2D Guillotine-Zuschnitt

Simon Praetorius, Andreas Richter[1]
TU-Dresden
WS 2008/09

Zusammenfassung

Diese Arbeit entstand im Zuge unseres mathematischen Grundpraktikums. Sie behandelt das Problem des unbeschränkten Materialzuschnitts durch Guillotine-Schnitte. Der erste exakte Algorithmus wurde von Gilmore und Gomory (1965) formuliert, der auf dynamische Programmierung setzte. Herz verbesserte diesen (1972) durch die Einführung von Rasterpunktmengen. Durch weitere Reduzierung dieser Menge (u.a. Scheithauer, Terno und Lindemann (1987)) konnte das Verfahren enorm beschleunigt werden. Wir werden in dieser Arbeit die von den genannten Autoren gefundenen Resultate nutzen und eine darauf aufbauende Implementierung eines Algorithmus anbieten, der das oben genannte Problem exakt löst.

Keywords: rectangular knapsack problem, rectangular packing problem, guillotine, uncontrained, dynamic programming, slopp

Inhaltsverzeichnis

[1] Betreuer Marat Mesyagutov

1 Einleitung

Zuschnittprobleme treten in einer Reihe von Anwendungsfällen auf. In der Stahlindustrie beim Zuschnitt von Stahlstangen, in der Textilindustrie bei Zuschnitt von Stoffteilen aus einer Stoffbahn, beim Glas-, Holz- und auch beim Papierzuschnitt. Dabei werden unterschiedliche Anforderungen an die durchzuführenden Schnitte gestellt.

In dieser Arbeit wollen wir uns mit dem Guillotine-Zuschnitt einer rechteckigen Grundplatte von einer unbegrenzten Menge an Typenteilen beschäftigen. Den ersten exakten Algorithmus formulierten Gilmore und Gomory (1965), der auf dynamische Programmierung setzte. Herz verbesserte diesen Algorithmus (1972), durch die Einführung von Rasterpunktmengen und den Einsatz von oberen Schranken für das spezielle Problem des ungewichteten Zuschnitts, bei dem nur die Fläche optimiert wird. Beasley verallgemeinerte (1985) den Algorithmus von Herz und kombinierte sein Erkenntnisse mit denen von Gilmore und Gomory. Scheithauer, Terno und Lindemann reduzierten die Rasterpunktmengen (1987) noch weiter, was eine weitere Beschleunigung des Algorithmus zu Folge hatte. Wir werden in dieser Arbeit die von den genannten Autoren gefundenen Resultate nutzen und eine darauf aufbauende Implementierung eines Algorithmus anbieten, der das oben genannte Problem exakt löst.

Zu Beginn werden wir einige grundlegende Definitionen machen und das Problem des Guillotine-Zuschnitts mathematisch formulieren. Dann folgen ein paar Grundlagen zur Dynamischen Programmierung, die von Bellman (1957) formuliert wurden. Anschließend werden die Rasterpunktmengen, wie sie von Herz und Scheithauer vorgestellt wurden, eingeführt und die Problembeschreibung auf die neuen Erkenntnisse angewandt. Ein Algorithmus, der dieses Problem mittels dynamischer Programmierung löst, wird dann folgen und abschließend werden einige numerische Testresultate präsentiert. Außerdem werden mehr-stufige Zuschnitte behandelt, die eine weitere Einschränkung an die Guillotine-Schnitte stellen und mit dem freien Zuschnitt verglichen.

Bemerkung. *Einige mathematische Vorbermerkungen: wir schreiben* $\log(\ldots)$ *für* $\log_2(\ldots)$. *Die Maxima bzw. Minima über leeren Mengen sind definiert als* $\max(\emptyset) := -\infty$ *bzw.* $\min(\emptyset) := \infty$.

2 Das Guillotine Zuschnittproblem

Aus einer Rechtecksplatte der Dimension $H \times W$ sind kleinere Rechtecke (*Typenteile*) T_i der Größe $h_i \times w_i$ mit Bewertung c_i zuzuschneiden. Wir bezeichnen eine *Instanz* des Guillotine-Zuschnittproblems mit $\mathbb{I} = (H, W, \underline{h}, \underline{w}, \underline{c})$, wobei $\underline{h} = (h_1, \ldots, h_m)$, $\underline{w} = (w_1, \ldots, w_m)$ und $\underline{c} = (c_1, \ldots, c_m)$ und m die Anzahl der möglichen Typenteile ist. Es wird das unbeschränkte maximierungs-Problem betrachtet, d.h. jedes Typenteil kann in unbeschränkter Quantität produziert werden und die Summe der Kosten der produzierten Teile soll maximiert werden. Die möglichen Schnitte werden dadurch eingeschränkt, dass nur Guillotine-Schnitte in horizontaler und vertikaler Richtung zugelassen werden.

Bemerkung. *Wählen wir für die jeweiligen Gewichte* $c_i = h_i \cdot w_i$, *so wird eine Zuschnittvariante mit minimalem Abfall gesucht. Durch eine andere Festlegung der Gewichte können Präferenzen bezüglich der Typenteile festgelegt werden.*

Definition. Ein *Guillotine-Schnitt* teilt ein Rechteck mit einem geraden Schnitt von einer Kante zu einer anderen Kante in zwei Teile. Dabei sind die Schnitte orthogonal zu den Rechteckskanten auszuführen, entweder horizontal oder vertikal.

Definition. Ein *Schnittmuster* (engl. pattern) ist ein Vektor $\alpha = (\alpha_1, \ldots, \alpha_m)$ der die Anzahl der durch eine Folge von Guillotine-Schnitten produzierten Typenteile angibt, d.h. das Teil T_i wurde α_i-mal produziert.

Für eine Instanz $\mathbb{I}$ des Guillotine-Problems wird die *Menge aller zulässigen Schnittmuster* mit $S(\mathbb{I})$ bezeichnet.

Damit können wir das Optimierungsproblem des unbeschränkten Guillotine-Zuschnitts mathematisch formulieren:

$$V(H, W) := c^T \alpha \to \max \quad \text{bei} \quad \alpha \in S(\mathbb{I}) \tag{1}$$

In dieser Arbeit werden wir uns nur mit der Lösung dieses Problems beschäftigen.

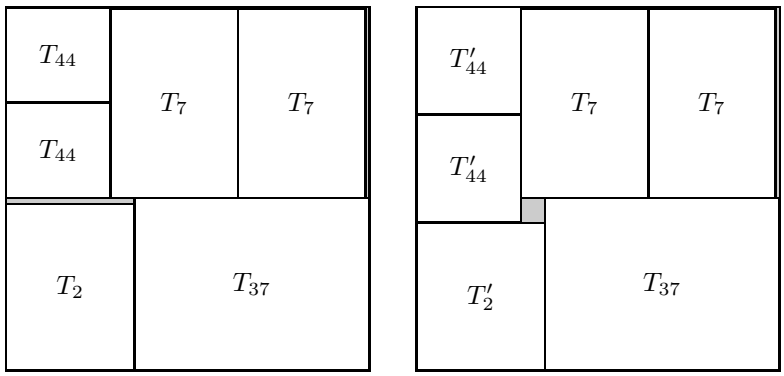

Fig. 1: *Links*: Ein Beispiel für einen Guillotine-Zuschnitt, *Rechts*: Kein Guillotine-Zuschnitt

Dazu wird im nächsten Abschnitt ein allgemeines Vorgehen zur Lösung spezieller Optimierungsaufgaben vorgestellt, mit dem wir das Guillotine-Zuschnittsproblem auch bearbeiten wollen (in Abschnitt 4).

3 Dynamische Programmierung

Richard Bellman beschäftigte sich ab 1949 mit mehrstufigen Entscheidungsproblemen in Verbindung mit der RAND-Corporation in Santa Monica. Im Rahmen seiner dortigen Forschungen bezeichnete und entwickelte er um 1957 das allgemeine Konzept der *dynamischen Programmierung*[2] (DP) in der Optimierung. Der Name entstand[3] aus einem Konzept der Physik, das einen zeitlichen Prozess, mit Hilfe von vorhergehenden Beobachtungen in kleineren Zeitspannen, beschreibt (dynamic) und Programmierung - angelehnt an Entscheidungsfindung, Planung und Optimierung.

[2] Entstanden ist das Grundkonzept schon in den 40er Jahren, wurde aber durch Bellman formalisiert

[3] [Dre02]

3.1 Optimalitätsprinzip von Bellmann

Ein Optimalitätsprinzip ist eine allgemeine Technik, um eine Lösung eines Optimierungsproblems zu finden. Dabei ist ein solches Vorgehen nicht für alle Probleme anwendbar. Bellman beschäftigte sich mit einem Optimalitätsprinzip, das die Gesamtlösung aus der Lösung von (kleineren) Teilproblemen zusammensetzt:

> *Eine optimale Politik hat die Eigenschaft, dass, wie auch immer der Anfangszustand und die erste Entscheidung ausfielen, die folgenden Entscheidungen für eine optimale Lösung sich auf den Zustand, der aus der ersten Entscheidung resultiert, beziehen müssen.*[4]

Wobei Bellman von Politik und Entscheidungen sprach, da er sich auf die optimierung von Entscheidungsprozessen bezog.

Zur Lösung eines Optimierungsproblems beschrieb Bellman eine Rekursionsgleichung:

$$f_n(x_n) = \min_{u_n \in U_n(x_n)} \{r_n(x_n, u_n) + f_{n-1}(t_n(x_n, u_n))\}, \quad n = 1, \ldots, N$$

mit dem Startzustand $f_0(x_0) = 0$. Dabei ist u_n eine Entscheidung im n-ten Schritt, $U_n(x_n)$ ein Bereich möglicher Entscheidungen (ev. abhängig vom Zustand x_n), $r_n(\ldots)$ die Kosten, die beim Übergang von einem Zustand x_n zu einem Zustand x_{n-1} entstehen und $t_n(\ldots)$ eine Transformations für einen Zustand x_n nach x_{n-1}. Die Optimallösung des Problems wird im Wert $f_N(x_N)$ gefunden.

3.2 Anwendung der Rekursionsvorschrift auf Zuschnittprobleme

Die Entscheidung, die wir in jedem Schritt fällen müssen ist, an welcher Stelle x ein Rechteck der Größe $h \times w$ geschnitten werden muss, so dass dieser Schnitt optimal bzgl. einer Bewertungsfunktion ist. Dabei ist noch zu unterscheiden, ob horizontal geschnitten werden soll, oder vertikal, bzw. ob es besser ist gar nicht zu schneiden und in das Rechteck direkt ein Typenteil zu platzieren. Also haben wir ein Entscheidungproblem in zwei Schritten: erstens welcher Schnitt gemacht wird (horizontal, vertikal, keinen Schnitt) und zweitens wo der jeweilige Schnitt platziert, bzw. welches Typenteil in das Rechteck gelegt werden soll.

Bezeichnen wir mit $V(h, w)$ den Wert eines Rechtecks $h \times w$, der durch ein optimales Schnittmuster entsteht. Beim unbeschränkten Zuschnittproblem, wie wir es hier betrachten, ergibt sich ein optimaler Schnitt unabhängig von vorher getroffenen Schnittentscheidungen. Das bedeutet, dass ein Rechteck $h \times w$ immer den optimalen Wert $V(h, w)$ besitzt, unabhängig davon, ob vorher schon ein gleiches Rechteck betrachtet wurde. Wenn Beschränkungen an die Anzahl der zu produzierenden Typenteile gestellt sind, ist diese Eigenschaft nicht mehr gegeben und es kann i.A. keine Lösung mittels dynamischer Programmierung gefunden werden.

[4] [BD62, Seite 15] Aus dem Englischen: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

Zur Beschreibung der speziellen Rekursionsvorschrift, angepasst auf das Guillotine-Zuschnittproblem, definieren wir die Funktion $v(h, w)$, die angibt, welcher maximale Wert durch platzieren eines Typenteils in dem Rechteck $h \times w$ erreicht werden kann:

$$v(h, w) := \max_i \{c_i : h_i \leq h, w_i \leq w\}$$

Damit lautet die Rekursionsvorschrift des Optimierungsproblems (1) folgendermaßen:

$$V(h, w) := \max \begin{cases} \max\{V(h, x) + V(h, w - x) \mid x \in \mathbb{N}, x \leq \frac{w}{2}\}, \\ \max\{V(y, w) + V(h - y, w) \mid y \in \mathbb{N}, y \leq \frac{h}{2}\}, \\ v(h, w),\ 0 \end{cases} \tag{2}$$

Das Prinzip der dynamischen Programmierung kann auf das Zuschnittproblem angewandt werden, da ein optimales Schnittmuster für ein Rechteck $h \times w$ aus der Summer zweier optimaler Schnittmuster von kleineren Problemen entsteht und die Addition monoton-separabel ist.[5]

4 Rasterpunkte

Beim Zerteilen eines Rechtecks ändern sich die Optimalwerte für die enstehenden Teilrechtecke nur an diskreten Stellen. Dies ist der Fall, wenn ein Schnitt ausgeführt wird, der das Hineinsetzten eines neuen/weiteren Typenteils ermöglicht. Betrachtet man den optimalen Schnittplan so kann ein neues Teil genau an den Positionen gesetzt werden, die sich aus Linearkombination aller Teile ergeben. Deswegen hat Herz (1972) die *Rasterpunktemenge* (engl. discretization points) eingeführt, die alle ganzzahligen Linearkombinationen der Breiten (bzw. Höhen) der Typenteile beinhaltet, d.h. für einen Vektor $\underline{l}$ von Längen ergibt sich die Menge

$$D(\underline{l}) := \{\textstyle\sum_i \alpha_i l_i : \alpha_i \in \mathbb{N}\}$$

Eine Beschränkung der Menge auf Kombinationen kleiner als ein vorgegebener Wert L ist auch sinnvoll, da die Breite (bzw. Höhe) der Grundplatte nicht überschritten werden darf:

$$D(\underline{l}, L) := \{l \in D(\underline{l}) : l \leq L\}$$

Diese Menge wird im Folgenden mit D abgekürzt, da wir uns in diesem Abschnitt nur auf verallgemeinerte Längen $\underline{l} = (l_1, l_2, \ldots)$ beziehen.

In der Rekursionsvorschrift wird vertikal (bzw. horizontal) über alle Wert $x \in (0, \lfloor \frac{w}{2} \rfloor]$ das Maximum gebildet. Wenn man die Auswahl einschränkt auf die Menge D, dann müssen die Zerteilungen, wie $\lfloor \frac{w}{2} \rfloor$, auch in der Menge D vorgenommen werden. Es wird also eine Funktion $\mu : 2^{\mathbb{N}} \times \mathbb{N} \to \mathbb{N}$ benötigt, die das größte Elemente aus D bestimmt, das kleiner als ein vorgegebener Wert x ist:

$$\mu_D(x) := \max\{l \in D : l \leq x\}$$

Die effiziente Implementierung dieser Funktion ist ausschlaggebend für die Effizienz des resultierenden Algorithmus (siehe Abschnitt 5.1).

[5] Siehe Vorlesung Diskrete Optimierung von G. Scheithauer SS 2008

Scheithauer, Terno und Lindemann haben in [STL87] eine weitere Vergröberung der Punktmenge vorgeschlagen. Die *reduzierte Rasterpunktmenge* bildet sich danach wie folgt:

$$D^{\mathrm{red}}(\underline{l}, L) := \{\mu_D(L - l) : l \in D\}$$

Mit den Definitionen $\mathcal{H} := D^{\mathrm{red}}(\underline{h}, H) \cup \{H\}$ und $\mathcal{W} := D^{\mathrm{red}}(\underline{w}, W) \cup \{W\}$ ergibt sich aus (2) dann folgendes Optimierungproblem für $h \in \mathcal{H}$ und $w \in \mathcal{W}$:

$$\mathcal{V}(h, w) := \max \begin{cases} \max\{V(h, x) + V(h, \mu_{\mathcal{W}}(w - x)) \mid x \in \mathcal{W}, x \leq \frac{w}{2}\}, \\ \max\{V(y, w) + V(\mu_{\mathcal{H}}(h - y), w) \mid y \in \mathcal{H}, y \leq \frac{h}{2}\}, \\ v(h, w),\ 0 \end{cases} \tag{3}$$

Scheithauer hat in [STL87] bzw. [Sch08] gezeigt, dass $\mathcal{V}(H, W) = V(H, W)$ gilt. Damit kann zur Lösung des Optimierungsproblem (2) das weitaus günstigere Problem (3) betrachtet werden, denn es gilt $\mathcal{H} \subseteq \{1, \ldots, H\}$ und $\mathcal{W} \subseteq \{1, \ldots, W\}$. In den meisten Fällen gilt sogar $\subsetneq$.

5 Implementierung des Verfahrens

Gilmore und Gomory, Herz und auch Beasley haben rekrusive oder auch iterative Verfahren zur Lösung von (2) bzw. (3) eingeführt. In [CFYE07] ist ein iterativer Algorithmus vorgeschlagen, für den wir uns entschieden haben, da es bei größeren Problemen bei rekursivem Vorgehen z.B. zu Stack-Überläufen kommen kann. Außerdem wird in unseren Tests die iterative Implementerung deutlich schneller ausgeführt, als eine rekursive (analog zu Herzs Algorithmus, unter Einbeziehung der reduzierten Rasterpunktmengen und unteren, bzw. oberen Schranken ähnlich wie in [HZ95] bzw. [HZ96]).

Die Werte für $v(h, w)$ werden in einem ersten Schritt für alle h und w bestimmt. Dies geschieht in den Zeilen 2 bis 8 des Algorithmus 1. Dann werden ausgehend von den kleinsten Werten in $\mathcal{W}$ bzw. $\mathcal{H}$ alle maximalen Schnittmuster iterativ ermittelt. Dadurch, dass vom kleinsten zu den größeren Rechtecken iteriert wird, sind alle nötigen Vergleichswerte für kleinere Rechtecke (bei den Zerteilungen) schon bekannt und müssen nicht, wie bei der rekursiven Vorschrift, erst ermittelt werden.

Zur Beschreibung des Schnittmusters werden drei Matrizen mitgeführt, die die Schnittrichtung ($guillotine_{ij}$), die Schnittposition ausgehend von der linken unteren Ecke des jeweiligen Rechtecks (pos_{ij}) bzw. das in ein Rechteck zu platzierende Typenteil ($item_{ij}$) angeben. Diese dienen zur späteren Rekonstruktion und Visualisierung der Schnittmuster. Das entspricht dem 2. Schritt der dynamischen Programmierung nach Bellman, der rückwertigen Rekonstruktion der getroffenen optimalen Entscheidungen.

Im Algorithmus 1 ist der Pseudo-Code für unser Verfahren angegeben.

Algorithmus 1 Bestimmung eines optimalen Schnittmusters

Input: Grundplatte H,W und Typenteile h_i, w_i mit Kosten c_i

Output: maximale Belegung $V^* := V(H, W)$ und zur Beschreibung des Schnittmusters (item_{ij}), (guillotine_{ij}) und (pos_{ij})

1: Bestimme $\mathcal{H} = \{\mathcal{H}_1, \ldots, \mathcal{H}_r\}$ und $\mathcal{W} = \{\mathcal{W}_1, \ldots, \mathcal{W}_s\}$
2: **for** $i \leftarrow 1, r$ **do** ▷ Initialisierung von $V(h, w)$ mit $v(h, w)$
3: **for** $j \leftarrow 1, s$ **do**
4: $k^* := \text{argmin}_k\{c_k : h_k \leq \mathcal{H}_i, w_k \leq \mathcal{W}_j\}$
5: $V(i, j) \leftarrow c(k^*)$
6: $\text{item}(i, j) \leftarrow k^*$
7: **end for**
8: **end for**
9: **for** $i \leftarrow 2, r$ **do**
10: **for** $j \leftarrow 2, s$ **do**
11: $n \leftarrow \mu_{\mathcal{H}}(\frac{\mathcal{H}_i}{2})$ ▷ Horizontale Schnitte
12: **for** $x \leftarrow 1, n$ **do**
13: $t \leftarrow \mu_{\mathcal{H}}(\mathcal{H}_i - \mathcal{H}_x)$
14: **if** $V(i, j) < V(x, j) + V(t, j)$ **then**
15: $V(i, j) \leftarrow V(x, j) + V(t, j)$
16: $\text{pos}(i, j) \leftarrow \mathcal{H}_x$
17: $\text{gui}(i, j) \leftarrow \texttt{h}$
18: **end if**
19: **end for**
20: $n \leftarrow \mu_{\mathcal{W}}(\frac{\mathcal{W}_j}{2})$ ▷ Vertikale Schnitte
21: **for** $y \leftarrow 1, n$ **do**
22: $t \leftarrow \mu_{\mathcal{W}}(\mathcal{W}_j - \mathcal{W}_y)$
23: **if** $V(i, j) < V(i, y) + V(i, t)$ **then**
24: $V(i, j) \leftarrow V(i, y) + V(i, t)$
25: $\text{pos}(i, j) \leftarrow \mathcal{W}_y$
26: $\text{guillotine}(i, j) \leftarrow \texttt{v}$
27: **end if**
28: **end for**
29: **end for**
30: **end for**

Untersucht man die Struktur des Algorithmus genauer, fällt auf, dass 4 Iterations-Schleifen durchlaufen werden. Die ersten beiden Schleifen durchlaufen alle Elemente von $\mathcal{H}$ und $\mathcal{W}$ (Zeile 9,10), wodurch sich dafür ein Aufwand von $\mathcal{O}(|\mathcal{H}| \cdot |\mathcal{W}|)$ abschätzen lässt. Der Aufwand der Zeile 12 kann duch $\mathcal{O}(|\mathcal{H}|)$ beschränkt werden. Dies folgt daraus, dass in bestimmten kritschen Fällen fast die komplette Menge $\mathcal{H}$ betrachtet werden muss. Analoges gilt für Zeile 21, wonach sich eine Zeitkomplexität für das gesamte Verfahrens von $\mathcal{O}(|\mathcal{H}| \cdot |\mathcal{W}| \cdot (|\mathcal{H}| + |\mathcal{W}|))$ ergibt.

Im Algorithmus wird häufig Gebrauch von der Funktion $\mu(\cdot)$ gemacht, z.B. in den Zeilen 11, 13, 20 und 22. Das motiviert dazu diese Funktion möglichst effizient zu implementieren. Eine weitere Frage die noch bleibt ist, wie die Menge der Rasterpunkte bestimmt werden kann (Siehe Zeile 1). Diese Frage wird in 5.2 geklärt.

5.1 Die FindMax-Funktion

Im Programm-Quelltext haben wir die Funktion μ mit „FINDMAX“ bezeichnet, da sie das maximale Element in einer Menge findet, das gewisse Bedingungen erfüllt. Wir gehen hier davon aus, dass die Menge, in der gesucht werden soll (wir verwenden dafür die Bezeichnung $\mathcal{D}$) eine geordnete Liste von Elementen $\mathcal{D}_1 < \ldots < \mathcal{D}_n$ ist. Zum Suchen in einer solchen Liste verwenden wir ein Halbierungsverfahren (engl. binary search), so dass ein gesuchter Wert $d_{\max}$ in der Zeit $\mathcal{O}(\log n)$ gefunden werden kann. Da die Funktion FINDMAX $\mathcal{O}(n^3)$-mal aufgerufen wird, lohnt es sich die gefundenen Stellen zwischenzuspeichern. Zusätzlich gibt es meistens obere Schranken s für die zu suchenden Indizes. Das kann auch bei der Implementierung berücksichtigt werden:

Algorithmus 2 findMax-Funktion

```
1:  function FINDMAX(D, d_max, s)
2:      if maxList_D[d_max] ≠leer then
3:          return maxList_D[d_max]
4:      end if
5:      Interval ← [1, s]
6:      repeat
7:          i ← ⌊½(Interval_1 + Interval_2)⌋
8:          if D_i ≤ d_max then
9:              if i + 1 > s oder D_{i+1} > d_max then
10:                 maxList_D[d_max] ← i
11:                 return i
12:             else
13:                 Interval ← [i + 1, rechts]
14:             end if
15:         else
16:             Interval ← [links, i − 1]
17:         end if
18:     until i ≤ 1 oder i ≥ n
19: end function
```

Eine Alternative zur FINDMAX-Funktion ist die Vorbelegung eines Feldes ($findMaxD_i$) mit den Werten $\mu_{\mathcal{D}}(i)$, für $\mathcal{D} = \mathcal{H}$ und $\mathcal{D} = \mathcal{W}$. Dazu kann die Vorschrift

$findMaxD(1 : \mathcal{D}_1 - 1) \leftarrow 0$
for $i \leftarrow 1, n-1$ **do**
  $findMaxD(\mathcal{D}_i : \mathcal{D}_{i+1} - 1) \leftarrow i$
end for
$findMaxD(\mathcal{D}_n : end) \leftarrow n$

genutzt werden. Je nach Optimierungsstrategien des verwendeten Compilers kann diese Vorschrift besser, oder schlechter sein, als die FINDMAX-Implementierung. In Fortran war dann ein Unterschied in der Laufzeit festzustellen, wenn die Funktion FINDMAX nicht direkt ins Hauptprogramm geschrieben wurde, sondern als externe Funktion implementiert wurde. Dann war die obige Vorschrift ein wenig schneller als die externe Funktion.

Die Initialisierung dieses Feldes kann in einer Zeit $\mathcal{O}(n)$ geschehen. Danach

ist das Auslesen von FINDMAX(x) in konstanter Zeit $\mathcal{O}(1)$ möglich.

5.2 Bestimmung der Rasterpunktmenge

Zur Bestimmung der Menge von Rasterpunkten für ein Intervall $[0, L]$ und eine Menge von Längen $\{l_1, \ldots, l_m\}$ werden in [CFYE07] zwei verschiedene Möglichkeiten aufgezeigt. Einmal mittels Durchnummerierung (DEE discretization by explicit enumeration, Aufwand $\mathcal{O}(m\delta)$ mit δ ist die Anzahl der Kegelkombination der l_i) und zum Anderen durch die Lösung eines einfachen Rucksackproblem (DDP discretization using dynamic programming, Aufwand $\mathcal{O}(m \cdot L)$). Der erste der beiden Algorithmen hat den Vorteil unabhängig von der Skalierung der Grundplatte zu sein. Die Autoren haben sich in ihrer Arbeit allerdings nicht mit der reduzierten Menge von Rasterpunkten beschäftigt. Scheithauer hat in [STL87] einen Algorithmus vorgeschlagen, der iterativ jeden Rasterpunkt durch Auflisten von Hilfspunkten ermittelt.

Die Zeit, die zur Bestimmung der Rasterpunkte benötigt wird, fällt im Vergleich zur Hauptrekursion nicht weiter ins Gewicht (siehe unten). Deswegen ist die Auswahl des Algorithmus willkürlich und wir haben uns für einen modifizierten Algorithmus von Scheithauer entschieden (Algorithmus 3). Dieser erzeugt auch keine reduzierte Menge von Rasterpunkten. Durch ein Durchlaufen aller erzeugten Rasterpunkte und Anwendung der oben eingeführten FINDMAX-Funktion lässt sich diese Menge jedoch, wie in ihrer Definition, weiter reduzieren. Dazu haben wir eine weitere Funktion implementiert, die sich in Algorithmus 4 wiederfindet.

Algorithmus 3 Bestimmung der Menge der Rasterpunkte

```
1:  function RASTERPUNKTE(Länge L, Teillängen l = {l_1, ..., l_m})
2:      l_min ← min_i{l_i}
3:      D_1 ← 0
4:      j ← 1
5:      repeat
6:          for i ← 1, m do
7:              k ← D_j + l_i
8:              h(k) ← k                                  ▷ Hilfspunkte
9:          end for
10:         for i ← D_j + 1, L do
11:             if h(i) ≠leer then
12:                 D_{j+1} ← h(i)
13:                 exit for
14:             end if
15:         end for
16:         j ← j + 1
17:     until D_j > L − l_min
18:     D_j ← L
19:     return D = (D_1, ..., D_j)
20: end function
```

Für den Algorithmus 4 setzen wir voraus, dass $\mathcal{D}_1 < \ldots < \mathcal{D}_m$ und $L \geq \mathcal{D}_m$ gilt. Die Rasterpunktmenge wird durch die Funktion RASTERPUNKTE auch so erzeugt, dass ein Hintereinanderausführen der beiden Funktionen zum gewünschten

Resultat führt. Die Implementierung muss zudem darauf achten, dass die Ordnung der Rasterpunkte durch die Reduzierung nicht verloren geht.

Algorithmus 4 Menge der Rasterpunkte reduzieren

```
1: function REDUCEDISCRETIZATION(Länge L, D = {D_1, ..., D_m})
2:     k ← m
3:     D^red ← ∅
4:     for i ← 1, m do
5:         k ← FINDMAX(D, L − D_i, k)
6:         D^red ← D^red ∪ {k}
7:     end for
8:     return D^red
9: end function
```

6 Der mehr-stufige Zuschnitt

In einigen Anwendungsfällen ist es nötig weitere Einschränkungen an die Art der Schnitte zu machen. So verlangen die k-stufigen Zuschnittprobleme, dass in jeder Stufe des Algorithmus nur Schnitte in einer Richtung ausgeführt werden und in der jeweils darauffolgenden Stufe nur Schnitte in die andere Richtung. Dabei vereinbart man[6], dass bei geradem k horizontal und bei ungeradem k vertikal begonnen wird.

Es wird zwischen exaktem und unexaktem Zuschnitt unterschieden. Der *exakte Zuschnitt* setzt zusätzlich voraus, dass nur Typenteile in ein Rechteck platziert werden dürfen, die die volle Breite (bei vertikalem) bzw. Höhe (bei horizontalem Zuschnitt) des Rechtecks ausnutzen. Beim *unexakten Zuschnitt* gilt diese Einschränkung nicht.

Um das Problem mathematisch zu formulieren werden wir die Zielfunktion V zusätzlich um zwei Parameter erweitern: $V(h, w, k, \delta)$ wobei h und w die Höhe bzw. Breite angibt, der Parameter $k \geq 0$ die Stufennummer und $\delta \in \{\mathtt{v}, \mathtt{h}\}$ die Richtung, in der geschnitten werden soll. Das Optimierungsproblem (3) wird nun umformuliert zu:

$$
\begin{aligned}
&V(h, w, 0, \mathtt{v} \text{ oder } \mathtt{h}) := v_{\mathtt{v}/\mathtt{h}}(h, w) \\
&V(h, w, k, \mathtt{v}) := \max_{x \in \mathcal{W}, x \leq \frac{w}{2}} \begin{cases} V(h, w, k-1, \mathtt{h}), \\ V(h, x, k-1, \mathtt{h}) + V(h, \mu_{\mathcal{W}}(w - x), k, \mathtt{v}) \end{cases} \\
&V(h, w, k, \mathtt{h}) := \max_{y \in \mathcal{H}, y \leq \frac{h}{2}} \begin{cases} V(h, w, k-1, \mathtt{v}), \\ V(y, w, k-1, \mathtt{v}) + V(\mu_{\mathcal{H}}(h - y), w, k, \mathtt{h}) \end{cases}
\end{aligned} \tag{4}
$$

Der Algorithmus ändert sich dahingehend, dass die Hauptrekursion k-mal durchlaufen wird, wobei im i-ten Durchlauf nur über den horizontalen bzw. vertikalen Teil optimiert wird (je nachdem, ob k gerade oder ungerade ist).

Zur Implementierung des exaktem Zuschnitts wird die initialisierende Schleife in Zeile 7 modifiziert und anstelle von $\leq$ die Gleichheit verlangt:

$$v_{\mathtt{v}}(h, w) := \max_i \{c_i : h_i = h, w_i \leq w\} \quad \text{bzw.} \quad v_{\mathtt{h}}(h, w) := \max_i \{c_i : h_i \leq h, w_i = w\}$$

[6] [CFYE07]

Damit haben die zu platzierenden Teile bei vertikalem/horizontalem Zuschnitt genau die gewünschte Breite/Höhe. Für den unexakten Zuschnitt gilt weiterhin $v_\delta(h,w) := v(h,w)$ wie im freien Zuschnitt.

Algorithmus 5 k-Stufiger Zuschnitt

Input: Grundplatte H,W und Typenteile h_i, w_i mit Kosten c_i, Anzahl der Stufen k

Output: maximale Belegung $V^* := V(H,W,k)$ und zur Beschreibung des Schnittmusters $(\text{item})_{ij}$, $(\text{guillotine})_{ijk}$ und $(\text{pos})_{ijk}$

1: Bestimme $\mathcal{H} = \{\mathcal{H}_1, \ldots, \mathcal{H}_r\}$ und $\mathcal{W} = \{\mathcal{W}_1, \ldots, \mathcal{W}_s\}$
2: **if** k ist gerade **then**
3: $\quad\delta \leftarrow$ `h`
4: **else**
5: $\quad\delta \leftarrow$ `v`
6: **end if**
7: initialisiere $V(h,w,0) \leftarrow v_\delta(h,w)\ \forall h,w$ ▷ Siehe Alg.1 Zeilen 2-8
8: **for** $l \leftarrow 1,k$ **do**
9: $\quad$**for** $i \leftarrow 2,r$ **do**
10: $\quad\quad$**for** $j \leftarrow 2,s$ **do**
11: $\quad\quad\quad$**if** $\delta =$ `h` **then** ▷ Horizontale Schnitte
12: $\quad\quad\quad\quad n \leftarrow \mu_{\mathcal{H}}(\frac{\mathcal{H}_i}{2})$
13: $\quad\quad\quad\quad$**for** $x \leftarrow 1,n$ **do**
14: $\quad\quad\quad\quad\quad t \leftarrow \mu_{\mathcal{H}}(\mathcal{H}_i - \mathcal{H}_x)$
15: $\quad\quad\quad\quad\quad$**if** $V(i,j,l) < V(x,j,l-1) + V(t,j,l)$ **then**
16: $\quad\quad\quad\quad\quad\quad V(i,j,l) \leftarrow V(x,j,l-1) + V(t,j,l)$
17: $\quad\quad\quad\quad\quad\quad \text{pos}(i,j,l) \leftarrow \mathcal{H}_x$
18: $\quad\quad\quad\quad\quad\quad \text{guillotine}(i,j,l) \leftarrow$ `h`
19: $\quad\quad\quad\quad\quad$**end if**
20: $\quad\quad\quad\quad$**end for**
21: $\quad\quad\quad$**else** ▷ Vertikale Schnitte
22: $\quad\quad\quad\quad n \leftarrow \mu_{\mathcal{W}}(\frac{\mathcal{W}_j}{2})$
23: $\quad\quad\quad\quad$**for** $y \leftarrow 1,n$ **do**
24: $\quad\quad\quad\quad\quad t \leftarrow \mu_{\mathcal{W}}(\mathcal{W}_j - \mathcal{W}_y)$
25: $\quad\quad\quad\quad\quad$**if** $V(i,j,l) < V(i,y,l-1) + V(i,t,l)$ **then**
26: $\quad\quad\quad\quad\quad\quad V(i,j,l) \leftarrow V(i,y,l-1) + V(i,t,l)$
27: $\quad\quad\quad\quad\quad\quad \text{pos}(i,j,l) \leftarrow \mathcal{W}_y$
28: $\quad\quad\quad\quad\quad\quad \text{guillotine}(i,j,l) \leftarrow$ `v`
29: $\quad\quad\quad\quad\quad$**end if**
30: $\quad\quad\quad\quad$**end for**
31: $\quad\quad\quad$**end if**
32: $\quad\quad$**end for**
33: $\quad$**end for**
34: $\quad$Wechsle die Richtung δ
35: **end for**

Die Struktur dieses Algorithmus unterscheidet sich nicht wesentlich von der des freien Zuschnitts, wodurch sich eine analoge Komplexitätsbetrachtung ergibt. Mit der Vereinbarung der Start-Schnittrichtung werden in der letzten Stufe

immmer vertikale Schnitte ausgeführt. Aus dem Schnittrichtungswechsel ergibt sich somit folgender Aufwand: $\mathcal{O}(|\mathcal{H}| \cdot |\mathcal{W}| \cdot (\lfloor \frac{k}{2} \rfloor |\mathcal{H}| + \lceil \frac{k}{2} \rceil |\mathcal{W}|))$.

Da der exakte Zuschnitt eine sehr starke Einschränkung des allgemeinen Verfahrens ist, müssen weitere Änderungen an dem Algorihmus vorgenommen werden. Es reicht nicht mehr aus die Menge der reduzierten Rasterpunkte für die Schnittpositionen und Rechtecksgrößen zu betrachten, da auch Schnitte nahe am rechten/oberen Rand nötig sind, um ein Typenteil exakt auszuschneiden. Um das zu sehen, genügt es z.B. eine Grundplatte der Dimension 4×4 und ein Typenteil der Größe 3×3 zu betrachten. Die Rasterpunktmenge der Breite (und auch der Höhe) sind damit gegeben durch $\{0, 3, 4\}$, die reduzierte Rasterpunkmenge allerdings $\{0, 4\}$, d.h. der exakte Zuschnitt der Breite (Höhe) 3 ist nicht möglich.

Das bedeutet in Zeile 1 muss die Definition abgändert werden zu

$$\mathcal{H} := D(\underline{h}, H) \cup \{H\} \quad \text{und} \quad \mathcal{W} := D(\underline{w}, W) \cup \{W\}$$

Weiterhin genügt es nicht Schnittpositionen bis zur Hälfte der Rechtecksbreite zu betrachten, sodass sich die Zeilen 12 und 22 wie folgt ändern: $n \leftarrow (i-1)$ bzw. $n \leftarrow (j-1)$.

Offensichtlich löst dann dieser Algorithmus das Problem des exakten Zuschnitts, aber nicht unbedingt mehr in optimaler Zeit. Es wird nicht auf die spezielle Struktur der Lösungen eingegangen, sondern das Standardverfahren so angepasst, dass auch Lösungen für den exakten Zuschnitt gefunden werden. Deshalb macht es Sinn die Struktur der Lösung des exakten Zuschnitts näher zu betrachten:

6.1 Exakter Zuschnitt

In Abbildung 2 findet sich ein Zuschnittmuster für den exakten Zuschnitt. Auffallend ist, dass mehrere (vertikale) Streifen, deren Breite komplett mit Typenteilen ausgefüllt ist, nebeneinander angordnet sind. Für die Breite der Streifen kommen nur Werte in Frage, die der Breite eines Typenteils entsprechen, da innerhalb eines Streifens kein Verschnitt auftreten darf. Sei $\underline{w} := (w_i)_{i \in I}$ der Vektor der Typenteilbreiten und I die Indizes aller Typenteile, dann definieren wir

$$I_{\mathtt{w}} \subseteq I: \quad w_i \neq w_j \ \forall i, j \in I_{\mathtt{w}}, i \neq j \quad \text{und} \quad \underline{w}^* := (w_i)_{i \in I_{\mathtt{w}}}$$

den Vektor der unterschiedlichen Typenteilbreiten. Analog wird auch $\underline{h}^*$ definiert mit der Indexmenge $I_{\mathtt{h}}$ als der Vektor der unterschiedlichen Typenteilhöhen.

Einen Streifen, den Typenteile in der Breite komplett ausfüllen, können wir als 1-dimensional betrachten, indem die Breite ignoriert und nur über die Höhe der Typenteile optimiert wird. Für alle möglichen Typenteilbreiten kann folglich ein einfaches 1-dimensionales Rucksackproblem gelöst werden, wobei nur Typenteile zu betrachten sind, die die exakte Breite des Streifens haben. Für jeden Streifen erhalten wir so einen optimalen Wert V_i $(i \in I_{\mathtt{w}})$.

In einem zweiten Schritt muss die optimale Anordnung von Streifen in der Grundplatte betrachtet werden. Dies führt wieder auf ein 1-dimensionales Rucksackproblem, da die Höhe der Streifen nicht beachtet werden braucht.

Zusammenfassend lässt sich ein Lösungsverfahren für den exakten 2-stufigen Zuschnitt angeben (wenn wir annehmen, dass zuerst vertikal geschnitten wird):

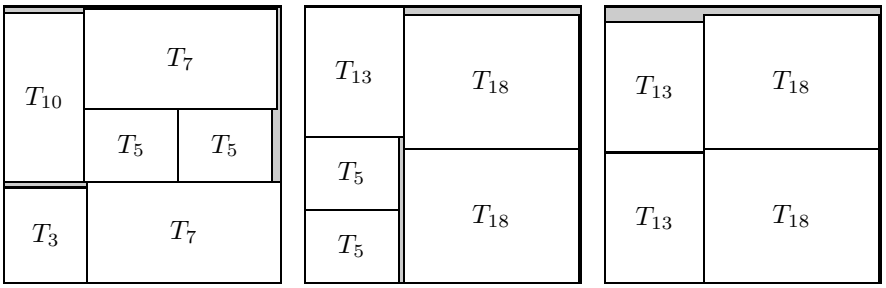

Fig. 2: *Links*: Freier Zuschnitt der Instanz *gcut2*, *Mitte*: Unexakter 2-stufiger Zuschnitt, *Rechts*: Exakter 2-stufiger Zuschnitt

Algorithmus 6 Exakter 2-stufiger Zuschnitt

1: Bestimme I_w (siehe oben)
2: **for** $i \in I_w$ **do**
3: Bestimme $I^i \subseteq I$ so, dass $w_j = w_i \; \forall j \in I^i$
4: $V_i \leftarrow$ Lösung des 1D Rucksackproblems der Instanz $(H, (h_i)_{i \in I^i}, (c_i)_{i \in I^i})$
5: **end for**
6: Löse das Rucksackproblem der Instanz $(W, (w_i)_{i \in I_w}, (V_i)_{i \in I_w})$

Durch Modifikation der Zeile 3 zu $w_j \leq w_i$ lässt sich mit diesem Vorgehen auch der unexakte Zuschnitt bearbeiten.

Zur Lösung der 1-dimensionalen Rucksackprobleme können wieder Rasterpunktmengen verwendet werden und ein Algorithmus ähnlich dem Algorithmus 1, wobei nur über eine Dimension (und somit auch nur über eine Schnittrichtung) iteriert wird, angesetzt werden. Da die Menge I^i in Zeile 3 des Algorithmus 6 i.A. sehr viel kleiner ist als I, reduziert sich die, in jedem Schritt neu zu berechnende, Rasterpunktmenge auf sehr wenige Punkte, insbesondere wenn wir annehmen, dass die Menge der Typenteile stark heterogen ist.

Betrachtet man die Komplexität dieses Algorithmus so werden $|I_w|$ Rucksackprobleme gelöst. Jedes Problem besitzt eine Komplixität von $\mathcal{O}(|I^i| \cdot |\mathcal{H}^i|)$ (wobei $|\mathcal{H}^i|$ die Anzahl der Rasterpunkte ist, die durch die Längen $(h_i)_{i \in I^i}$ im i-ten Schritt erzeugt werden, es gilt meistens $|\mathcal{H}^i| \ll H$), wodurch sich zusammen mit dem in Zeile 6 zu lösenden Problem ein Zeitaufwand von $\mathcal{O}(\sum_{i \in |I_w|} |I^i| \cdot |\mathcal{H}^i| + |I_w| \cdot |\mathcal{W}|)$ ergibt.

In Tabelle 3 (siehe Abschnitt 7) kann man die Auswirkung dieser starken Reduktion der Rasterpunktmengen und damit auch der Komplexität ablesen.

7 Numerische Testresultate

Die Algorithmen und Funktionen wurden in Fortran95 implementiert und mittels dem Compiler von Intel (Ver. 10.1) erstellt. Als Testinstanzen dienen die Probleme *gcut1* bis *gcut13*, die von Beasley formuliert wurden, aus der OR-Library[7]. Zusätzlich haben wir, analog zu [CFYE07] die Probleme um weitere vier (*gcut14*-*gcut17*) erweitert, um schwierigere Instanzen zu bekommen, für die

[7] http://people.brunel.ac.uk/mastjjb/jeb/info.html

ein Laufzeitvergleich erst möglich ist. Dazu wurde die Instanz *gcut13* um die Typenteile aus den Instanzen 9-12 erweitert und die Dimension der Grundplatte auf 3500×3500 vergrößert. Die Instanzen *gcut18-gcut20* sind auf spezielle Situationen zugeschnitten. Die Instanz 18 besitzt besonders kleine Typenteile im Verhältnis zur Grundplatte, für die 19 wurde die Anzahl der Typenteile stark vergrößert und in der Instanz 20 bekommt die Grundplatte eine sehr große Ausdehnung.

Die Testergebnisse in Tabelle 1 listen die Daten der einzelnen Instanzen auf, zusätzlich die Anzahl der Rasterpunkte, die für die Höhe bzw. Breite der Grundplatte erzeugt wurden, die optimale Belegung der Platte und die Laufzeit, auf unserem Testrechner. Die Tests wurden auf einem Intel Pentium D 6400 mit 1024 MB Arbeitsspeicher durchgeführt. Dabei haben wir auf Parallelisierung des Algorithmus verzichtet, so dass effektiv nur die „halbe“ Kapazität des Prozessors genutzt werden konnte.

Tab. 1: Testresultate für freien Zuschnitt

Instanz	**#Teile**	**Dimension**	$\mathcal{H}$	$\mathcal{W}$	**opt. Lösung**	**Zeit (sec)**
gcut1	10	250×250	13	5	56460	0.000
gcut2	20	250×250	17	24	60536	0.000
gcut3	30	250×250	44	26	61036	0.000
gcut4	50	250×250	45	50	61698	0.004
gcut5	10	500×500	10	13	246000	0.000
gcut6	20	500×500	12	18	238998	0.000
gcut7	30	500×500	23	19	242567	0.000
gcut8	50	500×500	44	59	246633	0.004
gcut9	10	1000×1000	15	7	971100	0.000
gcut10	20	1000×1000	14	20	982025	0.000
gcut11	30	1000×1000	20	38	980096	0.000
gcut12	50	1000×1000	49	42	979986	0.004
gcut13	32	3000×3000	647	1849	8997780	5.304
gcut14	42	3500×3500	1861	2451	12245410	45.375
gcut15	52	3500×3500	1879	2595	12246032	54.735
gcut16	62	3500×3500	2147	2615	12248836	67.596
gcut17	82	3500×3500	2357	2636	12248892	80.361
gcut18	30	2500×2500	2233	2235	8935894	65.036
gcut19	200	2500×2500	2063	2073	6250000	60.560
gcut20	50	5000×5000	4449	4411	25000000	508.364

Nachfolgend sind die Testresultate des zwei- und drei-stufigen Zuschnitts aufgelistet, wobei die Laufzeiten mit dem Algorithmus 5 erzeugt sind. Dabei werden exakter und unexakter Zuschnitt getrennt betrachtet. Die Daten der Testinstanzen können aus der obigen Tabelle des freien Zuschnitts übernommen werden.

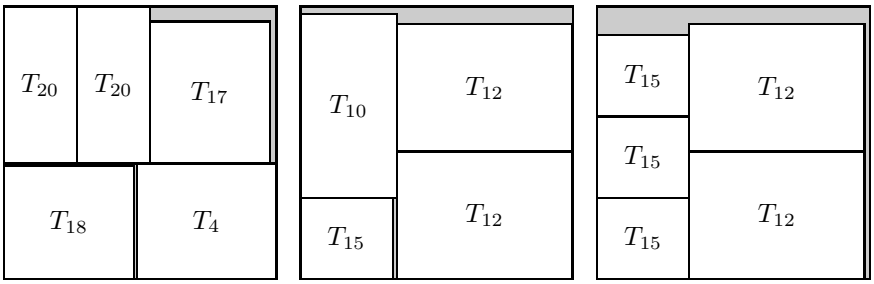

Fig. 3: *Links*: Freier Zuschnitt der Instanz *gcut6*, *Mitte*: Unexakter 2-stufiger Zuschnitt, *Rechts*: Exakter 2-stufiger Zuschnitt

Tab. 2: Testresultate für 2- bzw. 3-stufigen Zuschnitt

Instanz	**Exakt**	**Zeit (sec)**	**Unexakt**	**Zeit (sec)**
2-stufiger Zuschnitt				
gcut1	56460	0.000	56460	0.000
gcut2	59476	0.000	60076	0.000
gcut3	56747	0.004	60133	0.000
gcut4	61698	0.008	61698	0.004
gcut5	246000	0.000	246000	0.000
gcut6	225771	0.004	235058	0.000
gcut7	230620	0.000	242567	0.000
gcut8	237252	0.020	245758	0.004
gcut9	971100	0.000	971100	0.000
gcut10	934548	0.000	982025	0.000
gcut11	960148	0.008	974638	0.000
gcut12	945226	0.028	977768	0.004
gcut13	8905900	76.429	8906216	7.517
gcut14	12036138	212.269	12216788	56.168
gcut15	12179271	230.278	12215614	66.965
gcut16	12158944	248.899	12210837	84.305
gcut17	12224718	269.121	12232948	101.33
gcut18	8890385	145.469	8892246	81.493
gcut19	6249318	121.863	6249619	72.837
3-stufiger Zuschnitt				
gcut1	56460	0.000	56460	0.008
gcut2	60536	0.004	60536	0.000
gcut3	61036	0.004	61036	0.000
gcut4	61698	0.012	61698	0.004
gcut5	246000	0.000	246000	0.000
gcut6	235058	0.000	238998	0.004
gcut7	242567	0.004	242567	0.000
gcut8	245758	0.028	245758	0.004
gcut9	971100	0.000	971100	0.000
gcut10	982025	0.000	982025	0.004
gcut11	974638	0.012	980096	0.000

Fortsetzung nächste Seite

Forts.

Instanz	Exakt	Zeit (sec)	Unexakt	Zeit (sec)
gcut12	979986	0.036	979986	0.004
gcut13	8997780	136.801	8997780	9.921
gcut14	12235770	380.928	12239634	75.617
gcut15	12238630	411.774	12239904	90.63
gcut16	12242500	444.652	12243100	118.775
gcut17	12244750	476.526	12245656	149.705
gcut18	8934994	260.292	8935894	121.324
gcut19	6250000	210.349	6250000	102.338

Vergleicht man die optimalen Zuschnittwerte von 2-stufigem und freiem Zuschnitt, fällt auf, dass der 2-stufige Zuschnitt zwar etwas schlechtere Werte liefert, diese sich aber bei größeren Instanzen immer mehr dem freien Zuschnitt annähern. Dabei ist klar, dass der exakte Zuschnitt schlechtere Werte als der unexakte Zuschnitt liefert, da eine weitere Einschränkung an die möglichen Schnittmuster gestellt wurde. Abbildung 4 verdeutlicht das Verhältnis zwischen 2-stufigem und freiem Zuschnitt. Dabei bezeichnen $V_{...}$ die jeweiligen Werte der optimalen Schnittmuster.

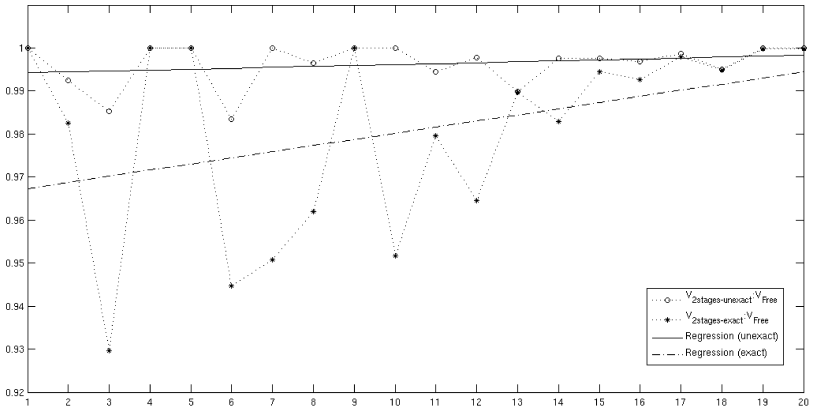

Fig. 4: Visualisierung des Verhältnis der Optimalwerte von 2-stufigem zu freiem Zuschnitt

Aus dem Algorithmus vom freien Zuschnitt geht eine theoretische Zeitkomplexität von $\mathcal{O}(|\mathcal{H}| \cdot |\mathcal{W}| \cdot (|\mathcal{H}| + |\mathcal{W}|))$ hervor. Daran ist schon ersichtlich, dass bei den Problemen mit großen Rasterpunktmengen die Laufzeit proportional zur dritten Potenz ansteigt. Dabei ist nicht unbedingt entscheident, wieviele verschiedene Typenteile produziert werden können, sondern die Struktur der Längen und Breiten, die dann zu den Rasterpunktmengen führen. So hat die Instanz *gcut19* viel mehr Typenteile als die Instanz *gcut18* und doch ist die Anzahl der Rasterpunkte etwas kleiner.

Der Vergleich der Laufzeit von freiem und mehr-stufigem Zuschnitt lässt auf eine Proportionalität zur Anzahl der Stufen schließen. Sei T_{free} die Laufzeit für eine Instanz mittels freiem Zuschnitt und $T_{\text{k-stages}}$ die entsprechende Laufzeit

mit der Einschränkung der k-Stufigkeit des Schnittmusters. Dann wird die folgende Beziehung vermutet:

$$T_{\text{k-stages}} \simeq \frac{2k}{3} T_{\text{free}}$$

Durch Anwendung des in Abschnitt 6.1 beschriebenen alternativen Algorith-

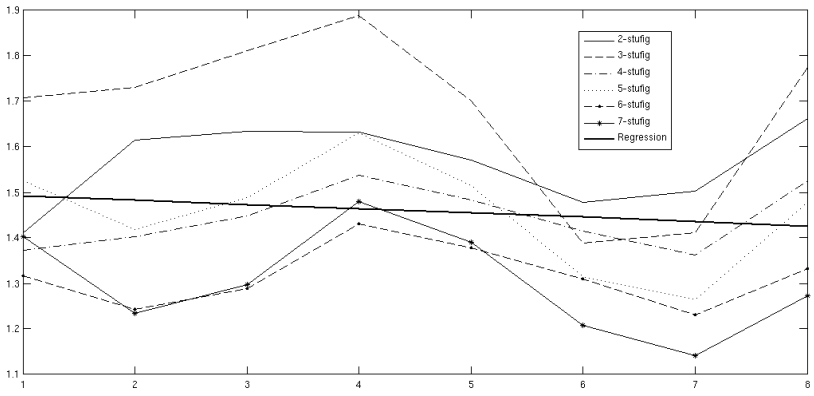

Fig. 5: Visualisierung des Verhältnis $\frac{k \cdot T_{\text{free}}}{T_{\text{k-stages}}}$ für verschiedene Instanzen des Guillotine-Problems. Der Mittelwert ergibt 1.46

mus zur Bestimmung des optimalen 2-stufigen Zuschnitts lassen sich die Laufzeiten stark reduzieren. Es wird ersichtlich, dass für solche Probleme die spezielle Struktur einer Lösung ausgenutzt werden sollte, um dementsprechend den Algorithmus aufzubauen.

Tab. 3: Laufzeiten in Sekunden für den 2-stufigen (un)exakten Zuschnitt mit alternativem Algorithmus 6

Instanz	**exakt**	**unexakt**	**Instanz**	**exakt**	**unexakt**
gcut1	0.000	0.000	gcut11	0.000	0.000
gcut2	0.000	0.000	gcut12	0.000	0.004
gcut3	0.000	0.000	gcut13	0.004	0.028
gcut4	0.000	0.004	gcut14	0.008	0.128
gcut5	0.000	0.000	gcut15	0.012	0.132
gcut6	0.000	0.000	gcut16	0.012	0.220
gcut7	0.000	0.000	gcut17	0.016	0.356
gcut8	0.000	0.004	gcut18	0.008	0.116
gcut9	0.000	0.000	gcut19	0.016	1.124
gcut10	0.000	0.000	gcut20	0.024	0.788

Literatur

[BD62] Richard E. Bellman and Stuart E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

[CFYE07] G.F. Cintra, F.K.Miyazawa, Y.Wakabayashi, and E.C.Xavier. Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation, 2007.

[Dre02] Stuart Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research INFORMS*, 50(1):48–51, 2002.

[Her72] J.C. Herz. Recursive computational procedure for two-dimensional stock cutting, 1972.

[HZ95] M. Hifi and V. Zissimopoulos. An approximation algorithm for solving unconstrained two-dimensional knapsack problems. *European Journal of Operational Research*, 84:618–632, 1995.

[HZ96] M. Hifi and V. Zissimopoulos. A recursive exact algorithm for weighted two-dimensional cutting. *European Journal of Operational Research*, 91:553–564, 1996.

[Sch08] Guntram Scheithauer. *Zuschnitt- und Packungsoptimierung*. Verlag: Vieweg+Teubner, 2008.

[STL87] Guntram Scheithauer, Johannes Terno, and Rudolf Lindemann. *Zuschnittprobleme und ihre praktische Lösung*. Verlag: VEB Fachbuchverlag Leipzig, 1987.

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