Bonusaufgabe: Zara Zackigs Zurückkehr

Teilnahme-ID: 63421

Bearbeiter/-in dieser Aufgabe:
Niels Glodny

25. April 2022

Inhaltsverzeichnis

1 Lösungsidee

1.1 Modellierung

Gegeben sind n Karten, die aus jeweils m Bits bestehen. Das exklusive Oder von k dieser Karten soll eine weitere Karte ergeben. Da das exklusive Oder von einem Wert mit sich selbst immer null ist, kann äquivalent nach $k+1$ Karten gesucht werden, deren exklusives Oder null ist. Die Operation XOR von Binärwörtern der Länge m entspricht der Addition von Vektoren aus $\mathbb{F}_2^m$, also dem m-dimensionalen Vektorraum über den Körper der Restklassen modulo 2. Da es sich bei $\mathbb{F}_2$ um einen Körper handelt, können viele der üblichen Rechnungen, im Besonderen die der linearen Algebra, wie üblich durchgeführt werden. Um die gängige Terminologie aus der Kodierungstheorie verwenden zu können, werden die Karten als Spalten einer $m \times n-$Matrix $\mathbf{H}$ dargestellt. Gegenüber den Eingabedateien ist diese Matrix also gespiegelt. Das Problem besteht nun daraus, eine Kombination von $t = k+1$ Spalten aus $\mathbf{H}$ zu finden, die sich (in $\mathbb{F}_2^m$) zu $\mathbf{0}$ addieren. Das entspricht dem Finden von einem $x \in \mathbb{F}_2^n$, für das

$$x\mathbf{H} = \mathbf{0} \qquad \text{und} \qquad wt(x) = t$$

gilt. $wt(x)$ bezeichnet dabei das Hamming-Gewicht des Vektors, also die Anzahl der von null verschiedenen Stellen. Dieses Problem wird als *Minimales-Codewort-Problem* (eng. minimum codeword problem) bezeichnet und wurde bereits eingehend untersucht, unter anderem, da es eine zentrale Rolle für die Sicherheit des McEliece-Kryptosystems spielt.

In diesem Kontext wird $\mathbf{H}$ als Kontrollmatrix (eng. parity check matrix) bezeichnet. Alle Vektoren $x \in \mathbb{F}_2^n$ für die $x\mathbf{H} = \mathbf{0}$ gilt, werden als Codewörter bezeichnet und bilden zusammen einen linearen Blockcode. Das Produkt eines beliebigen Vektors $z \in \mathbb{F}_2^m$ mit $\mathbf{H}$ wird als Syndrom von z bezeichnet. Ein Codewort ist also ein Vektor, dessen Syndrom der Nullvektor ist.

Berlekamp, McEliece und Tilborg [1] haben gezeigt, dass dieses Problem zur Klasse der NP-vollständigen Probleme gehört, indem sie das 3-dimensionale Matchingproblem darauf reduziert haben, welches wiederum eins von Karps 21 NP-vollständigen Problemen ist. Folglich gilt es als sehr unwahrscheinlich, dass es einen Algorithmus gibt, der das Problem in polynomieller Laufzeit löst. Im Folgenden sind zwei mögliche Lösungsansätze dargestellt. Der zweite ist ein Brute-Force-Ansatz, bei dem das Problem zunächst mithilfe von linearer Algebra vereinfacht und dann durch Ausprobieren aller restlichen Möglichkeiten gelöst wird. Dieses Verfahren ist sehr effektiv, wenn die Karten viele Bits haben und diese Bits linear unabhängig sind, da dann nur sehr wenige Stellen zum Ausprobieren übrig bleiben. Haben die Karten aber wenig Bits oder gibt es sehr viele Karten, kommt der Brute-Force-Ansatz schnell an seine Grenzen.

Der erste Ansatz basiert auf dem *Information-Set-Decoding* und ist in den meisten Fällen sehr viel effizienter, wenn auch immer noch exponentiell in der Laufzeit. Es handelt sich um einen Las-Vegas-Algorithmus, was heißt, dass die Laufzeit nicht deterministisch ist und der Algorithmus auch nicht verwendet werden kann, um mit Sicherheit zu zeigen, dass es keine Lösung gibt. Information-Set-Decoding-Algorithmen sind die besten heute bekannten Algorithmen zum Lösen des Minimalen-Codewort-Problems.

1.2 Lösungsansatz 1: Information-Set-Decoding

Da das Problem eine zentrale Rolle für die Sicherheit von mehreren Post-Quantum-Kryptosystemen spielt, wurden verschiedene Algorithmen entwickelt, die in der Praxis deutlich effizienter als ein Brute-Force-Ansatz sind. Diese basieren auf dem sogenannten Information-Set-Decoding (ISD). Es handelt sich um randomisierte Las-Vegas-Algorithmen, das heißt, dass sie zwar immer das richtige Ergebnis zurückgegeben, die Laufzeit aber variiert. Im Kern der Algorithmen steht eine Prozedur, die mit einer Wahrscheinlichkeit p eine Lösung findet. Findet sie keine Lösung, wird sie so lange wiederholt, bis sie eine Lösung gefunden hat.

Die Ursprungsform des ISD geht auf Prange [4] zurück. Später wurde der Algorithmus unter anderem von Lee und Brickell [3] und Stern [5] verbessert. Bei allen Varianten gibt es eine Abwägung zwischen der Laufzeit pro Versuch, der Wahrscheinlichkeit, dass der Versucht glückt und der Komplexität des Algorithmus. Der Algorithmus von Lee und Brickell verwendet beispielsweise mehr Zeit pro Versuch als der von Prange, hat dafür aber eine deutlich höhere Erfolgswahrscheinlichkeit. Zur Lösung des Problems wurde der Algorithmus von Lee und Brickell verwendet, da er für die gegebenen Beispieldaten völlig ausreichend und nicht unnötig komplex ist. Das vereinfacht auch die theoretische Analyse. Für jeden Versuch wird

Algorithmus 1 Finden der Umlegungen

procedure LeeBrickellISD($\mathbf{H}_0, t$)
  while True **do**
    generiere zufällige Permutation π
    $\mathbf{H} \leftarrow \mathbf{H}_0$, mit den Spalten nach π permutiert
    $\mathbf{H}$ in reduzierte Stufenform (eng. reduced row-echelon form) bringen
    sei I die Menge aller Spalten, die die erste Eins einer Zeile enthalten
    for $x \leftarrow p$-große Teilmengen von Spalten $\notin I$ **do**
      sei s die Summe der Spalten x
      if $wt(s) == t - p$ **then**
        $e_m = \begin{cases} 1, \text{wenn } m \in x \text{ oder } (s(\mathbf{H}_I^{-1}))_m = 1 \\ 0, \text{andernfalls} \end{cases}$
        permutiere e mit π^{-1}
        return e
      end if
    end for
  end while
end procedure

eine zufällige Permutation generiert und die Spalten von $\mathbf{H}$ entsprechend permutiert. Daraufhin wird $\mathbf{H}$ in reduzierte Stufenform gebracht, beispielsweise mit dem Gauß-Jordan-Algorithmus. Die Matrix wird nun nach Spalten in zwei Teile geteilt: Die Submatrix $\mathbf{H}_I$ enthält alle Spalten, die die erste Eins einer

Zeile haben, und ist folglich invertierbar und eine Submatrix aus den restlichen Spalten. Da die Spalten am Anfang zufällig permutiert wurden, wird so eine in jedem Durchlauf zufällige Auswahl von Spalten gefunden, die eine invertierbare Submatrix bilden. Nun werden alle Summen von p-großen Teilmengen der Spalten $\notin I$ berechnet. Wenn im gesuchten Wort in dieser zufälligen Permutation genau p Einsen in den Positionen $\notin I$ sind, können die restlichen Einsen in Spalten $\in I$ über lineare Algebra direkt bestimmt werden. Das ist genau dann der Fall, wenn das Hamming-Gewicht der Summe genau $t - p$ entspricht, denn zusammen mit den p Einsen aus dem Teil $\notin I$ soll das Wort ein Gewicht von t haben. Schließlich wird das Wort rekonstruiert und zurückgegeben. p ist ein Parameter des Algorithmus und sollte eine relativ kleine Zahl sein, da die Laufzeit pro Versuch exponentiell mit p steigt. In den meisten Fällen erhöht ein größeres p aber die Erfolgswahrscheinlichkeit pro Versuch, da mehr Einsen in den Stellen $\notin I$ zugelassen werden. Für $p = 0$ ergibt sich der ISD-Algorithmus nach Prange.

1.2.1 Laufzeit- und Speicherkomplexität

Um die Analyse zu vereinfachen, wird davon ausgegangen, dass alle Zeilen von $\mathbf{H}$ linear unabhängig sind. Um dies sicherzustellen, können vor dem Ausführen des Algorithmus die linear abhängigen Zeilen mit dem Gauß-Verfahren entfernt und m entsprechend angepasst werden. Zunächst wird die Laufzeit- und Speicherkomplexität von einem einzigen Durchlauf analysiert. Die zufällige Permutation π kann beispielsweise mit dem Fisher-Yates-Algorithmus in $\Theta(n)$ generiert werden, das Permutieren der Matrix benötigt dann aber $\Theta(nm)$ Laufzeit, da n Spalten kopiert werden müssen, die jeweils aus m Elementen bestehen. Daraufhin muss $\mathbf{H}$ in reduzierte Stufenform gebracht werden, beispielsweise mit dem Gauß-Jordan-Algorithmus. Eine detaillierte Beschreibung des Gauß-Jordan-Algorithmus und der dazugehörigen Laufzeitanalyse würde hier den Rahmen sprengen. Die implementierte Version des Algorithmus hat eine Laufzeit von $\Theta(n^2 m)$. Die Menge I enthält folglich genau m Spalten. m bezieht sich hier, wie oben beschrieben, auf die Anzahl der linear unabhängigen Zeilen in $\mathbf{H}$. Für das Aufteilen in I muss in jeder Zeile nach der ersten Eins gesucht werden, was eine Laufzeit von $O(nm)$ hat. Die for-Schleife in Zeile 7 benötigt $\binom{n-m}{p}$ Durchläufe. Das Berechnen der Summe der Spalten, deren Hamming-Gewicht und das Überprüfen der if-Abfrage benötigt $\Theta(pm)$ Laufzeit. Ohne die Zeilen 10, 11 und 12 hat die for-Schleife also eine Laufzeit von $\binom{n-m}{p}\Theta(pm)$. Die Zeilen 10, 11 und 12 werden pro Aufruf der Prozedur nur ein einziges Mal ausgeführt. Werden die in Zeile 6 gefundenen Positionen gespeichert und wiederverwendet, ist die Berechnung von e in $\Theta(pm)$ möglich. Insgesamt ergibt sich pro Iteration eine Laufzeit von

$$\begin{aligned} T(n,m) &= \Theta(n) + O(nm) + \Theta(n^2 m) + \binom{n-m}{p} \cdot \Theta(pm) + \Theta(pm) \\ &= \Theta(n^2 m) + \binom{n-m}{p} \cdot \Theta(pm) \end{aligned}$$

Es zeigt sich also, das die Laufzeit für eine einzige Iteration exponentiell mit p und polynomiell mit n und m steigt. Wie bereits erwähnt sollte also für den Parameter p eine relativ kleine Zahl gewählt werden. Für die erwartete Gesamtlaufzeit ist aber nicht nur die Laufzeit einer Iteration wichtig, sondern auch die Erfolgswahrscheinlichkeit.

Die t Einsen könnten in n verschiedenen Positionen auftreten, sodass es $\binom{n}{t}$ Möglichkeiten für e gibt. Der Algorithmus ist immer dann erfolgreich, wenn genau p der Einsen in $n - m$ Spalten und $t - p$ der Einsen in den m Spalten $\in I$ sind. Durch die zufällige Permutation sind die Einsen gleichmäßig zufällig auf die Spalten verteilt. Folglich gilt nach dem Lottomodell für die Erfolgswahrscheinlichkeit eines einzigen Versuchs

$$P(\text{Erfolg}) = \frac{\binom{n-m}{p} \cdot \binom{m}{t-p}}{\binom{n}{t}}.$$

Der Erwartungswert für die Anzahl der Versuche, bis ein Ereignis X der Wahrscheinlichkeit $P(X)$ auftritt, ist $E(x) = \frac{1}{P(X)}$. Folglich gilt, wenn X die Anzahl der Durchläufe ist,

$$E(X) = \frac{\binom{n}{t}}{\binom{n-m}{p} \cdot \binom{m}{t-p}}.$$

In der Praxis ist dieser Wert gerade für Beispiele, in denen t im Vergleich zu $n - m$ relativ klein ist, ebenfalls vergleichsweise gering. Im Folgenden ist $E(X)$ für verschiedene p für die Beispiele von der BWINF-Website aufgeführt.

Beispiel	$E(X)$, $p=0$	$E(X)$, $p=1$	$E(X)$, $p=2$
stapel1.txt	1.33	4.00	
stapel1.txt	1.82	2.22	
stapel2.txt	1.11	10.09	
stapel3.txt	13.68	4.45	3.31
stapel4.txt	51.58	10.44	4.78
stapel5.txt	332.57	29.34	6.63

Es ist zu erkennen, dass meistens schon wenige Durchläufe genügen, um eine Lösung zu finden. Für die Implementation wurde $p = 1$ verwendet, da die Erfolgswahrscheinlichkeit auch für größere Instanzen ausreicht und die Implementation dadurch vereinfacht wird. Es wäre natürlich auch denkbar, p an die Dimensionen der Eingabedaten und die Geschwindigkeit des Computers dynamisch anzupassen.

Der Erwartungswert für die Laufzeit liegt damit bei

$$E(T) = \frac{\binom{n}{t}}{\binom{n-m}{p} \cdot \binom{m}{t-p}} \cdot \left(\Theta(n^2 m) + \binom{n-m}{p} \cdot \Theta(pm) \right)$$

Die Speicherkomplexität beträgt $\Theta(nm)$, da die $m \times n$-Matrix einmal kopiert werden muss, um die Permutation anzuwenden. Theoretisch wäre diese Kopie durch ein effizienteres Verfahren zur Anwendung der Permutation vermeidbar, der Speicherbedarf wird aber vermutlich keine Einschränkung für den Algorithmus darstellen, da die Laufzeiten bei derartig großen Instanzen nicht mehr vertretbar wären. Neben der Kopie der Matrix werden die Permutation und die Positionen der Pivots des Gauß-Algorithmus gespeichert, was zusätzlich $\Theta(n)$ Speichereinheiten benötigt.

1.3 Lösungsansatz 2: Brute-Force

Die Idee des Brute-Force-Ansatzes ist es, die Matrix zunächst über das Gauß-Verfahren in Stufenform zu bringen. Daraufhin muss substituiert werden, um zur reduzierten Stufenform zu gelangen. Da das Gleichungssystem $x\mathbf{H} = \mathbf{0}$ aber immer unterbestimmt ist, damit es überhaupt eine Lösung mit $x \neq 0$ gibt, gibt es freie Variablen. In diesem Fall wird rekursiv versucht, zuerst die 0 und dann die 1 an dieser Stelle zu substituieren. Hat das so gefundene Wort die richtige Anzahl an Einsen, wird es als Lösung ausgegeben, andernfalls werden per Backtracking andere Werte für die freien Variablen eingesetzt. Das Verfahren kann unter anderem dadurch etwas beschleunigt werden, dass abgebrochen wird, wenn bereits die maximale Anzahl an Einsen verwendet wurde oder es nicht mehr genug Variablen gibt, um die nötige Anzahl an Einsen zu erreichen.

Es folgt eine grobe Abschätzung der Laufzeit dieses Verfahrens. Die Anzahl der freien Variable, für die also per Backtracking potenziell beide Möglichkeiten ausprobiert werden müssen, ist $n - m$. Pro rekursivem Aufruf wird im worst case $O(nm)$ Zeit für die Substitution verwendet. Zusammen mit dem Gauß-Verfahren, das zuvor ausgeführt wird, um zur Stufenform zu gelangen, ergibt sich eine Laufzeit von

$$T(n, m) = \Theta(n^2 m) + O(2^{n-m}).$$

Das ist bei Instanzen, bei denen n und m relativ nah beieinander liegen bereits deutlich besser als ein reines Brute-Force, bei dem alle Kombinationen von Karten ausprobiert werden würden, was eine Laufzeit von $O(2^n)$ benötigen würde. Die Laufzeit ist dennoch in der Praxis meist deutlich langsamer als die des Information-Set-Decoding. Das Verfahren hat aber den Vorteil, dass es vollständig deterministisch ist und es auch mit Sicherheit bestätigen kann, dass es keine Lösung gibt, wenn das der Fall ist. Wenn n und m in etwa gleich groß sind, kann dieses Verfahren aufgrund des dann relativ kleinen Exponenten effizienter als ISD sein.

1.4 Erweiterung: Generierung von größeren Beispielen

Beide Ansätze können die Beispiele von der BWINF-Website in angemessener Zeit lösen. Der ISD-Algorithmus ist aber in der Lage, auch deutlich größere Beispiele zu lösen. Daher wurde die Aufgabe so erweitert, dass auch Beispiele von 2-4-facher Größe gelöst werden sollen. Mit einem Brute-Force-Ansatz sind derart große Beispiele dann nicht mehr lösbar. Zusätzlich müssen größere Beispieldateien generiert werden. Hierfür wäre es denkbar, ähnlich wie im McEliece-Kryptosystem einen linearen Code mit bestimmten

minimalen Distanz zu generieren. So könnte man sich sicher sein, dass es nur die eine Menge an Karten gibt, die im exklusiven Oder null ergeben. Allerdings können auch einfach zufällig generierte Karten verwendet werden, da es bei zufällig ausgewählten Vektoren mit ausreichend Bits äußerst unwahrscheinlich ist, dass es eine zweite Lösung gibt. Danach werden k Karten ausgewählt und eine weitere Karte durch das exklusive Oder der k Karten ausgetauscht, die in der Aufgabenstellung beschrieben.

1.5 Teilaufgabe b)

Nachdem Zara die $k+1$ richtigen Karten gefunden hat, möchte sie Haus h aufschließen. Zunächst sortiert sie die Karten. Dann probiert sie, das Haus mit der h-ten Karte aufzuschließen. Funktioniert dies nicht, hat die Sicherheitskarte einen Index $\leq h$. Dann kann sie mit der $h+1$-ten Karte das Haus aufschließen, da diese die h-te Karte wäre, wenn die Sicherheitskarte entfernt würde.

2 Umsetzung

Die Lösung wurde in `C++` implementiert. Die beiden Ansätze wurden in jeweils einer Klasse, `ISDSolver` und `BruteforceSolver` implementiert. Eine `Utils`-Klasse enthält Methoden, die von beiden Algorithmen verwendet werden, beispielsweise eine Implementation des Gauß-Jordan-Algorithmus über $\mathbb{F}_2$.

2.1 Information-Set-Decoding

Zunächst wird die Implementation des ISD-Algorithmus von Lee und Brickell erläutert. Vor dem eigentlichen Algorithmus wird, wie bereits kurz erwähnt, einmal der Gauß-Jordan-Algorithmus auf der Matrix ausgeführt. Das führt dazu, dass linear abhängige Zeilen entfernt werden und die Implementation vereinfacht wird, da die Matrix nun immer mindestens so viele Spalten wie Zeilen hat. Bei jedem Versuch wird eine zufällige Permutation für die Spalten über die `std::shuffle` Methode der `C++`-Standardbibliothek generiert und die Matrix entsprechend dieser Permutation kopiert. Danach wird der Gauß-Jordan-Algorithmus ausgeführt, die Pivots gesucht und deren Zeilen und Spalten gespeichert. Hier wird der Gauß-Jordan-Algorithmus sehr häufig, nämlich in jedem Versuch, ausgeführt und ist damit der Flaschenhals für die Laufzeit. Daher wird eine effizientere Version des Algorithmus verwendet, die statt Vektoren aus der Standardbibliothek `dynamic_bitset`s aus der Boost-Bibliothek verwendet. Außerdem wird das Entfernen von null-Zeilen ausgelassen, da bereits zu Beginn die linear abhängigen Zeilen entfernt wurden und statt `std::swap` wird der XOR-Swap-Algorithmus verwendet, da die Speicherverwaltung für die temporäre Variable in `std::swap` viel Zeit benötigt.

Da $p = 1$ verwendet wird, kann nun einfach über alle Spalten iteriert werden, in denen keine Pivots gefunden wurden und deren Hamming-Gewicht berechnet werden. Hat die Spalte das richtige Gewicht, können über die gespeicherten Pivot-Positionen die notwendigen Bits in e gefunden werden, ohne wirklich das Syndrom mit dem Inversen der Submatrix $\mathbf{H}_I$ zu multiplizieren. Schlussendlich wird die Permutation rückgängig gemacht und das Ergebnis ausgegeben.

Aufgrund seiner Struktur ist dieser Algorithmus sehr einfach zu parallelisieren: alle Kerne eines Prozessors können den Algorithmus unabhängig voneinander ausführen, bis ein Thread eine Lösung gefunden hat. Um dieses Potenzial auch auszuschöpfen wurde OpenMP verwendet, um die Hauptschleife mehrfach gleichzeitig auszuführen.

2.2 Brute-Force-Ansatz

Der Brute-Force-Algorithmus ist im Allgemeinen ineffizienter und musste stark optimiert werden, um alle Beispiele von der BWINF-Website in annehmbarer Zeit lösen zu können. Dafür werden in der `vorberechnen()`-Funktion einige Dinge vorberechnet, sodass beim eigentlichen Lösen weniger in der Matrix gesucht werden muss. Konkret wird abgespeichert, welche Spalten Pivotelemente haben, da die Werte für diese Spalten durch Resubstitution ermittelt werden können, statt 0 und 1 auszuprobieren. Für die Berechnung ist außerdem wichtig, in welcher Zeile das jeweilige Pivotelement ist. Daraufhin werden alle Indizes von Einsen in den Zeilen und Spalten vorberechnet, um effizient über diese Positionen iterieren zu können. Zuletzt werden alle Zeilen und Spalten in `std::bitsets` gespeichert, ebenfalls um effizienter mit ihnen rechnen zu können. Da `std::bitsets` eine zur Compilezeit konstante Anzahl an Bits benötigen, braucht auch die `BruteforceSolver`-Klasse einen zur Compilezeit konstanten Template-Parameter, der die Anzahl der Bits, mit der gerechnet werden soll, angibt. Zur Laufzeit wird dann je nach Größe der Eingabedatei zwischen 256, 1024 und 8129 Bits ausgewählt. Die `dynamic_bitsets` aus der Boost-Bibliothek benötigen

viel Laufzeit zur dynamischen Zuweisung ihres Speicherplatzes, weshalb es in diesem Fall effizienter ist, verschiedene Größen von statischen `std::bitset`s zu verwenden. Das eigentliche Lösen wurde dann rekursiv in der Methode `resub` implementiert. Die Methode bekommt als Parameter eine Spalte, ab der substituiert werden soll, das bisherige Codewort, die Anzahl der Einsen, die noch gesetzt werden können und das Syndrom, also die Summe der Spalten, die im aktuellen Codewort eine Eins haben. Die letzteren beiden Argumente könnten auch bei jedem Methodenaufruf neu berechnet werden, werden aber aus Performancegründen übergeben. Daraufhin wird so lange substituiert, bis eine Spalte erreicht wird, die kein Pivotelement enthält, wo also sowohl 0 als auch 1 möglich wären. Die Suche kann optimiert werden, indem abgebrochen wird, wenn es nicht mehr genug Spalten gibt, um das Zielgewicht zu erreichen oder das Zielgewicht bereits überschritten wurde. Schlussendlich wird nacheinander 0 und 1 für die freie Variable eingesetzt und die Methode rekursiv für die nächste Variable aufgerufen.

Da der Brute-Force-Ansatz mitunter sehr lange brauchen kann, ist es wichtig, dass der Benutzer Feedback über den Fortschritt bekommt. Es gestaltet sich aber als durchaus schwierig, den aktuellen Fortschritt zu berechnen. Daher werden die Effekte der beiden Abbruchbedingungen vernachlässigt. Dann kann aus dem aktuellen Codewort der Fortschritt ermittelt werden. In der Praxis ist dieser Fortschritt am Anfang zu pessimistisch, reicht aber aus, um den Nutzer eine grobe Idee zu geben, wie lange der Algorithmus noch brauchen wird. Außerdem bricht der Algorithmus ab, wenn er eine Lösung gefunden hat, der Fortschritt bezieht sich jedoch auf die Zeit, die benötigt wird, um alle Möglichkeiten auszuprobieren. Das heißt, dass der Algorithmus meist schneller terminiert, als man es nach dem Fortschritt erwarten würde – dies zu verhindern würde aber keinen Sinn ergeben, da der Algorithmus natürlich noch nicht weiß, wann er eine Lösung finden wird. Der Code zum Ausgeben des Fortschrittes ist in `printProgress()` zu finden.

Zum Compilieren des Codes wird ein `C++-14`-fähiger Compiler benötigt, der zusätzlich OpenMP unterstützt. Als Build-System wird CMake verwendet. Der Code kann wie üblich über

```
cmake -S . -B build -D CMAKE_BUILD_TYPE=Release
cmake --build build
```

kompiliert und dann mit `./stapel [Algorithmus] [Pfad]` ausgeführt werden, wobei zwischen den Algorithmen `isd` für Information-Set-Decoding und `bruteforce` für den Brute-Force-Ansatz ausgewählt werden kann. Der Pfad sollte auf die zu lösende Eingabedatei zeigen.

Der Beispielgenerator wurde in Python implementiert und kann unter Linux oder MacOS über `python3 beispielgenerator.py` ausgeführt werden.

3 Beispiele

stapel0.txt

Zunächst die Ausgabe des Information-Set-Decoding-Algorithmus:

```
$ ./stapel isd ../Eingabe/stapel0.txt
Lösungsverfahren: Information-Set-Decoding
Die richtigen Karten wurden gefunden:
 Karte   2 (00111101010111000110100110011001)
 Karte   3 (11111110001011010001000000110111)
 Karte   5 (11010111111010111101101111110000)
 Karte   9 (10101100111111011010100011100000)
 Karte  11 (10111000011001110000101010111110)

Das Lösen hat 5 ms gedauert.
Der Information-Set-Decoding-Algorithmus hat 1 Versuche benötigt.
```

Ausgaben/stapel0_isd.txt

Und die Ausgabe des Bruteforce-Algorithmus:

```
$ ./stapel bruteforce ../Eingabe/stapel0.txt
Lösungsverfahren: Brute-Force
Die richtigen Karten wurden gefunden:
 Karte   2 (00111101010111000110100110011001)
 Karte   3 (11111110001011010001000000110111)
 Karte   5 (11010111111010111101101111110000)
 Karte   9 (10101100111111011010100011100000)
 Karte  11 (10111000011001110000101010111110)
```

```
9
Das Lösen hat 0 ms gedauert.
```

Ausgaben/stapel0_bruteforce.txt

Beide Algorithmen lösen dieses Beispiel also korrekt und in sehr kurzer Laufzeit.

stapel1.txt

Auch für dieses Beispiel kommt der ISD-Algorithmus zur richtigen Lösung:

```
$ ./stapel isd ../Eingabe/stapel1.txt
Lösungsverfahren: Information-Set-Decoding
Die richtigen Karten wurden gefunden:
 Karte   1 (00100000111100111110111101111100)
 Karte   2 (11010011010111011010011010010111)
 Karte   4 (00110100001010100100001111010010)
 Karte   6 (11110011101011001001000010111110)
 Karte   7 (00110110000110101101011111111010)
 Karte   9 (11110111100100010100100001001110)
 Karte  11 (00100011100111011010111011100011)
 Karte  14 (11000111111010110100000101110100)
 Karte  15 (00010001110100110001111101100100)

Das Lösen hat 6 ms gedauert.
Der Information-Set-Decoding-Algorithmus hat 1 Versuche benötigt.
```

Ausgaben/stapel1_isd.txt

Der Brute-Force-Algorithmus kommt zum gleichen Ergebnis und benötigt dafür weniger als eine Millisekunde. Aus Platzgründen wird die Ausgabe hier nicht abgedruckt, ist aber im Zip-Ordner enthalten.

stapel2.txt

Die Ausgabe ist auch für das dritte Beispiel korrekt:

```
$ ./stapel isd ../Eingabe/stapel2.txt
Lösungsverfahren: Information-Set-Decoding
Die richtigen Karten wurden gefunden:
 Karte   9 (011010111010001101110100011000011100000110001101011000101110110...)
 Karte  20 (001010111110001010110101101111100100110000000000001101001100111101...)
 Karte  26 (101010110000011011000001011111111100110001100111001010110111111101...)
 Karte  53 (100000000001001001100110010001100000000001010101101001001000010100...)
 Karte  57 (001010000110000100101110111010110111010111000100100110101111011101...)
 Karte  71 (110000110001001101110001011001001011010101100110110101011010100100...)
 Karte  76 (101011111100100100101001111011000100111110000010101001100100000111...)
 Karte  78 (110111100001010011011111001100001110100110111011110101111101101011...)
 Karte  89 (011010010010110001010011111111101011000001000101100111010100101010...)
 Karte  95 (011101100111100011100111100011011011101001010000001000001011000000...)
 Karte  99 (111011101010111001111101111000111001101111011010101011111100011010...)

Das Lösen hat 13 ms gedauert.
Der Information-Set-Decoding-Algorithmus hat 2 Versuche benötigt.
```

Ausgaben/stapel2_isd.txt

Bei den größeren Beispielen werden nicht alle Bits der Karten angezeigt, da diese nicht mehr in ein Terminalfenster passen würden. Die restlichen Bits können über die 0-indexierten Kartennummern aus der Eingabedatei entnommen werden. Ohnehin ist Zara vermutlich eher an den Kartennummern interessiert.

Auch dieses Beispiel löst der Brute-Force-Algorithmus korrekt in unter einer Millisekunde. Bei den ersten drei Beispielen zeigt sich, dass in manchen Fällen auch der Brute-Force-Algorithmus schneller sein kann. Das ist immer dann der Fall, wenn die Karten viele Bits haben, es aber nur wenig Karten gibt. Dann ist das Gleichungssystem nahezu vollständig bestimmt und der Brute-Force-Algorithmus muss kaum noch Werte ausprobieren.

stapel3.txt

Die Ausgabe des ISD-Algorithmus:

```
$ ./stapel isd ../Eingabe/stapel3.txt
Lösungsverfahren: Information-Set-Decoding
Die richtigen Karten wurden gefunden:
 Karte  23 (10110111010010111110110011000101011101000011111100001000001100 11...)
 Karte  26 (10111000101111100010111110101010101100110001000011011001100010 11...)
 Karte  34 (11001011010111111111011101000100010010000010110011101010011111 1010...)
 Karte  43 (01010000101101110011110001110011010011001111111100000100000010000...)
 Karte  71 (10110000001001100110110101000100110011100101100110101111011111110...)
 Karte  98 (01111101100010101100110011101011010100110100011001111110101011...)
 Karte 110 (10111111010101001100000101101100111101000010100010100100001001 11...)
 Karte 120 (01110001111110101000010011100011111111110011110110111000101010001...)
 Karte 127 (01110110100111001000011011100100101010111111001011110000000101100...)
 Karte 133 (00100000111001110001101010001111111001111000101111010011001010110...)
 Karte 144 (11000001011001000110100111011111111101111101101101011111010001011...)

Das Lösen hat 8 ms gedauert.
Der Information-Set-Decoding-Algorithmus hat 3 Versuche benötigt.
```

Ausgaben/stapel3_isd.txt

Dieses Beispiel hat der Brute-Force-Algorithmus ebenfalls in 8 Millisekunden mit der gleichen Lösung gelöst. Hier muss der Brute-Force-Algorithmus bereits mehrere Stellen ausprobieren, da es mehr Karten als Bits gibt.

stapel4.txt

Dieses Beispiel wird vom ISD-Algorithmus auch in sehr kurzer Zeit gelöst:

```
$ ./stapel isd ../Eingabe/stapel4.txt
Lösungsverfahren: Information-Set-Decoding
Die richtigen Karten wurden gefunden:
 Karte   0 (11100010000000111101001111111001001100111011101001000111001111 00...)
 Karte  23 (00101100000111101111000010000001011011111110000110011111111111000...)
 Karte  76 (00000111011010010101101110001111101001110010100011000001000001 10...)
 Karte  83 (00110110010111001001001111001111101010011100000100000001100010 1010...)
 Karte  91 (00101100001110001110011110000100110000000000011101101100111010001...)
 Karte  95 (01000011001001101011001111011101111010010110111010111110110111 11...)
 Karte 116 (10000001000101110011010100011000110100110100111100100010001000 01...)
 Karte 146 (11000101110001001000001011101000100110011001111011101001010110 11...)
 Karte 154 (10101010000011111111100111101111000100010011101000001001011110100...)
 Karte 159 (10001100001011001101001011000110110101001101000010000101101010 10...)
 Karte 160 (11110010110001100010100110001001110001111010011100100011010100 10...)

Das Lösen hat 4 ms gedauert.
Der Information-Set-Decoding-Algorithmus hat 13 Versuche benötigt.
```

Ausgaben/stapel4_isd.txt

In diesem Fall hat der Algorithmus 13 Versuche gebraucht, bis eine Lösung gefunden wurde. Nach der theoretischen Analyse unter 1.2.1 beträgt der Erwartungswert für die Anzahl der Versuche bei diesem Beispiel 10. Zumindest in der Größenordnung stimmt die Analyse also mit der Praxis überein. Die Ausgabe des Brute-Force-Algorithmus ist

```
$ ./stapel bruteforce ../Eingabe/stapel4.txt
Lösungsverfahren: Brute-Force
Fortschritt: 1.89182e-06%
Fortschritt: 6.31626e-06%
Fortschritt: 1.26667e-05%
Fortschritt: 1.96247e-05%
Fortschritt: 2.79436e-05%
Fortschritt: 3.96354e-05%
Fortschritt: 5.1066e-05%
Fortschritt: 6.19621e-05%
Die richtigen Karten wurden gefunden:
 Karte   0 (11100010000000111101001111111001001100111011101001000111001111 00...)
 Karte  23 (00101100000111101111000010000001011011111110000110011111111111000...)
 Karte  76 (00000111011010010101101110001111101001110010100011000001000001 10...)
 Karte  83 (00110110010111001001001111001111101010011100000100000001100010 1010...)
 Karte  91 (00101100001110001110011110000100110000000000011101101100111010001...)
 Karte  95 (01000011001001101011001111011101111010010110111010111110110111 11...)
 Karte 116 (10000001000101110011010100011000110100110100111100100010001000 01...)
 Karte 146 (11000101110001001000001011101000100110011001111011101001010110 11...)
 Karte 154 (10101010000011111111100111101111000100010011101000001001011110100...)
```

```
 Karte 159 (1000110000101100110100101100011011010100110100001000010110101010...)
 Karte 160 (1111001011000110001010011000100111000111101001110010001101010010...)

Das Lösen hat 25493 ms gedauert.
```

Ausgaben/stapel4_bruteforce.txt

Hier benötigt der Brute-Force-Algorithmus bereits deutlich länger, kommt aber dennoch zum gleichen Ergebnis.

stapel5.txt

Die Ausgabe des ISD-Algorithmus:

```
$ ./stapel isd ../Eingabe/stapel5.txt
Lösungsverfahren: Information-Set-Decoding
Die richtigen Karten wurden gefunden:
 Karte  70 (1000010011101010001111100100110110011011100101010100010000001001)
 Karte  77 (1101010001001101000111111110000110100010100111000100001001011011)
 Karte 163 (1010111011001100100110001100110001011101001000000011011111100100)
 Karte 167 (0101111111000111000000101111100010111010110101000100000011001000)
 Karte 185 (1010000110101100101110111001100011011110111111010111000101111110)

Das Lösen hat 6 ms gedauert.
Der Information-Set-Decoding-Algorithmus hat 13 Versuche benötigt.
```

Ausgaben/stapel5_isd.txt

Für dieses Beispiel benötigt der Brute-Force-Algorithmus bereits 3:45 Minuten. Damit kommt der Algorithmus hier klar an seine Grenzen. Die Zeit ist nur dadurch noch relativ gering, dass die erste Karte die Nummer 70 ist und der Algorithmus relativ früh abbrechen kann. Ohne Abbrechen nach einer Lösung würde der Algorithmus hier bereits mehrere Stunden benötigen. Mit immer noch unter 10ms Laufzeit zeigt sich hier die Effizienz des ISD-Algorithmus.

3.1 Größere Beispiele

Auf die folgenden, deutlich größeren, Beispiele wird nur der ISD-Algorithmus angewendet, da der Brute-Force-Algorithmus nicht in angemessener Zeit terminieren würde.

Das Beispiel `stapelc0.txt` wurde mit dem `beispielgenerator.py`-Skript generiert und hat 1000 Karten, die jeweils 500 Bits enthalten. Davon sind 14 Karten Öffnungskarten. Damit ist das Beispiel etwa 5-Mal so groß wie `stapel5.txt`. Der Suchraum wurde gegenüber `stapel5.txt` um einen Faktor in der Größenordnung von 2^{800} vergrößert. Dennoch ist der Algorithmus in der Lage, in sehr kurzer Zeit eine Lösung zu finden.

```
$ ./stapel isd ../Eingabe/stapelc0.txt
Lösungsverfahren: Information-Set-Decoding
Die richtigen Karten wurden gefunden:
 Karte  58 (1010010010010000100010111100100110001010100010000100111011111100...)
 Karte 125 (0011110110011110000101101110011000111100110001100000000110010110...)
 Karte 200 (1010111000000001011101000110011111011001010101010100000010000010100...)
 Karte 320 (0101010011110110100000100001000011101001110111101100111111101110...)
 Karte 441 (0111100101100010111000111000111000111001010001001100110011010001...)
 Karte 463 (0010110010010111100101011101011011001001110101110010111010011111...)
 Karte 525 (0000111011110000100000010010011110010010000111100010110100000100...)
 Karte 588 (0000000000111110000000111010111110110010111101111010110110000001...)
 Karte 643 (1101111011000110001010110100110110000100100101101111011111010101000...)
 Karte 655 (1100010001011001111010001111100001010100101000111101010111011110...)
 Karte 697 (0001100001111111100110000000001010010011101011101011000100000000...)
 Karte 718 (0110001110111111100000010111011100000110100001001100110001100110...)
 Karte 812 (1010001101001011011110110110100100001001000111011010100100000111...)
 Karte 977 (0110000100001010010111000111001010110000000010000110100101110010...)
 Karte 987 (1001101111011111111111101111101111000001010010101110011100111110...)

Das Lösen hat 1533 ms gedauert.
Der Information-Set-Decoding-Algorithmus hat 2179 Versuche benötigt.
```

Ausgaben/stapelc0_isd.txt

Der Beispielgenerator gibt zusätzlich zur Eingabedatei die verwendeten Öffnungskarten aus und die Ausgabe stimmt damit überein. Der theoretisch ermittelte Erwartungswert für die Anzahl der Versuche

ist bei diesem Beispiel etwa 2300, sodass die Größenordnung auch hier mit der Theorie übereinstimmt. Die tatsächlich benötigte Anzahl an Versuchen schwankt jedoch – wie zu erwarten – stark.

Dieses Beispiel zeigt aber auch die Grenzen des Verfahrens auf: noch deutlich schwierigere Instanzen wären aufgrund der exponentiellen Laufzeit nicht mehr lösbar. Etwas schwierigere Beispiele könnten gelöst werden, wenn größere Werte für p oder einer der weiterentwickelten Algorithmen verwendet würde.

Wie bereits beschrieben bedeutet schwierig aber nicht unbedingt, dass n, m und k groß sind. Das neu generierte Beispiel `stapelc1.txt` hat $n = 2500$, $m = 2480$ und $k = 20$. Da die Differenz von n und m relativ gering ist, können beide Verfahren auch dieses Beispiel in vergleichsweise kurzer Zeit lösen. Hier die Ausgabe des ISD-Algorithmus:

```
$ ./stapel isd ../Eingabe/stapelc1.txt
Lösungsverfahren: Information-Set-Decoding
Die richtigen Karten wurden gefunden:
 Karte  26 (1100111101010101110101111010100010111110111101100011000001010000...)
 Karte  38 (1110000001001011111000011101010101001000001010011101111001001011...)
 Karte 255 (0101000100011011110101100100110110001010000010000110010100000110...)
 Karte 260 (1100001100011011100100110111100000100101001111101001000101100010...)
 Karte 534 (1111010001100110110011110100100101011100101101010000111110100110...)
 Karte 542 (1110001101010100011100001001110111010010110111101000010101010...)
 Karte 576 (1110010111010110000010111110111011001011011001000000010000110001...)
 Karte 728 (1100000111100110011100111001010001100111011000111110011101111001...)
 Karte 866 (1111010101110000101001110101110111010101011000111100110100100001...)
 Karte 940 (0011101100100010010110011111000101000010000100101000011010100100...)
 Karte 1072 (1011011101111100110111010110101110110111000010100000100001000110...)
 Karte 1157 (0001010010011011100011111010101100011100110011001011111010110101...)
 Karte 1317 (1101101000101010111100100111011010010011011000001111000100101001...)
 Karte 1353 (1101100010110111011001000010111110101010100011100001010011100010...)
 Karte 1579 (1101000100101111011100110010010101001010111111011011010010101010...)
 Karte 1635 (0100000001110010011011110001000001111111110100001101111101000100...)
 Karte 1832 (0000101011011100011000010011011101001101111001111010111101011011...)
 Karte 1981 (0100110101010111100101001010001100010010111110110111010101010100...)
 Karte 2013 (1111011001111010101101110011101010001101101100001101101000100010...)
 Karte 2051 (1001001111011011101111101110010101011010010001101001100011110010...)
 Karte 2315 (0101001001000010000000100000010011011100111000000011010101010111...)

Das Lösen hat 3332 ms gedauert.
Der Information-Set-Decoding-Algorithmus hat 8 Versuche benötigt.
```

Ausgaben/stapelc1_isd.txt

Die Ausgabe des Brute-Force-Algorithmus ist ebenfalls korrekt und im Zip-Ordner zu finden.

Noch schwierigere Instanzen dieses Problems werden aktuell als so schwierig angesehen, dass sie als Grundlage für kryptografische Verfahren verwendet werden. Bernstein, Lange und Peters empfehlen beispielsweise zugrundeliegende Codes, die in diesem Kontext 1632 Karten mit 1269 Bits und 33 Öffnungskarten entsprechen würden, um ein McEliece-Kryptosystem mit einem Sicherheitsniveau von 80 Bits zu erhalten. [2] Der Algorithmus schneidet also schon vergleichsweise gut ab, auch wenn es noch Verbesserungspotential gibt.

4 Quellcode

```
#ifndef STAPEL_ISDSOLVER_H
#define STAPEL_ISDSOLVER_H

#include <vector>
#include <random>

#include "Utils.h"

class ISDSolver {
public:
    ISDSolver();

    int versuche = 0;

    std::vector<std::vector<int>> solve(Utils::Instance i);

private:
```

```
19      std::vector<int> zufaelligePermutation(int n);

21      static void efficientGauss(std::vector<boost::dynamic_bitset<>> &bit_mat);

23      std::mt19937 rng;
};
25

27  #endif //STAPEL_ISDSOLVER_H
```

ISDSolver.h

```
1   #include <algorithm>

3   #include "ISDSolver.h"

5   using namespace std;

7   /*
     * Effizientere Variante des Gauss-Verfahrens, die Bitsets verwendet.
9    * Im ISD-Algorithmus wird die Laufzeit vom Gauss-Algorithmus dominiert,
     * sodass die Optimierungen lohnenswert sind.
11   */
    void ISDSolver::efficientGauss(std::vector<boost::dynamic_bitset<>> &bit_mat) {
13      int h = 0;
        int k = 0;
15      while (h < bit_mat.size() && k < bit_mat[0].size()) {
            int i_max = h;
17          while (i_max < bit_mat.size() && bit_mat[i_max][k] == 0) {
                i_max += 1;
19          }
            if (i_max == bit_mat.size()) {
21              k++;
            } else {
23              // XOR-Swap-Algorithmus, std::swap wäre durch die Speicherverwaltung
                // der temporären Variable ein Flaschenhals
25              if (h != i_max) {
                    bit_mat[h] ^= bit_mat[i_max];
27                  bit_mat[i_max] ^= bit_mat[h];
                    bit_mat[h] ^= bit_mat[i_max];
29              }
                for (int i = h + 1; i < bit_mat.size(); i++) {
31                  if (bit_mat[i][k]) {
                        bit_mat[i] ^= bit_mat[h];
33                  }
                }
35              h++;
                k++;
37          }
        }
39
        // Resubstitution
41      for (int row = bit_mat.size() - 1; row >= 0; row--) {
            auto leading_one = bit_mat[row].find_first();
43          // diese Zeile von allen darüberliegenden Zeilen entfernen, wenn diese
            // eine 1 an der entsprechenden Stelle haben
45          for (int i = 0; i < row; i++) {
                if (bit_mat[i][leading_one]) {
47                  bit_mat[i] ^= bit_mat[row];
                }
49          }
        }
51  }

53  std::vector<std::vector<int>> ISDSolver::solve(Utils::Instance instance) {

55      // Einmal Gauss-Jordan-Algorithmus anpassen. Dadurch werden linear
        // abhängige Zeilen aus der Matrix entfernt, sodass sich die Anzahl der
57      // Zeilen nochmal ändert.
        Utils::gauss(instance.H);
59
        auto n_cols = instance.H[0].size();
61      auto n_rows = instance.H.size();
```

```
63      int t = instance.k + 1;
65      // In allen Threads geteilt, sodass die anderen Threads abbrechen können,
        // wenn ein Thread eine Lösung gefunden hat.
67      vector<int> return_value(0);
69  #pragma omp parallel default(none) firstprivate(rng) shared(return_value, n_cols, n_rows,
        instance, t, versuche)
        {
71          vector<boost::dynamic_bitset<>> H_perm;
            for (int row = 0; row < n_rows; row++) {
73              H_perm.emplace_back(n_cols, 0);
            }
75          boost::dynamic_bitset<> pivots(n_cols, 0);
            vector<int> m_i;
77          m_i.reserve(n_rows);
            vector<int> k_i;
79          k_i.reserve(n_rows);
81          while (return_value.empty()) {
    #pragma omp critical
83              versuche += 1;
85              // die Spalten von H permutieren und in H_perm abspeichern
                auto permutation = zufaelligePermutation(n_cols);
87              for (int row = 0; row < n_rows; row++) {
                    for (int col = 0; col < n_cols; col++) {
89                      H_perm[row][col] = instance.H[row][permutation[col]];
                    }
91              }
93              // H_perm in reduzierte Spaltenform bringen
                efficientGauss(H_perm);
95
                // Pivots finden, um leicht das Inverse der Teilmatrix zu bilden
97              m_i.clear();
                k_i.clear();
99              pivots.clear();
                for (int row = 0; row < n_rows; row++) {
101                 auto col = H_perm[row].find_first();
                    pivots.set(col);
103                 m_i.push_back(col);
                    k_i.push_back(row);
105             }
107             // p wird auf 1 festgelegt. Folglich kann einfach über die Spalten
                // iteriert werden. Dadurch wird über alle p-großen Teilmengen
109             // iteriert
                int p = 1;
111             for (int s = 0; s < n_cols; s++) {
                    // Nur über die Spalten iterieren, die nicht in der
113                 // invertierbaren Submatrix sind
                    if (pivots.test(s))
115                     continue;
117                 vector<int> syndrom(n_rows, 0);
119                 for (int j = 0; j < n_rows; j++)
                        syndrom[j] = H_perm[j][s];
121
                    // Hamming-Gewicht bestimmen
123                 int w = 0;
                    for (int bit: syndrom)
125                     w += bit;
127                 if (w == t - p) {
                        vector<int> solution(n_cols, 0);
129
                        // Die ausgewählte Spalte wird verwendet
131                     solution[s] = 1;
133                     // Berechnen, welche Spalten aus der invertierbaren
                        // Submatrix ebenfalls verwendet werden müssen
```

```
                    for (int i = 0; i < n_rows; i++) {
                        if (syndrom[k_i[i]] == 1) {
                            solution[m_i[i]] = 1;
                        }
                    }

                    // Permutation rückgängig machen
                    vector<int> solution_perm(n_cols);
                    for (int i = 0; i < n_cols; i++) {
                        solution_perm[permutation[i]] = solution[i];
                    }

                    return_value = Utils::get_true_positions(solution_perm);
                }
            }

        }
    }
    return {return_value};
}

ISDSolver::ISDSolver() {
    std::random_device device;
    rng = mt19937(device());
}

vector<int> ISDSolver::zufaelligePermutation(int n) {
    vector<int> permutation(n);
    for (int i = 0; i < n; i++) {
        permutation[i] = i;
    }

    shuffle(permutation.begin(), permutation.end(), rng);
    return permutation;
}
```

ISDSolver.cpp

```
#ifndef STAPEL_BRUTEFORCESOLVER_H
#define STAPEL_BRUTEFORCESOLVER_H

#include <chrono>
#include <vector>
#include <bitset>
#include <unordered_set>
#include <iostream>
#include <omp.h>
#include <algorithm>
#include <cmath>

#include "Utils.h"

using namespace std;

template<int size>
class BruteforceSolver {

    vector<int> row_of_var;
    vector<vector<int>> matrix;

    vector<bitset<size>> row_bitsets;
    vector<bitset<size>> cols_bitsets;

    chrono::time_point<chrono::steady_clock> fortschritt_zeit = chrono::steady_clock::now();

    vector<vector<int>> ones_in_row;
    vector<vector<int>> ones_of_col;

    vector<bitset<size>> results;
    int n_cols;

    int K;

    /*
```

```
37      * Alle Variablen < col werden rekursiv resubstituiert
        */
39      void resub(int col, bitset<size> codeword, int c, bitset<size> syndrome) {

41          // Variablen setzten, die resubstituiert werden können
            // row_of_var[col] >= 0: Diese Variable hat ein Pivotelement im Gauss-Verfahren
43          //                       gehabt und kann resubstituiert werden
            while (col >= 0 && row_of_var[col] >= 0 && c >= 0 && c <= col + 1) {
45
                // Der Wert dieser Spalte wird aus den bereits substituierten
47              // Spalten berechnet, die Summe der bisher substituierten Variablen,
                // skalar multipliziert mit der Zeile, in der das Pivotelement dieser
49              // Variable ist.
                bool v = (row_bitsets[row_of_var[col]] & codeword).count() % 2;
51
                // Wenn die Variable auf 1 gesetzt wurde, muss die Spalte zum aktuellen
53              // Syndrom hinzugefügt werden
                if (v) {
55                  codeword[col] = 1;
                    syndrome ^= cols_bitsets[col];
57                  --c;
                }
59              col--;
            }
61
            // Abbrechen, wenn es nicht mehr genug Spalten gibt, um das Zielgewicht
63          // zu erreichen
            if (c > col + 1) {
65              printProgress(codeword);
                return;
67          }

69          // Abbrechen, wenn das Zielgewicht bereits überschritten wurde
            if (c < 0) {
71              printProgress(codeword);
                return;
73          }

75          // Die Substitution hat die letzte Variable erreicht
            if (col < 0) {
77              results.push_back(codeword);
                return;
79          }

81          // Abbrechen, wenn bereits eine Lösung gefunden wurde
            if (!results.empty()) {
83              return;
            }
85
            // Die Variable 'col' ist frei -> rekursiv versuchen, 0 oder 1 zu substituieren
87          if (row_of_var[col] < 0) {
                codeword[col] = 0;
89              resub(col - 1, codeword, c, syndrome);

91              codeword[col] = 1;
                resub(col - 1, codeword, c - 1, syndrome ^ cols_bitsets[col]);
93              return;
            }
95
            cout << "Ein Fehler ist aufgetreten.\n";
97      }

99      void printProgress(bitset<size> codeword) {
            // maximal einmal pro Sekunde ausgeben
101         if (chrono::duration_cast<chrono::milliseconds>(chrono::steady_clock::now() -
        fortschritt_zeit).count() >=
                3000) {
103             fortschritt_zeit = chrono::steady_clock::now();

105             // Der Fortschritt wird aus dem aktuellen Codewort berechnet.
                // Das ist möglich, da die Rekursion immer zuerst 0 und dann 1
107             // ausprobiert.
                double progress = 0;
```

```
            int count = 0;
            for (int i = n_cols - 1; i >= 0; i--) {
                if (row_of_var[i] < 0) {
                    count += 1;
                    if (codeword[i] == 1) {
                        progress += pow < double > (2.0, -count);
                    }
                }
            }
            cout << "Fortschritt: " << (progress * 100.0) << "% \n";
        }
    }

public:
    vector<vector<int>> solve() {

        auto starting_time = chrono::steady_clock::now();
        bitset<size> solution(0);

        resub(n_cols - 1, solution, K + 1, bitset<size>());

        vector<vector<int>> r;
        for (const auto &s: results) {
            r.push_back(Utils::get_true_positions(s));
        }
        return r;
    }

    explicit BruteforceSolver(Utils::Instance instance) {
        matrix = std::move(instance.H);
        K = instance.k;
        n_cols = matrix[0].size();
    }

    void vorberechnen() {

        // die Matrix in reduzierte Zeilenstufenform bringen
        Utils::gauss(matrix);

        // vorberechnen, welche Zeile das Pivotelement einer Spalte hat
        row_of_var = vector<int>(n_cols, -1);
        for (int i = 0; i < matrix.size(); i++) {
            int j = 0;
            while (matrix[i][j] != 1 && j < matrix[0].size()) {
                j++;
            }
            if (matrix[i][j] == 1) {
                row_of_var[j] = i;
            }
        }

        // Positionen der Einsen einer Zeile vorberechnen, ohne das Pivotelement
        for (auto &row: matrix) {
            vector<int> ones;
            bool first = true;
            for (int i = 0; i < row.size(); i++) {
                if (row[i] == 1) {
                    if (first) {
                        first = false;
                    } else {
                        ones.push_back(i);
                    }
                }
            }
            ones_in_row.push_back(ones);
        }

        // Positionen der Einsen in einer Spalte vorberechnen
        ones_of_col = vector<vector<int>>(n_cols, vector<int>(0));
        for (int col = 0; col < n_cols; col++) {
            if (row_of_var[col] >= 0) {
                ones_of_col[col] = ones_in_row[row_of_var[col]];
            }
```

```
        }

        // Spalten in Bitsets umwandeln, um effizient das Syndrom berechnen zu können
        for (int i = 0; i < n_cols; i++) {
            bitset<size> word;
            word.reset();
            for (int j = 0; j < matrix.size(); j++) {
                if (matrix[j][i])
                    word.set(j, matrix[j][i]);
            }
            cols_bitsets.push_back(word);
        }

        // Zeilen in Bitsets umwandeln, um effizienter rechnen zu können
        for (const auto &row: matrix) {
            bitset<size> bitset;
            for (int i = 0; i < row.size(); i++)
                bitset[i] = row[i];
            row_bitsets.push_back(bitset);
        }
    }
};

#endif //STAPEL_BRUTEFORCESOLVER_H
```

BruteforceSolver.h

```
#include "BruteforceSolver.h"

/*
 * Da der BruteforceSolver ein Template ist, befindet sich die Implementation
 * in der Header-Datei.
 */
```

BruteforceSolver.cpp

```
#include <bitset>
#include <iostream>
#include <vector>
#include <algorithm>
#include <fstream>

#include "Utils.h"

using namespace std;

void Utils::gauss(std::vector<std::vector<int>> &matrix) {
    int h = 0;
    int k = 0;

    while (h < matrix.size() && k < matrix[0].size()) {
        int i_max = h;

        while (i_max < matrix.size() && matrix[i_max][k] == 0) {
            i_max += 1;
        }
        if (i_max == matrix.size()) {
            k++;
        } else {
            swap(matrix[h], matrix[i_max]);
            for (int i = h + 1; i < matrix.size(); i++) {
                if (matrix[i][k]) {
                    for (int j = k; j < matrix[0].size(); j++) {
                        matrix[i][j] ^= matrix[h][j];
                    }
                }
            }
            h++;
            k++;
        }
    }

    // 0-Zeilen entfernen
```

```
38      matrix.erase(std::remove_if(matrix.begin(), matrix.end(), [](auto el) {
            return std::count(el.begin(), el.end(), 1) == 0;
40      }), matrix.end());

42
        // In reduzierte Spaltenform bringen
44      for (int row = matrix.size() - 1; row >= 0; row--) {
            auto leading_one = std::distance(matrix[row].begin(), std::find(matrix[row].begin(),
        matrix[row].end(), 1));
46
            // diese Zeile von allen darüberliegenden Zeilen entfernen, wenn diese eine 1 an der
        entsprechenden Stelle haben
48          for (int i = 0; i < row; i++) { // n times --> O(nm)
                if (matrix[i][leading_one]) {
50                  std::transform(matrix[i].begin(), matrix[i].end(), matrix[row].begin(), matrix
        [i].begin(),
                                   std::bit_xor<>());
52              }
            }
54      }
    }
56
    Utils::Instance Utils::readInstanceFromFile(std::ifstream &ifs) {
58      int n, k, m;

60      ifs >> n;
        ifs >> k;
62      ifs >> m;

64      vector<vector<int>> cards;
        for (int i = 0; i < n; i++) {
66          vector<int> card;
            for (int j = 0; j < m; j++) {
68              char s;
                ifs >> s;
70              card.push_back(s == '1');
            }
72          cards.push_back(card);
        }
74      return Instance{cards, k};
    }
76
    std::vector<std::vector<int>> Utils::transpose(std::vector<std::vector<int>> mat) {
78      vector<vector<int>> mat_transpose;
        for (int bit = 0; bit < mat[0].size(); bit++) {
80          vector<int> equation;
            equation.reserve(mat.size());
82          for (auto &card: mat) {
                equation.push_back(card[bit]);
84          }
            mat_transpose.push_back(equation);
86      }
        return mat_transpose;
88  }
```

Utils.cpp

Literatur

[1] E. Berlekamp, R. McEliece und H. van Tilborg. „On the Inherent Intractability of Certain Coding Problems (Corresp.)“ In: *IEEE Transactions on Information Theory* 24.3 (Mai 1978), S. 384–386. ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055873. URL: http://ieeexplore.ieee.org/document/1055873/ (besucht am 11. 04. 2022).

[2] Daniel J. Bernstein, Tanja Lange und Christiane Peters. „Attacking and Defending the McEliece Cryptosystem“. In: *Post-Quantum Cryptography*. Hrsg. von Johannes Buchmann und Jintai Ding. Bd. 5299. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 31–46. ISBN: 978-3-540-88402-6. DOI: 10.1007/978-3-540-88403-3_3. URL: http://link.springer.com/10.1007/978-3-540-88403-3_3 (besucht am 20. 04. 2022).

[3] P. J. Lee und E. F. Brickell. „An Observation on the Security of McEliece's Public-Key Cryptosystem". In: *Advances in Cryptology — EUROCRYPT '88*. Hrsg. von D. Barstow u. a. Bearb. von G. Goos und J. Hartmanis. Bd. 330. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, S. 275–280. ISBN: 978-3-540-50251-7. DOI: 10.1007/3-540-45961-8_25. URL: http://link.springer.com/10.1007/3-540-45961-8_25 (besucht am 11. 04. 2022).

[4] E. Prange. „The Use of Information Sets in Decoding Cyclic Codes". In: *IEEE Transactions on Information Theory* 8.5 (Sep. 1962), S. 5–9. ISSN: 0018-9448. DOI: 10.1109/TIT.1962.1057777. URL: http://ieeexplore.ieee.org/document/1057777/ (besucht am 11. 04. 2022).

[5] Jacques Stern. „A Method for Finding Codewords of Small Weight". In: *Coding Theory and Applications*. Hrsg. von Gérard Cohen und Jacques Wolfmann. Bd. 388. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 1989, S. 106–113. ISBN: 978-3-540-51643-9. DOI: 10.1007/BFb0019850. URL: http://link.springer.com/10.1007/BFb0019850 (besucht am 11. 04. 2022).