Studiengang:	Informatik
Prüfer:	Prof. U. Hertrampf
Betreuer:	Prof. U. Hertrampf
begonnen am:	01. Juli 2003
beendet am:	04. Februar 2004
CR-Klassifikation:	F.1.3

Diplomarbeit Nr. 2114

Polynomialzeitbeschränkte Häufigkeitsberechnungen mit 3 Fehlern

C. F. Minnameier

Fakultät Informatik, Elektrotechnik und Informationstechnik
Institut für Formale Methoden der Informatik, Universität Stuttgart
Universitätsstraße 38, D-70569 Stuttgart

Zusammenfassung

Diese Arbeit beschäftigt sich mit ressourcenbeschränkten Häufigkeitsberechnungen, und zwar mit polynomialzeitbeschränkten. Wir betrachten die in diesem Zusammenhang von Hinrichs/Wechsung aufgestellte Vermutung, dass die Klassen aller im Sinne von Häufigkeitsberechnungen mit d Fehlern in Polynomialzeit berechenbaren Mengen für 2^d+d oder mehr Elemente alle gleich sind, d.h.:

$$\forall d, k \in \mathbb{N} : (2^d, 2^d + d)\mathrm{P} = (2^d + k, 2^d + d + k)\mathrm{P}.$$

Für $d < 3$ ist die Vermutung einfach zu beweisen. Wir beweisen sie für $d = 3$ und liefern damit ein gewichtiges Indiz für ihre allgemeine Gültigkeit. Das angewendete Verfahren scheint außerdem geeignet, die Vermutung für größere d zu beweisen.

Inhaltsverzeichnis

1 Einleitung

1.1 Häufigkeitsberechnungen

Der Begriff der Häufigkeitsberechnungen wurde Anfang der 60er Jahre von G.F.Rose geprägt [Ros60]. Gemeint sind damit approximierende parallele Berechnungen für n verschiedene Elemente. Eine Menge P heißt (m, n)-berechenbar, g.d.w. ein Algorithmus existiert, der für jede Eingabe von n Elementen für diese mindestens m korrekte Aussagen über deren Zugehörigkeit zu P macht.

Es werden 3 unterschiedliche Betrachtungsweisen für Häufigkeitsberechnungen unterschieden: Die rekursionstheoretische, die ressourcenbeschränkte und die Berechnung durch endliche Automaten. Es folgt eine kurze Übersicht über die wichtigsten (wichtigsten im Allgemeinen, aber auch im Hinblick auf diese Arbeit) Ergebnisse auf den jeweiligen Gebieten.

Das wichtigste Ergebnis für die Berechenbarkeitstheorie stammt von Trakhtenbrot. Er zeigte in [Tra63], dass die Menge aller (m, n)-berechenbaren Funktionen mit $m/n > 1/2$ genau der Menge aller überhaupt berechenbaren Funktionen entspricht, und dass im Gegenzug nicht allgemein berechenbare, (1,2)-berechenbare Funktionen existieren. Dass die Komplexitätsklassen, die Inhalt der Hinrichs/Wechsung'schen Vermutung (siehe unten) sind, ohne Ressourcenbeschränkung zusammenfallen würden, ist also bereits erwiesen, weil für sie $m/n > 1/2$ gilt.

Ein wichtiges Resultat für das Modell der endlichen Automaten erbrachte Kinber in [Kin75], nämlich dass $(m, n) = (m', n') \Leftrightarrow (m > n/2) \wedge (m' > n'/2) \vee (m = m') \wedge (n = n')$. Der in [Kin75] aufgeführte Beweis hierfür ist allerdings fehlerhaft. Dies konnten Austinat, Diekert, Hertrampf und Peterson in [ADHP00] zeigen und einen korrekten Beweis erbringen. Außerdem zeigte Kinber, dass für $n > 1$ kein allgemein anwendbares Verfahren existiert, um eine Entscheidung aus einem $(n - 1, n)$-Algorithmus zu extrahieren.

Kommen wir nun zu dem Modell, mit dem wir uns auch in dieser Arbeit befassen: Den ressourcenbeschränkten Häufigkeitsberechnungen. Zuerst zeigten Kummer und Stephan in [KS95] einige Inklusions- und Nichtinklusionsrelationen zwischen ressourcenbeschränkten Häufigkeitsklassen im Allgemeinen. Später konnten Hinrichs und Wechsung zeigen, dass für ressourcenbeschränkte (also insbesondere auch für polynomialzeitbeschränkte) Häufigkeitsberechnungen mit d Fehlern gilt: $(m, n) \supsetneq (m + 1, n + 1) \ \forall \ m < 2^d$. Weiterhin konnten sie beweisen, dass für jede Fehleranzahl d die so benachbarten Klassen ab einer gewissen Anzahl Elemente alle gleich sind [HW97]. Die genannten Schranken sind allerdings Ramsey-Zahlen und damit sehr

groß. Schließlich äußerten sie auch die Vermutung, dass sogar alle Klassen, für die sie keine Ungleichheit zeigen konnten, gleich sind, also:

$$\forall d \in \mathbb{N}, m \geq 2^d, n = m + d : (m, n)\mathrm{P} = (m + 1, n + 1)\mathrm{P}.$$

Diese Vermutung wird im folgenden schlicht Hinrichs/Wechsung'sche Vermutung genannt. Der Beweis der Vermutung für $d = 3$ ist Inhalt dieser Diplomarbeit.

1.2 Die Hinrichs/Wechsung'sche Vermutung für $d < 3$

$\underline{d = 1:}$ (2,3)P = (3,4)P = (4,5)P = ...

Wir betrachten die Vermutung für $d = 1$. Die eine Richtung, $(n-2, n-1)$P $\supseteq (n-1, n)$P $\forall\ n > 3$, ist einfach zu zeigen und funktioniert nach demselben Prinzip, das auch in Abschnitt 2.2 angewendet wird, um die entsprechende Relation für $d = 3$ zu zeigen. Wir werden also zeigen, dass $(n-2, n-1)$P $\subseteq (n-1, n)$P $\forall\ n > 3$ gilt:

Gegeben sind n Elemente. Gesucht ist ein n-Bit-Vektor, der gegenüber der charakteristischen Funktion der n Elemente in höchstens einer Komponente abweicht. Es existieren n Möglichkeiten aus den n Elementen $(n-1)$ auszuwählen. Jede dieser n Auswahlen von $(n-1)$ Elementen verwenden wir als Eingabe für den $(n-2, n-1)$P-Algorithmus und erhalten so n Ausgaben à $(n-1)$ Bits, von denen maximal je eines falsch sein kann, zusammen also $n(n-1)$ Bits, von denen maximal n falsch sein können.

Vorbemerkung: Wenn wir über die Zugehörigkeit eines Elements mit Sicherheit entscheiden können, können wir auch den gesuchten Vektor angeben, indem wir das Zugehörigkeits-Bit mit der Antwort des (n-2,n-1)P-Algorithmus für die restlichen Elemente kombinieren.

Beachte: Für jedes Element erhalten wir $(n-1)$ Bits (eine „Bit-Reihe"). Eine Bit-Reihe besteht entweder nur aus konsistenten Angaben (nur 0en oder nur 1en), oder inkonsistenten Angaben im Verhältnis $1 : (n-2)$, $2 : (n-3)$, ..., $\lfloor (n-1)/2 \rfloor : \lceil (n-1)/2 \rceil$. (Dabei meint $k : l$, dass die Bit-Reihe des Elements k Bits mit dem Wert 0 und l Bits mit dem Wert 1 enthält oder umgekehrt.) Für ungerade n kann es vorkommen, dass wir für ein Element gleich viele 1- wie 0-Antworten erhalten, in allen anderen Fällen können wir für jedes Element eine Majorität in seiner Bit-Reihe identifizieren. Für ein so beantwortetes Element ist jedoch die Hälfte der Angaben falsch, d.h. es können in einer vollständigen Beantwortung maximal 2 solche Elemente vorkommen, denn allein in 3 solchen Elementen würden $(n-1)/2 * 3 = 3/2n - 3/2$ Fehler gemacht, und es gilt $3/2n - 3/2 > n\ \forall\ n > 3$.

Fall 1) Es kommen ein oder zwei Elemente mit einer solchen $(j : j)$ Beantwortung vor.
Dann muss jede vorkommende Majorität korrekt sein, denn $(n-1)/2$ [mindestens eine $(j : j)$-Bitreihe] + $(n+1)/2$ [falsche Majorität] + $(n-3)$ [mindestens ein Fehler für jede weitere Bitreihe] $= 2n - 3 > n\ \forall\ n > 3$.
Damit sind wir fertig. (Siehe Vorbemerkung.)

Fall 2) Es kommt kein Element mit solcher Beantwortung vor:

> **Fall 2.1)** Wir erhalten ausschließlich inkonsistente Bit-Reihen.
> Die Majoritäten müssen korrekt sein, denn $\lceil (n-1)/2 \rceil$ [für eine falsche Majorität] + $(n-1)$ [mindestens 1 Fehler pro weitere inkonsistente] $\geq (n-1) * 3/2 > n \; \forall \; n > 3$.
>
> **Fall 2.2)** Wir erhalten maximal eine inkonsistente Bit-Reihe.
> Dann geben wir die Majoritäten an, denn darunter können keine zwei Fehler sein, denn unter zwei Fehlern müsste mindestens eine konsistente sein: $(n-1)$ [für eine falsche Konsistente] + $\lceil (n-1)/2 \rceil$ [für eine falsche Majorität] $\geq (n-1) * 3/2 > n \; \forall \; n > 3$.
>
> **Fall 2.3)** Wir erhalten mindestens eine konsistente und mindestens 2 inkonsistente Bit-Reihen.
> Dann muss jede konsistente korrekt sein, denn $(n-1)$ [für eine falsche Konsistente] + 2 [mindestens 1 Fehler pro weitere inkonsistente] $= n + 1 > n$.

$\underline{d = 2:}$ **(4,6)P = (5,7)P = (6,8)P = ...**

Der Fall $d = 2$ wäre ähnlich zu beweisen, wir belassen es aber dabei zu erwähnen, dass er außerdem aus [KS95] abgeleitet werden kann.

2 Definitionen, Ausblick und Zielsetzung

2.1 Definitionen

Gegeben seien:

P, die Problemmenge, d.h. für gegebene Elemente wollen wir über die Zugehörigkeit zu P entscheiden.

T, die Menge für deren Elemente wir über die Zugehörigkeit von P entscheiden sollen. Wir wollen die Elemente von T als Vektor V_T anordnen, wobei die Reihenfolge beliebig sein soll. Einen $|T|$-elementigen Bit-Vektor C_{V_T} nennen wir die charakteristische Funktion von V_T, wenn gilt: $C_{V_T}(i) = c(V_T(i))$, wobei c die charakteristische Funktion auf P meint.

A, ein Algorithmus, der für eine Problemmenge P und jede Menge T von 12 Elementen bzw. einem zugehörigen Vektor V_T wie oben beschrieben, einen 12-Bit-Vektor ausgibt, der sich von C_{V_T} in höchstens 3 Komponenten unterscheidet.

Begrifflichkeiten:

Als Antwort a bezeichnen wir einen 13-stelligen Vektor mit 12 Bits und einem Stern, z.B. *001011100∗010*. Das ist unsere Notation für die Ausgabe von A auf eine Eingabe, die eine Auswahl von 12 der 13 Elemente aus T darstellt. Der $*$ in a markiert die Position des Elements in V_T, das bei der Eingabe für A weggelassen wurde und dem somit in der Ausgabe von A kein Bitwert zugeordnet wurde. Die Antwort geht aus dem Einfügen des $*$s an der besagten Position in die Ausgabe des (9,12)-Algorithmus hervor. Mit $a(i)$ $1 \leq i \leq 13$ meinen wir, wie für Vektoren üblich, die i-te Komponente von a. Wir werden die Komponenten einer Antwort Bitpositionen und ihre Werte salopp Bits nennen, ungeachtet der Tatsache, dass darunter auch der $*$ vorkommt.

Als Antwortsequenz M bezeichnen wir eine Matrix der Dimension m x 13 ($1 \leq m \leq 13$) über $\{0, 1, *\}$, deren 1 bis 13 Zeilen Antworten sind, wobei keine Spalte von A mehr als einen $*$ enthalten soll. Mit z_A bezeichnen wir die Anzahl der Zeilen von A, und mit $A(i)$, $1 \leq i \leq z_A$, meinen wir die i-te Zeile von A. Beachte, dass wir jede so definierte Matrix als Antwortsequenz bezeichnen, ungeachtet der Tatsache, dass einige dieser Matrizen niemals

von A ausgegeben werden können, weil sie zu keinem Vektor C_{V_T} passen. Dieser Umstand wird in Kapitel 4 erörtert.

Für eine Antwortseqeuenz A bezeichne $A_{1..n}$ mit $n \leq z_A$ die n-zeilige Antwortsequenz, die man erhält, wenn man in A die Zeilen $n+1$ bis z_A streicht.

Als „linken Teil“ einer Antwortsequenz bezeichnen wir die Spalten ohne $*$.

Als „rechten Teil“ einer Antwortsequenz bezeichnen wir dementsprechend die Spalten, die einen $*$ enthalten.

Als „vollständig“ bezeichnen wir eine Antwortsequenz mit 13 Zeilen.

Als Nullspalte bezeichnen wir fortan eine Spalte einer Matrix, die nur Nullen (keine 1en, keinen $*$) enthält. Falls in diesem Zusammenhang von den Bits einer Antwort, einer möglichen Wahrheit oder der Lösung die Rede ist, werden die Bitpositionen auf Höhe der Nullspalten der betrachteten Matrix 'Nullspaltenbits' genannt. Nullspaltenbits können also auch von 0 abweichende Werte annehmen.

Abstände:

Da manche Vektoren einen $*$ enthalten, erweitern wir den Bit-Operator XOR für die Anwendung auf den $*$ wie folgt: Für $e \in \{0,1,*\}$ sei $\mathrm{XOR}(e,*) = \mathrm{XOR}(*,e) = 0$. Damit gelte für x, y Vektor, W Vektormenge, M Antwortsequenz mit z_M Zeilen und $1 \leq r < s \leq 13$:

$$d(x,y) := \sum_{i=1}^{13} XOR(x(i), y(i))$$

$$d_{r..s}(x,y) := \sum_{i=r}^{s} XOR(x(i), y(i))$$

$$d(x,W) := \max_{m \in W}(d(x,m))$$

$$d_{r..s}(x,W) := \max_{m \in W}(d_{r..s}(x,m))$$

$$d(x,M) := \max_{1 \leq i \leq z_M}(d(x,M(i)))$$

$$maxdist_M := \max_{1 \leq i < j \leq z_M}(d(M(i),M(j)))$$

Weitere Begrifflichkeiten:

$W_M \subseteq \{0,1\}^{13}$, mit M Antwortsequenz, sei die Menge der 13-Bit-Vektoren, die nach einer gegebenen Antwortsequenz M noch als charakteristische Funktion für V_T in Frage kommen. Wir nennen diese Menge die Menge der möglichen Wahrheiten. Es ist $W_M = \{w \in \{0,1\}^{13} \mid d(w, M) \leq 3\}$, und es gilt: $C_{V_T} \in W_M$, falls für M überhaupt ein C_{V_T} existiert.

Als Lösung l für eine Antwortsequenz M bezeichnen wir einen 13-Bit-Vektor über $\{0, 1\}$, für den $d(l, W_M) \leq 3$ und somit insbesondere $d(l, C_{V_T}) \leq 3$ gilt.

Ein Vektor v, der in Komponente i ein ? enthält, meint die Menge von Vektoren, die man erhält, wenn man $v(i)$ beliebig auf 0 bzw. 1 setzt.

Mit (v, e), v Vektor $e \in \{0, 1, ?\}$ meinen wir einen Vektor, der eine Komponente mehr als v enthält und dessen neue, letzte Komponente den Wert e hat, also $(v, e) = (v_1, v_2, ..., v_{|v|}, e)$.

Sei M^+ die Matrix, die aus einer Matrix M durch Anfügen einer Nullspalte ganz rechts an die Matrix entsteht und M^- die Matrix, die aus einer Matrix M durch Löschen der rechtesten Nullspalte entsteht.

Operatoren:

• Flip(i,...) mit $1 \leq i \leq 13$:
Flip(i, a), $a \in \{0,1\}^{13} := (a_0, a_1, ..., a_{i-1}\ , \overline{a_i}, a_{i+1}, \ ...\ , a_{12}, a_{13})$
Flip(i, W), $W \subseteq \{0, 1\}^{13} := \{\text{Flip}(i, v) \mid v \in W\}$
Flip(i, M), M Antwortsequenz := Antwortsequenz M' mit $z_{M'} = z_M$ und $M'(j) = \text{Flip}(i, M(j))\ \forall\ 1 \leq j \leq z_M$.

• SwapCol$(i, j, ...)$, mit $1 \leq i < j \leq 13$:
Swap$(i, j, a), a \in \{0,1\}^{13} := (a_0, \ ...\ , a_{i-1}, a_j, a_{i+1}, \ ...\ , a_{j-1}, a_i, a_{j+1}\ , \ ...\ , a_{13})$
SwapCol(i, j, W), $W \subseteq \{0,1\}^{13} := \{\text{Swap}(i, j, v) \mid \text{v} \in W\}$
SwapCol(i, j, M), M Antwortsequenz := Antwortsequenz M' mit $z_{M'} = z_M$ und $M'(k) = \text{Swap}(i, j, M(k))\ \forall\ 1 \leq k \leq z_M$.

• SwapRow(i, j, M), mit $1 \leq i < \text{j} \leq z_M$ und M, M' Antwortsequenz:
SwapRow$(i, j, M) := M'$ mit $M'(i) = M(j)$, $M'(j) = M(i)$, und $M'(k) = M(k)$ für $1 \leq k \leq z_M$, $i \neq k \neq j$.

2.2 Grundzüge des Beweises

Die Hinrichs/Wechsung'sche Vermutung besagt für $d = 3$: (8,11)P = (9,12)P = (10,13)P = ...
Tatsächlich wird (8,11)P = (9,12)P aus unserem Beweis nicht direkt hervorgehen, es lässt sich aber durch minimale Änderungen im Beweis zeigen, und wir erlauben uns daher diesen Fall zu vernachlässigen. Wir beginnen die Gleichungskette stattdessen mit einem direkten Beweis für (9,12)P = (10,13)P, weil für die Erweiterung auf die Klassen mit mehr Elementen gewisse Vorgaben nötig sind, die nur so gewährleistet sind.
Wir wollen beweisen, dass alle Klassen $(n, n+3)$P für $n \geq 9$ gleich sind. Dafür reicht es aus zu zeigen, dass $(n, n+3)\text{P} \supseteq (n+1, n+4)\text{P}$ und $(n, n+3)\text{P} \subseteq (n+1, n+4)\text{P}$ für $n \geq 9$ gilt.

Der erste Teil, $(n, n+3)\text{P} \supseteq (n+1, n+4)\text{P}$, gilt sogar für alle $n \geq 1$ bei allen ressourcenbeschränkten und nicht ressourcenbeschränkten Häufigkeitsberechnungen und ist sehr einfach zu beweisen:
Gegeben sei eine Menge $M \in (n+1, n+4)\text{P}$ und ein zugehöriger Polynomialzeitalgorithmus A. Nun wollen wir in Polynomialzeit für eine gegebene Menge T mit $|T| = n+3$ eine Ausgabe erzeugen, die jedem Element aus T ein Bit zuordnet und die von der charakteristischen Funktion auf M eingeschränkt auf T nur in maximal 3 Funktionswerten abweicht. Dafür erweitern wir einfach die Menge T um ein Dummy-Element, das in T bislang nicht vorhanden war, und verwenden die entstandene Menge T' mit $n+4$ Elementen als Eingabe für A. A ordnet nun in Polynomialzeit jedem der $n+4$ Elemente ein Bit zu und weicht dabei laut Vorgabe von der charakteristischen Funktion auf P eingeschränkt auf T' nur in maximal 3 Funktionswerten ab. Die geforderte Ausgabe erhalten wir, indem wir das Dummy-Element samt zugeordnetem Bit streichen, denn von den $n+4$ Zuordnungen waren maximal 3 falsch, somit sind von den verbliebenen $n+3$ Zuordnungen immer noch maximal 3 falsch.

Der zweite Teil, $(n, n+3)\text{P} \subseteq (n+1, n+4)\text{P}$ für $n \geq 9$, macht den größten Teil dieser Arbeit aus:
Zuerst zeigen wir (9,12)P $\subseteq$ (10,13)P. Gegeben sei also eine Menge $M \in (9, 12)P$ und der zugehörige Polynomialzeitalgorithmus A. Nun sollen wir in Polynomialzeit für eine gegebene Menge T mit |T| = 13 eine Ausgabe erzeugen, die jedem Element aus T ein Bit zuordnet und die von der charakteristischen Funktion auf M eingeschränkt auf T nur in maximal 3 Funktionswerten abweicht. Wir werden dafür alle 13 möglichen 12-elementigen Teilmengen T' (die man durch Weglassen eines Elements von T erhält) von T als Eingabe für A verwenden. Diese Berechnungen sind auf das 13-fache

der maximalen Rechenzeit von A beschränkt und laufen damit in Polynomialzeit ab. Wir erhalten 13 Ausgaben, die jeweils jedem Element aus dem gewählten T' ein Bit zuordnen und dabei gegenüber der charakteristischen Funktion von M eingeschränkt auf T' in maximal 3 Zuordnungen abweichen. Wir werden zeigen, dass wir für jede möglichen Abfolge von 13 solcher Antworten eines solchen Algorithmus A eine Zuordnung für alle 13 Elemente von T angeben können, die ihrerseits von der charakteristischen Funktion von P eingeschränkt auf T in maximal 3 Bits abweicht.

Dafür müssen wir alle möglichen Sequenzen der 13 Antworten des (9,12)P-Algorithmus betrachten. Zu Beginn (vor der ersten Antwort) besitzen wir keinerlei Wissen über die 13 betrachteten Elemente. Das bedeutet, die tatsächliche charakteristische Funktion dieser 13 Elemente könnte jede der 2^{13} Zuordnungen sein, die 13 Elementen Bitwerte zuweist. Eine Antwort kann verhindern, dass einige dieser Zuordnungen weiterhin in Frage kommen.

So schließt jede Antwortsequenz Zuordnungen als mögliche tatsächliche charakteristische Funktion aus und im Gegenzug verbleibt für jede Antwortsequenz eine Menge W von Zuordnungen, die wir die Menge der (noch) möglichen Wahrheiten (also der noch als tatsächliche charakteristische Funktion in Frage kommenden Zuordnungen,) nennen wollen.

Wir betrachten unseren Beweis als beendet, wenn wir für jede vollständige Antwortsequenz A gezeigt haben, dass diese entweder gar nicht vorkommen kann, weil keine charakteristische Funktion existiert zu der die Antwortsequenz passen würde, oder dass wir die Menge der verbleibenden möglichen Wahrheiten mit einer Zuordnung mit höchstens 3 Fehlern approximieren können, die dann unsere gewünschte Ausgabe darstellt.

Nachdem wir so (9,12)P $\subseteq$ (10,13)P gezeigt haben, werden wir zeigen, dass sich der Beweis induktiv für alle $n \geq 9$ auf $(n, n+3)\mathrm{P} \subseteq (n+1, n+4)\mathrm{P}$ übertragen lässt. (Unsere Argumentation für den Induktionsschritt würde mit (8,11)P $\subseteq$ (9,12)P als Induktionsanfang allerdings nicht funktionieren.)

2.3 Zielsetzung

Wie schon in der Einleitung bemerkt, wollen wir alle möglichen Antwortsequenzen untersuchen, die ein (9,12)P-Algorithmus ausgeben könnte. Für jede vollständige Antwortsequenz A wollen wir zeigen, dass mindestens einer der folgenden Fälle eintritt:

a) $W_A = \emptyset$, d.h. $\forall\ v \in \{0,1\}^{13}$: $d(v, A) > 3$.
Das bedeutet, die Antwortsequenz A kann in der Realität nicht vorkommen, denn die Antworten machen widersprüchliche Aussagen über C_{V_T}. Bei gegebenem C_{V_T} wird ein korrekter (9,12)P-Algorithmus eine solche Antwortsequenz nicht ausgeben.

b) $\exists\ n$ mit $1 \leq n \leq 13$: $\forall\ w, w' \in W_A$: $w(n) = w'(n)$.
Wir können die Parität eines der Bits von C_{V_T} mit Sicherheit feststellen. Dann ergibt sich die Lösung als Kombination aus diesem Bit und der Antwort die man erhält, wenn man den (9,12)P-Algorithmus nach allen Bits außer diesem fragt, denn von diesen Bits unterscheiden sich maximal 3 von denen in C_{V_T}.

c) Wir finden ein l mit $d(l, w) \leq 3\ \forall\ w \in W_A$.
Die Menge der zulässigen Wahrheiten wird so „klein“, dass wir sie und damit den enthaltenen Vektor C_{V_T} effektiv mit einem 13-Bit-Vektor bis auf maximal 3 Fehler approximieren können, den wir dann Lösung nennen.

Mit abc_A meinen wir von nun an, dass für A mindestens einer der Fälle a), b), c) eintritt. Entsprechend bedeute z.B. ac_A mindestens einer der Fälle a) und c) und b_A, dass für A Fall b) eintritt, usw...

Das nachfolgende Lemma birgt offensichtlich eine Möglichkeit, die Anzahl der tatsächlich zu betrachtenden Antwortsequenzen einzuschränken. Wir werden später darauf zurückgreifen.

Lemma 2.3.1: Gilt für eine Antwortsequenz A mindestens einer der Fälle a) b) c), so gilt auch mindestens einer der Fälle für jede Matrix A', die sich durch Anfügen weiterer Zeilen aus A erstellen lässt, d.h.
$(abc_A \wedge A'_{1..z_A} = A) \Rightarrow abc_{A'}$

Begründung: Es gilt: $W_{A'} \subseteq W_A$. Damit sind die folgenden Implikationen trivial: $a_A \Rightarrow a_{A'}$, $b_A \Rightarrow ab_{A'}$, $c_A \Rightarrow ac_{A'}$.

3 Grundlagen zur Einschränkung des Berechnungsaufwands

3.1 Symmetrieoperatoren und Symmetrie

Wir wollen zeigen, dass für jede beliebige Antwortsequenz Fall a), b) oder c) eintritt. Weil es uns wegen der hohen Anzahl an möglichen Antwortsequenzen nicht möglich ist, tatsächlich alle zu untersuchen, sind wir gezwungen, Symmetrien auszunutzen. Solche Symmetrien sollen nun erklärt werden:

Definition der Symmetrieoperatoren:

• Das 'Bitflipping' aller Bits einer Spalte einer Matrix. Bitflipping meint hier, dass 1en zu 0en und 0en zu 1en werden. Es ist in Anführungszeichen gesetzt, weil die Spalten unserer Matrizen nicht ausschließlich Bits enthalten, sondern auch $*$e. Diese $*$e sollen von unserem Bitflipping unbeeinflusst bleiben. Falls eine Matrix A mittels Bitflipping einer Spalte i in Matrix B übergeführt werden kann, schreiben wir:
$A \overset{Flip(i)}{\rightarrow} B$

• Das Vertauschen zweier Spalten einer Matrix. Falls eine Matrix A mittels Vertauschen zweier Spalten i, j in Matrix B übergeführt werden kann, schreiben wir:
$A \overset{SwapCol(i,j)}{\rightarrow} B$

• Das Vertauschen zweier Zeilen einer Matrix. Falls eine Matrix A mittels Vertauschen zweier Zeilen i, j in Matrix B übergeführt werden kann, schreiben wir:
$A \overset{SwapRow(i,j)}{\rightarrow} B$

Lemmata zu den Symmetrieoperatoren:

Lemma 3.1.1: $A \overset{Flip(i)}{\rightarrow} B \Rightarrow W_B = \text{Flip(i, } W_A)$
(Das Bitflipping einer Spalte i einer Matrix A verursacht in der Menge W_A ausschließlich ein Bitflipping derselben Spalte i.)

Beweis:
$W_B = \{w \in \{0,1\}^{13} \mid d(w, B) \leq 3 \}$
Mit $A \overset{Flip(i)}{\rightarrow} B$ ergibt sich $d(w, A) = d(\text{Flip}(i, w), B)$.

Damit ist $W_B = \{\text{Flip}(i,w) \mid w \in \{0,1\}^{13} : d(w,A) \leq 3\}$
$= \{\text{Flip}(i,w) \mid w \in W_A\}$
$= \text{Flip}(i,W_A)$

Lemma 3.1.2: $A \overset{SwapCol(i,j)}{\rightarrow} B \Rightarrow W_B = \text{SwapCol(i,j,W}_A)$
(Das Vertauschen zweier Spalten i, j einer Matrix A verursacht in der Menge W_A ausschließlich ein Vertauschen derselben Spalten i, j.)

Beweis:
$W_B = \{w \in \{0,1\}^{13} \mid d(w,B) \leq 3\ \}$
Mit $A \overset{SwapCol(i,j)}{\rightarrow} B$ ergibt sich $d(w,A) = d(\text{Swap}(i,j,w),B)$.

Damit ist $W_B = \{\text{Swap}(i,j,w) \mid w \in \{0,1\}^{13} : d(w,A) \leq 3\}$
$= \{\text{Swap}(i,j,w) \mid w \in W_A\}$
$= \text{SwapCol}(i,j,W_A)$

Lemma 3.1.3: $A \overset{SwapRow(i,j)}{\rightarrow} B \Rightarrow W_B = W_A$.
$W_A = \{w \mid d(w,A) \leq 3\}$. In der Definition von $d(w,A)$ spielt aber die Anordnung der Zeilen von A keine Rolle. Damit ist offensichtlich, dass das Lemma gilt.

Definition: Wenn sich A_1 mittels wiederholter Anwendung von Symmetrieoperatoren in A_2 überführen lässt, dann sagen wir A_1 ist symmetrisch zu A_2. Schreibe: $A_1 \equiv A_2$.

Da die so definierte Symmetrierelation eine Äquivalenzrelation auf der Menge aller Antwortsequenzen bildet, weisen wir darauf hin, dass die Matrizen durch $\equiv$ in Äquivalenzklassen $[A]$ eingeteilt werden.

3.2 Bedeutung von Symmetrie

Lemma 3.2.1: $(A_1 \equiv A_2) \Rightarrow (abc_{A_1} \Leftrightarrow abc_{A_2})$
Bemerkung: Um also, wie in Abschnitt 2.3 beschrieben, zu zeigen, dass für alle Antwortsequenzen A abc_A gilt, genügt es für jede Äquivalenzklasse $[A]$ symmetrischer Matrizen nur für ein $A \in [A]$ stellvertretend zu zeigen, dass abc_A gilt.

Beweis:
Angenommen, wir führen n Symmetrieoperationen auf A_1 durch, um A_1 in A_2 zu überführen, dann ergibt sich eine Abfolge von Matrizen $A_1 = M_1 \rightarrow ... \rightarrow M_n \rightarrow M_{n+1} = A_2$. Wir zeigen: Gilt für M_i $(1 \leq i \leq n)$ abc_{M_i}, so gilt auch $abc_{M_{i+1}}$, egal mit welchem Symmetrieoperator M_i in M_{i+1} übergeführt wurde.

Fall 1) $M_i \overset{Flip(j,M_i)}{\rightarrow} M_{i+1}$

- $a_{M_i} \Rightarrow a_{M_{i+1}}$
mit Lemma 3.1.1: $W_{M_{i+1}} = \text{Flip(j, } W_{M_i}) = \text{Flip(j, } \emptyset) = \emptyset$

- $b_{M_i} \Rightarrow b_{M_{i+1}}$
mit Lemma 3.1.1: $W_{M_{i+1}} = \text{Flip}(j, W_{M_i})$
Ist nun in W_{M_i} ein Bit aller möglichen Wahrheiten gleich, so ist es auch in $W_{M_{i+1}}$ gleich (nur eventuell invertiert, wenn es gerade Bit j ist).

- $c_{M_i} \Rightarrow c_{M_{i+1}}$
Für u, v Vektor, W Vektormenge gilt:
$d(u, v) = d(\text{Flip}(j, u), \text{Flip}(j, v))$
$\Rightarrow d(u, W) = d(\text{Flip}(j, u), \text{Flip}(j, W))$
mit Lemma 3.1.1: $W_{M_{i+1}} = \text{Flip}(j, W_{M_i})$:
$(\exists\ l \in \{0,1\}^{13}$ mit $d(l, W_{M_i}) \leq 3) \Rightarrow$
$(\exists\ l'$ mit $d(l', \text{Flip}(j, W_{M_{i+1}})) = d(l, W_{M_i}) \leq 3)$.
(nämlich $l' = \text{Flip}(j, l)$)

Fall 2) $M_i \overset{SwapCol(j,k,M_i)}{\rightarrow} M_{i+1}$

- $a_{M_i} \Rightarrow a_{M_{i+1}}$
mit Lemma 3.1.2: $\text{W}_{M_{i+1}} = \text{SwapCol}(j,\ k,\ W_{M_i}) = \text{SwapCol}(j,\ k,\ \emptyset) = \emptyset$

- $b_{M_i} \Rightarrow b_{M_{i+1}}$

mit Lemma 3.1.2: $W_{M_{i+1}} = \text{SwapCol}(j, k, W_{M_i})$.

War Bit j in allen Wahrheiten gleich, so ist es nun Bit k und umgekehrt. War ein anderes Bit in allen Wahrheiten gleich, so trifft dies noch immer zu.

- $c_{M_i} \Rightarrow c_{M_{i+1}}$

Für u, v Vektor, W Vektormenge gilt:
$d(u, v) = d(\text{Swap}(j, k, u), \text{Swap}(j, k, v))$
$\Rightarrow d(u, W) = d(\text{Swap}(j, k, u), \text{SwapCol}(j, k, W))$
mit Lemma 3.1.2: $W_{M_{i+1}} = \text{SwapCol}(j, k, W_{M_i})$:
$(\exists\ l \text{ in } \{0,1\}^{13} \text{ mit } d(l, W_{M_i}) \leq 3) \Rightarrow$
$(\exists\ l' \text{ mit } d(l', \text{SwapCol}(j, k, W_{M_{i+1}})) = d(l, W_{M_i}) \leq 3)$.
(nämlich $l' = \text{Swap}(j, k, l)$)

Fall 3) $M_i \overset{SwapRow(j,k,M_i)}{\rightarrow} M_{i+1}$

Die Gleichungen:

- $a_{M_i} \Rightarrow a_{M_{i+1}}$

- $b_{M_i} \Rightarrow b_{M_{i+1}}$

- $c_{M_i} \Rightarrow c_{M_{i+1}}$

sind alle trivial mit Lemma 3.1.3: $W_{M_{i+1}} = W_{M_i}$.

Nun ist gezeigt, dass $abc_{A_1} \Rightarrow abc_{A_2}$ gilt. Aus Symmetriegründen gilt auch die Umkehrung.

Lemma 3.2.2: Gegeben zwei Matrizen A, B, mit $z_A < z_B$ und $A = B_{1..z_A}$. Dann gilt: $A \equiv A' \Rightarrow \exists B' \equiv B$ mit: $A' = B'_{1..z_{A'}}$ und $maxdist_{B'} = maxdist_B$.

Beweis:
Wende die Symmetrieoperatoren, die A in A' überführen, auf B an. Dadurch werden die oberen z_A Zeilen von B ungeachtet der Zeilen $z_A + 1$ bis z_B zu A' umgeformt. Außerdem bleiben durch die Anwendung der Symmetrieoperatoren alle Zeilenabstände erhalten.

Die Anwendung der Symmetrieoperatoren

Beim Erreichen des in Abschnitt 2.3 genannten Ziels soll uns ein Computerprogramm behilflich sein.

Es steht nun fest: Für eine Menge $M = \{A_1, A_2, ..., A_n\}$ symmetrischer Matrizen reicht es aus, stellvertretend für eine Matrix A_i $(1 \leq i \leq n)$ zu zeigen, dass abc_{A_i} gilt. Diesen Umstand werden wir ausgiebig nutzen. Zuerst indem wir uns bei der Betrachtung auf Gruppen gewisse Normen erfüllender Matrizen beschränken von denen wir wissen, dass zu jeder nicht die Normen erfüllenden Matrix A eine die Normen erfüllende Matrix B existiert mit $A \equiv B$. Neben dem Feststellen, ob abc_A für eine gewisse Matrix A gilt, wird die Erkennung von Symmetrie zwischen Antwortsequenzen eine wichtige Aufgabe des Algorithmus sein. Was die Operatoren SwapCol und SwapRow betrifft, so werden wir im Algorithmus nicht das Vertauschen von Spalten und Zeilen, sondern gleich das Permutieren derselben anwenden, was ja nichts anderes ist, als die gleichzeitige Durchführung mehrerer Vertauschungen.

4 Einschränkung des Berechnungsaufwands

Prinzipiell wäre es möglich, den (9,12)P-Algorithmus die Antworten für die für 13 Elemente möglichen 13 Eingaben (Fragen) in willkürlicher Reihenfolge berechnen zu lassen.

So könnte man 13! sich nur in der Position der $*$e unterscheidende Antwortsequenzen erhalten, denn so viele Möglichkeiten gibt es, die $*$e auf die Spalten zu verteilen. Wie in der Einleitung erwähnt, wollen wir jedoch die $*$e in der Diagonalen haben, was wir durch eine vorgegebene Reihenfolge für die Eingaben für A leicht erreichen können. Dennoch gibt es noch immer 2^{12*13} Möglichkeiten, die übrigen Matrizeneinträge mit Bitwerten zu versehen. Wir wollen zeigen, dass wir weitaus weniger Matrizen betrachten müssen.

Einschränkung auf Antwortsequenzen A mit $maxdist_A \leq 6$

Für jede Matrix A mit $maxdist_A \geq 7$ gilt a_A, d.h. wenn der maximale, tatsächlich vorkommende Zeilenabstand von A mindestens 7 ist, kann diese Matrix niemals von einem korrekten (9,12)-Algorithmus ausgegeben werden, weil $W_A = \emptyset$ ist. Seien nämlich $A(i)$ und $A(j)$ zwei Zeilen von A mit $d(A(i), A(j)) \geq 7$, dann gilt $\forall\, w \in \{\ 0{,}1\ \}^{13}$: $d(w, A(i)) > 3 \vee d(w, A(j)) > 3$.

Einteilung der Antwortsequenzen in maxdist-Gruppen

Wir müssen also nur vollständige Antwortsequenzen mit $maxdist_A \leq 6$ betrachten. Diese teilen wir nun in 7 Klassen „$maxdist = i$“ $(0 \leq i \leq 6)$, mit „$maxdist = i$“ $:= \{A \mid maxdist_A = i\}$ ein. Eine Antwortsequenz gehört also genau dann zur Menge „$maxdist = i$“, wenn ihr maximaler, tatsächlich vorkommender Zeilenabstand i beträgt.

4.1 Normierte Matrizen

Nun definieren wir Normen für Antwortsequenzen. Wir sagen eine Matrix ist normiert, wenn sie den folgenden drei Normen gehorcht:

Norm 1: „Ein normiertes maxdist-Paar in Zeile 1 und 2“

Eine Matrix A erfüllt Norm 1, wenn gilt:

- $A(1) = 0^{12}*$
- $A(2) = 1^{maxdist_A}0^{11-maxdist_A}*0$

(Beachte, dass damit gilt: $d(A(1), A(2)) = maxdist_A$)

Norm 2: „∗e in der Diagonale“

Eine Matrix A erfüllt Norm 2, wenn gilt:

$\forall\ 1 \leq i \leq z_A$: $A(i)(14-i) = *$.

In jeder Zeile i soll also der $*$ an Position $14-i$ stehen, das heißt die ∗e sollen in der Diagonale von rechts oben nach links unten der Matrix stehen.

Wir sagen eine Matrix A erfüllt Norm 2 bis (einschließlich) in Zeile n, wenn $A_{1..n}$ Norm 2 erfüllt.

Norm 3: „1en-links-Regel für bislang identische Spalten“

- Jede Matrix A mit $z_A = 1$ erfüllt Norm 3.

- Eine Matrix A mit $z_A > 1$ erfüllt Norm 3, wenn die folgenden beiden Kriterien erfüllt sind:

1) $A_{1..(z_A-1)}$ erfüllt Norm 3. (Also A erfüllt Norm 3, wenn man die letzte Zeile streicht.)

2) Für alle Paare von Spalten i, j mit $i < j$, die in $A_{1..(z_A-1)}$ identisch sind, gilt: $A(z_A, i) = 0 \Rightarrow A(z_A, j) = 0$.

Das bedeutet, wenn in A eine Gruppe von Spalten existiert, die bis zu einer gewissen Zeile identisch sind, dann müssen Einsen und Nullen in der folgenden Zeile stets so verteilt werden, dass innerhalb einer solchen Gruppe die Einsen links stehen.

Wir sagen eine Matrix A erfüllt Norm 3 bis (einschließlich) in Zeile n, wenn $A_{1..n}$ Norm 3 erfüllt.

4.2 Einschränkung auf normierte Matrizen

Wir wissen: Es genügt alle Matrizen der 7 maxdist-Gruppen zu betrachten. Nun wollen wir zeigen, dass es ausreicht, diejenigen vollständigen Matrizen aus den maxdist-Gruppen zu betrachten, die wir nach der Definition im letzten Abschnitt als normiert bezeichnen.
Die Behauptung folgt direkt aus den Lemmata 3.2.1 ($(A \equiv A') \Rightarrow (abc_A \Leftrightarrow abc_{A'})$) und 4.2.1.

Lemma 4.2.1 : Zu jeder vollständigen Matrix A, die nicht normiert ist, existiert eine normierte vollständige Matrix A' mit $A \equiv A'$.

Beweis:

Sei A eine Matrix, die nicht Norm 1 genügt.

Dann können wir mit höchstens zwei Zeilenvertauschungen dafür sorgen, dass für die obersten beiden Zeilen von A gilt: $d(A(1), A(2)) = maxdist_A$.

Im nächsten Schritt sorgen wir mit Spaltenvertauschungen dafür, dass $A(1)(13) = *$ und $A(2)(12) = *$ gilt.

Schließlich flippen wir alle Spalten i mit $A(1)(i) = 1$, und falls $A(2)(13) = 1$ ist, flippen wir auch Spalte 13.

Weil $d(A(1), A(2)) = maxdist_A$ ist und $d_{12..13}(A(1), A(2)) = 0$ ist (wegen der $*$e), muss $d_{1..11}(A(1), A(2)) = maxdist_A$ gelten, und damit muss $A(2)$ in den ersten 11 Bits genau $maxdist_A$ Einsen enthalten, die wir abschließend durch Spaltenvertauschungen nach links bringen können.

Somit ist Norm 1 erfüllt. Beachte, dass Norm 2 und Norm 3 somit bis Zeile 2 erfüllt sind.

Sei A eine Matrix, die Norm 1 genügt und für die gilt:

$A_{1..n}$ genügt Norm 2 und 3, aber $A_{1..(n+1)}$ genügt nicht Norm 2.

Betrachten wir $A(n+1)$. Dies ist die oberste Zeile von A, die Norm 2 verletzt, d.h. $A(n+1)(14-(n+1)) \neq *$. Weil in jeder Spalte ein $*$ steht und weil der obere Teil von A ($A_{1..n}$) Norm 2 entspricht, muss der $*$ in Spalte $14-(n+1)$ in einer der unteren Zeilen vorkommen. Diese Zeile vertauschen wir mit Zeile $n+1$.

Somit erfüllt A Norm 2 bis mindestens Zeile $n+1$ und Norm 3 noch immer mindestens bis Zeile n, denn die Anwendung dieses Verfahrens verändert nicht die Zeilen $A(1)$ bis $A(n)$, in diesem Teil bereits bestehende Normen bleiben also erhalten.

$A_{1..n}$ genügt Norm 2 und 3, aber $A_{1..(n+1)}$ genügt nur Norm 2 und nicht Norm 3.

Betrachten wir $A(n+1)$. Dies ist die oberste Zeile von A, die Norm 3 verletzt, d.h. $\exists\ i, j$ mit $i < j$: Spalte i und j sind in $A_{1..n}$ identisch und $A(n+1)(i) = 0 \wedge A(n+1)(j) = 1$. Sicherlich können wir aber die Spalten von A so umsortieren, dass Norm

3 bis einschließlich Zeile $A(n+1)$ erfüllt ist. Beachte, dass wir dafür nur Spalten vertauschen müssen, die in $A_{1..n}$ identisch sind. Somit ändern sich die Zeilen $A(1)$ bis $A(n)$ nicht, und damit bleiben bestehende Normen in diesem Teil erhalten.

Somit erfüllt A nun Norm 2 und 3 mindestens bis Zeile $n+1$.

Es ist offensichtlich, dass wir durch wiederholtes Anwenden unserer „Korrekturverfahren" A in eine normierte Matrix A' überführen können, und weil wir nur Symmetrieoperatoren anwenden, gilt nach Definition $A \equiv A'$.

4.3 Einschränkung auf Matrizen A mit $maxdist_A \in \{1,2,3,4\}$

• $maxdist$ **= 0:** Da Zeile 1 und 2 nach den Abschnitten 4.1 und 4.2 wie in der folgenden Matrix vorgegeben sind und sich weitere Antworten von den ersten beiden in keinem Bit unterscheiden dürfen, ergibt sich für $maxdist = 0$ die Matrix:

$$
\begin{array}{c}
0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,*\\
0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,*\,0\\
\vdots\\
0\,*\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\\
*\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0
\end{array}
$$

Behauptung: $l = 0000000000000$ ist dann eine Lösung.

Beweis: Angenommen, l wäre keine Lösung, dann muss die tatsächliche charakteristische Funktion mindestens vier 1en enthalten. Egal an welche Bitpositionen wir diese 1en stellen, es gibt immer eine Antwort, die keine der Positionen mit einem $*$ abdeckt und die somit 4 Fehler enthalten hätte.

Wir können diesen Fall also als erledigt betrachten und im Algorithmus nur $maxdist \in \{1,...,6\}$ prüfen.

Beachte: Um die Korrektheit unserer Aussagen für den Fall $maxdist = 0$ nicht speziell für (9,12)P $\subseteq$ (10,13)P, sondern für $(9+k, 12+k)$P $\subseteq$ $(10+k, 13+k)$P zu beweisen, muss man nur die Matrix und den Lösungsvektor um k Nullspalten, bzw. k Nullen erweitern.

• $maxdist \in \{\mathbf{5,6}\}$**:**

Wenn sich die ersten beiden Zeilen einer Matrix in den Bits 1 bis 5 oder 1 bis 6 Bits unterscheiden, macht eine von beiden in diesen Bits 3 Fehler gegenüber jeder möglichen Wahrheit. Das bedeutet, dass die Bits 7 bis 13 (also insbesondere 7 bis 11) in einer der beiden Zeilen korrekt sein müssen. Die Bits 7 bis 11 sind aber in beiden Zeilen identisch, nämlich gleich 0. Wir wissen also ein Bit mit Sicherheit und können die Lösung wie in Fall b) bilden. Natürlich greift diese Argumentation auch bei größeren Matrizen.

4.4 Baumstruktur und Pfadabbruchbedingungen

Wir müssen also für jedes $maxdist \in \{1, ..., 4\}$ alle vollständigen, normierten Antwortsequenzen betrachten und zeigen, dass für jede solche Matrix mindestens einer der Fälle a), b) oder c) eintritt. Für eine bessere Anschaulichkeit der nachfolgenden Überlegungen werden wir für jedes $maxdist$ einen Baum entwickeln, dessen Knoten die genannten Antwortsequenzen repräsentieren.

Die Wurzel W sei dabei ein Knoten, der als Beschriftung die zweizeilige Matrix, die Norm 1 gehorcht, enthält, und mit M_W meinen wir die Matrix, mit der eine solche Wurzel beschriftet ist. Die Wurzel sei auf Tiefe 2 und jeder weitere Knoten wie üblich auf der inkrementierten Tiefe seines Vaterknotens. Die unübliche Wahl der Tiefe 2 für die Wurzel rechtfertigen wir damit, dass so die Tiefe eines Knotens mit der Anzahl der Zeilen der durch den Knoten definierten Matrix (Definition folgt) übereinstimmt.

Jeder weitere Knoten K im Baum wird eine Zeile $\alpha(K)$ als Beschriftung erhalten, und mit jedem solchen Knoten K in Tiefe h identifizieren wir eine Matrix M_K mit $z_{M_K} := h$, die wir wie folgt rekursiv über den Vaterknoten K' definieren: $M_{K_{1..(h-1)}} := M_{K'}$ und $M_K(h) := \alpha(K)$.

Die Wurzel W und induktiv jeder Knoten K erhalte nun Kinderknoten nach dem folgenden Prinzip:
Für jede mögliche Zeile a, die wir an M_K anhängen können, ohne dass die entstehende Matrix die Normierung verletzt oder $maxdist$ verändert wird, hängen wir an K einen Kinderknoten K' mit Beschriftung $alpha(K') = a$ an.

Wir beschränken den Baum bis Tiefe 13. Somit gilt:
$\{M_B \mid B$ ist Blatt im Baum für $maxdist = i\ \}$
$= \{M \mid M$ ist vollständige, normierte Antwortsequenz mit $maxdist_M = i\ \}$

Wir müssen also für alle Blätter B des Baumes zeigen, dass abc_{M_B} gilt.

Beachte hierbei folgende Pfadabbruchbedingungen:

1) Wenn wir bei der Erstellung des Baumes auf einen Knoten K mit abc_{M_K} treffen, können wir auf den Aufbau des Teilbaumes mit Wurzel K verzichten, weil wir durch Lemma 2.3.1 wissen, dass dann auch für alle M_B mit B ist Blatt im Teilbaum mit Wurzel K abc_{M_B} gilt.

2) Wenn wir bei der Erstellung des Baumes für zwei Knoten K und K' auf gleicher Tiefe feststellen: $K \equiv K'$, dann können wir auf den Aufbau eines der beiden Teilbäume mit Wurzel K bzw. K' verzichten, denn:
Nach Lemma 3.2.2 existiert für jede vollständige, normierte Antwortsequenz M, die Blatt im Teilbaum mit Wurzel K ist und die somit aus K durch Anfügen von Zeilen entsteht, eine vollständige, normierte Antwortsequenz M' mit $M' \equiv M$, die aus K' durch Anfügen von Zeilen entsteht und die somit Blatt im Teilbaum mit Wurzel K' ist (und umgekehrt).

5 Realisierung des Baumaufbaus

5.1 Aufgabe des Algorithmus

Der Algorithmus soll sequentiell für jedes $maxdist \in \{ 1,...,4 \}$ den Antwortsequenzen-Baum, der im letzten Kapitel vorgestellt wurde, in Breitensuche aufbauen. Wie im letzten Kapitel begründet, können wir uns den Aufbau eines Teilbaumes mit Wurzel K ersparen, falls:

- K so mit einer Zeile beschriftet ist, dass M_K nicht normiert wäre. (Siehe Lemma 4.2.1.)
- Für M_K einer der Fälle a), b), c) zutrifft. (Siehe Lemma 2.3.1.)
- Wir für einen Knoten K', und die Matrix $M_{K'}$ die er repräsentiert, Symmetrie $M_K \equiv M_{K'}$ feststellen, unter der Voraussetzung, dass wir den Teilbaum mit Wurzel K' weiter aufbauen. (Siehe Lemmata 3.2.1 und 3.2.2.)

Es würde ausreichen zu zeigen, dass in keinem maxdist-Baum ein Knoten K mit „M_K ist vollständige Matrix, für die keiner der Fälle a) b) c) zutrifft“, vorkommt. Tatsächlich werden wir sogar feststellen, dass keiner der maxdist-Bäume jemals bis Tiefe 13 (dort befänden sich die Matrizen mit 13 Zeilen) aufgebaut wird, weil auf jedem Pfad $(W, K_1, K_2, ..., B)$ von der Wurzel zu einem Blatt schon vorher für eine Matrix M_{K_i} Fall a) b) oder c) festgestellt wird.

5.2 Erweiterte Datenhaltung in der Implementierung

Der Algorithmus baut die maxdist-Bäume nacheinander in Breitensuche auf. Dabei weicht er von unseren theoretischen Überlegungen lediglich in der Datenhaltung ab: Um die Berechnung zu beschleunigen werden wir für jede Tiefe des Baumes eine Liste erstellen, deren Elemente folgende Daten enthalten:

- Eine Matrix, nämlich für jeden Knoten K die Matrix M_K, nicht nur $\alpha(K)$.
- Eine „Wahrheitsverwaltung“ mit:
  - der Menge W_{M_K}
  - der Anzahl der möglichen Wahrheiten $|\mathrm{W}_{M_K}|$
  - einem Vektor EZV (Einsen-Zähl-Vektor) mit 13 Komponenten (vom Typ Integer) wie folgt:

$$EZV(i) = \sum_{w \in W_{M_K}} w(i)$$

Jeder maxdist-Baum wird ausgehend von der Wurzel, die mit der Matrix

$$\begin{array}{c} 0000\ 0000\ 0000\ * \\ 1^{maxdist}0^{11-maxdist}*0 \end{array}$$

beschriftet ist, aufgebaut.

Wir initialisieren also die Liste für Tiefe 2 mit einem Element, das diese Matrix wie angegeben gemäß Kapitel 4 und eine initiale Wahrheitsverwaltung wie folgt erhält:

Initiale Wahrheitsverwaltung:
Durchlaufe alle 2^{13} 13-Bit-Vektoren und nimm diejenigen in W_{init} auf, die von M_K einen Abstand ≤ 3 haben.

Für jede zu W hinzukommende Wahrheit w:

- Inkrementiere $|\mathrm{W}_{M_K}|$;
- Falls w(i) = 1 ist inkrementiere EZV(i);

Nun ist die Liste für Tiefe 2 fertig initialisiert. Unser Algorithmus wird von nun an ausgehend von der Liste für die vorangegangene Tiefe $(n-1)$ wie folgt die nächste Tiefe n erstellen:

Wir durchlaufen alle 2^{13} 13-Bit-Vektoren und erstellen aus jedem eine potentielle Folgeantwort a, indem wir das Bit $14-n$ durch einen $*$ ersetzen.

Wir erklären diese potentielle Folgeantwort a sogleich für unzulässig falls:

- $d(a, M_K) >$ maxdist ist, oder
- Das Anfügen von a an M_K Norm 3 (die 1en-links-Regel) verletzen würde.

Wenn a zulässig ist, ziehen wir einen Kinderknoten K' für K in Betracht mit $\alpha(K') = a$. In diesem Fall werden wir a an M_K anfügen und die Wahrheitsverwaltung für $M_{K'}$ wie folgt berechnen:

$W_{M_{K'}} := W_{M_K} \setminus \{w \in W_{M_K} \mid d(w, a) > 3\}$.

Für jede aus W wegfallende Wahrheit w:

- Dekrementiere $|\mathrm{W}_{M_K}|$;
- Falls $w(i) = 1$ ist dekrementiere EZV(i);

Anhand der Wahrheitsverwaltung prüfen wir dann, ob für $M_{K'}$ Fall a) b) oder c) eintritt. Ist dies der Fall, so können wir uns die Weiterentwicklung von $M_{K'}$ sparen und nehmen die Matrix deshalb nicht in die neue Liste auf.

Falls keiner der Fälle a) b) c) eintritt, müssen wir $M_{K'}$ - oder eine dazu symmetrische Matrix - weiterentwickeln. Wir prüfen daher für alle bereits in der neuen Liste eingetragenen Matrizen (die wir ja auf jeden Fall weiter betrachten werden), ob eine darunter symmetrisch zu $M_{K'}$ ist. Nur falls dies nicht der Fall ist, wird $M_{K'}$ tatsächlich in die neue Liste für Tiefe n aufgenommen.

Die aufwändige Symmetrieprüfung übernimmt der im nächsten Kapitel vorgestellte Matcher-Algorithmus.

6 Der Matcher / Effiziente Erkennung von Symmetrien von Antwortsequenzen

Wie in Kapitel 3 erklärt wurde, sind zwei Antwortsequenzen zueinander symmetrisch, wenn sie durch die Symmetrieoperatoren Spaltenvertauschung, Spaltenbitflipping oder Zeilenvertauschung ineinander übergeführt werden können.

Wir werden im Verlaufe unseres Beweises häufig prüfen, ob sich eine (normierte) Matrix A_1 mittels wiederholter Anwendung der Symmetrieoperatoren in eine (normierte) Matrix A_2 verwandeln lässt. Wir müssen dafür nicht alle Möglichkeiten, die Symmetrieoperatoren auf A_1 anzuwenden, prüfen, denn tatsächlich hängen die Symmetrieoperatoren stark zusammen, wenn man sich bereits bei Ihrer Anwendung vor Augen hält, dass die gewonnene Matrix wieder unserer Normierung gehorchen soll. (Sonst bestünde ja keine Chance, A_2 zu erzeugen.)

6.1 Der Zusammenhang der Symmetrieoperatoren

Norm 2 verbindet Zeilen- und Spaltentausch.

Bei dem Versuch, mittels der Symmetrieoperatoren eine normierte Matrix A_1 in eine normierte Matrix A_2 zu überführen, müssen, weil A_2 unverändert bleibt, die $*$e in der durch Anwendung von Symmetrieoperatoren aus A_1 entstandenen Matrix wieder in jeder Zeile i an Position $14 - i$ stehen. (Also von rechts oben an diagonal nach links unten verlaufen.) Deswegen werden Spaltenvertauschungen nur innerhalb der zwei Gruppen: Spalten mit $*$ und Spalten ohne $*$, aber nicht gruppenübergreifend erlaubt sein. Während wir außerdem Spaltenvertauschungen innerhalb der Spalten ohne $*$ ohne weitere Konsequenzen durchzuführen erlauben, sollen Spaltenvertauschungen innerhalb der Spalten mit $*$ automatische Zeilenvertauschungen nach sich ziehen. Genauer: Vertauscht man zwei Spalten i, j mit $(14 - z_A) \leq i, j \leq 13$ (also Spalten mit $*$), dann sollen automatisch die Zeilen $(14 - i)$ und $(14 - j)$ vertauscht werden.

Durch die Vorgabe, die $*$e nach jeder Symmetrieoperation immer in der Diagonalen stehen zu haben, werden Spaltenvertauschungen und Zeilenvertauschungen gekoppelt. Wir erlauben deshalb als Symmetrieoperator nur noch das Vertauschen von Spalten, beachte aber, dass wir durch die Kopplung noch immer (im erlaubten Rahmen) gezielt Zeilen vertauschen können.

Beispiele:

1) Vertauschen zweier Spalten ohne $*$, hier Spalte 3 und 8:

$$\begin{array}{l} a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_{10}\,a_{11}\,a_{12}\ * \\ b_1\,b_2\,b_3\,b_4\,b_5\,b_6\,b_7\,b_8\,b_9\,b_{10}\,b_{11}\ *\ b_{13} \\ c_1\,c_2\,c_3\,c_4\,c_5\,c_6\,c_7\,c_8\,c_9\,c_{10}\ *\ c_{12}\,c_{13} \\ d_1\,d_2\,d_3\,d_4\,d_5\,d_6\,d_7\,d_8\,d_9\ *\ d_{11}\,d_{12}\,d_{13} \\ e_1\,e_2\,e_3\,e_4\,e_5\,e_6\,e_7\,e_8\ *\ e_{10}\,e_{11}\,e_{12}\,e_{13} \end{array} \quad \overrightarrow{swapCol(3,8)} \quad \begin{array}{l} a_1\,a_2\,a_8\,a_4\,a_5\,a_6\,a_7\,a_3\,a_9\,a_{10}\,a_{11}\,a_{12}\ * \\ b_1\,b_2\,b_8\,b_4\,b_5\,b_6\,b_7\,b_3\,b_9\,b_{10}\,b_{11}\ *\ b_{13} \\ c_1\,c_2\,c_8\,c_4\,c_5\,c_6\,c_7\,c_3\,c_9\,c_{10}\ *\ c_{12}\,c_{13} \\ d_1\,d_2\,d_8\,d_4\,d_5\,d_6\,d_7\,d_3\,d_9\ *\ d_{11}\,d_{12}\,d_{13} \\ e_1\,e_2\,e_8\,e_4\,e_5\,e_6\,e_7\,e_3\ *\ e_{10}\,e_{11}\,e_{12}\,e_{13} \end{array}$$

2) Vertauschen zweier Spalten mit $*$, hier Spalte 10 und 12:

$$\begin{array}{l} a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_{10}\,a_{11}\,a_{12}\ * \\ b_1\,b_2\,b_3\,b_4\,b_5\,b_6\,b_7\,b_8\,b_9\,b_{10}\,b_{11}\ *\ b_{13} \\ c_1\,c_2\,c_3\,c_4\,c_5\,c_6\,c_7\,c_8\,c_9\,c_{10}\ *\ c_{12}\,c_{13} \\ d_1\,d_2\,d_3\,d_4\,d_5\,d_6\,d_7\,d_8\,d_9\ *\ d_{11}\,d_{12}\,d_{13} \\ e_1\,e_2\,e_3\,e_4\,e_5\,e_6\,e_7\,e_8\ *\ e_{10}\,e_{11}\,e_{12}\,e_{13} \end{array} \quad \overrightarrow{swapCol(10,12)} \quad \begin{array}{l} a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_{12}\,a_{11}\,a_{10}\ * \\ b_1\,b_2\,b_3\,b_4\,b_5\,b_6\,b_7\,b_8\,b_9\ *\ b_{11}\,a_{10}\,b_{13} \\ c_1\,c_2\,c_3\,c_4\,c_5\,c_6\,c_7\,c_8\,c_9\,c_{12}\ *\ c_{10}\,c_{13} \\ d_1\,d_2\,d_3\,d_4\,d_5\,d_6\,d_7\,d_8\,d_9\,d_{12}\,d_{11}\ *\ d_{13} \\ e_1\,e_2\,e_3\,e_4\,e_5\,e_6\,e_7\,e_8\ *\ e_{12}\,e_{11}\,e_{10}\,e_{13} \end{array}$$

$$\overrightarrow{swapRow(2,4)} \quad \begin{array}{l} a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8\,a_9\,a_{12}\,a_{11}\,a_{10}\ * \\ d_1\,d_2\,d_3\,d_4\,d_5\,d_6\,d_7\,d_8\,d_9\,d_{12}\,d_{11}\ *\ d_{13} \\ c_1\,c_2\,c_3\,c_4\,c_5\,c_6\,c_7\,c_8\,c_9\,c_{12}\ *\ c_{10}\,c_{13} \\ b_1\,b_2\,b_3\,b_4\,b_5\,b_6\,b_7\,b_8\,b_9\ *\ b_{11}\,a_{10}\,b_{13} \\ e_1\,e_2\,e_3\,e_4\,e_5\,e_6\,e_7\,e_8\ *\ e_{12}\,e_{11}\,e_{10}\,e_{13} \end{array}$$

Beachte, dass man so (indem man Spalte 10 und 12 vertauscht) auch gezielt Zeile 2 und 4 hätte vertauschen können.

Jede Symmetrie ist durch Festlegung der Zeilenpermutation, anschließendes Bitflipping und abschließende Spaltenvertaschungen innerhalb des linken Teils einer Matrix herbeiführbar.

Wir begnügen uns hier mit einer kurzen Begründung, weil die Richtigkeit des Verfahrens nicht von der Richtigkeit dieser Behauptung abhängt - schließlich sind wir nicht verpflichtet, alle Symmetrien zu entdecken. Weil sich durch Spaltenvertauschungen (und auch durch damit verbundene Zeilenvertauschungen) die Zusammensetzung der Bits einer Spalte nicht ändert, kann man jedes Spaltenbitflipping auch durchführen, nachdem alle Zeilenvertauschungen abgeschlossen sind. Wir werden deshalb vorgeben, dass zuerst die Zeilenanordnung festgelegt werden muss, und erst anschließend Bitflipping durchgeführt wird.

Für A_2 gilt Norm 1 $\Rightarrow$ Erste Zeile legt Bitflipping fest

Zu dem Zeitpunkt da wir Bitflipping einsetzen ist die Anordnung der Zeilen (und damit auch die der $*$-Spalten) endgültig festgelegt. Da die ersten beiden Zeilen wie in Abschnitt 4.1 gefordert aussehen müssen, ist auch das Bitflipping festgelegt in dem Sinne, dass genau die Spalten, deren oberstes Bit (das in Zeile 1, bzw. für Spalte 13 in Zeile 2 steht) auf 1 gesetzt ist, geflippt werden.

6.2 Symmetrieerkennung mit festgelegter Reihenfolge für die Symmetrieoperatoren

Wir wollen nun also zuerst die Spaltenvertauschungen durchführen. Da sich damit für die Spalten jede beliebige Reihenfolge herbeiführen lässt, werden wir direkt Permutationen auf die Spalten anwenden. Ein naiver Ansatz hierfür wäre, für eine Matrix alle 13! Spaltenpermutationen auszuprobieren und diese jeweils mit der anderen zu matchen. Allerdings haben wir bereits in Abschnitt 6.1 begründet, warum es nur sinnvoll ist, die $13 - Zeilenanzahl$ ersten Spalten (die ohne $*$) zu permutieren und die letzten $Zeilenanzahl$ Spalten (die mit $*$) untereinander zu permutieren, was schon einen geringeren Aufwand bedeuten würde.

Da wir aber dann für jede Permutation noch die Zeilenvertauschungen und Bitflipping durchführen müssten, um zu prüfen, ob die Matrizen identisch sind, wollen wir zuerst ein notwendiges, aber nicht hinreichendes Kriterium für die Symmetrie zweier Antwortsequenzen mit einer gewissen Spaltenpermutation P heranziehen:

Symmetrieoperatoren ändern Zeilenabstände nicht $\Rightarrow$ gewisse Permutationen scheiden aus

Weder durch Spaltenvertauschungen, noch durch Bitflipping ändert sich der Abstand zwischen zwei Zeilen einer Matrix. Das bedeutet, wenn wir die paarweisen Zeilenabstände beider Matrizen ermitteln und in zwei Halbmatrizen, die wir Zeilenabstandsmatrizen (oder Zeilenabstandshalbmatrizen) nennen werden, abspeichern, so können wir als notwendige Bedingung festhalten, dass es eine Zeilenpermutation für jede der beiden Matrizen geben muss, die die Zeilenabstandsmatrix dieser Matrix in die Zeilenabstandsmatrix der anderen überführt.

Da wir insbesondere nur die Spalten ohne $*$, bzw. die mit $*$ untereinander vertauschen dürfen, können wir sogar für jede der Matrizen zwei Zeilenabstandsmatrizen erstellen, eine für den linken Teil (erste $13 - Zeilenanzahl$ Spalten) und eine für den rechten Teil (letzte $Zeilenanzahl$ Spalten). Wieder muss gelten, dass unter einer bestimmten Zeilenpermutation der einen Matrix die linken und die rechten Zeilenabstandsmatrizen paarweise identisch werden.

Vorweg wollen wir prüfen, ob überhaupt die Chance besteht, die Abstandsmatrizen mittels Zeilenpermutation und daraus resultierender Umstellung der Einträge ineinander überzuführen. Da die Einträge durch die Zeilenvertauschungen ja nur innerhalb der Matrix durchgetauscht werden

können, müssen auf jeden Fall von Anfang an dieselben Einträge in den Matrizen gleich oft vorkommen. Wir speichern die Häufigkeit der Einträge in sogenannten Eintragszählvektoren (nicht zu verwechseln mit den in Abschnitt 5.2 vorgestellten Einsenzählvektoren EZV). Ein Eintragszählvektor ist ein $(maxdist + 1)$-elementiger Vektor. Für jeden der möglichen Zeilenabstände 0,1,...,maxdist ist eine Komponente reserviert. Hier vermerken wir, wie oft der jeweilige Zeilenabstand in der Zeilenabstandshalbmatrix (kurz: ZAHM) eingetragen ist.

Matrix X:	ZAHM X1:	ZAHM X2:
0 0 0 0 0 0 0 0 0 0 0 0 *	4 2 2 2	0 1 1 1
1 1 1 1 0 0 0 0 0 0 0 * 0	– 2 2 2	– 0 0 1
1 1 0 0 0 0 0 0 0 0 * 1 0	– – 4 2	– – 0 2
0 0 1 1 0 0 0 0 0 * 0 1 0	– – – 2	– – – 1
0 1 1 0 0 0 0 0 * 1 0 0 0		

Eintragszählvektor für X1: (0,0,8,0,2) / Eintragszählvektor für X2: (4,5,1,0,0)

Matrix Y:	ZAHM Y1:	ZAHM Y2:
0 0 0 0 0 0 0 0 0 0 0 0 *	4 2 2 2	0 1 0 1
1 1 1 1 0 0 0 0 0 0 0 * 0	– 2 2 2	– 1 0 2
1 1 0 0 0 0 0 0 0 1 * 0 0	– – 4 2	– – 0 1
0 0 1 1 0 0 0 0 0 * 0 0 0	– – – 2	– – – 1
1 0 0 1 0 0 0 0 * 1 0 0 1		

Eintragszählvektor für Y1: (0,0,8,0,2) / Eintragszählvektor für Y2: (4,5,1,0,0)

Wie man sieht, sind in diesem Beispiel die Eintragszählvektoren identisch ($X1 = Y1 \wedge X2 = Y2$), das heißt wir können nicht ausschließen, dass eine Zeilenpermutation existiert, unter der die Zeilenabstandshalbmatrizen identisch werden.

Nachdem wir die Zeilenabstandsmatrizen von Y für alle Zeilenpermutationen erstellt und mit den Zeilenabstandsmatrizen von X verglichen haben, stellen wir fest, dass die einzige Permutation p, die in Frage kommt, die folgende ist: $\frac{12345}{34215}$. Wir werden eine solche Permutation im folgenden kurz mit 34215 notieren. Dies ist insbesondere für das Verständnis der Ausgabe des Algorithmus (Anhang B) wichtig.

Gibt es keine Permutation der Zeilen einer Matrix, die die Zeilenabstandsmatrizen identisch macht, dann können die Matrizen nicht zu einer Äquivalenzklasse gehören, denn dann kann es uns insbesondere nicht gelingen, durch Spaltenvertauschungen in den ersten 8 Spalten oder durch Bitflipping die Zeilen paarweise (matrizenübergreifend) aneinander anzugleichen. Deswegen ist die eben vorgestellte Bedingung eine notwendige.

Für alle Permutationen, die die Forderung erfüllen (in unserem Fall ausschließlich p), führt man dann die entsprechenden Spaltenvertauschungen der $*$-Spalten durch, so dass die gewünschte Permutation der Zeilen erreicht wird. Dann flippt man alle Spalten so, dass Zeile $1 = 0000\ 0000\ 0000 * $ ist, und dass die zweite Spalte im letzten Bit eine 0 stehen hat. Wir dürfen nun keine Spalten mehr flippen, sonst würden wir die Bedingung für Zeile 1 oder 2 zerstören.

Außerdem haben wir alle Spaltenvertauschungen für den rechten Teil der Matrix Y (Spalten mit $*$) abgehandelt. Dort dürfen wir die Spaltenreihenfolge nicht mehr ändern, sonst stehen die $*$e nicht mehr in der Diagonalen. Das bedeutet, dass die rechten Teile der Matrizen zu diesem Zeitpunkt übereinstimmen müssen, sonst war die Permutation nicht geeignet und wir gehen zur nächsten Permutation aus unserer Liste über.

Falls dann der rechte Teil der Matrizen X und Y übereinstimmt, widmen wir uns nun dem linken Teil. Man beachte, dass der Abstand der ersten beiden Zeilen im linken Teil der Matrix X *maxdist* beträgt. Das muss auch der Abstand im linken Teil unserer nach der Permutation p abgewandelten Matrix Y sein. Da wir die Spalten so geflippt haben, dass der linke Teil der ersten Zeile nur 0en enthält, gilt: Es ergeben sich auf jeden Fall *maxdist* 1en im linken Teil in Zeile 2 der nach Permutation p abgewandelten Matrix Y. Wir vertauschen die Spalten so, dass diese 1en in Zeile 2 ganz links stehen.

Abschließend versuchen wir durch Vertauschung der ersten *maxdist* Spalten untereinander und der Spalten (*maxdist*+1) bis 13-z_Y (das ist die rechteste Spalte ohne $*$) untereinander die Gleichheit der Matrizen zu erreichen. Diese Prüfung vollziehen wir nicht durch das aufwändige Durchprobieren aller Permutationen, sondern sehr viel schneller:

Für jede Spalte der Matrix X durchlaufen wir die Spalten der Matrix Y von links nach rechts bis wir eine zu zur Spalte von X identische Spalte in Y gefunden haben und sortieren die Spalte in Y an die Position an der wir uns gerade in X befinden. Wenn es mehrere solche Spalten gibt nehmen wir die erstbeste und wenn es keine gibt, können wir abbrechen.

7 Erweiterung des Beweises (9,12)P = (10,13)P auf alle Klassen mit 3 Fehlern und mehr Elementen.

Es soll gezeigt werden, dass sich der durch den Algorithmus mittels der Auflistung aller (bis auf Symmetrien) möglichen Antwortsequenzen erbrachte Beweis (9,12)P $\subseteq$ (10,13)P induktiv für alle $k \geq 1$ zu $(9+k, 12+k)\text{P} \subseteq (10+k, 13+k)\text{P}$ erweitern lässt. Weil ja bereits in der Einleitung die Gegenrichtung gezeigt wurde, wäre damit $(9+k, 12+k)\text{P} = (10+k, 13+k)\text{P}$ für alle $k \geq 1$ gezeigt.

7.1 Vorbemerkungen

Lemma 7.1.1: $A \equiv B \Rightarrow A^+ \equiv B^+$.

Beweis:
Man kann nach Voraussetzung A mittels Spaltenvertauschungen und daraus resultierenden Zeilenvertauschungen und daraus resultierendem Bitflipping in B überführen. Nun fügen wir an beide Matrizen ganz rechts eine weitere Nullspalte an. Um zu zeigen, dass $A^+ \equiv B^+$ gilt, wenden wir auf A^+ einfach die Operatoren an, die A in B überführen. Dies wirkt sich auf den Teil von A^+ der A entspricht, also A^+ ohne die rechteste Spalte, so aus, dass dieser Teil B wird. Da die rechteste Spalte außerdem unverändert bleibt (durch Zeilenvertauschungen bleibt die Nullspalte erhalten und Spaltenoperationen greifen gar nicht darauf zu, weil sie den Spaltenindex nie benutzen), haben wir damit B^+ erzeugt.

Lemma 7.1.2: Kennt man ein Bit von C_{V_T} (Fall b)), so weiß man dass alle Nullspaltenbits in C_{V_T} den Wert 0 haben.

Beweis:
Wir kennen ein Bit von C_{V_T} bedeutet, dass alle möglichen Wahrheiten in diesem Bit übereinstimmen. Das bedeutet aber, dass keine Wahrheit existiert, die ein Nullspaltenbit auf 1 gesetzt hat. Sonst könnte man nämlich dieses Nullspaltenbit (ein Fehler weniger) zusammen mit dem 'bekannten' Bit (ein Fehler mehr) invertieren und hätte eine mögliche Wahrheit, die im Widerspruch dazu steht, dass alle Wahrheiten in diesem Bit übereinstimmen. Also müssen in jeder möglichen Wahrheit und damit auch in C_{V_T} alle Nullspaltenbits den Wert 0 haben.

7.2 Wahrheitsgerüste

Häufig wird es vorkommen, dass eine betrachtete Antwortsequenz M eine oder mehrere Nullspalten enthält. Diese Nullspalten werden (bedingt durch die Normierung) immer nebeneinander innerhalb der Spalten $(maxdist + 1)$ bis $(13 - z_M)$ vorkommen. Seien in einer Matrix M nun tatsächlich n Nullspalten vorhanden, nämlich die Spalten i (mit $(maxdist+1) \leq i \leq (13-z_M)$) bis $j = (13-z_M)$. Dann können wir für die Menge W_M den Begriff des Wahrheitsgerüstes wie folgt einführen:

Betrachten wir nur die Bits der $w \in W_M$, die nicht auf Höhe der Nullspalten sind, also für jedes w die Bitpositionen 1 bis $(i-1)$ und $j+1$ bis 13. Einen auf diese Bits beschnittenen Vektor w nennen wir dann 'Wahrheitsgerüst' g_w, denn:
Für jedes g_w definieren wir $d(g_w, M) = d_{1..(i-1)}(w, M) + d_{(14-z_M)..13}(w, M)$. Es gilt $d(g_w, M) \leq d(w, M) \leq 3$. Der Abstand eines jeden Wahrheitsgerüstes von M ist also kleiner gleich 3.

Im Gegenzug definieren wir für jedes solche Wahrheitsgerüst wieder einen Vektor der ursprünglichen Stelligkeit:
Für jedes g_w aus den Wahrheitsgerüsten gehört $w_g \in \{\ 0{,}1\ \}^{13}$ mit $w_g(k) = 0$ für $k \in \{i..j\}$ und $w_g(k) = g(k)$ sonst, (also g in den „Nullspaltenbits mit 0en aufgefüllt“,) zu W, weil durch die Erweiterung um 0en keine weiteren Fehler gegenüber M entstehen und damit gilt $d(w_g, M) = d(g_w, M) \leq 3$.

Führt man den Gedanken zu Ende, erkennt man, dass sich W_M wie folgt wieder aus den Wahrheitsgerüsten gewinnen lässt:
Betrachte für jedes g_w und jedes w_g wie eben definiert $d(g_w, M)$. Momentan ist $d_{i..j}(w_g, M) = d(0...0, 0...0) = 0$. Mit $d_{i..j}(w_g, M) + d(g_w, M) = d(w_g, M)$, wären also in den Nullspaltenbits von w_g, also in den Bitpositionen i bis j noch $3 - d(g_w, M)$ Fehler erlaubt, in dem Sinne, dass $d(w_g, M) \leq 3$ bleibt.

Das bedeutet, jedes w', das aus w_g durch Bitflipping von bis zu $3 - d(g_w, M)$ Nullspaltenbits hervorgeht (genauer: $max(3 - d(g_w, M), j - i + 1)$, denn es gibt nur $j - i + 1$ Nullspaltenbits), ist eine mögliche Wahrheit.

Man sieht schnell ein, dass auf diese Weise auch tatsächlich alle möglichen Wahrheiten wieder erzeugt werden.

Beispiel:

In M seien die Spalten $i = 6$ bis $j = 9$ Nullspalten.
Sei $w \in W_M = 1100100100100$. Damit ist $g_w = 11001....0100$. Sei weiterhin $d(g_w, M) = 2$. Damit ist $w_g = 1100100000100$ und wir wissen, dass alle $w' \in \{0,1\}^{13}$, die durch bis zu ein Bitflipping in einer der Positionen 6 bis 9 entstehen (also w_g, w_1, w_2, w_3, w_4 wie angegeben) zu W_M gehören:

$w_g = 1\,1\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0$
$w_1 = 1\,1\,0\,0\,1\,1\,0\,0\,0\,0\,1\,0\,0$
$w_2 = 1\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1\,0\,0$
$w_3 = 1\,1\,0\,0\,1\,0\,0\,1\,0\,0\,1\,0\,0$
$w_4 = 1\,1\,0\,0\,1\,0\,0\,0\,1\,0\,1\,0\,0$

7.3 Die Übertragung des Beweises auf die weiteren Komplexitätsklassen

Sei $\widehat{M_k}$ die Menge von Antwortsequenzen, die wir für den Beweis $(9+k, 12+k)\mathrm{P} \subseteq (10+k, 13+k)\mathrm{P}$ für ein gewisses $k \geq 0$ betrachten und von der wir wissen, dass sie alle Matrizen (im Sinne aller vorangegangenen Überlegungen) abdeckt.

Behauptung: Der Beweis (9,12)P = (10,13)P lässt sich induktiv zum Beweis für die Gleichheit aller weiteren Klassen $(9+k, 12+k)\mathrm{P} = (10+k, 13+k)\mathrm{P}$ mit $k \geq 1$ erweitern.

Beweis: Die Aussage folgt unmittelbar aus den folgenden Lemmata 7.3.1 und 7.3.2.

Lemma 7.3.1 Es reicht für den Beweis für jedes k aus, $\widehat{M_k} = \{M^+ \mid M \in \widehat{M_{k-1}}\}$ zu betrachten, weil diese Menge für dieses k alle Matrizen (im Sinne aller vorangegangenen Überlegungen) abdeckt.
Angenommen, die Betrachtung einer so definierten Menge $\widehat{M_k}$ reiche nicht aus, d.h. es existiere eine Matrix A, die wir außer den genannten betrachten müssten, weil sie mit keiner der durch Anhängen einer Nullspalte an eine Matrix M aus $\widehat{M_{k-1}}$ entstandenen Matrix M^+ in einer Äquivalenzklasse ist: $\exists A \in \widehat{M_k} : \forall B \in \widehat{M_{k-1}} : A \notin [B^+]$

Fall 1) Angenommen, A hat mindestens eine Nullspalte, dann streichen wir eine Nullspalte und erhalten A^-. Da A^- eine korrekte Antwortsequenz für ein Element weniger als A ist, muss A^- oder eine zu A^- symmetrische Matrix $B \equiv A^-$ im Beweis für $k-1$ betrachtet worden sein. A^- selbst kann nicht betrachtet worden sein, das stünde im Widerspruch zur Annahme, dass A nicht unter den Matrizen mit eingefügter Nullspalte ist.
Nun ist $B \in \widehat{M_{k-1}}$ und damit $B^+ \in \widehat{M_k}$. Mit $A^- \equiv B$ und Lemma 7.1.1 folgt $(A^-)^+ \equiv B^+$, und weil $(A^-)^+ \equiv A$ (nur die Spaltenreihenfolge kann sich geändert haben) folgt im Widerspruch zur Annahme $A \equiv B^+$.

Fall 2) Angenommen, A hat keine Nullspalte, dann streichen wir eine andere Spalte und wir erhalten eine Matrix A' ohne Nullspalte. Da A' eine korrekte Matrix für ein Element weniger als A ist, muss eine Matrix A'' existieren mit $A'' \equiv A'$ und $A'' \in \widehat{M_{k-1}}$. Man mache sich klar, dass bei der Umwandlung von A' in A'' keine Nullspalte entstehen konnte, weil durch Zeilen und Spaltenvertauschungen und Bitflipping nur eine Einsspalte in ei-

ne Nullspalte umgewandelt werden könnte. Die erste Zeile war aber in dieser Matrix 0...0$*$. Also enthielt jede Spalte mindestens eine 0. (Die letzte Spalte kann nie Nullspalte werden, weil sie einen $*$ enthält.) Also kommt in A'' keine Nullspalte vor. Das aber steht im Widerspruch dazu, dass jede Matrix in $\widehat{M}_{k-1}$ mindestens eine Nullspalte hat. (Für $k = 0$ entnehmen wir dies der Ausgabe des Algorithmus, und für größere k geht es dann induktiv aus dieser Überlegung hervor).

Lemma 7.3.2 Sei A^+ eine Antwortsequenz aus $\widehat{M_k}$, A^+ habe also $13 + k$ Spalten. Dann gilt: $abc_A \Rightarrow abc_{A^+}$.

Fall 1) $W_A = \emptyset \Rightarrow W_{A^+} = \emptyset$

$$
\begin{aligned}
& W_A = \emptyset \\
\Rightarrow\ & \forall\ w \in \{0,1\}^{13+k-1}\text{: } d(w, A) > 3 \\
\Rightarrow\ & \forall w' \in \{(w, ?)\} : d(w', A^+) = d_{1..13+k-1}(w', A^+) + d_{13+k..13+k}(w', A^+) \\
& \qquad\qquad = d(w, A) + d_{13+k..13+k}(w', A^+) \geq d(w, A) > 3. \\
\Rightarrow\ & W_{A^+} = \emptyset
\end{aligned}
$$

Fall 2) $\exists\ n, i : 1 \leq n \leq 13 + k - 1,\ i \in \{0,1\}\text{: } \forall w \in W_A\text{: } w[n] = i$
$\Rightarrow \exists\ m, j : 1 \leq m \leq 13 + k,\ j \in \{0,1\}\text{: } \forall\ w \in W_{A^+} : w[m] = j$

Wie in Lemma 7.1.2 begründet, bedeutet die Kenntnis eines Bits, dass in jedem Wahrheitsgerüst bereits 3 Fehler gegenüber der Antwortsequenz gemacht werden. Das Hinzufügen einer Nullspalte beeinflusst die Wahrheitsgerüste nicht. Jedes Wahrheitsgerüst macht also weiterhin 3 Fehler gegenüber der Antworsequenz. Das bedeutet, dass sich die Wahrheitsgerüste von A nur durch Einfügen von Nullen als Nullspaltenbits zu Wahrheiten für A^+ ergänzen lassen. Wir kennen also die Nullspaltenbits und damit noch immer mindestens ein Bit.

Fall 3) $\exists\ l \in \{0,1\}^{13+k-1}\text{: } d(l, W_A) \leq 3 \Rightarrow d((l, 0),\ W_{A^+}) \leq 3.$

Wir wollen zeigen, dass für eine Antwortsequenz A für $13 + k - 1$ Elemente, der daraus resultierenden Menge der Wahrheiten W_A und einem zugehörigen Vektor l (bei dem wir o.B.d.A. davon ausgehen, dass in den Nullspaltenbits (bezogen auf A) 0en stehen sollen) folgendes gilt:

Erweitern wir die Antwortsequenz A um eine Nullspalte zu A^+ und erweitern wir l an derselben Position um ein Bit, (das wir auf 0 setzen) zu l', so gilt: l' ist Lösung für W_{A^+}.

Beweis:
Betrachte für die Mengen der möglichen Wahrheiten W_A und W_{A^+} die Wahrheitsgerüste, also die enthaltenen Vektoren ohne die Nullspaltenbits. Wir stellen fest, dass die Wahrheitsgerüste von W_A und W_{A^+} dieselben sind, denn es sind alle Vektoren, die von den Antworten ohne die Nullspalten um höchstens 3 Bits abweichen, und A und A^+ sind ohne die Nullspalten identisch. Betrachtet man l und $(l, 0)$ ohne die Nullspaltenbits (bezogen auf A bzw. A^+), so sind diese auch identisch. Das bedeutet, für jedes Wahrheitsgerüst g_w gilt: $d(g_w, g_l) = d(g_w, g_{(l,0)})$.

Letztlich ist also nur zu zeigen, dass zu jeder Wahrheit $w' \in W_{A^+}$ eine Wahrheit $w \in W_A$ existiert, die auf dem gleichen Gerüst basiert, und die in den Nullspaltenbits mindestens gleich viele Fehler gegenüber l macht wie w' in den Nullspaltenbits gegenüber $(l, 0)$ macht. Dann wäre die Gesamtabweichung (die sich natürlich aus der Abweichung im Gerüst und der Abweichung in den Nullspaltenbits zusammensetzt) eines solchen w' gegenüber $(l, 0)$ nie größer als eine von einem gewissen w gegenüber l und wir hätten bewiesen: ist l Lösung für W_A dann ist auch $(l, 0)$ Lösung für W_{A^+}.

Um zu zeigen, dass eine Wahrheit aus W_{A^+} in den Nullspaltenbits nie mehr Fehler gegenüber $(l, 0)$ macht als eine gewisse Wahrheit aus W_A in den Nullspaltenbits gegenüber l macht, betrachten wir für jedes Wahrheitsgerüst g_w folgende Fälle:

Fall 3.1): $d(g_w, A) = 3$

In diesem Fall müssen für jede auf g_w basierende Wahrheit w aus W_{A^+} (und übrigens auch aus W_A) alle Nullspaltenbits auf 0 gesetzt sein, sonst ergäbe sich ein weiterer Fehler, und der auf g_w basierende Vektor wäre keine mögliche Wahrheit mehr. Da sich dies mit unseren Bits in $(l,0)$ deckt, entstehen keine Fehler in den Nullspaltenbits und wir wissen: $d(w, (l, 0)) = d(g_w, g_{(l,0)}) = d(g_w, g_l) \leq 3$ und damit: l' ist Lösung für W_{A^+}.

Fall 3.2) $d(g_w, A) = 2$

In diesem Fall darf jede auf g_w basierende Wahrheit $w \in W_{A^+}$ keine oder eine 1 in den Nullspaltenbits enthalten.

Dass die w, die keine 1 in den Nullspaltenbits haben, der Lösungseigenschaft von $(l, 0)$ nicht widersprechen haben wir bereits in Fall 1) gezeigt.

Die Wahrheiten $w' \in W_{A^+}$, die eine 1 in den Nullspaltenbits enthalten, weichen in den Nullspaltenbits von $(l, 0)$ um Abstand 1 ab. Da aber die Antwortsequenz A auch mindestens eine Nullspalte enthält, existiert auch

ein auf g_w basierendes $w \in W_A$, mit einer 1 in den Nullspaltenbits. Das bedeutet, $d(w', (l, 0)) = d(g_{w'}, g_{(l,0)}) + 1 = d(g_w, g_l) + 1 \leq 3$.

Fall 3.3) $d(g_w, A) = 1$

In diesem Fall darf jede auf g_w basierende Wahrheit aus W_{A^+} keine, eine 1 oder zwei 1en in den Nullspaltenbits enthalten. Man beachte, dass die Gleichung $\mathrm{d}(\mathrm{g}_w, \mathrm{A}) = 1$ bedeutet, dass sich zwei Antworten aus A in maximal zwei Bits unterscheiden können. Wir befinden uns also in einem Fall mit $maxdist \in \{1, 2\}$. Beachte: Alle von unserem Algorithmus berechneten Antwortsequenzen für $maxdist \in \{1, 2\}$ enthalten mindestens zwei Nullspalten, dies ist aus der Ausgabe des Algorithmus zu ersehen.

Dass die Wahrheiten w, die keine 1 bzw. eine 1 in den Nullspaltenbits haben, der Lösungseigenschaft von $(l, 0)$ nicht widersprechen haben wir bereits in Fall 3.1) bzw. Fall 3.2) gezeigt.

Die Wahrheiten $w' \in \mathrm{W}_{A^+}$, die zwei 1en in den Nullspaltenbits enthalten, weichen in den Nullspaltenbits von $(l,0)$ um Abstand 2 ab. Nun dürfte aber eine Wahrheit w in W_A, die auf unserem Wahrheitsgerüst g_w basiert, auch $3 - d(g_w, A) = 2$ 1en in den Nullspaltenbits enthalten und da wie gesagt in W_A wirklich zwei Nullspalten existieren, existiert in W_A auch ein w, das auf g_w basiert und zwei 1en in den Nullspalten hat. Das bedeutet, $d(w', (l, 0)) = d(g_{w'}, g_{(l,0)}) + 2 = d(g_w, g_l) + 2 \leq 3$

(Hätte A nur eine Nullspalte enthalten, dann könnte von den zwei 1en die durch $d(g_w, A)$ in den Nullspaltenbits noch erlaubt wären nur eine vorkommen, und daraus könnten wir lediglich $d(g_w, g_l) \leq 2$ folgern. Da aber dann noch 2 Fehler in den Nullspaltenbits hinzukommen könnten, könnten wir so insgesamt nur $d(w, l) \leq 4$ festhalten und das wäre zu wenig um die Lösungseigenschaft von $(l, 0)$ zu beweisen. Dies ist der Grund dafür, warum wir die Induktion nicht ausgehend von einem direkten Beweis für (8,11)P = (9,12)P begonnen haben.)

Fall 3.4) $d(g_w, A) = 0$

Das bedeutet für jedes Zeilenpaar (a_i, a_j), $(1 \leq i < j \leq z_A)$ von A: $d(g_w, a_i) = d(g_w, a_j) = 0$ und damit $d(a_i, a_j) = 0$, weil a_i und a_j in den Nullspaltenbits natürlich übereinstimmen. Fall ein solches w bzw. g_w existiert, befinden wir uns im Fall $maxdist = 0$, dessen Lösbarkeit trivial ist unabhängig von k. (Siehe hierzu Abschnitt 4.3)

Literatur

[Ros60] Gene F. Rose. An extended notion of computability. In *Abstracts of the International Congress for Logic, Methodology, and Philosophy of Science*, Seite 14, Stanford, California, 1960.

[Tra63] B.A.Trakhtenbrot. On frequency computation of functions. *Algebra i Logika*, 2:25-32, 1963. (Russisch)

[ADHP00] Holger Austinat, Volker Diekert, Ulrich Hertrampf und Holger Petersen. Regular frequency computations. In *Proceedings of the RIMS Symposium on Algebraic Systems, Formal Languages and Computation*, Seiten 35-42, Kyoto, Japan, 2000.

[Kin75] E.B.Kinber. Frequency-computable functions and frequency-enumerable sets. *Candidate Dissertation*, Riga, 1975. (Russisch)

[KS95] Martin Kummer und Frank Stephan. The power of frequency computation. In Horst Reichel, editor, *Proceedings of the Tenth International Congress on Fundamentals of Computation Theory* (FCT 1995), LNCS 969, Seiten 323-332, 1995.

[HW97] Maren Hinrichs und Gerd Wechsung. Time bounded frequency computations. *Information and Computation*. 139(2):234-257, 1997.

A Der Algorithmus in Pseudocode

A.1 Die Funktionen und ihre Aufgaben

1) Breitensuche und Unterfunktionen

breitensuche: Erstellt Tiefe für Tiefe einen vollständigen maxdist-Baum.

initialisiere_wahrheitsverwaltung: Erstellt eine Wahrheitsverwaltung mit Rücksicht auf eine einzelne Zeile.

> - Erstellt ein Feld aller Vektoren mit Abstand ≤ 3 zur betrachteten Zeile.
> - Zählt die möglichen Wahrheiten.
> - Zählt für jede Position quer über alle möglichen Wahrheiten die auf 1 gesetzten Bits.

aktualisiere_wahrheitsverwaltung : Aktualisiert eine Wahrheitsverwaltung mit Rücksicht auf eine neue Zeile.

> - Löscht jede mögliche Wahrheit mit Abstand > 3 zur betrachteten Zeile.
> - Zählt die verbliebenen möglichen Wahrheiten.
> - Zählt für jede Position quer über alle möglichen Wahrheiten die auf 1 gesetzten Bits.

nächste_tiefe: Berechnet die nächste Tiefe eine maxdist-Baumes aus der letzten Tiefe.

fall_abc: Prüft mittels Wahrheitsverwaltung ob einer der Fälle a), b), c) für die Matrix eintritt.

lösungsfindung: Versucht für eine gegebene Menge von Wahrheiten eine Lösung zu finden.

2) Matcher und Unterfunktionen

matrix_matcher: Prüft ob zwei Matrizen symmetrisch sind und gibt, falls Symmetrie vorliegt, die Zeilenpermutation aus, mittels der man die momentan betrachtete in die bereits früher betrachtete symmetrische überführen kann. (Nicht umgekehrt!)
Die Zeilenpermutation ist zur manuellen Kontrolle völlig ausreichend.

passende_permutationen: Erstellt eine Liste von Permutationen für die für zwei Abstandsmatrizen ein Matching erzielt wird.

permutiere_abstandsmatrix: „Permutiert“ eine Abstandsmatrix nicht wirklich, stattdessen wird auf der Abstandsmatrix eine Operation wie folgt ausgeführt: Eine gegebene Abstandsmatrix A' einer Matrix A wird umgewandelt zur Abstandsmatrix B' einer Matrix B die durch Anwenden von p auf A entstehen würde. (Ohne den Umweg über B.)

permutiere_&_restrukturiere_matrix: Wendet eine Zeilenpermutation auf eine Matrix an und vertauscht die Zeilen im rechten Zeil so, dass Norm 1 ggf. wiederhergestellt wird.

matrizenkopf_wiederherstellen: Führt Bitflipping und Zeilenvertauschungen aus um die Norm für die ersten beiden Zeilen einer Matrix wiederherzustellen.

linken_matrizenteil_angleichen: vertauscht Spalten im linken Teil einer Matrix um einen Matching-Versuch zu vollenden.

A.2 Pseudocode

```
%=====================================================
%=================== Hauptfunktion ====================
%=====================================================

PROCEDURE erstelle_alle_bäume IS
BEGIN
FOR maxdist IN 1..6 DO
      breitensuche;
OD;
END search_all_trees;

%=====================================================
%=============== Breitensuche und Unterfunktionen ===============
%=====================================================

PROCEDURE breitensuche RETURN boolean IS
BEGIN
erstelle neue ausgabedatei samt datei-header;
initialisiere Zeile 1 und 2 einer Matrix in letzte_tiefe gemäß momentanem maxdist;
erstelle eine Wahrheitsverwaltung bezüglich Zeile 1;
% benutze hierfür die Funktion „initialisiere_Wahrheitsverwaltung“
aktualisiere die Wahrheitsverwaltung mit Rücksicht auf Zeile 2
% benutze hierfür die Funktion „aktualisiere_wahrheitsverwaltung“
WHILE letzte_tiefe not_empty DO
          setze tiefen-header in der ausgabedatei;
          berechne die nächste tiefe und definiere: letzte_tiefe := nächste_tiefe;
          % benutze hierfür die Funktion „nächste_tiefe“
OD;
setze die datei_ende Marke in der Ausgabedatei;
END breitensuche;

FUNCTION initialisiere_wahrheitsverwaltung RETURN truth_administration IS
ausgabe: wahrheitsverwaltung;
BEGIN
FOR every possible vector DO
      prüfe Abstand zu (0,0,0,0,0,0,0,0,0,0,0,0,2);
      % if d > 3 überspringe Vektor
      nimm den Vektor im Feld der möglichen Wahrheiten auf;
      inkrementiere die Anzahl möglicher Wahrheiten;
      inkrementiere die Komponenten des 1en-Zähl-Vektors die im betrachteten Vektor
      den Wert 1 haben;
OD;
RETURN ausgabe;
END initialisiere_wahrheitsverwaltung;
```

```
FUNCTION aktualisiere_wahrheitsverwaltung (alte_verwaltung: wahrheitsverwaltung;
            antwort: Zeile) RETURN wahrheitsverwaltung IS
BEGIN
FOR all mögliche_wahrheiten IN alte_verwaltung DO
      prüfe Abstand zu Antwort;
      % If d ≤ 3 überspringe mögliche_wahrheit
      Dekrementiere die Komponenten des 1en-Zähl-Vektors die in der möglichen
      Wahrheit den Wert 1 haben;
      lösche die mögliche Wahrheit aus mögliche_wahrheiten;
      dekrementiere die Anzahl möglicher Wahrheiten;
OD;
RETURN alte_verwaltung;
END aktualisiere_wahrheitsverwaltung;

FUNCTION nächste_tiefe(letzte_tiefe: matrizen_liste) RETURN matrizen_liste IS
nächste_tiefe : matrizen_liste;
BEGIN
FORall matrizen IN letzte_tiefe DO
     FOR all mögliche nächste Zeilen DO
          füge Zeile an Matrix an;
          prüfe maxdist;
          % If maxdist > 3 überspringe diese Zeile
          Prüfe Norm 2;
          % If norm 2 verletzt überspringe diese Zeile
          Aktualisiere die wahrheitsverwaltung
          % Lösche alle Vektoren mit einem Abstand > 3 zur neuen Zeile.
          % Benutze hierfür die Funktion „aktualisiere_wahrheitsverwaltung“.
          Prüfe die Fälle a, b, c
          % Falls ein solcher Fall auftritt, schreibe es in die Ausgabe und gehe
          % zur nächsten Zeile. % Benutze hierfür die Funktion „fall_abc“.
          Prüfe auf Symmetrie zu einem früheren Fall
          % Falls eine solche Symmetrie auftritt, schreibe es in die Ausgabe und
          % gehe zur nächsten Zeile.
          % Benutze hierfür die Funktion „matrix_matcher“ für alle Matrizen in
          % der nächste_tiefe liste.
          Hänge neue Matrix an die nächste_tiefe Liste an;
          % Diese Matrix muss dann weiter betrachtet werden.
     OD
OD
RETURN nächste_tiefe;
END nächste_tiefe;
```

```
FUNCTION fall_abc(wahrheitsverwaltung: wahrheitsverwaltung) RETURN fall IS
BEGIN
IF anzahl_wahrheiten = 0 THEN RETURN a;
FI;
IF eine der Komponenten des 1en-Zähl-Vektors hat den Wert 0 oder Anzahl_Wahrheiten
   THEN RETURN b;
FI;
IF eine Lösung gefunden werden kann THEN RETURN c;
% benutze hierfür die Funktion „lösungsfindung“
FI;
RETURN d; % das bedeutet, keiner der Fälle a, b, c tritt ein
END fall_abc;

FUNCTION lösungsfindung(wahrheitsverwaltung: wahrheitsverwaltung) RETURN
              Boolean IS
BEGIN
v := die erste mögliche_wahrheit in wahrheitsverwaltung;
FOR v' mit d(v',v) ≤ 3 DO
      IF der betrachtete Vektor ist Lösung THEN RETURN True;
      OD;
RETURN False;
END lösungsfindung;
```

```
%=================================================
%====================== Der Matcher ==============
%=================================================

FUNCTION matrix_matcher(X,Y : Matrix) RETURN Boolean IS
Permutationen : Permutationsliste := NEW Permutationslistenelement;
Korrekte_Zeilen_Permutation : Permutation;
Symmetrie : Boolean := false;
BEGIN
Erstelle die Zeilenabstandsmatrizen für den linken und den rechten Teil von X und Y;
Prüfe ob die Abstandsmatrizen für den linken Teil die selben Einträge haben
% ELSE RETURN False;
Prüfe ob die Abstandsmatrizen für den rechten Teil die selben Einträge haben
% ELSE RETURN False;
Erstelle eine Liste von Zeilen-Permutationen die für die Abstandmatrizen ein Matching
erzielen;
% benutze hierfür die Funktion „passende_permutationen“
FOR alle diese Zeilen-Permutationen DO
      wende die Permutation auf Matrix Y an;
      % benutze hierfür die Funktion „permutiere_&_restrukturiere_matrix“
      % die Funktion soritert auch die Spalten im rechten Teil der Matrix gemäß Norm 1
      stelle mittels Symmetrieoperationen die normierte FORm für die ersten beiden
      Zeilen wieder her;
      % benutze hierfür die Funktion „matrizenkopf_wiederherstellen“
      prüfe ob nun die rechten Teile der Matrizen übereinstimmen
      % Falls nicht, überspringe diese Permutation
      versuche ein Matching für die linken Teile der Matrizen zu erzielen
      % benutze hierfür die Funktion „linken_matrizenteil_angleichen“
      % falls ein Matchin möglich ist schreibe die benutzte Permutation in die
      % ausgabe_datei
OD
END matrix_matcher;

FUNCTION passende_permutationen(A1,B1,A2,B2: Abstands_matrix) RETURN
            permutations_liste IS
BEGIN
FOR every permutation DO
      Wende die Permutation auf B1 und B2 an.
      % benutze hierfür die Funktion „permutiere_abstandsmatrix“
            IF A1 = B1 AND A2 = B2 THEN
            Hänge Permutation an Liste an;
            FI;
      OD;
      RETURN permutationen;
END passende_permutationen;
```

```
FUNCTION permutiere_abstandsmatrix(A: Halbmatrix; P: Permutation) RETURN
        Halbmatrix IS
BEGIN
FOR all entries in A DO
    A'(i,j) := A(P(i),P(j))
    % Ein Eintrag (i,j) enthält den Abstand zwischen Zeile i und j.
    % Nach der Permutation (der Matrizen, nicht der Abstandsmatrizen) soll dort der
    % Abstand der Zeilen p(i) und p(j) stehen.
OD
RETURN A';
END permutiere_abstandsmatrix;

FUNCTION permutiere_&_restrukturiere_matrix(A: Matrix; p: Permutation) RETURN
        Matrix IS
BEGIN
Wende p auf die Zeilen von A an
% A'(i) := A(P(i)); für Zeilen
Sortiere die Spalten im rechten Teil gemäß Norm 1
% A'(14-i) := A(14-Q(i)); für Spalten
RETURN A';
END permutiere_&_restrukturiere_matrix;

FUNCTION matrizenkopf_wiederherstellen(A: Matrix) RETURN Matrix IS
BEGIN
forme A um bis Zeile 1 „000000000000∗“ ist
% Bitflipping für Spalten i with A(1)(i) = 1
forme A um bis Zeile 2 mit „..∗0“ endet
% IF A(2)(13) = 1 THEN Bitflipping für Spalte 13
forme A um bis Zeile 2 die Form „11..00∗0“ hat
% sortiere einfach die Spalten 1 bis 11
RETURN A;
END matrizenkopf_wiederherstellen;

FUNCTION linken_matrizenteil_angleichen(A,B: Matrix) RETURN Boolean IS
BEGIN
% zu Beginn ist der unangeglichene Teil der Matrizen der gesamte linke Teil der Matrizen
WHILE der unangeglichene Teil nicht leer ist DO
        betrachte die linkeste Spalte des unangeglichenen Teils in A;
        IF eine solche Zeile im unangeglichenen Teil von B existiert
            THEN tausche diese Spalte in B nach links (womit der angeglichene Teil um
                    eine Spalte wächst);
            ELSE RETURN False;
        FI;
OD;
RETURN True;
END linken_matrizenteil_angleichen;
```

B Die Ausgabe des Algorithmus

B.1 FILE = (Maxdist = 1)

Depth 2

000000000000∗
10000000000∗0 Matrix 1-2-1

0000000000∗00 Proceed with 1-3-1
1000000000∗00 sym. to 1-3-1 via perm. 213
0000000000∗10 Proceed with 1-3-2
1000000000∗01 sym. to 1-3-2 via perm. 213

Depth 3

000000000000∗
10000000000∗0
0000000000∗00 Matrix 1-3-1

000000000∗000 Proceed with 1-4-1
100000000∗000 Proceed with 1-4-2
000000000∗010 Proceed with 1-4-3

000000000000∗
10000000000∗0
0000000000∗10 Matrix 1-3-2

000000000∗000 sym. to 1-4-3 via perm. 1243
000000000∗010 sym. to 1-4-3 via perm. 3241

Depth 4

000000000000∗
10000000000∗0
0000000000∗00
000000000∗000 Matrix 1-4-1

00000000∗0000 solution 0000000000000
10000000∗0000 Proceed with 1-5-1
00000000∗0010 solution 0000000000000

000000000000∗
10000000000∗0
0000000000∗00
100000000∗000 Matrix 1-4-2

00000000∗0000 sym. to 1-5-1 via perm. 12354
10000000∗0000 sym. to 1-5-1 via perm. 21453

000000000000∗
10000000000∗0
0000000000∗00
000000000∗010 Matrix 1-4-3

00000000∗0000 solution 0000000000000
00000000∗0010 Proceed with 1-5-2

Depth 5

000000000000∗
10000000000∗0
0000000000∗00
000000000∗000
10000000∗0000 Matrix 1-5-1

0000000∗00000 solution 0000000000000
1000000∗00000 Proceed with 1-6-1

000000000000∗
10000000000∗0
0000000000∗00
000000000∗010
00000000∗0010 Matrix 1-5-2

0000000∗00000 solution 0000000000000
0000000∗00010 solution 0000000000010

Depth 6

000000000000∗
10000000000∗0
0000000000∗00
000000000∗000
10000000∗0000
1000000∗00000 Matrix 1-6-1

000000∗000000 solution 0000000000000
100000∗000000 solution 1000000000000

B.2 FILE = (Maxdist = 2)

Depth 2

000000000000*
11000000000*0 Matrix 2-2-1

0000000000*00 Proceed with 2-3-1
1000000000*00 Proceed with 2-3-2
1100000000*00 sym. to 2-3-1 via perm. 213
1010000000*00 Proceed with 2-3-3
0000000000*10 Proceed with 2-3-4
1000000000*10 Proceed with 2-3-5
1000000000*01 sym. to 2-3-5 via perm. 213
1100000000*01 sym. to 2-3-4 via perm. 213
1000000000*11 Proceed with 2-3-6

Depth 3

000000000000*
11000000000*0
0000000000*00 Matrix 2-3-1

000000000*000 solution 0000000000000
100000000*000 Proceed with 2-4-1
110000000*000 Proceed with 2-4-2
101000000*000 solution 0000000000000
100000000*100 Proceed with 2-4-3
000000000*010 Proceed with 2-4-4
100000000*010 Proceed with 2-4-5
100000000*001 sym. to 2-4-3 via perm. 3214

000000000000*
11000000000*0
1000000000*00 Matrix 2-3-2

000000000*000 sym. to 2-4-1 via perm. 1243
100000000*000 Proceed with 2-4-6
010000000*000 Proceed with 2-4-7
110000000*000 sym. to 2-4-1 via perm. 2143
101000000*000 Proceed with 2-4-8
100000000*100 Proceed with 2-4-9
010000000*100 Proceed with 2-4-10
000000000*010 Proceed with 2-4-11
100000000*010 Proceed with 2-4-12
100000000*001 sym. to 2-4-12 via perm. 2134
110000000*001 sym. to 2-4-11 via perm. 2134
100000000*011 solution 1000000000000

000000000000*
11000000000*0
1010000000*00 Matrix 2-3-3

000000000*000 solution 0000000000000
100000000*000 sym. to 2-4-8 via perm. 1243
110000000*000 solution 1100000000000
101000000*000 solution 1010000000000
011000000*000 Proceed with 2-4-13
100100000*000 solution 0000000000000
100000000*100 solution 1000000000100
100000000*010 solution 1000000000010
100000000*001 solution 1000000000001

000000000000*
11000000000*0
0000000000*10 Matrix 2-3-4

000000000*000 sym. to 2-4-4 via perm. 1243
100000000*000 sym. to 2-4-11 via perm. 1243
100000000*100 Proceed with 2-4-14
000000000*010 sym. to 2-4-4 via perm. 3241
100000000*010 sym. to 2-4-11 via perm. 3241
100000000*011 sym. to 2-4-14 via perm. 3214

000000000000*
11000000000*0
1000000000*10 Matrix 2-3-5

000000000*000 sym. to 2-4-5 via perm. 1243
100000000*000 sym. to 2-4-12 via perm. 1243
110000000*000 sym. to 2-4-3 via perm. 4123
101000000*000 solution 1000000000010
100000000*100 Proceed with 2-4-15
000000000*010 sym. to 2-4-11 via perm. 4231
100000000*010 solution 1000000000010
010000000*010 sym. to 2-4-10 via perm. 3421
100000000*001 Proceed with 2-4-16
100000000*011 Proceed with 2-4-17

000000000000*
11000000000*0
1000000000*11 Matrix 2-3-6

100000000*000 solution 1000000000000
100000000*100 Proceed with 2-4-18
000000000*010 sym. to 2-4-14 via perm. 4213
100000000*010 sym. to 2-4-17 via perm. 1243
100000000*001 sym. to 2-4-17 via perm. 2143
110000000*001 sym. to 2-4-14 via perm. 4123
100000000*011 Proceed with 2-4-19
010000000*011 Proceed with 2-4-20

Depth 4

000000000000*
11000000000*0
0000000000*00

100000000*000 Matrix 2-4-1

00000000*0000 solution 0000000000000
10000000*0000 Proceed with 2-5-1
01000000*0000 Proceed with 2-5-2
11000000*0000 Proceed with 2-5-3
10100000*0000 solution 0000000000000
10000000*1000 solution 0000000000000
01000000*1000 solution 0000000000000
10000000*0100 Proceed with 2-5-4
00000000*0010 Proceed with 2-5-5
10000000*0010 Proceed with 2-5-6
10000000*0001 sym. to 2-5-4 via perm. 32145

000000000000*
11000000000*0
0000000000*00
110000000*000 Matrix 2-4-2

00000000*0000 solution 0000000000000
10000000*0000 sym. to 2-5-3 via perm. 12354
11000000*0000 solution 1100000000000
10100000*0000 solution 0000000000000
10000000*1000 solution 0000000000000
10000000*0100 solution 1100000000000
10000000*0010 solution 0000000000000
10000000*0001 solution 1100000000000

000000000000*
11000000000*0
0000000000*00
100000000*100 Matrix 2-4-3

00000000*0000 solution 0000000000000
10000000*0000 sym. to 2-5-4 via perm. 12354
11000000*0000 solution 1100000000000
10100000*0000 solution 0000000000000
10000000*1000 solution 0000000000000
10000000*0100 solution 1000000000100
01000000*0100 Proceed with 2-5-7
10000000*0010 solution 1000000000010
10000000*0001 solution 1000000000100

000000000000*
11000000000*0
0000000000*00
000000000*010 Matrix 2-4-4

00000000*0000 solution 0000000000000
10000000*0000 sym. to 2-5-5 via perm. 12354
10000000*1000 solution 0000000000000
00000000*0010 Proceed with 2-5-8
10000000*0010 Proceed with 2-5-9

000000000000*
11000000000*0
0000000000*00
100000000*010 Matrix 2-4-5

00000000*0000 solution 0000000000000
10000000*0000 sym. to 2-5-6 via perm. 12354
11000000*0000 solution 0000000000000
10100000*0000 solution 0000000000000
10000000*1000 solution 0000000000000
10000000*0100 solution 1000000000010
00000000*0010 sym. to 2-5-9 via perm. 12354
10000000*0010 solution 1000000000010
01000000*0010 solution 0000000000000
10000000*0001 solution 1000000000010

000000000000*
11000000000*0
1000000000*00
100000000*000 Matrix 2-4-6

00000000*0000 sym. to 2-5-1 via perm. 12534
10000000*0000 solution 1000000000000
01000000*0000 sym. to 2-5-2 via perm. 35412
11000000*0000 sym. to 2-5-1 via perm. 21534
10100000*0000 solution 1000000000000
10000000*1000 Proceed with 2-5-10
10000000*0100 sym. to 2-5-10 via perm. 12435
00000000*0010 solution 1000000000000
10000000*0010 solution 1000000000000
10000000*0001 solution 1000000000000
11000000*0001 solution 1000000000000
10000000*0011 solution 1000000000000

000000000000*
11000000000*0
1000000000*00
010000000*000 Matrix 2-4-7

00000000*0000 sym. to 2-5-2 via perm. 12534
10000000*0000 sym. to 2-5-2 via perm. 34512
01000000*0000 sym. to 2-5-2 via perm. 43512
11000000*0000 sym. to 2-5-2 via perm. 21534
10000000*1000 Proceed with 2-5-11
01000000*0100 sym. to 2-5-11 via perm. 12435
00000000*0010 sym. to 2-5-11 via perm. 34125
11000000*0001 sym. to 2-5-11 via perm. 34215

000000000000*
11000000000*0
1000000000*00
101000000*000 Matrix 2-4-8

00000000*0000 solution 0000000000000
10000000*0000 solution 1000000000000
11000000*0000 solution 1100000000000
10100000*0000 solution 1010000000000
10010000*0000 solution 0000000000000

10000000*1000 solution 1000000000000
10000000*0100 solution 0000000000000
10000000*0010 solution 1000000000000
10000000*0001 solution 1000000000000

000000000000*
11000000000*0
1000000000*00
100000000*100 Matrix 2-4-9

00000000*0000 solution 0000000000000
10000000*0000 sym. to 2-5-10 via perm. 12534
11000000*0000 solution 1100000000000
10100000*0000 solution 0000000000000
10000000*1000 solution 1000000001000
10000000*0100 solution 1000000000100
01000000*0100 sym. to 2-5-7 via perm. 45312
10000000*0010 solution 1000000000010
10000000*0001 solution 1000000000001

000000000000*
11000000000*0
1000000000*00
010000000*100 Matrix 2-4-10

00000000*0000 solution 0000000000000
01000000*0000 sym. to 2-5-11 via perm. 12534
11000000*0000 solution 1100000000000
10000000*0100 sym. to 2-5-7 via perm. 54312
01000000*0100 solution 0100000000100

000000000000*
11000000000*0
1000000000*00
000000000*010 Matrix 2-4-11

00000000*0000 sym. to 2-5-5 via perm. 12534
10000000*0000 solution 1000000000000
01000000*0000 sym. to 2-5-11 via perm. 35124
10000000*1000 Proceed with 2-5-12
00000000*0010 sym. to 2-5-9 via perm. 42513
10000000*0010 Proceed with 2-5-13
10000000*0011 solution 1000000000000

000000000000*
11000000000*0
1000000000*00
100000000*010 Matrix 2-4-12

00000000*0000 sym. to 2-5-6 via perm. 12534
10000000*0000 solution 1000000000000
11000000*0000 sym. to 2-5-4 via perm. 51234
10100000*0000 solution 1000000000000
10000000*1000 solution 1000000000000
10000000*0100 solution 1000000000010
00000000*0010 sym. to 2-5-13 via perm. 12354
10000000*0010 solution 1000000000010
10000000*0001 solution 1000000000000
10000000*0011 solution 1000000000000

000000000000*
11000000000*0
1010000000*00
011000000*000 Matrix 2-4-13

00000000*0000 solution 0000000000000
11000000*0000 solution 1100000000000
10100000*0000 solution 1010000000000
01100000*0000 solution 0110000000000

000000000000*
11000000000*0
0000000000*10
100000000*100 Matrix 2-4-14

00000000*0000 solution 0000000000000
10000000*0000 sym. to 2-5-12 via perm. 12534
10000000*0100 solution 1000000000100
01000000*0100 solution 0000000000000
10000000*0010 solution 1000000000010

000000000000*
11000000000*0
1000000000*10
100000000*100 Matrix 2-4-15

00000000*0000 solution 0000000000000
10000000*0000 solution 1000000000000
11000000*0000 solution 1100000000000
10100000*0000 solution 0000000000000
10000000*1000 solution 1000000000000
10000000*0100 solution 1000000000100
10000000*0010 solution 1000000000010
10000000*0001 solution 1000000000000

000000000000*
11000000000*0
1000000000*10
100000000*001 Matrix 2-4-16

00000000*0000 solution 1000000000010
10000000*0000 solution 1000000000000
11000000*0000 solution 1000000000001
10100000*0000 solution 1000000000000
10000000*1000 solution 1000000000000
10000000*0100 solution 1000000000000
10000000*0010 solution 1000000000010
10000000*0001 solution 1000000000001
10000000*0011 Proceed with 2-5-14

000000000000*
11000000000*0

1000000000*10
100000000*011 Matrix 2-4-17

10000000*0000 solution 1000000000000
10000000*1000 solution 1000000000010
00000000*0010 sym. to 2-5-12 via perm. 52314
10000000*0010 solution 1000000000010
10000000*0001 sym. to 2-5-14 via perm. 12354
10000000*0011 Proceed with 2-5-15

000000000000*
11000000000*0
1000000000*11
100000000*100 Matrix 2-4-18

10000000*0000 solution 1000000000000
10000000*0100 solution 1000000000100
10000000*0010 solution 1000000000010
10000000*0001 solution 1000000000001

000000000000*
11000000000*0
1000000000*11
100000000*011 Matrix 2-4-19

10000000*0000 solution 1000000000000
00000000*0010 solution 1000000000011
10000000*0010 sym. to 2-5-15 via perm. 12534
10000000*0001 sym. to 2-5-15 via perm. 21534
11000000*0001 solution 1000000000011
10000000*0011 solution 1000000000011
01000000*0011 solution 1000000000011

000000000000*
11000000000*0
1000000000*11
010000000*011 Matrix 2-4-20

00000000*0010 solution 0000000000010
11000000*0001 solution 1100000000001
10000000*0011 solution 1000000000011
01000000*0011 solution 0100000000011

Depth 5

000000000000*
11000000000*0
0000000000*00
100000000*000
10000000*0000 Matrix 2-5-1

0000000*00000 solution 0000000000000
1000000*00000 solution 1000000000000
0100000*00000 Proceed with 2-6-1
1100000*00000 Proceed with 2-6-2
1010000*00000 solution 0000000000000
1000000*10000 solution 0000000000000
1000000*01000 solution 0000000000000
1000000*00100 solution 1000000000000
0000000*00010 solution 1000000000000
1000000*00010 solution 1000000000000
1000000*00001 solution 1000000000000

000000000000*
11000000000*0
0000000000*00
100000000*000
01000000*0000 Matrix 2-5-2

0000000*00000 solution 0000000000000
1000000*00000 sym. to 2-6-1 via perm. 123465
0100000*00000 sym. to 2-6-1 via perm. 123564
1100000*00000 Proceed with 2-6-3
1000000*10000 solution 0000000000000
0100000*01000 solution 0000000000000
0000000*00010 solution 0000000000000

000000000000*
11000000000*0
0000000000*00
100000000*000
11000000*0000 Matrix 2-5-3

0000000*00000 solution 0000000000000
1000000*00000 sym. to 2-6-2 via perm. 123465
0100000*00000 sym. to 2-6-3 via perm. 123465
1100000*00000 solution 1100000000000
1010000*00000 solution 0000000000000
1000000*10000 solution 0000000000000
1000000*01000 solution 0000000000000
0100000*01000 solution 0000000000000
1000000*00100 solution 1100000000000
1000000*00010 solution 0000000000000
1000000*00001 solution 1100000000000

000000000000*
11000000000*0
0000000000*00
100000000*000
10000000*0100 Matrix 2-5-4

0000000*00000 solution 0000000000000
1000000*00000 solution 1000000000000
1100000*00000 solution 1100000000000
1010000*00000 solution 0000000000000
1000000*10000 solution 0000000000000
1000000*01000 solution 0000000000000
1000000*00100 solution 1000000000100
1000000*00010 solution 1000000000000
1000000*00001 solution 1000000000000

000000000000*

11000000000*0
0000000000*00
100000000*000
00000000*0010 Matrix 2-5-5

0000000*00000 solution 0000000000000
1000000*00000 solution 1000000000000
0100000*00000 solution 0000000000000
1000000*10000 solution 0000000000000
0000000*00010 Proceed with 2-6-4
1000000*00010 Proceed with 2-6-5

000000000000*
11000000000*0
0000000000*00
100000000*000
10000000*0010 Matrix 2-5-6

0000000*00000 solution 0000000000000
1000000*00000 solution 1000000000000
1100000*00000 solution 0000000000000
1010000*00000 solution 0000000000000
1000000*10000 solution 0000000000000
1000000*01000 solution 0000000000000
1000000*00100 solution 1000000000000
0000000*00010 sym. to 2-6-5 via perm. 123465
1000000*00010 solution 1000000000010
1000000*00001 solution 1000000000000

000000000000*
11000000000*0
0000000000*00
100000000*100
01000000*0100 Matrix 2-5-7

0000000*00000 solution 0000000000000
1100000*00000 solution 1100000000000
1000000*00100 solution 1000000000100
0100000*00100 solution 0100000000100

000000000000*
11000000000*0
0000000000*00
000000000*010
00000000*0010 Matrix 2-5-8

0000000*00000 solution 0000000000000
1000000*00000 sym. to 2-6-4 via perm. 123645
0000000*00010 solution 0000000000010
1000000*00010 sym. to 2-6-4 via perm. 425613

000000000000*
11000000000*0
0000000000*00
000000000*010
10000000*0010 Matrix 2-5-9

0000000*00000 solution 0000000000000
1000000*00000 sym. to 2-6-5 via perm. 123645
1000000*01000 solution 0000000000000
0000000*00010 sym. to 2-6-4 via perm. 426513
1000000*00010 solution 1000000000010
0100000*00010 solution 0000000000000

000000000000*
11000000000*0
1000000000*00
100000000*000
10000000*1000 Matrix 2-5-10

0000000*00000 solution 0000000000000
1000000*00000 solution 1000000000000
1100000*00000 solution 1100000000000
1010000*00000 solution 0000000000000
1000000*10000 solution 1000000000000
1000000*01000 solution 1000000001000
1000000*00100 solution 0000000000000
1000000*00010 solution 1000000000000
1000000*00001 solution 1000000000000

000000000000*
11000000000*0
1000000000*00
010000000*000
10000000*1000 Matrix 2-5-11

0000000*00000 solution 0000000000000
1000000*00000 solution 1000000000000
1100000*00000 solution 1100000000000
1000000*01000 solution 1000000001000

000000000000*
11000000000*0
1000000000*00
000000000*010
10000000*1000 Matrix 2-5-12

0000000*00000 solution 0000000000000
1000000*00000 solution 1000000000000
1000000*01000 solution 1000000001000
1000000*00010 solution 1000000000000

000000000000*
11000000000*0
1000000000*00
000000000*010
10000000*0010 Matrix 2-5-13

0000000*00000 sym. to 2-6-5 via perm. 126345
1000000*00000 solution 1000000000000
1000000*01000 solution 1000000000000
0000000*00010 sym. to 2-6-5 via perm. 426513

1000000*00010 solution 1000000000010
1000000*00011 solution 1000000000000

000000000000*
11000000000*0
1000000000*10
100000000*001
10000000*0011 Matrix 2-5-14

1000000*00000 solution 1000000000000
1000000*10000 solution 1000000000000
1000000*00010 solution 1000000000010
1000000*00001 solution 1000000000001
1000000*00011 Proceed with 2-6-6

000000000000*
11000000000*0
1000000000*10
100000000*011
10000000*0011 Matrix 2-5-15

1000000*00000 solution 1000000000000
0000000*00010 solution 1000000000011
1000000*00010 solution 1000000000010
1000000*00001 sym. to 2-6-6 via perm. 123645
1000000*00011 solution 1000000000011

Depth 6

000000000000*
11000000000*0
0000000000*00
100000000*000
10000000*0000
0100000*00000 Matrix 2-6-1

000000*000000 solution 0000000000000
100000*000000 solution 1000000000000
010000*000000 Proceed with 2-7-1
110000*000000 sym. to 2-7-1 via perm. 4651327
100000*100000 solution 0000000000000
000000*000010 solution 0000000000000

000000000000*
11000000000*0
0000000000*00
100000000*000
10000000*0000
1100000*00000 Matrix 2-6-2

000000*000000 solution 0000000000000
100000*000000 solution 1000000000000
010000*000000 sym. to 2-7-1 via perm. 4751326
110000*000000 solution 1100000000000
101000*000000 solution 0000000000000
100000*100000 solution 0000000000000

100000*010000 solution 0000000000000
100000*001000 solution 0000000000000
100000*000100 solution 1000000000000
100000*000010 solution 0000000000000
100000*000001 solution 1000000000000

000000000000*
11000000000*0
0000000000*00
100000000*000
01000000*0000
1100000*00000 Matrix 2-6-3

000000*000000 solution 0000000000000
100000*000000 sym. to 2-7-1 via perm. 4571326
010000*000000 sym. to 2-7-1 via perm. 5471326
110000*000000 solution 1100000000000
100000*010000 solution 0000000000000
010000*001000 solution 0000000000000

000000000000*
11000000000*0
0000000000*00
100000000*000
00000000*0010
0000000*00010 Matrix 2-6-4

000000*000000 solution 0000000000000
100000*000000 solution 1000000000000
010000*000000 solution 0000000000000
000000*000010 solution 0000000000010
100000*000010 Proceed with 2-7-2

000000000000*
11000000000*0
0000000000*00
100000000*000
00000000*0010
1000000*00010 Matrix 2-6-5

000000*000000 solution 0000000000000
100000*000000 solution 1000000000000
100000*010000 solution 0000000000000
000000*000010 sym. to 2-7-2 via perm. 1234576
100000*000010 solution 1000000000010

000000000000*
11000000000*0
1000000000*10
100000000*001
10000000*0011
1000000*00011 Matrix 2-6-6

100000*000000 solution 1000000000000
100000*000010 solution 1000000000010
100000*000001 solution 1000000000001

100000*000011 solution 1000000000011

Depth 7

000000000000*
11000000000*0
0000000000*00
100000000*000
10000000*0000
0100000*00000
010000*000000 Matrix 2-7-1

00000*0000000 solution 0000000000000
10000*0000000 solution 1000000000000
01000*0000000 solution 0100000000000
11000*0000000 Proceed with 2-8-1
00000*0000010 solution 0000000000000

000000000000*
11000000000*0
0000000000*00
100000000*000
00000000*0010
0000000*00010
100000*000010 Matrix 2-7-2

00000*0000000 solution 0000000000000
10000*0000000 solution 1000000000000
00000*0000010 solution 0000000000010
10000*0000010 solution 1000000000010

Depth 8

000000000000*
11000000000*0
0000000000*00
100000000*000
10000000*0000
0100000*00000
010000*000000
11000*0000000 Matrix 2-8-1

0000*00000000 solution 0000000000000
1000*00000000 solution 1000000000000
0100*00000000 solution 0100000000000
1100*00000000 solution 1100000000000

B.3 FILE = (Maxdist = 3) (nur Tiefe 2 und 7)

Depth 2

000000000000*
11100000000*0 Matrix 3-2-1

0000000000*00 Proceed with 3-3-1
1000000000*00 Proceed with 3-3-2
1100000000*00 sym. to 3-3-2 via perm. 213
1110000000*00 sym. to 3-3-1 via perm. 213
1001000000*00 Proceed with 3-3-3
1101000000*00 sym. to 3-3-3 via perm. 213
0000000000*10 Proceed with 3-3-4
1000000000*10 Proceed with 3-3-5
1100000000*10 Proceed with 3-3-6
1001000000*10 Proceed with 3-3-7
1000000000*01 sym. to 3-3-6 via perm. 213
1100000000*01 sym. to 3-3-5 via perm. 213
1110000000*01 sym. to 3-3-4 via perm. 213
1101000000*01 sym. to 3-3-7 via perm. 213
1000000000*11 Proceed with 3-3-8
1100000000*11 sym. to 3-3-8 via perm. 213

Depth 7

000000000000*
11100000000*0
1000000000*00
010000000*000
11000000*0000
0010000*00000
101000*000000 Matrix 3-7-1

00000*0000000 solution 0000000000000
10000*0000000 solution 1000000000000
01000*0000000 solution 0100000000000
11000*0000000 solution 1100000000000
00100*0000000 solution 0010000000000
10100*0000000 solution 1010000000000
01100*0000000 Bit 4 is always valued 0
11100*0000000 solution 1110000000000
10010*0000000 solution 0000000000000
10000*1000000 solution 0000000000000
01000*1000000 Bit 4 is always valued 0
10000*0100000 solution 0000000000000
11000*0100000 solution 1000000000000
10000*0010000 solution 0000000000000
00100*0010000 Bit 4 is always valued 0
10000*0001000 solution 0000000000000
10100*0001000 solution 1000000000000
10000*0000100 solution 0000000000000
01100*0000100 Bit 4 is always valued 0
00000*0000010 solution 0000000000000
10000*0000010 solution 0000000000000
10000*0000001 solution 1000000000000
11100*0000001 Bit 4 is always valued 0

B.4 FILE = (Maxdist = 4) (gekürzt)

Depth 2

000000000000∗
11110000000∗0 Matrix 4-2-1

0000000000∗00 solution 0000000000000
1000000000∗00 Proceed with 4-3-1
1100000000∗00 solution 1100000000000
1110000000∗00 sym. to 4-3-1 via perm. 213
1111000000∗00 solution 1111000000000
1000100000∗00 solution 0000000000000
1100100000∗00 solution 1100000000000
1110100000∗00 solution 1111000000000
1100110000∗00 solution 0000000000000
0000000000∗10 Proceed with 4-3-2
1000000000∗10 solution 1000000000010
1100000000∗10 Proceed with 4-3-3
1110000000∗10 solution 1110000000010
1000100000∗10 solution 1000000000000
1100100000∗10 solution 1100100000010
1000000000∗01 solution 1000000000001
1100000000∗01 sym. to 4-3-3 via perm. 213
1110000000∗01 solution 1110000000001
1111000000∗01 sym. to 4-3-2 via perm. 213
1100100000∗01 solution 1100100000001
1110100000∗01 solution 1110000000000
1000000000∗11 Proceed with 4-3-4
1100000000∗11 solution 1100000000011
1110000000∗11 sym. to 4-3-4 via perm. 213
1100100000∗11 solution 1100000000010

Depth 3

000000000000∗
11110000000∗0
1000000000∗00 Matrix 4-3-1

000000000∗000 solution 0000000000000
100000000∗000 solution 1000000000000
010000000∗000 Proceed with 4-4-1
110000000∗000 solution 1100000000000
011000000∗000 solution 0110000000000
⋮
010000000∗011 solution 1100000000010
110000000∗011 solution 1000000000000
111000000∗011 solution 1000000000000
110010000∗011 solution 1100000000010
110000000∗111 solution 1100000000010

000000000000∗
11110000000∗0
0000000000∗10 Matrix 4-3-2

000000000∗000 solution 0000000000000
100000000∗000 solution 1000000000000
110000000∗000 solution 1100000000000
111000000∗000 solution 1110000000000
100010000∗000 solution 0000000000000
⋮
100000000∗011 solution 1000000000011
110000000∗011 solution 1100000000010
111000000∗011 solution 1110000000011
110010000∗011 solution 1100000000010
110000000∗111 solution 1100000000010

000000000000∗
11110000000∗0
1100000000∗10 Matrix 4-3-3

000000000∗000 solution 0000000000000
100000000∗000 solution 1000000000000
110000000∗000 solution 1100000000000
001000000∗000 sym. to 4-4-4 via perm. 1243
101000000∗000 solution 1010000000000
⋮
101100000∗000 solution 1000000000000
110010000∗011 solution 1100000000001
101010000∗011 solution 0010000000011
110000000∗111 solution 1100000000001
101000000∗111 solution 1010000000001

000000000000∗
11110000000∗0
1000000000∗11 Matrix 4-3-4

000000000∗000 solution 0000000000000
100000000∗000 solution 1000000000000
010000000∗000 solution 0100000000000
110000000∗000 solution 1000000000000
111000000∗000 solution 1110000000000
⋮
011100000∗011 solution 1111000000010
110010000∗011 solution 0100000000011
011010000∗011 solution 1110000000011
110000000∗111 solution 1100000000001
011000000∗111 solution 1110000000011

Depth 4

000000000000∗
11110000000∗0
1000000000∗00
010000000∗000 Matrix 4-4-1

00000000∗0000 solution 0000000000000
10000000∗0000 solution 1000000000000
01000000∗0000 solution 0100000000000
11000000∗0000 solution 1100000000000
00100000∗0000 Bit 5 is always valued 0
⋮
00100000∗0011 Bit 5 is always valued 0
11100000∗0011 Bit 5 is always valued 0
11001000∗0011 solution 1100000010000
11000000∗1011 Bit 5 is always valued 0
11000000∗0111 Bit 5 is always valued 0

000000000000∗
11110000000∗0
1000000000∗00
111000000∗000 Matrix 4-4-2

00000000∗0000 solution 0000000000000
10000000∗0000 solution 1000000000000
01000000∗0000 solution 1110000000000
11000000∗0000 solution 1100000000000
01100000∗0000 solution 0110000000000
⋮
11100000∗0011 Bit 5 is always valued 0
11010000∗0011 Bit 5 is always valued 0
11001000∗0011 solution 1000000000000
11000000∗1011 Bit 5 is always valued 0
11000000∗0111 Bit 5 is always valued 0

000000000000∗
11110000000∗0
1000000000∗00
011100000∗000 Matrix 4-4-3

00000000∗0000 Bit 5 is always valued 0
10000000∗0000 Bit 5 is always valued 0
01000000∗0000 Bit 5 is always valued 0
11000000∗0000 Bit 5 is always valued 0
01100000∗0000 Bit 5 is always valued 0
⋮
11100000∗1001 Bit 5 is always valued 0
01100000∗0101 Bit 5 is always valued 0
11100000∗0101 Bit 5 is always valued 0
01000000∗0011 Bit 5 is always valued 0
11100000∗0011 Bit 5 is always valued 0

000000000000∗
11110000000∗0
1000000000∗00
011000000∗010 Matrix 4-4-4

00000000∗0000 Bit 5 is always valued 0
10000000∗0000 Bit 5 is always valued 0
01000000∗0000 Bit 5 is always valued 0
11000000∗0000 Bit 5 is always valued 0
01100000∗0000 Bit 5 is always valued 0
⋮
00010000∗0011 Bit 5 is always valued 0
11010000∗0011 Bit 5 is always valued 0
11001000∗0011 Bit 4 is always valued 0
11000000∗1011 Bit 4 is always valued 0
11000000∗0111 Bit 4 is always valued 0

000000000000∗
11110000000∗0
1100000000∗10
101000000∗010 Matrix 4-4-5

00000000∗0000 Bit 5 is always valued 0
10000000∗0000 Bit 5 is always valued 0
01000000∗0000 Bit 5 is always valued 0
11000000∗0000 Bit 5 is always valued 0
00100000∗0000 Bit 5 is always valued 0
⋮
10010000∗1011 Bit 5 is always valued 0
11000000∗0111 Bit 5 is always valued 0
10100000∗0111 Bit 5 is always valued 0
01100000∗0111 Bit 4 is always valued 0
10010000∗0111 Bit 5 is always valued 0

000000000000∗
11110000000∗0
1100000000∗10
101000000∗001 Matrix 4-4-6

00000000∗0000 Bit 5 is always valued 0
10000000∗0000 Bit 5 is always valued 0
01000000∗0000 Bit 5 is always valued 0
11000000∗0000 Bit 5 is always valued 0
00100000∗0000 Bit 5 is always valued 0
⋮
10010000∗1011 Bit 1 is always valued 1
11000000∗0111 Bit 5 is always valued 0
10100000∗0111 Bit 5 is always valued 0
01100000∗0111 Bit 4 is always valued 0
10010000∗0111 Bit 1 is always valued 1

C ADA-Code

C.1 Kopf

```
–=======================================================
–=======================================================
–============= Paket zum Aufbau aller maxdist-Bäume: Kopf ==============
–=======================================================
–=======================================================

With ADA.Text_IO; use ADA.Text_IO;

PACKAGE Breitensuche IS

–=======================================================
–====================== Typ-Vereinbarungen ======================
–=======================================================

TYPE Zahl0bis13 IS RANGE 0..13;
maxdist_abfrage : character := 'z';
maxdist: Zahl0bis13 := 0;
Zeilenanzahl : Zahl0bis13 := 1;
Elementanzahl : Zahl0bis13 := 13;
TYPE Zeile IS ARRAY (1..13) OF Zahl0bis13;
TYPE Matrix IS ARRAY (1..Elementanzahl) OF Zeile; – Instanzierung mit Zahl0bis13
TYPE Halbmatrix IS ARRAY (1..Elementanzahl) OF Zeile; – Instanzierung mit Zahl0bis13
TYPE Eintragszählvektor IS ARRAY (0..13) OF Integer;
– Wir verwenden den Datentyp Integer weil in einer Halbmatrix mehr als 13 Einträge stehen

TYPE Permutationslistenelement;
TYPE Permutation IS Array (1..Elementanzahl) OF Zahl0bis13;
TYPE Permutationsliste IS ACCESS Permutationslistenelement;
TYPE Permutationslistenelement IS RECORD
Eintrag: Permutation;
next: Permutationsliste;
END RECORD;

TYPE Anzahl_1en_pro_Spalte_Vektor IS ARRAY(1..13) OF Integer;
TYPE Wahrheitsmenge IS ARRAY(1..8192) OF Zeile; –598 reicht auch
TYPE Wahrheitsverwaltung IS RECORD
mögliche_Wahrheiten : Wahrheitsmenge;
Anzahl_1en_pro_Spalte : Anzahl_1en_pro_Spalte_Vektor;
Anzahl_Wahrheiten : Integer;
END RECORD;

TYPE Matrizenlistenelement;
TYPE Matrizenliste IS ACCESS Matrizenlistenelement;
TYPE Matrizenlistenelement IS RECORD
MatrixZZZ : Matrix;
WahrheitsverwaltungZZZ : Wahrheitsverwaltung;
next : Matrizenliste;
laufende_Nummer : Integer; – Nur für übersichtliche Ausgabe in Textdatei nötig
END RECORD;

ausgabe_datei : ada.text_io.file_type; – DATEIAUSGABE
globale_ausgabe_permutation : permutation; – DATEIAUSGABE
```

–==========
–========== Funktion Breitensuche und Unterfunktionen ==========
–==========

PROCEDURE Breitensuche_Aufbau;
– Erstellt den Matrizenbaum für ein gegebenes maxdist in Breitensuche

FUNCTION Matrizenkopf_Init(maxdist: Zahl0bis13) RETURN Matrix;
– Deklariert die ersten zwei Zeilen einer Matrix für gegebenes maxdist der Norm 1 entsprechend.

FUNCTION Wahrheitsmengen_Init(Dummy: Zahl0bis13) RETURN Wahrheitsverwaltung;
– Gibt eine Wahrheitsverwaltung, also ein Feld aller Zeilen und die Anzahl derselben, die von der Zeile 0000 0000 0000 * einen Abstand kleiner gleich 3 haben. Dieses Repertoire von Wahrheiten legen wir zu Anfang fest und verwenden es für alle maxdist-Bäume.

FUNCTION Beschneide_Wahrheitsmenge(bisher_Wahrheitsverwaltung: Wahrheitsverwaltung; Antwort: Zeile) RETURN Wahrheitsverwaltung;
– Ändert eine Wahrheitsverwaltung derart, dass die gegebene Menge von Wahrheiten und deren Anzahl um diejenigen Wahrheiten reduziert werden, die von der neuen Antwort einen Abstand > 3 haben.

FUNCTION Nächste_Tiefe(Letzte_Tiefe: Matrizenliste) RETURN Matrizenliste;
– Erstellt die nächste Tiefe eines Maxdist-Baumes. Diese Funktion greift auf die globale Variable Zeilenanzahl zu, damit meinen wir die der Matrizen der NEUEN Tiefe.

FUNCTION mögliche_Folgezeile(Probant : Zeile; Bisher_Matrix : Matrix) RETURN Boolean;
– Prüft für eine Zeile Probant und eine Matrix, ob die Zeile angefügt werden könnte ohne maxdist zu verletzen. Diese Funktion greift ebenfalls auf die globale Variable Zeilenanzahl zu. Beachte, wir meinen die der NEUEN Tiefe.

FUNCTION Einsen_links(Matrix1 : Matrix) RETURN Boolean;
– Prüft, ob die neue Zeile einer Matrix nicht der 1en-links-Regel widerspricht.
Die 1en-links-Regel lässt sich auch so ausdrücken: Wenn zwei Spalten bis einschließlich Zeile k (Zeilenanzahl-1) gleich sind, so darf es nicht vorkommen, dass in der linken Spalte in Zeile k (Zeilenanzahl) eine 0 steht und in der rechten eine 1. Durch die ersten beiden Zeilen ist nach Norm 1 gewährleistet, dass nur innerhalb der Spalten 1..maxdist und maxdist+1 und dem Rest identische Spalten vorkommen können. Durch die *e ist gewährleistet, dass wir die rechten Zeilenanzahl Spalten ganz vernachlässigen können. Da wir für die Spalten von links nach rechts prüfen, ob weiter rechts eine identische Spalte vorkommt, müssen wir nur bis zur ersten gefundenen gehen weil falls weiter rechts weitere identische sind, werden wir diese später mit der gefundenen betrachten. Wieder benutzen wir die globale Variable Zeilenanzahl und beachten wieder...

FUNCTION Spaltengleichheit_bis_Zeile(bis_Zeile, Spalte1, Spalte2 : Zahl0bis13; Matrix1 : Matrix) RETURN Boolean;
– Diese Funktion prüft, ob 2 Spalten einer Matrix bis zu einer bestimmten Zeile identisch sind Sie ist hier eingerückt, weil sie nur für die Einsen_links-Funktion gebraucht wird.

FUNCTION FALL_ABC(Wahrheitsverwaltung1 : Wahrheitsverwaltung) RETURN Boolean;
– Diese Funktion prüft, ob für eine Matrix der Fall a) b) oder c) eintritt, also ob die Menge der Wahrheiten leer wird, ob alle Wahrheiten in einer Bitposition übereinstimmen, oder ob eine Lösung existiert. Für Fall a) haben wir die Anzahl der Wahrheiten, die eine Matrix offen lässt, mitverwaltet. Für Fall b) haben wir die Anzahl der 1en in den Spalten aller Wahrheiten mitgezählt, falls keine 1 oder nur 1en (also genau soviele wie Wahrheiten da sind) vorhanden sind, tritt Fall b) ein. Ob Fall c) eintritt prüfen wir mit der Funktion Lösungsfindung, der wir die Menge der Wahrheiten übergeben.

FUNCTION Lösungsfindung(Wahrheitsverwaltung1 : Wahrheitsverwaltung) RETURN Boolean;
– Diese Funktion prüft, ob zu einer gegebenen Menge von Wahrheiten eine Lösung existiert. Eine Lösung muss zu jeder Wahrheit (also insbesondere zur ersten im Feld) einen Abstand kleiner gleich 3 haben Wir prüfen also für alle 13-Bit-Vektoren, die sich aus der ersten Wahrheit durch höchstens 3 Bitflips gewinnen lassen, ob einer davon als Lösung taugt. Die Fkt. ist eingerückt, weil sie nur für die Funktion Fall_ABC gebraucht wird.

FUNCTION Lösung(Probant : Zeile; Wahrheitsverwaltung1 : Wahrheitsverwaltung) RETURN Boolean;
– Diese Funktion prüft, ob die Zeile „Probant“ Lösung für die in Wahrheitsverwaltung1.mögliche_Wahrheiten gespeicherte Wahrheitsmenge ist, indem sie über alle Wahrheiten läuft und FALSE ausgibt, sobald der Abstand der Zeile „Probant“ zu einer der Wahrheiten > 3 ausgibt. Kommt dies niemals vor, wird TRUE ausgegeben. Die Fkt. ist eingerückt, weil sie nur für die Fkt. Lösungsfindung gebraucht wird.

FUNCTION Symmetrie_bisher(bisher_Liste : Matrizenliste; Probant : Matrix) RETURN Boolean;
– Diese Funktion soll unter Anwendung des Matchers, der samt Unterfunktionen als nächstes aufgeführt wird, feststellen, ob die bisher_Liste bereits ein Matrizenlistenelement mit einer zum Probanten symmetrischen Matrix enthält.

–==========
–========== Funktion Matcher und Unterfunktionen ==========
–==========

FUNCTION Matcher(X,Y : Matrix) RETURN Boolean;
– Versucht stufenweise ein Matching zweier Matrizen herzustellen. Zuerst wird geprüft, ob die Zeilenabstandshalbmatrizen paarweise die selben Einträge besitzen (Eintrags_Vergleich). Dann werden die Permutationen in eine Liste aufgenommen, unter denen die Zeilenabstandshalbmatrizen paarweise identisch werden (Passende_Permutationen). Dann werden für jede dieser Permutationen die Zeilen der Matrix umsortiert und die ersten beiden Zeilen der Normierung enstprechend restrukturiert (Restrukturiere 1 und 2). Danach müssen die rechten Teile der Matrizen (die *-Spalten) bereits übereinstimmen. Nur in diesem Fall wird versucht, den linken Teil durch Spaltenvertauschungen ebenfalls zu einer Übereinstimmung zu führen (Restrukturiere3). Falls die Matrizen so als symmetrisch erkannt werden, setzten wir die Boolsche-Variable 'Symmetrie' auf TRUE, wodurch das Betrachten weiterer Permutationen ausbleibt. Wir geben also die erste Permutation, für die Symmetrie entdeckt wurde, aus.

FUNCTION Abstandsmatrix1_Erstellung(X: Matrix) RETURN Halbmatrix;
– Erstellt für eine Antwortsequenz die Zeilenabstandshalbmatrix für die Bits 1-8

FUNCTION Abstandsmatrix2_Erstellung(X: Matrix) RETURN Halbmatrix;
– Erstellt für eine Antwortsequenz die Zeilenabstandshalbmatrix für die Bits 9-13

FUNCTION Zeilenabstand_eingeschränkt_auf(von, bis: Zahl0bis13; Zeile1, Zeile2: Zeile) RETURN Zahl0bis13;
– Gibt den Abstand zweier Zeilenausschnitte (von...bis) an. Steht in einer Zeile eine 0 und in der anderen eine 1, werten wir dies als Unterschied, sonst nicht.

FUNCTION Eintrags_Vergleich(A,B: Halbmatrix) RETURN Boolean;
– Prüft, ob in zwei Halbmatrizen gleich viele Einträge (0en, 1en, 2en,...) vorkommen. Nur dann kann es auch eine Zeilenpermutation geben, durch die sie identisch werden.
FUNCTION Passende_Permutationen(A1,B1,A2,B2: Halbmatrix) RETURN Permutationsliste;
– Erstellt für die 4 Abstandhalbmatrizen zweier Matrizen die Permutationen der Zeilen der einen

Matrix, die zur anderen passen würden.

FUNCTION Aufnahmeprobe(A1,B1,A2,B2: Halbmatrix; Testpermutation: Permutation; Hilfszeiger: Permutationsliste) RETURN Permutationsliste;
– Hängt die Testpermutation bei Hilfszeiger an eine Liste an,

FUNCTION Permutiere_Halbmatrix(A: Halbmatrix; P: Permutation) RETURN Halbmatrix;
– Ordnet eine Abstands-Halbmatrix entsprechend einer Permutation um.

FUNCTION Halbmatrix_Match(H,K: Halbmatrix) RETURN Boolean;
– Stellt fest, ob zwei Halbmatrizen identisch sind.

FUNCTION Next_Permutation(P: Permutation) RETURN Permutation;
– Diese Funktion soll zu einer gegebenen Permutation die Permutation finden, die nach der Intrepretation als Zahl mit Basis „Zeilenanzahl“ die nächstgrößere Permutation ist. Bei Eingabe (1,2,3,4,5) erwarten wir also die Ausgabe (1,2,3,5,4) und bei Eingabe (1,4,5,7,6,3,2) erwarten wir (1,4,6,2,3,5,7). Das Prinzip: Wir durchwandern die Permutation von rechts aus, bis wir eine Zahl finden, die kleiner ist, als eine der bisher betrachteten. Denn nur dann können wir diese durch eine der bisherigen ersetzen und erhalten überhaupt eine „größere“ Permutation. Nachdem nun gewährleistet ist, dass wir eine größere Permutation erhalten müssen wir dafür sorgen, dass wir die kleinste größere erhalten. Dafür muss der bisher betrachtete Teil von links nach rechts aufsteigend sortiert werden. Bemerkung: Man könnte evtl. die Sortierung des bisher betrachteten Teils on-the-fly, also in der ersten WHILE- Schleife durchführen, das würde eine binäre Suche nach der kleinsten größeren bisher betrachteten Zahl erleichtern, wir wollen aber das Programm einfach halten und es scheint bei so kleinen Feldern nicht notwendig. Diese Funktion fängt die Eingabe (7,6,5,4,3,2,1) NICHT KONSTRUKTIV ab.

FUNCTION Restrukturiere1(A: Matrix; Q: Permutation) RETURN Matrix;
– Ordnet eine Matrix entsprechend einer Permutation um. Das bedeutet, die Zeilenreihenfolge liegt fest und damit auch die Anordnung der rechten Spalten mit *.

FUNCTION Restrukturiere2(A: Matrix) RETURN Matrix;
– Restrukturiert den Matrizenkopf, d.h. es werden die Zeilen 1 und 2 nach den Formvorschriften wiederhergestellt. Danach wollen wir kein Bitflipping mehr anwenden.

FUNCTION Spalten_Bitflip(A: Matrix; i: Zahl0bis13) RETURN Matrix;
– „Bitflipping“ einer Spalte. Ein * (repräsentiert durch eine 2) soll dabei unberührt bleiben.

FUNCTION Matrizen_rechter_Teil_Match(A, B: Matrix) RETURN Boolean;
– Prüft, ob die hinteren 'Zeilenanzahl' Spalten (die nämlich einen * enthalten) von A und B übereinstimmen. Nachdem eine Matrix A Restrukturiere 1 und 2 erfahren hat, haben wir keine Möglichkeit mehr, diese zu vertauschen. Wenn das nicht der Fall ist muss Restrukturiere3 gar nicht angewendet werden.

FUNCTION Restrukturiere3(A,B: Matrix) RETURN Boolean;
– Versucht die Matrix B an die Matrix A anzugleichen. Nachdem eine Matrix A Restrukturiere 1 und 2 erfahren hat, wird nun geprüft, ob durch Umsortieren der Spalten im linken Teil eine exakte Übereinstimmung mit einer Matrix B erreicht werden kann.

FUNCTION Finde(A, B: Matrix; i, Untergrenze, Obergrenze: Zahl0bis13) RETURN Zahl0bis13;
– Versucht, eine zur Spalte i der Matrix A identische Spalte innerhalb der Untergrenze bis Obergrenze Spalten von B zu finden. Bei Erfolg wird die Spaltennummer ausgegeben, bei Misserfolg 0.

FUNCTION Spaltentausch(A: Matrix; i, j : Zahl0bis13) RETURN Matrix;
– Vertauscht zwei Spalten i,j einer Matrix.

–==========
–========== Hilfsfunktionen ==========
–==========

FUNCTION Inkrementiere(Eingabe: Zeile) RETURN Zeile;
– Die Eingabe sollte nur aus 0en und 1en bestehen. Die Binärzahlinterpretation (von links nach rechts) der Ausgabe ist das Inkrement der Binärzahlinterpretation (von links nach rechts) der Eingabe.

FUNCTION Flip(Zeile1 : Zeile; Position : Integer) RETURN Zeile;
– Flippt 0/1 in einer gegebenen Zeile an einer gegebenen Position. Eine 2 bleibt unberührt.

FUNCTION String_aus_Zeile(Zeile1 : Zeile) RETURN String;

PROCEDURE Schreibe_Matrix(Matrix1 : Matrix);

PROCEDURE Schreibe_Zeile(Zeile1 : Zeile);

PROCEDURE Initialisiere_Ausgabedatei(Dummy : Zahl0bis13);

PROCEDURE put_depth_header;

PROCEDURE Permutationsausgabe(P: Permutation);

END Breitensuche;

C.2 Rumpf

```
–=========================================================
–=========================================================
–============ Paket zum Aufbau aller maxdist-Bäume: Rumpf ==========
–=========================================================
–=========================================================

With ADA.Text_IO; use ADA.Text_IO;

PACKAGE BODY Breitensuche IS


package Int_Io is new Integer_Io (Integer); use Int_Io;
– Zur Ausgabe von Integer-Werten (zu Testzwecken)


–=========================================================
–============ Funktion Breitensuche und Unterfunktionen ==========
–=========================================================

PROCEDURE Breitensuche_Aufbau IS
queue1, queue2 : Matrizenliste := NULL; – queue1: die letzte vollständige Tiefe,
                                        – queue2: die zu erstellende Tiefe
Matrizenkopf : Matrizenlistenelement;
BEGIN
Initialisiere_Ausgabedatei(0); – DATEIAUSGABE: Erstellt die Text-Datei im Aufrufverzeichnis
Matrizenkopf.MatrixZZZ := Matrizenkopf_Init(maxdist); – Initialisiert Zeile 1 & 2 (Norm 1)
– Wahrheitsmengen_Init(0) sind die Wahrheiten, die Zeile 1 nicht widersprechen, streiche davon
– die Zeile 2 widersprechenden.
Matrizenkopf.WahrheitsverwaltungZZZ := Beschneide_Wahrheitsmenge(Wahrheitsmengen_Init(0),
                                       Matrizenkopf.MatrixZZZ(2));
Matrizenkopf.laufende_Nummer := 1;
queue1 := NEW Matrizenlistenelement;
– das oben erstellte Matrizenlistenelement bildet den Ausgangspunkt unserer Berechnungen...
queue1.All := Matrizenkopf;
Zeilenanzahl := 2;
WHILE (queue1 /= NULL) LOOP
        ———————- AUSGABETEIL ———————
        Put(„Depth “); Put(Integer(Zeilenanzahl+1), 0); Put(„: “); – BILDSCHIRMAUSGABE
        put_depth_header; – DATEISAUSGABE
        ————————————————————

        ———————- BERECHNUNG ———————
        Zeilenanzahl := Zeilenanzahl + 1;
        queue2 := nächste_Tiefe(queue1);
        queue1 := queue2;
        ————————————————————
END LOOP;
New_Line;
New_Line(ausgabe_datei, 2);
Put_Line(ausgabe_datei, „—— End of File ——“); – DATEIAUSGABE
Close(ausgabe_datei);
END Breitensuche_Aufbau;
–=========================================================
```

```
–=====================================================
– Festlegung der ersten 2 Zeilen gemäß Norm 1.
FUNCTION Matrizenkopf_Init(maxdist: Zahl0bis13) RETURN Matrix IS
Ausgabe: Matrix;
BEGIN
Ausgabe(1) := (0,0,0,0,0,0,0,0,0,0,0,0,2);
Ausgabe(2) := (0,0,0,0,0,0,0,0,0,0,0,2,0);
FOR i IN 1..maxdist LOOP
      Ausgabe(2)(Integer(i)) := 1;
END LOOP;
RETURN Ausgabe;
END Matrizenkopf_Init;
–=====================================================

–=====================================================
FUNCTION Wahrheitsmengen_Init(Dummy: Zahl0bis13) RETURN Wahrheitsverwaltung IS
Binärzähler : Zeile;
Ausgabe : Wahrheitsverwaltung;
BEGIN
Binärzähler := (0,0,0,0,0,0,0,0,0,0,0,0,0);
Ausgabe.Anzahl_Wahrheiten := 0;
Ausgabe.Anzahl_1en_pro_Spalte := (0,0,0,0,0,0,0,0,0,0,0,0,0);
FOR i in 1..8192 LOOP
      IF Zeilenabstand_eingeschränkt_auf(1,12, (0,0,0,0,0,0,0,0,0,0,0,0,2), Binärzähler) ¡= 3
         – Zeilenabst._eing._auf(1,12, (0,0,0,0,0,0,0,0,0,0,0,0,2), x) = Zeilenabst._eing._auf(1,13,
      (0,0,0,0,0,0,0,0,0,0,0,0,2), x)
      THEN Ausgabe.Anzahl_Wahrheiten := Ausgabe.Anzahl_Wahrheiten + 1;
         Ausgabe.mögliche_Wahrheiten(Ausgabe.Anzahl_Wahrheiten) := Binärzähler;
         FOR j IN 1..13 LOOP
               IF Binärzähler(j) = 1 THEN Ausgabe.Anzahl_1en_pro_Spalte(j) :=
                  Ausgabe.Anzahl_1en_pro_Spalte(j)+1;
               END IF;
               END LOOP;
         END IF;
      Binärzähler := Inkrementiere(Binärzähler);
END LOOP;
RETURN Ausgabe;
END Wahrheitsmengen_Init;
–=====================================================

–=====================================================
FUNCTION Beschneide_Wahrheitsmenge(bisher_Wahrheitsverwaltung: Wahrheitsverwaltung;
            Antwort: Zeile) RETURN Wahrheitsverwaltung IS
Ausgabe : Wahrheitsverwaltung := bisher_Wahrheitsverwaltung;
Zähler : Integer := 1;
BEGIN
WHILE Zähler ¡= Ausgabe.Anzahl_Wahrheiten LOOP
         IF Zeilenabstand_eingeschränkt_auf(1, 13, Ausgabe.mögliche_Wahrheiten(Zähler),
                                             Antwort) > 3 THEN
            – notiere 1en pro Spalte nach dem Löschen dieser Wahrheit
            FOR i IN 1..13 LOOP
                  IF Ausgabe.mögliche_Wahrheiten(Zähler)(i) = 1 THEN
                     Ausgabe.Anzahl_1en_pro_Spalte(i) := Ausgabe.Anzahl_1en_pro_Spalte(i)-1;
                  END IF;
            END LOOP;
            – überschreibe die besagte Wahrheit mit der letzten Wahrheit im Feld
```

```
            Ausgabe.mögliche_Wahrheiten(Zähler) :=
            Ausgabe.mögliche_Wahrheiten(Ausgabe.Anzahl_Wahrheiten);
            – überschreibe den Ursprungsort der letzten Wahrheit mit 2en (nicht notwendig)
            Ausgabe.mögliche_Wahrheiten(Ausgabe.Anzahl_Wahrheiten) :=
            (2,2,2,2,2,2,2,2,2,2,2,2,2,2);
            – dekrementiere die Anzahl der Wahrheiten
            Ausgabe.Anzahl_Wahrheiten := Ausgabe.Anzahl_Wahrheiten-1;
         ELSE
            – gehe zur nächsten Wahrheit (aber nur, wenn die letzte erhalten bleibt,
            – sonst betrachten wir gleich die dorthin kopierte)
            Zähler := Zähler+1;
         END IF;
END LOOP;
RETURN Ausgabe;
END Beschneide_Wahrheitsmenge;
–=====================================================

–=====================================================
FUNCTION Nächste_Tiefe(Letzte_Tiefe: Matrizenliste) RETURN Matrizenliste IS
Neue_Tiefe, Laufzeiger_alt, Laufzeiger_neu : Matrizenliste := NULL;
– nächste potentielle Matrix-Erweiterungs-Zeile
Zeilenzähler, Probezeile : Zeile := (0,0,0,0,0,0,0,0,0,0,0,0,0,0);
Erstellte_Matrix : Matrix;
Matrizenlistenprobeelement : Matrizenlistenelement;
BEGIN
Neue_Tiefe := NEW Matrizenlistenelement; – DUMMY oder ANKER
Neue_Tiefe.laufende_Nummer := 0; Laufzeiger_neu := Neue_Tiefe; Laufzeiger_alt := Letzte_Tiefe;
– solange die alte Tiefe nicht vollständig abgearbeitet wurde
WHILE Laufzeiger_alt /= NULL LOOP
         – versuche die dort momentan betrachtete Matrix zu erweitern
         Erstellte_Matrix := Laufzeiger_alt.MatrixZZZ;
         – DATEIAUSGABEBLOCK:
         Schreibe_Matrix(Erstellte_Matrix); – ¡= Matrix
         Put(ausgabe_datei, Integer(maxdist), 0); Put(ausgabe_datei, „-“); – ¡= Nummerierung:
         Put(ausgabe_datei, Integer(Zeilenanzahl-1), 0); Put(ausgabe_datei, „-“);
         Put(ausgabe_datei, Laufzeiger_alt.laufende_Nummer, 0); New_Line(ausgabe_datei, 2);
         – ENDE VON DATEIAUSGABEBLOCK
         FOR i IN 1..8192 LOOP
               Probezeile := Zeilenzähler; – wir nehmen jede mögliche 13-Bitkette
               IF Probezeile(Integer(14-Zeilenanzahl)) = 0 THEN – sonst jede Zeile doppelt
                  – und erstellen daraus eine potentielle Folgezeile mit *
                  Probezeile(Integer(14-Zeilenanzahl)) := 2; – dann prüfen wir, ob diese Zeile
                  – tatsächlich einen Abstand kleiner gleich maxdist zur bisherigen Matrix hat
                  IF mögliche_Folgezeile(Probezeile, Erstellte_Matrix) THEN – falls ja
                     – fügen wir sie an die bisherige Matrix an
                     Erstellte_Matrix(Zeilenanzahl) := Probezeile;
                     – und falls das die 1en links Regel nicht verletzt
                     IF Einsen_links(Erstellte_Matrix) THEN
                        – erstellen wir ein Matrizenlistenelement dafür
                        Matrizenlistenprobeelement.MatrixZZZ := Erstellte_Matrix;
                        Matrizenlistenprobeelement.WahrheitsverwaltungZZZ :=
                        Beschneide_Wahrheitsmenge(Laufzeiger_alt.WahrheitsverwaltungZZZ,
                                                  Probezeile);
                        – jede mögliche Zeile wird in die Ausgabedatei geschrieben
                        IF Matrizenlistenprobeelement.WahrheitsverwaltungZZZ.Anzahl
                           _Wahrheiten /= 0 THEN
```

```
                        Schreibe_Zeile(Erstellte_Matrix(Zeilenanzahl)); – DATEIAUSGABE
                    END IF; – für flag zur dateiausgabe
                    – Wenn keiner der Fälle ABC eintritt
                    IF NOT(FALL_ABC(Matrizenlistenprobeelement.
                                        WahrheitsverwaltungZZZ)) THEN
                        – und wir keine symmetrische Matrix bereits aufgenommen haben
                        – (.next wegen DUMMY)
                        IF NOT(Symmetrie_bisher(Neue_Tiefe.next, Erstellte_Matrix)) THEN
                            Matrizenlistenprobeelement.laufende_Nummer :=
                            Laufzeiger_neu.laufende_Nummer + 1;
                            – DATEIAUSGABEBLOCK
                            Put(ausgabe_datei, „Proceed with „);
                            Put(ausgabe_datei, Integer(maxdist), 0);
                            Put(ausgabe_datei, „-“);
                            Put(ausgabe_datei, Integer(Zeilenanzahl), 0);
                            Put(ausgabe_datei, „-“);
                            Put(ausgabe_datei, Matrizenlistenprobeelement.laufende_Nummer, 0);
                            – ENDE VON DATEIAUSGABEBLOCK
                            – nehmen wir das Matrizenlistenelement in die neue Liste auf
                            Laufzeiger_neu.next := NEW Matrizenlistenelement;
                            – Aufnahme von Erstellte_Matrix und Wahrheitsverwaltung
                            Laufzeiger_neu.next.ALL := Matrizenlistenprobeelement;
                            –aktualisiere Listenende
                            Laufzeiger_neu := Laufzeiger_neu.next;
                            END IF;
                        END IF;
                    New_Line(ausgabe_datei); – nächste Zeile in Ausgabedatei
                    END IF;
                END IF;
            END IF;
            Zeilenzähler := Inkrementiere(Zeilenzähler);
        END LOOP;
        Laufzeiger_alt := Laufzeiger_alt.next;
END LOOP;
Put(Laufzeiger_neu.laufende_Nummer, 0); – BILDSCHIRMAUSGABE
Put(„ matrices“); New_Line; – BILDSCHIRMAUSGABE
RETURN Neue_Tiefe.next; – DUMMY weglassen
END Nächste_Tiefe;
–=====================================================

–=====================================================
FUNCTION mögliche_Folgezeile(Probant : Zeile; Bisher_Matrix : Matrix) RETURN Boolean IS
BEGIN
– BEACHTE: Die neue Zeile wird die zeilenanzahlte Zeile sein
FOR i IN 1..(Zeilenanzahl-1) LOOP
    IF Zeilenabstand_eingeschränkt_auf(1, 13, Probant, Bisher_Matrix(i)) > maxdist
    THEN RETURN FALSE;
    END IF;
END LOOP;
RETURN TRUE;
END mögliche_Folgezeile;
–=====================================================

–=====================================================
FUNCTION Einsen_links(Matrix1 : Matrix) RETURN Boolean IS
Zähler : Zahl0bis13 := 0;
```

```
gefunden : Boolean := FALSE;
BEGIN
– Aufteilung in 2 FOR-Schleifen, denn nur Spalten aus (1..maxdist) bzw. dem Rest können
– wegen Zeile 2 untereinander identische „Vorgeschichten“ haben.
FOR i IN 1..maxdist LOOP
      Zähler := i;
      WHILE (Zähler<maxdist) AND gefunden = FALSE LOOP
               Zähler := Zähler + 1;
               IF Spaltengleichheit_bis_Zeile(Zeilenanzahl-1, i, Zähler, Matrix1)
               THEN gefunden := TRUE;
               END IF;
      END LOOP;
      – existiert in diesem Teil eine gleiche Spalte, steht der Zähler nun auf der linkesten solchen
      IF gefunden
      THEN
         IF (Matrix1(Zeilenanzahl)(Integer(i)) = 0) AND
            (Matrix1(Zeilenanzahl)(Integer(Zähler)) = 1) THEN
            RETURN FALSE;
         END IF;
      END IF;
      gefunden := FALSE;
END LOOP;
– nun der Test für die Spalten maxdist+1 bis 13-Zeilenanzahl
– die #1en-Regel muss nur im Teil ohne * untersucht werden
FOR i IN (maxdist+1)..(13-Zeilenanzahl) LOOP
      Zähler := i;
      WHILE (Zähler<13-Zeilenanzahl) AND gefunden = FALSE LOOP
               Zähler := Zähler + 1;
               IF Spaltengleichheit_bis_Zeile(Zeilenanzahl-1, i, Zähler, Matrix1) THEN
                  gefunden := TRUE;
               END IF;
      END LOOP;
      – existiert in diesem Teil eine gleiche Spalte, steht der Zähler nun auf der linkesten solchen
      IF gefunden THEN
         IF (Matrix1(Zeilenanzahl)(Integer(i)) = 0) AND
            (Matrix1(Zeilenanzahl)(Integer(Zähler)) = 1) THEN
            RETURN FALSE;
         END IF;
      END IF;
      gefunden := FALSE;
END LOOP;
RETURN TRUE;
END Einsen_links;
–=====================================================

–=====================================================
FUNCTION Spaltengleichheit_bis_Zeile(bis_Zeile, Spalte1, Spalte2 : Zahl0bis13;
               Matrix1 : Matrix) RETURN Boolean IS
BEGIN
FOR i IN 1..bis_Zeile LOOP
      IF Matrix1(i)(Integer(Spalte1)) /= Matrix1(i)(Integer(Spalte2)) THEN RETURN FALSE;
      END IF;
END LOOP;
RETURN TRUE;
END Spaltengleichheit_bis_Zeile;
–=====================================================
```

```
–=====================================================
FUNCTION FALL_ABC(Wahrheitsverwaltung1 : Wahrheitsverwaltung) RETURN Boolean IS
BEGIN
– FALL A)
IF Wahrheitsverwaltung1.Anzahl_Wahrheiten = 0 THEN
   Put(ausgabe_datei, „Inconsistent matrix“); – DATEIAUSGABE
   RETURN TRUE;
END IF;
– FALL B)
FOR i IN 1..Zeilenanzahl LOOP
      IF (Wahrheitsverwaltung1.Anzahl_1en_pro_Spalte(Integer(i)) = 0) THEN
         – DATEIAUSGABE
         Put(ausgabe_datei, „Bit “);
         Put(ausgabe_datei, Integer(i), 0);
         Put(ausgabe_datei, „ is always valued 0“);
         RETURN TRUE;
      END IF;
      IF (Wahrheitsverwaltung1.Anzahl_1en_pro_Spalte(Integer(i))) =
         (Wahrheitsverwaltung1.Anzahl_Wahrheiten) THEN
         – DATEIAUSGABE
         Put(ausgabe_datei, „Bit “);
         Put(ausgabe_datei, Integer(i), 0);
         Put(ausgabe_datei, „ is always valued 1“);
         RETURN TRUE;
      END IF;
END LOOP;
– FALL C)
IF Lösungsfindung(Wahrheitsverwaltung1) THEN RETURN TRUE;
END IF;
RETURN FALSE;
END FALL_ABC;
–=====================================================

–=====================================================
FUNCTION Lösungsfindung(Wahrheitsverwaltung1 : Wahrheitsverwaltung) RETURN Boolean
            IS
Probant : Zeile := (0,0,0,0,0,0,0,0,0,0,0,0,0);
BEGIN
– erstmal prüfen, ob die oberste Wahrheit selbst Lösung ist
Probant := Wahrheitsverwaltung1.mögliche_Wahrheiten(1);
IF Lösung(Probant, Wahrheitsverwaltung1) THEN
   Put(ausgabe_datei, „solution “); Schreibe_Zeile(Probant); – DATEIAUSGABE
   RETURN TRUE;
END IF;
FOR i IN 1..13 LOOP – nun prüfen, ob sie durch eine Modifikation Lösung werden kann.
      Probant := Flip(Probant, i); – führe Modifikation 1 durch
      IF Lösung(Probant, Wahrheitsverwaltung1) THEN – Lösung nach erster Modifikation?
         Put(ausgabe_datei, „solution “); Schreibe_Zeile(Probant); – DATEIAUSGABE
         RETURN TRUE
      ; END IF;
      FOR j IN i+1..13 LOOP
            Probant := Flip(Probant, j); – führe Modifikation 2 durch
            – Lösung nach zweiter Modifikation?
            IF Lösung(Probant, Wahrheitsverwaltung1) THEN
               Put(ausgabe_datei, „solution “); Schreibe_Zeile(Probant); – DATEIAUSGABE
               RETURN TRUE;
```

```
            END IF;
            FOR k IN j+1..13 LOOP
                Probant := Flip(Probant, k); – führe Modifikation 3 durch
                – Lösung nach dritter Modifikation?
                IF Lösung(Probant, Wahrheitsverwaltung1) THEN
                   Put(ausgabe_datei, „solution “); Schreibe_Zeile(Probant);– DATEIAUSGABE
                   RETURN TRUE;
                END IF;
                Probant := Flip(Probant, k); – mache Modifikation 3 rückgängig
            END LOOP;
            Probant := Flip(Probant, j); – mache Modifikation 2 rückgängig
        END LOOP;
        Probant := Flip(Probant, i); – mache Modifikation 1 rückgängig
END LOOP;
RETURN FALSE;
END Lösungsfindung;
–=====================================================

–=====================================================
FUNCTION Lösung(Probant : Zeile; Wahrheitsverwaltung1 : Wahrheitsverwaltung) RETURN
            Boolean IS
BEGIN
FOR i IN 1..Wahrheitsverwaltung1.Anzahl_Wahrheiten LOOP
        IF Zeilenabstand_eingeschränkt_auf(1, 13, Wahrheitsverwaltung1.mögliche_Wahrheiten(i),
           Probant) > 3 THEN
           RETURN FALSE;
        END IF;
END LOOP;
RETURN TRUE;
END Lösung;
–=====================================================

–=====================================================
FUNCTION Symmetrie_bisher(bisher_Liste : Matrizenliste; Probant : Matrix) RETURN
              Boolean IS
Verlaufszeiger : Matrizenliste := bisher_Liste;
Matrizenzähler : Integer := 0;
BEGIN
WHILE (Verlaufszeiger /= NULL) LOOP
          Matrizenzähler := Matrizenzähler + 1;
          IF Matcher(Verlaufszeiger.MatrixZZZ, Probant) THEN
             – DATEIAUSGABE
             Put(ausgabe_datei, “symmetric to „);
             Put(ausgabe_datei, Integer(maxdist), 0);
             Put(ausgabe_datei, „-“);
             Put(ausgabe_datei, Integer(Zeilenanzahl), 0);
             Put(ausgabe_datei, „-“);
             Put(ausgabe_datei, Matrizenzähler, 0);
             Permutationsausgabe(globale_ausgabe_permutation);
             RETURN TRUE;
          END IF;
          Verlaufszeiger := Verlaufszeiger.next;
END LOOP;
RETURN FALSE;
END Symmetrie_bisher;
–=====================================================
```

```
–=====================================================
–=============== Funktion Matcher und Unterfunktionen ===============
–=====================================================

–=====================================================
FUNCTION Matcher(X,Y : Matrix) RETURN Boolean IS
– ZAHM bedeutet ZeilenAbstandsHalbMatrix
ZAHM_X1, ZAHM_X2, ZAHM_Y1, ZAHM_Y2 : Halbmatrix;
Testmatrix : Matrix;
Permutationen : Permutationsliste := NEW Permutationslistenelement;
BEGIN
ZAHM_X1 := Abstandsmatrix1_Erstellung(X); ZAHM_X2 := Abstandsmatrix2_Erstellung(X);
ZAHM_Y1 := Abstandsmatrix1_Erstellung(Y); ZAHM_Y2 := Abstandsmatrix2_Erstellung(Y);
– 1. Prüfungsinstanz, ob die Abstandsmatrizen überhaupt unter irgendeiner
– Zeilenpermutation zusammenpassen können
IF Eintrags_Vergleich(ZAHM_X1, ZAHM_Y1) = FALSE THEN RETURN FALSE;
END IF;
IF Eintrags_Vergleich(ZAHM_X2, ZAHM_Y2) = FALSE THEN RETURN FALSE;
END IF;
– Nur falls wir hier ankommen kann es überhaupt sein, dass die Abstandsmatrizen unter
– einer gewissen Permutation identisch werden. PL_Hilfszeiger (global) zeigt auf
– Permutationen (global). Die Permutationsliste wird über den PL_Hilfszeiger erstellt.
Permutationen := Passende_Permutationen(ZAHM_X1, ZAHM_Y1, ZAHM_X2, ZAHM_Y2);
WHILE (Permutationen /= NULL) LOOP
        Testmatrix := Restrukturiere1(Y, Permutationen.Eintrag);
        Testmatrix := Restrukturiere2(Testmatrix);
        IF Matrizen_rechter_Teil_Match(X, Testmatrix) THEN
           IF Restrukturiere3(X, Testmatrix) = TRUE THEN
              globale_ausgabe_permutation := Permutationen.Eintrag;
              RETURN TRUE;
           END IF;
        END IF;
        Permutationen := Permutationen.next;
END LOOP;
RETURN FALSE;
End Matcher;
–=====================================================

–=====================================================
FUNCTION Abstandsmatrix1_Erstellung(X: Matrix) RETURN Halbmatrix IS
Abstände : Halbmatrix;
BEGIN
FOR i IN 1..(Zeilenanzahl-1) LOOP
     FOR j IN (i+1)..Zeilenanzahl LOOP
          Abstände(i)(Integer(j)) := Zeilenabstand_eingeschränkt_auf(1, (13-Zeilenanzahl),
                                     X(i), X(j));
     END LOOP;
END LOOP;
RETURN Abstände;
END Abstandsmatrix1_Erstellung;
–=====================================================

–=====================================================
FUNCTION Abstandsmatrix2_Erstellung(X: Matrix) RETURN Halbmatrix IS
Abstände : Halbmatrix;
BEGIN
```

```
FOR i IN 1..(Zeilenanzahl-1) LOOP
      FOR j IN (i+1)..Zeilenanzahl LOOP
            Abstände(i)(Integer(j)) := Zeilenabstand_eingeschränkt_auf((14-Zeilenanzahl), 13,
                                       X(i), X(j));
      END LOOP;
END LOOP;
RETURN Abstände;
END Abstandsmatrix2_Erstellung;
–=====================================================

–=====================================================
FUNCTION Zeilenabstand_eingeschränkt_auf(von, bis: Zahl0bis13; Zeile1, Zeile2: Zeile)
            RETURN Zahl0bis13 IS
abstand: Zahl0bis13 := 0;
BEGIN
FOR i IN von..bis LOOP
      IF ((Zeile1(Integer(i)) = 0 AND Zeile2(Integer(i)) = 1)
         OR (Zeile1(Integer(i)) = 1 AND Zeile2(Integer(i)) = 0))
      THEN abstand := (abstand+1);
      END IF;
END LOOP;
RETURN abstand;
END Zeilenabstand_eingeschränkt_auf;
–=====================================================

–=====================================================
FUNCTION Eintrags_Vergleich(A,B: Halbmatrix) RETURN Boolean IS
Vektor_A, Vektor_B : Eintragszählvektor := (0,0,0,0,0,0,0,0,0,0,0,0,0,0);
Gleichheit : Boolean;
BEGIN
FOR i IN 1..(Zeilenanzahl-1) LOOP
      FOR j IN (i+1)..Zeilenanzahl LOOP
            Vektor_A(Integer(A(i)(Integer(j)))) := Vektor_A(Integer(A(i)(Integer(j))))+1;
            Vektor_B(Integer(B(i)(Integer(j)))) := Vektor_B(Integer(B(i)(Integer(j))))+1;
      END LOOP;
END LOOP;
Gleichheit := (Vektor_A = Vektor_B);
RETURN Gleichheit;
END Eintrags_Vergleich;
–=====================================================

–=====================================================
FUNCTION Passende_Permutationen(A1,B1,A2,B2: Halbmatrix) RETURN Permutationsliste
            IS
Startpermutation, Endpermutation : Permutation := (0,0,0,0,0,0,0,0,0,0,0,0,0,0);
Permutationen : Permutationsliste := NEW Permutationslistenelement; – Dummy-Element
Hilfszeiger : Permutationsliste := Permutationen;
BEGIN
– Initialisierung der Startpermutation/Endpermutation
FOR i IN 1..Zeilenanzahl LOOP
      Startpermutation(i) := i;
      Endpermutation(i) := Zeilenanzahl+1-i;
END LOOP;
– Schleife über alle Permutationen
WHILE Startpermutation /= Endpermutation LOOP
         Hilfszeiger := Aufnahmeprobe(A1,B1,A2,B2, Startpermutation, Hilfszeiger);
```

```
        Startpermutation := Next_Permutation(Startpermutation);
END LOOP;
– Nun noch für die Endpermutation
Hilfszeiger := Aufnahmeprobe(A1,B1,A2,B2, Startpermutation, Hilfszeiger);
RETURN Permutationen.next; – das erste Element war ja ein Dummy
End Passende_Permutationen;
–=====================================================

–=====================================================
FUNCTION Aufnahmeprobe(A1,B1,A2,B2: Halbmatrix; Testpermutation: Permutation;
            Hilfszeiger: Permutationsliste) RETURN Permutationsliste IS
Testmatrix1,Testmatrix2: Halbmatrix;
Ausgabehilfszeiger: Permutationsliste := Hilfszeiger;
BEGIN
Testmatrix1 := Permutiere_Halbmatrix(B1, Testpermutation);
Testmatrix2 := Permutiere_Halbmatrix(B2, Testpermutation);
IF ((Halbmatrix_Match(A1, Testmatrix1)) AND (Halbmatrix_Match(A2, Testmatrix2))) THEN
   Ausgabehilfszeiger.next := NEW Permutationslistenelement;
   Ausgabehilfszeiger := Ausgabehilfszeiger.next;
   Ausgabehilfszeiger.Eintrag := Testpermutation;
END IF;
RETURN Ausgabehilfszeiger;
END Aufnahmeprobe;
–=====================================================

–=====================================================

FUNCTION Halbmatrix_Match(H,K: Halbmatrix) RETURN Boolean IS
Gleichheit : Boolean := TRUE;
BEGIN
FOR i IN 1..(Zeilenanzahl-1) LOOP
      FOR j IN (i+1)..Zeilenanzahl LOOP
            Gleichheit := Gleichheit AND (K (i) (Integer(j)) = H (i) (Integer(j)) );
      END LOOP;
END LOOP;
RETURN Gleichheit;
END Halbmatrix_Match;
–=====================================================

–=====================================================
FUNCTION Permutiere_Halbmatrix(A: Halbmatrix; P: Permutation) RETURN Halbmatrix IS
Ausgabematrix : Halbmatrix;
BEGIN
FOR i IN 1..(Zeilenanzahl-1) LOOP
      FOR j IN (i+1)..Zeilenanzahl LOOP
            IF P(i)<P(j) THEN
               Ausgabematrix(i)(Integer(j)) := A(P(i))(Integer(P(j)));
            ELSE Ausgabematrix(i)(Integer(j)) := A(P(j))(Integer(P(i)));
            END IF;
      END LOOP;
END LOOP;
RETURN Ausgabematrix;
END Permutiere_Halbmatrix;
–=====================================================
```

```
–=====================================================
FUNCTION Next_Permutation(P: Permutation) RETURN Permutation IS
tauschhilfe, kleinstmöglich : Zahl0bis13 := 1;
momentan_betrachtet : Zahl0bis13 := Zeilenanzahl;
Q : Permutation := P;
BEGIN
WHILE Q(momentan_betrachtet-1) ¿= Q(momentan_betrachtet) LOOP
        momentan_betrachtet := momentan_betrachtet-1;
        IF momentan_betrachtet = 1 THEN
            Put_Line(„Absturz durch Next_Permutation“);
            – Abfangen einer falschen Eingabe wie oben beschrieben, im nächsten
            – Schleifendurchlauf wird auf Q(0) zugegriffen... Eigentlich sollte
            – die aufrufende Funktion Passende_Perm. diesen Aufruf gar nicht durchführen
        END IF;
END LOOP;
– nun ist momentan_betrachtet die Stelle, die durch einen möglichst kleinen größeren Eintrag
– ersetzt werden muss
– wir durchlaufen den bisher betrachteten Teil auf der Suche nach diesem Eintrag
kleinstmöglich := momentan_betrachtet;
FOR i IN (momentan_betrachtet)..Zeilenanzahl LOOP
      IF Q(i)>Q(momentan_betrachtet-1) AND Q(i) ¡= Q(kleinstmöglich) THEN
          kleinstmöglich := i;
      END IF;
END LOOP;
– kleinstmöglich ist nun der kleinste Eintrag im bisher betrachteten Teil, wir vertauschen...
tauschhilfe := Q(momentan_betrachtet-1);
Q(momentan_betrachtet-1) := Q(kleinstmöglich);
Q(kleinstmöglich) := tauschhilfe;
– nun müssen wir noch den davon rechts liegenden Teil sortieren
FOR i IN (momentan_betrachtet)..Zeilenanzahl LOOP
      FOR j IN i+1..Zeilenanzahl LOOP
            IF Q(j)<Q(i) THEN
                tauschhilfe := Q(i);
                Q(i) := Q(j);
                Q(j) := tauschhilfe; – dann steht in Q(i) der kleinste Eintrag
            END IF;
      END LOOP;
END LOOP;
RETURN Q;
END Next_Permutation;
–=====================================================

–=====================================================
FUNCTION Restrukturiere1(A: Matrix; Q: Permutation) RETURN Matrix IS
Zeilenmatrix, Endmatrix: Matrix;
BEGIN
FOR i IN 1..Zeilenanzahl LOOP
      Zeilenmatrix(i) := A(Q(i));
END LOOP;
– Bringt die Zeilen in die richtige Reihenfolge
Endmatrix := Zeilenmatrix;
– Matrix wird kopiert und dann die hinteren Spalten korrigiert
FOR i IN 1..Zeilenanzahl LOOP
      –Spalte(14-i) der Endmatrix := Spalte(14-Permutation(i)) der Zeilenmatrix
      FOR j IN 1..Zeilenanzahl LOOP – Diese FOR-Schleife kopiert die Spalten
            Endmatrix(j)(Integer(14-i)) := Zeilenmatrix(j)(14-Integer(Q(i)));
```

```
      END LOOP;
END LOOP;
– Nun sind die *e wieder in der Diagonalen
RETURN Endmatrix;
END Restrukturiere1;
–=========================================================

–=========================================================
FUNCTION Restrukturiere2(A: Matrix) RETURN Matrix IS
Nullspaltenmarker, Einsspaltenmarker : Zahl0bis13;
B: Matrix := A;
BEGIN
FOR i IN 1..12 LOOP
      IF B(1)(i) = 1 THEN
         B := Spalten_Bitflip(B,Zahl0bis13(i));
      END IF;
END LOOP;
– Zeile 1 ist dann 0000 0000 0000 *
IF B(2)(13) = 1 THEN
   B := Spalten_Bitflip(B,13);
END IF;
– Zeile 2 endet dann mit *0
Nullspaltenmarker:=1;
Einsspaltenmarker:=13-Zeilenanzahl;
WHILE (Nullspaltenmarker<Einsspaltenmarker) LOOP
         WHILE ((B(2)(Integer(Nullspaltenmarker)) /= 0) AND
                  (Nullspaltenmarker<Einsspaltenmarker)) LOOP
                  Nullspaltenmarker := (Nullspaltenmarker+1);
         END LOOP;
         WHILE ((B(2)(Integer(Einsspaltenmarker)) /= 1) AND
                  (Nullspaltenmarker<Einsspaltenmarker)) LOOP
                  Einsspaltenmarker := (Einsspaltenmarker-1);
         END LOOP;
         B := Spaltentausch(B, Nullspaltenmarker, Einsspaltenmarker);
         Nullspaltenmarker := (Nullspaltenmarker+1);
         Einsspaltenmarker := (Einsspaltenmarker-1);
END LOOP;
RETURN B;
END Restrukturiere2;
–=========================================================

–=========================================================
FUNCTION Spalten_Bitflip(A: Matrix; i: Zahl0bis13) RETURN Matrix IS
B: Matrix := A;
BEGIN
FOR j IN 1..Zeilenanzahl LOOP
      IF A(j)(Integer(i)) = 1 THEN
         B(j)(Integer(i)) := 0;
      ELSIF A(j)(Integer(i)) = 0 THEN
            B(j)(Integer(i)) := 1;
      ELSE NULL;
      END IF;
END LOOP;
RETURN B;
END Spalten_Bitflip;
–=========================================================
```

```
–=====================================================
FUNCTION Matrizen_rechter_Teil_Match(A, B: Matrix) RETURN Boolean IS
BEGIN
FOR i IN 14-Zeilenanzahl..13 LOOP
      FOR j IN 1..Zeilenanzahl LOOP
            IF (A(j)(Integer(i)) /= B(j)(Integer(i))) THEN
               RETURN FALSE;
            END IF;
      END LOOP;
END LOOP;
RETURN TRUE;
END Matrizen_rechter_Teil_Match;
–=====================================================

–=====================================================
FUNCTION Restrukturiere3(A,B: Matrix) RETURN Boolean IS
j: Zahl0bis13;
Testmatrix: Matrix := B;
BEGIN
FOR i IN 1..maxdist LOOP
      j := Finde(A, Testmatrix, i, i, maxdist);
      IF j /= 0 THEN
         Testmatrix := Spaltentausch(Testmatrix, i, j);
      ELSE RETURN FALSE; –(Abbruch von Restrukturiere3)
      END IF;
END LOOP;
FOR i IN (maxdist+1)..(13-Zeilenanzahl) LOOP
      j := Finde(A, Testmatrix, i, maxdist+1, (13-Zeilenanzahl));
      IF j /= 0 THEN
         Testmatrix := Spaltentausch(Testmatrix, i, j);
      ELSE RETURN FALSE; –(Abbruch von Restrukturiere3)
      END IF;
END LOOP;
RETURN TRUE;
END Restrukturiere3;
–=====================================================

–=====================================================
FUNCTION Finde(A, B: Matrix; i, Untergrenze, Obergrenze: Zahl0bis13) RETURN
            Zahl0bis13 IS
Spalten_Gleichheit : Boolean;
gefunden : Zahl0bis13 := 0;
BEGIN
FOR s IN Untergrenze..Obergrenze LOOP
      Spalten_Gleichheit := TRUE;
      FOR z IN 1..Zeilenanzahl LOOP
            Spalten_Gleichheit := (Spalten_Gleichheit AND
            A(z)(Integer(i))=B(z)(Integer(s)));
      END LOOP;
      IF Spalten_Gleichheit
         THEN gefunden := s;
      END IF;
END LOOP;
RETURN gefunden;
END Finde;
–=====================================================
```

```
–=======================================================
FUNCTION Spaltentausch(A: Matrix; i, j : Zahl0bis13) RETURN Matrix IS
B : Matrix;
BEGIN
B := A;
FOR k IN 1..Zeilenanzahl LOOP
      B(k)(Integer(i)) := A(k)(Integer(j));
END LOOP;
FOR k IN 1..Zeilenanzahl LOOP
      B(k)(Integer(j)) := A(k)(Integer(i));
END LOOP;
RETURN B;
END Spaltentausch;
–=======================================================

–=======================================================
–====================== Hilfsfunktionen ==================
–=======================================================

–=======================================================
FUNCTION Inkrementiere(Eingabe: Zeile) RETURN Zeile IS
Ausgabe : Zeile := Eingabe;
Position : Zahl0bis13 := 1;
BEGIN
– Abfangen eines Sonderfalls, die Ausgabe in diesem Fall sollte niemals benutzt werden
IF Ausgabe = (1,1,1,1,1,1,1,1,1,1,1,1,1) THEN
   RETURN (0,0,0,0,0,0,0,0,0,0,0,0,0);
END IF;
– Für die gewöhnlichen Fälle:
WHILE Ausgabe(Integer(Position)) = 1 LOOP
         Position := Position+1;
END LOOP;
– Position ist nun die Position der ersten 0 von links an. Wir inkrementieren, indem wir...
– alle bisherigen Positionen von 1 auf 0 setzen...
FOR i IN 1..(Position-1) LOOP
      Ausgabe(Integer(i)) := 0;
END LOOP;
– und die momentane von 0 auf 1 setzen.
Ausgabe(Integer(Position)) := 1;
RETURN Ausgabe;
END Inkrementiere;
–=======================================================

–=======================================================
FUNCTION Flip(Zeile1 : Zeile; Position : Integer) RETURN Zeile IS
Ausgabe : Zeile := Zeile1;
BEGIN
IF Ausgabe(Integer(Position)) = 1 THEN
   Ausgabe(Integer(Position)) := 0;
ELSE IF Ausgabe(Integer(Position)) = 0 THEN
                Ausgabe(Integer(Position)) := 1;
       END IF;
END IF;
RETURN Ausgabe;
END Flip;
–=======================================================
```

```
–=======================================================
PROCEDURE Schreibe_Matrix(Matrix1 : Matrix) IS
BEGIN
New_Line(ausgabe_datei);
– Schreibe_Matrix wird in nächste_Tiefe aufgerufen und da ist Zeilenanzahl schon inkrementiert.
For i IN 1..(Zeilenanzahl-1) LOOP
        Schreibe_Zeile(Matrix1(i));
        IF i (Zeilenanzahl-1) THEN
            New_Line(ausgabe_datei);
        ELSE Put(ausgabe_datei, „Matrix “); – DATEIAUSGABE
        END IF;
END LOOP;
END Schreibe_Matrix;
–=======================================================

–=======================================================
PROCEDURE Schreibe_Zeile(Zeile1 : Zeile) IS
BEGIN
– DATEIAUSGABE
Put(ausgabe_datei, String_aus_Zeile(Zeile1));
Put(ausgabe_datei, “              „); – nach jeder Zeile etwas Platz lassen
END Schreibe_Zeile;
–=======================================================

–=======================================================
FUNCTION String_aus_Zeile(Zeile1 : Zeile) RETURN String IS
Ausgabe : String(1..13) := „0000000000000“;
BEGIN
FOR i IN 1..13 LOOP
       CASE Zeile1(i) IS
            WHEN 0 => Ausgabe(i) := ’0’;
            WHEN 1 => Ausgabe(i) := ’1’;
            WHEN 2 => Ausgabe(i) := ’*’;
            WHEN OTHERS => Put_Line(„Unangemessener Matrizeneintrag“);
       END CASE;
END LOOP;
RETURN Ausgabe;
END String_aus_Zeile;
–=======================================================

–=======================================================
PROCEDURE Initialisiere_Ausgabedatei(Dummy : Zahl0bis13) IS
Datei_Name : String(1..13) := „maxdist=?.txt“;
BEGIN
CASE maxdist IS
       WHEN 1 => Datei_Name(9) := ’1’;
       WHEN 2 => Datei_Name(9) := ’2’;
       WHEN 3 => Datei_Name(9) := ’3’;
       WHEN 4 => Datei_Name(9) := ’4’;
       WHEN 5 => Datei_Name(9) := ’5’;
       WHEN 6 => Datei_Name(9) := ’6’;
       WHEN OTHERS => Put_Line(„unangemessens maxdist in Initialsiere_Ausgabedatei“);
END CASE;
create(file => ausgabe_datei, name => Datei_Name);
– DATEIAUSGABE
Put(ausgabe_datei, „————————————————————“);
```

```
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „— —“);
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „— FILE = (Maxdist = „ & Datei_Name(9) & „)“ & “ —“);
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „— —“);
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „————————————————————“);
New_Line(ausgabe_datei, 1);
END Initialisiere_Ausgabedatei;
–=========================================================

–=========================================================
PROCEDURE Permutationsausgabe(P: Permutation) IS
Permutationsstring : String(1..Integer(Zeilenanzahl));ß BEGIN
FOR i IN 1..(Zeilenanzahl) LOOP
      CASE P(i) IS
          WHEN 0 => Permutationsstring(Integer(i)) := '0';
          WHEN 1 => Permutationsstring(Integer(i)) := '1';
          WHEN 2 => Permutationsstring(Integer(i)) := '2';
          WHEN 3 => Permutationsstring(Integer(i)) := '3';
          WHEN 4 => Permutationsstring(Integer(i)) := '4';
          WHEN 5 => Permutationsstring(Integer(i)) := '5';
          WHEN 6 => Permutationsstring(Integer(i)) := '6';
          WHEN 7 => Permutationsstring(Integer(i)) := '7';
          WHEN 8 => Permutationsstring(Integer(i)) := '8';
          WHEN 9 => Permutationsstring(Integer(i)) := '9';
          WHEN OTHERS => Put_Line(„Keine Matrizen mit mehr als 9 Zeilen vorgesehen in
                                        Permutationsausgabe“);
      END CASE;
END LOOP;
Put(ausgabe_Datei, „ via permutation “);
Put(ausgabe_Datei, Permutationsstring);
END Permutationsausgabe;
–=========================================================

–=========================================================
PROCEDURE put_depth_header IS
BEGIN
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „————————————————————————————————————————“);
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „————————————————————————————————————————“);
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „Depth „);
Put(ausgabe_datei, Integer(Zeilenanzahl), 0);
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „————————————————————————————————————————“);
New_Line(ausgabe_datei, 1);
Put(ausgabe_datei, „————————————————————————————————————————“);
New_Line(ausgabe_datei, 1);
END put_depth_header;
–=========================================================

END Breitensuche;
```

C.3 Hauptprogramm

```
–=============================================================
–=============================================================
–====================== Hauptprogramm ======================
–=============================================================
–=============================================================

WITH Ada.Text_IO; USE Ada.Text_IO;
WITH Breitensuche;
USE Breitensuche;

PROCEDURE main IS
PACKAGE Int_Io is NEW Integer_Io (Integer); USE Int_Io;
BEGIN
– maxdist Auswahl:
Put („Enter maxdist (1-4): „);
WHILEmaxdist_abfrage /= ’1’ AND
        maxdist_abfrage /= ’2’ AND
        maxdist_abfrage /= ’3’ AND
        maxdist_abfrage /= ’4’ LOOP
        Get_Immediate ( Item =¿ maxdist_abfrage);
END LOOP;
Put(maxdist_abfrage);
– Ende der Auswahl
– Typkonvertierung für maxdist
CASE maxdist_abfrage IS
       WHEN ’1’ => maxdist := 1;
       WHEN ’2’ => maxdist := 2;
       WHEN ’3’ => maxdist := 3;
       WHEN ’4’ => maxdist := 4;
       WHEN OTHERS =¿ Put(„Fehlerhafte maxdist-Eingabe wurde akzeptiert“);
END CASE;
New_Line;
New_Line;
– Start der Aufbau-Prozedur für das gewählte maxdist
Breitensuche_Aufbau;
END main;
```

Erklärung

Hiermit versichere ich, diese Arbeit
selbstständig verfasst und nur die
angegebenen Quellen benutzt zu haben.

_____

(C. F. Minnameier)