, der bis jetzt noch nicht betrachtet wurde, ist die entstandene Redundanz. Um die ursprüngliche Information wiederherzustellen, müssen

zusätzliche Informationen gespeichert werden. Diese möchten wir natürlich so gering wie möglich halten. Betrachten wir dazu vergleichsweise die Redundanz R_1 von Variante 1 (Kapitel 8.1):

$$R_1 = \frac{n-k}{n} = 1 - \frac{k}{n}$$

Diese wird mit zunehmender Länge des Datenworts immer kleiner, allerdings stoßen wir sehr schnell an die Grenzen der Länge der codierbaren Daten (siehe Abbildungen 8, Seite 85 und 9, Seite 85).

Durch die Verwendung einer Datenmatrix für Variante 2 können natürlich größere Datenmengen codiert werden. Deren Größe ist $k_1 \cdot k_2$ Buchstaben, womit wir maximal $k_1 \cdot 251$ Byte codieren können, da der $[n_2, k_2]$ -Code ja nur eine Verlängerung um 4 Buchstaben ist und in $GF(2^8)$ ein Codewort maximal Länge $q - 1 = 255$ haben kann. Der größtmögliche Wert für k_1 ist abhängig von der gewünschten Wahrscheinlichkeit, mit der decodiert werden soll, der Stabilität und dem Decodierungsverfahren. Damit pendelt der Wert zwischen 171 und 237. Im ungünstigsten Fall können also maximal $171 \cdot 251 = 42.921$ Byte codiert werden, was einer Redundanz von 33,99% entspricht.

Für etwas anschaulichere Ergebnisse beschränken wir uns ab jetzt auf den GS Algorithmus zur Decodierung, wobei die Parameter so gewählt werden, dass unser Datenwort zu 97,72% in der Lösungsmenge zu finden ist. Die Redundanz R_2 für die Verwendung der Variante 2 berechnen wir mittels:

$$R_2 = 1 - \frac{k_1 k_2}{n_1 n_2}$$

Wobei wir ja bereits festgelegt haben, dass $n_2 = k_2 + 4$ ist. Welche Werte für n_1 gewählt werden müssen, können wir den Abbildungen 12 und 13 (jeweils rechte Grafik) bzw. den dazugehörigen Berechnungen entnehmen.

Vergleichen wir nun unsere zwei Varianten, wobei wir in Variante 1 gesehen haben, dass wir bei einer Stabilität $S = 73,7\%$ maximal 118 Bytes und bei $S = 93,4\%$ maximal 208 Bytes (Abbildung 8, Seite 85 und Abbildung 9, Seite 85 jeweils rechte Grafik) codieren können. Um Variante 2 mit diesen beiden vergleichen zu können, verlangen wir von dem Algorithmus die gleiche Datenmenge zu verwenden, d.h. $k_1 \cdot k_2 = 118$, bzw. 208. Abbildungen 14 und 15 zeigen uns, dass Variante 1 ganz klar eine bessere Redundanz hat (unter 55% bzw. unter 20%), jedoch Variante 2 bei günstiger Wahl von k_1 und k_2 mit rund 65% und 40% nicht allzu weit entfernt ist.

Doch die Stärke von Variante 2 zeigt sich erst bei größeren Datenmengen. Wie schon gezeigt, können, abhängig von verschiedenen Voraussetzungen, zwischen etwas weniger als 43kB und etwas mehr als 59kB pro Datenmatrix codiert werden. Größere Datenmengen müssten in kleinere Blöcke unterteilt werden. Nehmen wir an wir wollen z.B. 32 kB codieren. In diesem Fall wissen wir, dass $k_1 \cdot k_2 \geq 32.000$ gelten muss, womit aber noch nicht klar ist, wie die Datenmatrix dimensioniert sein muss, um möglichst wenig Redundanz aufzuweisen. Genauso wie bei dem Vergleich von gerade eben, wollen wir daher überlegen, mit welcher Redundanz bei welcher Wahl von k_1 und k_2 zu rechnen ist.

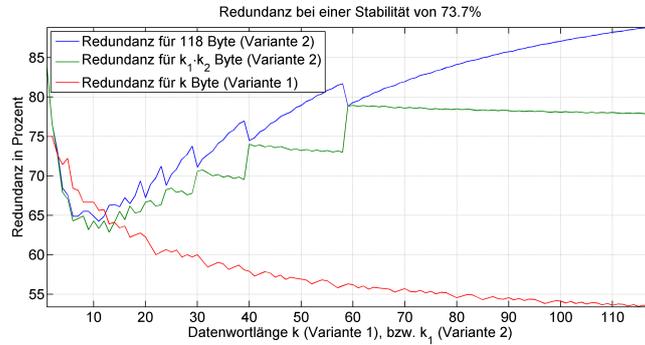


Abbildung 14: Redundanz zu gegebener Datenwortlänge von 118 Byte.

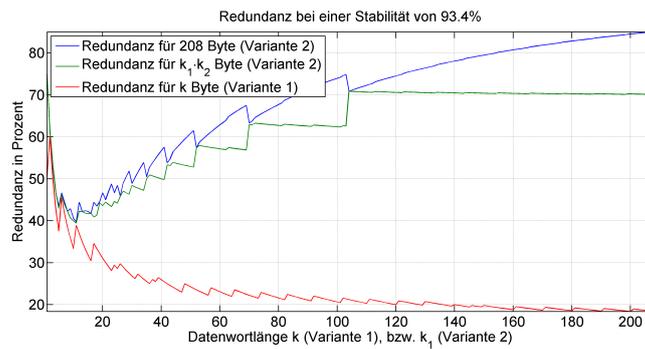


Abbildung 15: Redundanz zu gegebener Datenwortlänge von 208 Byte.

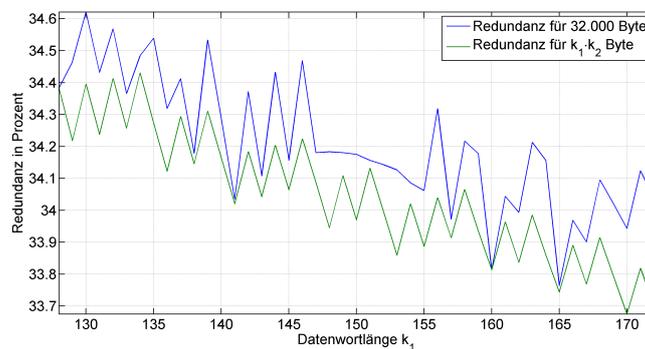


Abbildung 16: Redundanz bei Datenmatrix mit 32kB und $S = 73,7\%$.

Um einen Vergleich zu haben, betrachten wir sowohl die Redundanz der Daten, also

$$1 - \frac{32.000}{n_1 n_2}$$

als auch die Redundanz der Datenmatrix

$$1 - \frac{k_1 k_2}{n_1 n_2}$$

da ja das Produkt $k_1 \cdot k_2$ im Normalfall echt größer als 32.000 sein wird. Wir werden dadurch sehen, dass trotz Hinzunahme von bis zu $n_1 - 1$ bzw. $n_2 - 1$ nicht vorhandener Symbole, die Redundanz nur geringfügig abweicht (im Unterschied zu Abbildungen 14 und 15 für große n_1). Die minimale Datenwortlänge k_1 berechnen wir uns mittels

$$k_1 = \left\lceil \frac{32.000}{251} \right\rceil$$

da maximal 251 Buchstaben in einer Spalte sein dürfen. Für wachsendes k_1 erhalten wir durch

$$k_2 = \left\lceil \frac{32.000}{k_1} \right\rceil$$

die zweite Datenwortlänge. Die Längen n_1 und n_2 der Codewörter lassen sich durch

$$n_2 = k_2 + 4$$

(da wir ja bis zu einem Fehler pro Spalte mit dem $[n_2, k_2]$ -Code korrigieren wollen) und

$$n_1 = \min_n \left\{ 1 \leq n - \epsilon - 1 - \left\lfloor \sqrt{(n - \epsilon)(k_1 - 1)} \right\rfloor \right\}$$

(wir wollen ja bis zu einen Fehler mit dem $[n_1, k_1]$ -Code und dem GS Algorithmus korrigieren können) berechnen. Wobei ϵ die von der Länge n_1 abhängig Anzahl an Fehlern aus Variante 1 ist ²⁷ (siehe Abbildung 6, Seite 83 und Abbildung 7, Seite 83). So ist es möglich, abhängig von k_1 die Redundanz $R_2 = 1 - \frac{k_1 k_2}{n_1 n_2}$ anzugeben.

In Abbildung 16 sehen wir, dass wir uns trotz einer Stabilität von nur $\mathcal{S} = 73,7\%$ bei einer Redundanz von ungefähr 34% befinden, wobei wir bei größerem k_1 noch unter diese Marke rutschen. Dies ist eine deutliche Verbesserung zu den etwas weniger als 55%, die uns Variante 1 liefert. Und auch im Fall der höheren Stabilität von $\mathcal{S} = 93,4\%$ sehen wir in Abbildung 17, dass wir eine Redundanz von ungefähr 13%, bei guter Wahl von k_1 sogar weniger, erreichen. Auch dieser Wert ist deutlich besser als die 20% von Variante 1.

Betrachten wir noch die maximal mögliche Datenmenge, die wir bei den jeweiligen Stabilitäten (bei Verwendung des GS Algorithmus) codieren können, um mit 97,72%iger Wahrscheinlichkeit sicher decodieren zu können. Während sowohl bei einer Stabilität von $\mathcal{S} = 73,7\%$, als auch bei $\mathcal{S} = 93,4\%$ $k_2 = 251$ und $n_2 = 255$ ist,

²⁷Wie bereits erwähnt entsprechen die Auslöschungen in Variante 2 den Fehlern aus Variante 1.

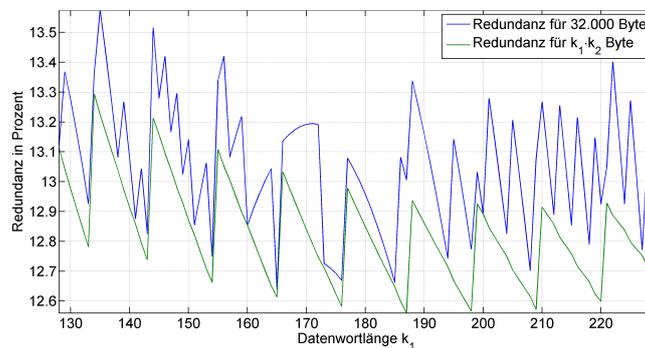


Abbildung 17: Redundanz bei 32kB und $S = 93,4\%$.

müssen wir zur Berechnung von k_1 ($n_1 = 255$) die Anzahl an Auslöschungen (siehe Abbildungen 6, Seite 83 und 7, Seite 83) berücksichtigen. Daher müssen wir zum einen mit $\epsilon = 82$, zum anderen mit $\epsilon = 25$ Auslöschungen rechnen. k_1 lässt sich dann mit Hilfe folgender Ungleichung berechnen:

$$1 \leq n_1 - \epsilon - 1 - \left\lfloor \sqrt{(n_1 - \epsilon)(k_1 - 1)} \right\rfloor$$

Damit ergeben sich folgende Werte:

Stabilität	$S = 73,7\%$	$S = 93,4\%$
maximales k_1	170	227
maximale Datenmenge	42.670 Byte	56.977 Byte
Redundanz	34,37...%	12,37...%

In Summe kann man sagen, dass die Verwendung der Variante 2 bei größeren Datenmengen sinnvoll ist. Allerdings benötigt diese Variante $n_2 < 256$ freie Bytes pro Cluster, wobei durchschnittlich zwischen 1.280 und 1.792 Bytes (siehe Kapitel 2.3) verfügbar sein sollten. Auf Grund der Größe der Datenmatrix $k_1 \times k_2$ wird es häufig nötig sein, die zu speichernden Daten in der letzten Zeile bzw. Spalte um die entsprechende Anzahl an Nullen zu ergänzen, um die Datenmatrix zu füllen. Wie gezeigt, ändert dies jedoch nicht allzu viel an der Redundanz. Für kurze Datensätze ist Variante 1 weiterhin besser geeignet.

8.3 Variante 3

Wenn wir nun die dritte Variante betrachten, dann sehen wir, dass wir von vornherein eine gegebene Größe für die Datenmatrix haben. Dies hat zwar den Vorteil, dass wir sehr genau abschätzen können mit welchem Datenverlust wir rechnen müssen (sofern wir unseren empirischen Messungen vertrauen), allerdings haben wir unter anderem folgende Nachteile:

- Zum einen haben kleinere Datenmengen, die codiert werden, eine enorm große Redundanz,
- zum anderen müssen, sollten wir z.B. mehrere Dokumente codieren wollen, mehrere Dateien zuerst zusammengefasst und anschließend gemeinsam codiert werden, da es ja außerhalb unserer Datenmatrix keinen Platz mehr im Slack Space gibt.

Ein weiterer Nachteil betrifft die Verteilung der Fehler. Im Unterschied zu Variante 2, bei der wir den $[n_2, k_2]$ -Code verwendet haben um überschriebene Cluster zu identifizieren, können wir das bei dieser Variante nicht machen. Das liegt daran, dass der gesamte freie Bereich der Cluster beschrieben wird und die Cluster normalerweise unterschiedliche Größen haben. Somit sind die Fehler in der Datenmatrix beliebig verteilt.

Aus diesem Grund muss die Redundanz so gewählt werden, dass der Produktcode die zu erwartende Anzahl an Fehlern

$$\mathbb{E} = \lceil (1 - S) \cdot (\text{Größe Slack Space}) \rceil$$

korrigieren kann. Wie wir aus [4, Seite 30] wissen, ist für einen Produktcode die Minimaldistanz $d = d_1 d_2$. Daher können wir, obwohl Produktcodes bei geeigneter Wahl eines Decodierungsverfahrens mehr leisten können, im Grunde nur

$$\left\lfloor \frac{d_1 d_2 - 1}{2} \right\rfloor$$

Fehler korrigieren. Um diese Anzahl möglichst groß zu machen, ist es natürlich sinnvoll, wenn d_1 und d_2 möglichst groß sind. Doch damit wird die Redundanz immer größer. Wenn wir also davon ausgehen, dass wir eine konkrete Datenmenge haben, dann sehen wir, dass $d_1 = d_2$ sein sollte, damit die Minimaldistanz und damit die Fehlerkorrektur möglichst groß ist.

Damit jedoch auf der anderen Seite die Redundanz R_3 , die wir mittels

$$R_3 = 1 - \frac{k_1 k_2}{n_1 n_2} = \frac{n_1 d_2 + n_2 d_1 - d_1 d_2}{n_1 n_2}$$

berechnen, möglichst gering ist, sollte der Zähler (der Nenner ist ja die Gesamtgröße des Slack Space und damit konstant) möglichst klein werden. Mit der Wahl von $d_1 = d_2$ ist natürlich $d_1 d_2$ größtmöglich. Damit muss also nur noch die Summe $n_1 + n_2$ so klein wie möglich sein, wobei $n_1 n_2$ konstant ist. Das erreichen wir indem $n_1 = n_2$ gewählt wird. Wir sehen also, dass wir, nachdem es nicht möglich ist etwas über die Fehlerverteilung zu sagen, die Datenmatrix quadratisch wählen sollten, sodass der $[n_1, k_1]$ -Code mit dem $[n_2, k_2]$ -Code übereinstimmt.

Betrachten wir mit diesen Überlegungen die geschätzte benötigte Redundanz in groben Zügen: Mit unseren soeben durchgeführten Überlegungen können wir sagen, dass $n = n_1 = n_2$ gewählt werden kann, wobei n^2 die Größe des Slack Space ist. Außerdem ist $d = d_1 = d_2$, weshalb wir unsere Redundanz R_3 vereinfachen können zu:

$$R_3 = \frac{2nd - d^2}{n^2}$$

Nachdem beim Decodieren mindestens $\mathbb{E} = \lceil (1 - S) \cdot (\text{Größe Slack Space}) \rceil$ Fehler korrigiert werden sollen, muss

$$\frac{d^2 - 1}{2} \geq (1 - S) n^2$$

sein. Aus dieser Ungleichung können wir für die Minimaldistanz d folgern, dass

$$d \geq \sqrt{2(1 - S)n^2 + 1}$$

ist. Für eine möglichst geringe Redundanz sollte diese klein sein, weshalb Zwecks Vereinfachung angenommen wird, dass $d = \sqrt{2(1-S)}n$ ist. Damit lässt sich die Berechnung der Redundanz vereinfachen zu folgender Schätzung:

$$R_3(S) \approx \frac{2n^2 \sqrt{2(1-S)} - 2(1-S)n^2}{n^2} = 2\sqrt{2(1-S)} - 2(1-S)$$

Einsetzen der bis jetzt verwendeten Stabilitäten zeigt uns, dass die Redundanz $R_3(S)$ weit größer als die bisher berechneten Redundanzen R_1 und R_2 ist:

$$R_3(0,737) = 0,9245\dots, \text{ bzw. } R_3(0,934) = 0,5946\dots$$

Natürlich kann ein Produktcode deutlich mehr Fehler korrigieren, wenn das Decodierungs-Verfahren entsprechend gewählt wird, wie man z.B. beim CIRC [13, Seite 80] oder in [4, Seite 30] nachlesen kann. Allerdings erscheint bei derartig großen Redundanzen, die bei beiden Stabilitäten S auftreten, nicht sicher, ob sich eine bessere Lösung als bei Variante 2 finden lässt. Vor allem da die Standardabweichung außer acht gelassen und nur der Erwartungswert betrachtet wurde.

Eine weitere Methode den Algorithmus zu verbessern ist natürlich die Wahl des GS Algorithmus zur Decodierung (analog zu Variante 2). Wir wissen jedoch, dass dies nur dazu führt, dass bei mehr als $\frac{d^2-1}{2}$ und weniger als t_{GS} Fehlern, das echte Datenwort Teil der Lösungsmenge ist. Man müsste sich für den Decodierungsalgorithmus also einen Schritt einfallen lassen bei dem überprüft wird, ob ein anderes Datenwort aus der Lösungsmenge ein Codewort liefert, dass besser in die Datenmatrix passt. Dies ist eine Komplikation die den Algorithmus sehr viel langsamer werden lässt.

Auch die Tatsache, dass wir das Alphabet erst nach Analyse der Größe des Slack Space wählen können scheint sehr mühsam und umständlich zu sein. Alles in allem erscheint eine weitere Analyse dieser Variante auf Grund der genannten Nachteile als nicht sinnvoll.

8.4 Schlussfolgerung

Slack Space kann immer dann gefunden werden, wenn Daten jedweder Art gespeichert werden. Das Betriebssystem ist, da auf vielen Computern gleich, nur eine der zu betrachtenden Möglichkeiten. Auch Anwendungen die auf vielen Computern laufen, wie z.B. Textverarbeitungs-, oder Tabellenkalkulationsprogramme, sowie Internetbrowser, usw. benötigen Speicherplatz und liefern uns daher Bereiche in denen wir eine Stabilität messen und daher Daten systematisch speichern können.

Ein wesentlicher Punkt, den es zu berücksichtigen gilt, ist die Begrenzung des Slack Space. Daher muss der vorhandene Speicherplatz möglichst platzsparend belegt werden. Um andererseits sicher zu stellen, dass trotz Datenverlusts die eigentliche Information nicht verloren geht, muss eine von der Stabilität abhängige Redundanz gewählt werden, die groß genug ist, um mit hoher Wahrscheinlichkeit die ursprünglichen Daten wiederherzustellen.

In den bisherigen Ausführungen in den Kapiteln 8.1, 8.2 und 8.3 haben wir drei Varianten kennen gelernt und analysiert mit denen Daten im Slack Space des Betriebssystems gespeichert werden können, sodass trotz des Verlusts von mehreren Clustern

die codierte Information wiederhergestellt werden kann. Sowohl bei Variante 1, als auch bei den Produktcodes der Varianten 2 und 3 verwenden wir zur Decodierung den GS Algorithmus (siehe Kapitel 8.1). Dazu folgende Anmerkung:

Wie wir im Kapitel über den GS Algorithmus und seine Eigenschaften (Kapitel 5.1 und Kapitel 6.1) gesehen haben, ist es durchaus möglich, dass uns der Algorithmus mehrere (bis zu L_m) mögliche Datenwörter liefert, deren Codewörter mit dem ausgelesenen Wort $\beta(x)$ an mindestens $n - t_{GS}$ Stellen übereinstimmen. Gehen wir von der *Maximum Likelihood*-Methode aus, dann würden wir das Datenwort wählen, dessen Codewort die meisten Übereinstimmungen mit $\beta(x)$ hat. Dies ist sinnvoll und liefert uns die richtigen Ergebnisse, wenn maximal $t = \lfloor \frac{n-k}{2} \rfloor$ geschehen sind. In diesem Fall würde es aber auch reichen den BM Algorithmus (Kapitel 5.2) zu verwenden, der uns ja nur eine Lösung liefert, sofern höchstens t Fehler aufgetreten sind.

Die Verwendung des GS Algorithmus ist also sinnvoll, um mehrere Fehler korrigieren zu können und in manchen Fällen Datenwörter zu erhalten, die der BM Algorithmus nicht hätte liefern können. Allerdings sollte man bei der Decodierung auch die weniger wahrscheinlichen Lösungen berücksichtigen und ausgeben, sofern vorhanden. Die Berechnung der durchschnittlichen Anzahl an Codewörtern in einer beliebigen t_{GS} -Kugelumgebung liefert uns eine gute Schätzung wie oft wir mit einem derartigen Fall rechnen müssen:

$$\bar{L}(t_{GS}) = q^{k-n} \sum_{i=0}^{t_{GS}} \binom{n}{i} (q-1)^i$$

Nachdem uns die Laufzeit des Algorithmus vorrangig nicht interessiert, ist also die Verwendung des in Kapitel 8.1 beschriebenen Decodierungsverfahrens, das die Verwendung des GS Algorithmus vorsieht, eine gute Wahl um tendenziell mehr Fehler zu korrigieren. Dies ist besonders für Variante 1 wichtig, da wir hier keinerlei Möglichkeit haben festzustellen ob Slack Space in den von uns benutzten Clustern überschrieben wurden. Diese erste Variante eignet sich besonders gut, um geringe Datenmengen, wie z.B. Passwörter oder ähnliches im Slack Space zu verstecken. Zwar muss mit einigen Fehlern gerechnet werden, der GS Algorithmus hat aber bei entsprechend hoher Redundanz gute Korrektoreigenschaften.

Abbildung 8, Seite 85 und Abbildung 9, Seite 85 zeigen uns die minimal benötigte Länge n für Codewörter, um entsprechend sicher zu decodieren. Sollten wir diese gleich mit $n = q - 1$ fixieren, finden wir in Abbildung 4, Seite 71 einen Überblick über die benötigten Parameter für den GS Algorithmus. Gehen wir von einer Stabilität von $S = 93,4\%$ aus, dann kann das Datenwort laut Abbildung 9 aus etwas mehr als 200 Byte bestehen, um mit hoher Wahrscheinlichkeit keinen Informationsverlust zu erhalten. Erneut in Abbildung 4 sehen wir, dass bei Wahl von $k = 200$ bereits 2 zusätzliche Fehler (also in Summe $27+2 = 29$ Fehler) Dank des GS Algorithmus korrigiert werden können.

Es lassen sich also Daten, die aus wenigen Hundert Byte bestehen, sehr gut mit Variante 1 codieren. Wie wir jedoch in Abbildung 16, Seite 92 und 17, Seite 94 sehen, schaffen wir es mit Variante 2 die Redundanz für größere Datenmengen wie z.B. 32kB auf rund 34%, bzw. 13% zu reduzieren (im Vergleich zu knapp 55%, bzw. 20% bei Variante 1, Abbildungen 14 und 15). Ein Wechsel auf diese Methode ist also durchaus

sinnvoll, sollten größere Datenmengen codiert werden müssen.

Überlegen wir uns noch kurz, ab welcher Datenmenge ein Umstieg von der ersten auf die zweite Variante sinnvoll ist. Wir haben bereits gesehen, dass bei Verwendung des GS Algorithmus und der Vorgabe, dass möglichst sicher (zu 97,72%) decodiert werden soll, abhängig von der Stabilität das Datenwort maximal 118 bzw. 208 Byte lang sein darf. Das bedeutet, wir haben pro Datenwort eine Redundanz von 137 ($\approx 53,7\%$), bzw. 47 ($\approx 18,4\%$) Byte. Zur einfacheren Berechnung betrachten wir nur die Vielfachen der Datenwortlängen, da so die Redundanz für Variante 1 konstant bleibt.

Für die Stabilität von 73,7% liefern uns die bereits gezeigten Formeln und Gleichungen, dass wir bereits bei $118 \cdot 3 = 354$ Bytes mit Variante 2 eine Redundanz von nur mehr rund 55% haben. Ab 472 Byte unterbietet die zweite Variante bereits die erste (siehe Abbildung 18, linke Grafik) mit einer Redundanz von unter 54% und einer Länge k_1 von knapp etwas mehr als 20, weshalb auch k_2 etwas mehr als 20 Byte hat. Dies ist wenig überraschend, da wir ja $R_2 = 1 - \frac{k_1 k_2}{n_1 n_2}$ minimal halten wollen, was für maximales $k_1 k_2$ der Fall ist.

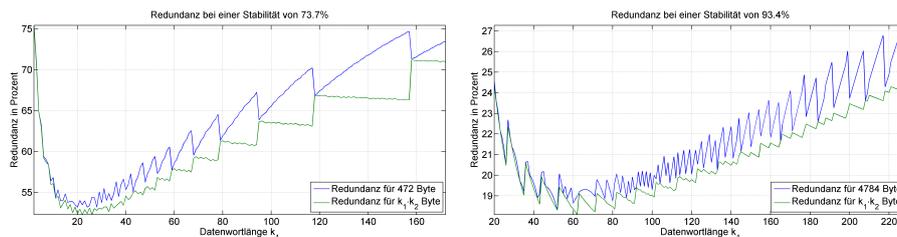


Abbildung 18: Benötigte Datenwortlänge k_1 damit Variante 2 bessere Redundanz liefert als Variante 1

Sollte die Stabilität 93,4% betragen, so müssen wir Vielfache von 208 betrachten, da für $k = 208$ die Codewortlänge $n = 255$ sein muss. Hier sehen wir, dass wir auf Grund der geringen Redundanz von nur $255 - 208 = 47$ Byte erst ab $208 \cdot 23 = 4784$ Byte eine knappe Verbesserung der Redundanz von Variante 2 (18,3%) gegenüber Variante 1 (18,4%) erreichen (siehe Abbildung 18, rechte Grafik). Je größer die Stabilität ist, desto länger lässt sich demnach Variante 1 verwenden.

Variante 3 ist leider nicht praktikabel, da, wie bereits in Kapitel 8.3 argumentiert, erst ein geeignetes Decodierungsverfahren gefunden werden muss, um die wahllos verteilten Fehler in der Datenmatrix möglichst effizient zu decodieren. Nachdem wir außerdem im Unterschied zu Variante 2 die Auslöschungen nicht identifizieren können, gehen uns wichtige Informationen verloren, die zu einer höheren Redundanz führen müssen. Daher ist eine Verwendung dieser Methode nicht zu empfehlen.

Als Letztes betrachten wir noch einmal die Stabilität des Slack Space im Betriebssystem. Wie wir in [12, Seite 190] nachlesen können, verlieren wir rund 7%, wenn ausschließlich Updates durchgeführt werden, während ein Service Pack zu einem Verlust von ca. 28% des Slack Space führt. Was die Wahl der Parameter k und n betrifft ist es also auch sinnvoll, sich zu überlegen, wie lange die Daten im Slack Space sein

sollen, bzw. ob mit der Einspielung eines Service Packs in der nächsten Zeit gerechnet werden muss. Ohne Service Packs kann man einen Datenverlust von 6-8% erwarten (wenn man Win XP außer acht lässt), weshalb die benötigte Redundanz deutlich geringer ausfällt als bei den 25-30% Datenverlust, wenn auch Service Packs eingespielt werden.

Grundsätzlich ist es wichtig, festzustellen mit welcher Stabilität des Slack Space (egal ob Betriebssystem oder sonstigem Speicherbereich) man rechnen kann, da dieser Wert Grundlage für alle Kalkulationen betreffend der Fehlerkorrektur und deren Wahrscheinlichkeit ist. Da wir diesen Wert im Vorhinein jedoch im Normalfall nicht kennen, können wir nur versuchen auf Grund der vorhandenen empirischen Werte zu schätzen. Sollte die tatsächliche Stabilität geringer als unsere Schätzung sein, ist eine Wiederherstellung der ursprünglichen Daten nur noch mit viel Glück möglich. Erwarten müssten wir in diesem Fall eigentlich eine lücken- oder fehlerhafte Decodierung.

9 Quellenverzeichnis

Literatur

- [1] Elwyn R. Berlekamp: *Algebraic Coding theory*, McGraw-HillBook Company, 1968
- [2] Johannes Blömer: *Skriptum zur Vorlesung Algorithmische Codierungstheorie*, Uni Paderborn, Sommersemester 2007
- [3] Brian Carrier: *File system forensic analysis*, Addison-Wesley Professional, 2005
- [4] G. Dorfer: *Skriptum zur Vorlesung Fehlerkorrigierende Codes*, Institut für Diskrete Mathematik und Geometrie TU Wien, 1040 Wien, Wiedner Hauptstr. 8-10, Sommersemester 2013
- [5] George David Forney Jr.: *On Decoding BCH Codes*, *IEEE Trans. Inform. Theory*, vol. 11, no. 4, pp. 549-557, October 1965
- [6] H. Kaiser: *Skriptum zur Vorlesung Algebra*, Institut für Algebra und Computermathematik TU Wien, 1040 Wien, Wiedner Hauptstr. 8-10, Wintersemester 2002/2003
- [7] R. Kötter: *Fast Generalized Minimum-Distance Decoding of Algebraic-Geometry and Reed-Solomon Codes*, *IEEE Trans. Inform. Theory*, vol. 42, no. 3, pp. 721-736, May 1996
- [8] R. Kötter: *On Algebraic Decoding of Algebraic-Geometric and Cyclic Codes*, *Linköping Studies in Science and Technology*, no. 419, Ph.D. Dissertation, Department of Electrical Engineering, Linköping U., 1996
- [9] R. Lidl, H. Niederreiter: *Finite Fields*, Cambridge University Press , Volume 20, 1997
- [10] F.J. MacWilliams, N.J.A. Sloane: *The Theory of Error-Correcting Codes*, North-Holland Publishing Company, Volume 16, 1977
- [11] R.J. McEliece: *The Guruswami-Sudan Decoding Algorithm for Reed-Solomon Codes*, *IPN Progress Report 42-153*, May 15, 2003, California Institute of Technology, Pasadena, California, and Communications Systems and Research
- [12] M. Mulazzani, S. Neuner et al.: *Advances in Digital Forensics IX, Chapter 13: Quantifying Windows File Slack Size and Stability*, *IFIP Advances in Information and Communication Technology*, Volume 410, 2013, pp. 183-193, January 2013
- [13] K. Pohlmann: *Compact Disc Handbuch*, IWT-Verlag, 1. Auflage, 1994
- [14] R. Roth, G. Ruckenstein: *Efficient Decoding of Reed-Solomon Codes beyond Half the Minimum Distance*, *IEEE Trans. Inform. Theory*, vol. 46, no. 1, pp. 246-257, January 2000
- [15] Internet: <http://de.wikipedia.org/wiki/NTFS>

A Additionstabelle für GF(16)

Additionstabelle für Beispiele in dieser Diplomarbeit für $GF(2^{16})$. Irreduzibles Polynom ist $x^4 + x + 1$, was bedeutet, dass $\alpha^4 = \alpha + 1$ ist. Außerdem ist $1 = \alpha^0 = \alpha^{15}$.

Binär	Potenz	0	α^0	α^1	α^2	α^3	α^4	α^5	α^6	α^7	α^8	α^9	α^{10}	α^{11}	α^{12}	α^{13}	α^{14}
0000	0	0	α^0	α^1	α^2	α^3	α^4	α^5	α^6	α^7	α^8	α^9	α^{10}	α^{11}	α^{12}	α^{13}	α^{14}
0001	α^0	α^0	0	α^4	α^8	α^{14}	α^1	α^{10}	α^{13}	α^9	α^2	α^7	α^5	α^{12}	α^{11}	α^6	α^3
0010	α^1	α^1	α^4	0	α^5	α^9	α^0	α^2	α^{11}	α^{14}	α^{10}	α^3	α^8	α^6	α^{13}	α^{12}	α^7
0100	α^2	α^2	α^8	α^5	0	α^6	α^{10}	α^1	α^3	α^{12}	α^0	α^{11}	α^4	α^9	α^7	α^{14}	α^{13}
1000	α^3	α^3	α^{14}	α^9	α^6	0	α^7	α^{11}	α^2	α^4	α^{13}	α^1	α^{12}	α^5	α^{10}	α^8	α^0
0011	α^4	α^4	α^1	α^0	α^{10}	α^7	0	α^8	α^{12}	α^3	α^5	α^{14}	α^2	α^{13}	α^6	α^{11}	α^9
0110	α^5	α^5	α^{10}	α^2	α^1	α^{11}	α^8	0	α^9	α^{13}	α^4	α^6	α^0	α^3	α^{14}	α^7	α^{12}
1100	α^6	α^6	α^{13}	α^{11}	α^3	α^2	α^{12}	α^9	0	α^{10}	α^{14}	α^5	α^7	α^1	α^4	α^0	α^8
1011	α^7	α^7	α^9	α^{14}	α^{12}	α^4	α^3	α^{13}	α^{10}	0	α^{11}	α^0	α^6	α^8	α^2	α^5	α^1
0101	α^8	α^8	α^2	α^{10}	α^0	α^{13}	α^5	α^4	α^{14}	α^{11}	0	α^{12}	α^1	α^7	α^3	α^6	α^1
1010	α^9	α^9	α^7	α^3	α^{11}	α^1	α^{14}	α^6	α^5	α^0	α^{12}	0	α^{13}	α^2	α^{10}	α^4	α^6
0111	α^{10}	α^{10}	α^5	α^8	α^4	α^{12}	α^2	α^0	α^7	α^6	α^1	α^{13}	0	α^{14}	α^3	α^9	α^{11}
1110	α^{11}	α^{11}	α^{12}	α^6	α^9	α^5	α^{13}	α^3	α^1	α^8	α^7	α^2	α^{14}	0	α^0	α^4	α^{10}
1111	α^{12}	α^{12}	α^{11}	α^{13}	α^7	α^{10}	α^6	α^{14}	α^4	α^2	α^9	α^8	α^3	α^0	0	α^1	α^5
1101	α^{13}	α^{13}	α^6	α^{12}	α^{14}	α^8	α^{11}	α^7	α^0	α^5	α^3	α^{10}	α^9	α^4	α^1	0	α^2
1001	α^{14}	α^{14}	α^3	α^7	α^{13}	α^0	α^9	α^{12}	α^8	α^1	α^6	α^4	α^{11}	α^{10}	α^5	α^2	0

B Matlab Programme

In diesem Anhang befinden sich sämtliche Algorithmen, die zur Berechnung der Beispiele in dieser Diplomarbeit notwendig sind. Erklärungen, warum diese funktionieren findet man in den entsprechenden Kapiteln dieser Arbeit. Das richtige Zusammensetzen, bzw. Verschachteln der Programmteile ist dem Leser überlassen.

B.1 Grundrechnungsarten in Galoisfeldern

Zur Berechnung der Beispiele in dieser Diplomarbeit wurde ein Matlab Programm geschrieben. Für das Galoisfeld $GF(16) = GF(2^4)$ wurde als primitives Element die Zahl „2“²⁸ und als irreduzibles Polynom $x^4 + x + 1 \in \mathbb{Z}_2[x] \cong 19$ verwendet.

B.1.1 Logarithmen- und Potenzlisten

Um in Galoisfeldern einigermaßen effizient rechnen zu können, benötigen wir zuerst einmal eine Logarithmus- und eine Potenzfunktion. Mit Hilfe dieser Funktionen (eigentlich handelt es sich um Arrays) lassen sich dann die anderen Grundrechnungsarten leicht durchführen. Nachdem man bei Matlab nicht auf das 0te Element eines Array zugreifen kann, müssen jedoch gewisse Fälle gesondert betrachtet werden, wie wir später sehen.

Listing 1: Code zur Erstellung von Logarithmen- und Potenzlisten im Galoisfeld $GF(q)$

```
1 % Erzeugen eines Galoisfeldes für spätere Berechnungen
2 % q = 16 wegen GF(2^4)
3 % q-1 = Grösse der multiplikativen Gruppe von GF(q)
4 % irreduziblesPolynom = x^4+x^1+1 (= (10011) = 19)
5 % erzeugendesElement = 2
6 q=16;
7 irreduziblesPolynom=19;
8 global Log2;
9 global Pot2;
10 global erzeugendesElement;
11 erzeugendesElement=2;
12 Pot2=ones(1,q-1);
13 Pot2(1)=erzeugendesElement;
14 Log2=zeros(1,q-1);
15 for i=2:(q-1)
16     Pot2(i)=Pot2(i-1)*erzeugendesElement;
17     if Pot2(i)>(q-1)
18         Pot2(i)=bitxor(Pot2(i),irreduziblesPolynom);
19     end;
20     Log2(Pot2(i-1))=i-1;
21 end;
```

B.1.2 Addition und Subtraktion

Da wir in einem Galoisfeld mit einer 2er Potenz arbeiten, sind alle Elemente additiv selbst invers. D.h. es ist $a + b = a - (-b) = a - b$ und daher liefern Addition und Subtraktion zweier Elemente das gleiche Ergebnis, weshalb sämtliche Subtraktionen

²⁸Die Elemente des Galoisfeldes wurden mit 0, 1, 2, ..., 15 bezeichnet

in $GF(16)$ in allen Algorithmen durch Additionen ersetzt wurden und daher auch nur diese programmiert werden musste.

Listing 2: Code zur Addition von 2 oder mehr Elementen aus dem Galoisfeld $GF(q)$

```

1 function summe = GF_Add(array)
2 % Input:
3 % array = Feld mit Elementen aus GF(q)
4 % Output:
5 % summe = Summe der Elemente aus array in GF(q)
6
7 summe=0;
8 for i=1:length(array)
9     summe=bitxor(summe,array(i));
10 end

```

B.1.3 Multiplikation

Listing 3: Code zur Multiplikation von 2 oder mehr Elementen aus dem Galoisfeld $GF(q)$

```

1 function produkt = GF_Prod(array)
2 % Input:
3 % array = Feld mit Elementen aus GF(q)
4 % Output:
5 % produkt = Produkt der Elemente aus array in GF(q)
6
7 global Log2;
8 global erzeugendesElement;
9 exponent=0;
10 for i=1:length(array)
11     if array(i)==0
12         produkt=0;
13         return;
14     end;
15     exponent=mod(exponent+Log2(array(i)),length(Log2));
16 end;
17 produkt=GF_Pot(erzeugendesElement,exponent);

```

B.1.4 Division

Die Division wurde durch Multiplikation mit dem Inversen Element ersetzt. Wir berechnen also statt der Division $\frac{a}{b}$ die Multiplikation $a \cdot b^{-1}$, d.h. $\frac{a}{b} = GF_Prod([a \text{ GF_Invers}(b)])$.

Listing 4: Code zur Berechnung inverser Elemente aus dem Galoisfeld $GF(q)$

```

1 function element_inv = GF_Invers(element)
2 % Input:
3 % element = Zu invertierendes Element aus GF(q)
4 % element darf nicht Null sein.
5 % Output:
6 % element_inv = element^{-1}
7
8 global erzeugendesElement;

```

```

9  global Log2;
10 element_inv=GF_Pot(erzeugendesElement ,mod(-Log2( element ) ,...
11 ... length(Log2)));

```

B.1.5 Potenzieren

Listing 5: Code zur Potenzierung von Elementen aus dem Galoisfeld

```

1  function pot = GF.Pot(basis , exponent)
2  % Input:
3  %   basis = Element aus GF(q)
4  %   exponent = natürliche Zahl
5  % Output:
6  %   pot = basis ^{exponent}
7
8  global Log2;
9  global Pot2;
10 pot=1;
11 if basis==0
12     if exponent~=0
13         pot=0;
14     end;
15     return;
16 end;
17 exponent2=mod(Log2( basis ) * exponent , length( Pot2 ));
18 if exponent2~=0
19     pot=Pot2( exponent2 );
20 end;

```

B.2 Arbeiten mit Arrays

Da wir unter anderem auch mit Polynomen mit Koeffizienten aus dem Galoisfeld $GF(16)$ rechnen können müssen, sind noch weitere Funktionen notwendig.

B.2.1 Multiplikation von 2 Polynomen

Listing 6: Code zur Multiplikation von 2 Polynomen mit Elementen aus dem Galoisfeld $GF(q)$

```

1  function pprod = GF.Polymult(f , g)
2  % Input:
3  %   f = Polynom mit Koeffizienten aus GF(q)
4  %   g = Polynom mit Koeffizienten aus GF(q)
5  % Output:
6  %   pprod = f*g
7
8  pprod=zeros(1 , length( f ) + length( g ) - 1);
9  for i = 1 : length( f )
10     for j = 1 : length( g )
11         pprod( i + j - 1 ) = GF.Add( [ pprod( i + j - 1 ) , GF.Prod( [ f( i ) , g( j ) ] ) ] );
12     end;
13 end;

```

B.2.2 Multiplikation (Skalar*(bivariatem) Polynom)

Listing 7: Code zur Multiplikation eines (bivariaten) Polynoms mit Elementen aus dem Galoisfeld $GF(q)$ mit einem Skalar aus $GF(q)$

```

1  function mprod = GF_Skalarmult( skalar , matrix )
2  % Input:
3  %   skalar = Element aus  $GF(q)$ 
4  %   matrix = (bivariates) Polynom mit Koeffizienten aus  $GF(q)$ 
5  % Output:
6  %   mprod = skalar*matrix
7
8  [m,n]=size( matrix );
9  mprod=zeros( m, n );
10 for i=1:m
11     for j=1:n
12         mprod( i , j)=GF_Prod( [ skalar , matrix( i , j ) ] );
13     end;
14 end;

```

B.2.3 Addition von 2 Matrizen

Listing 8: Code zur Addition zweier (bivariater) Polynome mit Elementen aus dem Galoisfeld $GF(q)$

```

1  function matrix = GF_Matrixadd( f , g )
2  % Input:
3  %   f = Polynom mit Koeffizienten aus  $GF(q)$ 
4  %   g = Polynom mit Koeffizienten aus  $GF(q)$ 
5  %   Die Matrizen f und g haben gleiche Dimension
6  % Output:
7  %   matrix = f+g
8
9  [m,n]=size( f );
10 matrix=zeros( m, n );
11 for i=1:m
12     for j=1:n
13         matrix( i , j)=GF_Add( [ f( i , j ) , g( i , j ) ] );
14     end;
15 end;

```

B.3 Codierungsverfahren

Jede Decodierungsmethode setzt prinzipiell eine andere Codierungsmethode voraus. Für die Codierungsmethode mit Generatorpolynom haben wir den BM Algorithmus, für die ursprüngliche Variante mit Auswertungen bei α^i mit $i = 0, 1, \dots, n - 1$ gibt es den GS Algorithmus.

B.3.1 Codieren durch Auswertung bei α^i

Listing 9: Ursprüngliche Codierungsmethode mit Auswertung

```

1  function cw = Kodiere_RS( dw , n )
2  % Input:
3  %   dw = zu kodierendes Datenwort
4  %   n = Länge des Codeworts (muss  $\geq$  length(dw) sein)
5  % Output:

```

```

6  %   cw = Codewort
7
8  % Parameter:
9  %   k = Langes des Datenworts
10 k=length(dw);
11
12 % Berechnen des Codeworts
13 cw=ones(1,n);
14 chienFaktor=dw;
15 for i=1:(n-1)
16     cw(i)=GF_Add(chienFaktor);
17     for j=2:k
18         chienFaktor(j)=GF_Prod([chienFaktor(j),Pot2(j-1)]);
19     end;
20 end;
21 cw(n)=GF_Add(chienFaktor);

```

B.3.2 Codieren durch Multiplikation mit Generatorpolynom

Da wir hier zwei Moglichkeiten kennen gelernt haben, geben wir auch das systematisch berechnete Codewort aus.

Listing 10: Codierung mit Generatorpolynom

```

1  function [cwSys cw] = Kodierte_Gen(dw, n)
2  % Input:
3  %   dw = zu kodierendes Datenwort
4  %   n = Lange des Codeworts (muss >= length(dw) sein)
5  % Output:
6  %   cwSys = Codewort bei systematischer Codierung
7  %   cw = Codewort
8
9  % Parameter:
10 %   k = Langes des Datenworts
11 k=length(dw);
12
13 % Berechnen des Generatorpolynoms
14 g=ones(1,n-k);
15 for i=1:(n-k)
16     g(1:i+1)=GF_Polymult(g(1:i),[Pot2(i) 1]);
17 end;
18
19 % Berechnen des systematischen Codeworts
20 cwSys=zeros(1,n);
21 cwSys(end-k+1:end)=dw;
22 position=n;
23 while position>n-k
24     for i=1:(n-k)
25         cwSys(position-i)=GF_Add([cwSys(position-i),GF_Prod([...
26             ... cwSys(position),g(n-k+1-i)])]);
27     end;
28     position=position-1;
29 end;
30 cwSys(end-k+1:end)=dw;
31
32 % Berechnen des Codeworts

```

```
33 cw=GF_Polymult(dw,g);
```

B.4 Guruswami-Sudan Algorithmus

Wie wir gesehen haben besteht der GS-Algorithmus aus der Kombination zweier Algorithmen und liefert uns nach dessen Durchführung eine Liste mit allen möglichen Nachrichtenwörtern.

Listing 11: Decodieralgorithmus GS

```
1 function [Liste_cw Liste_dw Fehler] = GS(m, k, aw)
2 % Input:
3 % m = Vielfachheit (ist Integer > 0)
4 % k = Länge des ursprünglichen Datenworts
5 % aw = ausgelesenes Wort
6 % Output:
7 % Liste_cw = Liste mit möglichen Codewörtern
8 % Liste_dw = Liste mit möglichen Datenwörtern
9 % Fehler = Unterschiede zwischen den Codewörtern (Liste_cw)
10 % und aw
11
12 % Berechnung des bivariaten Interpolationspolynoms
13 polynom=Koetter(m,k,aw);
14
15 % Berechnung der f(x) für die gilt: (y-f(x)) | polynom
16 Liste_dw=RR(polynom,k);
17
18 % Berechnung der Unterschiede zwischen den Codewörtern
19 %(Liste_cw) und aw
20 n=length(aw);
21 L=size(Liste_dw,1);
22 Fehler=zeros(1,L);
23 Liste_cw=zeros(L,n);
24 for i=1:L
25 Liste_cw(i,:)=Kodierte_RS(Liste_dw(i,:),n);
26 Fehler(i)=sum(Liste_cw(i,:)~=aw);
27 end;
```

B.4.1 Kötters Interpolationsalgorithmus

Listing 12: Kötters Interpolationsalgorithmus

```
1 function ip = Koetter(m, k, aw)
2 % Input:
3 % m = Vielfachheit (ist Integer > 0)
4 % k = Länge des ursprünglichen Datenworts
5 % aw = ausgelesenes Wort
6 % Output:
7 % ip = Interpolationspolynom mit m-fachen Nullstellen
8
9 % Parameter:
10 % n = Länge des ausgelesenen Wortes
11 % v = Ordnung die zur Berechnung des Rangs benutzt wird
```

```

12 % L = max Anz. Lösungen (siehe Kapitel 6.1)
13 n=length(aw);
14 v=k-1;
15 L=floor(sqrt(n*(m+1)*m/v+(v+2)^2/(4*v^2)-(v+2)/(2*v)));
16
17 % Initialisiere mögliche Interpolationspolynome, sowie den Rang
18 % Über den 3. Index von g(.,.,.) werden die Polynome angesprochen
19 % 1. Index = Zeile (=y)
20 % 2. Index = Spalte (=x)
21 g=zeros(L+1,2,L+1);
22 rang=zeros(1,L+1);
23 for i=1:L+1
24     g(i,1,i)=1;
25     rang(1,i)=i-1+v*sum(0:i-1);
26     rang(2,i)=0;
27 end;
28
29 % Kötters Interpolations Algorithmus
30 for i=1:n
31     a=GF.Pot(erzeugendesElement,i-1);
32     b=aw(i);
33     for r=0:m-1
34         for s=0:m-1-r
35             % Initialisiere Diskrepanz
36             delta=zeros(1,L+1);
37             for j=1:L+1
38                 for p=r:length(g(1,:,j))-1
39                     for q=s:length(g(:,1,j))-1
40                         % Überprüfe, ob Binomialkoeffizient ungerade ist
41                         % (da in GF 2^p a+a=0)
42                         if mod(nchoosek(p,p-r)*nchoosek(q,q-s),2)==1
43                             delta(j)=GF.Add([delta(j),GF.Prod([g(q+1,...
44                                 ... p+1,j),GF.Pot(a,(p-r)),GF.Pot(b,(q-s))])]);
45                         end;
46                     end;
47                 end;
48             end;
49             % Bei welchen Indices existiert eine Diskrepanz~=0
50             liste_Indices=find(delta);
51
52             % Berechnen des min. Polynoms mit vorhandener Diskrepanz
53             [rang_min,j_temp]=min(rang(1,liste_Indices));
54             j_min=liste_Indices(j_temp);
55             delta_min=delta(j_min);
56             g_min=g(:,j_min);
57
58             % Vergrößern der Matrizen der Interpolationspolynome
59             % (falls notwendig)
60             if sum(g_min(:,end))~=0
61                 g(:,end+1,j)=zeros(L+1,1);
62                 g_min(:,end+1)=zeros(L+1,1);
63             end;
64
65             for j=liste_Indices

```

```

66         if j==j_min
67             temp_matrix=zeros( size(g(:, :, j)));
68             % Entspricht Multiplikation mit x:
69             temp_matrix(:, 2:end)=g(:, 1:end-1, j);
70             g(:, :, j)=GF_Skalarmult(delta_min, GF_Matrixadd(...
71                 ... GF_Skalarmult(a, g_min), temp_matrix));
72             % Berechnung des neuen Rang
73             rang(1, j)=rang(1, j)+j+floor(rang(2, j)/v);
74             rang(2, j)=rang(2, j)+1;
75         else
76             g(:, :, j)=GF_Matrixadd(GF_Skalarmult(delta_min, ...
77                 ... g(:, :, j)), GF_Skalarmult(delta(j), g_min));
78         end;
79     end;
80 end;
81 end;
82 end;
83
84 [minimum, index]=min(rang(1, :));
85 % Löschen von unnötigen Nullen
86 nullen=0;
87 while sum(g(:, end-nullen, index))==0
88     nullen=nullen+1;
89 end;
90 ip=g(:, 1:end-nullen, index);

```

B.4.2 Roth-Ruckensteins Faktorisierungsalgorithmus

Listing 13: Roth-Ruckensteins Faktorisierungsalgorithmus

```

1 function Liste = RR(polynom, k)
2 % Input:
3 % polynom = zu faktorisiertes bivariate Polynom
4 % k = Länge des ursprünglichen Datenworts
5 % Output:
6 % Liste = Liste der Polynome f(x) für die: (y-f(x)) | polynom
7
8 % Initialisierungen für den Faktorisierungsalgorithmus nach
9 % Roth-Ruckenstein
10 global grad;
11 grad=k-1;
12 global temp_Liste;
13 temp_Liste=zeros( length(polynom(:, 1)), grad+1);
14 global t;
15 t=1;
16 global loesung;
17 loesung=1;
18 red_Polynom=Reduktion(polynom);
19 global deg;
20 deg(1)=-1;
21 z=1;
22 if not(any(red_Polynom(1, :)))
23     loesung=2;
24     while not(any(red_Polynom(z, :))) && z<length(polynom(:, 1))
25         z=z+1;

```

```

26     end;
27 end;
28 DFS(0, red_Polynom(z: end, :));
29 Liste=temp_Liste(1:loesung-1,:);
30
31 % Hilfsfunktion zur itaerativen Berechnung der Polynome f(x)
32 function DFS(u, polynom)
33 global temp_Liste;
34 global loesung;
35 global grad;
36 global deg;
37 global Koeffizient;
38 global Vorgaenger;
39 global t;
40 if not(any(polynom(1, :)))
41     temp=u;
42     while temp>0
43         temp_Liste(loesung, deg(temp)+1)=Koeffizient(temp);
44         temp=Vorgaenger(temp);
45     end;
46     loesung=loesung+1;
47 elseif deg(u+1)<grad
48     Nullstellen=Horner(polynom(:, 1));
49     for a=Nullstellen
50         v=t;
51         t=t+1;
52         Vorgaenger(v)=u;
53         deg(v+1)=deg(u+1)+1;
54         Koeffizient(v)=a;
55         red_Polynom=Reduktion(Transformiere(polynom, a));
56         DFS(v, red_Polynom);
57     end;
58 end;
59
60 % Hilfsfunktion zur Division von polynom durch x^m für
61 % maximales m
62 function pol=Reduktion(polynom)
63 i=1;
64 while not(any(polynom(:, i)))
65     i=i+1;
66 end;
67 pol=zeros(size(polynom));
68 pol(:, 1:end+1-i)=polynom(:, i:end);
69
70 % Hilfsfunktion zur Berechnung der Nullstellen eines Polynoms
71 function loesungen=Horner(polynom)
72 % polynom darf nur 1-Dimensional sein
73 global Pot2
74 temp=ones(1, length(Pot2))*polynom(end);
75 for j=(length(polynom)-1):-1:1
76     for i=length(Pot2)
77         temp(i)=GF_Add([GF_Prod([temp(i), Pot2(i)], polynom(j))]);
78     end;
79 end;

```

```

80 % Pot2(temp==0) liefert die Nullstellen des Polynoms in GF
81 if polynom(1)==0
82     loesungen=[0,Pot2(temp==0)];
83 else
84     loesungen=Pot2(temp==0);
85 end;
86
87 % Hilfsfunktion zur Transformation des Polynoms  $Q(x,y) \rightarrow Q(x,xy+a)$ 
88 function return_polynom = Transformiere(pol, a)
89 [Zeilen, Spalten]=size(pol);
90 polynom = zeros(Zeilen, Zeilen+Spalten);
91 for i=1:Zeilen
92     for j=1:i
93         if mod(nchoosek(i-1,j-1),2)==1
94             polynom(j,j+j+Spalten-1)=GF_Matrixadd(polynom(j,...
95                 ... j+j+Spalten-1),GF_Skalarmult(GF_Pot(a,i-j),pol(i,:)));
96         end;
97     end;
98 end;
99 % Löschen unnötiger Nullen
100 j = 0;
101 while (j<Zeilen+Spalten) && not(any(polynom(:,end-j)))
102     j = j+1;
103 end;
104 return_polynom=polynom(:,1:Zeilen+Spalten-j);

```

B.5 Berlekamp-Massey Algorithmus

Da uns der BM-Algorithmus nur ein Codewort liefert inkludiert dieses Programm auch Codezeilen, um sich das ursprüngliche Nachrichtenwort aus dem Codewort zu berechnen. Da im Vorfeld nicht klar ist ob systematisch kodiert wurde oder nicht, werden die in beiden Fällen passenden Nachrichtenwörter berechnet und ausgegeben.

Listing 14: Berlekamp-Massey Algorithmus

```

1 function [cw dwSys dw] = BM(aw, k)
2 % Input:
3 % aw = ausgelesenes Wort
4 % k = Länge des ursprünglichen Datenworts
5 % Output:
6 % cw = Codewort
7 % dwSys = Datenwort bei systematischer Codierung
8 % dw = Datenwort bei Multiplikation mit  $g(x)$ 
9
10 % Berlekamp-Massey Algorithmus zur Berechnung des vermeintlich
11 % gesendeten Codeworts
12 cw=aw;
13 n=length(aw);
14 t=floor((n-k)/2);
15
16 % Berechnen der Syndrome
17 s=zeros(1,2*t);
18 chienFaktor=aw;
19 for i=1:(2*t)
20     for j=2:n

```

```

21     chienFaktor(j)=GF_Prod([ chienFaktor(j), Pot2(j-1)]);
22     end;
23     s(i)=GF_Add(chienFaktor);
24     end;
25
26     % Algorithmus zur Berechnung von sigma(z) (=Fehlerlokalisierungs-
27     % polynom) und omega(z) (=Fehlerwertepolynom)
28     sigma=zeros(1,2*t+1);
29     sigma(1)=1;
30     tau=zeros(1,2*t+1);
31     tau(1)=1;
32     omega=zeros(1,2*t+1);
33     omega(1)=1;
34     gamma=zeros(1,2*t+1);
35     D=0;
36     B=0;
37     for i=1:(2*t)
38         % Berechne delta = [z^i](1+s(z))*sigma(z)
39         delta=sigma(i+1);
40         for j=1:i
41             delta=GF_Add([ delta ,GF_Prod([ s(j), sigma(i+1-j) ])]);
42         end;
43         % sigma = sigma - delta * z * tau
44         sigma_alt=sigma;
45         for j=2:length(sigma)
46             sigma(j)=GF_Add([ sigma(j), GF_Prod([ delta , tau(j-1) ])]);
47         end;
48         % omega = omega - delta * z * gamma
49         omega_alt=omega;
50         for j=2:length(omega)
51             omega(j)=GF_Add([ omega(j), GF_Prod([ delta , gamma(j-1) ])]);
52         end;
53         if delta==0 || D>i/2 || (delta~=0 && D==i/2 && B==0)
54             % D und B bleiben gleich
55             tau=[0 tau(1:end-1)];
56             gamma=[0 gamma(1:end-1)];
57         else
58             D=i-D;
59             B=1-B;
60             delta_inv=GF_Invers(delta);
61             for j=1:length(tau)
62                 tau(j)=GF_Prod([ sigma_alt(j), delta_inv ]);
63             end;
64             for j=1:length(gamma)
65                 gamma(j)=GF_Prod([ omega_alt(j), delta_inv ]);
66             end;
67         end;
68     end;
69
70     % Berechne die Nullstellen von sigma, deren Inversen die
71     % Fehlerorte sind, sofern diese überhaupt vorhanden sind.
72     e=find(sigma,1,'last');
73     if e>1
74         % Berechne die Fehlerorte

```

```

75 fehlerorte=zeros(1,e-1);
76 chienFaktor=sigma(1:e);
77 counter=0;
78 i=1;
79 while i<e && counter<n
80     if GF_Add(chienFaktor)==0
81         fehlerorte(i)=GF_Invers(GF_Pot(erzeugendesElement,...
82             ... counter));
83         i=i+1;
84     end;
85     for j=2:length(chienFaktor)
86         chienFaktor(j)=GF_Prod([chienFaktor(j),Pot2(j-1)]);
87     end;
88     counter=counter+1;
89 end;
90 % Überprüfe, ob alle Lösungen von sigma berechnet wurden, was
91 % bei maximal floor((n-k)/2) Fehlern auf jeden Fall passiert
92 if e-1==find(fehlerorte,1,'last')
93     % Berechne die Fehlerwerte
94     fehlerwerte=zeros(1,e-1);
95     for i=1:(e-1)
96         % Berechnung Zähler
97         zaehler=omega(e);
98         for j=1:(e-1)
99             zaehler=GF_Add([zaehler,GF_Prod([omega(e-j),GF_Pot(...
100                 ... fehlerorte(i),j])])]);
101         end;
102         % Berechnung Nenner
103         nenner=fehlerorte(i);
104         for j=1:(e-1)
105             if j~=i
106                 nenner=GF_Prod([nenner,GF_Add([fehlerorte(i),...
107                     ... fehlerorte(j)])]);
108             end;
109         end;
110         fehlerwerte(i)=GF_Prod([zaehler,GF_Invers(nenner)]);
111     end;
112     % Korrigieren der Fehler, wobei Fehler bei l=2^0 die 0-te
113     % Stelle bedeutet und das in Matlab die Position 1 ist, ...
114     for i=1:e-1
115         cw(Log2(fehlerorte(i))+1)=GF_Add([cw(...
116             ... Log2(fehlerorte(i))+1),fehlerwerte(i)]);
117     end;
118     nichtDekodierbar=false;
119 else
120     % Falls der Dekoder - wegen zu vieler Fehler - nicht
121     % dekodieren kann, werden leere Arrays retourniert
122     cw=[];
123     nichtDekodierbar=true;
124 end;
125 end;
126
127 % Berechnen des Datenworts aus dem berechneten
128 % Codewort cw falls möglich

```

```

129 if nichtDekodierbar
130     dwSys=[];
131     dw=[];
132 else
133     % Ablesen des systematischen Datenworts
134     dwSys=cw(n-k+1:end);
135
136     % Berechnen des Datenworts mit Division durch
137     % Generatorpolynom
138     g=ones(1,2*t+1);
139     for i=1:(2*t)
140         g(1:i+1)=GF_Polymult(g(1:i),[Pot2(i) 1]);
141     end;
142     dw_temp=cw;
143     position=0;
144     while stelle <k
145         g_temp=GF_Polymult(g,dw_temp(n-position));
146         for i=1:(2*t)
147             dw_temp(n-position-i)=GF_Add([dw_temp(n-position-i),...
148                 ... g_temp(2*t+1-i)]);
149         end;
150         position=position+1;
151     end;
152     dw=dw_temp(end-k+1:end);
153 end;

```