Bonusaufgabe - Zara Zackigs Zurueckkehr

Teilnahme-ID: 60302

Bearbeitet von
Florian Bange

25. April 2022

Inhaltsverzeichnis

1 Definition des XORs

XOR bzw. $\oplus$ sei zunaechst auf zwei Bits/Wahrheitswerte, wie folgt ueber die Gleichheit, definiert:

Seien a, b zwei Wahrheitswerte.

$$a\ XOR\ b \iff a \oplus b = \neg(a \iff b)$$

Die dazugehoerige Wahrheitstabelle sieht wie folgt aus:

a	b	a $\iff b$	$\neg(a \iff b)$	a XOR b
0	0	1	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

Bzw. sieht die Verknuepfungstabelle des XORs so aus:

$\oplus$	0	1
0	0	1
1	1	0

2 Darstellung durch $\mathbb{Z}_2$ mit der Addition

Die Menge $\mathbb{Z}_2 = \{\ 0, 1\ \}$ bildet zusammen mit der Addition eine abelsche Gruppe.
Die dazugehoerige Verknuepfungstabelle sieht wie folgt aus:

+	0	1
0	0	1
1	1	0

Wie zu erkennen ist, ist diese Verknuepfungstabelle identisch zu der des XORs.
Somit kann die Verknuepfung XOR mit der Addition in $\mathbb{Z}_2$ dargestellt werden.

3 Eigenschaften des XORs

Aufgrund dessen, dass das XOR mit der abelschen Gruppe ($\mathbb{Z}_2$, +) dargestellt werden kann, gelten fuer das XOR die gleichen Eigenschaften, wie fuer die abelsche Gruppe ($\mathbb{Z}_2$, +):

Seien a, b, c beliebige Wahrheitswerte (0, oder 1), bzw. $a, b, c \in \mathbb{Z}_2$.

1. Assoziativitaet: (a $\oplus$ b) $\oplus$ c = a $\oplus$ (b $\oplus$ c)
2. Neutrales Element 0: a $\oplus$ 0 = a
3. Selbstinvers: a $\oplus$ a = 0
4. Kommutativitaet: a $\oplus$ b = b $\oplus$ a

4 Bitfolgen

Bitfolgen bestehen aus mehren aneinandergereihten Bits.
Eine Bitfolge b aus m Bits besteht aus den Bits $b_1, \ \dots\ , b_m$ und laesst sich entweder darstellen als aneinandergereihte Bits:

$$b_1\ \dots\ b_m$$

Oder als Vektor:

$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

Die Bitfolge 0 - die Null-Bitfolge - sei definiert als der Null-Vektor.
Sprich

$$0 = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

Die jeweilige Laenge ergibt sich, wie beim Null-Vektor aus dem Zusammenhang.

Wird das XOR auf mehrere Bitfolgen angewendet, wird auf die Bits aller Bitfolgen an Stelle i nacheinander das XOR angewendet. Dies wird fuer alle Stellen i der Bitfolgen durchgefuehrt. Das jeweilige Ergebnis wird in der resultierenden Bitfolge an Stelle i notiert. Dafuer muessen alle mit dem XOR verbundenen Bitfolgen die gleiche Laenge haben.

Am sinnvollsten ist dazu die Darstellung der Bitfolgen als Vektoren zusammen mit der Vektoraddition fuer Vektoren aus $\mathbb{Z}_2^m$. Dies sieht allgemein wie folgt aus:
Fuer n Bitfolgen $b_1, \ , \ \dots \ , \ b_n$ mit je m Bits, wobei

$$b_i = \begin{bmatrix} b_{i,1} \\ \vdots \\ b_{i,m} \end{bmatrix}$$

ist, ist die entstehende Bitfolge u mit

$$u = b_1 + \cdots + b_n,$$

wobei $+$ hier die Vektoraddition (fuer Vektoren aus $\mathbb{Z}_2^m$) ist.
Alternativ lassen sich diese Schreibweisen waehlen:

$$\begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix} = \begin{bmatrix} b_{1,1} \\ \vdots \\ b_{1,m} \end{bmatrix} + \cdots + \begin{bmatrix} b_{n,1} \\ \vdots \\ b_{n,m} \end{bmatrix}$$

Oder auch

$$u_1 = b_{1,1} + \cdots + b_{n,1}$$
$$\vdots$$
$$u_n = b_{1,m,} + \cdots + b_{n,m}$$

Beispiel:
Moechte man folgende Bitfolgen mit dem XOR verbinden, geht dies, wie anschlieszend in der Tabelle gezeigt.

$$1001, 1100, 1101$$

	1	0	0	1
$\oplus$	1	1	0	0
$\oplus$	1	1	0	1
$=$	1	0	0	0

Die zuvor beschriebene Vorgehensweise sieht fuer das Beispiel wie folgt aus:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Weiter sei angemerkt, dass fuer das XOR mit Bitfolgen die gleichen Eigenschaften gelten, wie bei einzelnen Bits, da das XOR fuer einzelene Bits elementweise angewendet wird.
Es gilt also ebenfalls die Assoziativitaet, das Neutralelement 0, die Eigenschaft des Selbstinversen und die Kommutativitaet.

5 Definition und Reduzierung der Aufgabe

Das Problem der Bonusaufgabe „Zara Zackigs Zurueckkehr“ (oder kurz das ZZZ-Problem) laesst sich wie folgt formal definieren:
Gegeben sind n Bitfolgen der Laenge m, und eine Ganzzahl $k > 0$.
Fuer die n Bitfolgen ist gesucht die Menge an k Bitfolgen $\{s_1, \ldots, s_k\}$, fuer welche eine weitere Bitfolge x exestiert, mit

$$s_1 \oplus \cdots \oplus s_k = x.$$

Existiert eine solche Loesung nicht, soll die leere Menge das Ergebnis sein.

Sei das Problem „Null-Bitfolge“ wie folgt definiert:
Gegeben sind n Bitfolgen der Laenge m und eine Ganzzahl $k > 0$.
Das Problem lautet, k der n gegebenen Bitfolgen zu finden, bei welchen gilt

$$s_1 \oplus \cdots \oplus s_k = 0,$$

Existiert eine solche Loesung nicht, soll die leere Menge das Ergebnis sein.

Das Null-Bitfolge Problem laesst sich auf das ZZZ-Problem reduzieren, indem man die n gegebenen Bitfolgen und $k-1$ als Eingaben fuer das ZZZ-Problem benutzt. Dadurch erhaelt man k der n Bitfolgen fuer welche gilt

$$s_1 \oplus \cdots \oplus s_{k-1} = s_k.$$

Das Null-Bitfolge Problem ist durch das Ergebnis des ZZZ-Problems ebenfalls geloest, da man die Gleichung

$$s_1 \oplus \cdots \oplus s_{k-1} = s_k$$

wie folgt umstellen kann:
Fuegt man zu beiden Seiten $\oplus\, s_k$ hinzu, erhaelt man

$$(s_1 \oplus \cdots \oplus s_{k-1}) \oplus s_k = s_k \oplus s_k.$$

Durch die Eigenschaft des Selbstinversen erhaelt man

$$(s_1 \oplus \cdots \oplus s_{k-1}) \oplus s_k = 0,$$

Weiter erhaelt man mit der Assoziativitaet

$$s_1 \oplus \cdots \oplus s_{k-1} \oplus s_k = 0.$$

Somit hat man k Bitfolgen gefunden, welche die Null-Bitfolge ergeben.
Fuer den Fall, dass die leere Menge das Ergebnis des ZZZ-Problems ist, gibt es ebenfalls keine Loesung fuer das Null-Bitfolge Problem, so das die leere Menge ebenfalls das Ergebnis ist.

Dass sich das neue Problem auf das ZZZ-Problem reduzieren laesst bedeutet, dass wenn das Null-Bitfolge Problem NP-schwer ist, auch das ZZZ-Problem NP-schwer ist.

Weiter sei gesagt, dass man das ZZZ-Problem ebenfalls auf das Null-Bitfolge Problem reduzieren kann. Dazu benutzt man die n gegebenen Bitfolgen und $k+1$ als Eingaben fuer das Null-Bitfolge Problem. Dadurch erhaelt man

$$s_1 \oplus \cdots \oplus s_{k+1} = 0.$$

Anschlieszend kann man irgendeine der $k+1$ gefundenen Bitfolgen auf beiden Seiten addieren und erhaelt

$$\begin{aligned} & s_1 \oplus \cdots \oplus s_k \oplus s_{k+1} \oplus s_{k+1} = 0 \oplus s_{k+1} \\ \Leftrightarrow\ & s_1 \oplus \cdots \oplus s_k + 0 = \oplus s_{k+1} \\ \Leftrightarrow\ & s_1 \oplus \cdots \oplus s_k = \oplus s_{k+1}. \end{aligned}$$

Dadurch hat man k Bitfolgen erhalten fuer welche eine weitere Bitfolge existiert, bei welcher die vorausgesetzte Bedingung gilt.
Fuer den Fall, dass die leere Menge das Ergebnis des Null-Bitfolge Problems ist, gibt es ebenfalls keine Loesung fuer das Null-Bitfolge Problem, so das die leere Menge ebenfalls das Ergebnis ist.

Dass wiederum bedeutet, dass man das ZZZ-Problem loesen kann, indem man das Null-Bitfolge Problem loest.

6 Komplexitaet

Das ZZZ-Problem ist NP-schwer.
Diese Aussage wird nun bewiesen, indem gezeigt wird, dass das Null-Bitfolge Problem NP-schwer ist. Dies ist moeglich, da das Null-Bitfolge Problem bereits auf das ZZZ-Problem reduziert wurde.

Um zu zeigen, dass ein Problem L NP-hard ist muss jedes Problem in NP durch eine Polynomialzeitreduktion auf L reduziert werden koennen.
Um dies zu zeigen, kann man ein als NP-vollstaendiges Problem auf L reduzieren. Das bedeutet, dass L mindestens so schwer ist, wie das Problem von welchem reduziert wird.

Das Problem von welchem reduziert wird, soll hier das „Weight Distribution“ [2] oder auch „Subspace Weights“ [1] genannte Problem sein, welches in [1] als NP-vollstaendig bewiesen wurde:
Gegeben sind eine binaere $m \times n$ Matrix A und eine Ganzzahl $k > 0$.
Die Entscheidungsfrage ist, ob es eine Menge an k Spalten aus A gibt, welche sich zum Null-Vektor aufaddieren.

Dieses Entscheidungsproblem laesst sich wie folgt durch Polynomialzeitreduktion zum ZZZ-Problem reduzieren:
Seien die Spalten der Matrix die Vektoren

$$\vec{v_1},\ \dots\ ,\ \vec{v_n}.$$

Diese Vektoren kann man als Bitfolgen interpretieren:

$$\vec{v_i} = \begin{bmatrix} v_{i,1} \\ \vdots \\ x_{i,m} \end{bmatrix}$$

wird zur Bitfolge

$$s_i = \vec{v_i}.$$

Somit erhaelt man die Bitfolgen

$$s_1,\ \dots\ ,\ s_n.$$

Diese Bitfolgen und k kann man nun als Eingabe fuer das Null-Bitfolge Problem benutzen.
Dadurch erhaelt man k Bitfolgen, welche aufaddiert die Null-Bitfolge (bestehend aus m Nullen) ergeben oder die leere Menge.
Erhaelt man k Bitfolgen, so handelt es sich um eine Ja-Instanz, erhaelt man die leere Menge, handelt es sich um eine Nein-Instanz.
Dass es sich um eine Ja-Instanz handelt, wenn k Bitfolgen das Ergebnis des Null-Bitfolge Problem sind, liegt daran, dass jede der k Bitfolgen einer Spalte der Matrix entspricht. Somit gibt es k Spalten der Matrix, welche den Null-Vektor ergeben.
Dass es sich um eine Nein-Instanz handelt, wenn die leere Menge zurueckgegeben wurde, ergibt ebenfalls Sinn. Denn in dem Fall konnten keine k Bitfolgen, sprich keine k Spalten, gefunden werden, welche die Null-Bitfolge, bzw. den Null-Vektor, bilden.

Aufgrund dessen, dass das ZZZ-Problem NP-schwer ist, kann man davon ausgehen, dass kein Algorithmus existiert, welcher das Problem effizient loest, bzw. in Polynomialzeit laeuft.

Dennoch werden nun zwei Loesungsansaetze vorgestellt, welche das Problem exakt, aber mit exponentieller Laufzeit loesen. Beide Loesungen werden das Null-Bitfolge Problem loesen, welches, wie bereits bewiesen, das ZZZ-Problem loest. Der erste Loesungsansatz basiert auf einem Bruteforceverfahren und der zweite auf dem Loesen eines linearen Gleichungssystems, welches in den Kapiteln „Loesung durch ein Gleichungssystem“ und „Loesung des Gleichungssystems mit bestimmtem Hamming-Gewicht“ erlaeutert wird.

7 Algorithmus 1: Loesung durch Bruteforce der Kombinationen

Das Null-Bitfolge Problem laesst sich loesen, indem alle moeglichen Kombinationen (ohne Wiederholung/Zuruecklegen) der Bitfolgen ausprobiert und ueberprueft werden.

Gegeben sind die n Bitfolgen der Laenge m:

$$b_1, \ \dots \ , \ b_n$$

Sowie die Anzahl an gesuchten Bitfolgen k.

Um bei diesem Loesungsweg effizient vorzugehen, werden nicht alle moeglichen Kombinationen erstellt, sondern alle Kombinationen von $\lfloor \frac{k}{2} \rfloor$ Bitfolgen (die abgerundete Haelfte von k, abgerundet auf die naechst kleinere Ganzzahl).
Sei

$$l = \lfloor \frac{k}{2} \rfloor.$$

Alle moeglichen Kombinationen der l Bitfolgen werden durchgegangen und das dazugehoerige XOR wird berechnet. Dadurch erhaelt man je ein Paar, bestehend aus den Indexen der gewaehlten Bitfolgen und der durch das XOR resultierenden Bitfolge:

$$(\{i_1, \ \dots \ , \ i_l\}, x),$$

wobei

$$x = b_{i_1} \ XOR \ \dots \ XOR \ b_{i_l}$$

ist.

Fuer jedes dieser Paare, welche durch die Kombinationen von l der n Bitfolgen entstehen, wird die Menge der Indexe in einer Liste gespeichert.
Dabei ist diese Liste ein Wert, assoziert mit der Bitfolge.
Es entsteht also eine Tabelle, in welcher von Bitfolgen x zu Listen L_x gezeigt wird. Diese Listen (L_x) beinhalten stets alle Mengen (der Groesze l) an Indexen, dessen Bitfolgen x ergeben.
Der Pseudocode zu der Erstellung dieser Tabelle, welcher als Uebergabeparameter die Bitfolgen und l erhaelt, sieht wie folgt aus:

```
1: procedure CREATETABLE(b_1 ... , b_n, size)
2:     table ← [ ]
3:     firstCombination ← getFirstCombination(n, k)
4:     currentCombination ← firstCombination
5:     while currentCombination ≠ ∅ do
6:         x ← b_currentCombination_1 XOR ... XOR b_currentCombination_size
7:         L_x ← table[x]
8:         if L_x = NULL then ▷ After obtaining L_x, check if the list is empty. If so, create it and set it in table.
9:             L_x ← [ ]
10:            table[x] ← L_x
11:        end if
12:        L_x.append(currentCombination)
13:        currentCombination ← getNextCombination(currentCombination)
14:    end while
15:    return table
16: end procedure
```

In diesem Pseudocode sollen die Methoden getFirstCombination und getNextCombination je die erste Kombination und die naechste Kombination an Indexen der Bitfolgen zurueckgeben. Wenn die letzte Kombination erreicht ist, soll eine leere Menge zurueckgegeben werden.
Dies wird in der While-Schleife ueberprueft, so dass alle Kombinationen einmal gewaehlt sein werden.
Des Weiteren ist wichtig, dass currentCombination eine Liste darstellen soll, welche ueber den Index aufgrufen wird (so, wie die Bitfolgen).
Die Tabelle „table“ soll hier, wie beschrieben, zeigen von Bitfolgen auf Listen an Mengen. Werte der Tabelle werden dabei ueber eckige Klammern (in welchen der Wert steht) aufgrufen.

Mit Hilfe dieser Tabelle kann man nun alle moeglichen Kombinationen an k Bitfolgen darstellen.
Dazu muss man unterscheiden, ob k eine gerade oder ungerade Zahl ist.

Ist k eine gerade Zahl, so wurden Kombinationen mit $\frac{k}{2}$ Elementen erstellt und man kann durch alle entstandenen Bitfolgen x iterrieren und fuer jede Liste L_x (welche man durch x erhaelt) Folgendes tun:
Man sucht in der Liste nach zwei Mengen, welche disjunkt sind (also kein gleiches Element enthalten).
Hat man zwei Mengen in L_x gefunden, welche disjunkt sind, so hat man ebenfalls k Bitfolgen (bzw. deren Indexe) gefunden, welche die Null-Bitfolge darstellen. Das liegt daran, dass beide Mengen mit der Bitfolge x assoziert sind (bzw., die, zu den Indexen gehoerenden Bitfolge, diese ergeben). Hat man nun zwei disjunkte Mengen, dessen Bitfolgen je x ergeben, so erhaelt man k Indexe an Bitfolgen ($k = 2 * \frac{k}{2}$, weil k hier gerade ist) welche, wenn man sie durch das XOR verbindet, die Null-Bitfolge ergeben, denn das XOR ist selbstinvers.
Der Pseudocode fuer diesen Fall sieht wie folgt aus:

```
1: procedure EVENK(table)
2:     for x ∈ table.values do
3:         L_x ← table[x]
4:         for indexesA ∈ L_x do
5:             for indexesB ∈ L_x do
6:                 if disjoint(indexesA, indexesB) then
7:                     return merge(indexesA, indexesB)
8:                 end if
9:             end for
10:        end for
11:    end for
12:    return NULL
13: end procedure
```

Ist k ungerade, so wurden Kombinationen mit $\frac{k-1}{2}$ Elementen erstellt, so dass man, wuerde man die vorherige Vorgehensweise anwenden, nur $\frac{k-1}{2} * 2 = k - 1$ Bitfolgen erhalten, welche die Null-Bitfolge ergeben.
Somit muss eine zusaetzliche Bitfolge eingefuehrt werden. Die Vorgehensweise fuer ein ungerades k ist diese:
Man iterriert fuer jede der n Bitfolgen b_i durch alle entstandenen x der Kombinationen und wendet das XOR auf b_i und x an. Dadurch entsteht eine weitere Bitfolge $z = b_i \ XOR \ x$.
Fuer diese neue Bitfolge ueberprueft man, ob sie in der Tabelle enthalten ist. Ist dies der Fall, so wird versucht, zwei Mengen aus L_x und L_z zu finden (eine aus L_x und eine aus L_z), fuer welche gilt, dass sie disjunkt sind und je nicht i enthalten.
Hat man solche zwei Mengen gefunden, so hat man $k = 1 + \frac{k-1}{2} * 2$ Bitfolgen gefunden, welche

$$0 = (b_i \ XOR \ x) \ XOR \ z = (b_i \ XOR \ x) \ XOR \ (b_i XOR x)$$

ergeben, wobei 0 fuer die Null-Bitfolge steht.
Der Pseudocode fuer diesen Fall sieht wie folgt aus:

```
1: procedure ODDK(b_1 ... ,n, table)
2:     for i ← 1 to n do
3:         for x ∈ table.values do
4:             z ← b_i XOR x
5:             L_x ← table[x]
6:             L_z ← table[z]
7:             if L_z ≠ NULL then
8:                 for indexesA ∈ L_x do
9:                     for indexesB ∈ L_x do
10:                        if disjoint(indexesA, indexesB, { i }) then
11:                            return merge(indexesA, indexesB, { i })
12:                        end if
13:                    end for
14:                end for
15:            end if
16:        end for
17:    end for
18:    return NULL
19: end procedure
```

In den beiden Pseudocodes sollen die Methoden disjoint ueberpruefen, ob Mengen disjunkt sind und die Methoden merge sollen Mengen verbinden (also vereinigen).
Weiter soll table.values die Werte (bzw. Schluessel) der Tabelle darstellen.

7.1 Laufzeit

Die (worst case) Laufzeit fuer diesen Algorithmus ist ...

1. fuer k gerade

$$\begin{aligned}&\mathcal{O}(\binom{n}{\frac{k}{2}} * c^2 * k)\\ &= \mathcal{O}(\binom{n}{\frac{k}{2}})\\ &= \mathcal{O}(n^{\frac{k}{2}})\end{aligned}$$

2. fuer k ungerade

$$\begin{aligned}&\mathcal{O}(\binom{n}{\frac{k-1}{2}} * n * c^2 * k)\\ &= \mathcal{O}(\binom{n}{\frac{k-1}{2}} * n)\\ &= \mathcal{O}(n^{\frac{k-1}{2}} * n)\\ &= \mathcal{O}(n^{\frac{k-1}{2}+1})\\ &= \mathcal{O}(n^{\frac{k+1}{2}})\end{aligned}$$

Dabei ist c die durchschnittliche Groeße der Listen L_x.
Die Laufzeit setzt sich also je zusammen, aus der Anzahl an Kombinationen und dem Durchlaufen der Listen, um die disjunkten Mengen zu finden. Je Ueberpruefung der Mengen, braucht es k Durchgaenge, um zu ueberpruefen, ob die Mengen disjunkt sind.

Der Faktor $c^2 * k$ wird hier ignoriert, da der Binomialkoeffizient deutlich schneller waechst als der Faktor. Des Weiteren kann der Binomialkoeffizient zu einem exponentiellen Ausdruck umgeformt werden, da folgende Umformung durchgefuehrt werden kann:

$$\begin{aligned}&\binom{n}{k}\\ &= \frac{n * (n-1) * \cdots * (n-k+1)}{k!}\\ &= \frac{1}{k!} * (n * (n-1) * \cdots * (n-k+1))\\ &< \frac{1}{k!} * n^k)\\ &< n^k\end{aligned}$$

Der Algorithmus hat also eine exponentielle Laufzeit und kann somit nur auf kleine n und k angewendet werden.
Doch fuer solche Werte bietet er eine exakte Loesung.

8 Loesung durch ein Gleichungssystem

Fuer das zuvor beschriebene Null-Bitfolge Problem werden nun n Entscheidungsvariablen eingefuehrt: $x_1 \ , \ \dots \ , \ x_n$.
Diese koennen entweder 0 oder 1 annehmen. Ist die Entscheidungsvariable x_i mit $1 \leq i \leq n$ 1, so ist die i-te Bitfolge Teil der Loesung, andernfalls nicht.

Fuer eine gueltige Loesung muss folgendes Gleichungssystem mit m Gleichungen ueber $\mathbb{Z}_2$ geloest werden:

$$a_{1,1} * x_1 + \cdots + a_{n,1} * x_n = 0$$

$$\dots$$

$$a_{1,m} * x_1 + \cdots + a_{n,m} * x_n = 0$$

Dabei stellen

$$a_{i,1} \ , \ \dots \ , \ a_{i,m}$$

die m Bits der i-ten Bitfolge dar.
Die Bits der gegebenen Bitfolgen werden also vertikal untereinander geschrieben und die Bitfolgen horizontal nebeneinander. Dabei erhaelt jede Spalte, also jede Bitfolge, eine Entscheidungsvariable.
Wodurch m (Anzahl der Bits) Reihen und n (Anzahl an Bitfolgen) Spalten entstehen.

Weiter muss

$$x_1 + \cdots + x_n = k$$

in $\mathbb{Z}$ (nicht in $\mathbb{Z}_2$!) gelten.
Dadurch ist gegeben, dass exakt die benoetigte Anzahl an k Bitfolgen gewaehlt wird.

Dass eine Loesung fuer die zuvor beschriebenen Gleichungen ebenfalls eine Loesung fuer das Null-Bitfolge Problem ist, ist einfach zu zeigen:

Seien ohne Einschraenkung der Allgemeinheit $x_1 \ , \ \dots \ , \ x_k$ die k Entscheidungsvariablen, welche 1 annehmen und die Gleichungen erfuellen.
Nun lassen sich die zuvor beschriebenen Gleichungen je auf k Summanden reduzieren, welche zusammen 0 in $\mathbb{Z}_2$ ergeben.
Diese Gleichungen sehen nun wie folgt aus:

$$a_{1,1} + \cdots + a_{k,1} = 0$$

$$\dots$$

$$a_{1,m} + \cdots + a_{k,m} = 0$$

Jede Gleichung ist (wie in Kapitel 2 beschrieben) equivalent zum XOR angewandt auf mehrere Bits:

$$a_{1,1} \oplus \cdots \oplus a_{k,1} = 0$$

$$\dots$$

$$a_{1,m} \oplus \cdots \oplus a_{k,m} = 0$$

Wie in Kapitel 4 beschrieben wurde ist dies wiederrum gleichwertig zum XOR auf Bitfolgen. Hier bei den Bitfolgen 1 bis k, welche zusammen die Null-Bitfolge ergeben.
Somit wurden k Bitfolgen gefunden, welche, verknuepft durch das XOR, die Null-Bitfolge ergeben.

9 Loesung des Gleichungssystems mit bestimmtem Hamming-Gewicht

Nun ist die Aufgabe folgende Gleichungen ueber $\mathbb{Z}_2$:

$$a_{1,1} * x_1 + \cdots + a_{n,1} * x_n = 0$$

$$\dots$$

$$a_{1,m} * x_1 + \cdots + a_{n,m} * x_n = 0$$

zu loesen.
Dieses Gleichungssystem ist aequivalent zu diesem:

$$A * \vec{x} = \vec{0}$$

mit

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{n,1} \\ \vdots & & \vdots \\ a_{1,m} & \cdots & a_{n,m} \end{bmatrix}$$

und

$$\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}.$$

Weiter muss die Gleichung

$$x_1 + \cdots + x_n = k$$

ueber $\mathbb{Z}$ gelten.

Die letzte Beschraenkung ($x_1 + \cdots + x_n = k$) bedeutet, dass die Loesung $\vec{x}$ des ersten Gleichungssystems genau k Einsen beinhalten muss.
Da in $\mathbb{Z}_2$ Eins das einzige Element ungleich Null ist, ist die Beschraenkung aequivalent dazu, dass

$$wt(\vec{x}) = k$$

sein muss.
Wobei $wt(\vec{u})$ das Hamming-Gewicht von $\vec{u}$ ist, welches definiert ist als die Anzahl an Elementen eines Vektors, welche nicht 0 sind, bzw.

$$wt(\vec{u}) = |\{i \in \{1, \ , \ \ldots \ , \ n\} : e_i \neq 0\}|$$

fuer einen Vektor $\vec{u}$ mit den Elementen $e_1, \ , \ \ldots \ , \ e_n$.

Es gilt also eine Loesung $\vec{x}$ des linearen Gleichungssystems

$$A * \vec{x} = \vec{0}$$

zu finden mit

$$wt(\vec{x}) = k,$$

wobei das Gleichungssystem ueber $\mathbb{Z}_2$ ist.

10 Algorithmus 2: Loesung durch Gausz-Verfahren und Bruteforce

In diesem Kapitel wird das zuvor beschriebene Problem angegangen, indem zunaechst das Gleichungssystem in $\mathbb{Z}_2$ geloest und anschlieszend nach einer Loesung der Loesungsmenge gesucht wird, welche die Bedingung des Hamming-Gewichts erfuellt.

10.1 Loesen des Gleichungssystems ueber $\mathbb{Z}_2$

Die zu loesenden Gleichungen lassen sich wie folgt mit Matrix und Vektoren darstellen:

$$A * \vec{x} = \vec{0}$$

Es ist also das homogene System zu A ueber $\mathbb{Z}_2$ zu loesen.

Aufgrund dessen, dass $\mathbb{Z}_2$ ein Koerper ist, laesst sich zum Loesen des Gleichungssystems ueber $\mathbb{Z}_2$ das

Gausz-Verfahren verwenden.
Dadurch erhaelt man eine Loesungsmenge, welche wie folgt aussieht:

$$\{\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \vec{v_1} * x_1 + \cdots + \vec{v_n} * x_n\}$$

Falls x_i mit $1 \leq i \leq n$ keine freie Variable ist, ist $\vec{v_1} = \vec{0}$.

10.2 Loesen der letzten Gleichung durch Bruteforce

Zuletzt ist die Loesung der Loesungsmenge zu finden, welche ein Hamming-Gewicht von k hat.

Dazu kann man bei Loesungsmengen mit kleiner Anzahl an freien Variablen alle moeglichen Kombinationen (ohne Wiederholung/Zuruecklegen) ausprobieren.
Dabei sollte man beachten, dass man niemals mehr als k Variablen auswaehlen darf, da bei jedem Vektor, welcher gewaehlt wird mindestens eine Variable aktiviert wird, welche nicht durch einen anderen Vektor wieder deaktiviert werden kann. Diese Variablen sind die freien Variablen.
Wichtig ist dabei, dass man fuer jede Anzahl i mit $1 \leq i \leq min(k,\ f(A))$ an ausgewaehlten freien Variablen alle Kombinationen (ohne Wiederholung/Zuruecklegen) probiert. Wobei $f(A)$ die Anzahl an freien Variablen der Loesungsmenge des homogenen Gleichungssystems zu A darstellt.
Sei

$$r = min(k,\ f(A)),$$

so muss man durch dieses Verfahren

$$\binom{f(A)}{1} + \cdots + \binom{f(A)}{r}$$

moegliche Belegungen ausprobieren. Denn der untere Wert des Binomialkoeffizienten darf nicht groeszer als der obere Wert sein, weil man (in diesem Fall) nicht mehr freie Variablen auswaehlen kann, als da sind.

Fuer jede Belegung u der freien Variablen kann man nun den resultierenden Vektor $\vec{u}$ berechnen, indem alle durch u ausgewaehlten Vektoren der Loesungsmenge addiert werden. Nun muss nun ueberprueft werden, ob

$$wt(\vec{u}) = k$$

gilt.
Ist dieser Vektor gefunden, so hat man stets eine Loesung des homogenen Systems mit k benutzen Spalten/Vektoren.

10.3 Laufzeit

Dieses Verfahren hat eine Laufzeit von

$$\mathcal{O}(p(A)) + \mathcal{O}(\binom{f(A)}{1} * 1 * n) + \cdots + \mathcal{O}(\binom{f(A)}{r} * r * n).$$

Dabei stellt p ein Polynom dar, welches fuer das Gausz-Verfahren benoetigt wird.
$\binom{f(A)}{1} + \cdots + \binom{f(A)}{r}$ ist erneut die Anzahl an moeglichen Kombinationen. Fuer jede dieser Kombinationen aus q $(1 \leq q \leq r)$ muessen maximal q Vektoren der Loesungsmenge addiert werden, um zu ueberprufen, ob die Kombination eine gueltige Loesung darstellt. Dadurch entsteht eine Laufzeit von maximal $\mathcal{O}(q*n)$ pro Ueberpruefung einer Kombination. Denn die Vektoren haben die Laenge n.

Aufgrund dessen, dass $\mathcal{O}(\binom{n}{k})$ sich, wie zuvor erlaeutert, zu $\mathcal{O}(n^k)$ umformen laesst, ergibt sich

$$\mathcal{O}(p(A)) + \mathcal{O}(f(A)^1 * 1 * n) + \cdots + \mathcal{O}(f(A)^r * r * n).$$

Da der letzte Summand deutlich schneller waechst als die ersten, ist die Laufzeit

$$\mathcal{O}(f(A)^r * r * n).$$

Da $r * n$ hier ein pseudopolynomieller Faktor der exponentiellen Laufzeit ist, kann er ebenfalls ignoriert werden, so dass die finale Laufzeit

$$\mathcal{O}(f(A)^r)$$

ist.

10.4 Anwendungsmoeglichkeit

Dieser Loesungsweg hat also ebenbenfalls eine exponentielle Laufzeit und laesst sich nur anwenden, wenn die Loesungsmenge des homogenen Systems der Matrix klein genug ist, bezueglich der Anzahl an freien Variablen.
Dies laesst sich aber durch das Gausz-Verfahren schnell herrausfinden.
Ist die Loesungsmenge allerdings klein genug, so kann eine exakte Loesung berechnet werden.

Des Weiteren kann dieser Algorithmus auch auf Faelle angewendet, in welchen die Anzahl an freien Variablen zu grosz ist, um alle Moeglichkeiten auszuprobieren.
Fuer diese Faelle kann man die Groesze der Kombinationen an freien Variablen beschraenken. Somit erhaelt man eine polynomielle Laufzeit (z.B. $\mathcal{O}(f(A)^3)$) mit welcher eine Loesung gefunden werden kann, aber nicht muss.

11 Implementierung

Das Programm wurde in Java 8 und objektorentiert geschrieben.
Dazu gibt es eine Klasse „BitSequence“, welche eine Bitfolge darstellt. Sie speichert ein Arary an booleans, welches die Bits darstellt und dessen Laenge als Attribute. Des Weiteren besitzt sie zwei Konstruktoren, welche (1) ein boolean Array und (2) einen String als Uebergabeparameter bekommen. Bei dem Konstruktor, welcher einen String erhaelt, soll dieser String ein binaerer String sein, welcher in ein boolean Array umgewandelt wird. Die BitSequence Klasse hat des Weiteren die Methoden „xor“, „getBit“ und „getSize“, welche je das XOR mit einer weiteren Bitfolge der selben Laenge erzeugen, einen Bit an einem bestimmten Index im boolean Array zurueckgeben und die Laenge der Bitsequence zurueckgeben. Auszerdem implementiert die Klasse das Comparable Interface, wodurch sie vergleichbar ist und sie ueberschreibt die equals und hashCode Methoden, durch Methoden, welche je den Inhalt der Objekte vergleichen und einen HashCode, basierend auf den Bits der Bitfolge, zurueckgeben.

Das ZZZ-Problem wird ebenfalls durch eine Klasse dargestellt, welche die Werte n, k und m, sowie eine Liste der gegebenen Bitfolgen als Attribute besitzt. Diese Klasse („ZZZProblem“) besitzt ebenfalls eine Methode zum Erstellen einer Matrix, wie sie in Kapitel „Loesung des Gleichungssystems mit bestimmtem Hamming-Gewicht“ beschrieben ist.

Fuer das Loesen des homogenen Systems einer Matrix, sowie fuer die Klassen, welche Matrizes und Vektoren darstellen wurde die JLinAlg Java-Bibliothek benutzt. Beim Loesen eines homogenen Systems erzeugt die Java-Bibliothek dabei ein Array aus Vektoren aus $\mathbb{Z}_2^n$ (Vector<F2>[ ]) Des Weiteren ermoeglicht sie, dass Rechnen (auch mit Vektoren und Matrix) im Koerper $\mathbb{Z}_2$.

Eine weitere Wichtigkeit der Umsetzung ist, dass das Programm zwischen den Algorithmen abwaegt, oder keinen waehlt, je nach der Eingabe.
Denn ist die Eingabe zu grosz, wird das Programm aufgrund der exponentiellen Laufzeiten zu lange brauchen.
Genauer wird zunaechst ueberprueft, ob $k \leq 4$ ist. In dem Fall wird Algorithmus 1 benutzt. Ist dies nicht der Fall, wird anschlieszend die Matrix passend zur Eingabe erstellt und das zur Matrix gehoerende homogene System geloest. Fuer diesen Loesungsraum wird dann Algorithmus 2 (bzw. dessen zweiter Teil) benutzt, wobei die maximale Groesze der Kombinationen 3 ist.
Wird dabei keine Loesung gefunden, war das Programm nicht im Stande, die Loesung zu berechnen und gibt ein leeres Array zurueck. In den anderen Faellen, wird ein Array der Indexe der Bitfolgen zurueckgegeben.

12 Aufgabenteil c - Beispiele

Nun folgen die Loesungen des Programms fuer die Eingaben der bwinf website.

In folgender Tabelle ist dargestellt, welcher Algorithmus welche Eingabe loesen kann:

Eingabe	Algorithmus 1	Algorithmus 2
stapel0	Ja	Ja
stapel1	Ja	Ja
stapel2	Nein	Ja
stapel3	Nein	Ja
stapel4	Nein	Nein
stapel5	Ja	Nein

In den folgenen Subsektionen sind die Ergebnisse angegeben. Fuer jede Eingabe sind die Indexe und die wirklichen Bitfolgen aufgelistet. Zusaetzlich ist die benoetigte Zeit in Millisekunden angegeben.

12.1 stapel0

Zeit: 21ms
Indexe: [2, 3, 5, 9, 11]
Bitfolgen:

0011110101011100011010011001 1

1111111000101101000100000011 0111

1101011111101011110110111111 0000

1010110011111101101010001110 0000

1011100001100111000010101011 1110

12.2 stapel1

Zeit: 2ms
Indexe: [1, 2, 4, 6, 7, 9, 11, 14, 15]
Bitfolgen:

0010000011110011111101111011 1100

1101001101011011010100110101 0111

0011010000101010010000111101 0010

1111001110101100100100001011 1110

0011011000011010110101111111 1010

1111011110010001010010000100 1110

0010001110011101101011101110 0011

1100011111101011010000010111 0100

0001000111010011000111110110 0100

12.3 stapel2

Zeit: 32ms
Indexe: [9, 20, 26, 53, 57, 71, 76, 78, 89, 95, 99]
Bitfolgen:
011010111010001101110100011000011100000110001101011000101110111001100110111
101110111001101011011110000111101110111010111111100111,

001010111110001010110101101111001001100000000000011010011001111011001011001
000100011010101101100101011101001000010111000110100001,

101010110000011011000001011111111100110001100111001010110111110110001111111
011111010001111111010000001011011011111111010011011110,

100000000001001001100110010001100000000001010101101001001000010001110101101
101010100101010001011101011001010001100101001001111011,

001010000110000100101110111010110110101110001001001101011110111011110111001
011000010011100101000011010011100010001000100111111100,

110000110001001101110001011001001011010101100110110101011010010000111101000
100010010100001100101010100100011000100011011101000000,

101011111100100100101001111011000100111110000101010011001000011110001000100
100100110100101010111111010110000011111110000000111011,

110111100001010011011111001100001110100110111011110101111101101110110110100
010011011011001110100010000110000010101111001011111111,

011010010010110001010011111111101011000001000101100111010100101011011000100
000011000011000110101101010111101101000001001010010011,

011101100111100011100111100011011011101001010000001000001011000010100011101
010000000110100110000110100101101101001011111011010000,

111011101010111001111011110000111001101111011010101011111000110100011010001 1
000001111010000100011001000000111011000100010111010000

12.4 stapel3

Zeit: 19ms
Indexe: [23, 26, 34, 43, 71, 98, 110, 120, 127, 133, 144]
Bitfolgen:
101101110100101111101100110001010111010000111111000010000011001111111110010
011000111000111100111111010001110101110000111110110101,

101110001011111000101111101010101011001100010000110110011000101101100000011
000011011110101000011001000100011010001100100110011001,

110010110101111111101110100010001001000001011001110101001111101000001001111
100011100010110000000010011101100100111000001100111110,

010100001011011100111100011100110100110011111110000010000001000000101111000
011101000010010011111011110010100111101101111100110111,

101100000010011001101101010001001100111001011001101011110111111010000011101
000110000011000000101011111110000000001011110100100001,

011111011000101011001100111010110101010011010001100111111010101100000001110

0011010100111111111010010000110001001111111000101101100,

10111111010101001100000101101100111101000010100010100100001001111010110100 1
0010110001110110001101010000001101010100111010001011 1,

0111000111111010100001001110001111111110011110110111000101010001011000110 00
10101000010100010100001010000010100001000101110101100,

0111011010011100100001101110010010101011111100101111000000101100110110010 11
0011100101110000001101001001110011110010101101001011 1,

0010000001110011100011010100011111100111100010111101001100101011000100110 001
0011110101100111100111111101111011001111100101010000 1,

1100000101100100011010011101111111101111101101101011111010001011000010111 10
0011100101010010110000011101110101101010110011101010 0

12.5 stapel5

Zeit: 228ms
Indexe: [77, 185, 163, 167, 70]
Bitfolgen:

1101010001001101000111111110000110100010100111000100001001011011

1010000110101100101110111001100011011110111111010111000101111110

1010111011001100100110001100110001011101001000000011011111100100

0101111111000111000000101111100010111010110101000100000011001000

1000010011101010001111100100110110011011100101010100010000001001

13 Aufgabenteil b

In Aufgabenteil b ist gefragt, wie man mithilfe der 11 gefundenen Karten am naechsten Wochenende das naechste Haus aufsperren kann, ohne dafuer mehr als zwei Fehlversuche zu benoetigen.
Diese Aufgabenstellung wird nun mit dem allgemeineren Problem angegangen, dass es n Haeuser gibt, so dass $n+1$ Karten gefunden werden.

Seien zunaechst die Folge

$$W = (w_1,\ \dots\ , w_n)$$

die Karten der in der Aufgabenstellung erwaehnten (aufsteigend sortierten) Codeworte und x die Sicherungskarte.

Weiter seien die gefundenen Karten

$$K = (k_1,\ \dots\ ,\ k_{n+1}).$$

Nun seien die Karten

$$S = (s_1,\ \dots\ ,\ s_{n+1})$$

K aufsteigend sortiert.
W, K und S sollen hier Folgen sein.

Da davon ausgegangen wird, dass die $n+1$ richtigen Karten gefunden wurden, besteht S aus W und x. Wichtig ist dabei, dass W innerhalb von S immer noch dieselbe Reihenfolge hat, da die Karten in beiden Folgen aufsteigend sortiert sind.
Somit wurde ausschlieszlich die zusaetzliche Karte x in die Folge W hinzugefuegt, wodurch S entsteht.

Sei der Index, an welchem x zu W hinzugefuegt wurde, j, wobei $1 \leq j \leq n+1$ gilt, so dass x innerhalb der Folge oder rechts an die Folge hinzugefuegt wird.
Dadurch verschieben sich alle Karten mit einem Index $i \geq j$ um eins nach rechts, so dass eine Karte w_i

nun an Index $i+1$ ist (so fern $i \geq j$). Die restlichen Karten (mit $i < j$) bleiben an ihrer Position.
Die Folge saehe dann wie folgt aus:

$$(w_1, \ \dots \ , \ w_{j-1}, \ x, \ w_j, \ \dots \ , \ w_n)$$

In dieser Folge ist zu erkennen, dass x am Index j liegt. Rechts von q sind die Elemente $e_j, \ \dots \ , \ e_n$, welche nun alle eine Position nach rechts gesetzt wurden (der Index in der Folge hat sich je um 1 erhoeht). Somit hat die Liste nun eine Laenge von $n+1$.

Dementsprechend gibt es fuer jeden Index i eines Codewortes w_i $(1 \leq j \leq n)$ zwei Moeglichkeiten, entweder w_i ist in S an Position i oder an Position $i+1$. Je nachdem, ob $i < j$ bzw. $i \geq j$ ist.
Das bedeutet, dass man fuer das Haus k ebenfalls zwei Moeglichkeiten fuer das Codewort hat, entweder s_k oder s_{k+1}.

Zusammengefasst muss man also die $n+1$ gefundenen Karten sortieren und kann anschlieszend das Haus k mit dem Schluessel an Index k und $k+1$ der sortierten Karten suchen. Dies haelt die Bedingung ein, dass maximal 3 Versuche benoetigt werden duerfen.

Auf die gegebene Aufgabe laesst sich dieses Verfahren einfach uebertragen, indem die 11 gefundenen Karten sortiert werden und fuer jedes Haus k ebenfalls die Karten k und $k+1$ probiert werden.

14 Quellcode

Nun folgt der Quellcode der wichtigsten Teile des Programs.

14.1 Erstellen der Matrix

Diese Methode ist Teil der „ZZZProblem“ Klasse und erstellt aus dem gegebenen ZZZ-Problem eine Matrix, wie sie in Kapitel „Loesung des Gleichungssystems mit bestimmtem Hamming-Gewicht“ beschrieben ist.

```
1     /**
       * Methode to create the matrix corresponding to the problem, where each bit sequences is a column
3      * @return The matrix
       */
5     public Matrix<F2> createMatrix() {
          // Define amount of rows and columns
7         int rows = this.m;
          int columns = this.n;
9
          // Create matrix
11        F2[][] matrixArray = new F2[rows][columns];

13        // Iterate through rows and columns
          for (int row = 0; row < rows; row++) {
15            for (int col = 0; col < columns; col++) {
                  // Get current BS' bit, transform to F2, set in matrix-array
17                boolean val = this.bitSequences.get(col).getBit(row);
                  F2 f2 = JLinAlgUtils.boolToF2Function.apply(val);
19
                  matrixArray[row][col] = f2;
21            }
          }
23
          return new Matrix<>(matrixArray);
25    }
```

14.2 Algorithmus 1

Die folgenden Methoden sind Teil der Klasse, welche den Algorithmus 1 implementiert.

Erstellen der Kombinationen

```java
/**
 * Methode to create all combinations of given size subsets of BitSequences
 * in bitSequences list.
 * @param size The size of the subsets
 * @return A HashMap of BitSequences b pointing to set of all indexes (int-arrays
 * sized size) creating b.
 */
private Map<BitSequence, Set<int[]>> getCombinations(int size) {
    Map<BitSequence, Set<int[]>> combinations = new HashMap<>();

    BitSequence zeroBS = new BitSequence(new boolean[this.problem.m]);
    int[] indexes = new int[size];
    this.getCombinationsImp(combinations, zeroBS, indexes, 0, -1, size);

    return combinations;
}

/**
 * Recursive implementation of getting all combinations
 * @param combinations The HashMap which the results will be added to
 * @param currentBS The current BitSequences created by the indexes
 * @param indexes The indexes used
 * @param indexInArray The current index in indexes
 * @param indexInList The current index in bitSequences list
 * @param k The amount of indexes not choosen yet
 */
public void getCombinationsImp(Map<BitSequence, Set<int[]>> combinations,
                               BitSequence currentBS, int[] indexes, int indexInArray,
                               int indexInList, int k) {
    // Base case: no indexes left to add
    if (k == 0) {
        Set<int[]> combination = combinations.computeIfAbsent(currentBS, k1 -> new HashSet<>());
        combination.add(Arrays.copyOf(indexes, indexes.length));
        return;
    }

    int start = indexInList + 1;
    int end = this.problem.n - k;
    int nextK = k - 1;
    int nextIndexInArray = indexInArray + 1;

    for (int i = start; i < end; i++) {
        indexes[indexInArray] = i;
        BitSequence nexBS = currentBS.xor(this.bitSequences.get(i));
        getCombinationsImp(combinations, nexBS, indexes, nextIndexInArray, i, nextK);
    }
}
```

Finden der Loesung fuer k gerade

```java
/**
 * Implementation of bruteforcing if k+1 is even.
 * @param combinations All combinations of the BitSequences
 * @return The result to the ZZZ-Problem
 */
private int[] findEvenSolution(Map<BitSequence, Set<int[]>> combinations) {
    // Go through all different combinations and the associated set of< indexes.
    // For each set, search for two disjoint arrays in the set and return
    // both (merged) if they could be found.
    for (Set<int[]> set : combinations.values()) {
        int[] disjointArray = this.findDisjointArrays(set);
        if (disjointArray != null) return disjointArray;
    }

    return null;
}

/**
```

```
     * Methode to find two disjoint arrays in a set of int-arrays
     * @param set The set with int-arrays
     * @return Any two disjoint array merged into one array, or null
     * if there is non
     */
    private int[] findDisjointArrays(Set<int[]> set) {
        // Go through all combinations of two int-arrays using
        // double-loop. Check if both current int-arrays are
        // disjoint, if so return them merged.
        for (int[] arr1 : set) {
            for (int[] arr2 : set) {
                if (this.isDisjoint(arr1, arr2)) {
                    return this.mergeArrays(arr1, arr2);
                }
            }
        }

        return null;
    }
```

Finden der Loesung fuer k ungerade

```
    /**
     * Implementation of bruteforcing if k+1 is odd.
     * @param combinations All combinations of the BitSequences
     * @return The result to the ZZZ-Problem
     */
    private int[] findOddSolution(Map<BitSequence, Set<int[]>> combinations) {
        // Go through all BS, for each go through the combinations.
        // XOR current BS with current combination and check for combinations
        // creating the XOR. Now check for any disjoint solutions of all BS.
        for (int i = 0; i < this.problem.n; i++) {
            BitSequence bs1 = this.bitSequences.get(i);
            int[] arrI = new int[]{i};

            for (BitSequence bs2 : combinations.keySet()) {
                Set<int[]> set = combinations.get(bs2);
                BitSequence bs3 = bs1.xor(bs2);

                Set<int[]> set2 = combinations.get(bs3);
                if (set2 == null) continue;

                int[] disjointArray = this.findDisjointArrays(set, set2, arrI);
                if (disjointArray != null) return disjointArray;
            }
        }
        return null;
    }

    /**
     * Methode to find two int-arrays int two sets of int-arrays which are
     * disjoint and don't contain any element of a third given int-array
     * @param set1 First set of int-arrays
     * @param set2 Second set of int-arrays
     * @param arr The third int-array
     * @return Any two matching arrays merged into one array with the third one,
     * or null if there is non
     */
    private int[] findDisjointArrays(Set<int[]> set1, Set<int[]> set2, int[] arr) {
        // Go through all combinations of two int-arrays using
        // double-loop. Check if both current int-arrays and the third array
        // are disjoint, if so return them merged.
        for (int[] arr1 : set1) {
            for (int[] arr2 : set2) {
                if (this.isDisjoint(arr1, arr2, arr)) {
                    return this.mergeArrays(arr1, arr2, arr);
                }
            }
        }

        return null;
```

```
}
```

14.3 Algorithmus 2

Loesen des homogenen Systems
Diese Methode loest das homogene System einer Matrix mit Hilfe der JLinAlg Java-Bibliothek und loest somit den ersten Teil des zweiten Algorithmus.

```
private Vector<F2>[] solveHomogenousSystem(Matrix<F2> matrix) {
    // Create zero vector
    Vector<F2> bVec = JLinAlgUtils.getZeroVector(matrix.getRows());

    // Solve Ax = 0 for solution space of x
    return LinSysSolver.solutionSpace(matrix, bVec).getGeneratingSystem();
}
```

Bruteforcing der richtigen Loesung der Loesungsmenge
Diese Methoden loesen den zweiten Teil des zweiten Algorithmus, indem sie die Loesung mit dem Hamming-Gewicht „weight“ in einer gegebenen Loesungsmenge suchen.
Diese Loesungsmenge ist gegebem als ein Array aus Vektoren aus $\mathbb{Z}_2^n$ (Vector<F2>[ ]).
Die erwaehnte maximale Groesze der Kombinationen an freinen Variablen ist hier angegeben als „max-Depth“.

```
/**
 * Methode to bruteforce for the solution with given hamming weight
 * @return The correct solution
 */
private Vector<F2> bruteforce(int maxDepth) {
    // Try out all any amount of free vectors to choose from 1 to weight (inclusive)
    for (int i = 1; i <= this.weight; i++) {
        // Stop searching if current amount of free variables is bigger then the
        // amount of free variables
        if (i > this.freeVariables) break;

        // Also stop searching if i is bigger then maxDepth
        if (i > maxDepth) break;

        int[] choosenVectors = new int[this.freeVariables];
        Vector<F2> solution = this.bruteforceImp(choosenVectors, 0, -1, i);

        if (solution != null)
            return solution;
    }

    return null;
}

/**
 * Actual methode to bruteforce for the solution using a recursive methode
 * @param values Currently chosen free variables as their indexes
 * @param indexInArray Current index in values array
 * @param indexInSpan Current index in span (aka. solution set, aka. free variables)
 * @param k Amount of not chosen free variables
 * @return The resulting vector
 */
private Vector<F2> bruteforceImp(int[] values, int indexInArray, int indexInSpan, int k) {
    // Base case: No elements left to add in values array
    if (k == 0) {
        // Create solution with current values array and check for needed
        // hamming weight.
        Vector<F2> solution = this.getSolutionVector(values);
        if (HammingUtils.getHammingWeight(solution, this.rows) == this.weight)
            return solution;

        return null;
```

```
43          }
45          int start = indexInSpan + 1; // Inclusive
            int end = this.freeVariables - k; // Inclusive
47
            for (int i = start; i <= end; i++) {
49              values[indexInArray] = i;
51              Vector<F2> solution = this.bruteforceImp(values, indexInArray + 1, i, k - 1);
                if (solution != null) return solution;
53          }
55          return null;
        }
```

15 Literatur

1. Elwyn R. Berlekamp, Robert J. McEliece and Henk C.A. van Tilborg, On the inherent intractability of certain coding problems, 1978
2. Rod G. Downey, Michael R. Fellows, Alexander Vardy and Geoff Whittle, The parametrized complexity of some fundamental problems in coding theory, 1999