Algorithmische Betrachtungen zum Zarankiewicz-Problem

Matthias Werner

Institut für Informatik

Softwaretechnologie und Programmierungstechnik

Technische Universität Bergakademie Freiberg

14. November 2013

Der Ausgangspunkt dieser Arbeit stammt aus der Ramsey-Theorie und lässt sich auch als sogenanntes Grid-Coloring Problem beschreiben. Ein Grid wird durch die Adjazenzmatrix eines vollständigen bipartiten Graphens $K_{m,n}$ definiert. In diesem Graphen sollen die Kanten mit c Farben so gefärbt werden, dass kein monochromatischer $K_{2,2}$ ($\cong C_4$) als Teilgraph induziert wird.

Die Prüfung einer notwendigen Bedingung bezüglich einer einzelnen Farbe wird auf das Zarankiewicz-Problem führen, ein Problem aus der extremalen Graphentheorie. Für bereits $m = n = 10$ benötigt ein einfacher Algorithmus auf einem Intel X5650 Prozessor mit 2.67 GHz theoretisch etwa 2315 Jahre. Im Rahmen dieser Arbeit sollen mittels tiefergehenden Strukturinformationen Algorithmen zur Lösung des Optimierungsproblems entworfen und analysiert werden.

Diese Arbeit steht unter einer Creative-Commons Lizenz, welche ausschließlich die nichtkommerzielle Verbreitung unter gleichen Bedingungen und mit Namensnennung gestattet.

Siehe: *http://creativecommons.org/licenses/by-nc-sa/3.0/*

Kontakt: *wmatthias at t-online dot de*

Inhaltsverzeichnis

1 Einführung **1**

1.1 Problemstellung . . . 1

1.2 Nichtfärbbarkeit und Komplexität . . . 5

2 Konzeption Overflow-Algorithmus **6**

2.1 Struktur- und Laufzeitbetrachtung . . . 6

2.2 Ansatz für Optimierung . . . 11

2.3 Prozeduren . . . 13

3 Konzeption Slot-Algorithmus **16**

3.1 Matrixkopf und Slots . . . 16

3.2 Höherwertige Mustergenerierung . . . 18

3.2.1 Dreierbildung . . . 18

3.2.2 Viererbildung . . . 19

3.3 Optimierung und Auswertung . . . 20

3.3.1 Slotfensterkombinationen . . . 20

3.3.2 Lösungen und Laufzeit . . . 22

3.3.3 Weitere Resultate . . . 24

4 Bewertung bezüglich des Grid-Coloring Problems **29**

5 Anhang **30**

5.1 Overflow-Algorithmus Pseudocode . . . 30

5.2 Weitere Dokumente . . . 34

1 Einführung

1.1 Problemstellung

Die Motivation zur Untersuchung C_4-freier Graphen liegt in der Frage, ob ein vollständig bipartiter Graph c-färbbar ist, ohne dass er einen monochromatischen C_4-Teilgraphen enthält. Dieses Problem werden wir später auch vereinfachend als ein Gitterfärbungsproblem darstellen. Zunächst seien folgende Definitionen gegeben:

Definition 1.1. *Mit $G = G(V, E)$ bezeichnen wir einen einfachen Graph bestehend aus $|V| < \infty$ Knoten und $|E| \leq \binom{|V|}{2}$ Kanten. $E(G)$ repräsentiert die Menge der Kanten und $V(G)$ die Menge der Knoten des Graphen G.*

C_n definiert einen Kreis der Länge n, das heißt der Graph C_n besteht aus n Knoten und einem geschlossenen Weg mit n Kanten, der alle Knoten durchläuft.

Ferner sei $K_{m,n}$ ein vollständiger bipartiter Graph mit den beiden Klassen V_1, $|V_1| = m$, und V_2, $|V_2| = n$, sodass jeder Knoten aus V_1 mit jedem Knoten aus V_2 verbunden ist.

Beispiel 1.1. *Der Graph C_5 ist ein Kreis mit 5 Kanten und 5 Knoten. $K_{3,2}$ zeigt einen vollständigen bipartiten Graphen mit $|V_1| = 3$ und $|V_2| = 2$ Knoten.*

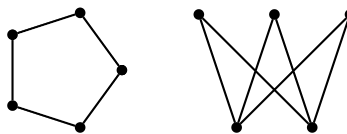

Abbildung 1.1: C_5 und $K_{3,2}$

Für vollständig bipartite Graphen wollen wir im Weiteren eine Definition für die Kantenfärbbarkeit angeben.

Definition 1.2. *Ein vollständiger bipartiter Graph $K_{m,n}$ heißt (rechteckfrei) c-färbbar, wenn die Kanten $E(K_{m,n})$ mit c Farben so gefärbt werden können, dass $K_{m,n}$ keinen monochromatischen Kreis C_4 bzw. $K_{2,2}$ enthält.*

Beispiel 1.2. *Wir betrachten den vollständig bipartiten Graphen $K_{3,3}$ und wollen ihn (rechteckfrei) zweifach färben.*

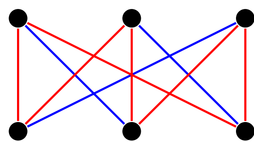

Abbildung 1.2: Rechteckfrei

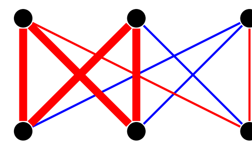

Abbildung 1.3: Rechteck-Konflikt

In Abbildung 1.2 ist eine zulässige Zweifärbung für den Graphen $K_{3,3}$ zusehen, während in Abbildung 1.3 die rotmarkierten Kanten ein Rechteck bilden, das heißt der Graph enthält damit einen monochromatischen $C_4 \cong K_{2,2}$.

Bemerkung 1.1. *Wenn $K_{m,n}$ c-färbbar ist, dann ist auch $K_{p,q}$ c-färbbar für $p \leq m$ und $q \leq n$.*

Bei genaueren Untersuchungen können wir feststellen, dass wir mit vorgegebenem c ab einer bestimmten Größe des Graphen $K_{m,n}$ keine zulässige Lösung mehr finden. Um dafür algorithmische Konzepte entwerfen zu können, benötigen wir eine geeignete Modellierung des folgenden Entscheidungsproblems:

Definition 1.3. *Gegeben sei die Anzahl der Farben $c \in \mathbb{N}$, $c \geq 2$, und die Dimension (m, n) des vollständigen bipartiten Graphens $K_{m,n}$ mit $m \geq 3$ und $n \geq 3$. Entsprechend zu Definition 1.2 ergibt sich das Entscheidungsproblem*

$$\mathrm{CRF}(m,n,c) := \begin{cases} 1, & K_{m,n} \text{ ist } c\text{-färbbar,} \\ 0, & K_{m,n} \text{ ist nicht } c\text{-färbbar.} \end{cases}$$

Wegen Bemerkung 1.1 lässt sich in Analogie zu (FGGP10, 3) die Menge der färbbaren Instanzen durch die Grenzfälle einfach beschreiben (Obstruction Set):

$$\mathrm{OBS}(c) := \{K_{m,n} \mid \mathrm{CRF}(m,n,c) = 0 \Rightarrow \forall p \leq m \wedge \forall q \leq n \wedge (p < m \vee q < n) : \mathrm{CRF}(p,q,c) = 1\}$$

Folgerung 1.1. *Aufgrund der sich durch die Bipartitheit ergebenden Symmetrie gilt für $\mathrm{OBS}(c)$, falls $K_{m,n} \in \mathrm{OBS}(c)$, so ist auch $K_{n,m} \in \mathrm{OBS}(c)$.*

Bemerkung 1.2. *Ein Graph $K_{m,n}$ ist nicht c-färbbar, falls $\exists p \leq m \wedge \exists q \leq n : K_{p,q} \in \mathrm{OBS}(c)$. Wäre $K_{m,n}$ c-färbbar, dann ergäbe sich auch für jeden Teilgraphen $K_{p,q}$ eine zulässige c-Färbung, wodurch $K_{p,q} \notin \mathrm{OBS}(c)$ für alle $p \leq m$ und $q \leq n$.*

Für den vollständigen bipartiten Graphen eignet sich eine Tabellendarstellung, in der Farbenindizes für die Kantenfärbung benutzt werden. Das Modell sieht wie folgt aus:

Modell - Matrixrepräsentation des Graphen $K_{m,n}$

$$\begin{array}{rcll} A = (a_{ij}) & = & \gamma_{ij} = \gamma(e_{ij}) & \\ \gamma & : & E(K_{m,n}) \to \{0, \ldots, c-1\} & \text{(Farbindex)} \\ e_{ij} = e(v_i, w_j) & \in & E(K_{m,n}) & \\ v_i & \in & V_1 \qquad i = 1, 2, \ldots, m & \\ w_j & \in & V_2 \qquad j = 1, 2, \ldots, n & \end{array} \tag{1.1}$$

$v_i \setminus w_j$	w_1	w_2	$\ldots$	w_n
v_1	γ_{00}	γ_{01}	$\ldots$	γ_{0n}
v_2	γ_{10}	γ_{11}	$\ldots$	γ_{1n}
$\vdots$	$\vdots$	$\vdots$		$\vdots$
v_m	γ_{m0}	γ_{m1}	$\ldots$	γ_{mn}

Abbildung 1.4: Matrixtableau zu (1.1)

$v_i \setminus w_j$	w_1	w_2	w_3
v_1	1	0	1
v_2	1	1	0
v_3	0	1	1

Abbildung 1.5: 2-Färbung für $K_{3,3}$ (Abbildung 1.2)

Man ist nun bestrebt, die Menge $\mathrm{OBS}(c)$ zu konkretisieren. In der Literatur sind bereits Ergebnisse für bestimmte c erzielt worden. Beispielsweise ergibt sich für $c = 2$ ((FGGP10)):

$i \setminus j$	3	4	5	6	7	8
3	1	1	1	1	0	0
4	1	1	1	1	0	0
5	1	1	0	0	0	0
6	1	1	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0

Tabelle 1.1: Lösungsmenge für $\mathrm{CRF}(i, j, 2)$, $\mathrm{OBS}(2) = \{K_{7,3}, K_{5,5}, K_{3,7}\}$

Wir wollen zunächst ein Teilproblem betrachten, welches sich aus dem folgenden Satz herleitet:

Satz 1.1. *(FGGP10, 4) Ist $K_{m,n}$ c-färbbar, dann muss $K_{m,n}$ einen rechteckfrei monochromatisch gefärbten Teilgraphen $A \subseteq K_{m,n}$ enthalten, für den gilt:*

$$|E(A)| \geq \left\lceil \frac{m \cdot n}{c} \right\rceil$$

Beweis. Ein c-färbbarer Graph $K_{m,n}$ lässt sich zerlegen in c rechteckfreie Teilgraphen jeweils einer Farbe, sodass wegen des Schubfachprinzips ein Teilgraph mindestens $\lceil \frac{mn}{c} \rceil$ Kanten haben muss. □

Damit ergibt sich eine notwendige Forderung bezüglich des Entscheidungsproblems $\mathrm{CRF}(m, n, c)$. Wenn es nicht möglich ist, einen vollständig bipartiten Graphen $K_{m,n}$ mit einer Farbe so zu färben, dass die Anzahl der gefärbten Kanten die Schranke $\lceil \frac{m \cdot n}{c} \rceil$ erreicht, dann ist $K_{m,n}$ auch nicht c-färbbar.

Eine reduzierte Formulierung des Sachverhalts findet sich mit der Definition C_4-gesättigter Graphen ((BF02)):

Definition 1.4. *Gegeben sei der Graph H. Ein Graph G heißt H-frei, wenn G keinen Teilgraphen enthält, der isomorph zu H ist. Ein Graph G heißt H-gesättigter Teilgraph eines Graphen K, falls G ein H-freier Teilgraph von K ist und für jede Kante $e \in E(K) \setminus E(G)$ gilt: $G \cup \{e\}$ ist nicht H-frei.*

Folgerung 1.2. *Sei G ein maximaler, C_4-gesättigter Teilgraph von $K_{m,n}$. Gilt $|E(G)| < \lceil \frac{m \cdot n}{c} \rceil$, so ist $K_{m,n}$ nicht (rechteckfrei) c-färbbar.*

Definition 1.5. *Mit $\mathrm{maxrf}(m, n)$ sei die Anzahl der Kanten des maximalen, C_4-gesättigten Teilgraphen $G(m, n)$ von $K_{m,n}$ bezeichnet, also $\mathrm{maxrf}(m, n) = |E(G(m, n))|$.*

Im Rahmen dieser Arbeit beschränken wir uns weitestgehend auf den Fall $c = 4$, insbesondere auf die Frage der Nicht-Vierfärbbarkeit von vollständigen bipartiten Graphen.

Aus Folgerung 1.2 lässt sich eine Matrixdarstellung ableiten, die nur Boolesche Werte erfasst. Wir geben uns die Dimension (m, n) vor und erhalten das Modell:

Optimierungsproblem - C_4-gesättigte Teilgraphen G von $K_{m,n}$

$$|E(G)| = \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij} \to \max \tag{1.2}$$

$$G = G(m,n) = V_1 \times V_2 = \{1,\dots,m\} \times \{1,\dots,n\} \tag{1.3}$$

$$A = (a_{ij}) = \begin{cases} 1, & \text{Kante } (v_i, w_j) \text{ gehört zu } G, \\ 0, & \text{Kante } (v_i, w_j) \text{ gehört nicht zu } G. \end{cases}$$

$$\forall (v_p, w_q) \in G, v_i \neq v_p, w_j \neq w_q : \{(v_i, w_j), (v_p, w_j), (v_p, w_q), (v_i, w_q)\} \not\subseteq E(G) \tag{1.4}$$

$$v_i \in V_1 \qquad i = 1, 2, \dots, m$$

$$w_j \in V_2 \qquad j = 1, 2, \dots, n$$

Die Zielfunktion in (1.2) liefert den größten C_4-freien Teilgraphen $G(m, n)$ des vollständigen bipartiten Graphen $K(m, n)$. In (1.3) findet sich die Gitterdarstellung (Grid) aus (FGGP10) wieder. Ebenso beschreibt (1.4) die Forderung der Rechteckfreiheit des bipartiten Graphen $G(m, n)$. Das ist gleichbedeutend zu:

$$\forall \{v_i, v_p\} \subset V_1, \forall \{w_j, w_q\} \subset V_2 : |E(\{v_i, v_p\} \times \{w_j, w_q\})| \leq 3$$

Definition 1.6. *Das Rechteck $R(i, j, p, q) := (\{v_i, v_p\} \times \{w_j, w_q\})$ mit $1 \leq i < p \leq m$ und $1 \leq j < q \leq n$ sowie $\{v_i, v_p\} \subset V_1$ und $\{w_j, w_q\} \subset V_2$ sei in A enthalten, wenn $a_{ij} = a_{iq} = a_{pj} = a_{pq} = 1$ gilt.*

Die Matrixrepräsentation $A = (a_{ij})$ für C_4-freie Graphen darf damit keine Rechteckkonfigurationen mit entsprechenden Einsen wie in Abbildung 1.6 enthalten.

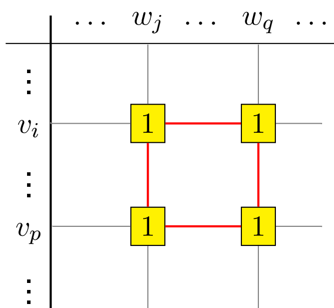

Abbildung 1.6: Rechteck im Gitter $G(m, n)$

$v_i \setminus w_j$	w_1	w_2	w_3	w_4	w_5
v_1	1	1	1	1	0
v_2	1	0	0	0	1
v_3	0	1	0	0	1
v_4	0	0	1	0	1
v_5	0	0	0	1	1

Abbildung 1.7: C_4-gesättigter Graph $G(5, 5)$

In Abbildung 1.7 ist eine optimale (rechteckfreie) Matrixkonfiguration mit $|E(G)| = 12$ Kanten für den maximalen, C_4-gesättigten Teilgraphen $G(5, 5)$ angegeben. Nach Satz 1.1 werden beispielsweise für eine Zweifärbung $\lceil \frac{5 \cdot 5}{2} \rceil = 13$ Kanten benötigt, womit $K(5, 5)$ nicht 2-färbbar sein kann. Für die gewünschte Vierfärbbarkeit können wir allein mit dem besagten Satz keine Aussage für $\mathrm{CRF}(5, 5, 4)$ treffen, da $12 > \lceil \frac{5 \cdot 5}{4} \rceil = 7$ erfüllt ist.

Bemerkung 1.3. *Die einzige einschränkende Bedingung der zulässigen Menge in unserem Modell ist die der Rechteckfreiheit (1.4). Zeilen und Spalten können jeweils permutiert werden. Tatsächlich spielt die Reihenfolge bzw. die Anordnung der Knoten innerhalb einer Klasse des bipartiten Graphen keine Rolle.*

1.2 Nichtfärbbarkeit und Komplexität

Für die Aussage zur Nichtfärbbarkeit von $K_{m,n}$ muss gezeigt werden, dass die Anzahl der Kanten $\mathrm{maxrf}(m,n)$ des C_4-gesättigten Teilgraphen $G(m,n)$ die Schranke aus Satz 1.1 unterschreitet.

In der Literatur wird das sogenannte Zarankiewicz Problem ((FGGP10), (BF02),(Rom75)) zitiert, welches sich äquivalent zu unserem definieren lässt.

Definition 1.7. *Gegeben sei $m \geq 3, n \geq 3$ und $r \geq 2, s \geq 2$. Die Zarankiewicz-Funktion $Z_{r,s}(m,n)$ gibt die minimale Kantenanzahl in einem bipartiten Graphen $k_{m,n}$ an[1], ab der ein zu $K_{r,s}$ isomorpher Teilgraph induziert wird. Mit $r = s = 2$ bezeichnet $K_{r,s}$ unseren C_4 aus Abbildung 1.3. Daraus lässt sich $\mathrm{maxrf}(m,n) = Z_{2,2}(m,n) - 1 =: Z_2(m,n) - 1$ schließen.*

Folgerung 1.3. *$K_{m,n}$ ist nicht c-färbbar, wenn $Z_2(m,n) \leq \lceil \frac{m \cdot n}{c} \rceil$ gilt.*

In der besagten Literatur werden weitere Schranken für $Z_2(m,n)$ angegeben, die die Konkretisierung von $\mathrm{OBS}(c)$ vereinfachen. Zahlreiche Anwendungen des Zarankiewicz-Problems finden sich auch in (Guy69). Wir werden uns im Folgenden den kombinatorischen Betrachtungen widmen, die Auskunft über die Komplexität des Zarankiewicz-Problems geben. Ohne tiefergreifende Strukturbetrachtungen erhält man nun mittels vollständiger Enumeration $2^{m \cdot n}$ auf Rechteckfreiheit zu testende Matrixkonfigurationen, das heißt ein Feldelement kann entweder den Wert 0 oder 1 annehmen.

Mit Hilfe der Bemerkung 1.3 lasen sich Äquivalenzklassen bilden, in denen die Matrizen zusammengefasst werden, die durch Zeilen- oder Spaltenpermutation ineinander überführbar sind. Es müssen nur noch die verschiedenen Repräsentanten betrachtet werden. Die Anzahl dieser Äquivalenzklassen lässt sich jedoch nicht trivial bestimmen. Für quadratische Matrizen sind die Anzahlen auf `http://oeis.org/A089006` bis zur Dimension 12×12 aufgeführt. Die Anzahl der $(0,1)$ Matrizen ließe sich noch weiter reduzieren, sodass nur nicht-isomorphe bipartite Graphen betrachtet werden.

Um eine gegebene Matrixkonfiguration auf Rechteckfreiheit zu testen, müssen alle Rechtecke untersucht werden (siehe Abbildung 1.6). In einer Matrix der Dimension $m \times n$ gibt es $\binom{m}{2}\binom{n}{2}$ verschiedene Kombinationen für mögliche Rechtecke, die nicht in A enthalten sein dürfen.

Folgende Gegenüberstellung zeigt den theoretischen Zeitaufwand der Generierung und Validierung auf einer CPU der Leistung 2.67 GHz (1 Instruktion je Zyklus, $2.67 \cdot 10^9$ IPS):

$k = m = n$	$\binom{m}{2}\binom{n}{2}$	BRUTEFORCE		ZEILEN & SPALTENSORTIERT	
		Belegungen	Theor. Zeit	Belegungen	Theor. Zeit
8	784	$1{,}845 \cdot 10^{19}$	$1{,}71 \cdot 10^5$ a	$9{,}182 \cdot 10^{11}$	66 h 44 min
11	3025	$2{,}659 \cdot 10^{36}$	$9{,}55 \cdot 10^{22}$ a	$7{,}880 \cdot 10^{23}$	$2{,}83 \cdot 10^{10}$ a
12	4356	$2{,}230 \cdot 10^{43}$	$1{,}15 \cdot 10^{30}$ a	$8{,}291 \cdot 10^{28}$	$4{,}29 \cdot 10^{15}$ a

Tabelle 1.2: Theoretischer Zeitaufwand ohne und mit Ausnutzung der Symmetrien

[1] im Gegensatz zu $K_{m,n}$ ist $k_{m,n}$ nicht notwendigerweise vollständig

2 Konzeption Overflow-Algorithmus

Da unsere Matrix A nur aus 0-1-Werten besteht, bietet sich die Verwendung von Binärvektoren an. Eine Zeile i wird durch einen Binärvektor als Zahl $a_i \in \{1, \dots, 2^n - 1\}$ repräsentiert. Da die Zeilenanordnung nach Bemerkung 1.3 keine Rolle spielt, werden die Zahlen sortierend erzeugt, sodass $a_{i+1} \geq a_i$.

Die Idee des Overflow-Algorithmus ist, dass je Zeile ein Bereich an Zahlen durchlaufen wird, um die Bitmuster in einer Zeile zu generieren. Jede daraus entstehende Instanz muss auf Rechteckfreiheit geprüft werden. Ist der Durchlauf in der letzten Zeile a_m beendet, wird die darüber liegende Zeile $m-1$ aufgerufen, deren Wert nun erhöht wird. Anschließend wird erneut die Zeile m von a_{m-1} bis $(101\dots1)_2$ durchlaufen. Wenn die Zeile $m-1$ ihrerseits den Zahlenbreich beendet hat, wird Zeile $m-2$ aufgerufen.

Wenn also eine Zeile einen „Überlauf“ (Overflow) erzeugt, in dem sie an die Grenze des Zahlenbereiches gelangt, so wird die vorherige Zeile inkrementiert, solange bis diese einen weiteren Überlauf generiert. Das Ende ist erreicht, wenn die erste Zeile a_1 „überläuft“.

Der Overflow-Algorithmus ist exakt, liefert also eine optimale Lösung, und wir geben eine Implementierung an, die ohne Rekursion auskommt.

2.1 Struktur- und Laufzeitbetrachtung

Wir wollen zunächst den Ablauf des Algorithmus veranschaulichen und anschließend nützliche Resultate in Bezug auf die Laufzeit herleiten. Die in Unterabschnitt 2.3 angegebene Implementation enthält weitere Optimierungen, die wir für unsere Laufzeitanalyse nicht einbeziehen werden. Stattdessen gehen wir von einer einfachen Inkrementierung in jedem Schritt aus, sodass lediglich die Sortierung $a_i \leq a_{i+1}$ berücksichtigt bleibt.

Der Overflow-Algorithmus durchläuft Zahlenbereiche, deren Zahlen wir als Gitterpunkte auffassen können:

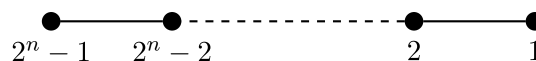

Abbildung 2.8: Zahlbereich einer Zeile in der Gitterpunktanordnung

Mit der simplen Startbelegung $A = (a_1, \dots, a_m) = (1, \dots, 1)$ erhalten wir zunächst folgendes Strukturverhalten des Algorithmus:

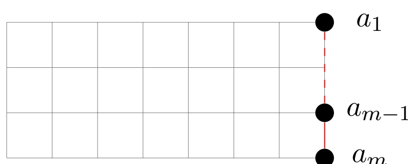

Abbildung 2.9: Erster Schritt - Anfangsstruktur

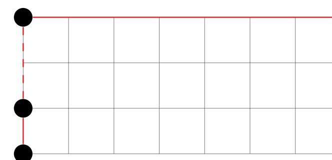

Abbildung 2.10: Letzter Schritt - Zielstruktur

Aufgrund der Sortierung $a_i \leq a_{i+1}$ ergeben sich Wege von $p := 1$ nach $q := a_m$, welche insgesamt stets $m-1$-Schritte nach Süden und $(a_m - 1)$-Schritte nach Westen zurücklegen. Gesucht ist die Anzahl aller Wege, die der Algorithmus verfolgt.

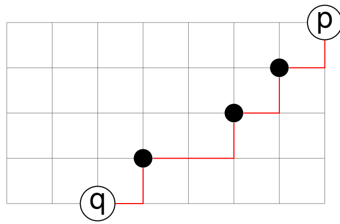

Abbildung 2.11: Beispiel eines Weges

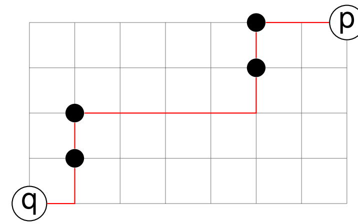

Abbildung 2.12: Beispiel eines Weges

p und q spannen nun das so erhaltene Gitter auf, das mit (p, q, m) definiert sei. Nun ist die Anzahl aller Wege von p nach q gesucht (nur Süd-West-Kombinationen).

Satz 2.1. *Gegeben seien die Zahlen* $\{a_1, \ldots, a_m\}$ *mit* $a_i \in \{1, \ldots, T\}$ *und* $a_i \leq a_{i+1}$, $i \in \{1, \ldots, m\}$ *sowie* $T := 2^n - 1$*. Sei* $d := a_m - a_1$ *und* $m \geq n \geq 3$.

Die Anzahl der Wege von a_1 *nach* a_m *im Zahlengitter* (a_1, a_m, m) *beträgt:*

$$\binom{a_m - a_1 + m - 1}{m - 1} = \binom{d + m - 1}{d}$$

Die Anzahl $N(m, n)$ *aller Wege von* 1 *nach* $T = 2^n - 1$ *beträgt:*

$$N(m, n) = \sum_{j=0}^{T-1} \binom{m + j - 1}{j} = \binom{T - 1 + m}{m}$$

Beweis. Gegeben sei das Gitter (p, q, m) mit $p, q \in \mathbb{N}$ und $m \geq 3$.

Wir suchen die Anzahl der Wörter der Länge $q - p + m - 1$ über dem Alphabet $\Gamma = \{W, S\}$. W beschreibt einen Schritt in westlicher und S einen Schritt in südlicher Richtung. Entsprechend dem Gitter müssen $q - p =: j$ Schritte in westlicher und $m - 1$ Schritte in südlicher Richtung vollzogen werden. Die Auswahl ist mit dem Binomialkoeffizienten beschrieben und die sich daraus ergebende Anzahl der S-W-Wege von p nach q beträgt $\binom{m+j-1}{j}$.

Wir lassen nun j stellvertretend für den Abstand $a_m - a_1$ von 0 bis $T - 1$ laufen und erhalten die Summe der Binomialkoeffizienten, welche sich bekanntermaßen zusammenfassen lässt:

$$N(m, n) = \sum_{j=0}^{T-1} \binom{m + j - 1}{j} = \binom{T - 1 + m}{m} = \binom{2^n - 2 + m}{m}$$

□

Folgerung 2.1. *Aufgrund der* 2^n *im Binomialkoeffizienten wird deutlich, dass wir die Spaltenanzahl zu minimieren haben. Wegen der Symmetrie des Zarankiewicz-Problems können wir die Zeilen- und Spaltenanzahl so wählen, dass stets* $m \geq n$ *erfüllt ist. Für ein* $p > q > 0$ *gilt schließlich:*

$$\binom{2^p - 2 + q}{q} > \binom{2^q - 2 + p}{p}$$

Insofern wir nicht-quadratische Dimensionen (m, n) betrachten, ist im Weiteren die Festlegung $m \geq n$ zu treffen.

Satz 2.2. *Wenn der Startwert der ersten Zeile von $a_1 = 1$ auf $a_1 = 2^p$ für $0 \leq p < n$ erhöht wird, ergibt sich eine relative Laufzeitverbesserung von:*

$$\frac{c_p}{c_0} := \frac{N(m,n,2^p,2^n-1)}{N(m,n,1,2^n-1)} = \prod_{i=1}^{m} \frac{2^n - 2^p + m - i}{2^n - 1 + m - i}$$

Gegeben sei $\beta(p,m,n) := \left(1 - \frac{2^p-1}{2^n-2}\right)^m$. Wächst m asymptotisch langsamer als $2^{\frac{n}{2}}$, so konvergiert $\beta(p,m,n)$ gegen $\frac{c_p}{c_0}$ für $0 \leq p < n$ und $n \to \infty$.

Beweis. Gegeben seien die Spaltenanzahl n und die Zeilenanzahl m, wobei $\binom{n}{2} > m \geq n$. Ferner sei $a_1 = 2^p$ mit $p \in [0, n-1]$. Die Laufzeiten betragen nach Satz 2.1 $c_p = \binom{2^n-1-2^p+m}{m}$ und $c_0 = \binom{2^n-2+m}{m}$. Da p hier stetig ist, muss die erweitere Definition des Binomialkoeffizienten verwendet werden[2]. Damit lässt sich folgende Abschätzung aufstellen:

$$\begin{aligned} \frac{c_p}{c_0} &= \frac{(2^n-1-2^p+m)!}{(2^n-1-2^p)!} \cdot \frac{(2^n-2)!}{(2^n-2+m)!} = \prod_{i=1}^{m} \frac{2^n - 2^p + m - i}{2^n - 1 + m - i} \\ &= \frac{(2^n-1-2^p)^m}{(2^n-2)^m} \cdot \underbrace{\prod_{i=1}^{m} \frac{1 + \dfrac{m-i+1}{2^n-1-2^p}}{1 + \dfrac{m-i+1}{2^n-2}}}_{=:\delta(p,m,n)\geq 1} \\ &\geq \left(\frac{2^n-1-2^p}{2^n-2}\right)^m = \left(1 - \frac{2^p-1}{2^n-2}\right)^m =: \beta(p,m,n) \end{aligned} \tag{2.5}$$

Der Faktor $\delta(p,m,n)$ wächst streng monoton in p und es gilt $\delta(0,m,n) = 1$. Der absolute Fehler ergibt sich zu $\Delta(p,m,n) := |\frac{c_p}{c_0} - \beta(p,m,n)| = \beta(p,m,n) \cdot (\delta(p,m,n) - 1)$. Eine weitere Abschätzung liefert:

$$\delta(p,m,n) \leq \left(\frac{1 + \dfrac{m}{2^n-1-2^p}}{1 + \dfrac{m}{2^n-2}}\right)^m \overset{p\leq n-1}{\leq} \left(\frac{1 + \dfrac{m}{2^{n-1}-1}}{1 + \dfrac{m}{2^n-2}}\right)^m = \left(1 + \frac{m}{2^n-2+m}\right)^m \tag{2.6}$$

Ist m unabhängig von n fixiert, so konvergiert der letzte Term gegen 1 für $n \to \infty$. Für $m = m(n) = \sqrt{2^n} = 2^{\frac{n}{2}}$ hingegen gilt:

$$\left(1 + \frac{2^{\frac{n}{2}}}{2^n - 2 + 2^{\frac{n}{2}}}\right)^{2^{\frac{n}{2}}} = \left(1 + \frac{1}{2^{\frac{n}{2}} - 1}\right)^{2^{\frac{n}{2}}} \xrightarrow{n\to\infty} \mathrm{e}$$

□

Die in Gleichung 2.6 gemachte Abschätzung ist recht grob. Für 36×6 erhält man für die obere Abschätzung $\delta(p,36,6) \leq (1 + \frac{36}{2^6+34})^{36} \approx 77917$. Tatsächlich ist aber $\delta(p,36,6) \leq \delta(5,36,6) \approx 70.51$.

[2] Für $\alpha \in \mathbb{C}$ gilt $\binom{\alpha}{k} = \frac{\alpha(\alpha-1)(\alpha-2)\cdots(\alpha-k+1)}{k!}$ für $k > 0$, $\binom{\alpha}{0} = 1$ und $\binom{\alpha}{k} = 0$ für $k < 0$

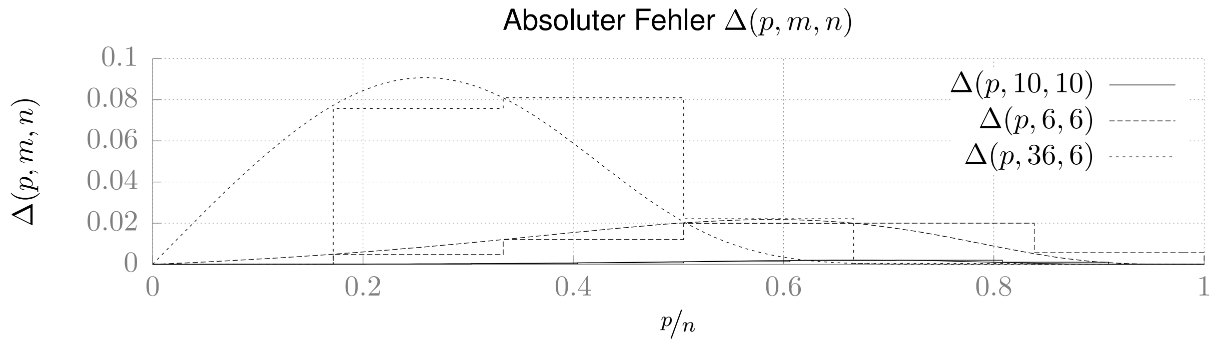

Abbildung 2.13: Absoluter Fehler $\Delta(p, m, n)$ bei Abschätzung von $\frac{c_p}{c_0}$ mit $\beta(p, m, n)$

Abbildung 2.13 zeigt den absoluten Fehler, wenn $\frac{c_p}{c_0} = \frac{N(m,n,2^p,2^n-1)}{N(m,n,1,2^n-1))}$ durch $\beta(p, m, n)$ abgeschätzt wird. Im Fall $\Delta(p, 36, 6)$ wird die hohe Abweichung deutlich, da m groß gegenüber n gewählt worden ist. Im Gegensatz dazu weist $\Delta(p, 6, 6)$ nur noch eine Abweichung von maximal 2 Prozentpunkten auf. Bereits bei 10×10 ist die Abschätzung $\beta(p, 10, 10)$ hinreichend genau.

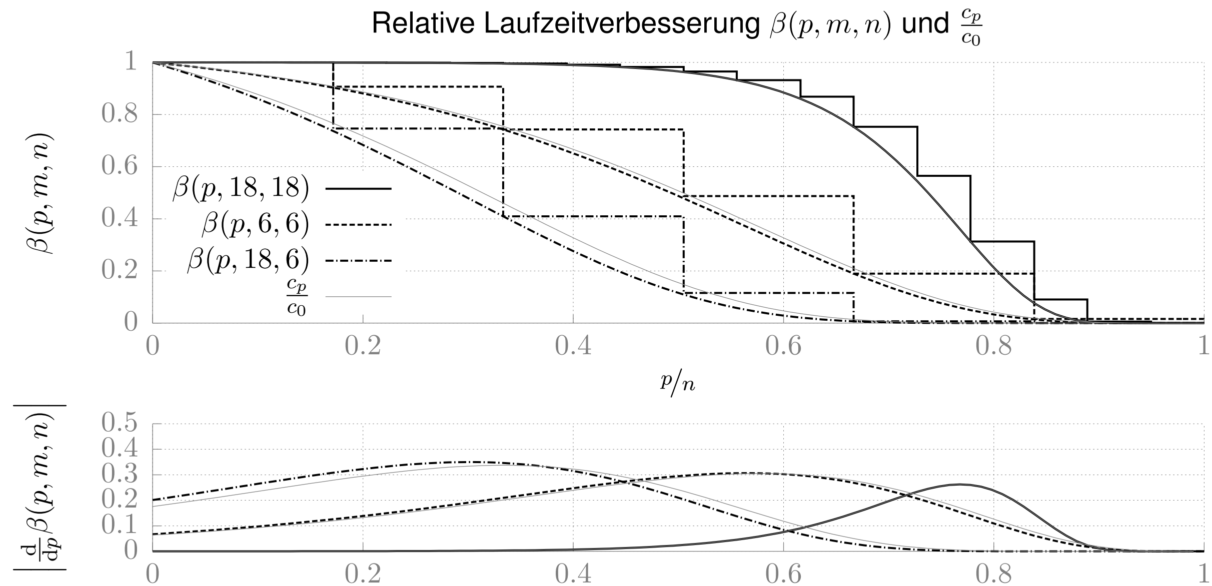

Abbildung 2.14: Relative Laufzeitverbesserung für $p \in [0, n)$, $a_1 = 2^p$

In Abbildung 2.14 wird die relative Laufzeitverbesserung des Overflow-Algorithmus deutlich, wenn a_1 bei 2^p startet statt bei $2^0 = 1$. Ebenfalls sichtbar ist die Approximationsgüte von $\beta(p, m, n)$ bezogen auf $\frac{c_p}{c_0}$. Diese Abschätzung konvergiert nach Satz 2.2 gegen $\frac{c_p}{c_0}$, sodass bei $\beta(p, 18, 18)$ in der Abbildung kein Unterschied mehr zu erkennen ist. Bei großer Spaltenanzahl n erfolgt die Beschleunigung wesentlich später als bei kleineren Spaltendimensionen. Dies ergibt sich offensichtlich aus dem Wachstumsverhalten des Binomialkoeffizienten und der Tatsache, dass sich die Gitterbreiten bei einer Bitinkrementierung exponentiell verhalten, genauer 2^{p+1} vs. 2^p. Kleine Bitpositionen haben demnach kaum Auswirkung auf die gesamte Laufzeit (wegen $\frac{2^{p+1}-2^p}{2^n} = \frac{2^p}{2^n}$).

Beispielsweise greift die Laufzeitverbesserung bei der Dimension $(18, 18)$ erst etwa nach der $10.$ Bitposition, sprich die Bitpositionen zwischen 2^0 und 2^{10} haben noch keinen signifikanten Einfluss auf die Laufzeit.

Mit Hilfe von Satz 2.2 können wir die wesentlich einfachere Ableitung von $\beta(p, m, n)$ für weitere Überlegungen verwenden. Das Maximum der relativen Laufzeitbeschleunigung liegt bei:

$$\begin{aligned} x_{\max}(m, n) := \arg\max\left\{\left|\frac{\mathrm{d}}{\mathrm{d}x}\beta(x, m, n)\right|\right\} &= \log_2\left(\frac{2^n - 1}{m}\right) \\ &= n - \log_2(m) - \log_2(1 - 2^{-n}) \\ &\approx n - \log_2(m) \end{aligned}$$

Für die Dimension $(18, 18)$ liegt das Maximum bei $x_{\max}(18, 18) = 13.83$.

Ein Ansatz für die Optimierung wäre also, die erste Zahl a_1 zu maximieren. Anhand der bzgl. a_1 „größten" Lösungen werden wir erkennen, dass das Führungsbit in a_1 relativ nah dem Wert $x_{\max}$ kommt. Leider ist nicht bekannt, welches Bitmuster die erste Zeile a_1 in der „größten" Optimallösung annimmt. Dennoch werden wir eine Abschätzung liefern, um die Verbesserung zu demonstrieren, wenn wir a_1 mit dem Führungsbit $p \approx n - \log_2(m)$ und einem passenden Muster fixieren können.

2.2 Ansatz für Optimierung

·	·	1	·	1	1	1	·	46
·	·	1	1	·	·	·	1	49
·	1	·	·	1	·	·	1	73
·	1	·	1	·	1	·	·	84
·	1	1	·	·	·	·	·	96
1	·	·	·	·	1	·	1	133
1	·	·	1	1	·	·	·	152
1	·	1	·	·	·	·	·	160
1	1	·	·	·	·	1	·	194

·	·	1	·	1	1	1	·	·	92
·	·	1	1	·	·	·	1	1	99
·	1	·	·	·	1	·	1	·	138
·	1	·	·	1	·	·	·	1	145
·	1	·	1	·	·	1	·	·	164
1	·	·	·	·	·	1	1	·	262
1	·	·	·	·	1	·	·	1	265
1	·	·	1	1	·	·	·	·	304
1	1	1	·	·	·	·	·	·	448

Abbildung 2.15: Overflow-Alg. Optimallösungen $(9,8)$ und $(9,9)$

·	·	·	1	1	1	1	15
·	·	1	·	·	1	·	18
·	·	1	·	1	·	·	20
·	·	1	1	·	·	·	24
·	1	·	·	·	1	·	34
·	1	·	·	1	·	·	36
·	1	·	1	·	·	·	40
·	1	1	·	·	·	1	49
1	·	·	·	·	·	1	65
1	·	·	·	·	1	·	66
1	·	·	·	1	·	·	68
1	·	·	1	·	·	·	72
1	·	1	·	·	·	·	80
1	1	·	·	·	·	·	96

·	·	·	1	·	1	1	·	22
·	·	·	1	1	·	·	1	25
·	·	1	·	·	1	·	1	37
·	·	1	·	1	·	1	·	42
·	1	·	·	·	·	1	1	67
·	1	·	·	1	1	·	·	76
·	1	1	1	·	·	·	·	112
1	·	·	·	·	·	·	1	129
1	·	·	·	·	·	1	·	130
1	·	·	·	·	1	·	·	132
1	·	·	·	1	·	·	·	136
1	·	·	1	·	·	·	·	144
1	·	1	·	·	·	·	·	160
1	1	·	·	·	·	·	·	192

Abbildung 2.16: Overflow-Alg. Optimallösungen $(14,7)$ und $(14,8)$

In Abbildung 2.15 und Abbildung 2.16 sind die Optimallösungen zu sehen, die der Algorithmus als Letztes ermittelt hat. Das Führungsbit in a_1 liegt bis auf eine Stelle bei $\lfloor n - \log_2(m) \rfloor$. Bezüglich der Führungsbits aller Zeilen lässt sich eine Gruppierung feststellen, die im Sinne des noch folgenden Abschnitt 3 als Slots bezeichnet werden können. Die Sloteinteilung ist im Allgemeinen nicht bekannt, genauso wenig wie das Muster, das in a_1 in der Optimallösung vorkommen darf.

Zur Demonstration des Laufzeitverhaltens soll a_1 dennoch auf ein spezifisches Muster fixiert werden, das anhand gewisser Schrankenbedingungen in der Optimallösung vorkommen muss. Das Führungsbit von a_1 wird dabei auf $p = \lfloor n - \log_2(m) \rfloor$ gesetzt.

Zur Ausformulierung der Optimierungsidee nützen folgende Resultate aus Abschnitt 3. Satz 3.1 liefert eine direkte Lösung des Problems, wenn $m > \binom{n}{2}$. In Satz 3.3 wird ein ideales Muster ermittelt, das zu einer Abschätzung des Zarankiewicz-Problems führt. Mit der Gleichung $m \cdot \binom{\beta}{2} = \binom{n}{2}$ erhält man für β:

$$\beta = \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{n(n-1)}{m}}$$

Die Annahme ist nun, dass das Muster $\beta_1 := \lfloor \beta + s \rfloor$ in a_1 im Sinne der Optimallösung vorkommen darf, wobei $s \in [0, 1)$. s gewichtet das Auftreten des nächst höheren Musters. Es ist nicht immer vorteilhaft, das größte Muster in a_1 zu platzieren. Vermutlich ergeben sich durch die Abschätzung für das Maximum in den nachfolgenden Zeilen (siehe Tabelle 2.7) häufiger unnötige Verzweigungen, wenn a_1 eine höhere Bitzahl aufweist. Eine niedrigere Bitzahl verlangsamt wiederum die Laufzeit. Beispielhaft ist das Problem $(10, 10)$. Für $\beta_1 = 3$ terminiert der Algorithmus nach einer Sekunde. Für das höhere Muster $\beta_1 = 4$ werden etwa 30 Sekunden benötigt, obwohl a_1 nun größer ist.

Insgesamt führen diese Überlegungen zu folgenden Problemen:

- Besagte Muster müssen nicht in a_1 liegen.
- Führungsbit p kann kleiner als $n - \log_2(m)$ sein, z.B. bei $(12, 8)$, $(12, 9)$.
- Muster soll gleiche Führungsbitposition für a_2 ermöglichen.

$$\begin{array}{r|cccccc|} a_1 & . & . & 1 & . & 1 & . \\ a_2 & . & . & 1 & 1 & . & . \\ a_3 & . & 1 & . & . & . & . \end{array} \quad \begin{array}{r|cccccccc|} a_1 & . & . & 1 & . & 1 & 1 & . & . \\ a_2 & . & . & 1 & 1 & . & . & 1 & . \\ a_3 & . & 1 & . & . & . & . & . & . \end{array} \quad \begin{array}{r|cccccccccc|} a_1 & . & . & 1 & . & 1 & 1 & 1 & . & . & . \\ a_2 & . & . & 1 & 1 & . & . & . & 1 & 1 & . \\ a_3 & . & 1 & . & . & . & . & . & . & . & . \end{array}$$

Tabelle 2.3: Belegung der ersten drei Zeilen nach Musterwahl in a_1

In Tabelle 2.3 wird a_1 und a_2 mit einem Muster vorbelegt. Die zweite Zeile erhält die gleiche Mustergröße wie a_1. Die Folgezeilen werden bezüglich der Rechteckfreiheit und der Zeilensortierung inkrementiert. Diese Annahmen scheinen äußerst restriktiv in Anbetracht der Anzahl der Lösungswege aus Satz 2.1. Der Algorithmus ist auch nicht mehr exakt. Es ist nicht klar, wie groß die Abweichungen der Führungsbits und der Musterwahl sein können. Die simple Startlösung $A = (1, \dots, 1)$ ist jedoch zu grob und durchläuft zuviele Lösungen. Folgende Tabelle 2.4 gibt einen Ausblick, welches Potenzial durch das Wissen über die erste Zeile entsteht.

	NAIV		LSG.	FIXIERT		LSG.	β_1
$\mathrm{maxrf}(7, 7)$	3 ms	(3 ms)	1	0,5 ms	(0,1 ms)	1	3
$\mathrm{maxrf}(14, 7)$	11 s	(2 s)	27 641	400 ms	(1,7 ms)	282	2
$\mathrm{maxrf}(8, 8)$	1 s	(72 ms)	16 596	14 ms	(1 ms)	128	3
$\mathrm{maxrf}(9, 9)$	24 s	(9 s)	4464	354 ms	(171 ms)	144	3
$\mathrm{maxrf}(10, 10)$	17 min	(34 s)	32 838	1,6 s	(1,6 s)	1	3
$\mathrm{maxrf}(11, 11)$	7 d 4 h	(9 h)	1 168 996	34 min	(6 min)	432	4
$\mathrm{maxrf}(12, 12)$	–	–	–	12 h 28 min	(5 h 7 min)	72	4

Tabelle 2.4: Anzahl Optimallsg. und Laufzeiten (Intel X5650 2.67 GHz, single thread)

Tabelle 2.4 listet für einige Dimensionen die Laufzeiten des Overflow-Algorithmus und die Anzahl der gefundenen Optimallösungen auf, wobei zwei Varianten getestet wurden. Als untere Schranke wurde stets $2 \cdot n + m - 3$ verwendet (Gleichung 3.8). Die naive Variante lässt den Algorithmus mit der Startbelegung $A = \{1, \dots, 1\}$ den kompletten Zahlenbereich durchlaufen. Die zweite Variante fixiert die erste Zeile a_1 mit einem spezifischen Muster der Größe β, siehe Tabelle 2.3, und dem Führungsbit $p = \lfloor n - \log_2(m) \rfloor$. Die zweite Zeile erhält das nächst kleinere Muster, falls $\beta > 2$.

Die Angaben in den Klammern notieren die benötigte Zeit für das erstmalige Auffinden der Optimallösung. In der vierten und siebten Spalte stehen die Anzahl aller gefundenen Optimallösungen. Hier wird deutlich, dass die naive Variante zuviele Lösungen durchläuft. Die zweite Variante nutzt jedoch Annahmen, die bei anderen Dimensionen auch zu nicht optimalen Ergebnissen führen. Die Wahl der Mustergröße für a_1 und a_2 zeigt die letzte Spalte mit $\beta_1 = \lfloor \beta + 0.3 \rfloor$.

Beide Varianten verwenden die vorausschauende Inkrementierung, siehe INCREMENT-Prozedur 5.5, siehe auch Tabelle 2.7.

2.3 Prozeduren

Wir wollen nun die Einzelheiten der Prozeduren des Overflow-Algorithmus diskutieren. Der Pseudo-Code findet sich im Unterabschnitt 5.1. Der Algorithmus besteht aus folgenden Prozeduren:

Prozedur	Beschreibung
MAIN(m, n, z)	Hauptroutine. Lösung des Zarankiewicz-Problems $Z_2(m, n)$ mit der unteren Schranke z
INCREMENTFIRSTROW$(A, B, t_{\max})$	Inkrementiert erste Zeile a_1 in A. B ist die Bit-Akkumulationsmatrix und $t_{\max}$ das derzeitige Maximum.
INCREMENTROW$(i, A, B, t_{\max})$	Inkrementiert Zeile $i \in \{2, \ldots, m-1\}$.
CONFIGUREROWS$(i, a, t, A, B, t_{\max})$	Passt Zeilen $i+1, \ldots, m-1$ an, nachdem Zeile i erhöht wurde. Falls $i+1$ überläuft, wird i weiter erhöht. a ist temporär neuer Wert für a_i. t ist die Bitanzahl von a.
INCREMENT(a, a_i, t)	Inkrement-Operation für a bezüglich a_i.
RESETTOLOWERROW(k, a, a_{k-1}, t)	Zeile k wird zurückgesetzt auf $k-1$.
INIT(A, B)	Initialisiere A und B.

Tabelle 2.5: Prozeduren des Overflow-Algorithmus

Im Folgenden werden einige der Prozeduren ausführlicher erörtert.

Prozedur 5.1 MAIN():
Zeile 3:
Die MAIN-Funktion initialisiert zunächst die Matrizen A, mit der oben genannten Startbelegung, und B. Letztere akkumuliert die Anzahl der Bits, sodass $b_1 = c(a_1)$ und $b_m = c(A) = \sum_{i=1}^{m} c(a_i)$, wobei $c(\cdot)$ die Anzahl der Bits misst. (Siehe auch INIT-Prozedur 5.7.)

Zeile 9:
Nun wird versucht die letzte Zeile solange zu inkrementieren, bis sie „überläuft". Der Startwert des Zahlenbereiches für a_m ist $\{\alpha, \ldots, (101\ldots1)_2\}$. Gilt $c(a_{m-1}) = 1$, so ist $\alpha = a_{m-1} + 1$. Hat a_{m-1} mehr als ein Bit, so würde die Gleichheit $a = a_{m-1}$ zur Kollision führen (verletzt Rechteckbedingung). Der nächstmögliche Wert wird in der INCREMENT-Prozedur 5.5 ermittelt. Diese Funktion erhöht a, sodass a nicht mehr mit a_{m-1} kollidiert, siehe Tabelle 2.6:

a_{m-1}	0	1	0	0	1	0		a_{m-1}	0	1	0	1	0	1
a	0	1	0	1	0	0		a	0	1	1	0	0	0

Tabelle 2.6: 2 Fälle beim Inkrementieren von $a =$ INCREMENT(a_{m-1})

Für das Auswählen des zweiten Bits im ersten Fall kann in C die Bitoperation `lsb = a & -a;` bzw. `lsb = a & (~a+1);` verwendet werden. Im zweiten Fall wird das zweite Bit ebenfalls nach links verschoben, allerdings werden die nachfolgenden Bits ignoriert.

Zeile 11:
Nun wird a, also der Wert der letzten Zeile, solange erhöht, bis es die Obergrenze $(101\ldots1)_2$ erreicht. In jedem Schritt wird eine neue Lösung generiert, die auf Zulässigkeit geprüft wird. Der erste Test geschieht per HASRECTANGLE. Hat a einen Bit mit a_{m-1} gemeinsam, so wird die Lösung verworfen und die Schleife fortgesetzt. Die Prozedur HASRECTANGLE könnte in C wie folgt aussehen:

```
int has_rectangle(unsigned int a, unsigned int b){
unsigned int c = a & b;
  if ( c && (c & (~c + 1)) != c )
          return 1;
        return 0;
}
```

Ist eine potenziell neue Optimallösung gefunden, wird per VALIDATEFORROW der abschließende Test für den Wert a gemacht. Gibt es keine Kollisionen, so wird die gefundene Optimallösung in C gespeichert.

Zeile 26:
Nachdem der Zahlenbereich in Zeile $i = m$ durchlaufen wurde, wird nun versucht, vorliegende Zeilen zu inkrementieren. Das geschieht per INCREMENTROW-Prozedur 5.3. Geschieht in Zeile $i = 2$ ein Überlauf, dann liefert die besagte Prozedur **false** und es wird die erste Zeile fortgesetzt. Dafür ist INCREMENTFIRSTROW-Prozedur 5.2 verantwortlich. Liefert auch diese Prozedur **false** , dann ist die Hauptroutine beendet.

Prozedur 5.3 INCREMENTROW():
Diese Prozedur erhöht den Wert in der Zeile $i \in \{2, \ldots, m-1\}$ und konfiguriert entsprechende Folgezeilen (per CONFIGUREROWS-Prozedur 5.4).

Zeile 2:
Falls $a_i = (101\ldots1)_2$ ist, sind keine weiteren Lösungen mehr in dieser Verzweigung zu betrachten. Denn wird a_i inkrementiert, so erhalten wir $(110\ldots)_2$ und in der nächsten Zeile kollidiert jeder Wert $a \geq a_i$ mit a_i innerhalb des geforderten Zahlenbereichs. **Zeile 5:**
Wir wollen Zeile a_i inkrementieren, aber lohnt sich das überhaupt? Wieviele Einsen können denn überhaupt in der nachliegenden Teilmatrix, bestehend aus den Folgezeilen $i+1, \ldots, m$, gesetzt werden und erreichen wir damit überhaupt das bereits bekannte Maximum $t_{\max}$?

Dafür wird ein Teilproblem $(m-i-1, n)$ gelöst, dass per CLIMIT aufgerufen wird. Zwei Möglichkeiten bieten sich zur Ergebnisfindung an:

1. Schätze $\operatorname{maxrf}(m-i-1, n)$ durch eine obere Schranke ab!
2. Verwende $\operatorname{maxrf}(m-i-1, n)$ als bereits bekannte Lösung!

Für **Fall** 1 kann die obere Schranke von I. Reiman für das Zarankiewicz-Problem $Z_2(\mu, \eta)$ direkt berechnet werden (Rei58), siehe auch Satz 3.3:

$$\operatorname{maxrf}(m, n) = Z_2(m, n) - 1 \leq \frac{n}{2} + \sqrt{nm(m-1) + \frac{n^2}{4}} =: HC(m, n)$$

CLIMIT(k) ist damit: $\min\{HC(m-k-1, n), HC(n, m-k-1)\}$.

Für **Fall** 2 wird das Teilproblem als bereits gelöst vorausgesetzt. Kennt man also die Lösungen für kleinere Dimensionen des Zarankiewicz-Problems, so können diese für größere Dimensionen herangezogen werden. Hier entstehen schärfere Bedingungen, sodass eher die Verzweigung verworfen werden kann. Das hat deutliche Auswirkungen auf die Laufzeit des Algorithmus:

	VARIANTE 1		VARIANTE 2	
	Fall 1	*Fall 2*	*Fall 1*	*Fall 2*
$\mathrm{maxrf}(8,8)$	3 s	1 s	19 ms	8 ms
$\mathrm{maxrf}(9,9)$	5 min 49 s	22 s	2 s	765 ms

Tabelle 2.7: Fallbetrachtung Laufzeit Overflow-Algorithmus für $\mathrm{maxrf}(k,k)$ Problem.

Variante 1 stellt den naiven Teil dar, in dem der Algorithmus den kompletten Zahlenbereich durchläuft. Variante 2 verwendet den Optimierungsansatz aus Unterabschnitt 2.2 (a_1 mit spezifischem Muster fixiert mit Führungsbit $p = \lfloor n - \log_2(m) \rfloor$).

Prozedur 5.4 CONFIGUREROWS():
Nachdem ein gültiger Wert für a_i gefunden wurde, sind die Folgezeilen $i+1, \ldots, m-1$ anzupassen. Überläufe in diesem Bereich müssen ebenso behandelt werden. Tritt in $i+1$ der Überlauf ein, so wird a_i weiter erhöht, bis auch hier irgendwann der Überlauf entsteht. In dem Fall wird **false** zurückgegeben.

Zeile 9:
Falls die Verzweigung zu keinem besseren Ergebnis führt, so ist die aktuelle Zeilenkonfiguration um eine Ebene zurückzusetzen ($k \leftarrow k-1$). Das geschieht mittels der RESETTOLOWERROW-Prozedur 5.6.

3 Konzeption Slot-Algorithmus

Der Overflow-Algorithmus liefert in größeren Dimensionen nicht mehr akzeptable Laufzeiten, sodass weitere strukturelle Überlegungen und gezielte Vereinfachungen notwendig sind, um die Komplexität wesentlich zu reduzieren. Bei größeren Dimensionen werden wir jedoch zunehmende Einbußen in der Genauigkeit hinnehmen müssen.

Der Slot-Algorithmus ist eine Umsetzung der strukturellen Erkenntnisse aus (SP12b) von B. Steinbach. B. Steinbach verwendet eine Einteilung der Matrix in sogenannte Slots, welche durch einen speziellen Matrixkopf beschrieben werden. Bezüglich des Matrixkopfes wird ein Repräsentant der $n!m!$ Matrixkombinationen gebildet. Der Matrixkörper wird entsprechend der Slot-Konfiguration mit weiteren Mustern aufgefüllt, die sich aus einer verbleibenden Bitmuster-Menge ergeben.

3.1 Matrixkopf und Slots

Wir beginnen mit dem Matrixkopf und der Definition der Slots.

Definition 3.1. *Gegeben sei die Dimension (m, n) für das Zarankiewicz-Problem und es gelte weiter $m \geq n \geq 3$. Ferner sei $P(n)$ die Menge der Partitionen bezüglich n:*

$$P(n) := \left\{ (v_1, \dots, v_k) : \sum_{j=1}^{k} v_j = n,\ v_j \geq v_{j+1} > 0,\ 0 < k \leq n \right\}$$

Die Zahl v_j sei als Slot(breite) bezeichnet.

Die Slots beschreiben im Matrixkopf die Breite durch v_j nebeneinanderstehenden Einsen:

1	1	1					(3)
·	·	·	1	1			(2)
·	·	·	·	·	1	1	(2)

Abbildung 3.17: Sloteinteilung $(3, 2, 2)$ Matrixkopf

Jede Lösung ließe sich so permutieren, dass wir in der ersten Zeile die maximale Anzahl an Bits festhalten und in den darauf folgenden diese Anzahl nicht überschreiten. Diese maximale Anzahl ist v_1 und beschreibt den ersten Slot. Jetzt verbleiben $n - v_1$ Spalten, in denen wir wieder unter Beibehaltung des Slots v_1 permutieren können, sodass in der zweiten Zeile die nächstkleinere Slotgröße v_2 entsteht. Insgesamt muss für die zu bildende Matrix gelten, dass die Anzahl der Bits b_i in Zeile a_i größer oder gleich die der Zeile a_{i+1} sind, also $b_i \geq b_{i+1}$, $i \in \{1, \dots, n-1\}$.

Nachdem alle Slots gebildet wurden, entsteht eine Partition, wobei im Slotfenster weitere Einsen eingefügt werden können. Ein Slotfenster beschreibt die nachfolgenden Zeilen innerhalb eines Slots im Matrixkopf. Hier können je Slot-Zeile maximal nur eine Eins hinzugefügt werden, andernfalls kommt es zur Rechteckbildung im Slot. Aufgrund der Möglichkeit innerhalb eines Slots die Spalten permutieren zu können, lassen sich diese Einserpositionen im Slotfenster einschränken.

In Abbildung 3.18 sind drei verschiedene Slotfensterbelegungen zu sehen. Jede weitere Slotfensterbelegung bezüglich der Sloteinteilung $(3, 2, 2)$ ließe sich durch Spaltenpermutationen auf eine der drei Belegungen zurückführen.

Die Bitanzahl der letzten Zeile a_k im Matrixkopf ist maßgebend, denn alle nachfolgenden Zeilen dürfen diese Bitanzahl nicht mehr überschreiten. In Abbildung 3.17 dürfen nur noch maximal zwei

1	1	1				
1	·	·	1	1		
1	·	·	·	·	1	1

1	1	1				
1	·	·	1	1		
·	1	·	·	·	1	1

1	1	1					
1	·	·	1	1			(3)
·	·	·	1	·	1	1	(3)

(3) (3) (3)

Abbildung 3.18: Sloteinteilung $(3, 2, 2)$ Matrixkopf und Slotfensterbelegung

Bits je Zeile im Matrixkörper verwendet werden, in Abbildung 3.18 stehen immerhin drei Bits zur Verfügung.

Die zusätzlich gesetzten Einsen führen zu Überlappungen mit anderen Slots, jedoch ermöglichen die Überlappungen eine höhere Bitanzahl in der ausschlaggebenden Zeile a_k.

1	1	1				
1			1	1		
1					1	1
1			X	X		
1					X	X

1	1	1				
1			1	1		
	1				1	1
1			X	X		
	1				X	X

1	1	1				
1			1	1		
			1		1	1
1			X	X		
			1		X	X

1	1	1			
1			1	1	
	1		1		1
1			X	X	
			1		X
	1		X		X

Abbildung 3.19: Slotüberlappungen und verbotene Muster im Matrixkörper

In Abbildung 3.19 sind die verbotenen Muster zu sehen, die durch die zusätzlichen Einsen in den Slotfenstern entstehen. Die letzte Matrix zeigt ein weiteres Muster in der Sloteinteilung $(4, 3, 2)$. Ziel ist es also, die Bitanzahl b_k so groß wie möglich zu halten, während im Gegensatz dazu die Überlappungen so gering wie möglich sein dürfen. Überlappungen verringern die Anzahl der Muster, die im Körper noch gesetzt werden können.

Abhängig von der Bitanzahl b_k und der Anzahl k der Slots können nun im Körper zunächst Muster der Länge $\beta \leq \min(b_k, k)$ eingesetzt werden. In Abbildung 3.19 wären 3er-Muster zu verwenden. Wenn alle 3er-Muster verbraucht sind, würde man 2er-Muster und gegebenfalls weitere Einsen zum Auffüllen verwenden.

Der Algorithmus geht nun den Weg, alle noch möglichen Zweier-Muster als Basis für höhere Muster zu nehmen. So gibt es eine Art Rezeptbibliothek für die höherwertigen Muster, wie diese sich aus niedrigeren Mustern zusammensetzen. Ein Dreier-Muster kann aus drei Zweiern zusammengebaut werden. Es sind gerade jene, die mit dem zu bauenden Dreier Rechtecke bilden:

	1		1		1	
	1		1			
	1				1	
			1		1	

	1		1		1		1	
	1		1		1			
	1						1	
			1				1	
					1		1	

	1		1		1		1		1	
	1		1		1		1			
	1								1	
			1						1	
					1				1	
							1		1	

Abbildung 3.20: Rezepte zur Musterbildung, $\beta = 3,\ 4,\ 5$

In Abbildung 3.20 sind die Rezepte für Dreier, Vierer und Fünfer zu sehen. Ein Fünfer benötigt einen Vierer und vier Zweier. Insgesamt aggregiert ein Fünfer somit 10 Zweier. Allgemein werden $\binom{\beta}{2}$ Zweier für die Mustergröße β verbraucht. Die Bilanz nach dem Aggregieren ist bezogen auf β Zeilen negativ, sodass sich das Aufschichten bei großer Zeilenanzahl nicht lohnt. Ein Dreier verbraucht drei Zweier. In der Bilanz ergibt das einen Bit weniger auf drei Zeilen gesehen.

$$\begin{aligned} 1 \times \text{Dreier} + 2 \times \text{Einser} = 5 &\quad < \quad 6 = 3 \times \text{Zweier} \\ 1 \times \text{Vierer} + 3 \times \text{Einser} = 7 &\quad < \quad 9 = 1 \times \text{Dreier} + 3 \times \text{Zweier} \\ 1 \times \text{Fünfer} + 4 \times \text{Einser} = 9 &\quad < \quad 12 = 1 \times \text{Vierer} + 4 \times \text{Zweier} \end{aligned}$$

Nachdem also der Kopfteil gebildet wurde, müssen noch hinreichend viele Zweier ($> m - k + 3$) übrig sein, um aufschichten zu können. Ziel ist es, die Zweier optimal zu Dreier, Vierer, etc. zusammenzubauen, ohne dass zuviele Zweier verbraucht werden.

1	1	1		
			1	1

1				1
1			1	
	1			1
	1		1	
		1		1
		1	1	

1	1	1		
1			1	1

	1			1
	1		1	
		1		1
		1	1	

Abbildung 3.21: Verfügbare Zweier nach Slotbelegung

In Abbildung 3.21 sind die verfügbaren Zweier augelistet, die nach der Kopfbildung übrig sind. Aus diesen können je nach Zeilen- und Slotanzahl höherwertige Muster gebildet werden. In besagter Abbildung schränken die zwei Slots die Muster bereits ein.

3.2 Höherwertige Mustergenerierung

Nachdem der Kopf erzeugt wurde, erfolgt das Anlegen der Zweierliste mit folgender Sortierung. Die Zahlen (Bitvektoren) mit demselben Führungsbit werden aufsteigend geordnet. Die Führungsbit-Gruppen werden absteigend sortiert. Der Algorithmus baut zuerst Dreier und wenn noch genügend Zweier vorhanden sind, werden Vierer gebildet. Jenachdem wird dabei auf die bereits vorhandenen Dreier zurückgegriffen und weiter aufgeschichtet. Es zeigte sich, dass bereits die Lösung zu 11×11 nicht mehr optimal war, da „falsche“ Dreier die nötige Menge an Vierern verhinderte. Dieser Umstand führte zu einem weiteren Viereralgorithmus mit besserer Qualität.

3.2.1 Dreierbildung

Wir erhalten entsprechend der Sortierung beispielsweise folgende Liste:

1				1	17
1			1		18
	1			1	9
	1		1		10
		1		1	5
		1	1		6
			1	1	3

Abbildung 3.22: Zweier-Sortierung

1				1
1			1	
	1			1
	1		1	
		1		1
		1	1	
			1	1

1	0	0	0	1
1	0	0	1	0
0	0	0	1	1

Abbildung 3.23: Dreier-Suchalgorithmus

Um Dreier zu bilden, brauchen wir dem Rezept entsprechend drei Zweier, siehe Abbildung 3.20. In jeder Spalte werden zwei Einsen benötigt, sodass wir innerhalb einer Führungsbit-Gruppe das erste Paar wählen. Der Algorithmus nimmt dieses Paar und bildet den dritten noch zu suchenden Zweier durch das Exklusiv-Oder. Wird der dritte Zweier gefunden, so werden alle drei aus der Zweierliste entfernt und der erzeugte Dreier in den Matrixkörper aufgenommen. Für die nachfolgenden Zweier wird analog verfahren, bis man in der letzten Gruppe angekommen ist.

Die Frage ist, wieviele Dreier werden benötigt. Es können maximal $m-k$ Zeilen mit Dreiern gefüllt werden. Wenn jedoch die Zweier nicht ausreichen, kann der Fall eintreten, dass die Zweier komplett aufgebraucht wurden und immer noch Zeilen übrig bleiben, die jetzt mit Einsen gefüllt werden müssten. Hier wären zuviele Zweier aggregiert worden und wir erhalten eine schlechtere Lösung als eigentlich nötig. Per Lösung eines Ungleichungssystems können wir eine Schranke angeben, die sicherstellt, dass genügend Zweier übrig bleiben.

$$\left.\begin{array}{rcl} x+y & \geq & m-k \\ 3x+y & = & z \\ x & \leq & m-k \end{array}\right\} \Rightarrow x \leq \min\left(m-k, \frac{z-m+k}{2}\right) \tag{3.7}$$

Mit z sei die Anzahl der verfügbaren Zweier bezeichnet, y und x seien die Anzahl der zu verwendenden Zweier bzw. Dreier und $m-k$ ist die Größe des Matrixkörpers (Zeilenanzahl).

3.2.2 Viererbildung

Man könnte nun analog zur Dreierbildung vorgehen und nach Auswahl eines Dreiers die passenden Zweier suchen, um entsprechend dem Rezept in Abbildung 3.20 den Vierer herzustellen. Dazu wird auf die bereits erzeugte Dreierliste zurückgegriffen und pro Element versucht, die passenden Zweier zu finden. Auch hier ergibt sich wieder eine obere Schranke, um nicht zuviele Zweier zu aggregieren.

$$\left.\begin{array}{rcl} x+(z-3v) & \geq & m-k \\ v & \leq & x \end{array}\right\} \Rightarrow v \leq \min\left(x, \frac{x+z-m+k}{3}\right)$$

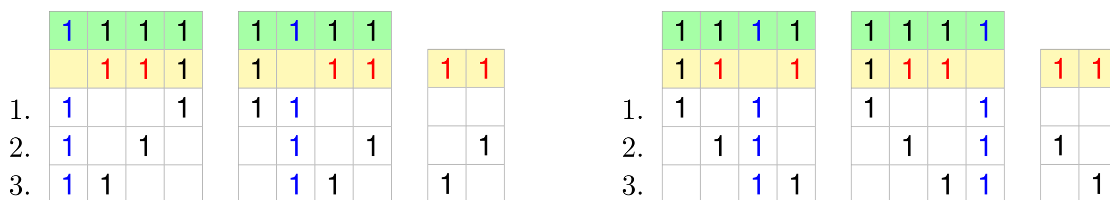

Abbildung 3.24: Vierer-Suchalgorithmus Fall 1 Abbildung 3.25: Vierer-Suchalgorithmus Fall 2

Entsprechend der Reihenfolge in der Zweierliste können nur zwei unterschiedliche Fälle eintreten. In der Abbildung 3.24 ist das Viererbit blau markiert und steht vor der Bitpaar-Kombination (rot), die gesucht werden soll. Wird also ein Zweier gemäß dem ersten Fall gefunden, der ein Bit mit dem Dreier gemeinsam hat, dann werden die zwei weiteren Zweier konstruiert und in der Liste gesucht. Es würde hier ausreichen, innerhalb der Führungsbitgruppe (blau) zu suchen.

Der zweite Fall gemäß Abbildung 3.25 tritt ein, wenn das Viererbit nicht vor der Bitpaar-Kombination (rot) steht.

Der zweite Zweier wird in der Liste nach dem ersten gesucht und falls er gefunden wurde, wird im verbleibenden Teil der Liste der dritte Zweier gesucht. So wird je mögliche Vierer-Kombination

maximal nur einmal die Liste durchsucht.

Mit diesem Vierer-Algorithmus werden jedoch abhängig von den gegebenen Dreiern nicht alle Vierer gefunden, die bezüglich der Slotvorgabe möglich wären. So findet der Algorithmus zum 11×11 Problem nicht die Optimallösung.

1	1	1	1							
1				1	1	1				
	1			1			1	1		
		1			1		1		1	
1								1	1	1
			1			1	1			1
	1				1					1
	1					1			1	
		1		1						1
		1				1		1		
			1	1					1	

Abbildung 3.26: 11×11 Optimallösung

1	1	1	1							
1				1	1	1				
	1			1			1	1		
		1			1		1		1	
1								1	1	1
	1					1				1
		1		1						1
		1				1		1		
			1				1			1
			1			1			1	
			1		1			1		

Abbildung 3.27: 11×11 Suboptimale Lösung

In Abbildung 3.27 findet der Vierer-Algorithmus keinen Vierer, wodurch statt 39 nur 38 Einsen gefunden werden. In Abbildung 3.26 ist die Optimallösung zu sehen, die durch einen alternativen Vierer-Algorithmus gefunden wurde, welcher aus einer Liste aller noch möglichen Dreiermuster sukzessive Vierer baut und dabei die Anzahl der verbleibenden Dreier und Zweier prüft.

3.3 Optimierung und Auswertung

3.3.1 Slotfensterkombinationen

Der Flaschenhals ist die Generierung der Slotfensterbelegungen je Partition. Der Algorithmus startet zunächst bei $k = 2$ Slots. Denn für $k = 1$ lässt sich die lokale Optimallösung in eine bessere mit $k = 2$ Slots transformieren, sodass wir eine neue untere Schranke erhalten für $m \geq n \geq 3$:

$$\mathrm{maxrf}(m, n) \geq m - 1 + 2 \cdot (n - 1) \tag{3.8}$$

Die dazugehörige Lösung ist in Abbildung 3.29 zu sehen.

1	1	1	1	1
1				
1				
1				
1				
1				
1				

Abbildung 3.28: Lokale Optimallösung für $k = 1$

1	1	1	1	
1				1
	1			1
		1		1
			1	1
				1
				1

Abbildung 3.29: Verbesserung mit $k = 2$ Slots

Je größer die Slotanzahl, desto mehr Kombinationen müssen unter den Slotfenstern erzeugt werden. Für eine gegebene Partition mit $k = 2$ Summanden (Slots) gibt es nur zwei Belegungen, da es nur ein Slotfenster gibt. Für $k = n$ gibt es nur eine Belegung, die keinen zusammenhängenden Graphen konstruiert, da pro Zeile nur ein Bit gesetzt werden darf.

Der implementierte Algorithmus verwendet eine leicht „vereinfachte" Kopferzeugungsfunktion, die nicht alle Bedingungen berücksichtigt und damit auch unzulässige Slotfensterbelegungen kreiert. Je größer die Slotanzahl, desto höher wird die Anzahl unzulässiger Kopfbelegungen, die rausgefiltert werden müssen. Jedoch können wir auf eine hohe Slotanzahl verzichten, denn hier findet der Algorithmus keine Optimallösungen.

k	UNGEFILTERT	GEFILTERT	LÖSUNGEN
2	8	8	0
3	30	30	0
4	150	78	3
5	894	109	0
6	8422	259	0
7	129115	312	0
8	1563762	609	0

Tabelle 3.8: Anzahl Kopfbelegungen und Lösungen für 9×9

In Tabelle 3.8 steigt die Menge unzulässiger Kopfbelegungen deutlich bei hoher Slotanzahl. Die Optimallösung für 9×9 liegt jedoch bei $k = 4$ Slots. Folgende Tabelle 3.9 zeigt auf, bei welcher Slotanzahl erstmalig die Optimallösung gefunden wurde.

$m \setminus n$	4	5	6	7	8	9	10	11
4	2							
5	2	2						
6	2	2	3					
7	2	2	3	3				
8	2	2	2	3	3			
9	2	2	3	3	3	4		
10	2	3	2	3	3	4	4	
11	2	3	2	3	3	4	4	5

Tabelle 3.9: Kleinste Slotanzahl bezüglich Optimallösung

Entsprechend der Ergebnisse verwenden wir folgende Schranken für die Slotanzahl k unter der Bedingung $m \geq n \geq 3$:

$$2 \leq k \leq \min\left(\left\lfloor \frac{m}{3} \right\rfloor + 2, n - 1\right) \tag{3.9}$$

Die obere Schranke ist nur empirischer Natur, weitere Ergebnisse siehe Tabelle 3.10.

3.3.2 Lösungen und Laufzeit

Eine Optimallösung wird eine optimale Unterbringung der Zweier erzielen mit einer optimalen Verwendung von Dreiern, Vierern, etc. Der im Rahmen dieser Seminararbeit implementierte Algorithmus erzeugt Dreier und Vierer, allerdings nicht immer optimal, sodass zuviele Zweier übrig bleiben.

$m \setminus n$	5	6	7	8	9	10	11	12	13	14	15	16
5	12 0 2 [3]											
6	14 0 2 [4]	16 1 3 [4]										
7	15 1 2 [4]	18 0 3 [4]	21 0 3 [4]									
8	17 0 2 [4]	19 0 2 [4]	22 1 3 [4]	24 3 3 [4]								
9	18 -1 2 [4]	21 0 3 [5]	24 0 3 [5]	26 2 3 [5]	29 3 4 [5]							
10	20 0 3 [4]	22 0 2 [5]	25 0 3 [5]	28 1 3 [5]	31 2 4 [5]	34 3 4 [5]						
11	21 -1 3 [4]	24 0 2 [5]	27 0 3 [5]	30 0 3 [5]	33 1 4 [5]	36 2 4 [5]	39 4 5 [5]					
12	22 -2 3 [4]	25 -1 2 [5]	28 0 3 [6]	32 0 5 [6]	36 0 6 [6]	38 (1) 1 4 [6]	41 (1) 4 4 [6]	43 (2) 8 5 [6]				
13	23 -3 3 [4]	27 0 3 [5]	30 0 3 [6]	33 0 3 [6]	37 1 6 [6]	40 0 4 [6]	42 (2) 5 3 [6]	46 (2) 6 6 [6]	47 (5) 12 4 [6]			
14	24 -4 3 [4]	28 -1 3 [5]	31 0 2 [6]	35 0 3 [6]	39 0 6 [6]	42 2 6 [6]	44 (1) 4 3 [6]	49 2 6 [6]	51 (2) 8 6 [6]	51 (5) 17 4 [6]		
15	25 -5 3 [4]	30 0 3 [5]	33 0 3 [6]	36 0 3 [7]	40 0 4 [7]	44 1 6 [7]	47 3 6 [7]	51 1 6 [7]	53 (2) 7 5 [7]	54 (4) 14 4 [7]	56 (4) 22 5 [7]	
16	26 -6 3 [4]	31 -1 3 [5]	34 0 3 [6]	38 0 3 [7]	42 0 3 [7]	46 0 6 [7]	50 0 6 [7]	53 0 6 [7]	55 (2) 8 5 [7]	57 (3) 12 5 [7]	58 (5) 20 5 [7]	61 (?) 27 6 [7]

Tabelle 3.10: Ergebnisse Slot-Algorithmus - Abweichungen, Zweierüberschuss, Slots

In der ersten Zeile einer Zelle steht die Anzahl gefundener Kanten und gegebenfalls rot in Klammern die Abweichung von der Optimallösung. In der zweiten Zeile steht die Anzahl überschüssiger Zweier. Negative Werte entsprechen betragsmäßig der Anzahl der Einser-Muster. Für $(9, 5)$ existiert ebenfalls eine Optimallösung mit 1 überschüssigen Zweier. In der dritten Zeile steht die Slotanzahl der gefundenen Lösung. In Klammern ist die obere Schranke nach Gleichung 3.9 festgehalten.

DIMENSION	LAUFZEIT	OPTIMAL	ERGEBNIS
$\mathrm{maxrf}(10,10)$	2 ms	Ja	34
$\mathrm{maxrf}(11,11)$	3 ms	Ja	39
$\mathrm{maxrf}(12,12)$	28 ms	Nein	43 statt 45
$\mathrm{maxrf}(21,12)$	9 min	Ja	63

Tabelle 3.11: Laufzeiten des Slot-Algorithmus (Intel X5650 2.67 GHz, single thread)

In Tabelle 3.11 sind die Zeiten des Slot-Algorithmus notiert, wobei die Slotanzahl nach Gleichung 3.9 und durch $k > 1$ beschränkt ist. Ohne die obere Schranke würde der Algorithmus schon für das 10×10 Problem 53 Sekunden benötigen.

Bei 21×12 bewirkt die hohe Zeilenanzahl eine schlechtere obere Schranke für die Slots. k liegt hier zwischen 2 und 9, obwohl die Optimallösung nur $k = 4$ Slots benötigt. Für $k \in \{2, \ldots, 8\}$ braucht der Algorithmus nur noch 8 Sekunden.

Solange nur Dreier und Vierer-Muster gebraucht werden, liefert der Algorithmus nützliche Ergebnisse. Für das 21×12 Problem mit der Sloteinteilung $(4, 4, 4)$ gibt es insgesamt 2 nichtisomorphe Dreierkonfigurationen mit 16 Dreiern, wobei ein Dreier in drei Zweier aufgelöst wird. Der Verbrauch der Zweier wird insgesamt optimal ausgenutzt. Verwendet werden $3\times$Vierer, $15\times$Dreier und $3\times$Zweier: $\binom{12}{2} - 3 \cdot \binom{4}{2} - 15 \cdot 3 - 3 = 0$. In anderen Sloteinteilungen konnte der Algorithmus keine weiteren Optimallösungen finden.

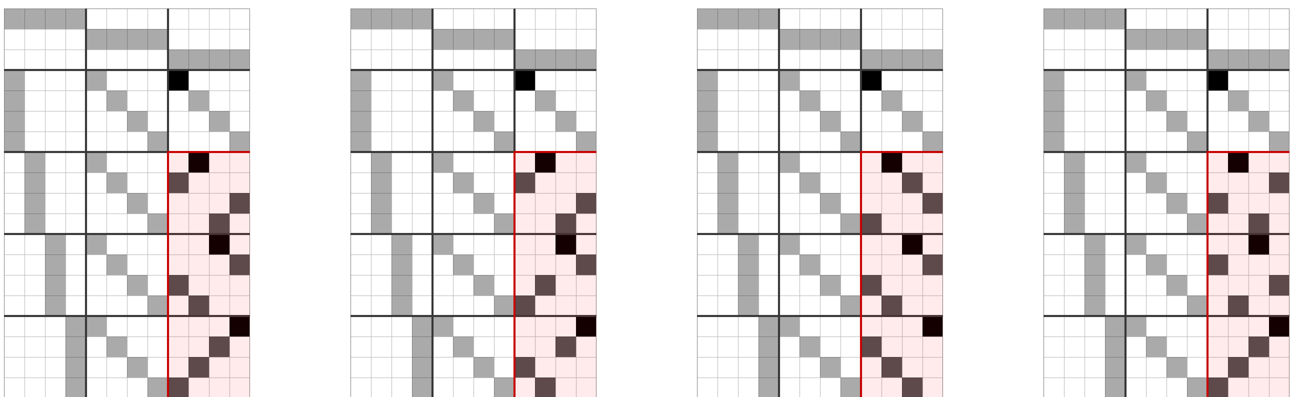

Abbildung 3.30: 16 Dreier - 4 Lösungsmuster

In Abbildung 3.30 sind vier Lösungsmuster der 16 Dreier für die Sloteinteilung $(4, 4, 4)$ zu sehen. Sie entsprechen der Darstellung des lateinischen Quadrates der Ordnung 4, wobei hier durch Umlabeln (Spaltenvertauschung in Matrix) die letzten drei zu einer Isotopieklasse gehören ((Sto10)).

		0	1	2	3
Block 1	0	0	1	2	3
Block 2	1	1	0	3	2
Block 3	2	2	3	0	1
Block 4	3	3	2	1	0

	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	1	0
3	3	2	0	1

	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

	0	1	2	3
0	0	1	2	3
1	1	3	0	2
2	2	0	3	1
3	3	2	1	0

Abbildung 3.31: Letzte Bitposition als Verknüpfungstafel (lateinische Quadrate der Ordnung 4)

3.3.3 Weitere Resultate

Wir haben bereits in Definition 3.1 die Annahme $m \geq n$ getroffen, welche wir jetzt genauer begründen. Der implementierte Algorithmus liefert freilich auch Lösungen zu (n, m) statt (m, n). Wenn jedoch die Spaltenanzahl größer ist als die Zeilenanzahl, müssen mehr Slots verwendet werden, was zu höherwertigen Mustern führt. Da wir nur Dreier und Vierer implementiert haben, ist demnach die kleinere Slotanzahl zu bevorzugen, da sonst schlechtere Ergebnisse erzielt werden.

Weiterhin liegt die Schwierigkeit in der Kopferzeugungsfunktion, die jedoch nur von der Spaltenanzahl abhängt. Hier müssen Partitionen von n und Kombinationen zwischen den Slotfenstern gebildet werden. Im Gegensatz dazu geschieht das Aufschichten in Polynomialzeit.

Ferner betrachten wir beim Aufschichten die verbleibende Anzahl von Zweiern. Das ergibt nur für $m \geq n$ einen Sinn, da $\binom{n}{2}$ Zweier auf m Zeilen verteilt werden müssen, sodass der Rest gegen Null geht. Im Falle $n \geq m$ wäre das Ziel, so hohe Muster wie möglich auf den wenigen Zeilen zu bilden. Jedoch beschränken die Slots die Größe der Muster, sodass viele Zweier übrig bleiben.

Durch den Slot-Algorithmus ergeben sich Theorem 2 und Theorem 3 aus (Guy69).

Satz 3.1. *(Guy69) Sei $m \geq \binom{n}{2}$ und $n \geq 2$. Dann ist:*

$$\operatorname{maxrf}(m, n) = m + \binom{n}{2}$$

Beweis. Wir haben $\binom{n}{2}$ Zweier zu verteilen. Nun ist jedoch $m \geq \binom{n}{2}$, das heißt wir benötigen Einsermuster für die restlichen Zeilen. Somit erhalten wir für die Anzahl der Einsen insgesamt:

$$\operatorname{maxrf}(m, n) = 2 \cdot \binom{n}{2} + \left(m - \binom{n}{2}\right) = m + \binom{n}{2}$$

□

Theorem 3 aus (Guy69) gibt eine weitere exakte Lösung ab einer bestimmten Zeilenanzahl. Wir erhalten dasselbe Resultat durch Lösung des Gleichungssystems aus Gleichung 3.7.

Satz 3.2. *(Guy69) Sei $m \leq \binom{n}{2} + 1$ und $m \geq \mu(n)$, wobei $\mu(n) \approx \left\lceil \frac{1}{3}\binom{n}{2} \right\rceil$. Dann ist:*

$$\operatorname{maxrf}(m, n) = \left\lfloor 1.5 \cdot m + 0.5 \cdot \binom{n}{2} \right\rfloor \tag{3.10}$$

Beweis. Wir lösen wieder unser Gleichungssystem, allerdings für die gesamte Matrix.

$$\begin{aligned} x + y &= m \\ 3x + y &= \tbinom{n}{2} \\ (x, y &\in \mathbb{N}) \end{aligned}$$

$$x = \frac{1}{2}\left(\binom{n}{2} - m\right), \quad y = \frac{1}{2}\left(3m - \binom{n}{2}\right)$$

x ist die Anzahl der Dreier und y die Anzahl der Zweier. Ein Dreier benötigt 3 Zweier. Insgesamt gibt es $\binom{n}{2}$ Zweier. Aus der Lösung für x erhalten wir die obere Schranke $m \leq \binom{n}{2}$ und durch y erhalten wir die untere Schranke $m \geq \binom{n}{2}/3$. Da eigentlich x und y natürliche Zahlen sind, können wir runden und erhalten dadurch für die obere Schranke: $m \leq \binom{n}{2} + 1$.

Obiges Gleichungssystem berücksichtigt nur Zweier sowie Dreier und Gleichung 3.10 liefert gegebenfalls eine zu kleine Kantenanzahl. Daher gilt die untere Schranke erst, wenn auf höhere Muster verzichtet werden kann: $m \geq \mu(n) \approx \lceil \binom{n}{2}/3 \rceil$. Als Ergebnis erhalten wir $3x + 2y$ Kanten:

$$\begin{aligned}\operatorname{maxrf}(m,n) = \lfloor 3x + 2y \rfloor &= \left\lfloor \frac{3}{2}\binom{n}{2} - \frac{3}{2}m + 3m - \binom{n}{2} \right\rfloor \\ &= \left\lfloor \frac{3}{2}m + \frac{1}{2}\binom{n}{2} \right\rfloor\end{aligned}$$

□

Satz 3.3. *(Rei58) Sei $m > 2$ und $n > 2$. Dann ist $\operatorname{maxrf}(m,n) \leq \frac{m}{2} + \sqrt{mn(n-1) + \frac{m^2}{4}}$.*

Beweis. Ein Muster der Größe β benötigt $\binom{\beta}{2}$ Zweier. Dies soll ein ideales Muster sein, sodass $m \cdot \binom{\beta}{2} = \binom{n}{2}$ gilt. Damit verbrauchen m solcher Muster genau $\binom{n}{2}$ Zweier. Daraus ergibt sich das Gleichungssystem $\beta^2 - \beta - \frac{n(n-1)}{m} \stackrel{!}{=} 0$.

Das Muster mit der idealen Größe β lautet nun (negative Mustergröße ausgeschlossen):

$$\beta = \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{n(n-1)}{m}}$$

Wir weisen den m Zeilen nun das ideale Muster zu. Die realen Mustergrößen sind ganzzahlig und damit im Allgemeinen kleiner als β, damit gilt:

$$\begin{aligned}\operatorname{maxrf}(m,n) \leq m \cdot \beta &= \frac{m}{2} + m\sqrt{\frac{1}{4} + \frac{n(n-1)}{m}} \\ &= \frac{m}{2} + \sqrt{\frac{m^2}{4} + mn(n-1)} =: HC(m,n)\end{aligned}$$

□

$m \setminus n$	4	5	6	7	8	9	10	11	12	13	14	15
4							1	1	1	1	1	1
5											1	1
6					1		1					
7							1		1		1	
8			1		1	1	1			1		1
9					1	1	1	1				1
10	1		1	1	1	1	1	1		1	1	1
11	1					1	1	1			1	1
12	1			1					1		1	1
13	1				1		1				1	1
14	1	1		1			1	1	1	1	1	2
15	1	1			1	1	1	1	1	1	2	4

Tabelle 3.12: Abweichung der oberen Schranke $\min(HC(m,n), HC(n,m))$

$m \setminus n$	4	5	6	7	8	9	10	11	12
4	0_1^0 ×								
5	0_1^0 ×	0_1^0 ×							
6	2_2^0 ×	2_2^0 ×	3_1^1 ×						
7	2_2^{-1} ×	2_3^1 ×	3_1^0 ×	3_1^0 ×					
8	2_2^{-2}	2_3^0 ×	2_2^0 ×	3_1^1 ×	3_1^3				
9	2_2^{-3}	2_3^{-1} ×	3_1^0 ×	3_1^0 ×	3_1^2	4_1^3			
10	2_2^{-4}	3_4^0 ×	2_3^0 ×	3_2^0 ×	3_1^1	4_1^2	4_1^3		
11	2_2^{-5}	3_4^{-1} ×	2_4^0 ×	3_3^0 ×	3_1^0 ×	4_1^1	4_1^2	5_1^4	
12	2_2^{-6}	3_4^{-2}	2_4^{-1} ×	3_2^0 ×	5_2^0 ×	6_2^0 ×	–	–	–
13	2_2^{-7}	3_4^{-3}	3_5^0 ×	3_3^0 ×	3_1^0 ×	6_2^1 ×	4_1^0	–	–
14	2_2^{-8}	3_4^{-4}	3_5^{-1} ×	2_4^0 ×	3_3^0 ×	6_2^0 ×	6_2^2	–	6_1^2
15	2_2^{-9}	3_4^{-5}	3_6^0 ×	3_3^0 ×	3_4^0 ×	4_3^0 ×	6_2^1	6_2^3	6_1^1
16	2_2^{-10}	3_4^{-6}	3_6^{-1} ×	3_5^0 ×	3_3^0 ×	3_4^0 ×	6_2^0 ×	6_2^0	6_1^0
17	2_2^{-11}	3_4^{-7}	3_6^{-2}	3_6^0 ×	3_4^0 ×	3_5^0 ×	3_3^1	6_2^1	3_2^2
18	2_2^{-12}	3_4^{-8}	3_6^{-3}	3_6^{-1} ×	3_3^0 ×	3_4^0 ×	3_3^0 ×	6_2^0	3_3^3
19	2_2^{-13}	3_4^{-9}	3_6^{-4}	3_7^0 ×	3_4^0 ×	3_5^0 ×	6_4^0 ×	3_3^0	3_3^0
20	2_2^{-14}	3_4^{-10}	3_6^{-5}	3_7^{-1} ×	3_3^0 ×	3_4^0 ×	3_3^0 ×	6_4^0 ×	3_3^1
21	2_2^{-15}	3_4^{-11}	3_6^{-6}	4_8^0 ×	3_5^0 ×	3_5^0 ×	4_5^0 ×	6_5^0 ×	3_3^0

Tabelle 3.13: Optimallösung - Kleinste Slotanzahl, Zweierüberschuss, Slotmuster

Tabelle 3.13 zeigt eine Übersicht der Resultate aus Satz 3.1 (grau) und Satz 3.2 (rot). „×" zeigt die Übereinstimmung mit Gleichung 3.10 an. Rote Zahlen entsprechen der unteren Schranke $\lceil \binom{n}{2}/3 \rceil$.

Die Zahlen repräsentieren folgende Informationen: $\textit{Slots}_{\textit{Slot-Muster-ID}}^{\textit{Zweierrest}}$

Die Slotmuster-ID indiziert innerhalb einer Spalte gleiche Slotstrukturen. So besitzen bspw. $(8, 8)$ bis $(11, 8)$ und $(13, 8)$ denselben Matrixkopf.

Abbildung 3.32: Verschiedene Slotmuster mit 3 Slots für $(6, 6)$, $(13, 6)$, $(15, 6)$

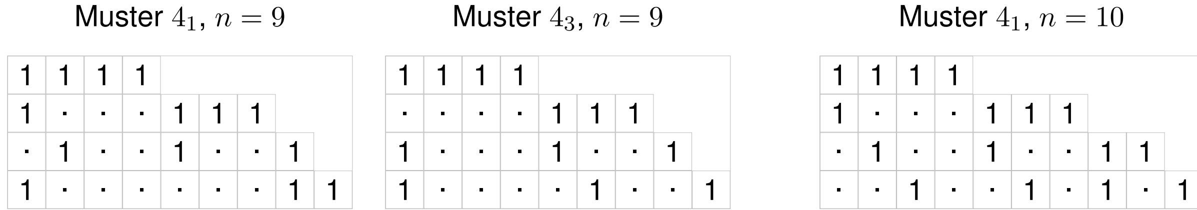

Abbildung 3.33: Verschiedene Slotmuster mit 4 Slots für $(9, 9)$, $(15, 6)$, $(10, 10)$

Der Slot-Algorithmus in der implementierten Form ist bezogen auf die Optimalität zu schwach. Die meisten Lösungen lassen sich bereits mit den Ergebnissen aus Satz 3.1 und Satz 3.2 ermitteln. Dennoch liefert der Algorithmus Einblicke in die Struktur der Bitmuster. Um jedoch bessere Ergebnisse für höhere Dimensionen zu erzielen, müssen Kombinationen aus Vierer, Fünfer, usw. betrachtet werden. Hier beginnt die Schwierigkeit, da nicht klar ist, welche Muster untereinander optimal zusammenpassen und welche Koeffizienten der Muster zur Optimallösung führen. Es verbleiben an der Stelle noch weitere Fragen:

1. Es scheinen nur spezielle Slotmuster in Abhängigkeit von (m, n) in Frage zu kommen. Kann die Menge der möglichen Slotmuster reduziert werden, sprich wie sieht die formale Beschreibung für die (Sub)Optimalität einzelner Slotmuster aus?
2. Ist die Folge der in der Optimallösung verwendeten Mustergrößen immer zusammenhängend (keine Sprünge)?
3. Lässt sich die Anzahl der Slots in Abhängigkeit von (m, n) weiter präzisieren?

Angenommen es gäbe keine Sprünge in den optimalen Mustergrößen und wir schließen Satz 3.1 aus (keine Einsermuster). Wir können die Verwendung der Muster β_i formal wie folgt zusammenfassen:

$$\sum_{i=1}^{t} c_i(\beta - i + 1) \rightarrow \max \tag{3.11}$$

$$\sum_{i=1}^{t} c_i \binom{\beta - i + 1}{2} \rightarrow \max \tag{3.12}$$

$$\sum_{i=1}^{t} c_i = m$$

$$\sum_{i=1}^{t} c_i \binom{\beta - i + 1}{2} \leq \binom{n}{2} \tag{3.13}$$

$$\beta \geq t + 1 \tag{3.14}$$

$$c_i > 0$$

$$t \geq 2$$

Es fehlen allerdings noch weitere Bedingungen, die Slotarchitektur ist hier unberücksichtigt. Lediglich die Wahl der Muster soll so optimiert werden, dass so wenig wie möglich Zweier übrig bleiben. Gleichung 3.11 optimiert die Kantenanzahl und Gleichung 3.12 in Zusammenhang mit Gleichung 3.13 die Verwendung der Zweier. Gleichung 3.14 legt die untere Schranke für das größte Muster β fest. $(\beta - i + 1)$ beschreibt die absteigende Folge der Mustergrößen. Es sollen mindestens zwei verschiedene Muster verwendet werden. Auf genauere Untersuchungen wird im Rahmen dieser Seminararbeit verzichtet.

Soll genau ein Muster verwendet werden, erhalten wir Spezialfälle. $(7,7)$ ist das erste Problem, dass ausschließlich Dreier verwendet, sodass keine Zweier übrigbleiben. $(7,7)$ beschreibt in der projektiven Geometrie die Fano-Ebene, die kleinste projektive Ebene.

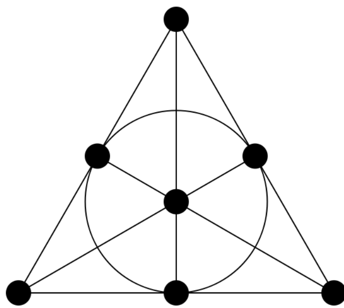

Abbildung 3.34: Fano-Ebene, $(7,7)$ (Wikipedia)

$p_i \setminus g_j$	g_1	g_2	g_3	g_4	g_5	g_6	g_7
p_1	1	1	1	.	.	.	.
p_2	1	.	.	1	1	.	.
p_3	1	.	.	.	.	1	1
p_4	.	1	.	.	1	.	1
p_5	.	1	.	1	.	1	.
p_6	.	.	1	.	1	1	.
p_7	.	.	1	1	.	.	1

Abbildung 3.35: Fano-Ebene Inzidenz-Matrix

Die Definition einer projektiven Ebene gibt u.a. an, dass zwei verschiedene Punkte genau eine Gerade beschreiben, und zwei verschiedene Geraden genau einen (Schnitt)Punkt haben (entspricht unserer Rechteckbedingung).

$(13,13)$ ist das erste Problem, dass ausschließlich mit Vierern belegt ist und keine Zweier übrig lässt. In der projektiven Geometrie wird damit die projektive Ebene der Ordnung 3 beschrieben.

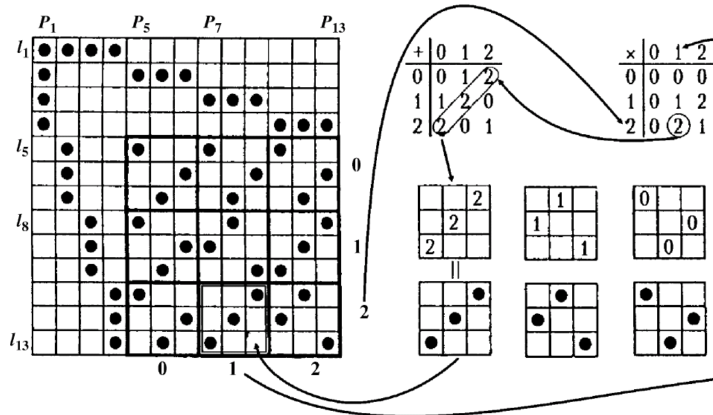

Abbildung 3.36: Inzidenz-Matrix der Projektiven Ebene Ordnung 3 und Galoiskörper (ST07, 994)

Allgemeiner lässt sich die Dimension solcher Spezialfälle wie folgt fassen (Guy69). Sei p eine Primzahlpotenz $(2,3,4,5,7,\ldots)$, dann ist:

$$\text{maxrf}(p^2+p+1, p^2+p+1) = (p+1)\cdot(p^2+p+1) = p^3+2p^2+2p+1$$

Für $m = n = p^2+p+1$ liefert Satz 3.3 auch die Gleichheit.

4 Bewertung bezüglich des Grid-Coloring Problems

In dieser Arbeit wurden zwei verschiedene Arten von Algorithmen entwickelt, die das Zarankiewicz-Problem lösen sollen. Der Overflow-Algorithmus liefert in der Grundform immer eine Optimallösung, jedoch sind die Laufzeiten ab etwa $(12, 12)$ nicht mehr praktikabel. Mithin ist der Suchraum, den der Algorithmus durchläuft, noch viel zu offen gefasst. Es wurde ein erster Ansatz für die Optimierung angegeben, welcher jedoch formal zu ungenau und Optimallösungen schuldig blieb.

Der Slot-Algorithmus verwendet tiefergehende Strukturinformationen beim Erzeugen der 0-1-Matrix bzw. der Adjazenzmatrix. Neben der Slotarchitektur werden Bitmuster mit einer bestimmten Anzahl an Bits erzeugt. Implementiert wurde das Generieren von Dreiern und Vierern, was bei kleineren Dimensionen noch zu akzeptablen Ergebnissen führt (Tabelle 3.10) und in Polynomialzeit geschieht. Die höherwertige Musterbildung ist ungleich schwieriger, da im Sinne der Optimalität nicht klar wurde, in welcher Form die Bitmuster untereinander abhängen. Der Slot-Algorithmus ist aufgrund dieser Ungenauigkeit ebenfalls ab etwa $(12, 12)$ nicht mehr praktikabel.

Verglichen mit den Ergebnissen aus (SP12a) zeigt sich, dass die beiden Algorithmen viel zu schwach sind, das heißt entweder zu ungenau oder zu rechenintensiv. Folgende Tabelle enthält die benötigte Zeit des SAT Solvers clasp für das Auffinden einer (rechteckfreien) Vierfarbenlösung im vollständigen bipartiten Graphen $K_{m,n}$:

$k = m = n$	SUCHRAUM	ZEIT
12	2^{288}	196 ms
13	2^{338}	326 ms
14	2^{392}	559 ms
15	2^{450}	46 min 30 s

Tabelle 4.14: Benötigte Zeit zum Auffinden einer Vierfärbung mit dem SAT Solver clasp

Das Vierfarbenproblem wird als boolesches Problem modelliert, bei dem alle monochromatischen Rechteckkombinationen verboten sind. Die Laufzeiten von clasp sind beachtlich, immerhin ist der Suchraum wesentlich größer, da bei vollständiger Enumeration 4^{mn} statt 2^{mn} Kombinationen möglich sind.

Der Overflow und der Slot-Algorithmus suchen zwar per se nach dem Maximum der Kantenanzahl einer Farbklasse, aber selbst als Entscheidungsproblem, bei dem die Kantenanzahl einer Farbe die Schranke $\lceil \frac{mn}{4} \rceil$ erreichen soll (Satz 1.1), benötigt der Overflow-Algorithmus für $(12, 12)$ etwa 8 h 32 min, nachdem er erstmalig die Lösung mit $12^2/4 = 36$ Kanten gefunden hat[3].

Der Slot-Algorithmus findet in 28 ms bereits eine Lösung mit 43 Kanten ($\text{maxrf}(12, 12) = 45$). Für $(15, 15)$ benötigt selbiger Algorithmus etwa vier Sekunden, jedoch werden nur 56 Kanten gefunden, die Schranke fordert aber $\lceil 15^2/4 \rceil = 57$ Kanten. Die Ungenauigkeit des Slot-Algorithmus führt also bei $(15, 15)$ zu einer falschen Aussage, indem die Schranke fälschlicherweise unterschritten bleibt.

[3]Damit ist immer noch nicht die Vierfärbbarkeit von $(12, 12)$ gezeigt.

5 Anhang

5.1 Overflow-Algorithmus Pseudocode

Beschreibung in Abschnitt 2.

Algorithmus 5.1 Hauptroutine

Input: Zeilenanzahl $m > 2$, Spaltenanzahl $n > 2$, untere Schranke z

Output: Liefert binäre Lösungsmatrix C des Optimierungsproblems aus Unterabschnitt 1.1. Durchläuft Zahlenbereich $\{a_{m-1}, \ldots, 2^n - 1\}$ in der letzten Zeile a_m, ruft anschließend Inkrement-Prozeduren für niedrigere Zeilen auf, solange es möglich ist.

```
 1: procedure MAIN(m, n, z)
 2:     t_max ← z
 3:     INIT(A,B)                                      ▷ init matrix A and B (bit accumulation)
 4:     while true do
 5:         t ← COUNTBITS(a_{m−1})
 6:         if t is 1 then
 7:             a ← a_{k−1}
 8:         else
 9:             INCREMENT(a, a_{k−1}, t)
10:         end if
11:         for a; a ≤ 2^n − 1; a ← a + 1 do                              ▷ loops last row
12:             if HASRECTANGLE(a, a_{m−1}) is true then     ▷ just check against previous row
13:                 continue
14:             end if
15:             t ← b_{m−1}+COUNTBITS(a)
16:             if t ≤ t_max then
17:                 continue
18:             else if VALIDATEFORROW(a,A,m-1) is true then ▷ store new solution, if a is valid in A
19:                 C_m ← a
20:                 t_max ← t
21:                 for i ← 1; i < m; i ← i + 1 do
22:                     C_i ← A_i
23:                 end for
24:             end if
25:         end for
26:         for i = m − 1; i > 1; i ← i − 1 do
27:             tmp← INCREMENTROW(i, A, B, t_max)
28:             if tmp is true then
29:                 break
30:             end if
31:         end for
32:         if i is 1 and tmp is false and INCREMENTFIRSTROW(A, B, t_max) is false then
33:             break                                                       ▷ finished.
34:         end if
35:     end while
36:     return C
37: end procedure
```

Algorithmus 5.2 Inkrementiert erste Zeile a_0, bis obere Grenze erreicht ist

Input: $t_{\max} > 0$, $A = \{a_1, \dots, a_m\}$, $B = \{b_1, \dots, b_m\}$, $m > 2$, $n > 2$
Output: Liefert $true$, falls a_0 inkrementiert werden konnte, sonst $false$.

```
 1: procedure INCREMENTFIRSTROW(A, B, t_max)
 2:     if a_1 ≥ (01...1)_2 then
 3:         return false
 4:     end if
 5:     a_1 ← a_1 + 1
 6:     b_1 ← COUNTBITS(a_1)
 7:     while INCREMENTROW(2, A, B, t_max) is false do
 8:         a_1 ← a_1 + 1
 9:         if a_1 > (01...1)_2 then
10:             return false
11:         else
12:             b_1 ← COUNTBITS(a_1)
13:         end if
14:     end while
15:     return true
16: end procedure
```

Algorithmus 5.3 Inkrementiert Zeile a_i im Bereich $i = 2, \dots, m-1$

Input: $t_{\max} > 0$, $A = \{a_0, \dots, a_m\}$, $B = \{b_0, \dots, b_m\}$, $m > i \geq 2$, $n > 2$
Output: Liefert $true$, falls a_i inkrementiert werden konnte, sonst $false$.

```
 1: procedure INCREMENTROW(i, A, B, t_max)
 2:     if a_i ≥ (101...1)_2 then
 3:         return false
 4:     end if
 5:     L ← b_{i−1}+CLIMIT(i)          ▷ estimate or solve sub-problem (m − i − 1, n)
 6:     if L ≤ t_max then
 7:         return false
 8:     end if
 9:     a ← a_i + 1
10:     t ← COUNTBITS(a)
11:     if t > 1 then
12:         while VALIDATEFORROW(a, A, i) is false do   ▷ validate with upper rows 1, ..., i − 1
13:             a ← a + 1
14:             if a > (101...1)_2 then
15:                 return false
16:             end if
17:         end while
18:         t ← COUNTBITS(a)
19:     end if
20:     a_i ← a
21:     b_i ← b_{i−1} + t
22:     CONFIGUREROWS(i, a, t, A, B, t_max)
23:     return true
24: end procedure
```

Algorithmus 5.4 Passt Zeilen $i+1,\dots,m-1$ an, erhöht ggf. auch wieder Zeile i, sofern möglich

Input: $a, t, t_{\max} > 0$, $A = \{a_1,\dots,a_m\}$, $B = \{b_1,\dots,b_m\}$, $m > i \geq 2$, $n > 2$
Output: Versucht valide Folgezeilen $i+1,\dots,m-1$ zu setzen und erhöht ggf. wieder Zeile i.

```
 1: procedure CONFIGUREROWS(i, a, t, A, B, t_max)
 2:     INCREMENT(a, a_i, t)
 3:     k ← i
 4:     while k < m − 1 do
 5:         k ← k + 1
 6:         if k > i then
 7:             L ← b_{k−1}+CLIMIT(k)              ▷ estimate or solve sub-problem (m − k − 1, n)
 8:             if L ≤ t_max then                  ▷ we won't reach the maximum ones in this branch
 9:                 RESETTOLOWERROW(k, a, a_{k−1}, t)
10:                 continue
11:             end if
12:         end if   % if(k > i)
13:         if t > 1 then
14:             while VALIDATEFORROW(a, A, k) is false do
15:                 a ← a + 1
16:                 if a > (101...1)_2 then
17:                     break
18:                 end if
19:             end while
20:             if a > (101...1)_2 then            ▷ overflow, so return to last row
21:                 if k is i then
22:                     return false
23:                 end if
24:                 RESETTOLOWERROW(k, a, a_{k−1}, t)
25:                 continue
26:             end if   % if(a > (101...1)_2)
27:             t ← COUNTBITS(a)
28:         end if   % if(t > 1)
29:         a_k ← a
30:         b_k ← b_{k−1} + t
31:         if k < m − 1 then                      ▷ fix a for next row
32:             INCREMENT(a, a_k, t)
33:         end if
34:     end while
35: end procedure
```

Algorithmus 5.5 Inkrementiert a bzgl. a_i und der aktuellen Bitanzahl von a

Input: Zeile a_i, neuer Wert a und dessen Bits $t > 0$, $a \geq a_i$
Output: Falls $t = 2$, dann $a = a + 1$. Falls $t > 2$, dann valide nächstgrößere Zahl mit 2 Bits erzeugen.

```
 1: procedure INCREMENT(a, a_i, t)
 2:     if t is 1 then
 3:         if a is 1 then
 4:             a ← 2
 5:         else
 6:             a ← a ∨ 1
 7:         end if
 8:     else if t is 2 then
 9:         b ← LSB(a_i)                          ▷ least significant bit
10:         a ← a_i + b
11:     else
12:         a ← FSB(a_i)                          ▷ first significant bit
13:         a ← a+LSHFSB(a xor a_i)               ▷ left shift of second bit
14:         t ← 2
15:     end if
16: end procedure
```

Algorithmus 5.6 Gehe eine Zeile zurück und setze a auf $a = a_{k-1} + 1$ und t

Input: Zeile a_{k-1}, neuer Wert a und dessen Bits $t > 0$
Output: Setzt a und t eine Zeile zurück und setzt entsprechenden Zeilenindex k.

```
1: procedure RESETTOLOWERROW(k, a, a_{k-1}, t)
2:     k ← k − 2
3:     a ← a_{k+1} + 1
4:     t ← COUNTBITS(a)
5: end procedure
```

Algorithmus 5.7 Initialisiere A und B

Input: A und B sind $m \times n$ Matrizen.
Output: Initialisiert A und B mit günstiger Startbelegung.

```
1: procedure INIT(A, B)
2:     for i ← 1; i ≤ m; i ← i + 1 do
3:         if i < n then
4:             a_i ← 2^{i−1} or 2^i
5:         else
6:             a_i ← 2^{n−1}
7:         end if
8:     end for
9: end procedure
```

5.2 Weitere Dokumente

Der C++ Quellcode der beiden vorgestellten Algorithmen ist auf meiner Website zu finden:

http://11235813tdd.blogspot.com/2012/02/zarankiewicz-problem.html

Ebenfalls werden hier die Quelldateien bereitgestellt, mit denen die Arbeit angefertigt wurde.

Die meisten Grafiken sind mit dem LaTeX Paket Ti*k*Z erstellt worden, das Diagramm in Abbildung 2.14 entstand in gnuplot und Abbildung 3.30 wurde mit processing angefertigt.

- processing v1.5.1: http://processing.org/
- Vorlage für Grid-Pattern in processing
  http://www.learningprocessing.com/examples/chapter-13/example-13-10/
- gnuplot v4.4: http://www.gnuplot.info/
- Ti*k*Z: http://sourceforge.net/projects/pgf/

Literatur

[BF02] BRYANT, Darryn E. ; FU, Hung-Lin: C_4-saturated bipartite graphs. In: *Discrete Math.* 259 (2002), December, 263–268. http://dx.doi.org/10.1016/S0012-365X(02)00371-0. – DOI 10.1016/S0012–365X(02)00371–0. – ISSN 0012–365X

[FGGP10] FENNER, Stephen ; GASARCH, William ; GLOVER, Charles ; PUREWAL, Semmy: Rectangle Free Coloring of Grids. (2010). http://arxiv.org/abs/1005.3750

[Guy69] GUY, Richard: A many-facetted problem of Zarankiewicz. Version: 1969. http://dx.doi.org/10.1007/BFb0060112. In: CHARTRAND, G. (Hrsg.) ; KAPOOR, S. (Hrsg.): *The Many Facets of Graph Theory* Bd. 110. Springer Berlin / Heidelberg, 1969. – ISBN 978–3–540–04629–5, 129-148. – 10.1007/BFb0060112

[Rei58] REIMAN, I.: Über ein Problem von K. Zarankiewicz. In: *Acta Mathematica Academiae Scientiarum Hungaricae* 9 (1958), S. 269–279

[Rom75] ROMAN, Steven: A problem of Zarankiewicz. In: *Journal of Combinatorial Theory, Series A* 18 (1975), Nr. 2, 187 - 198. http://dx.doi.org/10.1016/0097-3165(75)90007-2. – DOI 10.1016/0097–3165(75)90007–2. – ISSN 0097–3165

[SP12a] STEINBACH, B. ; POSTHOFF, Ch.: Most Complex Four-Colored Rectangle-free Grids - Solution of an Open Multiple-Valued Problem. In: *42nd International Symposium on Multiple-Valued Logic* (2012)

[SP12b] STEINBACH, B. ; POSTHOFF, Ch.: Solving Ultra Large Scale Grid Problems. In: *21st International Workshop on Post-Binary ULSI Systems* (2012)

[SP13] STEINBACH, B. ; POSTHOFF, Ch.: Solution of the Last Open Four-Colored Rectangle-free Grid - an Extremely Complex Multiple-Valued Problem. In: *43rd International Symposium on Multiple-Valued Logic* (2013). – (submitted)

[ST07] SAMOYLOVICH, M. ; TALIS, A.: Steiner systems (schemes) and special features of the nanostructures symmetry description. In: *Journal of the European Ceramic Society* 27 (2007), Nr. 2-3, 993-999. http://dx.doi.org/10.1016/j.jeurceramsoc.2006.04.164. – DOI 10.1016/j.jeurceramsoc.2006.04.164. – ISSN 0955–2219

[Sto10] STONES, Douglas S.: The Many Formulae for the Number of Latin Rectangles. In: *Electr. J. Comb.* 17 (2010), Nr. 1