Julius-Maximilians-Universität Würzburg
Institut für Informatik
Lehrstuhl für Informatik I
Effiziente Algorithmen und wissensbasierte Systeme

Bachelorarbeit

Optimieren von Schnittplänen

Adrian Christoph Loy

Eingereicht am 23.07.2015

Betreuer:
Prof. Dr. Alexander Wolff
Fabian Lipp, M. Sc.

Zusammenfassung

Obwohl für das Cutting Stock Problem im Speziellen, sowie Packungs- und Zuschnittprobleme im Allgemeinen viel Forschungsaufwand betrieben wurde, ist für den eigentlichen Schneidevorgang, also das Trennen der Teile voneinander nachdem diese zugeordnet wurden, bisher kaum Literatur vorhanden. Dennoch besteht auch hier Optimierungspotenzial, da die Anzahl der Schnitte nicht konstant ist, sondern von der Reihenfolge, in dem die Schnitte durchgeführt werden, abhängig sein kann. Ziel dieser Arbeit ist das automatische Erstellen von möglichst effizienten Guillotinen-Schnittplänen für Lösungen des 2-dimensionalen Cutting Stock Problems. Nachdem gezeigt wird, dass es sich dabei um ein NP vollständiges Problem handelt, wird mithilfe eines ganzzahligen linearen Programms eine Möglichkeit angegeben, eine optimale Lösung zu finden. Außerdem stellen wir eine Heuristik vor, die mithilfe von größtmöglichen Matchings eine in der Praxis einsetzbare Lösung liefert, die in den meisten Fällen Schnitte gegenüber der trivialen Lösung einspart.

Inhaltsverzeichnis

1 Einführung

Unter dem Begriff Packungs- und Zuschneideprobleme (engl. *cutting and packing problems, C&P*) wird eine große Menge an geometrischen Problemen zusammengefasst, unter anderem das Rucksackproblem, das Behälterproblem, Bin Packing sowie viele weitere [Dyc90]. Diese Probleme wurden von Wäscher et al. [WHS07] kategorisiert und weisen folgende Gemeinsamkeiten auf: Gegeben sind zwei Mengen von Elementen, nämlich

- eine Menge von großen Objekten (Eingabe, Vorrat)
- eine Menge von kleinen Objekten (Ausgabe, Nachfrage)

Die Objekte sind dabei alle aus der gleichen geometrischen Dimension. Das Problem besteht nun darin, die kleinen Objekte so innerhalb der großen Objekte zu positionieren, dass

- alle kleinen Objekte innerhalb der Grenzen des entsprechenden großen Objektes liegen
- sich die kleinen Objekte nicht überlappen
- eine gegebene Zielfunktion optimiert wird.

Packungs- und Zuschneideprobleme haben eine große Praxisrelevanz für die Industrie. Beispiele dafür sind das Beladen von Containern, das Zuschneiden von Glasplatten, Metallblechen, Papierbögen und anderen Materialien oder auch das Scheduling von Jobs auf Maschinen. Wegen der vielen Anwendungsmöglichkeiten ist diese Familie an Problemen Gegenstand intensiver Forschung: Die „EURO Special Interest Group on Cutting and Packing“ listet in ihrer Bibliografie über 900 relevante wissenschaftliche Arbeiten auf (siehe [oCaP12]). In Folge dessen gibt es bereits gute Approximationen und exakte Algorithmen, die das Spektrum der Packungs- und Zuschneideprobleme bereits recht gut abdecken.

Auch in der Druckindustrie taucht ein Problem aus dieser Kategorie auf: Druckaufträge unterschiedlicher Größen sollen möglichst platzsparend auf Druckbögen von standardisierten Abmaßen platziert werden. Der Topologie von Wäscher et al. [WHS07] folgend klassifizieren wir dieses C&P-Problem als *Two-Dimensional Cutting Stock Problem*. Nach dem Druck werden die Druckaufträge des Bogens durch eine Vielzahl von Schnitten an einer Schneidemaschine voneinander getrennt. Obwohl zahlreiche Literatur für das Cutting Stock Problem, in diesem Fall also dem Erstellen der Layouts dieser Druckbögen, gibt, wurde dem damit eng verbundenen Schneideprozess bisher wenig Aufmerksamkeit zuteil.

Die dafür eingesetzten Schneidemaschinen bestehen aus einer breiten Öffnung, in die die zu schneidenden Bögen hinein geschoben werden. In der Öffnung befindet sich eine lange Klinge – auch Guillotine genannt – die auf Knopfdruck abgesenkt wird und somit darunterliegende Bögen in zwei Teile zerschneidet. Damit der Schnitt exakt ausgeführt wird, muss der Bogen mit einer Seite an der hinteren Wand der Öffnung anliegen. Die Klinge ist auf einer Schiene montiert und kann von der Maschine parallel zu dieser Wand automatisch verschoben werden. Um einen Bogen zu zerschneiden, wird zunächst an einem Interface manuell ein Schnittplan erstellt. Dieser Vorgang ist bisher nicht automatisiert und muss für jedes Layout neu durchgeführt werden.

Ein Schnittplan legt die Reihenfolge der Schnitte und die jeweils anliegenden Seiten der zu schneidenden Teile fest. Dadurch ist für jeden Schnitt auch die Position der Guillotine bestimmt. Vom Menschen müssen nur noch die richtigen Teile der Reihe nach in die Öffnung geschoben werden, die Guillotine positioniert sich automatisch. Nach dem Schnitt müssen etwaige Papierreste entfernt und Teile, die erst später weiter zerschnitten werden, ausgelagert werden.

Der Zeitaufwand des gesamten Schneideprozesses ist nicht nur vom Layout des Bogens sondern auch von der Erfahrung des Menschens abhängt, der den Schnittplan erstellt. So müssen beispielsweise bei manchen Schnittplänen mehr Teile gedreht und verschoben werden als bei anderen. Falls die Guillotine für unterschiedliche Teile auf den gleichen Abstand eingestellt werden muss, kann es unter Umständen möglich sein beide Teile nebeneinander in die Maschine zu schieben und gleichzeitig zu schneiden. Es gibt also offensichtlich effiziente und weniger effiziente Schnittpläne.

Diese Arbeit widmet sich dem Thema Schnittpläne für Lösungen des Two-Dimensional Cutting Stock Problem zu erstellen, bei denen möglichst viele Schnitte durch gemeinsames Schneiden eingespart werden. Obwohl das Anordnen der Teile für einen Schnitt dadurch länger dauern kann, ist insgesamt eine Zeitersparnis zu erwarten, da weniger Schnitte durchgeführt werden müssen. Andere Ansätze den Prozess zu beschleunigen, wie eine minimale Anzahl an Drehungen oder dass die zu verschiebenden Elemente möglichst klein sein sollten, werden nicht weiter verfolgt.

In Kapitel 2 wird das Problem zunächst formal definiert und einige Begriffe eingeführt. Außerdem wird durch eine Reduktion von Vertex Cover bewiesen, dass es sich um ein NP-vollständiges Problem handelt. In Kapitel 3 stellen wir ein ganzzahliges lineares Programm vor, das eine optimale Lösung ermittelt. In Kapitel 4 wird eine Heuristik erläutert, die in der Praxis gute Ergebnisse liefert. Zuletzt fassen wir die gewonnenen Erkenntnisse zusammen und weisen auf mögliche Themen für zukünftige Forschung hin.

2 Problemstellung und Definitionen

2.1 Formalisierung des Problems

Ein reales Beispiel eines Druckbogens ist in Abbildung 2.1 gezeigt: als *Schnittelement* bezeichnen wir die beschrifteten Rechtecke. Das Problem besteht darin, den gegebenen Bogen mit möglichst wenigen Schnitten so zu zerschneiden, dass alle Schnittelemente freigestellt sind. Das bedeutet, dass keine Elemente mehr miteinander oder mit Papierresten verbunden sind.

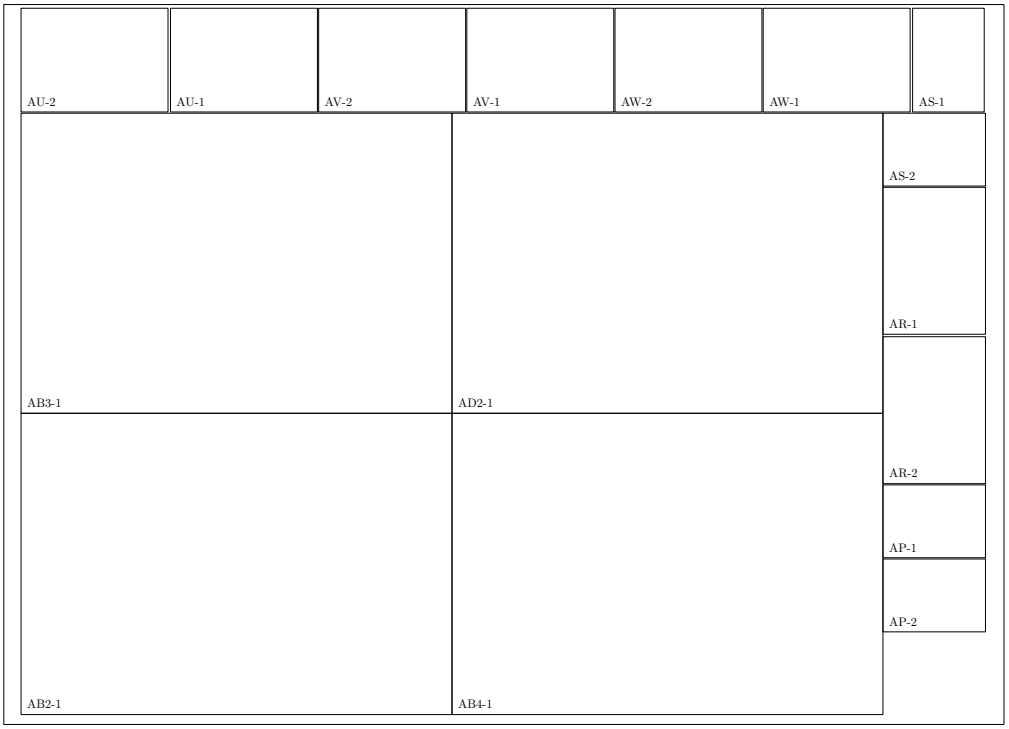


Abb. 2.1: Das Layout eines in der Praxis verwendeten Druckbogens mit beschrifteten Schnittelementen

Als *Schnittkante* bezeichnen wir Kanten von Elementen, an denen geschnitten werden muss. Solange es keine Schnittelemente gibt, die direkt an den Rand des Druckbogens angrenzen (da dadurch Probleme beim Druckvorgang entstehen, ist dies normalerweise gewährleistet), sind alle Kanten Schnittkanten, siehe Abbildung 2.2. Falls zwei Schnittkanten die gleiche Länge haben und *direkt*, also ohne Zwischenraum, nebeneinander

angeordnet sind (siehe Kanten c_7 und c_8 in Abbildung 2.2) werden sie zwangsläufig mit dem gleichen Schnitt geschnitten. Folglich werden wir Paare von solchen Schnittkanten wie eine einzige Schnittkante behandeln. Da jedes Schnittelement vier Kanten hat, ist die Anzahl der Schnittkanten linear in der Anzahl der Schnittelemente. Die Menge aller Schnittkanten bezeichnen wir mit C, die Mächtigkeit von C mit n.

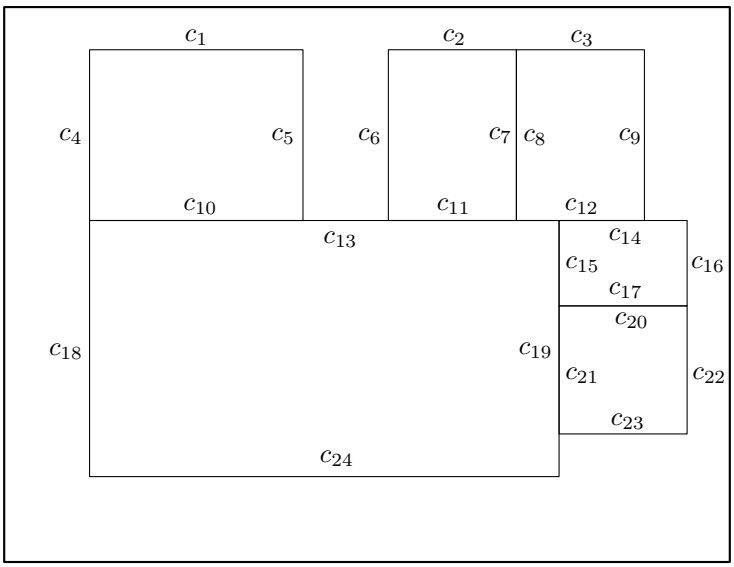


Abb. 2.2: Ein weiteres Beispiel für einen Druckbogen mit beschrifteten Schnittkanten.

Damit alle Schnittelemente freigestellt werden können, muss jede Schnittkante einem *Guillotinen-Schnitt* (siehe [CH95] und [AP10]) zugeordnet werden. Als Guillotinen-Schnitt bezeichnen wir alle Schnitte, die ein Rechteck in zwei kleinere Rechtecke zerschneiden. Guillotinen-Schnitte verlaufen folglich stets parallel zu den entsprechenden Rändern des Papierbogens und schneiden von einer Seite durchgehend bis zur anderen. Guillotinen-Schnitte zerteilen das Objekt stets in zwei *Blöcke*. Unterschiedliche Blöcke können gemeinsam geschnitten werden, falls der Schnitt bei beiden Blöcken im gleichen Abstand erfolgt (siehe Abbildung 2.3). So wird im Vergleich zu der Alternative, beide

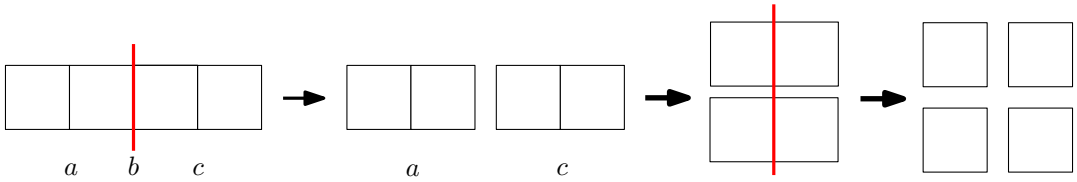


Abb. 2.3: Ein Minimalbeispiel für das Einsparen eines Schnittes durch Zusammenlegen. Alle Schnittelemente haben die Länge 1.

Blöcke einzeln zu schneiden, ein Schnitt eingespart. Dadurch besteht Potenzial zur Optimierung. Wir gehen davon aus, dass beliebig viele Schnittkanten gemeinsam geschnitten werden können. Dies ist in der Realität nur bedingt der Fall, da die Schneidemaschine

und das Guillotinen-Messer eine gewisse Breite haben und somit nicht beliebig viele Teile gleichzeitig unter das Messer passen.

Voraussetzung für das gemeinsame Schneiden ist, dass sich die Schnittkanten in unterschiedlichen Blöcken befinden und alle direkt mit dem gemeinsamen Abstand schneidbar sind. Mit direkt ist in diesem Zusammenhang gemeint, dass es nicht zulässig ist, den erforderlichen Abstand durch hintereinander legen verschiedener Schnittblöcke zu erreichen.

Für jede Schnittkante $c \in C$ definieren wir die *Distanzmenge* M_c, welche alle Abstände beinhaltet, die für das Schneiden dieser Schnittkante in Frage kommen. Die Distanzmengen für den Bogen aus Abbildung 2.3 sind $M_a = M_c = \{1, 2, 3\}$ und $M_b = \{1, 2\}$. Wir führen außerdem den Begriff der *Abhängigkeit* ein: Zwei Schnittkanten a, b sind dann voneinander abhängig, wenn in der Distanzmenge M_a Abstände vorkommen, mit denen a nur dann geschnitten werden kann, wenn sich b in der Reihenfolge in einer bestimmten Position relativ zu a befindet. Beispielsweise kann im Beispiel 2.3 der Schnitt a nur dann im Abstand 2 geschnitten werden, wenn Kante b nach Kante a geschnitten wird. Folglich sind diese beiden Kanten voneinander abhängig. In Abbildung 2.2 sind die Elemente der Mengen $\{c_4, c_5, c_6, c_7, c_8, c_9\}$ und $\{c_{14}, c_{17}, c_{20}, c_{23}\}$ jeweils paarweise abhängig voneinander. Jedes Element der einen Menge ist jedoch zu jedem Element der anderen Menge unabhängig, da sie sich in unterschiedlichen „Streifen“ befinden. Wir nennen zwei Schnittkanten *benachbart*, wenn sie

- Kanten desselben Schnittelements sind (z. B. die Kanten c_4 und c_5 in Abb. 2.2) oder
- parallel zueinander sind und keine andere Kante dazwischen liegt (z.B. die Kanten (c_5 und c_6 in Abb. 2.2).

Als D bezeichnen wir die Menge aller im Bogen vorkommenden und für Schnitte relevanten Abstände. Praktisch betrachtet gilt, dass die Guillotine der Schneidemaschine niemals auf eine Distanz eingestellt werden muss, die nicht in D liegt. Die Menge D ist die Vereinigung aller Distanzmengen. Es gilt $|D| \in O(n^2)$, da es n viele Distanzmengen gibt und die Größe jeder Distanzmenge in $O(n)$ liegt, da nur Abstände von der Schnittkante zu anderen Schnittkanten oder dem äußeren Rand sinnvoll sein können.

Ein *Schnitt* $s = (C', d)$ besteht aus einer Menge von Schnittkanten $C' \subset C$ und einem Abstand d, welcher die Distanz des Schnittes zu einer parallelen Seite des Blocks angibt. Dieser Abstand gibt an, auf welche Distanz die Guillotine für den Schneidvorgang eingestellt werden muss. Eine geordnete Menge S von Schnitten ist eine gültige Lösung des Schnittreihenfolgenproblems, wenn sie folgenden Anforderungen genügt:

1. Alle Schnittkanten sind genau einem Schnitt zugeordnet: $\forall c \in C \exists! (C', d) \in S : c \in C'$
2. Keine Schnittkante wird mit einem Abstand geschnitten, der nicht Element von M_c ist: $\forall (C', d) \in S (\forall c \in C' : d \in M_c)$.
3. Alle Schnitte sind in der gegebenen Reihenfolge Guillotinen-Schnitte.

Das Optimierungsproblem der Schnittreihenfolge ist die Frage nach der Schnittreihenfolge mit der minimalen Anzahl an Schnitten. Eine Schnittreihenfolge besteht aus maximal n vielen Schnitten mit jeweils einem dazugehörigen Abstand. Eine Lösung ist folglich in polynomieller Größe der Eingabe und das Problem der Schnittreihenfolge liegt somit in NP.

2.2 Beweis der NP-Vollständigkeit

Das Problem ist NP-vollständig. Um dies zu beweisen, wird eine Polynomialzeitreduktion des NP-vollständigen Problems der Knotenüberdeckung [Sch01] (im englischen *Vertex Cover* bzw. VC) auf das Problem der Schnittreihenfolge angegeben. Das Problem Vertex Cover ist wie folgt definiert:

- *gegeben:* Ungerichteter Graph $G = (V, E)$
- *gesucht:* Minimale Teilmenge von Knoten $V' \subseteq V$, sodass jede Kante von G einen Knoten aus V' enthält

Wir zeigen die Gültigkeit von VC $\preceq_p$ SCHNITTREIHENFOLGE indem wir zunächst angeben, wie der Bogen – also die Eingabe eines Algorithmus für Schnittreihenfolge – aus einem Graphen – der Eingabe von VC – konstruiert wird. Anschließend erläutern wir einige Bedingungen, die für derartig konstruierte Bögen gelten. Schließlich wird gezeigt, dass aus einer optimalen Lösung der Schnittreihenfolge für solche Instanzen die Lösung für VC abgeleitet werden kann.

Sei $V = \{v_1, \ldots, v_n\}$ die Menge der Knoten und $E = \{e_1, \ldots, e_m\}$ die Kantenmenge von $G = \{V, E\}$. Jeder Knoten $v_i \in V$ wird durch den Abstand $d_i = (n + i)n^2$ repräsentiert. Diese Abstände sind so gewählt, dass ein Abstand d_i mit $i \leq n$ nie als Summe zweier anderer Abstände $d_j,\ d_k$ mit $j, k \leq n$ ausgedrückt werden kann. Dies gilt offensichtlich, da $d_j + d_k \geq 2n^3 + 2n^2$ und $d_i \leq 2n^3$ für $1 \leq i, j, k \leq n$. Außerdem wird jede Kante $e_i \in E$ durch eine Schnittkante $c_i \in C$ repräsentiert. Diese Schnittkanten nennen wir *repräsentative Schnittkanten*. Sie sollen mit *genau* den Abständen schneidbar sein, welche die Knoten repräsentieren zu denen die Kante adjazent ist. Dafür konstruieren wir für jede repräsentative Schnittkante einen Teilstreifen wie in Abbildung 2.4 dargestellt. Die Anordnung ist so gewählt, dass die repräsentative Kante erst dann geschnitten werden kann, wenn der entsprechende Block (bestehend aus nur zwei Elementen) komplett freigestellt ist. Dadurch lassen sich die repräsentativen Schnittkanten mit nur zwei möglichen Distanzen schneiden. Diese sind so gewählt, dass sie die Knoten repräsentieren, welche durch diese Kante verbunden werden. Da die repräsentativen Schnittkanten *immer* mit diesen zwei Distanzen geschnitten werden können, sind sie unabhängig zu allen anderen Schnittkanten.

Der gesamte Bogen ergibt sich durch paralleles Anordnen der Teilstreifen, dargestellt in Abbildung 2.5. In derartig konstruierten Bögen kann eine repräsentative Schnittkante niemals mit einer nicht-repräsentativen Schnittkante zusammengelegt werden. Mehrere repräsentative Kanten können genau dann miteinander geschnitten werden, wenn der-

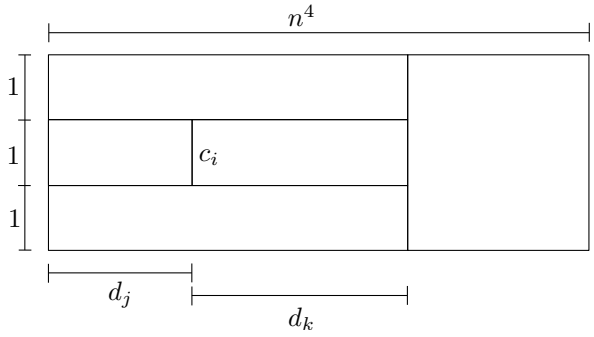


Abb. 2.4: Eine nicht maßstabsgetreue Darstellung des Teilstreifens zur Kante $e_i = \{v_j, v_k\}$. Die Teilstreifen enthalten stets eine repräsentative Kante und sind so aufgebaut, dass diese nur im Abstand d_j oder d_k geschnitten werden kann. Dies bedeutet, dass im Graphen der Knotenüberdeckung die Kante e_i die Knoten d_j und d_k verbindet.

selbe Abstand in ihren Distanzmengen vorkommt. Bezogen auf den Graphen bedeutet dies, dass mehrere Kanten (entsprechen den repräsentativen Schnittkanten) genau dann mit einem Knoten (entspricht einem Abstand in den Distanzmengen) abgedeckt werden können, wenn die Kanten alle zu diesem Knoten adjazent sind.

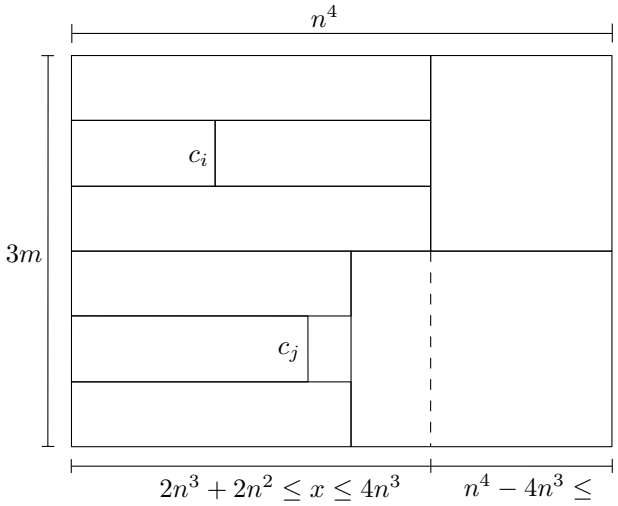


Abb. 2.5: Aufbau des aus dem Eingabegraphen konstruierten Bogens, in dem die repräsentativen Schnittkanten markiert sind. Außerdem sind einige Längen sowie Abschätzungen angegeben. In solchen Bögen können repräsentative Schnittkanten höchstens mit anderen repräsentativen Schnittkanten zusammengelegt werden. Die Darstellung ist nicht maßstabsgetreu.

Betrachtet man die vorhandenen Abstände, wird deutlich, warum repräsentative Schnittkanten nur mit anderen repräsentativen Schnittkanten zusammen geschnitten werden können: Ein Teilstreifen hat die Höhe 3. Da es m viele Teilstreifen gibt, haben alle vertikalen Abstände höchstens die Länge $3n^2$. Die Abstände, die Knoten repräsentieren und

mit denen repräsentative Kanten geschnitten werden können, werden durch die Vorschrift $(n+i)n^2$ gebildet und betragen somit mindestens n^3+n^2 und höchstens $2n^3$. Die Abstände, die durch aufsummieren von jeweils zwei dieser Abstände entstehen liegen folglich zwischen $2n^3+2n^2$ und $4n^3$. Da der Bogen insgesamt die Breite n^4 hat, haben die Schnittelemente am rechten Rand mindestens die Länge $\Theta(n^4)-4n^3$. Alle Abstände in Distanzmengen von nicht-repräsentativen Schnittkanten sind also alle entweder wesentlich größer oder kleiner als die repräsentativen Distanzen.

Sei S die optimale Schnittreihenfolge für einen so konstruierten Bogen und $S' \subset S$ die Teilmenge, die alle Schnitte mit repräsentativen Kanten beinhaltet. Da gezeigt wurde, dass alle Schnitte in S' Abstände verwenden, mit denen nicht-repräsentativen Kanten nicht geschnitten werden können, ist die Entscheidung, wie die repräsentativen Schnittkanten miteinander geschnitten werden völlig unabhängig davon, wie die restlichen Schnittkanten geschnitten werden: Egal wie die repräsentativen Schnittkanten zusammengelegt werden, die Schnitte der nicht-repräsentativen Schnittkanten müssen nicht angepasst werden. Dies liegt daran, dass die repräsentativen Schnittkanten unabhängig zu allen anderen Schnitten sind. Wenn dies nicht der Fall wäre, könnte durch Wahl eines anderen Abstandes für einen Schnitt mit repräsentativen Schnittkanten die Reihenfolge der Schnitte insgesamt geändert werden müssen, damit dieser neue Abstand zur Verfügung steht. Da dies jedoch nicht der Fall ist, hat die optimale Schnittreihenfolge auch minimal viele Schnitte für die repräsentativen Schnittkanten.

Die Distanzen in den Distanzmengen der (Graphkanten-)repräsentierenden Schnittkanten entsprechen den Knoten im Graphen, welche zu dieser Kante adjazent sind. Da eine Schnittreihenfolge jeder Schnittkante eine Distanz aus ihrer Distanzmenge zuordnet, gewinnen wir eine Knotenauswahl, die alle Kanten abdeckt, indem wir die Knoten zur Auswahl hinzufügen, deren korrespondierenden Distanzen als Abstände für Schnitte in der optimalen Schnittreihenfolge gewählt wurden. Da diese Schnittreihenfolge minimal viele Schnitte ausgewählt hat, ist auch unsere Knotenauswahl minimal. Folglich ist dies eine Reduktion von VC auf das Problem der Schnittreihenfolge. Alle Distanzen in Bögen, welche nach der beschriebenen Vorgabe konstruiert werden, liegen in $O(n^4)$ und enthalten $5|E|$ viele Schnittelemente. Daraus folgt, dass dies auch eine polynomielle Reduktion ist. Das Problem der Schnittreihenfolge ist also NP-schwer. Da das Problem, wie in Abschnitt 2.1 gezeigt wurde, in NP liegt ist es folglich auch NP-vollständig.

3 Formulierung als ganzzahliges lineares Programm

Selbst komplexe geometrische Probleme lassen sich, wie auch viele andere NP-vollständige Probleme, oft als *ganzzahliges lineares Programm* (ILP) modellieren. Obwohl das Problem ein ILP exakt zu lösen ebenfalls NP-vollständig ist [Pap81] und somit – unter der Annahme $NP \neq P$ – nicht garantiert in polynomieller Zeit lösbar ist, gibt es Lösungsverfahren, die für viele Fälle praktisch nutzbar sind. Diese Eigenschaft macht die Formulierung als ILPs zum naheliegenden und oft verwendeter Ansatz, um mit NP-vollständigen Problemen umzugehen. Auch das hier behandelte Problem der Schnittreihenfolgen lässt sich durch das in diesem Kapitel vorgestellte Programm exakt lösen. Wir betrachten zunächst nur bereits freigestellte Streifen wie in Abbildung 3.1 dargestellt, bevor das Modell auf allgemeine Bögen erweitert wird.

3.1 ILP für freigestellte Streifen

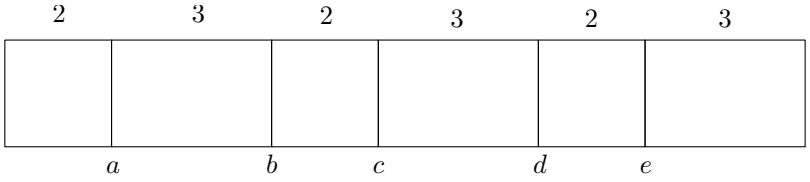


Abb. 3.1: Ein einfaches Beispiel für einen Druckbogen, der aus sechs Elementen und fünf Schnittkanten besteht und sich mit minimal drei Schnitten schneiden lässt. Es gilt $D = \{2, 3, 5, 7, 8, 10, 12, 13\}$

Die Lösung des ILP besteht aus einer Reihenfolge der Schnittkanten und der Information, welche Schnittkanten zusammen geschnitten werden können. Eine gültige Schnittreihenfolge, die der Lösung des ILP entspricht (dafür fehlt nur noch die Information, in welchen Abständen die Schnitte durchgeführt werden sollen), kann in $O(n)$ gefunden werden. Die Lösung des ILP soll natürlich der optimalen Lösung des Schnittreihenfolgenproblems entsprechen. Es soll also die Reihenfolge ermitteln, bei der die Anzahl der eingesparten Schnitte maximal ist. Bevor das ILP exakt angegeben wird, soll zunächst die logische Bedeutung der verwendeten Variablen erläutert werden. Wir betrachten in diesem Kapitel den Wert 1 als „wahr“ und 0 als „falsch“.

Die Reihenfolge stellen wir mithilfe einiger binärer Indikatorvariablen dar: Für $(i \neq j)$ bedeutet $x_{i,j} = 1$, dass die Kante i vor j geschnitten wird und somit in der Reihenfolge

vor j steht. Falls die Kanten i und j zusammen geschnitten werden, ist sowohl $x_{i,j} = 0$ als auch $x_{j,i} = 0$.

Um Informationen über die Anzahl der Schnitte zu verwalten, führen wir weitere binäre Indikatorvariablen ein: Für $i < j$ ist $y_{i,j}$ genau dann 1, wenn die Kanten i und j im Bezug auf eine explizite Reihenfolge zusammen geschnitten werden. Der Bezug auf eine Reihenfolge bedeutet, dass diese Variablen nicht *allgemein* angeben, ob zwei Kanten theoretisch zusammengelegt werden können. Betrachten wir beispielsweise den in Abbildung 3.1 dargestellten Bogen: In der trivialen Schnittreihenfolge a, b, c, d, e lassen sich keine Schnitte zusammenlegen, da es zu jedem Zeitpunkt nur einen Block gibt, der weiter zerteilt werden kann. Somit sind bei dieser Lösung alle Variablen vom Typ $y_{i,j}$ gleich 0. Schneidet man jedoch zunächst an der Kante c, lassen sich anschließend jeweils die Schnitte b,d und a,e zusammenfassen. Für diese Reihenfolge gilt folglich $y_{b,d} = y_{a,e} = 1$.

Die Zielfunktion Die zu maximierende Zielfunktion des ILP soll die Anzahl der eingesparten Schnitte berechnen. Da die Summe aller y Variablen diesem Wert nur dann entspricht, wenn niemals mehr als zwei Schnittkanten gemeinsam geschnitten werden, führen wir für die Formulierung der Zielfunktion noch weitere Indikatorvariablen z_i ein. Das ganzzahlige lineare Programm lässt sich nun wie folgt formulieren:

$$\max \sum_{i=2}^{n} z_i \tag{3.1}$$

Unter den Nebenbedingungen:

$$z_j \in \{0, 1\} \text{ für alle } j = 2, \ldots, n \tag{3.2}$$

$$z_j \leq \sum_{i=1}^{j-1} y_{i,j} \text{ für alle } j = 2, \ldots, n \tag{3.3}$$

Dank Bedingung 3.3 berechnet die Zielfunktion exakt die Anzahl der gesparten Schnitte. Um dies zu verdeutlichen, bezeichnen wir für jeden Schnitt s die Anzahl der gemeinsam geschnittenen Schnittkanten mit k_s. Da jeweils nur ein Schnitt statt k_s vielen benötigt wird um diese Kanten zu schneiden, werden pro Schnitt $k_s - 1$ Schnitte eingespart. Wir sagen, dass sich eine Variable vom Typ $y_{i,j}$ auf den Schnitt s *bezieht* wenn gilt: c_i und c_j sind in der Menge der Schnittkanten von s. Variablen vom Typ z_j beziehen sich auf den Schnitt s, wenn gilt: Die Kante c_j liegt in der Menge der Schnittkanten von s. Es haben genau die Variablen vom Typ y den Wert 1, die einem Schnitt zugeordnet sind. Jede Variable vom Typ z hat zu genau einem Schnitt Bezug. Außerdem beziehen sich auf jeden Schnitt s (außer dem, der Kante c_1 schneidet) genau k_s viele Variablen vom Typ z. Durch die Bedingungen des Typs 3.3 können pro Schnitt s alle Variablen des Typs z_j mit Bezug zu s den Wert 1 annehmen, außer derjenigen, die von diesen den

geringsten Index k hat. Der Grund dafür ist folgender: Da die Variablen des Typs $y_{i,j}$ ja nur für $i < j$ definiert sind, gibt es unter den Variablen des Typs y mit Bezug zu s keine Variable der Form $y_{i,k}$. Also ist die Summe über die Variablen vom Typ z mit Bezug zu s gleich k_s und die Summe über alle Variablen vom Typ z dementsprechend gleich der Summe der insgesamt eingesparten Schnitte.

Nebenbedingungen für die Reihenfolge und das Zusammenlegen Damit eine gültige Reihenfolge eingehalten wird und Schnitte korrekt zusammengelegt werden, benötigen wir einige weitere Nebenbedingungen.

$$y_{i,j} \in \{0,1\} \text{ für alle } i = 1,\ldots,n-1 \text{ und } j = i+1,\ldots,n \tag{3.4}$$

$$y_{i,j} + y_{j,k} - 1 \leq y_{i,k},\ y_{i,j} + y_{i,k} - 1 \leq y_{j,k},\ y_{i,k} + y_{j,k} - 1 \leq y_{i,j} \tag{3.5}$$
$$\text{für alle } i = 1,\ldots,n-1 \text{ und } j = i+1,\ldots,n \text{ und } k = j+1,\ldots,n$$

Die Triplets an Ungleichungen in Bedingungen vom Typ 3.5 spiegeln folgende logische Verknüpfung wieder: $((y_{i,j} \wedge y_{j,k}) \Rightarrow y_{i,k}) \wedge ((y_{i,j} \wedge y_{i,k}) \Rightarrow y_{j,k})) \wedge ((y_{i,k} \wedge y_{j,k}) \Rightarrow y_{i,j}))$. Dieser Ausdruck ist genau dann nicht erfüllt, wenn genau eine Variable 0 ist. Dadurch wird sichergestellt, dass Kante a mit Kante b und Kante a mit Kante c nur dann gemeinsam geschnitten werden dürfen, wenn auch Kante b mit Kante c geschnitten wird.

$$x_{i,j} \in \{0,1\} \text{ für alle } i,j = 1,\ldots,n \text{ mit } i \neq j \tag{3.6}$$

$$x_{i,j} \geq x_{i,k} + x_{k,j} - 1 \text{ für alle } i,j,k = 1,\ldots,n \text{ mit } i \neq j\ i \neq k\ j \neq k \tag{3.7}$$

Diese Bedingungen sorgen für Transitivität innerhalb der Variablen vom Typ x.

$$x_{i,j} + x_{j,i} + y_{i,j} = 1 \text{ für alle } i = 1,\ldots,n-1 \text{ und } j = i+1,\ldots,n \tag{3.8}$$

Falls die Schnittkanten i und j nicht miteinander geschnitten werden – und $y_{i,j}$ folglich auf 0 gesetzt ist – sorgen die Bedingungen vom Typ 3.8 für Antisymmetrie bei den Variablen vom Typ x. Zusammen mit der Transitivität folgt, dass diese Variablen eine Reihenfolge der Schnittkanten definieren. Falls zwei Kanten i und j zusammen geschnitten werden, sorgen die Bedingungen des Typs 3.8 dafür, dass die beiden Variablen $x_{i,j}$ und $x_{j,i}$ auf 0 gesetzt werden. Das bedeutet, dass weder Kante i vor Kante j, noch Kante j vor Kante i geschnitten wird. Da die beiden gemeinsam, sprich gleichzeitig, geschnitten werden, ist diese Eigenschaft notwendig.

$$y_{i,j} - 1 \leq x_{k,j} - x_{k,i},\ y_{i,j} - 1 \leq x_{k,i} - x_{k,j} \tag{3.9}$$
$$\text{für alle } i = 1,\ldots,n-1 \text{ und } j = i+1,\ldots,n \text{ und } k = 1,\ldots,n$$

Die Bedingungen vom Typ 3.9 spiegeln den Ausdruck $y_{i,j} \Rightarrow (x_{k,i} \Leftrightarrow x_{k,j})$ wieder. Dieser drückt aus, dass alle anderen Schnittkanten entweder vor den zusammengelegten Schnitten oder nach diesen liegen müssen. Folglich gilt, dass gemeinsam geschnittene Kanten in der durch die Variablen vom Typ x definierten Reihenfolge direkt benachbart sind. Da die Antisymmetrie 3.8 nicht für gemeinsam geschnittene Schnittkanten gilt, ist zwischen diesen Elementen keine Reihenfolge definiert; durch Bedingung 3.8 wird für diesen Fall sowohl $x_{i,j}$ als auch das Komplement $x_{j,i}$ auf 0 gezwungen. Da sie jedoch wegen Bed. 3.9 direkt benachbart sind, ist eine eindeutige Reihenfolge definiert, wenn zusammen geschnittene Schnittkanten jeweils als ein einziges Element in dieser Reihenfolge aufgefasst werden. Wir notieren diese Reihenfolge so, dass die zusammengefassten Elemente durch Einklammerung markiert werden. Die Notation $c, \{b, d\}, \{a, e\}$ als Ergebnis des ILP für den Bogen aus Abbildung 3.1 entspricht also einer (in diesem Fall optimalen) Schnittreihenfolge bestehend aus drei Schnitten: Zuerst an Kante c, dann gemeinsam an den Kanten b und d, dann gemeinsam an den Kanten a und e.

Modellierung der Distanzen Mehrere Schnittkanten können nur dann gemeinsam geschnitten werden, wenn sie auch tatsächlich mit dem gleichen Abstand geschnitten werden können. Wir benötigen also noch eine weitere Familie von Variablen, um die Information der Distanzmengen zu modellieren. Für diesen Zweck führen wir für jeden Abstand $d \in D$ und jede Schnittkante $c_i \in C$ eine weitere binäre Variable $u_{i,d}$ ein. Diese hat genau dann den Wert 1, wenn die Kante c_i im Abstand d geschnitten wird. Da eine Kante nur mit Abständen aus ihrer Distanzmenge geschnitten werden kann, können für eine Kante c_i grundsätzlich nur die Variablen vom Typ $u_{i,d}$ den Wert 1 annehmen, für die $d \in M_{c_i}$ gilt.

$$u_{i,d} \in \{0, 1\} \text{ für alle } i = 1, \ldots, n \text{ und } d \in D \tag{3.10}$$

$$\sum_{d \in D} u_{i,d} = 1 \text{ für alle } i = 1, \ldots, n \tag{3.11}$$

In einer Schnittreihenfolge wird eine Kante mit genau einem Abstand geschnitten, dies wird für jede Kante durch eine Bedingung des Typs 3.11 garantiert. Um sicherzustellen, dass nur Kanten zusammengelegt werden können, wenn sie zusammen mit dem gleichen Abstand geschnitten werden benötigen wir noch eine weitere Gruppe von binären Variablen. Wir führen für jeden Abstand $d \in D$ und jedes Paar von Schnittkanten $c_i, c_j \in C$ mit $i < j$ eine weitere binäre Variable $w_{i,j,d}$ ein. Diese hat genau dann den Wert 1, wenn die Kanten c_i und c_j im Abstand d miteinander geschnitten werden.

$$w_{i,j,d} \in \{0, 1\} \text{ für alle } i = 1, \ldots, n-1 \text{ und } j = i+1, \ldots, n \text{ und } d \in D \tag{3.12}$$

$$2w_{i,j,d} \leq u_{i,d} + u_{j,d} \text{ für alle } i = 1, \ldots, n \text{ und } j = i+1, \ldots, n \text{ und } d \in D \tag{3.13}$$

Die Bedingungen des Typs 3.13 ergeben sich aus der Tatsache, dass Kanten mit demselben Abstand geschnitten werden müssen, um gemeinsam geschnitten werden zu können.

Mit diesen Variablen können wir nun die Bedingungen formulieren, die dafür sorgen, dass alle gemeinsam geschnittenen Kanten auch tatsächlich mit demselben Abstand geschnitten werden:

$$y_{i,j} \leq \sum_{d \in D} w_{i,j,d} \text{ für alle } i = 1, \ldots, n-1 \text{ und } j = i+1, n \tag{3.14}$$

Eine Variable vom Typ $y_{i,j}$ kann also nur dann den Wert 1 annehmen, wenn mindestens eine der entsprechenden Variablen von Typ $w_{i,j,d}$ den Wert 1 hat. Kanten können also nur dann zusammengelegt werden, wenn sie mit demselben Abstand geschnitten werden. Eine Kante c_i kann mit nur einem Abstand geschnitten (Bed. n 3.11) werden. Dieser muss gleich allen anderen Abständen sein, mit denen die Schnittkanten, welche mit c_i zusammengelegt werden, geschnitten werden, da es für jedes Paar dieser Schnittkanten eine Variable des Typs y und somit eine Bedingung des Typs 3.14 gibt.

Spezifische Nebenbedingungen Kanten, die nicht zu allen anderen Kanten unabhängig sind, dürfen nicht mit einem beliebigen Abstand aus ihrer Distanzmenge geschnitten werden. Manche dieser Abstände sind dann nur in bestimmten Schnittreihenfolgen eine gültige Wahl. Beispielsweise kann die Kante a im Bogen aus Abbildung 3.1 nur dann mit der Distanz 3 geschnitten werden, wenn zuvor bereits an b geschnitten wurde. Mit Abstand 5 kann die Kante jedoch nur geschnitten werden, falls an der Kante b noch *nicht* geschnitten wurde, dafür jedoch bereits an Kante c. In unserem Modell würde das bedeuten, dass $u_{a,2}$ nur dann den Wert 1 annehmen kann, wenn $x_{b,a}$ ebenfalls auf 1 gesetzt ist. Entsprechend muss $u_{a,5} = 1$ auch $x_{a,b} = x_{c,a} = 1$ implizieren. Bedingungen dieser Art erzeugt der Algorithmus 1.

Auch für das Zusammenlegen von Schnitten sind solche spezifischen Nebenbedingungen, die von der Eingabe abhängen und nicht allgemein formuliert werden können, notwendig: So können beispielsweise benachbarte Schnittkanten niemals gemeinsam geschnitten werden. Auch können nur Schnittkanten, die zum Zeitpunkt der Durchführung des Schnittes in unterschiedlichen Blöcken sind, zusammen geschnitten werden. Soll also die Kante a mit Kante e im Bogen aus Abbildung 3.1 geschnitten werden, muss zuvor an einer der dazwischen liegenden Kanten geschnitten worden sein. In unser Modell übersetzt würde das bedeuten: Aus $y_{a,e} = 1$ muss $(x_{b,a} \vee x_{c,a} \vee x_{d,a})$ folgen (wegen den Bedingungen vom Typ 3.9 ist es egal, ob hier die Variablen vom Typ x bezüglich a oder bezüglich e stehen).

Algorithmus 2 zeigt, wie diese Bedingungen konstruiert werden. Für alle benachbarten Schnittkanten sowie Paare von Schnittkanten, die keine gemeinsamen Distanzen in ihren Distanzmengen haben, muss die entsprechende Variable vom Typ w für alle Abstände auf 0 gesetzt werden. Dies geschieht in Zeile 3. Aus den Bedingungen des Typ 3.14 folgt, dass dadurch auch die Variable des Typs y für diese Kanten den Wert 0 haben muss. Bei voneinander abhängigen Schnittkanten muss noch dafür gesorgt werden, dass sie nur dann gemeinsam geschnitten werden können, wenn die Kanten in der gewählten Reihenfolge zum Zeitpunkt des Schnittes in unterschiedlichen Schnittblöcken liegen. Betrachten wir

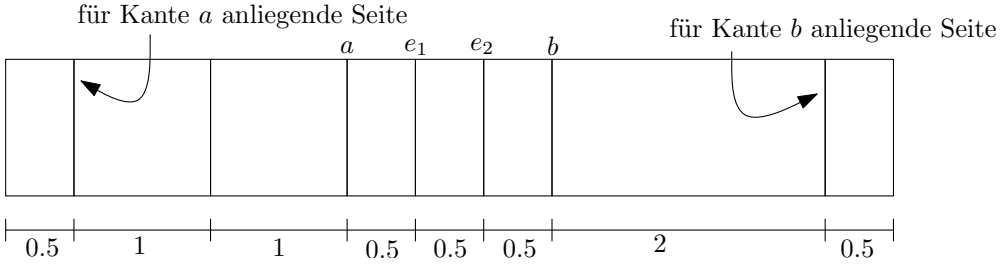


Abb. 3.2: Falls die Kante a gemeinsam mit Kante b im Abstand 2 geschnitten werden soll, muss zuvor an e_1 oder e_2 geschnitten worden sein. Des Weiteren müssen die beiden am Schneidegerät anliegenden Seiten zur Verfügung stehen und somit ebenfalls schon geschnitten sein.

dazu den in Abbildung 3.2 dargestellten Bogen. Gesucht sind die Bedingungen, die gelten müssen, um die Kanten a und b gemeinsam im Abstand 2 zu schneiden. Offensichtlich muss entweder Kante e_1 oder Kante e_2 vor den beiden Kanten geschnitten werden. Als Bedingung mithilfe der Variablen vom Typ x formuliert: $w_{a,b,2} \leq x_{e_1,a}+x_{e_2,a}+x_{e_1,b}+x_{e_2,b}$ (wegen den Bedingungen des Typs 3.9 könnte man bei dieser Bedingung die Variablen $x_{e_1,b}$ und $x_{e_2,b}$ auch weglassen). Allgemeiner gesagt, können zwei Kanten nur dann gemeinsam geschnitten werden, wenn eine Schnittkante dazwischen bereits geschnitten wurde. Wir definieren E als die Menge dieser Kanten. Wir erhalten die gewünschten Eigenschaften durch Bedingungen des Typs $w_{a,b,d} \leq \sum_{c \in E} x_{c,a} + x_{c,b}$ (siehe Zeile 8 in Algorithmus 2).

Algorithmus 1: BuildTypeUSpecificConstraints(Liste C)

```
   Eingabe : Liste von Schnittkanten C
1  foreach i in C do
2      for d ∈ D do
3          if d ∉ M_i ∩ M_j then
4              Neue Nebenbedingung: u_{i,d} = 0
5          else
               // Distanz für i muss zur Verfügung stehen:
6              C_danach = Menge aller Kanten zwischen i und der für i anliegenden Seite
7              Neue Nebenbedingung: |C_danach| u_{i,d} ≤ Σ_{c∈C_danach} x_{i,c}
8              c_davor = Kante an der für i anliegenden Seite
9              Neue Nebenbedingung: u_{i,d} ≤ x_{c_davor,i}
```

Korrektheit Dieses ILP liefert eine Schnittreihenfolge mit minimaler Anzahl von Schnitten. Durch die Variablen vom Typ x können alle möglichen Reihenfolgen an Schnittkanten repräsentiert werden. Da für jedes Paar von Schnittkanten eine Variable vom Typ y existiert, kann grundsätzlich jeder gemeinsame Schnitt bezüglich aller Schnittkante

Algorithmus 2: BuildTypeWSpecificConstraints(Liste C)

Eingabe : Liste von Schnittkanten C

foreach (i, j) in C mit $i < j$ **do**
  if i und j benachbart **then**
    Neue Nebenbedingung: $w_{i,j,d} = 0$ für alle $d \in D$
  else if i und j sind voneinander abhängig **then**
    for $d \in D$ **do**
      if $d \in M_i \cap M_j$ **then**
        `// Müssen in unterschiedlichen Blöcken liegen:`
        E = Menge aller zwischen i und j liegenden Kanten
        Neue Nebenbedingung: $w_{i,j,d} \leq \sum_{c \in E} x_{c,i} + x_{c,j}$

dargestellt werden. Außerdem kann für jeden Schnitt jede in Frage kommende Schnittdistanz durch entsprechende Variablen des Typs u dargestellt werden. Folglich gibt es für jede gültige Schnittreihenfolge eine entsprechende Variablenbelegung.

Aus der Lösung des ILP kann außerdem in linearem Zeitaufwand eine gültige Lösung gewonnen werden. Durch die Variablen des Typs y ist festgelegt, welche Kanten gemeinsam geschnitten werden. Alle Schnittkanten, für die keine Variable des Typs y in der Lösung den Wert 1 hat, werden einem eigenen Schnitt zugeordnet. Somit sind alle Schnittkanten exakt einem Schnitt zugeordnet. Die Reihenfolge dieser Schnitte ist durch die Belegung der Variablen vom Typ x festgelegt. Für jede Kante wird mithilfe der Variablen vom Typ u genau ein Abstand assoziiert. Unter gemeinsam geschnittenen Kanten ist dieser gleich. Somit ist für jeden Schnitt ein Abstand festgelegt. Da außerdem für Streifen, auf die wir die Eingabe für das ILP an dieser Stelle noch beschränkt haben, alle Schnitte an Kanten automatisch Guillotinen-Schnitte sind, ist die repräsentierte Schnittreihenfolge folglich garantiert eine gültige.

Da alle möglichen Schnittreihenfolgen dargestellt werden können, aus der Lösung stets eine gültige Schnittreihenfolge gewonnen werden kann und diese dank der Zielfunktion (siehe Abschnitt 3.1) diejenige ist, welche maximal viele Schnitt einspart, lässt sich mithilfe des ILP eine optimale Lösung für das Problem finden.

3.2 Erweiterung auf allgemeine Eingaben

Damit das ILP für alle lösbaren Instanzen eine optimale Schnittreihenfolge findet, müssen noch einige weitere Nebenbedingungen eingeführt werden. Zum einen müssen Kanten, welche ohne Lücke direkt aneinander liegen, stets gemeinsam geschnitten werden. Im Bogen aus Abbildung 3.3 sind $(c_4, c_6), (c_6, c_5)$ und (c_5, c_7) Beispiele für solche Kantenpaare. Dies erzwingen wir mit folgenden Nebenbedingungen:

$$y_{i,j} = 1 \text{ für alle } i, j \text{ wenn gilt: } c_i \text{ und } c_j \text{ liegen aneinander} \tag{3.15}$$

Wenn wir den Bogen aus Abbildung 3.3 betrachten, sieht man einige weitere Eigenschaften, die eine Schnittreihenfolge erfüllen muss um gültig zu sein. So können unter anderem die Kanten c_2, c_{10} und c_{14} nicht als erstes geschnitten werden, da dies keine Guillotinen-Schnitte wären oder andere Elemente zerstören werden würden. Dies liegt daran, dass diese Kanten *mittig senkrecht* auf anderen Kanten stehen und diese (und das Schnittelement dieser Kante) zerschneiden, wenn sie vor diesen geschnitten werden würden. Einer Kante c_i steht mittig senkrecht auf einer Kante c_j, wenn die Kanten folgende Eigenschaften erfüllen:

- Die Kanten stehen senkrecht auf einander.
- Ein Endpunkt der Kante c_i liegt auf einem Nicht-Endpunkt der Kante c_j.

Damit in diesem Fall kein Schnitt erzeugt wird, der die Kante c_j zerschneidet, führen wir folgende Nebenbedingungen ein:

$$x_{i,j} = 1 \text{ für alle } i,j \text{ für die gilt: } c_j \text{ ist mittig senkrecht auf } c_i \tag{3.16}$$

Außerdem führen wir den Begriff *nahezu mittig senkrecht* ein. Eine Kante c_i steht nahezu mittig senkrecht auf eine Kante c_j, wenn die Kanten folgende Eigenschaften erfüllen:

- Die Kanten stehen senkrecht aufeinander.
- Die Kanten schneiden sich nicht.
- Die Gerade, auf der die Kante c_i liegt, schneidet die Kante c_j an einem Nicht-Endpunkt p.
- Die Gerade, auf der die Kante c_i liegt, schneidet keine anderen zu c_j parallelen Kanten zwischen p und der Kante c_i an einem Nicht-Endpunkt.

Außerdem definieren wir die Kanten, die einen Endpunkt auf der Strecke von p zu c_i haben und sich auf der gleichen Seite von c_i befinden, wie das zu c_i gehörende Schnittelement, als Menge der *alternativen Kanten* $A_{i,j}$. Dies sind stets Schnittkanten des Teilstreifens, in dem auch die Kante c_i liegt. Abbildung 3.4 zeigt ein Minimalbeispiel: Hier stehen die Kanten c_1 und c_4 nahezu mittig senkrecht auf der Kante c_6. Die einzige alternative Kante ist c_4. Anschaulich betrachtet erhält man nahezu mittig senkrechte Kantenpaare, indem man bei mittig senkrechten Kantenpaaren eine Lücke zwischen den Schnittelementen einführt. In dieser Lücke können sich auch weitere Schnittelemente befinden. Diese besitzen dann Kanten aus der Menge der alternativen Schnittkanten. Die Definition ist notwendig, da es für solche Konstellationen zwei Möglichkeiten gibt, die senkrecht stehende Kante zu schneiden. So können die Kanten c_1 und c_2 aus Abbildung 3.4 geschnitten werden, sobald an der Kante c_4 oder der Kante c_6 geschnitten wurde. Wir führen also folgende Nebenbedingungen ein:

$$x_{j,i} + \sum_{k}^{A_{i,j}} x_{k,i} \geq 1 \tag{3.17}$$

für alle i,j für die gilt: c_i nahezu mittig senkrecht auf c_j

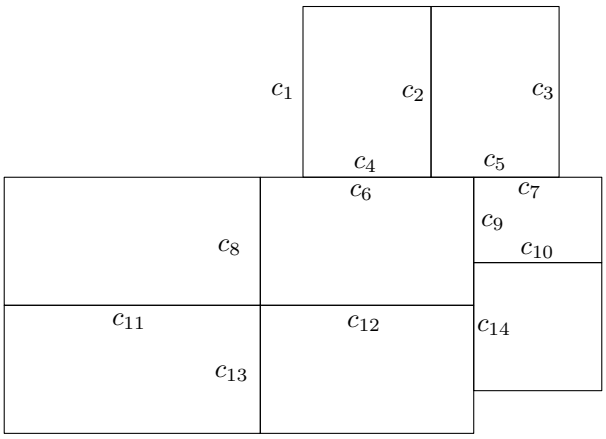


Abb. 3.3: Ein Beispiel für einen allgemeinen Bogen.

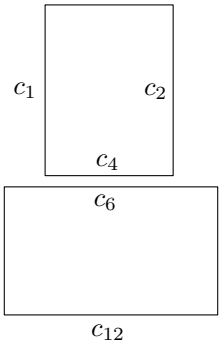


Abb. 3.4: Minimalbeispiel für nahezu mittig senkrechte Kanten. Die Kanten c_1 und c_2 stehen nahezu mittig senkrecht auf der Kante c_6.

Falls vier Kanten in einem Kreuz angeordnet sind, werden ebenfalls weitere Bedingungen gebraucht, um sicherzustellen, dass ausschließlich Guillotinen-Schnitte verwendet werden. Ein Beispiel für so eine Konstellation sind die Kanten c_8, c_{12}, c_{13} und c_{11} im Bogen aus Abbildung 3.3. Wie man sieht, gibt es für dieses Kreuz zunächst zwei Möglichkeiten: Entweder man schneidet zuerst c_8 und c_{13} zusammen, oder c_{11} und c_{12}. Für den zweiten Schnitt gibt es dann keine speziellen Bedingungen mehr. Es müssen also immer zwei gegenüberliegende Seiten gemeinsam geschnitten werden. Dies wird durch folgende Familie von Bedingungen erzwungen:

$$y_{i,j} + y_{k,l} \geq 1 \text{ für alle } i, j, k, l \text{ für die gilt: } c_i, c_j, c_k, c_l \text{ bilden ein Kreuz} \tag{3.18}$$

All diese Bedingungen für allgemeine Bögen müssen offensichtlich ebenfalls in der Vorverarbeitung eingefügt werden, da sie von der speziellen Struktur der Eingabe abhängig sind. Sie sorgen dafür, dass es sich auch bei allgemeinen Bögen bei allen Schnitten um Guillotinen-Schnitte handelt. Folglich liefert das ILP nach wie vor eine korrekte Lösung.

4 Heuristischer Algorithmus

Für den allgemeinen Fall ist es recht schwer Approximationsalgorithmen mit einer garantierten Güte zu finden, weil sich oft problematische Spezialfälle konstruieren lassen, für die keine eingesparten Schnitte gefunden werden. Allerdings weisen die in der Praxis vorkommenden Bögen alle gewisse Regelmäßigkeiten auf. Zudem wird auch beim Erstellen der Bögen versucht, möglichst unkomplizierte und mit wenigen Schnitten zerschneidbare Layouts zu generieren. Des Weiteren stehen nur bestimmte Formate (z. B. Visitenkarte, Flyer) für die Schnittelemente zur Auswahl. Die Breiten und Höhen der Elemente können also nicht völlig beliebig sein. In diesem Kapitel wird eine Heuristik vorgestellt, welche eine Greedy-Strategie verfolgt und für reale Bögen in 97% der Testfälle Schnitte einspart.

4.1 Funktionsweise des Algorithmus

Wir definieren einen gewichteten Multigraphen $G = (V, E, w)$, welcher den Bogen, dessen dazugehörige Menge an Schnittelementen wir mit L bezeichnen, repräsentiert. Die Knoten sind in diesem Fall die Schnittelemente, d. h. $V = L$. Wir verbinden zwei Knoten mit Kanten, wenn die entsprechenden Schnittelemente *überdeckungsfrei benachbart* sind. Zu einem Element A ist ein Element B benachbart, wenn es kein anderes Element gibt, das der gleichen Seite von A zugewandt ist und einen kleineren Abstand zu A hat (siehe Abbildung 4.1).

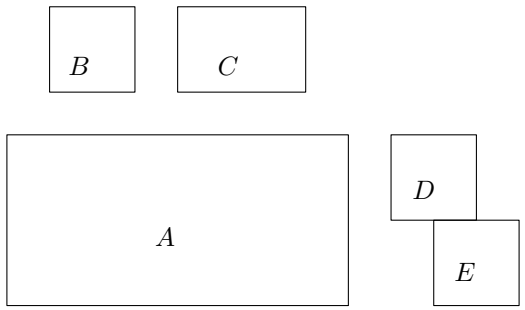


Abb. 4.1: Darstellung zur Veranschaulichung von Nachbarn. Zum Schnittelement A sind die Elemente B, C und D benachbart (aber nicht überdeckungsfrei benachbart).

Zwei Elemente sind überdeckungsfrei benachbart, wenn das kleinstmögliche Rechteck, welches beide Elemente überdeckt, kein anderes Element überdeckt (siehe Abb. 4.2). Elemente die nicht benachbart sind, können logischerweise auch nicht überdeckungsfrei

benachbart sein. Die Schnittkante zwischen zwei überdeckungsfreien Nachbarn kann,

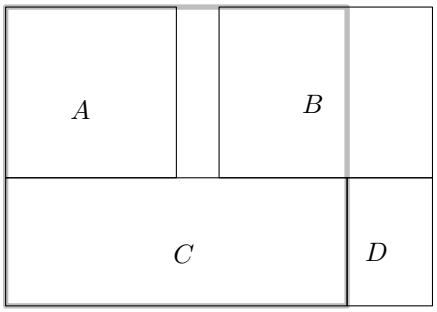


Abb. 4.2: Darstellung zur Veranschaulichung von überdeckungsfreien Nachbarn. Die Schnittelemente A und B, sowie C und D sind jeweils überdeckungsfrei benachbart. Alle anderen Paare sind nicht überdeckungsfrei benachbart, da das kleinste beide beinhaltende Rechteck (grau eingezeichnet für die Elemente A und C) sich stets mit anderen Elementen überlappt.

sobald der Block mit diesen zwei Elementen freigestellt ist, mit bis zu vier unterschiedlichen Distanzen geschnitten werden (siehe Abbildung 4.3). Für jede dieser möglichen

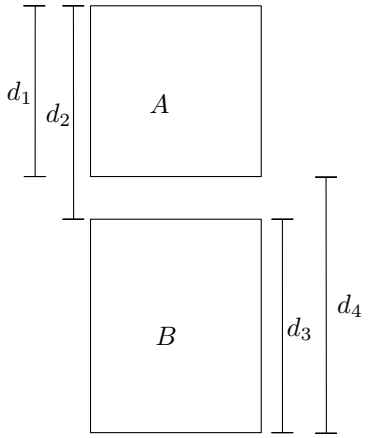


Abb. 4.3: Falls der Block mit A und B freigestellt ist, sind an diesem Block Schnitte mit den vier eingezeichneten Distanzen möglich.

Distanzen fügen wir im Graphen eine Kante mit der Distanz als Gewicht ein. Zwei Knoten in G können also mit bis zu vier Kanten verbunden sein. Ein Beispiel für einen Bogen und deb korrespondierenden Graphen ist in Abbildung 4.4 gezeigt.

Die zugrunde liegende Idee für den Algorithmus ist, dass die Schnittreihenfolge entgegen der tatsächlich Reihenfolge aufgebaut wird und pro Iteration der Schnitt ausgewählt wird, der möglichst viele Schnittkanten gemeinsam schneidet. So wird im Bogen aus Abbildung 4.4 als erstes ein Schnitt, der im Abstand 2 durchgeführt wird und die Kanten zwischen A und B, zwischen C und D und zwischen E und F (oder G und F) schneidet, zur Lösung hinzugefügt. In der konstruierten Lösung wird dieser Schnitt jedoch am Ende

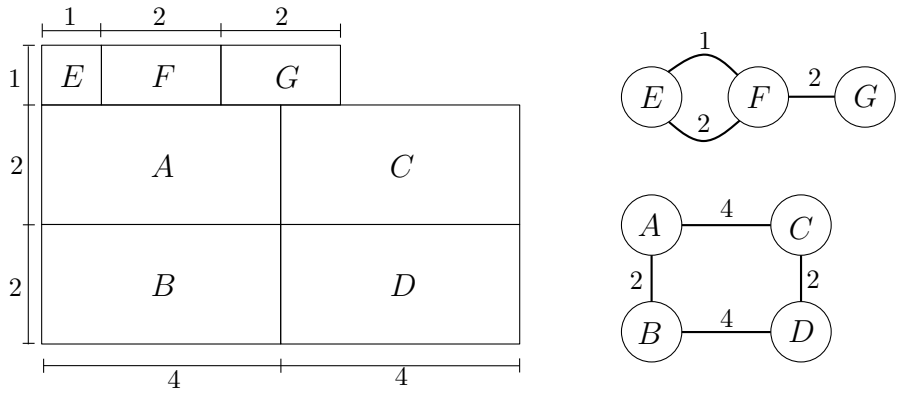


Abb. 4.4: Ein Bogen und der korrespondierende Graph, welcher Knoten für die Schnittelemente hat und Kanten, falls diese überdeckungsfrei benachbart sind. Das Gewicht repräsentiert die Distanz, mit denen der Block geschnitten werden kann.

der Reihenfolge stehen.

Dieser Bottom-up Ansatz bietet sich an, da so sehr früh Schnitte gefunden werden, welche viele Blöcke gemeinsam schneiden und so Schnitte einsparen. Die Problematik eines Top-down Ansatzes liegt in der Schwierigkeit die Schnitte am Anfang zu bewerten und unter ihnen eine Auswahl zu treffen: Meistens können mit diesen noch nicht mehrere Blöcke gemeinsam geschnitten werden. Allerdings kann eine falsche Wahl dies für spätere Schnitte verhindern. Die Größe des Suchbaumes verhindert jedoch, die Auswirkungen jeder Möglichkeit zu prüfen.

Das Verfahren in Pseudocode zeigt Algorithmus 3. Ein Beispiel für den iterativen Ablauf des Algorithmus ist in den Abbildungen 4.6a, 4.6b, 4.6c und 4.6d dargestellt.

Problematisch ist das Vorkommen von Schnittkanten, die mit Distanzen zusammen geschnitten werden können, die durch Summe von mehreren Längen entstehen. Denn diese Distanzen erscheinen erst dann im Graphen, wenn zuvor in der richtigen Reihenfolge verschmolzen wurde. Ein Beispiel, in dem kein Schnitt gespart wird obwohl dies möglich wäre zeigt Abbildung 4.5.

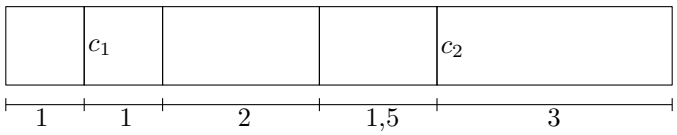


Abb. 4.5: Hier kann ein Schnitt gespart werden, da die Schnittkanten c_1 und c_2 gemeinsam mit Abstand 3 geschnitten werden können. In der ersten Iteration „sieht “ der Algorithmus jedoch keine mehrfach auftretenden Distanzen. Falls nun zunächst die beiden Elemente mit Länge 1 verschmolzen werden, wird die Kante c_1 bereits im Abstand 1 geschnitten. Folglich werden dann keine Schnitte eingespart.

Algorithmus 3: GreedyHeuristic(Liste L)

Eingabe : Liste von Schnittelementen L
Ausgabe : Liste von Schnitten S

```
1  S = {}
   // Setze pro Element die Nachbarn (nicht überdeckungsfreie Nachbarn)
2  L = SetNeighbours(L)
3  while |L| > 1 do
4      E = BuildGraphEdges(L)
5      g = MostFrequentWeight(E)
6      if E == ∅ then
7          SolveConflict(L)
8      for e ∈ E do
9          if e.Weight ≠ g then
10             E = E\{e}
11     E' = GetMaximumMatching(L, E)
12     (L, L') = MergeElements(L, E')
13     S = S ∪ GenerateCuts(L', E', g)
14 Füge zu S die Schnitte hinzu, die die Kanten des Elements in L freischneiden
15 return S
```

Algorithmus 4: BuildGraphEdges(Liste L)

Eingabe : Liste von Schnittelementen L
Ausgabe : Eine Menge von Kanten E, welche alle mengenüberdeckungsfreie Nachbarn mit gewichteten Kanten verbinden

```
1  E = ∅
2  for v ∈ L do
3      for u ∈ v.GetNeighbours do
4          cover = SmallestCoveringRectangle(u, v)
5          somethingOverlaps = false
6          for w ∈ v.neighbours ∪ u.neighbours do
7              if cover.overlaps(w) then
8                  somethingOverlaps = true
9          if somethingOverlaps = false then
10             E = E ∪ GetEdges(v, u)
11 return E
```

Algorithmus 5: MergeElements(Liste L, Liste E')

Eingabe : Liste von Schnittelementen L, Liste von Kanten E'

Ausgabe : Die Liste an Schnittelementen L, in der einige Elemente verschmolzen wurden. Eine Liste an Schnittelement L', welche nur die verschmolzenen Elemente enthält.

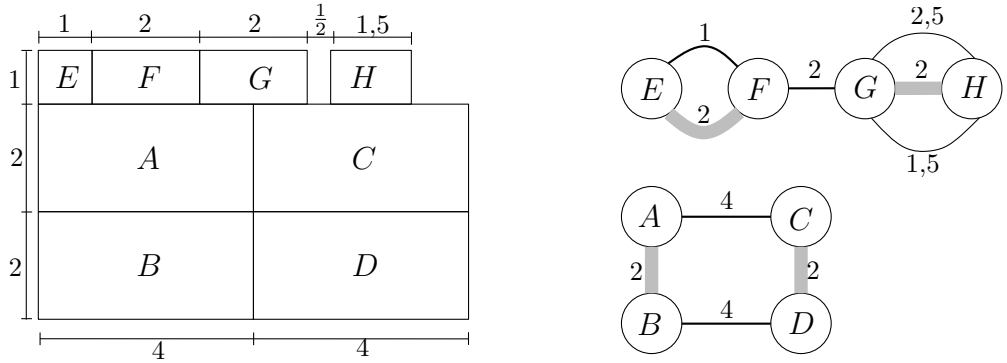
```
1 L' = ∅
2 for {u, v} ∈ E' do
3 |   L = L\{u, v}
4 |   v = SmallestCoveringRectangle(u, v)
5 |   v.CoveredNodes = {u, v}
6 |   v.Neighbours = u.Neighbours ∪ v.Neighbours
7 |   L = L ∪ {v} L' = L' ∪ {v}
8 return (L, L')
```

Algorithmus 6: GenerateCuts(Liste L, Liste E, Double g)

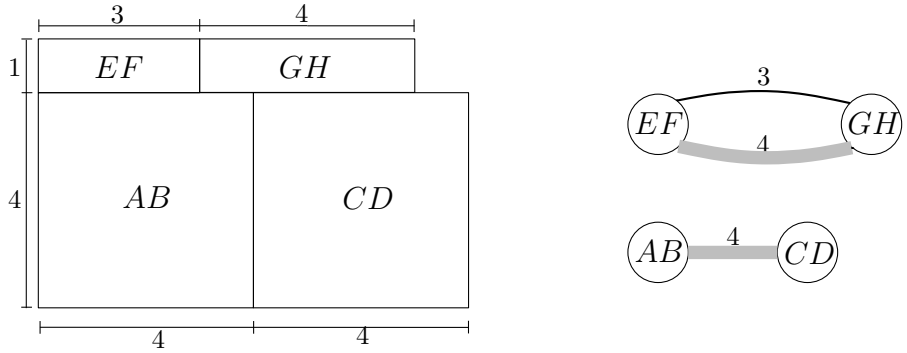
Eingabe : Liste von verschmolzenen Elementen L, Liste von Kanten E (ist Matching), Gewicht g der Kanten in E

Ausgabe : Eine Menge von Schnitten S, welche die Elemente in L jeweils so zerschneiden, dass die von ihnen überdeckten Elemente freigestellt sind. Dabei werden alle Kanten aus E gemeinsam geschnitten.

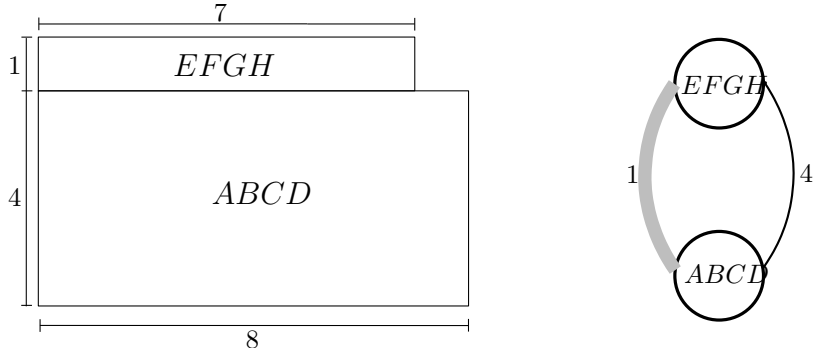
```
1  S = ∅
2  s = ∅
3  for e ∈ E do
4  |   s.C = s.C ∪ {e}
5  s.d = g
6  S = S ∪ {s}
7  for v ∈ L do
8  |   if Schnitt s stellt die Elemente v.CoveredNodes nicht komplett frei then
9  |   |   S' = Schnitte, welche die Reste an den zwei Elementen in v.CoveredNodes
   |   |   abtrennen
10 |   |   S = S ∪ S'
11 return S
```



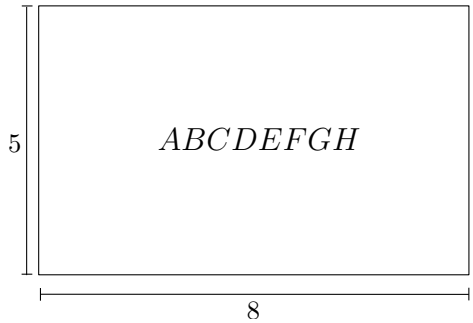
(a) Iteration 1: Füge die Schnitte $s_1 = \{\{c_{EF}, c_{GH}, c_{AB}, c_{CD}\}, 2\}$, $s_2 = \{\{c_H\}, 1.5\}$ vorne in der Schnittreihenfolge ein.



(b) Iteration 2: Füge Schnitt $s_3 = \{\{c_{EF,GH}, c_{AB,CD}\}, 4\}$ vorne in der Schnittreihenfolge ein.



(c) Iteration 3: Füge Schnitt $s_4 = \{\{c_{GH}\}, 4\}$ und $s_5 = \{\{c_{EFGH,ABCD}\}, 1\}$ vorne in der Schnittreihenfolge ein.



(d) Terminiere sobald der Graph keine Kanten mehr enthält. Die Schnittreihenfolge entspricht der umgekehrten Reihenfolge, in der sie hinzugefügt wurden (Last In First Out). Bei diesem Beispiel entspricht die gefundene einer optimalen Schnittreihenfolge.

GreedyHeuristic Algorithmus 3 speichert zunächst zu jedem Schnittelement die Menge der Nachbarn (Zeile 2). Anschließend wird pro Iteration der while Schleife zunächst der Graph konstruiert, indem den Knoten die Kantenmenge mithilfe von BuildGraphEdges (siehe 4) hinzugefügt wird. BuildGraphEdges setzt für jedes überdeckungsfreie Paar maximal vier Kanten. Anschließend wird das Kantengewicht ermittelt, das am häufigsten vorkommt. Alle Kanten mit anderen Gewichten werden entfernt. Unter diesen Schnittkanten wird nun der Schnitt gesucht, der möglichst viele Schnittkanten auf einmal schneidet. Da benachbarte Schnittkanten nicht miteinander schneidbar sind, entspricht die maximal mögliche Auswahl an miteinander schneidbaren Schnittkanten dem größtmöglichen Matching im Graphen. Beispielsweise können im Bogen aus Abbildung 4.4 die beiden Schnittkanten im oberen Streifen (E F und F G) nicht miteinander geschnitten werden. Diese Kanten entsprechen den zwei zu F adjazenten Kanten mit Gewicht 2 im Graphen. Offensichtlich kann nur eine von beiden Kanten für ein nicht erweiterbares Matching ausgewählt werden.

MergeElements Die Schnittkanten, die nun bereits einem Schnitt zugeordnet sind, müssen aus dem Graphen entfernt werden, damit sie nicht noch einmal ausgewählt werden können. Dies geschieht durch den Aufruf von MergeElements(L,E') (siehe Algorithmus 5). Die Elemente, für die in dieser Iteration eine Kante ausgewählt wurde, werden nun zu einem Element verschmolzen. Da die Elemente überdeckungsfreie Nachbarn sind, überlappen sich auch nach dem Verschmelzen keine Elemente. Das Verschmelzen entspricht letztlich dem Entfernen der beiden Elemente aus der Liste der Schnittelemente L und das Einfügen des kleinsten überdeckenden Rechtecks als neues Schnittelement. Dadurch sind alle bereits zugeordneten Kanten aus dem Bogen entfernt. Im „neuen“ Bogen werden dann iterativ weitere Elemente verschmolzen. Dies geschieht solange, bis es nur noch ein einziges Element gibt.

GenerateCuts GenerateCuts(L, E) liefert nun die Schnitte, die jeweils aus den verschmolzenen Elementen die „ursprünglichen“ Schnittelemente freischneiden. Dabei können Schnitte eingespart werden, da alle Kanten aus dem größtmöglichen Matching gemeinsam geschnitten werden. Manche Blöcke brauchen mehrere Schnitte, um die vorherigen Elemente (pro Element im Attribut *CoveredNodes* gespeichert) komplett freizustellen. Dies ist beispielsweise dann notwendig, wenn Schnittelemente nicht direkt aneinander liegen, wie die Elemente G und H im Beispiel 4.6a. Dann wird noch ein Schnitt für das Element gebraucht, an dem der Rest hängt.

Konflikte In einigen seltenen Fällen kann es vorkommen, dass der Algorithmus die Elemente genau so verschmilzt, dass der korrespondierende Graph keinerlei Kanten mehr hat. Falls solch ein Konflikt auftritt, muss dieser zunächst aufgelöst werden, um eine Endlosschleife zu verhindern. Ein Beispiel für einen Bogen, bei dem ein Konflikt entstehen kann, zeigt Abbildung 4.6: Wird in dem linken Layout das Element B mit Element C verschmolzen, treten keine Konflikte auf. Falls jedoch das Element B mit Element

A verschmolzen wird, kommt es später zum rechts dargestellten Konflikt: Der Graph enthält keine Kanten mehr, da es keine überdeckungsfreie Nachbarn mehr gibt. Solche Konflikte können dann auftreten, wenn Elemente aus unterschiedlichen Streifen zu früh verschmolzen werden. Im Beispiel aus Abbildung 4.6 muss als allererstes der gesamte Bogen vertikal zwischen den Elementen A und B zerschnitten werden. Dies ist im verschmolzenen rechten Bogen nicht mehr möglich.

Für Konflikte definieren wir die Menge der am Konflikt beteiligten Schnittelemente K. Ein Schnittelement gehört zu dieser Menge, wenn der Konflikt nicht mehr vorhanden wäre, wenn das Element entfernt werden würde. Im Beispiel aus Abbildung 4.6 ist das Element G als einziges nicht am Konflikt beteiligt. Zusätzlich sind auch die Schnittelemente am Konflikt beteiligt, die innerhalb des kleinsten überdeckenden Rechtecks von zwei anderen am Konflikt beteiligten Elementen liegen. Diese Bedingung würde beispielsweise im Bogen aus Abbildung 4.6 für Elemente gelten, wenn sie sich im „Loch “ in der Mitte des Bogens befinden würden. Des Weiteren definieren wir das Rechteck R_K als das kleinstmögliche Rechteck, welches alle am Konflikt beteiligten Elemente überdeckt.

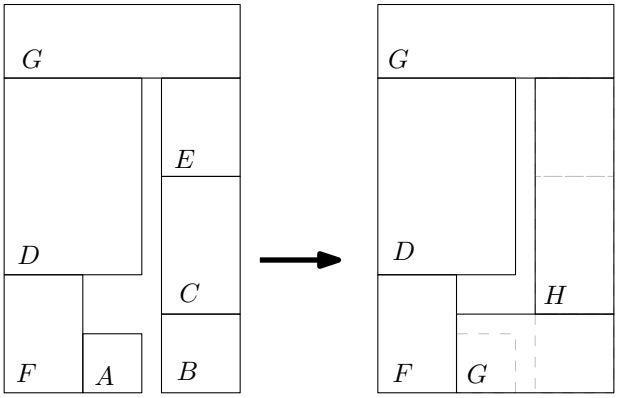


Abb. 4.6: Beispiel für das Auftreten eines Konflikts: Unter Umständen kann es für den linken Bogen zum rechts gezeigten Konflikt kommen. Die Menge der am Konflikt beteiligten Schnittelemente ist $\{D, F, G, H\}$.

SolveConflict SolveConflict löst einen Konflikt auf und sorgt dafür, dass genau diese Konstellation nicht noch einmal auftreten kann. Um einen Konflikt aufzulösen, suchen wir zunächst einen Schnitt s_{solve}, welcher R_K guillotinenmäßig zerschneidet. Zusätzlich wird gefordert, dass s_{solve} kein von R_K komplett überdecktes *Ursprungselement* zerschneidet. Der Schnitt wird nicht zu S hinzugefügt. Dieser Schnitt wird zwangsläufig einige der zerschmolzenen Elemente zerschneiden, sonst würde gar kein Konflikt vorliegen. Für diese Elemente machen wir die Verschmelzungen, durch die dieses Element entstanden ist, rückgängig: Die repräsentierenden Knoten werden aus dem Graphen entfernt, die von diesen Elementen überdeckten Ursprungselemente hinzugefügt und die entsprechenden Schnitte aus S entfernt.

Es gibt in jedem Konflikt mindestens eine gültige Möglichkeit für s_{solve}, da die Menge

der von R_K überdeckten ursprünglichen Elemente andernfalls nicht mit Guillotinen-Schnitten voneinander trennbar wären. Wir gehen jedoch stets davon aus, dass eine gültige Lösung für die Eingabe existiert.

Damit der gleiche Konflikt nicht mehrmals auftreten kann, wird dafür gesorgt, dass die rückgängig gemachten Verschmelzungen nicht noch einmal durchgeführt werden können. Für diesen Zweck entfernen wir für jedes wieder neu eingefügte Ursprungselement die Ursprungselemente auf der anderen Seite des Schnittes aus der Menge der Nachbarn. Damit stellen wir sicher, dass der Schnitt s_{solve} nicht wieder blockiert werden kann. Solange diese Nachbarschaftsverbote gültig sind, können die aufgelösten Elemente nicht wieder neu gebildet werden. Daraus folgt, dass der gelöste Konflikt nicht wieder entstehen kann. Dies ist ebenfalls garantiert, sobald eines der neu eingefügten Ursprungselemente mit einem nicht neu eingefügten Ursprungselement verschmolzen wird, da dies die Bildung des aufgelösten Elements ebenfalls unterbindet. Sobald dieser Fall eintritt, können die Nachbarschaftsverbote, die aus diesem Konflikt hervorgegangen sind, wieder aufgehoben werden. Da Nachbarschaftsverbote beim Verschmelzen auf das neue Element übertragen werden, ist dies auch notwendig, damit die Heuristik eine Lösung findet.

Zur Veranschaulichung betrachten wir den Ablauf der Konfliktlösung für das Beispiel aus Abbildung 4.6. Der Schnitt s_{solve} verläuft zwischen den Elementen D und H und zerteilt nur das Element G. Dieses wird daraufhin wieder durch die Elemente A und B ersetzt. Aus Sicht des Algorithmus sind diese nun jedoch nicht benachbart und können folglich auch nicht wieder zum Element G verschmolzen werden. Sobald A mit F oder B mit H verschmolzen wurde, wird das Nachbarschaftsverbot bezüglich A-beinhaltende und B-beinhaltende Blöcke wieder aufgehoben. Dadurch kann in einem späteren Schritt über den Schnitt s_{solve} verschmolzen werden.

4.2 Korrektheit und Laufzeit

Korrektheit Im Folgenden wird gezeigt, dass die Heuristik für lösbare Instanzen stets eine gültige Schnittreihenfolge liefert. Für die Schleife in Algorithmus 3 gilt stets folgende Schleifeninvariante: *Die Schnitte in der Menge S zerschneiden alle verschmolzenen Elemente so, dass die von ihnen überdeckten Elemente freigestellt werden.* Bevor die Schleife zum ersten Mal durchlaufen wird, gilt die Invariante offensichtlich, da noch keine Elemente verschmolzen wurden. GenerateCuts(L,E') (siehe Algorithmus 6) wird in jedem Schleifendurchlauf aufgerufen und fügt in Zeile 11 für alle in dieser Iteration verschmolzenen Elemente die Schnitte hinzu, die jeweils die überdeckten Elemente freistellen. Falls es zu einem Konflikt kommt, wird ein verschmolzenes Element aufgelöst und alle Schnitte, die beim Verschmelzen dieses Elementes hinzugefügt werden, werden wieder entfernt. Folglich gilt die Invariante nach jeder Schleifeniteration. Aus der Abbruchbedingung der Schleife folgt unmittelbar, dass sie solange durchlaufen wird, bis alle Elemente zu einem einzigen Element verschmolzen sind. Mit der Gültigkeit der Invariante und den Schnitten für das letzte Element (siehe Zeile 14 in Algorithmus 3) folgt, dass die Heuristik eine korrekte Schnittreihenfolge zurückgibt.

Die Anzahl der Durchläufe ist endlich: Solange keine Konflikte auftreten, wird die Knotenanzahl pro Iteration um mindestens 1 verringert. Das gilt, da für jede Kante im Matching E' zwei Elemente miteinander verschmolzen werden. Das Matching wiederum besteht aus mindestens einer Kante, da der Graph Kanten enthält solange es überdeckungsfreie Nachbarn gibt. Da in einem Bogen mit endlich vielen Elementen nur endlich viele verschiedene Konflikte auftreten können und der gleiche Konflikt nicht mehrmals auftreten kann (dafür sorgt SolveConflict), terminiert der Algorithmus in endlicher Zeit.

Laufzeit Die Best-Case Laufzeit der Heuristik liegt in $O(|L|^2)$. Dabei ist $|L|$ die Anzahl der Elemente des Schnittbogens. Die Bestimmung der Nachbarn (Zeile 2 in Algorithmus 3) für jedes Element lässt sich in $O(|L|^2)$ durchführen. Da alle Knoten im Graphen höchstens Grad 4 haben, gilt $O(|E|) \subseteq O(|L|)$ für alle Iterationen. Die for-Schleife benötigt mit der Abschätzung für $O(|E|)$ asymptotisch $O(|L|)$ Schritte. Größtmögliche Matchings können in $O(\sqrt{|L|}|E|)$ gefunden werden [MV80]. Sowohl GenerateCuts als auch MergeElements haben eine asymptotische Laufzeit von $O(|E|) \subseteq O(|L|)$.

Die Analyse der Laufzeit von BuildGraphEdges (siehe Algorithmus 4) ist etwas aufwendiger. Dafür betrachten wir zunächst einen Graphen $G' = (V, E')$. Auch in diesem entspricht die Menge der Schnittelemente der Knotenmenge. Jedoch sind in G' Schnittelemente genau dann adjazent, wenn sie benachbart (also nicht unbedingt überdeckungsfrei benachbart) sind. Im Gegensatz zu G gibt es für G' keine obere Schranke für den Grad der Knoten. Allerdings sind derartig konstruierte Graphen stets planar, da sich aus dem Bogen leicht eine Einbettung des Graphens in die Ebene gewinnen lässt. Dafür wird jedes Schnittelement im Bogen (entspricht den Knoten in G') durch einen Punkt in dessen Schwerpunkt ersetzt. Die Kanten können als Strecken zwischen den entsprechenden Punkten eingezeichnet werden. Aus der Planarität von G' und dem eulerschen Polyedersatz folgt, dass die Anzahl der Kanten in G' wie folgt beschränkt ist: $|E| \leq 3|V| - 6$. Ein Algorithmus, der jeweils für jeden Knoten in G' alle Nachbarn besucht, braucht $2|E|$ viele Schritte und hat somit eine asymptotische Laufzeit von $O(|V|)$. Folglich läuft BuildGraphEdges in $O(|L|)$.

Wir betrachten zunächst den Best-Case, in dem keine Konflikte auftreten. Für diesen ist die zeitaufwendigste Operation innerhalb der while-Schleife von Algorithmus 3 das Bestimmen des größtmöglichen Matchings mit einer Laufzeit von $O(|L|^{1.5})$. Die Anzahl der Durchläufe der while-Schleife hängt von der Anzahl der auftretenden Konflikten sowie der Größe der gefundenen Matchings ab. Im Best-Case is die Größe des Matchings stets maximal, also $\lfloor |L|/2 \rfloor$. Da die Anzahl der Elemente somit in jeder Iteration halbiert wird, gibt es folglich $\log(|L|)$ Durchläufe. Damit ergibt sich eine Best-Case Laufzeit der Schleife von $O(\log(|L|) \cdot |L|^{1.5})$ und eine Best-Case Laufzeit des gesamten Algorithmus von $O(|L|^2)$ (setzen der Nachbarn). Im Worst-Case tritt die maximal mögliche Anzahl an Konflikten auf. Deswegen ist die Worst-Case Laufzeit nicht mehr polynomiell, sondern exponentiell. Für praxisrelevante Bögen kann dieser Fall jedoch ausgeschlossen werden. Bei den in Kapitel 4.3 beschriebenen Tests war die Anzahl der Iterationen der while-

Schleife stets geringer als die Anzahl der Elemente.

4.3 Testergebnisse

Eigenschaften und Annahmen der Implementierung Die Heuristik wurde im Rahmen dieser Arbeit in Java implementiert. Für die Darstellung des Graphen wurde auf die Bibliothek *JGraphT* [N+08] (lizensiert unter LGPL) zurückgegriffen. Diese Bibliothek bietet auch zwei Algorithmen für größtmögliche Matchings an. Der wesentlich schnellere von diesen ist die Implementierung des von John E. Hopcroft und Richard M. Karp entworfenen Algorithmus für größtmögliche Matchings in bipartiten Graphen [HK73], welcher eine Laufzeit von $O(|E|\sqrt{|V|})$ hat. Wir gehen davon aus, dass der größte Abstand zwischen zwei überdeckungsfreien Nachbarn kleiner ist als das kleinste Schnittelement. Das bedeutet, dass kein Schnittelement in die Lücke zwischen zwei anderen Schnittelemente passt. Bögen aus der Praxis erfüllen diese Anforderung in der Regel, da der Bogen andernfalls nicht sehr platzsparend gepackt wäre. Für solche Bögen ist der korrespondierende Graph bipartit und der Algorithmus von Hopcroft und Karp kann für die Bestimmung des größtmöglichen Matchings verwendet werden. Für andere Bögen kann nicht garantiert werden, dass der implementierte Algorithmus eine gültige Schnittreihenfolge liefert. Dieser Fall trat in den hier getesteten Datensätzen niemals auf.

Alle Tests liefen unter folgenden Rahmenbedingungen:

- CPU: Dual-Core AMD E-450 APU (1,65 GHz)
- Arbeitsspeicher: 4 GiB
- Betriebssystem: Debian GNU/Linux 7.8 (wheezy) 64bit
- Java version 1.8.0

4.3.1 Ergebnisse auf künstlich konstruierten Datensätzen

Zunächst konstruieren wir einen Datensatz mit einer bestimmten Klasse von Bögen. Alle Bögen bestehen aus quadratischen Schnittelementen der gleichen Größe, die in einem homogenen Gitter angeordnet werden. Die Bögen unterscheiden sich in der Anzahl der Schnittelemente, die wir mit n bezeichnen, wobei n stets eine Quadratzahl ist.

Naive Lösung Als *naive* Lösung bezeichnen wir eine Schnittreihenfolge, die niemals mehrere Blöcke gemeinsam schneidet und somit keine Schnitte einspart. Alle naiven Lösungen benötigen gleich viele Schnitte. Eine Möglichkeit unsere Bögen naiv zu zerschneiden ist, den Bogen zunächst mit $\sqrt{n}-1$ vielen parallelen Schnitten in $\sqrt{n}$ viele Streifen zu zerteilen. Für jeden von diesen werden nun weitere $\sqrt{n}-1$ viele Schnitte benötigt, damit alle Elemente freigestellt sind. Die Anzahl der Schnitte der naiven Lösung beträgt folglich $\sqrt{n}-1+(\sqrt{n}\cdot(\sqrt{n}-1))=n-1$ und liegt in $\Theta(n)$. In Abbildung 4.7 ist die naive Lösung für einen Bogen aus dem Datensatz mit 16 Elementen eingezeichnet.

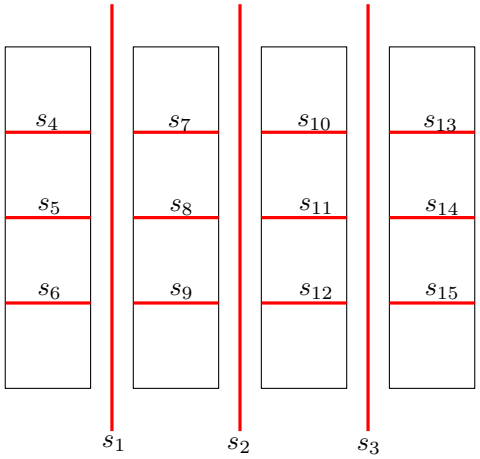


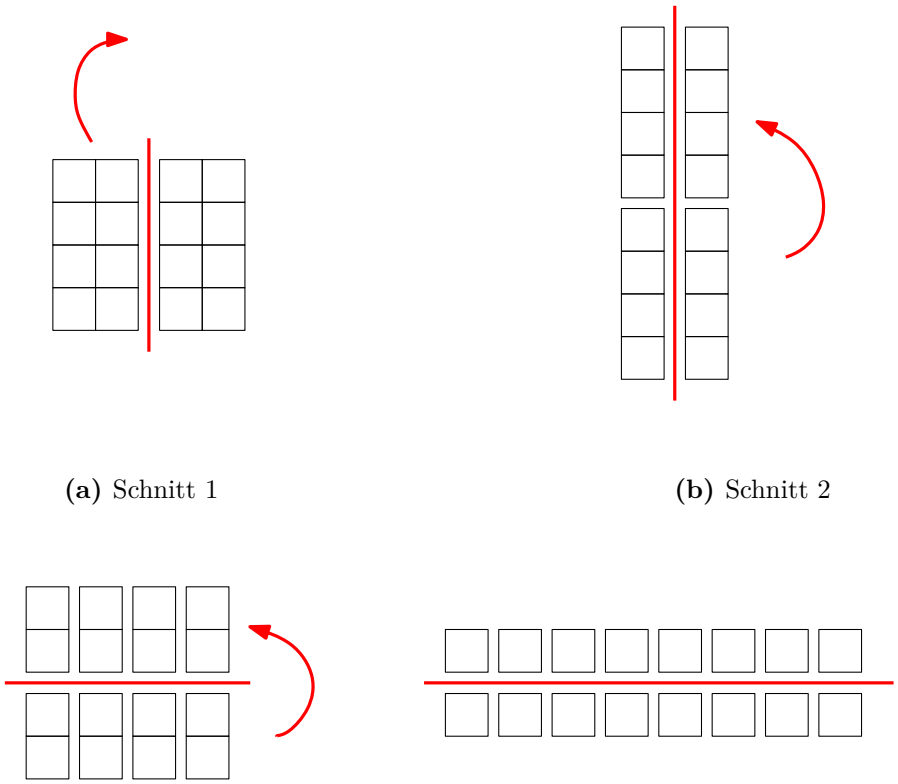
Abb. 4.7: Eine naive Lösung für einen Bogen mit 16 Elementen und den eingezeichneten Schnitten.

Optimale Lösung Für die Bögen aus unserem konstruiertem Datensatz lässt sich auch die Anzahl der Schnitte einer optimalen Lösung angeben. Für den Bogen mit 16 Elementen ist so eine Reihenfolge in Abbildung 4.8 angegeben. Der Bogen wird zunächst vertikal mittig zerteilt. Anschließend können bereits die zwei daraus entstehenden Blöcke gemeinsam vertikal geschnitten werden. Für größere Bögen mit noch mehr vertikalen Schnittkanten würden mit dem Schnitt vier Blöcke gemeinsam geschnitten werden. Mit dieser Vorgehensweise werden $\lceil \log \sqrt{n} \rceil$ viele Schnitte für alle vertikalen Schnittkanten benötigt. Schneidet man alle horizontalen Schnittkanten analog, erhält man für die Anzahl der Schnitte einer optimalen Lösung $2\lceil \log \sqrt{n} \rceil$.

Testergebnisse Da der verwendete Matching-Algorithmus bei der gleichen Eingabe nicht immer die gleiche Kantenmenge liefert, variiert bei manchen Eingaben die Anzahl der gefundenen Schnitte von Aufruf zu Aufruf. Insbesondere die Wahl des ersten Matchings hat einen Einfluss auf das Ergebnis. Deshalb wurde die Heuristik für jede Eingabe 10 Mal ausgeführt. Das Ergebnis dieser 10 Durchläufe ist in Abbildung 4.9 dargestellt. In Abbildung 4.10 wird die jeweils beste Lösung mit der optimalen sowie der naiven Lösung verglichen. Die in Abbildung 4.10 veranschaulichten Laufzeiten entsprechen jeweils der Durchschnitt dieser 10 Aufrufe.

In der Praxis sind die Druckelemente jedoch nicht quadratisch, sondern rechteckig. Wir betrachten nun die Ergebnisse für einen Datensatz, der im Gegensatz zum vorherigen nur gitterartig angeordnete Rechtecke verwendet. Tatsächlich gibt es reale Druckbögen, die nach diesem Muster aufgebaut sind. Auch hier lassen wir die Heuristik 10 mal für jede Eingabe laufen, siehe 4.12.

Auf diesem Datensatz liefert die Heuristik in mehreren Fällen eine optimale Lösung, siehe Abbildung 4.13. Der Grund dafür ist, dass es bei manchen Bögen nun nur noch zwei Möglichkeit für das erste Matching gibt: Entweder werden alle Elemente mit einem



(a) Schnitt 1 **(b)** Schnitt 2

(c) Schnitt 3 **(d)** Schnitt 4

Abb. 4.8: Eine optimale Schnittreihenfolge mit 4 Schnitten für den Bogen mit 16 Elementen aus Abb. 4.7. Die Pfeile deuten die Verschiebung der Blöcke für den nächsten Schnitt an.

vertikalen, oder einem horizontalen Nachbar verschmolzen. Außerdem treten bei Bögen des rechteckigen Testdatensatzes nie Kollisionen auf, während bei drei der quadratischen Bögen jeweils ein Konflikt auftgetreten ist. Diese Konflikte führen auch zu den Ausreißern in der Laufzeit, siehe Abbildung 4.11.

Bei quadratischen Rastern gibt es wesentlich mehr Möglichkeiten für das erste Matching, die in der Regel nicht so gleichmäßig sind. Ein Beispiel mit den beiden jeweiligen Bögen mit 64 Elementen zeigt Abbildung 4.15: Bei beiden Bögen wird mit dem als erstes gefundenen Schnitt die gleiche Anzahl an Schnitten gespart, jedoch wird das Matching in der nächsten Iteration bei dem Bogen mit rechteckigen Elementen größer sein. Tatsächlich wird für den in Abbildung 4.15 gezeigten rechteckigen Bogen stets eine optimale Lösung gefunden. Wenn die Anzahl der Elemente pro Reihe (entspricht $\sqrt{n}$) ungerade ist, liefert die Heuristik, wie in 4.13 ersichtlich, schlechtere Resultate. Auch dies läßt

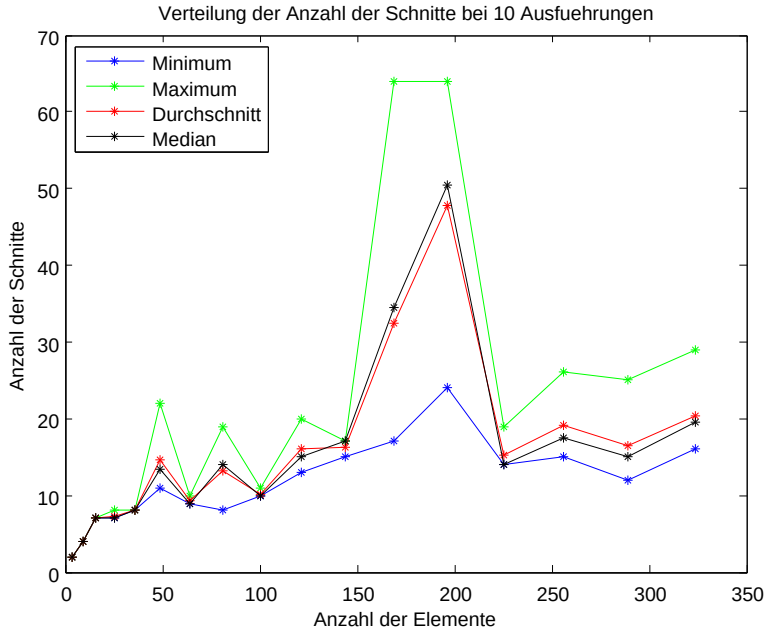


Abb. 4.9: Minimum, Maximum, Durchschnitt und Median der Anzahl der Schnitte für 10 Durchläufe der Heuristik auf dem Datensatz der quadratischen Gitter.

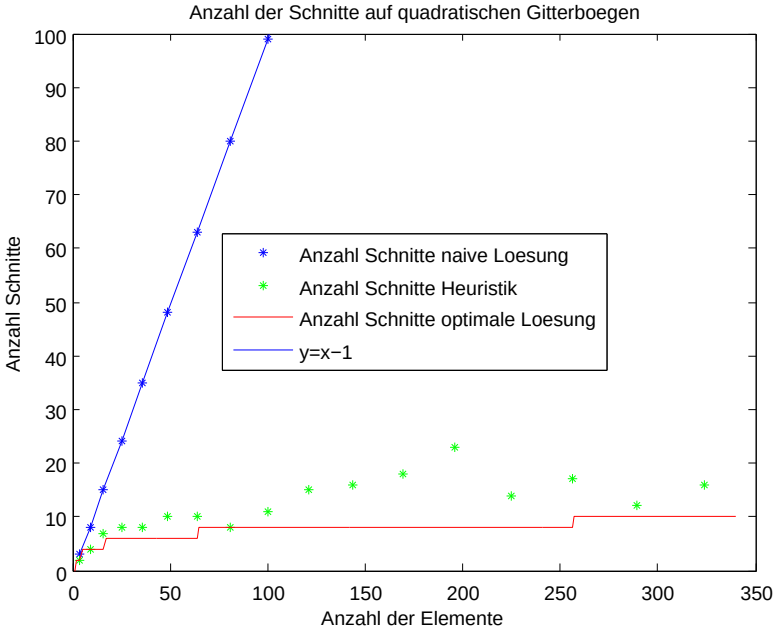


Abb. 4.10: Vergleich von Heuristik, optimaler und naiver Lösung auf dem quadratischen, gitterartigen künstlichen Datensatz. Im Vergleich zur naiven Lösung benötigt die Heuristik deutlicher weniger Schnitte.

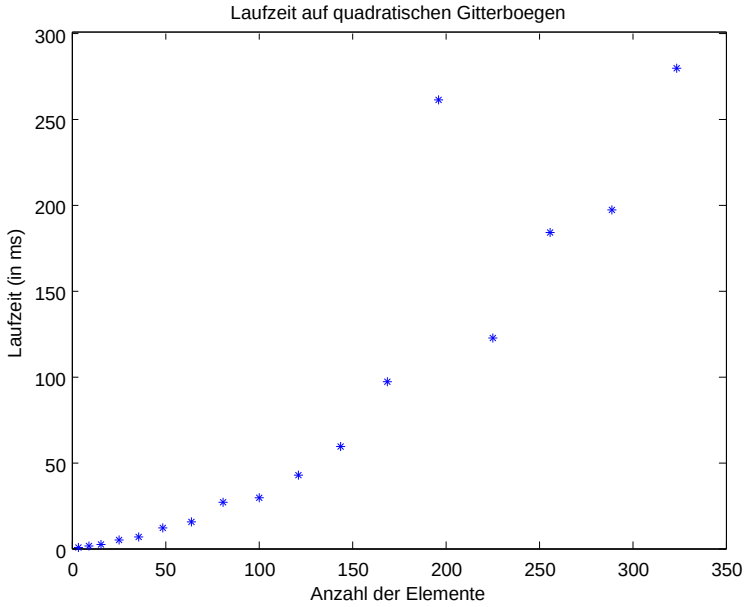


Abb. 4.11: Laufzeit von 10 Ausführungen der Heuristik auf dem Datensatz der quadratischen Gitterbögen.

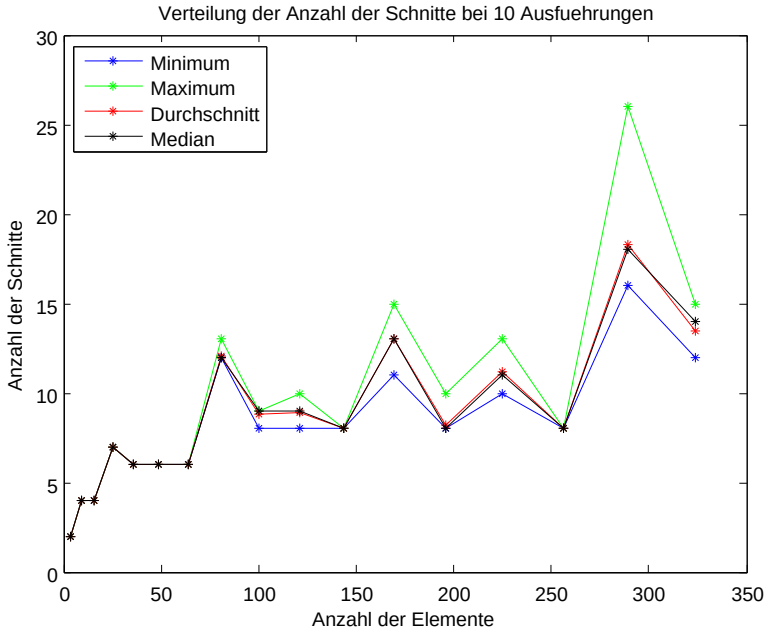


Abb. 4.12: Minimum, Maximum, Durchschnitt und Median der Anzahl der Schnitte für 10 Durchläufe der Heuristik auf dem Datensatz der rechteckigen Gitter.

sich durch unregelmäßige Matchings erklären, da das Matching pro Reihe dann nicht aufgehen kann und die ausgelassenen Elemente zueinander versetzt sein können.

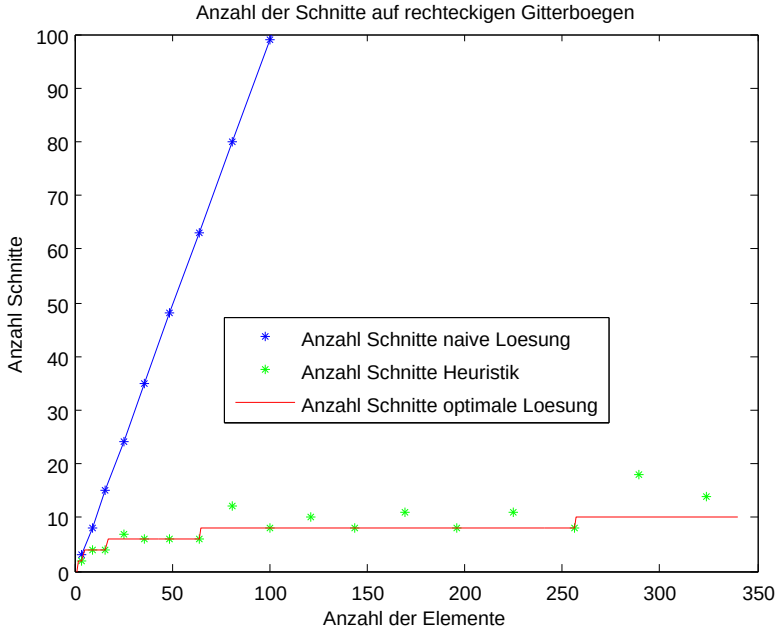


Abb. 4.13: Vergleich von Heuristik, optimaler und naiver Lösung auf dem rechteckigen, gitterartigen künstlichen Datensatz. Im Vergleich zur naiven Lösung benötigt die Heuristik deutlicher weniger Schnitte. Bei vielen Bögen wird eine optimale Lösung gefunden.

4.3.2 Ergebnisse auf realem Datensatz

Die Heuristik wurde auch auf einem praxisrelevanten Datensatz einer großen Online-Druckerei getestet. Der Datensatz besteht aus 100 Druckbögen. Die Ergebnisse sind in Abbildung 4.16 dargestellt. Die Heuristik spart bei 92 Bögen mindestens einen Schnitt ein. Unter den restlichen acht Bögen bestehen fünf aus nur zwei Elementen. Bei diesen können generell keine Schnitte eingespart werden. Die Heuristik liefert folglich bei mindestens 97% der Bögen ein besseres Ergebnis als die naive Lösung.

Besonders viele Schnitte werden auf den Bögen mit einer großen Anzahl an Elementen eingespart. Einen dieser Bögen (mit $n = 149$) zeigt Abbildung 4.18. Die Lösung der Heuristik für diesen Bogen besteht aus 17 Schnitten. Hier werden 89% der Schnitte der naiven Lösung eingespart. Das gute Ergebnis lässt sich leicht erklären: Der Bogen hat eine ähnliche Struktur wie die Bögen aus dem konstruierten Datensatz mit rechteckigen Elementen, für die die Heuristik allgemein recht gute Ergebnisse liefert. Auf den Bögen mit weniger Elementen, liegt die Lösung der Heuristik näher an der naiven Lösung. Das bedeutet aber nicht zwangsläufig, dass die Ergebnisse der Heuristik in diesem Bereich schlechter sind, da die optimale Schnittanzahl nicht bekannt ist. Schließlich kann es auch

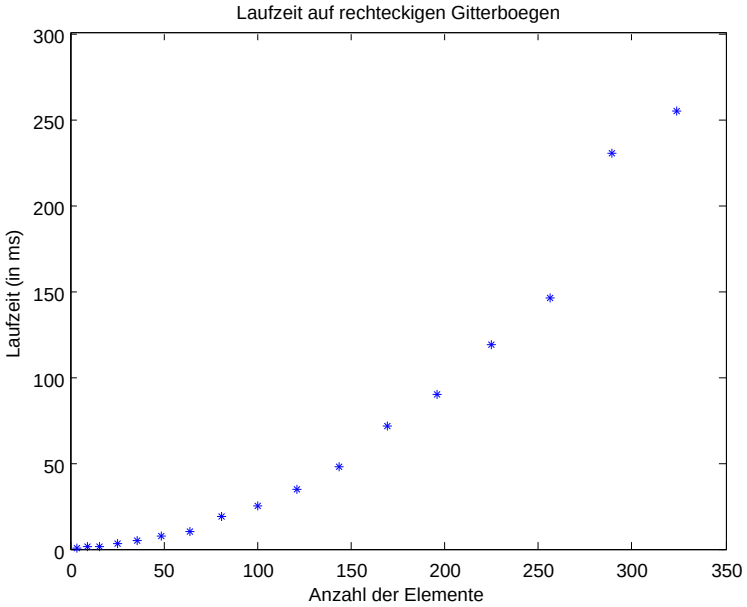
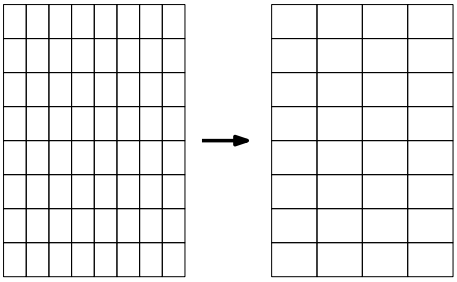
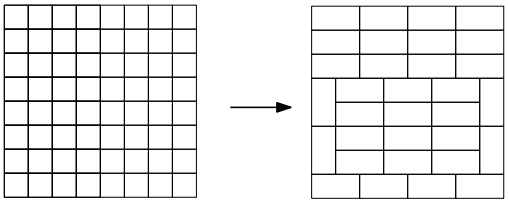


Abb. 4.14: Laufzeit von 10 Ausführungen der Heuristik auf dem Datensatz der rechteckigen Gitterbögen.



(a) Verschmelzung im rechteckigen Fall entsprechend dem vertikalen Matching



(b) Da die Kantengewichte zwischen Zeilen und Spalten die gleichen sind, ist das Matching im quadratischen Fall meistens unregelmäßig

Abb. 4.15: Vergleich der ersten Verschmelzung zwischen dem Bogen mit rechteckigen bzw. quadratischen Elementen.

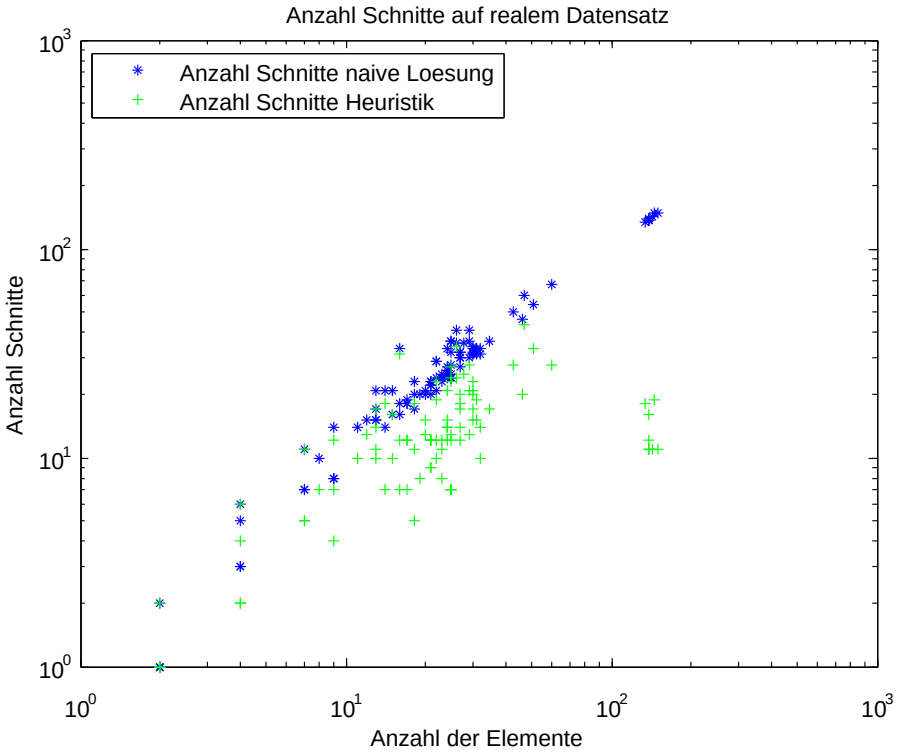


Abb. 4.16: Vergleich von Heuristik und naiver Lösung auf einem praxisrelevanten Datensatz bestehend aus 100 Druckbögen. Die Achsen sind logarithmisch skaliert. Die Heuristik liefert bei 97% der Bögen eine niedrigere Schnittanzahl als die naive Lösung.

Bögen geben bei der die naive Lösung eine optimale ist: Es gibt keine Garantie dafür, dass überhaupt Schnitte eingespart werden können.

Die Laufzeit der Heuristik ist sehr gering: Die Heuristik benötigt für keinen Bogen mehr als 50ms. Für Bögen mit nahezu 150 Elementen wird nur doppelt so viel Zeit benötigt wie für Bögen mit 50 Elementen. Ein Grund dafür ist, dass die Bögen mit gitterartigen Strukturen und vielen Elementen kaum mehr Schnitte brauchen wie „ungünstig“ strukturierte Bögen mit wenigen Elementen. Die Anzahl der Iterationen der while-Schleife in Algorithmus 3 hängt aber letztlich vor allem von der tatsächlichen Anzahl der zurückgegebenen Schnitte ab. Die Ausreißer in der Laufzeit lassen sich teilweise durch auftretende Konflikte erklären. Insgesamt gab es vier Bögen, bei denen es zu Konflikten kam. Beispielsweise hat der Bogen mit 51 Elementen die höchste Anzahl an auftretenden Konflikten, nämlich vier.

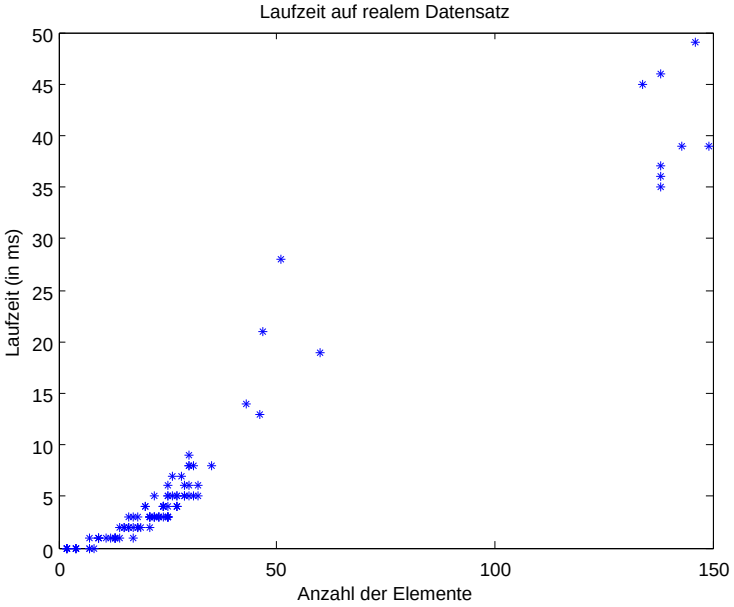


Abb. 4.17: Laufzeit der Heuristik auf dem realen Datensatz

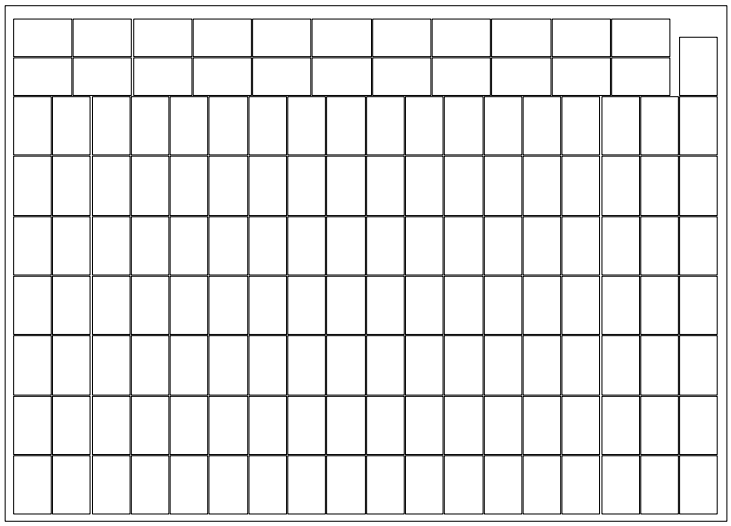


Abb. 4.18: Ein Bogen aus dem Datensatz aus der Praxis mit 149 Elementen.

5 Zusammenfassung und Ausblick

Diese Arbeit behandelte das noch wenig erforschte Problem der Schnittplanoptimierung. Zunächst wurde in Kapitel 2.2 durch eine Reduktion von Vertex-Cover gezeigt, dass es sich um ein NP-vollständiges Problem handelt. Für die Reduktion wird aus dem gegebenen Graphen nach einem bestimmten Schema ein Druckbogen konstruiert, in dem einige Schnittkanten Knoten repräsentieren. Aus den gewählten Schnittdistanzen dieser Schnittkanten lässt sich dann die Lösung der Instanz des Vertex-Cover Problems gewinnen.

In Kapitel 3 wurde ein ganzzahliges lineares Programm zur Bestimmung einer optimalen Lösung beschrieben. Dies wurde erreicht, indem jede Schnittkante einem Schnitt zugeordnet wird. Für diesen Zweck werden mehrere Familien von binären Indikatorvariablen verwendet, welche die Reihenfolge der Schnitte, das gemeinsame Schneiden von Schnittkanten und die gewählten Distanzen modellieren. Die zu maximierende Zielfunktion berechnet die Anzahl der eingesparten Schnitte.

Anschließend wurde in Kapitel 4 eine Heuristik vorgestellt, welche iterativ Elemente verschmilzt und dabei genau die Schnitte hinzufügt, welche diese Verschmelzungen rückgängig machen. Dabei können Konflikte auftreten, die aufgelöst werden müssen. Außerdem wurde die Heuristik auf konstruierten und realen Druckbögen getestet. Dabei zeigte sich, dass die Heuristik auf gitterartig strukturierten Bögen oft eine optimale Lösung findet und auf 97% der Bögen aus der Praxis im Vergleich zum naiven Ansatz Schnitte einspart.

Es wäre interessant, das ILP zu implementieren um so die Heuristik auf allen realen Bögen mit der optimalen Lösung vergleichen zu können. Außerdem bietet die Heuristik viele Ansätze für weitere Optimierungen. Beispielsweise ist zu erwarten, dass die auftretenden Konflikte durch einen modifizierten Matching Algorithmus reduziert oder sogar gänzlich vermieden werden können. Dieser sollte möglichst nur Kanten zwischen vertikalen oder zwischen horizontalen Nachbarn auswählen und verhindern, dass gewissen Schnitte „verbaut“ werden. Da die Laufzeit der Heuristik recht gering ist, liegt auch der Ansatz nahe, einen Suchbaum aufzubauen: Anstatt nur ein Matching auszuwählen, werden mehrere Alternativen in Betracht gezogen und die Suche verzweigt sich entsprechend. Da sich für manche Bögen eine optimale Lösung nur finden lässt, wenn teilweise Matchings, die nicht größtmöglich sind, ausgewählt werden, kann der Suchbaum allerdings sehr groß werden. Eventuell könnte dann der Einsatz von Monte-Carlo-Tree-Search-Verfahren [BPW+12] oder die Anwendung einer Pruning-Strategie sinnvoll sein.

Betrachtet man jedoch die ursprüngliche Problemstellung aus der Praxis, in der es um die Beschleunigung des Schneidevorgangs an sich geht, stellen sich auch noch andere

Anforderungen. Da das Guillotinen-Messer der Schneidemaschine eine begrenzte Länge hat, sollten Heuristik und ILP so angepasst werden, dass die gemeinsame Breite von zusammen geschnittenen Blöcken diese Länge nicht überschreitet. Auch können nicht beliebig große Blöcke ausgelagert werden. Dieser müssen deshalb stets in der Maschine bleiben.

Auf den Zeitaufwand für die Umsetzung eines Schnittplans nehmen außerdem noch andere Faktoren als die Schnittanzahl, auf deren Optimierung sich diese Arbeit beschränkt hat, großen Einfluss. So ist es natürlich von enormen Vorteil, wenn der Block, der bereits unter dem Messer der Guillotine liegt, als nächstes geschnitten wird. Auch Drehungen und andere Verschiebeoperationen von Blöcken kosten Zeit und sollten vermieden werden. Ein möglicher Ansatz wäre hier, Schnittpläne mit bereits optimierter Schnittanzahl so zu mutieren, dass die Schnittanzahl konstant bleibt, der Schnittplan aber hinsichtlich anderer Bewertungskriterien verbessert wird. Tatsächlich werden die Schnittpläne der hier vorgestellten Algorithmen wohl erst durch eine derartige Nachverarbeitung – ob automatisiert oder manuell beim Schneidvorgang – praktikabel: Andernfalls kann der Schnittplan beispielsweise fordern, dass der bereits in der Maschine anliegende Block für den nächsten Schnitt um 180° gedreht wird, obwohl dieser Schnitt auch ohne diese Drehung mit einem anderen Abstand durchführbar wäre.

Literaturverzeichnis

[AP10] Rasmus R. Amossen und David Pisinger: Multi-dimensional bin packing problems with guillotine constraints. *Computers & Operations Research*, 37(11):1999–2006, 2010.

[BPW+12] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis und Simon Colton: A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[CH95] Nicos Christofides und Eleni Hadjiconstantinou: An exact algorithm for orthogonal 2-D cutting problems using guillotine cuts. *European Journal of Operational Research*, 83(1):21–38, 1995.

[Dyc90] Harald Dyckhoff: A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145–159, 1990.

[HK73] John E. Hopcroft und Richard M. Karp: An $n^{2.5}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[MV80] Silvio Micali und Vijay V Vazirani: An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In: *21st Annual Symposium on Foundations of Computer Science, 1980*, Seiten 17–27. IEEE, 1980.

[N+08] Barak Naveh *et al.*: JGraphT. `http://jgrapht.sourceforge.net`, 2008.

[oCaP12] EURO Special Interest Group on Cutting and Packing: Bibliography of Cutting and Packing Problems. `http://paginas.fe.up.pt/~esicup/tiki-index.php?page=Typology`, 2012. Zugriff: 2015-06-13.

[Pap81] Christos H. Papadimitriou: On the Complexity of Integer Programming. *J. ACM*, 28(4):765–768, 1981.

[Sch01] Uwe Schöning: Theoretische Informatik kurzgefasst. Spektrum, 2001.

[WHS07] Gerhard Wäscher, Heike Haußner und Holger Schumann: An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130, 2007.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 23. Juli 2015

. . . . . .
Adrian Loy