RWTH AACHEN UNIVERSITY

Lehrstuhl für Operations Research
und Supply Chain Management
der RWTH Aachen
Univ.-Prof. Dr. Marco Lübbecke
- Fakultät für Wirtschaftswissenschaften -

Diplomarbeit

Eine Heuristik zum Erkennen von Staircase-Strukturen in Matrizen

Verfasser: Dipl.-Ing. Mathias Luers

Betreuer: Univ.-Prof. Dr. Marco Lübbecke
Martin Bergner, M.Sc.

Aachen, Februar 2012

Inhaltsverzeichnis

Verwendete Symbole

Lateinische Buchstaben

A	Nebenbedingungsmatrix
M	Anzahl der Zeilen (Nebenbedingungen)
N	Anzahl der Spalten (Variablen)
$\bar{n}$	Durchnittliche Anzahl an Variablen pro Block
$\tilde{n}$	Maximale Breite aller Zeilen; Näherungswert für $\overline{n}$
T	Tatsächliche Blockanzahl
$\tilde{T}$	Angestrebte Blockanzahl
$\bar{v}$	Durchnittliche Anzahl an verbindenden Variablen pro Block
$\tilde{v}$	Minimale Breite aller Zeilen; Näherungswert für $\overline{v}$

Abkürzungen

GCG	Generic Column Generation
LP	Lineares Programm (Optimierungsproblem)
MIP	Mixed Integer Program (engl., Gemischt-Ganzahliges Problem)
RAP	Resource Allocation Problem
ROC	Rank-Order-Clustering
SCIP	Solving Constraint Integer Progams

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Gemischt-ganzzahlige Probleme (engl. Mixed Integer Program, MIP) sind oft schwer zu lösen, weshalb unterschiedliche Lösungsstrategien entwickelt wurden. Moderne MIP-Löser nutzen häufig Schnittebenen in Verbindung mit einem branch-and-cut-Algorithmus [VW10]. Allerdings wird bei diesem Vorgehen eine mögliche Struktur des MIPs ignoriert. In [VW10] wird berichtet, dass seit 1960 für zahlreiche MIPs Dantzig-Wolfe-Dekomposition [DW60] erfolgreich eingesetzt wurde, bei der die Struktur des Problems ausgenutzt wird. Jedoch wird dort auch auf den Schwachpunkt bisheriger Ansätze hingewiesen: Alle Methoden mussten auf die (bekannte) Struktur des jeweiligen Problems zugeschnitten werden, um eine gute Performance zu erreichen. Der Nutzer muss wissen, dass eine Struktur existiert, wie sie aussieht und wie sie sich mittels eines Algorithmus ausnutzen lässt [GL10].

Deshalb gibt es in letzter Zeit einige Bestrebungen, Löser für allgemeine MIPs zu entwickeln, die nur wenig Nutzerinteraktion erfordern. Ein Ansatz dazu ist der Löser `GCG` (Generic Column Generation) [GL10], der seinerseits auf dem `SCIP`-Framework [Ach09] basiert. Bei `GCG` wird eine Dantzig-Wolfe-Dekomposition durchgeführt und die entstehenden Teilprobleme zur Spaltenerzeugung werden auf generische Weise behandelt. In [GL10] wird gezeigt, dass dieser Ansatz erfolgreich ist, falls als Input für den Algorithmus ein Problem mit bekannter Blockdiagonalstruktur verwendet wird. Natürlich ist es wünschenswert, diesen Ansatz auch auf Probleme mit verborgener Struktur zu erweitern. In diesem Zusammenhang ist die Arbeit [BCF$^+$11] zu erwähnen, in der die automatische Erkennung von (beidseitig begrenzten) Blockdiagonalstrukturen mittels Zerlegung von Hypergraphen demonstriert wird. Damit konnte der gesamte Prozess (bis auf die Vorgabe bestimmter Parameter) automatisiert werden.

In dieselbe Richtung zielt auch diese Arbeit: Sie soll die `GCG`-Implementierung dahingehend erweitern, dass auch die automatische Erkennung einer weiteren Struktur – der Staircase – möglich ist. Dazu wird das Paper [JR94] herangezogen und die dort vorgeschlagene Heuristik zur Erkennung von Staircase-Strukturen in `GCG` implementiert. Diese Heuristik besteht aus den folgenden zwei Schritten, die in Abbildung 1.1 visualisiert sind:

1. Zeilen und Spalten der Nebenbedingungsmatrix werden permutiert. Der verantwortliche Algorithmus heißt Rank Order Clustering (ROC) und ordnet die Nichtnulleinträge der Matrix in einem bandförmigen Muster entlang der Diagonalen an (s. Kap. 2.2).

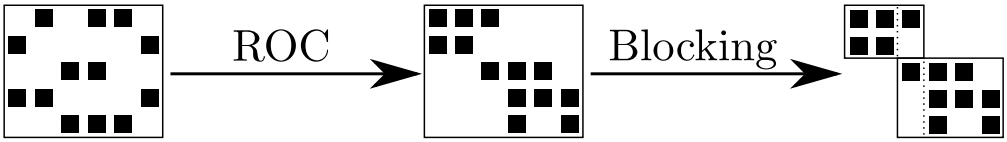


Abbildung 1.1: Zwei Schritte auf dem Weg zur Staircase-Struktur: Rank-Order-Clustering (ROC) und Blocking.

2. Die permutierte Matrix muss in Blöcke unterteilt werden (s. Kap. 2.3).

Die Heuristik ist so konstruiert, dass sie möglichst wenig *verbindende Variablen* erzeugt. Verbindende Variablen (eine Definition erfolgt in Kapitel 2.1) gehören zu zwei Blöcken, so wie es in Abbildung 1.1 auf dem rechten Bild für die dritte Variable zutrifft. Eine Heuristik wird deshalb eingesetzt, da das Finden der Zeilen- und Spaltenpermutationen, die die minimale Anzahl an verbindenden Variablen erzeugt, ein NP-schweres Problem ist [JR94].

Nach der Implementierung der Staircase-Heuristik wird zunächst untersucht, ob sie bei Instanzen mit garantierter Staircase-Struktur diese Struktur detektiert. Dazu werden in Kapitel 3.2 Instanzen des Resource Allocation Problems (RAP) untersucht. Sie weisen in ihrer Ausgangsformulierung eine Staircase-Struktur auf, werden permutiert und als Input für die Staircase-Heuristik verwendet.

Außerdem wird in den Kapiteln 3.3 und 3.4 getestet, ob die Staircase-Heuristik in der Lage, bei Instanzen mit unbekannter Struktur eine Struktur zu detektieren, wie gut und bei wie vielen Instanzen dies gelingt. Dazu wird die Heuristik auf alle Instanzen der MIPLIB2003 [AKM06] sowie auf eine Auswahl aus der MIPLIB2010 [KAA$^+$11] angewendet. In Kapitel 4 werden zwei Verbesserungen für die Staircase-Heuristik vorgeschlagen, um Schwachstellen zu beseitigen, die nach Auswertung der Ergebnisse sichtbar werden. Abgeschlossen wird die Arbeit in Kapitel 5 mit einem Fazit und Anstößen für weitere Untersuchungen.

2. Grundlagen

Dieses Kapitel befasst sich mit den Grundlagen, die für das Verständnis der vorliegenden Arbeit wichtig sind. Zunächst werden in Abschnitt 2.1 einige grundlegende Begriffe wie gemischt-ganzzahliges Programm (engl. Mixed Integer Program, MIP) definiert. Um das Ziel – die Zerlegung eines MIPs in Blöcke – zu erreichen, sind zwei Schritte erforderlich: Zunächst werden Zeilen und Spalten der Matrix $\boldsymbol{A}$ (s. Def. 1) permutiert, welche die Koeffizienten der Nebenbedingungen des Problems enthält. Dies geschieht mittels des *Rank-Order-Clustering-Algorithmus* (ROC-Algorithmus), der in Abschnitt 2.2 erläutert wird. Der ROC-Algorithmus ordnet die Nichtnulleinträge der Matrix $\boldsymbol{A}$ bandförmig entlang der Diagonalen der Matrix an. In einem zweiten Schritt werden mittels des *Blockings* die Nebenbedingungen und Variablen in Blöcke eingeteilt, um das Problem in Teilprobleme zu zerlegen (s. Abschn. 2.3).

2.1. Grundlegende Begriffe

Gemischt-ganzzahliges Programm Als erstes werden gemischt-ganzzahlige Programme definiert, da diese Problemklasse in dieser Arbeit untersucht wird.

Definition 1 *Gemischt-Ganzzahliges Lineares Programm (engl. Mixed Integer Program ,MIP)*
Gegeben $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, $p \in \{0, 1, \ldots, n\}$.
Dann heißt

$$\begin{aligned} \max \quad & \boldsymbol{c}^T \boldsymbol{x} \\ & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{Z}^p \times \mathbb{R}^{n-p} \end{aligned} \tag{2.1}$$

gemischt-ganzzahliges lineares Programm (Optimierungsproblem).

Die Matrix $\boldsymbol{A}$ wird im Folgenden *Nebenbedingungsmatrix* genannt.

Staircase-Struktur Gegeben seien $\boldsymbol{x}_t = (x_{t1}, \ldots, x_{tn})$ und $t = 1, \ldots, T$. Das resultierende MIP hat eine Staircase-Struktur, wenn es die folgende Form aufweist:

$$
\begin{array}{lll}
\min & \boldsymbol{c}_1\boldsymbol{x}_1 + \boldsymbol{c}_2\boldsymbol{x}_2 + \ldots + \boldsymbol{c}_T\boldsymbol{x}_T & \\
\text{s. t.} & \boldsymbol{A}_{11}\boldsymbol{x}_1 & \geq \boldsymbol{b}_1 \\
& \boldsymbol{A}_{21}\boldsymbol{x}_1 + \boldsymbol{A}_{22}\boldsymbol{x}_2 & \geq \boldsymbol{b}_2 \\
& \boldsymbol{A}_{32}\boldsymbol{x}_2 + \boldsymbol{A}_{33}\boldsymbol{x}_3 & \geq \boldsymbol{b}_3 \\
& \ddots & \vdots \\
& \boldsymbol{A}_{T(T-1)}\boldsymbol{x}_{(T-1)} + \boldsymbol{A}_{TT}\boldsymbol{x}_T & \geq \boldsymbol{b}_T \\
& \boldsymbol{x}_t \geq 0, \; t = 1, \ldots, T. &
\end{array}
$$

Dabei sorgen die Matrizen $\boldsymbol{A}_{21}, \ldots, \boldsymbol{A}_{T(T-1)}$ für eine Kopplung der Teilprobleme.

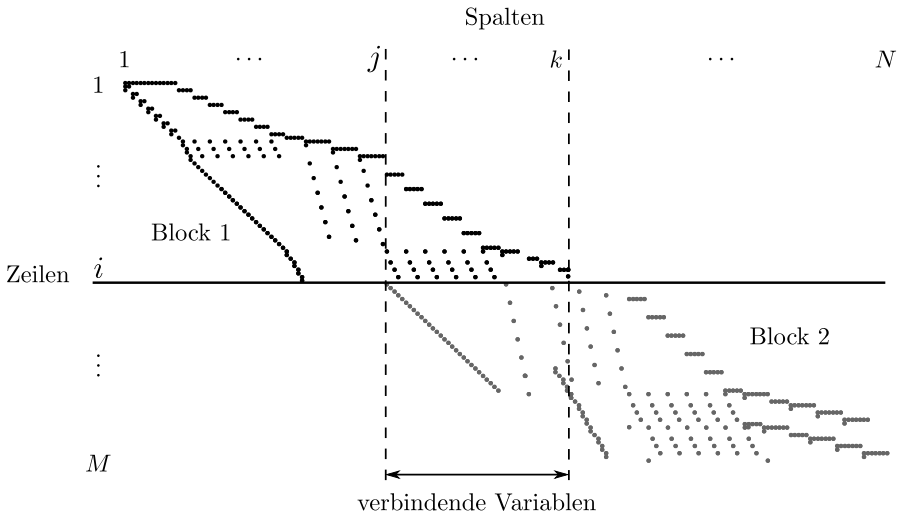


Abbildung 2.1: Einteilung der Nebenbedingungsmatrix $\boldsymbol{A} \in \mathbb{R}^{M \times N}$ in zwei Blöcke

Blöcke und verbindende Variablen Die angestrebte Zerlegung der Nebenbedingungsmatrix $\boldsymbol{A}$ in Blöcke – nachfolgend nur noch *Blocking* genannt – lässt sich am besten anhand einer Graphik erklären. Abbildung 2.1 zeigt die Struktur der Nebenbedingungsmatrix $\boldsymbol{A} \in \mathbb{R}^{M \times N}$ für ein willkürliches MIP nach Ablauf des ROC-Algorithmus, wobei M und N die Anzahl der Nebenbedingungen (Zeilen) bzw. Variablen (Spalten) sind. Jeder Punkt repräsentiert einen Nichtnulleintrag, d. h. eine Variable, deren Koeffizient ungleich Null ist. Das gegebene MIP soll der Einfachheit wegen in zwei Blöcke unterteilt werden, es kann aber prinzipiell auch in mehr Blöcke zerlegt werden. In der Konvention dieser Arbeit erfolgt das Blocking nach einer Zeile, in diesem Beispiel nach Zeile i. Dadurch werden die Zeilen $1, \ldots, i$ Block 1 zugeordnet (schwarz), während die Zeilen $i + 1, \ldots, M$ zu Block 2 gehören (grau). Für die Zuordnung der Variablen zu Blöcken werden zunächst diejenigen Variablen ermittelt, die zu zwei Blöcken gehören. Dies sind

die Variablen $j+1, \ldots, k$, die sich zwischen den beiden vertikalen, gestrichelten Linien befinden. Sie sind als *verbindende Variablen* definiert und werden gesondert gespeichert. Die restlichen Variablen $1, \ldots, j$ sowie $k+1, \ldots, N$ tauchen nur in einem Block auf und können so eindeutig Block 1 bzw. Block 2 zugeordnet werden. Für alle Einträge „rechts oben“ und „links unten“, d. h. $a_{p,q} \in \boldsymbol{A}$ mit $(p \leq i \ \wedge \ q > k) \ \vee \ (p > i \ \wedge \ q \leq j)$, muss gelten $a_{p,q} = 0$.

Resource Allocation Problem Das *Resource Allocation Problem*, auch bekannt als *temporal knapsack problem* oder *unsplittable flow on a line*, wird in [Fur11] beschrieben und in dieser Arbeit in Kapitel 3.2 zur Untersuchung der Staircase-Heuristik eingesetzt.

Gegeben ist eine Menge von Prozessen $j = 1, \ldots, n$ mit Ressourcenverbrauch w_j und Nutzen p_j. Die Gesamtmenge verfügbarer Ressourcen beträgt C. Jeder Prozess hat einen Start- und Endzeitpunkt o_j bzw. f_j und heißt aktiv zum Zeitpunkt t, falls $o_j \leq t \leq f_j$. Beim RAP geht es darum, eine Teilmenge der Prozesse auszuwählen, die den Nutzen maximiert, ohne dass zu irgendeinem Zeitpunkt die Gesamtmenge verfügbarer Ressourcen überschritten wird.

Eine erste Version des RAP ergibt sich, wenn die Nebenbedingungen zum Startzeitpunkt jedes Prozesses formuliert werden und T_z als die Menge aller zum Zeitpunkt t_z aktiven Prozesse ist:

$$\max \sum_{j=1}^{n} p_j x_j \tag{2.2}$$

$$\sum_{j \in T_z} w_j x_j \leq C \quad z = 1, \ldots, n \tag{2.3}$$

$$x_i \in \{0, 1\} \quad i = 1, \ldots, n \tag{2.4}$$

Ein Prozess ist ausgewählt, falls $x_j = 1$ ist. Formuliert man die Nebenbedingungen wie in Gleichung (2.3), entstehen allerdings Nebenbedingungen, die von anderen dominiert werden. Es ist jedoch möglich, die Anzahl der Nebenbedingungen so zu reduzieren, dass nur nicht dominierte Nebenbedingungen formuliert werden. Dazu werden die Nebenbedingungen nur dann zum Startzeitpunkt eines Prozesses formuliert, wenn das nächste Ereignis die Beendigung eines Prozesses ist, wobei als Ereignis in diesem Kontext der Start und die Beendigung eines Prozesses gilt. Die Prozesse, für die das zuvor gesagte zutrifft, seien die Prozesse $k = 1, \ldots, m$. Somit ergeben sich $m < n$ Nebenbedingungen für die finale Formulierung des RAP:

$$\max \sum_{j=1}^{n} p_j x_j \tag{2.5}$$

$$\sum_{j \in T_k} w_j x_j \leq C \quad k = 1, \ldots, m \tag{2.6}$$

$$x_i \in \{0, 1\} \quad i = 1, \ldots, n \tag{2.7}$$

2.2. Rank-Order-Clustering-Algorithmus

Die Durchführung des Rank-Order-Clustering-Algorithmus (ROC-Algorithmus) ist der erste Schritt zum Auffinden einer möglichen Staircase-Struktur (vgl. Abbildung 1.1). Der von King [Kin80] entwickelte ROC-Algorithmus überführt eine Matrix mit beliebiger Struktur der Nichtnulleinträge in eine Matrix, bei der sich die Nichtnulleinträge bandförmig entlang der Diagonalen anordnen. Allerdings hat die ursprüngliche Form des ROC-Algorithmus zwei große Einschränkungen: Zum einen erfordert sie die Speicherung der Matrix als zweidimensionales Array, was bei großen dünnbesetzten Matrizen zu einem übermäßigen Speicherverbrauch führt. Des Weiteren ist die Laufzeit des Algorithmus von $\mathcal{O}(MN(M+N))$, wobei M und N die Anzahl der Zeilen und Spalten sind.

Deshalb wird in dieser Arbeit eine andere Variante des ROC-Algorithmus implementiert, die ebenfalls von King [KN82] entwickelt wurde. Sie hat den Vorteil, dass sie Radixsort verwendet und somit eine geringere Laufzeit von $\mathcal{O}(K)$ hat, wobei K die Anzahl der Nichtnulleinträge ist. Außerdem werden die Nichtnulleinträge der Matrix in Listen gespeichert, was zu einer effizienteren Speichernutzung führt. Zur Unterscheidung von der ursprünglichen Form des ROC-Algorithmus wird diese Variante in der Literatur und auch in dieser Arbeit als ROC2-Algorithmus bezeichnet.

2.2.1. Prinzip des ROC2-Algorithmus

	1	2	3	4	5	6
1		1		1	1	
2	1					1
3			1	1		
4	1	1				1
5			1	1	1	

Tabelle 2.1: 5×6 Beispielmatrix zur Erläuterung des ROC2-Algorithmus.

Der ROC2-Algorithmus wird anhand der Beispielmatrix in Tabelle 2.1 erläutert. Als erstes müssen die Zeilen der Matrix vertauscht werden. Dazu wird eine Zeilenreihenfolgeliste mit (1 2 3 4 5) initialisiert. Nun werden die Spalten von der letzten bis zur ersten Spalte in einer Schleife durchlaufen. Folglich wird mit Spalte 6 begonnen. Es werden die Zeilen unterstrichen, die einen Nichtnulleintrag (eine „1“) in dieser Spalte haben. Dies sind die Zeilen zwei und vier. Diese beiden Elemente werden unter Beibehaltung ihrer Reihenfolge an den Anfang der Zeilenreihenfolgeliste verschoben. Die Zeilenreihenfolgeliste lautet nun (2 4 1 3 5). Der beschriebene Vorgang wird anschließend für Spalte 5 wiederholt. Diese hat Einträge in Zeile 1 und 5, welche unterstrichen und an den Anfang

der Zeilenreihenfolgeliste verschoben werden. Dieser Vorgang wird für alle Spalten wiederholt und ist in Tabelle 2.2 zusammengefasst. Als neue Reihenfolge der Zeilen ergibt sich (4 2 1 5 3).

		Zeileneinträge				
Spalte	6	1	2	3	4	5
	5	2	4	1	3	5
	4	1	5	2	4	3
	3	1	5	3	2	4
	2	5	3	1	2	4
	1	1	4	5	3	2
Neue Reihenfolge		4	2	1	5	3

Tabelle 2.2: Zeilenpermutation mittels ROC2-Algortihmus

	1	2	3	4	5	6
4	1	1				1
2	1					1
1		1		1	1	
5			1	1	1	
3			1	1		

Tabelle 2.3: Beispielmatrix nach dem Vertauschen der Zeilen.

Das Aussehen der Beispielmatrix nach Vertauschung der Zeilen ist in Tabelle 2.3 wiedergegeben und Ausgangspunkt für die Vertauschung der Spalten. Diese erfolgt komplett analog zur Vertauschung der Zeilen wenn die Begriffe „Zeile“ und „Spalte“ vertauscht werden. Das Vertauschen der Spalten wird in Tabelle 2.4 gezeigt. Zu beachten ist, dass die *vertauschten* Zeilen rückwärts durchlaufen werden, also in der Reihenfolge $3, 5, 1, 2, 4$. Die neue Reihenfolge der Spalten nach der Vertauschung ergibt sich gemäß der letzten Zeile in Tabelle 2.4 zu (1 6 2 4 5 3). Damit ist eine Iteration des ROC2-Algorithmus abgeschlossen. Tabelle 2.5 dokumentiert das Aussehen der Beispielmatrix nach Vertauschung von Zeilen und Spalten. Gut zu erkennen ist, wie der ROC2-Algorithmus die Nichtnulleinträge in einem Band entlang der Diagonalen angeordnet hat.

Algorithmus 1 auf S. 8 fasst den Ablauf des ROC zusammen. Die Zeilen 2-5 beschreiben die Vertauschung von Zeilen und 6-9 die Vertauschung der Spalten. Der Algorithmus wird solange fortgesetzt, bis sich weder bei den Zeilen noch bei den Spalten die Reihenfolge ändert.

		Spalteneinträge					
Zeile	3	1	2	3	4	5	6
	5	3	4	1	2	5	6
	1	3	4	5	1	2	6
	2	4	5	2	3	1	6
	4	1	6	4	5	2	3
Neue Reihenfolge		1	6	2	4	5	3

Tabelle 2.4: Spaltenpermutation mittels ROC2-Algortihmus

	1	6	2	4	5	3
4	1	1	1			
2	1	1				
1			1	1	1	
5				1	1	1
3				1		1

Tabelle 2.5: Beispielmatrix nach dem Vertauschen der Zeilen und Spalten.

Algorithmus 1 ROC2-Algorithmus in Anlehnung an [Kin80]

1: **repeat**
2: **for** Spalte ← LetzteSpalte, ErsteSpalte **do** ▷ Zeilenvertauschung
3: Finde die Zeilen mit Nichtnulleinträgen;
4: Verschiebe die Zeilen mit Nichtnulleinträgen an den Anfang der Zeilenreihenfolgeliste und behalte die Reihenfolge der Einträge bei
5: **end for**
6: **for** Zeile ← LetzteZeile, ErsteZeile **do** ▷ Spaltenvertauschung
7: Finde die Spalten mit Nichtnulleinträgen;
8: Verschiebe die Spalten mit Nichtnulleinträgen an den Anfang der Spaltenreihenfolgeliste und behalte die Reihenfolge der Einträge bei
9: **end for**
10: **until** Keine Veränderung

2.2.2. Eigenschaften des ROC2-Algorithmus

Anzahl der Iterationen In dem vorangegangenen Beispiel wurde lediglich eine Iteration des ROC2-Algorithmus betrachtet. Allerdings hat dort eine Iteration ausgereicht, um die Matrix in ihren finalen Zustand zu überführen. Weitere Iterationen hätten keine weitere Vertauschung von Zeilen und Spalten bewirkt. Jedoch sind im Allgemeinen mehrere Iterationen des ROC2-Algorithmus erforderlich, bevor keine Veränderung der Zeilen- und Spaltenreihenfolge mehr auftritt. Da jede Iteration des ROC2-Algorithmus die Position von mindestens einer Zeile oder Spalte fixiert, sind maximal $M + N$ Iterationen notwendig, um die finale Permutation der Matrix zu erreichen [JR94].

Anordnung der Zeilen und Spalten Liest man die Einträge der Matrix als binäre Werte (z. B. 000101 für die letzte Zeile in Tabelle 2.5), so sind die Zeilen nach Ablauf des ROC2-Algorithmus in absteigender Reihenfolge dieser Werte angeordnet. Dasselbe gilt für die Spalten, wenn man die Einträge spaltenweise liest.

Finden unabhängiger Blöcke Falls in einer Matrix unabhängige Blöcke existieren, wird der ROC2-Algorithmus sie finden, unabhängig davon, wie die Matrix zu Beginn des ROC2-Algorithmus aussieht [Kin80]. Diese Eigenschaft ist sehr nützlich für das Ziel, eine Staircase-Struktur mit möglichst wenig verbindenden Variablen zu finden.

2.3. Blocking-Algorithmus

Der zweite Schritt auf dem Weg zum Auffinden einer möglichen Staircase-Struktur einer Matrix ist das *Blocking* (vgl. Abbildung 1.1). Das Blocking folgt dem Vorgehen von [JR94] und hat das Ziel, die Anzahl der entstehenden verbindenden Variablen zu minimieren. In Kapitel 2.1 wurde bereits dargelegt, wie Variablen und Nebenbedingungen einem Block zugeordnet werden, wenn die Zeile bekannt ist, nach welcher geblockt wird (vgl. Abbildung 2.1). Wie die Anzahl der Blöcke und die Stellen, an denen geblockt wird, bestimmt werden, wird in den folgenden beiden Abschnitten erläutert.

2.3.1. Bestimmung der Blockanzahl

Die Anzahl der Blöcke, in die die Nebenbedingungsmatrix unterteilt werden soll, ist anfangs unbekannt und muss bestimmt werden. Zu diesem Zweck müssen zunächst einige Variablen definiert werden. Seien `ibegin[i]` und `iend[i]` der Index der ersten bzw. letzten Spalte in Zeile i, die einen Nichtnulleintrag enthalten. Daraus ergibt sich die „Breite“ der Nichtnulleinträge in Zeile i zu `width[i]` = `iend[i]` − `ibegin[i]`. Sei nun $\bar{n}$ die durchschnittliche Anzahl von Variablen in einem Block, $\bar{v}$ die durchschnittliche Anzahl von

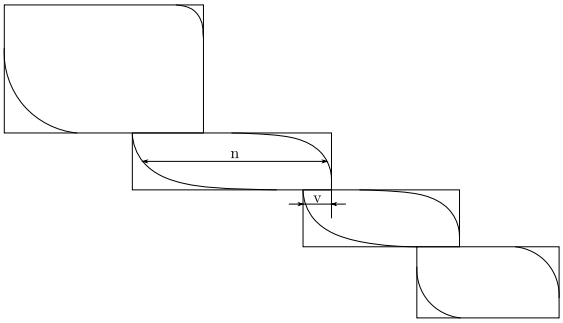


Abbildung 2.2: Die schematische Darstellung der Nichtnulleinträge der Nebenbedingungsmatrix $\boldsymbol{A}$ zeigt, wie Schätzwerte für n und v zur Bestimmung der Blockanzahl ermittelt werden. In Anlehnung an [JR94]

verbindenden Variablen und T die Anzahl der Blöcke. Damit lässt sich eine Gleichung für die Gesamtzahl der Variablen N aufstellen (vgl. Abbildung 2.2)

$$T\bar{n} - (T-1)\bar{v} = N \quad . \tag{2.8}$$

Diese kann nach der gesuchten Blockanzahl T aufgelöst werden:

$$T = \frac{N - \bar{v}}{\bar{n} - \bar{v}} \tag{2.9}$$

Da $\bar{n}$ und $\bar{v}$ nicht bestimmt werden können, bevor die Blockanzahl T feststeht (welche gerade gesucht wird), werden die folgenden Näherungswerte für $\bar{n}$ und $\bar{v}$ gewählt:

$$\tilde{n} = \max_{i \in I}(\texttt{width[i]}), \qquad I = \{1, \ldots, M\} \tag{2.10}$$

$$\tilde{v} = \min_{i \in I}(\texttt{width[i]}), \qquad I = \{1, \ldots, M\} \tag{2.11}$$

Somit kann abschließend ein Näherungswert für die angestrebte Blockanzahl $\tilde{T}$ berechnet werden

$$\tilde{T} = \text{nint}\left(\frac{N - \tilde{v}}{\tilde{n} - \tilde{v}}\right) \quad , \tag{2.12}$$

wobei die Funktion $\text{nint}(x)$ ihrem Argument x die nächste ganze Zahl zuordnet (nearest integer function).

2.3.2. Zuordnung von Variablen und Nebenbedingungen zu Blöcken

In diesem Abschnitt wird erläutert, wie die Zeilen bestimmt werden, nach denen ein neuer Block beginnt. Das Ziel ist, mit dem Blocking möglichst wenig verbindende Variablen zu erzeugen. Deshalb wird zunächst ein Ausdruck für eine Variable hergeleitet, die angibt, wie viele verbindende Variablen entstehen, wenn nach Zeile i geblockt wird.

Dazu wird zunächst die Variable `jmax[i]` eingeführt, die den Index der Spalte mit dem letzten Nichtnulleintrag in oder vor Zeile i angibt:

$$\texttt{jmax[i]} = \max\left(\texttt{iend[i]}, \texttt{jmax[i-1]}\right) \text{ mit } \texttt{jmax[0]} = \texttt{iend[0]} \quad . \tag{2.13}$$

Mithilfe dieser Variablen ist es möglich, die Anzahl der verbindenden Variablen zu berechnen, die bei einem Blocking nach Zeile i entsteht:

$$\texttt{minV[i]} = 1 + \texttt{jmax[i]} - \texttt{ibegin[i+1]} \quad . \tag{2.14}$$

Da die Anzahl verbindender Variablen minimiert werden soll, wird in dem Array `minV` nach lokalen Minima gesucht. `minV[i]` ist ein lokales Minimum, falls

$$\texttt{minV[i]} < \texttt{minV[i-1]} \ \wedge \ \texttt{minV[i]} < \texttt{minV[i+1]} \quad . \tag{2.15}$$

Alle lokalen Minima in `minV` werden in der Variablen `constrictions` gespeichert. `constrictions` enthält somit Indices der Zeilen, an denen die bandförmige Matrixstruktur „Einschnürungen“ aufweist und somit potentielle Kandidaten für das Blocking sind.

	1	2	3	4	5	6
1	1	1	1			
2	1	1				
3			1	1	1	
4				1	1	1
5				1		1

Tabelle 2.6: Beispielmatrix nach Ablauf des ROC2-Algorithmus.

Die Bedeutung der bisher eingeführten Variablen wird an dieser Stelle mittels eines Beispiels veranschaulicht. Die in Tabelle 2.6 gezeigte Struktur der Matrix ist aus dem Beispiel auf Seite 8 aufgegriffen. Allerdings sind die Zeilen- und Spaltenindizes im Gegensatz zu den permutierten Indizes aus dem vorangegangenen Beispiel hier in geordneter Reihenfolge wiedergegeben, um Missverständnisse zu vermeiden.

Die Werte der Arrays `ibegin` und `iend` (erster und letzter Nichtnulleintrag in einer Zeile) lassen sich leicht ablesen zu

$$\begin{aligned} \texttt{ibegin} &= [1\ 1\ 3\ 4\ 4] \quad \text{und} \\ \texttt{iend} &= [3\ 2\ 5\ 6\ 6] \quad . \end{aligned}$$

Mithilfe der Gleichungen (2.13) und (2.14) ergeben sich daraus `jmax` und `minV` zu

$$\begin{aligned} \texttt{jmax} &= [3\ 3\ 5\ 6\ 6] \quad \text{und} \\ \texttt{minV} &= [3\ 1\ 2\ 3] \quad . \end{aligned}$$

Das einzige lokale Minimum in `minV` befindet sich an zweiter Stelle. Deshalb enthält das Array `constrictions` = [2] nur einen Eintrag. Die Beispielmatrix wird nach der zweiten Zeile geblockt, da dort die einzige Einschnürung vorliegt. Die Zeilen 1 und 2 werden Block 1, die Zeilen 3-5 Block 2 zugeordnet. Da `minV[2]` = 1 ist, entsteht eine verbindende Variable. Dies ist Variable 3 im Beispiel, da sie Nichtnulleinträge in beiden Blöcken hat. Die restlichen Variablen 1-2 und 4-6 werden Block 1 bzw. 2 zugeordnet. Damit ist das Beispiel abgeschlossen.

Das Beispiel war so konstruiert, dass es nur eine Einschnürung gab. Allgemein können jedoch viele Einschnürungen auftreten. Die Aufgabe besteht darin, eine Auswahl aus der Menge der Einschnürungen zu treffen, so dass möglichst die angestrebte Blockanzahl $\tilde{T}$ erreicht wird. Deshalb geht der Blocking-Algorithmus wie folgt vor: Das Array `constrictions`, das alle Einschnürungen der Matrix enthält, wird der Reihenfolge nach durchlaufen. Es wird allerdings nur dann geblockt, wenn der entstehende Block mindestens $M/2\tilde{T}$ umfasst. Dadurch wird verhindert, dass viele kleine Blöcke erzeugt werden, jedoch entstehen im schlechtesten Fall doppelt so viele Blöcke wie angestrebt. Falls weniger als $M/\tilde{T}$ Zeilen verbleiben, werden diese dem letzten Block zugeordnet und das Blocking ist abgeschlossen.

3. Ergebnisse

In diesem Kapitel wird zunächst die verwendete Hardware und Software beschrieben. Im Anschluss daran werden Ergebnisse diskutiert, die bei der Anwendung der Staircase-Heuristik auf Instanzen des RAP sowie der MIPLIB2003 und MIPLIB2010 entstehen.

3.1. Hardware- und Software-Setup

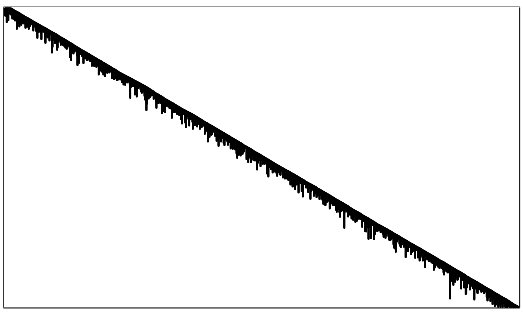
Alle Rechnungen wurden auf dem Bull Cluster der RWTH Aachen durchgeführt. Dabei kamen Intel Xeon X5675 "Westmere EP" Prozessoren mit 3.06 GHz, 12 Kernen und 24 Threads zum Einsatz. Der verfügbare Hauptspeicher betrug 24 GB. Als Betriebssystem wurde Scientific Linux 6.2 mit Kernel-Version 2.6.32 verwendet.

Auf Seiten der Software wurde das auf dem `SCIP`-Framework [Ach09] basierende `GCG` [GL10] genutzt. Die Zeit für die Berechnungen wurde auf vier und zwei CPU-Stunden für die RAP- bzw. MIPLIB-Instanzen limitiert. Der Lösungsvorgang wurde nach der Erkennungsphase abgebrochen, so dass das eigentliche MIP nicht gelöst wurde. Deshalb ist es nicht möglich, Aussagen über eine mögliche Zeitersparnis durch das Ausnutzen von Staircase-Strukturen zu treffen.

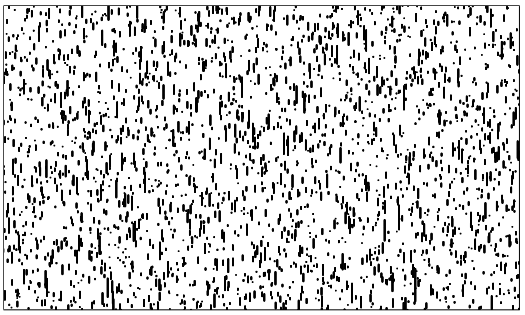
3.2. Resource Allocation Problem

Die Heuristik zur Erkennung von Staircase-Strukturen wird zunächst an 60 Instanzen des Resource Allocation Problems (RAP; vgl. Kapitel 2.1) getestet. Diese Instanzen haben die Eigenschaft, dass sie in ihrer Ausgangsformulierung bereits eine Staircase-Struktur aufweisen. Um die Wirksamkeit der Heuristik zu testen, werden die Zeilen und Spalten der Nebenbedingungsmatrix jeder Instanz zufällig permutiert und als Input für die Heuristik verwendet. Anschließend wird nach Ablauf der Heuristik die detektierte Struktur mit der des Ausgangsproblems verglichen. Alle Instanzen sind öffentlich zugänglich unter `http://www.or.deis.unibo.it/research.html`.

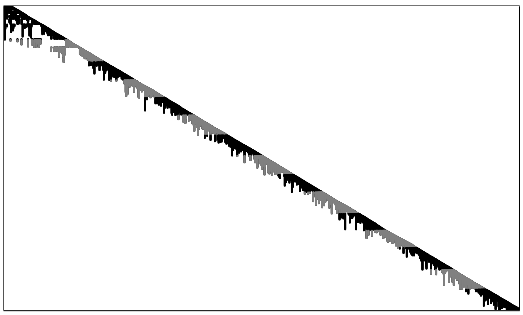
Abbildung 3.1 zeigt beispielhaft die Anwendung der Heuristik auf eine RAP-Instanz. Jeder Punkt repräsentiert einen Nichtnulleintrag in der Nebenbedingungsmatrix. Beim Ausgangsproblem ist die Staircase-Struktur gut zu erkennen, während das permutierte



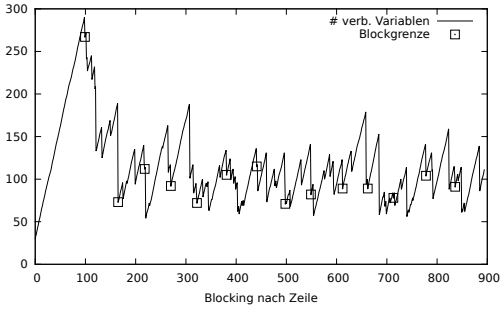
(a) Ausgangsproblem



(b) Permutiertes Problem (Input)



(c) Erkannte Struktur und Blocking (Output)



(d) Anzahl verbindender Variablen bei Blocking nach Zeile i (`minV`)

Abbildung 3.1: Anwendung der Heuristik zur Erkennung von Staircase-Strukturen auf die RAP-Instanz I42.

Problem keine erkennbare Struktur aufweist (s. Abbildung 3.1a und 3.1b). Der ROC2-Algorithmus der Heuristik bewirkt, dass in Abbildung 3.1c wieder eine Staircase-Struktur identifiziert werden kann. Auch wenn kein quantitativer Parameter zur Beurteilung der Güte der Staircase-Struktur definiert wurde, so erscheint die erkannte Struktur doch optisch ansprechend. Als einzige Ausnahme kann man hier die ersten beiden Blöcke anführen, die größere, rechteckige leere (weiße) Bereiche enthalten, die in den restlichen Blöcken und im Ausgangsproblem nicht beobachtet wurden. Diese leeren Bereiche entstanden in allen getesteten Instanzen immer in den ersten Blöcken. Ein Grund für die Entstehung oder eine Möglichkeit zur Beseitigung dieser Strukturen ist nicht bekannt. Insgesamt wurde die Nebenbedingungsmatrix in 15 Blöcke (schwarz und grau) unterteilt.

Als letztes ist in Abbildung 3.1d der Verlauf der Anzahl der verbindenden Variablen dargestellt, die bei einem Blocking nach der entsprechenden Zeile entstehen. Dies ist folglich die graphische Repräsentation des Arrays `minV`, das durch Gleichung (2.15) definiert ist. Es ist gut zu sehen, dass die durch Quadrate repräsentierten Blockgrenzen an Stellen lokaler Minima gesetzt wurden. Allerdings entsteht der Eindruck, dass durch eine intelligentere Wahl der Blockgrenzen die Zahl der verbindenden Variablen weiter gesenkt werden kann.

Name	Zeilen	Spalten	Nichtnull-einträge	τ	Blöcke	Iterationen	verbindende Variablen		CPU-Sekunden
							#	%	
I41	768	2071	23040	7	11	11	1087	52.5	1086
I42	896	2422	26880	9	15	13	1440	59.5	1775
I43	1024	2756	30720	9	15	12	1540	55.9	1869
I44	1152	3104	34560	14	20	12	1877	60.5	961
I45	1280	3433	38400	14	20	16	1916	55.8	1786
I46	1408	3789	42240	12	17	17	2995	79.0	3662
I47	1536	4154	46080	13	20	17	2244	54.0	2922
I48	1664	4476	49920	10	17	16	2981	66.6	5066
I49	1792	4797	53760	11	20	17	3390	70.7	4527
I50	1920	5129	57600	18	20	25	2280	44.5	11098
I51	768	4948	23040	15	20	22	2396	48.4	640
I52	896	5769	26880	17	20	25	3606	62.5	1430
I53	1024	6721	30720	17	20	27	3620	53.9	2069
I54	1152	7382	34560	20	20	26	2140	29.0	1559
I55	1280	8266	38400	20	20	46	3010	36.4	7251
I57	1536	9865	46080	20	20	38	3069	31.1	13322
I61	768	2071	23040	7	11	11	1087	52.5	793
I62	896	2422	26880	10	17	13	1269	52.4	1294
I63	1024	2763	30720	10	15	18	1613	58.4	2052
I64	1152	3103	34560	16	20	25	1667	53.7	4361
I65	1280	3434	38400	9	15	15	2563	74.6	2701
I66	1408	3774	42240	10	18	10	2188	58.0	2251
I67	1536	4164	46080	17	20	17	1746	41.9	4580
I68	1664	4488	49920	9	16	28	2663	59.3	11287
I70	1920	5142	57600	14	20	24	3739	72.7	11101
I71	768	2916	23236	8	13	12	2088	71.6	371
I72	896	3424	26718	11	17	13	2117	61.8	721
I73	1024	3832	30777	12	20	20	2009	52.4	1109
I74	1152	4316	34508	13	20	31	2620	60.7	3863
I75	1280	4771	38570	13	20	15	3008	63.0	2232
I76	1408	5403	42184	18	20	21	3600	66.6	4745
I77	1536	5793	46134	16	20	29	3655	63.1	8555
I78	1664	6167	49891	20	20	30	1752	28.4	9860
I79	1792	6800	53656	14	20	22	3863	56.8	8923
I81	768	5210	23236	15	20	20	3227	61.9	645
I82	896	6057	26901	20	20	34	1835	30.3	1786
I83	1024	6901	30554	20	20	38	3133	45.4	2883
I84	1152	7737	34641	20	20	37	3551	45.9	3360
I85	1280	8656	38545	20	20	54	2279	26.3	8816
I86	1408	9370	42273	20	20	45	3363	35.9	9029
I87	1536	10271	46161	20	20	58	2701	26.3	9879
I89	1792	11992	53756	20	20	49	3630	30.3	13762
I91	768	3117	27076	7	12	15	1892	60.7	788
I92	896	3594	31340	9	15	16	2507	69.8	833
I93	1024	4176	35754	14	20	18	2184	52.3	3058
I94	1152	4671	40138	11	18	13	3171	67.9	2875
I95	1280	5209	44684	12	20	13	2711	52.0	2974
I96	1408	5628	49315	16	20	33	3543	63.0	9657
I97	1536	6215	53802	12	20	19	4279	68.8	5713
I98	1664	6730	58181	16	20	29	3868	57.5	13655
I99	1792	7172	62593	15	20	29	3642	50.8	12058
arithm. Mittel			39156	14,1	18,5	23,8		54,0	4776

Tabelle 3.1: Ergebnisse der Anwendung der Staircase-Heuristik auf RAP-Instanzen.

In Tabelle 3.1 sind die Ergebnisse für die Anwendung der Staircase-Heuristik auf die RAP-Instanzen zusammengefasst. Dabei wurde bei neun der 60 Instanzen das Zeitlimit überschritten, so dass diese nicht in der Tabelle aufgeführt sind.

Die Erkennung einer Staircase-Struktur der permutierten RAP-Instanzen mit durchschnittlich 39156 Nichtnulleinträgen dauert im Mittel knapp 80 Minuten. Außerdem ist zur Laufzeit des ROC2-Algorithmus anzumerken, dass dieser die Grenze im schlechtesten Fall von $M+N$ Iterationen (s. Abschnitt 2.2.2) mit durchschnittlich 23,8 Iterationen deutlich unterschreitet.

Darüber hinaus fällt auf, dass die realisierte Blockanzahl T größer gleich der angestrebten Blockanzahl $\tilde{T}$ ist[1]. Die Ursache dafür ist die Konstruktion des Blocking-Algorithmus: Für eine angestrebte Blockanzahl von $\tilde{T}$ hat ein Block durchschnittlich $M/\tilde{T}$ Zeilen. Allerdings erlaubt der Blocking-Algorithmus ein Blocking bereits wieder nach $M/2\tilde{T}$ der Zeilen. Bei einer ausreichend hohen Zahl von lokalen Minima in `minV` (Einschnürungen), wie sie bei den RAP-Instanzen vorliegen (vgl. Abbildung 3.1d), entstehen folglich Blöcke mit durchschnittlich weniger als $M/\tilde{T}$ Zeilen und somit mehr als $\tilde{T}$ Blöcke.

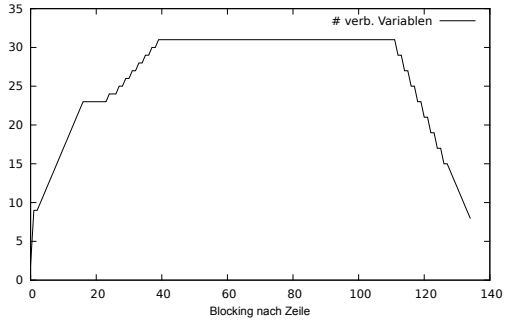
Abschließend werden Anmerkungen zur Interpretation der vorletzten Spalte in Tabelle 3.1 gemacht, die den prozentualen Anteil an verbindenden Variablen an der Gesamtheit aller Variablen zeigt. Es sei daran erinnert, dass die Staircase-Heuristik darauf abzielt, die Anzahl verbindender Variablen zu minimieren, da bei schwacher Kopplung der Teilprobleme Dantzig-Wolfe-Dekomposition effizient funktioniert [Las70]. Daher liegt der Schluss nahe, dass Instanzen mit einem geringen Anteil verbindender Variablen besonders von der Staircase-Heuristik profitieren. Dabei muss jedoch folgender Effekt berücksichtigt werden: Für jeden weiteren Block entsteht eine zusätzliche Serie verbindender Variablen, womit auch ihr Anteil an der Gesamtheit aller Variablen steigt. Folglich gibt es einen Trade-off zwischen kleinen Blöcken und vielen verbindenden Variablen einerseits sowie großen Blöcken und wenig verbindenden Variablen andererseits. Die Instanzen mit einem niedrigen Anteil verbindender Variablen (z. B. I54) haben das (willkürlich gesetzte) Blocklimit von 20 Blöcken erreicht und bestehen somit aus weniger Blöcken als wenn kein Blocklimit gesetzt worden wäre. Daher sinkt auch ihr Anteil verbindender Variablen im Vergleich zu den Instanzen, die das Blocklimit unterschreiten.

3.3. MIPLIB2003

Um mehr über das Potential und die Grenzen der Staircase-Heuristik zu erfahren, wurde diese auf alle 60 Instanzen der MIPLIB2003 [AKM06] angewendet.

[1] Diese Aussage gilt zwar nicht allgemein, aber zumindest für alle getesteten Instanzen. Ist die Anzahl der Einschnürungen (Anzahl Elemente im Array `constrictions`) kleiner als $\tilde{T}$ oder liegen die Einschnürungen „ungünstig“, können auch weniger Blöcke als angestrebt entstehen.

(a) *air04* nach Ablauf des ROC: Ein sinnvolles Blocking ist nicht möglich, da das Band zu breit ist.



(b) *pp08a*: Es existieren keine lokalen Minima im Array `minV`.

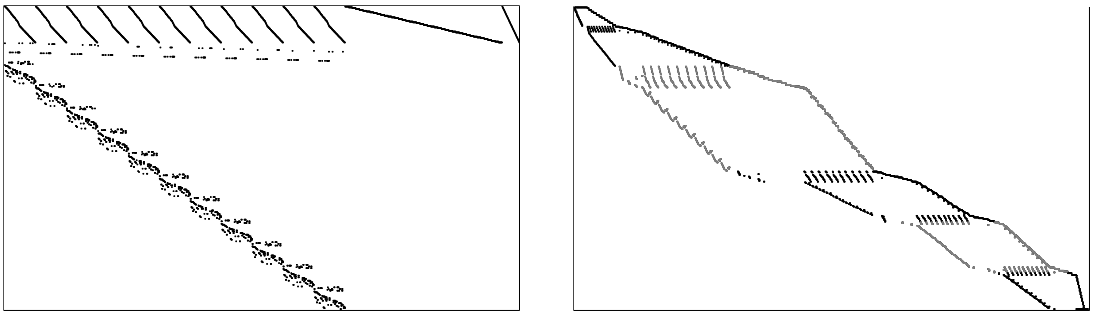
Abbildung 3.2: Für die zwei gezeigten Instanzen konnte kein Blocking gefunden werden.

Bei 15 Instanzen wurde das Zeitlimit von zwei Stunden überschritten und die Detektion abgebrochen. Von den verbleibenden 45 Instanzen wurde bei 30 die Erkennung nach dem ROC gestoppt. Die Ursache dafür ist, dass der ROC2-Algorithmus bei diesen Instanzen ein breites Band ausbildet, wie es in Abbildung 3.2a zeigt. Die maximale Zeilenbreite ist nur geringfügig kleiner als die Gesamtbreite der Matrix, woraus sich mittels Gleichung (2.12) ein Wert für die angestrebte Blockanzahl $\tilde{T} = 1$ ergibt. Deshalb können die Nebenbedingungsmatrizen dieser 30 Instanzen nicht sinnvoll in Blöcke unterteilt werden.

Die verbleibenden 15 Instanzen überführte der ROC2-Algorithmus in eine Struktur, für die sich eine angestrebte Blockanzahl von $\tilde{T} \geq 2$ ergab. Allerdings tritt bei fünf dieser Instanzen eine weitere Schwierigkeit auf, die aus Abbildung 3.2b deutlich wird. Sie zeigt den Verlauf des Arrays `minV`, das angibt, wie viele verbindende Variablen entstehen, wenn nach einer Zeile geblockt wird. Der Blocking-Algorithmus sucht wie in Abschnitt 2.3 beschrieben nach lokalen Minima in `minV`, kann aufgrund des Verlaufs jedoch keine lokalen Minima finden. Deswegen können die fünf Instanzen mit diesem charakteristischen Verlauf von `minV` nicht in Blöcke zerlegt werden.

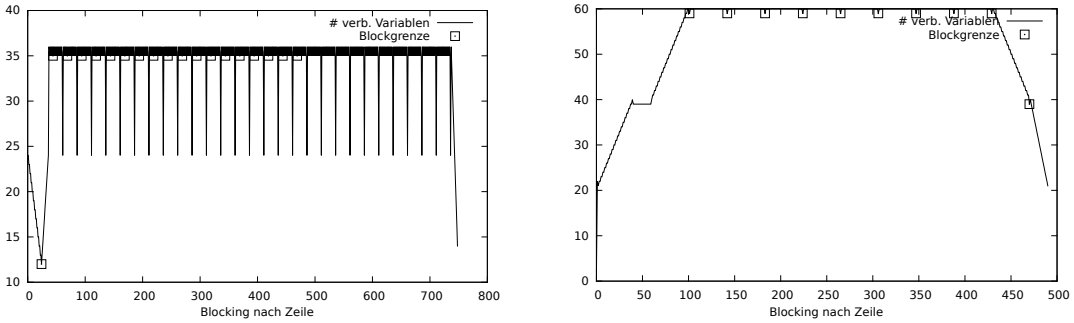
Somit ist nur bei zehn von 45 (22%) innerhalb des Zeitlimits gerechneten Instanzen der MIPLIB2003 von der Heuristik ein Blocking durchgeführt worden. In Abbildung 3.3 ist beispielhaft die *fiber*-Instanz dargestellt, für die ein Blocking gefunden wurde. Die Qualität des Blockings kann infrage gestellt werden, da der Anteil der verbindenden Variablen an der Gesamtzahl aller Variablen mit 79% hoch ist.

Als letzte graphische Analyse soll der Blick nochmal auf den Verlauf des Arrays `minV` in Abbildung 3.4 gerichtet werden. Im linken Graphen (Abbildung 3.4a) ist zu erkennen, dass es zwei Gruppen von lokalen Minima gibt: Die eine Gruppe mit 35 und die andere mit 24 entstehenden verbindenden Variablen. Auf den ersten Blick ist klar, dass es sinnvoll wäre, die Blockgrenzen bei den Minima mit Wert 24 zu wählen. Allerdings berücksichtigt der Blocking-Algorithmus in keiner Weise die „Güte“ oder das „Niveau“



(a) Ausgangsproblem **(b)** Erkannte Struktur

Abbildung 3.3: Anwendung der Staircase-Heuristik auf die *fiber*-Instanz aus der MIPLIB2003.



(a) *Set1ch*: Ungünstige Wahl der Blockgrenzen an Stellen lokaler Minima mit hohem Funktionswert.

(b) Trotz der großen Ähnlichkeit zu dem Graphen in Abb. 3.2b werden hier 11 Blöcke detektiert.

Abbildung 3.4: Der Verlauf des Arrays `minV` zeigt zwei Schwächen des Blocking-Algorithmus.

der lokalen Minima, so dass alle Blockgrenzen bei Minima mit Wert 35 gewählt werden, was zu einem schlechten Blocking führt.

Der rechte Graph (Abbildung 3.4b) verdeutlicht eine weitere ungewollte Eigenschaft des Blocking-Algorithmus: Der Verlauf des Graphen ist fast identisch mit dem Verlauf des Graphen aus Abbildung 3.2b und folglich haben beide Instanzen (nach dem ROC) eine ähnliche Struktur. Jedoch unterteilt der Blocking-Algorithmus in dem einen Fall das Problem in 11 Blöcke, während in dem anderen Fall keine Unterteilung stattfindet und keine Staircase-Struktur detektiert wird. Der Grund für dieses Verhalten sind die schwach ausgeprägten (aber vorhandenen) Minima am oberen Rand von Abbildung 3.4b, wohingegen der Graph in Abbildung 3.2b keine Minima hat. Diese starke Veränderung des Outputs bei nur schwacher Änderung des Inputs ist eine negative Eigenschaft des Algorithmus.

In Tabelle 3.2 sind die Ergebnisse zusammengefasst. Der obere Teil zeigt die 10 Instanzen,

Name	Zeilen	Spalten	Nichtnull-einträge	$\tilde{T}$	Blöcke	Itera-tionen	verbindende Variablen		CPU-Sekunden
							#	%	
a1c1s1	3312	3648	10178	16	17	22	3304	90,6	309,0
fiber	363	1298	2944	3	5	8	1027	79,1	5,2
gesa2-o	1248	1224	3672	11	13	16	1039	84,9	25,2
gesa2	1392	1224	5064	11	13	15	1042	85,1	37,5
mod011	4480	10958	22254	2	3	13	5638	51,5	636,2
modglob	291	422	968	5	8	13	339	80,3	1,1
pp08aCUTS	246	240	839	5	7	11	189	78,8	0,5
set1ch	492	712	1412	11	11	5	570	80,1	1,1
tr12-30	750	1080	2508	20	20	5	623	57,7	3,0
vpm2	234	378	917	4	6	5	279	73,8	0,3
arithm. Mittel			5538	9,3	10,8	12,0		76,4	113,2
aflow30a	479	842	2091	2	1	6	0	0	2,3
aflow40b	1442	2728	6783	2	1	6	0	0	23,5
mas74	13	151	1706	2	1	1	0	0	0,1
mas76	12	151	1640	2	1	1	0	0	0,1
pp08a	136	240	480	7	1	6	0	0	0,1

Tabelle 3.2: Ergebnisse der Anwendung der Staircase-Heuristik auf Instanzen aus der MIPLIB2003.

bei denen eine Staircase-Struktur gefunden wurde. Für den Mittelwert dieser Instanzen gelten dieselben Aussagen wie für die RAP-Instanzen: Es entstehen mehr Blöcke als angestrebt, der ROC2-Algorithmus benötigt deutlich weniger Iterationen als im schlechtesten Fall ($M+N$) und es gibt einen hohen Anteil verbindender Variablen (76,4%), der allerdings immer im Zusammenhang mit der Blockanzahl gesehen werden muss.

Abschließend sind im unteren Teil von Tabelle 3.2 die fünf Instanzen dargestellt, bei denen das Array `minV` einen ähnlichen Verlauf aufweist, wie er in Abbildung 3.2b gezeigt ist. Wegen der fehlenden Einschnürungen konnten sie nicht in mehrere Blöcke unterteilt werden.

3.4. MIPLIB2010

Insgesamt wurden aus der MIPLIB2010 [KAA+11] 39 Instanzen getestet. Es wurden nur Instanzen mit weniger als 5000 Nichtnulleinträgen ausgewählt, um die Laufzeit kurz zu halten. Außerdem wurde eine Instanz aus der Auswahl genommen, da sie nur aus einer Nebenbedingung besteht und daher nicht in Blöcke unterteilt werden kann.

Bei der Betrachtung der in Tabelle 3.3 zusammengefassten Resultate erhält man diesel-

Name	Zeilen	Spalten	Nichtnull-einträge	$\tilde{T}$	Blöcke	Itera-tionen	verbindende Variablen #	verbindende Variablen %	CPU-Sekunden
berlin_5_8_0	1532	1083	4507	3	5	10	567	52,4	19,3
bg512142	1307	792	3953	7	12	16	550	69,4	23,6
binkar10_1	1026	2298	4496	7	11	16	2039	88,7	34,3
enlight13	169	338	962	6	6	12	176	52,1	0,7
enlight14	196	392	1120	7	6	13	194	49,5	1,0
enlight15	225	450	1290	7	6	14	216	48,0	1,6
enlight16	256	512	1472	8	7	15	258	50,4	2,3
enlight9	81	162	450	4	4	8	84	51,9	0,1
g200x740i	940	1480	2960	19	20	22	1264	85,4	23,0
go19	441	441	1885	9	14	20	362	82,1	5,5
ic97_potential	1046	728	3138	4	7	10	634	87,1	9,1
neos-1426635	796	520	3400	2	4	6	221	42,5	4,9
neos-1440460	989	468	4302	2	4	6	204	43,6	7,8
neos15	552	792	1766	17	20	6	627	79,2	1,8
p80x400b	480	800	1600	2	2	6	286	35,8	1,6
arithm. Mittel			2487	6,9	8,5	12,0		61,2	9,1
k16x240	256	480	960	2	1	4	0	0	0,3
neos-1225589	675	1300	2525	2	1	5	0	0	3,1
p100x588b	688	1176	2352	2	1	6	0	0	3,4
r80x800	880	1600	3200	2	1	7	0	0	7,5
ran14x18	284	504	1008	2	1	5	0	0	0,5
ran16x16	288	512	1024	2	1	5	0	0	0,5

Tabelle 3.3: Ergebnisse der Anwendung der Staircase-Heuristik auf Instanzen aus der MIPLIB2010.

ben Erkenntnisse, wie sie für die MIPLIB2003 im vorangegangenen Abschnitt ausführlich beschrieben wurden und bestätigen diese damit. Neue Erkenntnisse lassen sich allerdings nicht ableiten. Bei 18 Instanzen ergab sich nach dem ROC eine angestrebte Blockanzahl von $\tilde{T} = 1$, sechs Instanzen hatten keine Einschnürungen, so dass sie nicht geblockt werden konnten und bei 15 Instanzen wurde eine Staircase-Struktur detektiert. Diese Zahlen für die MIPLIB2010 sind zusammen mit denen der MIPLIB2003 in Tabelle 3.4 zusammengefasst.

	MIPLIB2003	MIPLIB2010	Summe
Instanzen gesamt	60	39	99
Im Zeitlimit (< 2h)	45	39	84
$\tilde{T} \geq 2$	15	21	36
erfolgreiche Detektion	10	15	25

Tabelle 3.4: Zusammenfassung der Testrechnungen der MIPLIB2003 und MIPLIB2010.

4. Verbesserungsvorschläge für die Staircase-Heuristik

Während der Implementierung der Staircase-Heuristik und der Auswertung der Ergebnisse wurden einige Kritikpunkte an der Heuristik offenbar. Dieses Kapitel dient dem Zweck, zwei Schwachstellen nochmals explizit darzulegen und mögliche Verbesserungsansätze zu skizzieren, die im Rahmen dieser Arbeit nicht ausgearbeitet und getestet wurden.

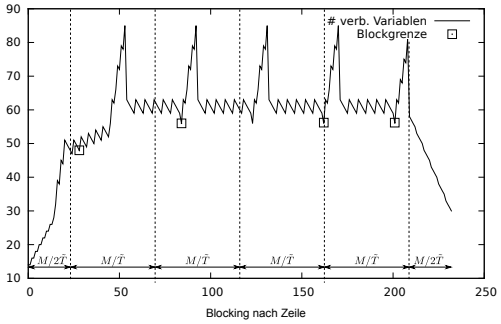


Abbildung 4.1: Verbesserungsvorschlag für die Wahl der Blockgrenzen: Durch Ordnen der lokalen Minima jedes Bereichs nach ihren Funktionswerten kann das Blocking verbessert werden.

Anhand von Abbildung 3.4a wurde bereits erläutert, dass wegen der ungünstigen Lage der lokalen Minima von `minV` für das Blocking Minima mit hohem Funktionswert ausgewählt wurden, obwohl auch Minima mit niedrigerem Funktionswert hätten gewählt werden können. Zur Verbesserung dieses Verhaltens wird vorgeschlagen, das Array `minV` in Bereiche der Breite $M/2\tilde{T}, M/\tilde{T}, M/\tilde{T}, \ldots, M/\tilde{T}, M/2\tilde{T}$ zu unterteilen, wie es in Abbildung 4.1 zu sehen ist. Im ersten und letzten Bereich wird keine Blockgrenze gesetzt. In jedem der mittleren Bereiche werden die lokalen Minima nach der Höhe ihres Funktionswertes sortiert und das Minimum mit dem kleinsten Funktionswert wird ausgewählt. Um zu kleine Blöcke zu verhindern, kann gefordert werden, dass zwischen zwei Minima eine Mindestanzahl an Zeilen liegen muss. Ist das für zwei Minima nicht der Fall, wird in einem Block auf das Minimum mit nächst höherem Funktionswert ausgewichen.

Der zweite Verbesserungsvorschlag adressiert die Feststellung, dass nach Ausführung des ROC2-Algorithmus eine Blockanzahl $\tilde{T} \geq 2$ angestrebt wird, aber kein Blocking durchgeführt werden kann, weil das Array `minV` keine lokalen Minima aufweist (vgl. Abbildung

3.2b). In diesem Fall ist es oft sinnvoll, die Matrix trotzdem in Blöcke zu unterteilen um somit eine Staircase-Struktur zu erhalten. Dies könnte beispielsweise realisiert werden, indem die Bedingung „<“ an die Nachbarpunkte eines lokalen Minimums (vgl. Gleichung (2.15)) durch „≤“ an einen oder beide Nachbarn gelockert wird oder die Matrix in $\tilde{T}$ gleich große Blöcke unterteilt wird.

5. Fazit und Ausblick

Es ist bekannt, dass Lösungen von MIPs mittels Dantzig-Wolfe-Dekomposition dann effizient gefunden werden, wenn die Struktur der Probleme ausgenutzt wird [VW10]. Deshalb wurde in dieser Arbeit die in [JR94] vorgeschlagene Heuristik zur Erkennung von Staircase-Strukturen in das `GCG/SCIP`-Framework implementiert. Um die Fragen zu beantworten, welche der Probleme eine Staircase-Struktur aufweisen und wie gut der Algorithmus es vermag, sie in eine Staircase-Struktur zu überführen, wurde die Heuristik an insgesamt 159 Instanzen des RAP, der MIPLIB2003 und MIPLIB2010 getestet.

Zunächst wurde der Frage nachgegangen, ob die Heuristik in der Lage ist, bei Problemen mit garantierter Staircase-Struktur diese zu finden. Dafür wurden die getesteten RAP-Instanzen, die in ihrer Ausgangsformulierung eine Staircase-Struktur aufweisen, permutiert und anschließend als Input für die Staircase-Heuristik verwendet. Bei allen 51 der 60 Instanzen, für welche die Berechnung das vierstündige Zeitlimit unterschritt, war die Heuristik in der Lage, eine Staircase-Struktur zu detektieren. Daraus lässt sich schlussfolgern, dass falls eine Instanz eine (verschleierte) Staircase-Struktur besitzt, die Staircase-Heuristik mit hoher Wahrscheinlichkeit in der Lage ist, diese Struktur zu detektieren. Die durchschnittliche Laufzeit der Erkennung von 80 Minuten war dabei akzeptabel.

In einem zweiten Schritt wurde die Heuristik auf Instanzen angewendet, die in ihrer Ausgangsformulierung keine Staircase-Struktur aufweisen. Hier galt es die Frage zu klären, ob die Heuristik bei diesen Instanzen eine Staircase-Struktur detektiert und wie gut das gelingt. Insgesamt wurden 99 Instanzen aus der MIPLIB2003 und MIPLIB2010 getestet. Von 84 innerhalb des Zeitlimits von zwei Stunden gerechneten Instanzen konnte bei weniger als einem Drittel eine Staircase-Struktur detektiert werden. Dafür gibt es zwei unterschiedliche Gründe.

Der erste Grund für eine nicht erkannte Staircase-Struktur ist, dass die Instanz keine Staircase-Struktur besitzt. Die Struktur ist nicht so allgemein, dass jede beliebige Matrix in die Staircase-Form überführt werden könnte. Nach Ablauf des ROC scheinen 57% der Testinstanzen keine Staircase-Struktur zu besitzen. Allerdings ist es möglich, dass eine andere Methode bei einigen dieser Instanzen doch eine Staircase-Struktur findet.

Der zweite Grund für eine Nichterkennung ist, dass es eine Staircase-Struktur gibt, die Heuristik aber kein Blocking durchführt. Dies geschieht z. B. , wenn der ROC2-Algorithmus die Matrix in ein schmales Band überführt, aber das Array `minV` keine

lokalen Minima aufweist. Für diesen Fall wurde im vorangegangenen Kapitel ein weniger restriktiver Blocking-Algorithmus vorgeschlagen, so dass die Matrix auf jeden Fall unterteilt werden kann.

Weitere Untersuchungen sollten sich der Frage widmen, ob durch Ausnutzen der detektierten Staircase-Strukturen der Aufwand zum Lösen von MIPs reduziert wird, so dass der zusätzliche Aufwand für die Erkennung gerechtfertigt wird.

Des Weiteren wäre eine nähere Betrachtung des Trade-offs zwischen kleinen Blöcken und vielen verbindenden Variablen einerseits sowie großen Blöcken und wenig verbindenden Variablen andererseits interessant. Ist es möglich, ein besseres Kriterium für die Blockanzahl oder den Anteil verbindender Variablen zu finden? Dieser Frage sollte in Zukunft nachgegangen werden.

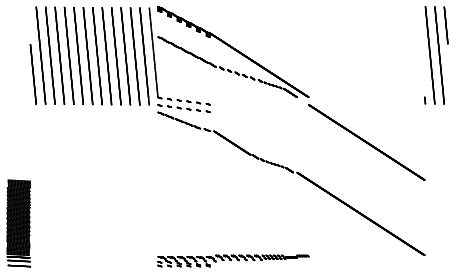
In eine ähnliche Richtung zielt auch die Frage, ob neben des Kriteriums „Minimiere die Anzahl der verbindenden Variablen“ auch andere Kriterien existieren, die zu einer guten Staircase-Struktur führen. Es könnte z. B. überprüft werden, ob möglichst viele Blöcke, möglichst gleich große Blöcke oder andere Kriterien zu einer besseren Detektion führen.

Alle bisher genannten Ansätze beziehen sich auf den Blocking-Algorithmus. Abschließend wird eine Anmerkung zum ROC2-Algorithmus gemacht. Bei diesem könnte untersucht werden, ob er möglicherweise „zu viel“ umsortiert und es besser wäre, die Permutationen von Zeilen und Spalten früher zu stoppen und ob es möglich ist, ein geeignetes Abbruchkriterium für diesen Fall zu finden.

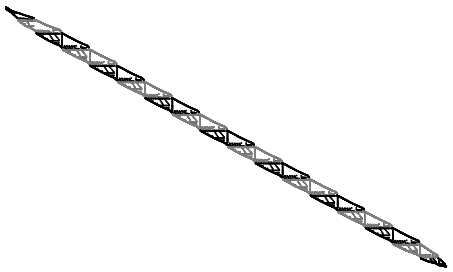
A. Graphiken der Instanzen mit detektierter Staircase-Struktur

Dieses Kapitel zeigt alle Instanzen der MIPLIB2003 und MIPLIB2010 bevor (links) und nachdem (rechts) die Staircase-Heuristik auf sie angewendet wurde. Die RAP-Instanzen sind nicht gezeigt, da alle Graphiken ähnlich zu Abbildung 3.1 sind.

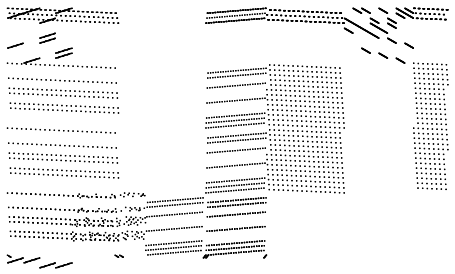
A.1. MIPLIB2003



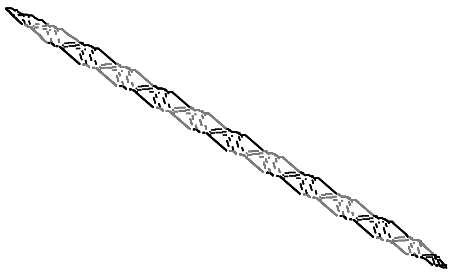
(a) *a1c1s1* (Input)



(b) *a1c1s1* (Output)

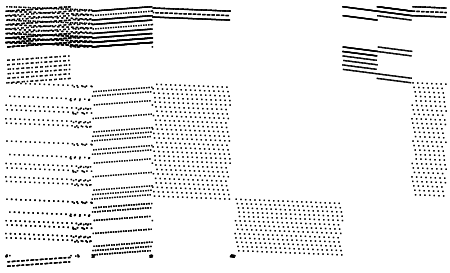


(c) *gesa2-o* (Input)

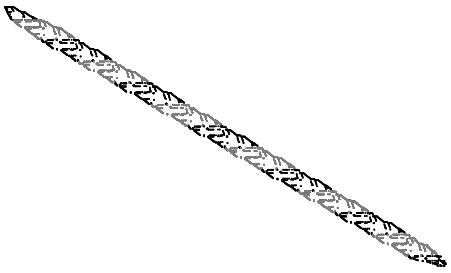


(d) *gesa2-o* (Output)

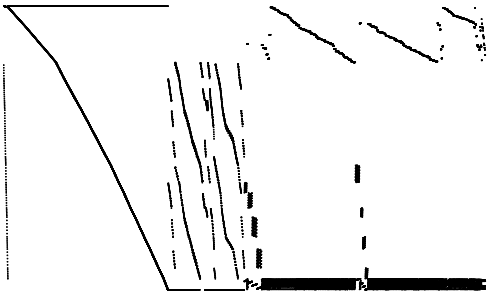
Abbildung A.1



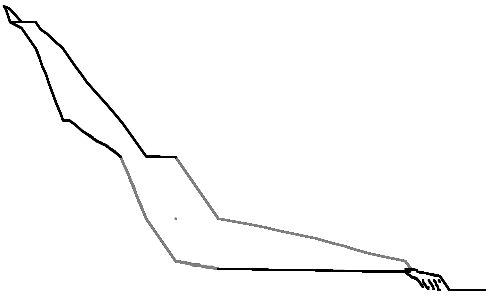
(a) *gesa2* (Input)



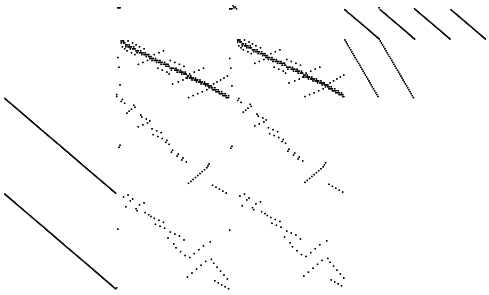
(b) *gesa2* (Output)



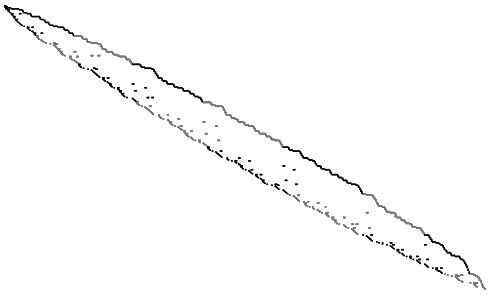
(c) *mod011* (Input)



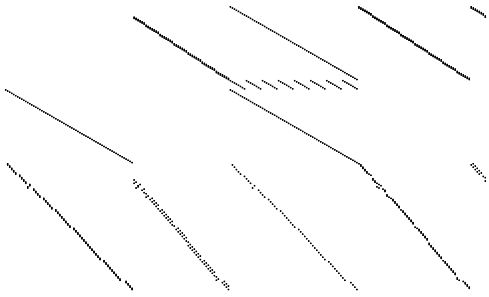
(d) *mod011* (Output)



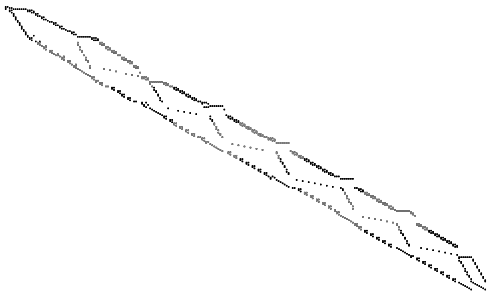
(e) *modglob* (Input)



(f) *modglob* (Output)

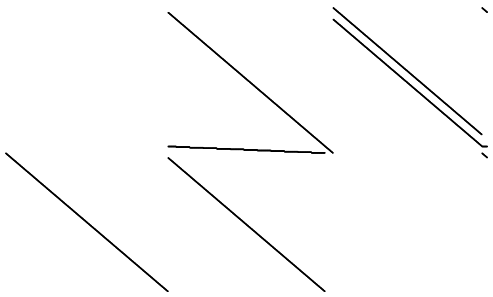


(g) *pp08aCUTS* (Input)

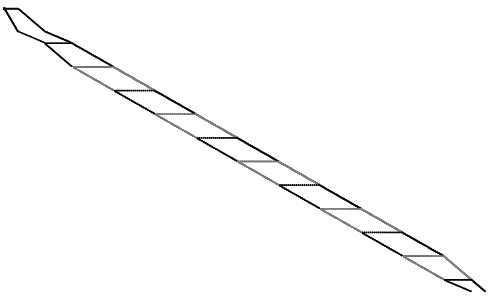


(h) *pp08aCUTS* (Output)

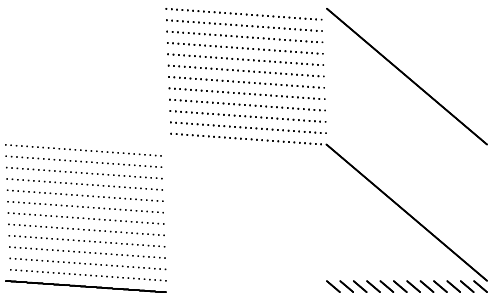
Abbildung A.2



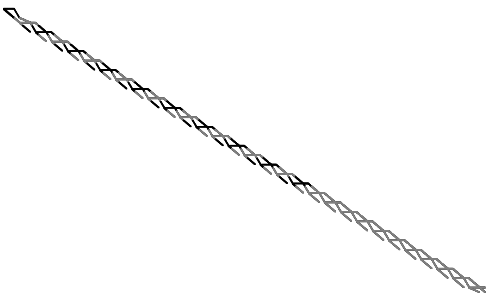
(a) *set1ch* (Input)



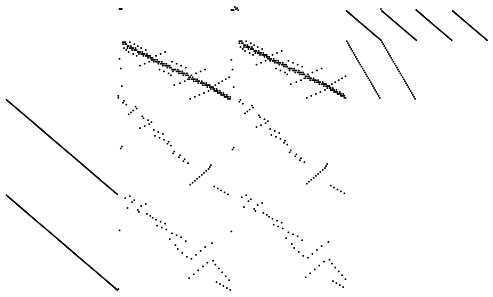
(b) *set1ch* (Output)



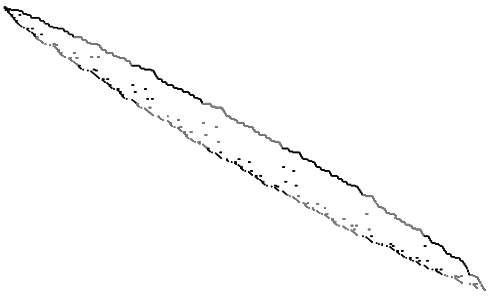
(c) *tr12-30* (Input)



(d) *tr12-30* (Output)



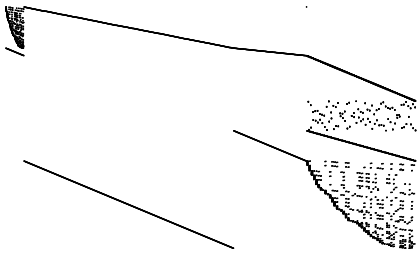
(e) *vpm2* (Input)



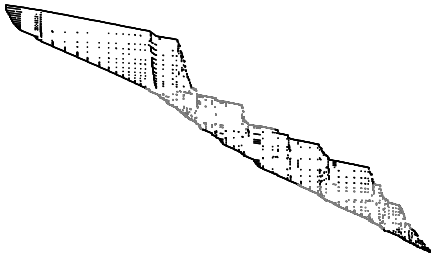
(f) *vpm2* (Output)

Abbildung A.3

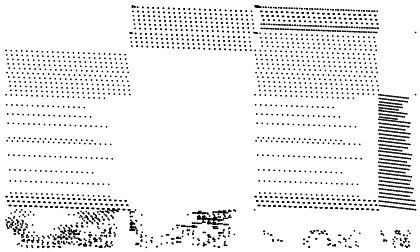
A.2. MIPLIB2010



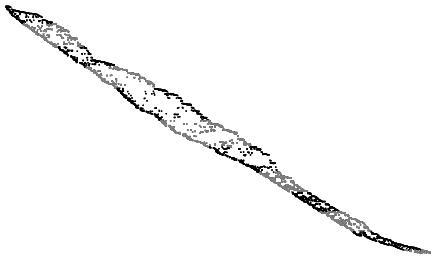
(a) *berlin_5_8_0* (Input)



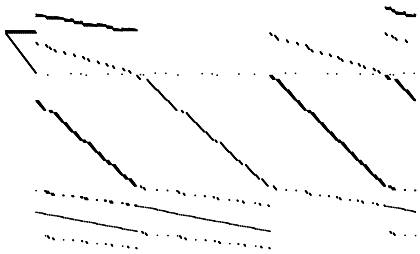
(b) *berlin_5_8_0* (Output)



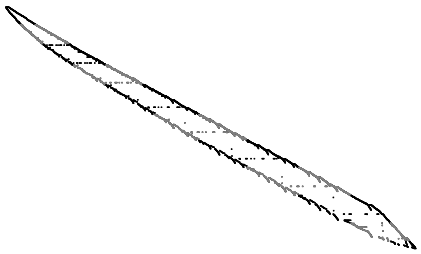
(c) *bg512142* (Input)



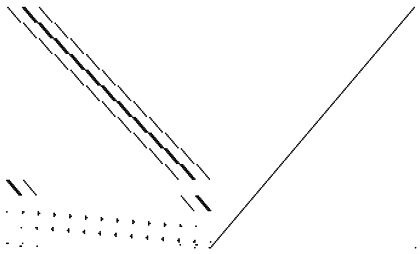
(d) *bg512142* (Output)



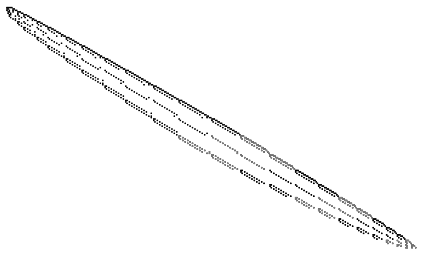
(e) *binkar10_1* (Input)



(f) *binkar10_1* (Output)

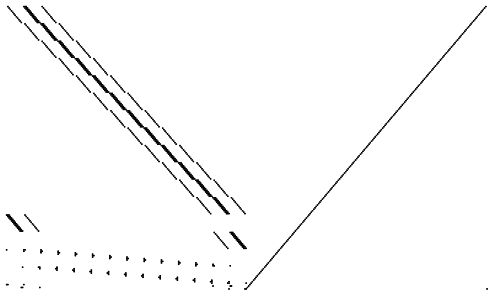


(g) *enlight13* (Input)

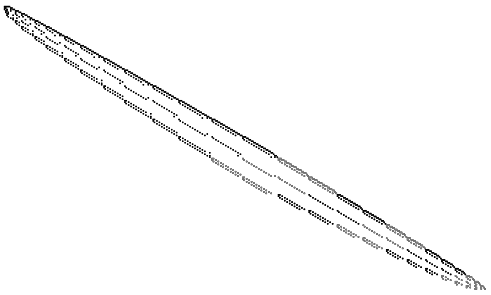


(h) *enlight13* (Output)

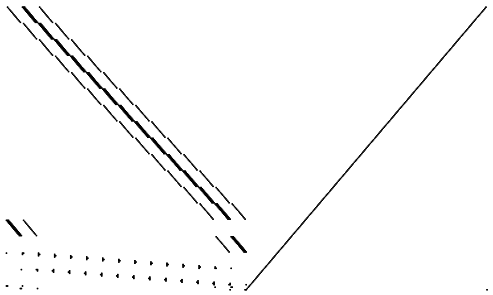
Abbildung A.4



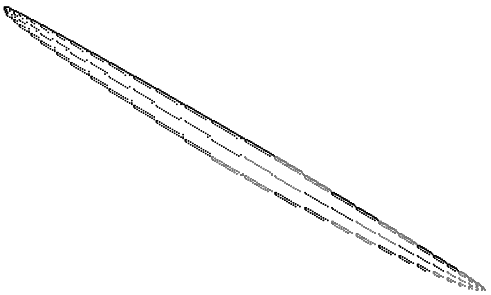
(a) *enlight14* (Input)



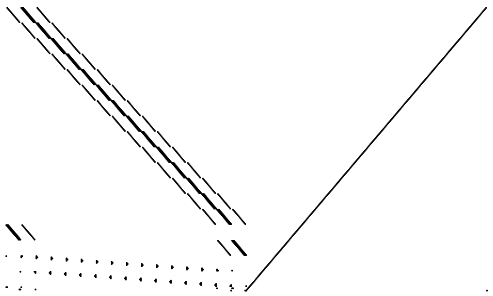
(b) *enlight14* (Output)



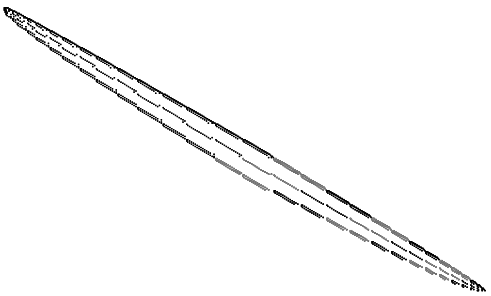
(c) *enlight15* (Input)



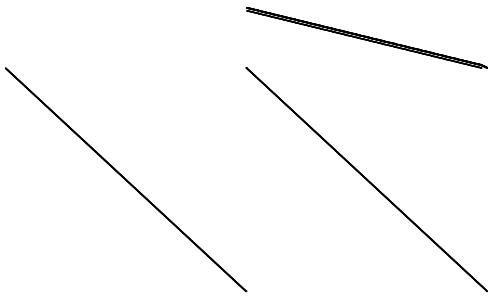
(d) *enlight15* (Output)



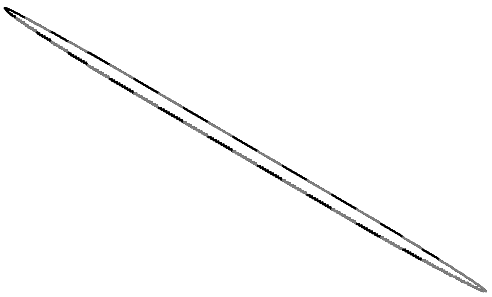
(e) *enlight16* (Input)



(f) *enlight16* (Output)

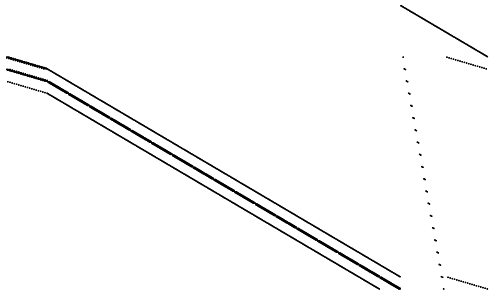


(g) *g200x740i* (Input)

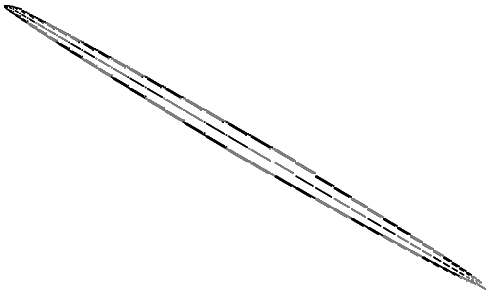


(h) *g200x740i* (Output)

Abbildung A.5

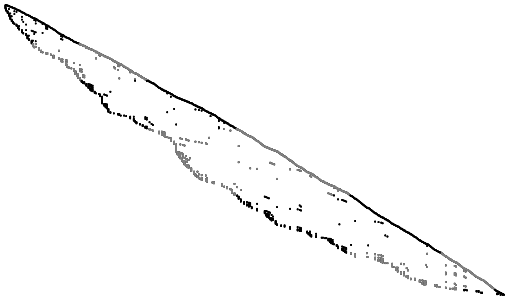


(a) *go19* (Input)

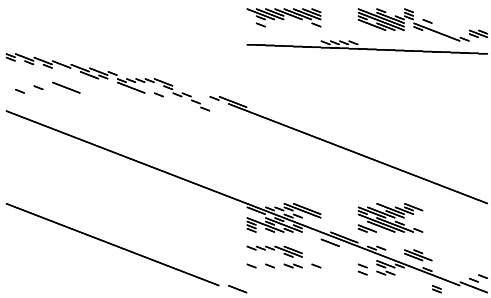


(b) *go19* (Output)

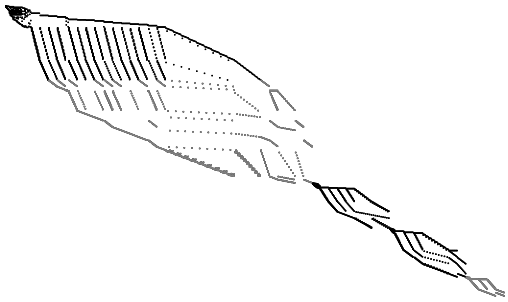
(c) *ic97_potential* (Input)



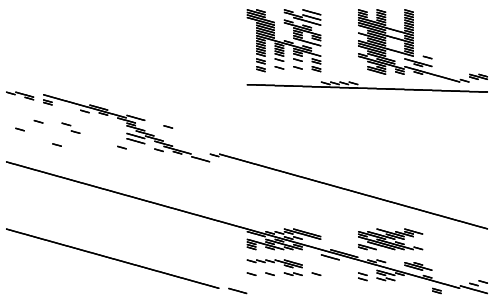
(d) *ic97_potential* (Output)



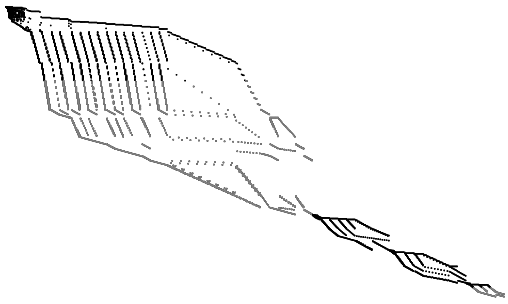
(e) *neos-1426635* (Input)



(f) *neos-1426635* (Output)

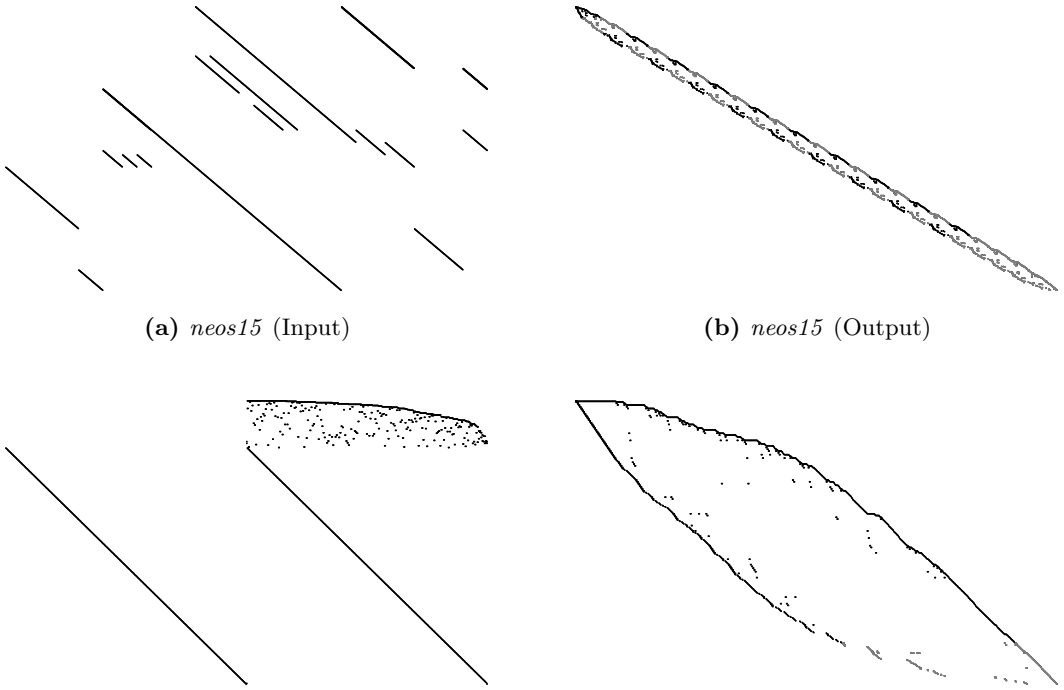


(g) *neos-1440460* (Input)



(h) *neos-1440460* (Output)

Abbildung A.6



(a) *neos15* (Input)

(b) *neos15* (Output)

(c) *p80x400b* (Input)

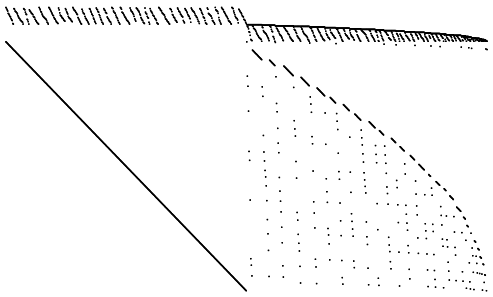
(d) *p80x400b* (Output)

Abbildung A.7

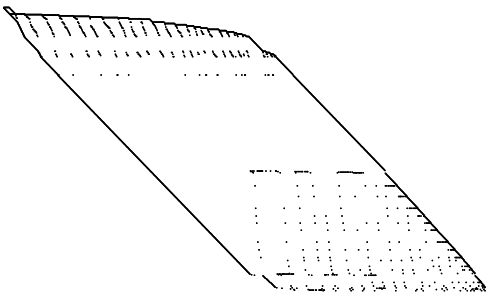
B. Graphiken nicht-geblockter Instanzen

In diesem Kapitel sind die Instanzen gezeigt, bei denen ein Blocking angestrebt wurde, aber wegen des Verlaufs von `minV` nicht durchgeführt werden konnte. Folglich entspricht dies den Instanzen im unteren Teil von Tabelle 3.2 und Tabelle 3.3.

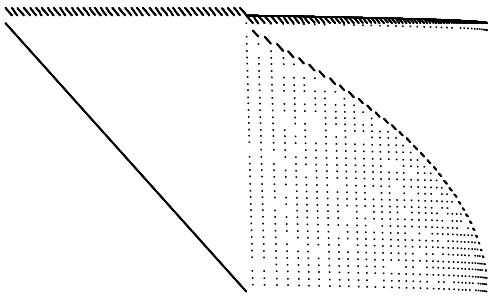
B.1. MIPLIB2003



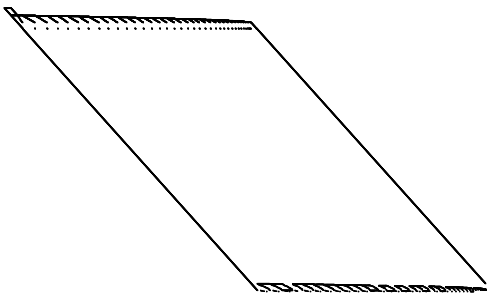
(a) *aflow30a* (Input)



(b) *aflow30a* (Output)



(c) *aflow40b* (Input)



(d) *aflow40b* (Output)

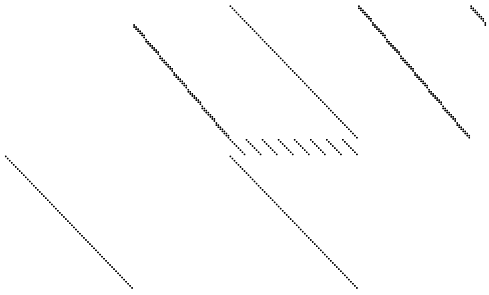
Abbildung B.1

(a) *mas74* (Input)

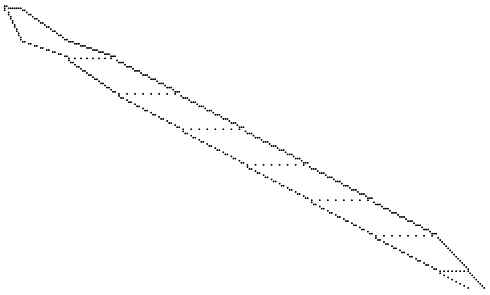
(b) *mas74* (Output)

(c) *mas76* (Input)

(d) *mas76* (Output)



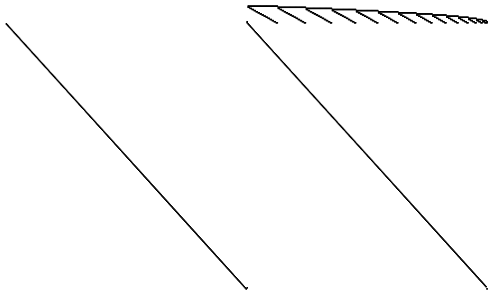
(e) *pp08a* (Input)



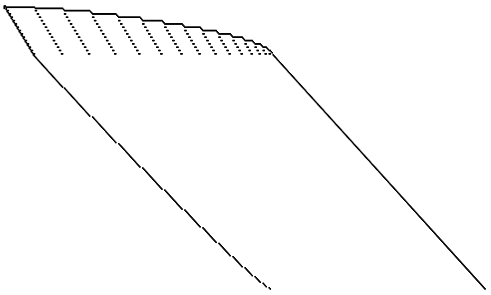
(f) *pp08a* (Output)

Abbildung B.2

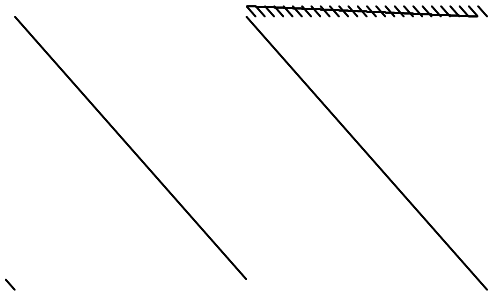
B.2. MIPLIB2010



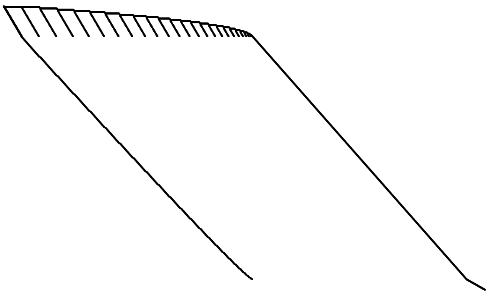
(a) *k16x240* (Input)



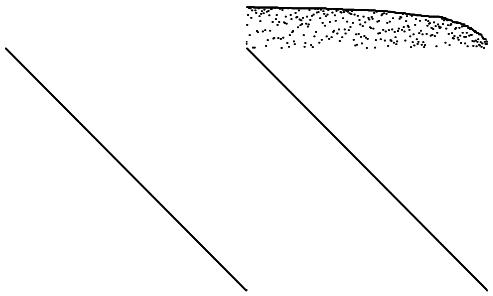
(b) *k16x240* (Output)



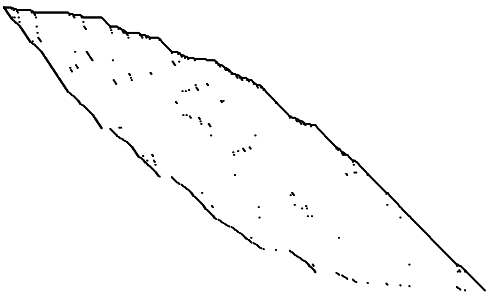
(c) *neos-1225589* (Input)



(d) *neos-1225589* (Output)

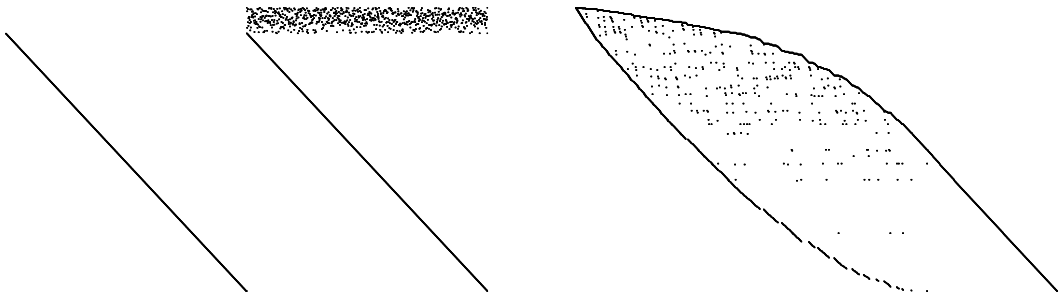


(e) *p100x588b* (Input)

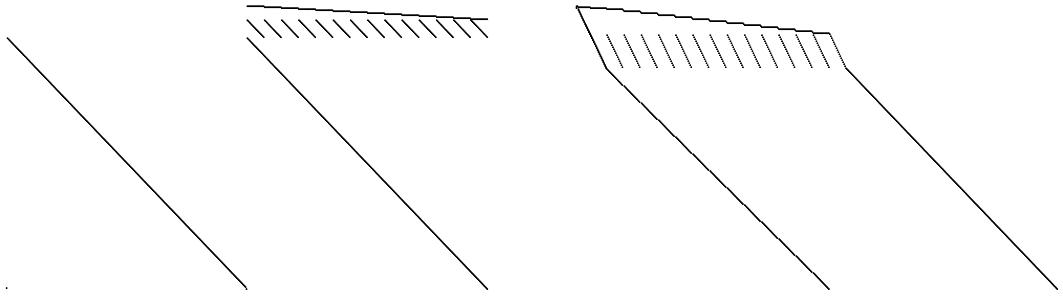


(f) *p100x588b* (Output)

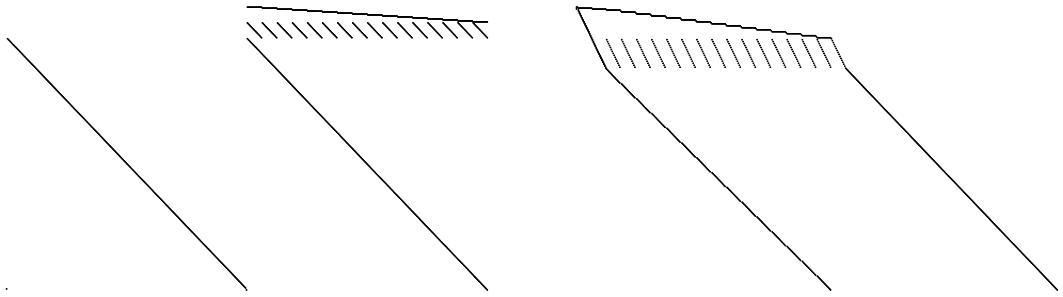
Abbildung B.3



(a) *r80x800* (Input) **(b)** *r80x800* (Output)



(c) *ran14x18* (Input) **(d)** *ran14x18* (Output)



(e) *ran16x16* (Input) **(f)** *ran16x16* (Output)

Abbildung B.4

Literaturverzeichnis

[Ach09] Achterberg, T.: SCIP: Solving constraint integer programs. In: *Mathematical Programming Computation* 1 (2009), S. 1–41. – http://mpc.zib.de/index.php/MPC/article/viewFile/4/6

[AKM06] Achterberg, T. ; Koch, T. ; Martin, A.: MIPLIB 2003. In: *Operations Research Letters* 34 (2006), Nr. 4, 361–372. `http://mpc.zib.de/index.php/MPC/article/viewFile/4/6`

[BCF+11] Bergner, M. ; Caprara, A. ; Furini, F. ; Lübbecke, M. ; Malaguti, E. ; Traversi, E.: Partial Convexification of General MIPs by Dantzig-Wolfe Reformulation. In: Günlük, Oktay (Hrsg.) ; Woeginger, Gerhard (Hrsg.): *Integer Programming and Combinatoral Optimization* Bd. 6655. Springer Berlin / Heidelberg, 2011, S. 39–51

[DW60] Dantzig, G.B. ; Wolfe, P.: Decomposition principle for linear programs. In: *Operations research* (1960), S. 101–111

[Fur11] Furini, F.: Decomposition and reformulation of integer linear programming problems. In: *4OR: A Quarterly Journal of Operations Research* (2011), S. 1–2

[GL10] Gamrath, G. ; Lübbecke, M.: Experiments with a generic Dantzig-Wolfe decomposition for integer programs. In: *Experimental Algorithms* (2010), S. 239–252

[JR94] Jayakumar, M.D. ; Ramasesh, R.V.: A clustering heuristic to detect staircase structures in large scale linear programming models. In: *European journal of operational research* 76 (1994), Nr. 1, S. 229–239

[KAA+11] Koch, T. ; Achterberg, T. ; Andersen, E. ; Bastert, O. ; Berthold, T. ; Bixby, R. E. ; Danna, E. ; Gamrath, G. ; Gleixner, A. M. ; Heinz, S. ; Lodi, A. ; Mittelmann, H. ; Ralphs, T. ; Salvagnin, D. ; Steffy, D. E. ; Wolter, K.: MIPLIB 2010. In: *Mathematical Programming Computation* 3 (2011), Nr. 2, 103–163. `http://mpc.zib.de/index.php/MPC/article/view/56/28`

[Kin80] King, J. R.: Machine-component grouping in production flow analysis: an

approach using a rank order clustering algorithm. In: *International Journal of Production Research* 18 (1980), Nr. 2, S. 213–232

[KN82] King, J. R. ; Nakornchai, V.: Machine-component group formation in group technology: review and extension. In: *International Journal of Production Research* 20 (1982), S. 117–133

[Las70] Lasdon, L.S.: *Optimization theory for large systems*. New York : Macmillan, 1970

[VW10] Vanderbeck, F. ; Wolsey, L. A.: Reformulation and Decomposition of Integer Programs. In: Jünger, M. (Hrsg.) ; Liebling, T. M. (Hrsg.) ; Naddef, D. (Hrsg.) ; Nemhauser, G. L. (Hrsg.) ; Pulleyblank, W. R. (Hrsg.) ; Reinelt, G. G.and R. G.and Rinaldi (Hrsg.) ; Wolsey, L. A. (Hrsg.): *50 Years of Integer Programming 1958-2008*. Springer Berlin Heidelberg, 2010. – ISBN 978–3–540–68279–0, S. 431–502

Eidesstattliche Versicherung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften anderer entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, 14. Februar 2012

Mathias Luers