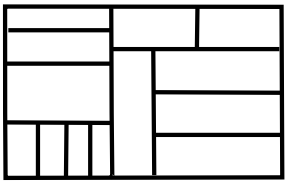
TECHNISCHE UNIVERSITÄT DRESDEN

2D-Guillotine-Zuschnitt Wang-Algorithmus

Francesco Kriegel & Matthias Lange



TU Dresden
Fakultät Mathematik

WS 2008 / 2009

9. Februar 2009

Inhaltsverzeichnis

Vorwort

Im Rahmen des mathematischen Praktikums während des Mathematikstudiums an der TU Dresden wurden uns, Francesco Kriegel und Matthias Lange, folgende Aufgabe gestellt:

2D Guillotine-Zuschnittproblem
mit oberen Schranken

Betrachtet wird das zweidimensionale Guillotine-Zuschnittproblem. Gegeben sind m Typen von Rechtecken $R_i = (w_i; h_i)$, $i = 1,\ldots,m$, und für jeden Typ R_i gibt es u_i Stücke von Rechtecken. Gesucht ist eine Teilmenge der Rechtecke, die aus einer Platte $(W; H)$ zuschneidbar ist, so dass der Abfall minimal ist und die Teile durch *Guillotine-Schnitte* ermittelbar sind, d.h. nur vertikale und horizontale Schnitte von einer Kante zur anderen.

Für das Problem soll der Algorithmus von Wang angewendet werden inklusive der Einschränkungen des Suchraumes. Numerische Experimente sind auszuführen.

Literatur

[1] G. Scheithauer. *Zuschnitt- und Packungsoptimierung: Problemstellungen, Modellierungstechniken, Lösungsmethoden*. 2008.

[2] P.Y. Wang. Two algorithms for constrained two-dimensional cutting stock problems. *Operational Research*, 31:573-586, 1983.

Betreuer:
Marat Mesyagutov, Tel.: 46334186,
E-Mail: marat.mesyagutov@mailbox.tu-dresden.de

Eine mathematische Einführung des Problems und mathematische Lösungsmethoden werden von Matthias Lange und Francesco Kriegel dargestellt und die Implementierung in Java wird von Francesco Kriegel bearbeitet.

1 Grundlagen und Einführung

1.1 Packungs- und Zuschnittsprobleme

In der Praxis begegnen uns Packungs- und Zuschnittsprobleme, wenn wir

- Gegenstände in einen begrenzten Stauraum „möglichst gut“ einordnen wollen
- Aus einem Stück Material „möglichst gut“ ein paar Nutzgegenstände herausschneiden wollen

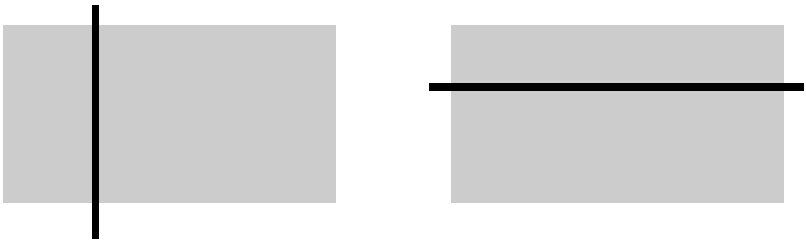
Das Problem stellt sich dabei im Zwei- wie im Dreidimensionalen, so könnte im Zweidimensionalen die Frage entstehen, wie man Stücke einer Metallplatte zuschneidet, im Dreidimensionalen ist es beispielsweise der Zuschnitt von Maschinenteilen aus Werkstoffklötzen oder das Beladen eines Transporters.
Wir beschränken uns hier auf die Betrachtung zweidimensionaler Probleme, im Folgenden verwenden wir die Bezeichnung „schneiden“ für die Festlegung eines Einzelteils. Da dieses „Zuschnittsproblem“ einem anderen Packungsproblem äquivalent ist, beschränken wir damit nicht die Menge der möglichen Fälle.
Unsere „Grundfläche“ ist ein Rechteck und unsere Einzelteile sind horizontal oder vertikal an diesem Rechteck ausgerichtete kleinere Rechtecke. Jedes Rechteck kann nur begrenzt oft aus dem großen Rechteck herausgeschnitten werden, und jedes herausgeschnittene Stück hat einen bestimmten „Wert“. Wir versuchen, Einzelteile mit einer möglichst großen „Wert-Gesamtsumme“ herauszuschneiden.
Wir geben zunächst eine halbformale Einführung in das Problem, angelehnt an [1]. In Kapitel 2 heben wir die für den Wang-Algorithmus notwendigen Definitionen explizit hervor.
Auf einem gegebenen Rechteck der Länge L und der Breite B sind kleinere Rechtecke von bestimmtem Typ anzuordnen. Der Rechtecktyp R_i

($i \in \{1, \ldots, m\}$) ist ein Tripel $R_i = (l_i, b_i, w_i)$, wobei l_i als dessen Länge, b_i als dessen Breite und w_i als dessen Wert interpretiert wird. Gesucht ist eine sich nicht überschneidende Anordnung der Rechtecke von maximaler Bewertung. Wieviele Stück welcher Rechtecktypen werden in welcher Weise angeordnet?

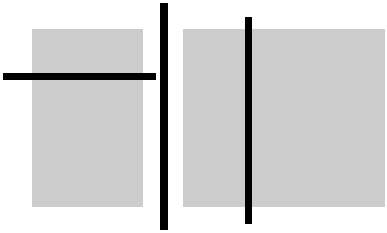
Bemerkung: Falls $w_i = l_i \cdot b_i$ gesetzt wird, interpretieren wir das als Zuschnittsproblem mit minimalem Abfall.

1.2 Guillotine-Anordnungen und Werte von Anordnungen

Es gibt einen Typ von Anordnungen, der in der Praxis oft leichter realisiert werden kann: Die Guillotine-Anordnung
Zur Begriffsklärung: Ein Guillotine-Schnitt zerlegt ein gegebenes Rechteck in zwei Einzelrechtecke

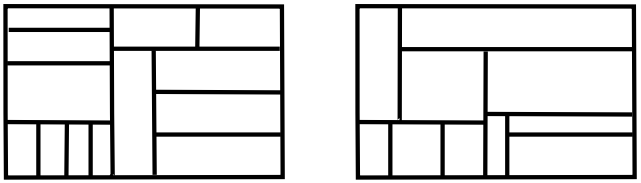


Jedes der beiden Teilrechtecke kann nun wieder guillotine-geschnitten werden:



Eine Guillotine-Anordnung ist nun eine Anordnung von Rechtecken, die durch mehrere Guillotine-Schnitte aus dem Rechteck entstanden ist. Hier

ein Beispiel für einen Guillotine-Zuschnitt und einen Nicht-Guillotine-Zuschnitt:



Sei nun $I = \{R_1, \ldots, R_m\}$ die Menge aller Rechtecktypen. Zur Vereinfachung der Darstellung fordern wir, dass die Rechtecke nicht gedreht werden dürfen. Wir erklären für jede Anordnung ein Tupel $a = (a_1, \ldots, a_m) \in \mathbb{N}^m$, wobei a_i die Anzahl der Rechtecke vom Typ i darstellt, die in dieser Anordnung vorkommen. Der Wert der Anordnung ist als $\sum_{i=1}^{m} a_i \cdot w_i$ erklärt.
In Abhängigkeit von der Problemstellung fordern wir mitunter, dass $a \leq w$ für ein Tupel $w = (w_1, \ldots, w_m) \in \mathbb{N}^m$, das die maximale Anzahl benötigter Rechtecke modelliert. Ein a, das diese Nebenbedingung erfüllt, heißt „zulässig“.

Definition 1.1

Sei $L' \leq L$,$B' \leq B$, dann definiere

$$v(L', B') := \max\{\sum_{i=1}^{m} a_i \cdot w_i | a = (a_1, \ldots, a_m) \textit{ zulässige Anordnung in } (L', B')\}$$

Bemerkung:
Wenn wir die maximalen Erträge bei vertikalem bzw horizontalem Zuschnitt mit

$$\begin{aligned} g(L', B') &:= \max\{v(r, B') + v(L' - r, B') | r \leq \tfrac{L'}{2}\} \\ h(L', B') &:= \max\{v(L', r) + v(L', B' - r) | r \leq \tfrac{B'}{2}\} \end{aligned}$$

erklären, so erhalten wir die Abschätzung

$$v(L', B') \geq \max\{g(L', B'), h(L', B')\}$$

Wir erkennen, dass sich damit rekursiv der maximal erreichbare Wert der möglichen Anordnungen berechnen lässt, wenn man die Werte der Einzelteile in der Berechnung berücksichtigt.

Bemerkung:
Mit

$$e(L', B') := \max\{0, \max\{w_i | l_i \leq L', b_i \leq B', i \in \{1, \ldots, m\}\}\}$$

können wir eine Abschätzung V des realen Maximalwerts v erklären:

$$V(L', B') := \max\{e(L', B'), g(L', B'), h(L', B')\} \leq v(L', B')$$

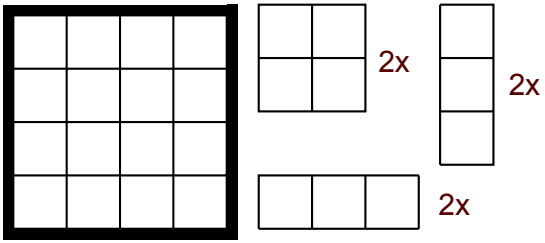
(wir ersetzen hierbei alle Vorkommen von v in g, h in der Rekursion durch bereits ermittelte Werte von V)
Die Überlegungen zur Abschätzung wollen wir hier nicht weiter vertiefen.

2 Formale Beschreibung des Problems und der Algorithmus von Wang

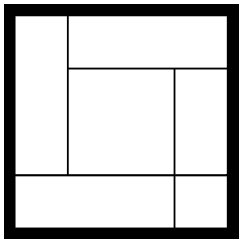
2.1 Beispiel

Wir verdeutlichen das Verfahren zunächst anhand eines Beispiels. Gegeben seien L und B sowie die einzelnen Rechtecktypen $R_i = (l_i, b_i, c_i, u_i)$ für $i \in I$. Wie in Kapitel 1 werden l_i als Länge, b_i als Breite und c_i als Wert interpretiert. Neu ist der Bezeichner $u_i \in \mathbb{N}$, der die maximale Anzahl von Rechtecken des Typs R_i angibt, die in der Anordnung vorkommen dürfen.

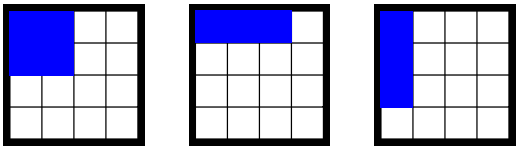
Wählen wir beispielsweise $L = B = 4$, $R_1 = (2, 2, 4, 2)$, $R_2 = (1, 3, 3, 2)$, $R_3 = (3, 1, 3, 2)$:



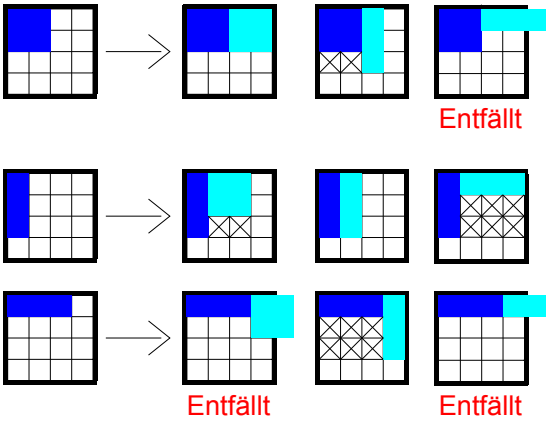
Die Werte haben die Form $c_i = l_i \cdot b_i$, was bedeutet, dass unser Problem auf die maximale Reduktion des Abfalls zurückzuführen ist. Sehen wir uns die Angelegenheit als klassisches Packungsproblem an, dann erkennen wir, dass wir dieses Optimum erreichen können:



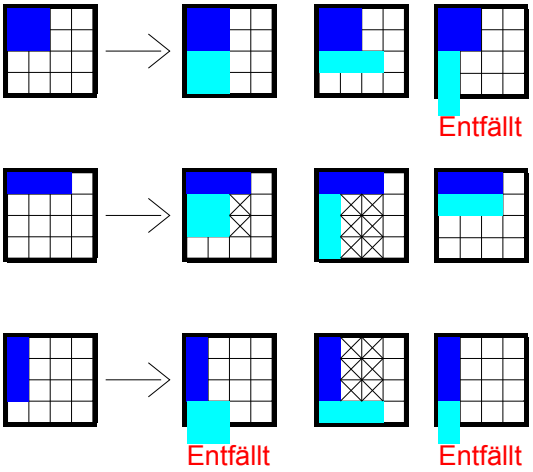
Hierbei handelt es sich aber nicht um eine Guillotine-Anordnung. Auf der Suche nach einer guten Guillotine-Anordnung wählen wir diesen Ansatz:



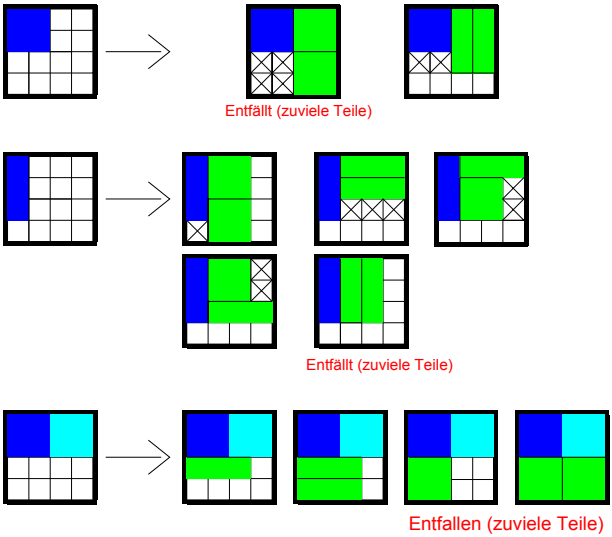
Die verfügbaren Teile werden in die Ecke links oben gesetzt. Wir fügen nun horizontal ein weiteres Teil an, und um die Vorgabe, dass wir eine Guillotine-Anordnung ermitteln, zu erfüllen, füllen wir die Anordnung zu einem Rechteck auf:

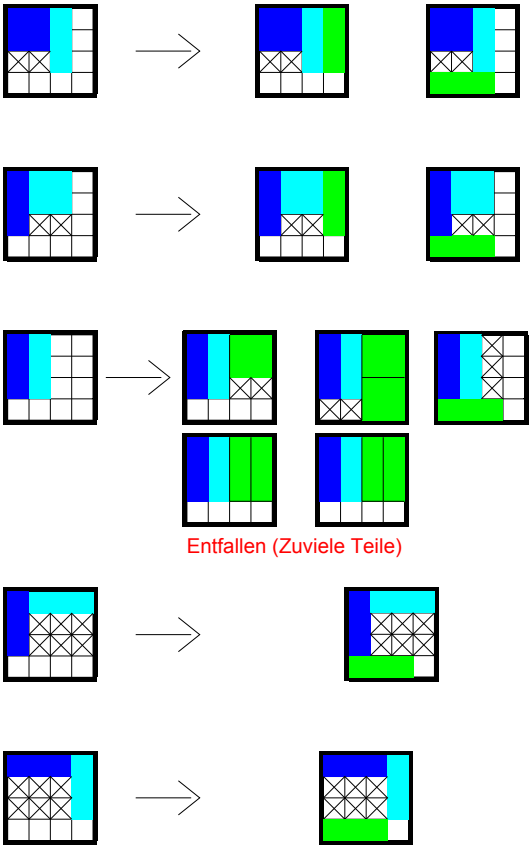


In manchen Anordnungen gehen ragt ein Rechteck über den Rand des Grundrechtecks hinaus, diese Lösungen werden direkt ausgeschlossen. Analog arbeiten für das vertikale Hinzufügen von Teilen:

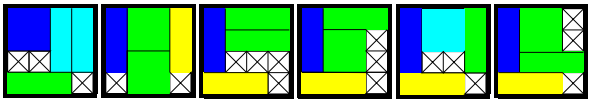


Wir fügen nun an jede mögliche Anordnung von zwei Teilen eine neue Anordnung von ein oder zwei Teilen nach den gleichen Regeln wie oben hinzu. Anordnungen, die über den Rand hinausragen werden wieder weggelassen. Da unser Problem gewisserweise symmetrisch ist (vertauschen wir Breiten und Längen, erhalten wir ebenfalls Lösungen), beschränken wir uns hier (in manchen Fällen) auf das horizontale Aneinanderfügen:

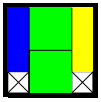




Manche Möglichkeiten scheiden hier aus, weil wir mehr als zwei Teile eines Typs verwenden. Nun fügen wir an jede erhaltene Anordnung aus dem letzten horizontal oder vertikal eine weitere mögliche Anordnung hinzu. Wir erhalten insgesamt noch diese Anordnungen:



Weitere neue Anordnungen (bis auf Spiegelung an der Diagonalen) finden wir nicht. Aus der Menge aller möglichen Anordnung wird eine mit höchstem Wert ausgewählt, es handelt sich dabei um:



2.2 Formale Beschreibung des Problems

Definition 2.1

Ein „Rechtecktyp“ ist ein Tupel $R = (l, b, w, m) \in \mathbb{N}^3 \times (\mathbb{N} \cup \{\infty\})$. Wir interpretieren dabei

- Den Wert l als „Länge“ von R
- Den Wert b als „Breite“ von R
- Den Wert w als „Wert“ von R
- Den Wert m als maximale Anzahl verfügbarer Rechtecke des Typs R

Definition 2.2

Ein „Packungsproblem“ ist ein Tripel $P = (L, B, \mathcal{R})$, wobei $L \in \mathbb{N}$ („Länge“), $B \in \mathbb{N}$ („Breite“) und $\mathcal{R}$ eine endliche Menge von Rechtecktypen ist.

Definition 2.3

1. Eine „Anordnung“ von Rechtecken des Typs $R_i = (l_i, b_i, w_i, m_i)$ ($i \in \{1, \ldots, n\}$) ist ein Tupel $A = (l, b, w, a)$, wobei $l, b, w \in \mathbb{N}$ als „Länge“,„Breite“ bzw „Wert“ interpretiert werden, und $a = (a_1, \ldots, a_n) \in \mathbb{N}^n$ als „Anzahlvektor“, der die Zahl a_i der in A verwendeten Rechtecke des Typs R_i darstellt. Die Menge aller Anordnungen wird mit $\mathbb{A}$ bezeichnet.
2. Die Anordnung $A = (l, b, w, a)$ von Rechtecken der Typen $\mathcal{R} = \{R_i | i \in \{1, \ldots, n\}\}$ heißt „zulässig“ im Packungsproblem $P = (L, B, \mathcal{R})$, wenn
   - $l \leq L$
   - $b \leq B$

- $a \leq m$, d.h. $\forall i \in \{1, \ldots, n\} : a_i \leq m_i$

Die Menge der zulässigen Anordnungen in P wird mit $\mathbb{A}_P$ bezeichnet.

Definition 2.4

Eine Anordnung $A = (l, b, w, a) \in \mathbb{A}_P$ heißt „Lösung" von P, wenn A maximalen Wert w bezüglich der Menge aller zulässigen Anordnungen aus A_P hat.

Bemerkung:

- Die Anordnung ist also algebraisch definiert worden. Eine geometrische Version fällt im nächsten Kapitel mit ab, wenn wir die Menge der zulässigen Anordnungen induktiv konstruieren.
- Realisiert wird das durch die Definition „primitiver Anordnungen" sowie zweier Operatoren auf $\mathbb{A}$, die das im Beispiel illustrierte horizontale und vertikale Zusammenfügen kleinerer Anordnungen modellieren. Durch eine Zulässigkeitsprüfung stellen wir dann fest, ob das Ergebnis wieder eine zulässige Anordnung ist.

2.3 Algorithmus von P. Y. Wang

Sei $P = (L, B, \mathcal{R})$ ein Packungsproblem mit $\mathcal{R} = \{R_i \mid i \in \{1, \ldots, n\}\}$.

Algorithmus 2.5

(1) Erzeuge die Grundanordnungen, die aus einem Rechteck bestehen. Aus einem Rechtecktyp $R_i = (l_i, b_i, l_i \cdot b_i, m_i)$ erhält man eine Anordnung $A_i = (l_i, b_i, l_i \cdot w_i, e_i)$. Dabei ist e_i der i-te Einheitsvektor bzw. die Abbildung $e_i \colon \{1, \ldots, n\} \to \{0, 1\} \colon j \mapsto \delta_{ij}$. Prüfe die erzeugten Anordnungen auf Zulässigkeit und füge die Lösungen zu Ω_0 hinzu. Setze $k := 0$.

(2) Erzeuge zu je zwei Lösungen aus Ω_k die horizontale und vertikale Anordnung und prüfe diese auf Zulässigkeit. Falls der Verschnitt $l \cdot b - w$ einer Lösung (l, b, w, a) die Zahl $\beta_a \cdot L \cdot B$ (absoluter Verschnitt) bzw. die Zahl $\beta_r \cdot l \cdot b$ (relativer Verschnitt) nicht überschreitet, so füge die Lösung zu Ω_{k+1} hinzu.

(3) Falls $\Omega_{k+1} \neq \emptyset$, dann setze $k := k + 1$ und gehe zu (2). Andernfalls ist der Algorithmus fertig. Alle zulässigen Anordnungen sind dann in $\bigcup_{j=1}^{k} \Omega_j$ zu finden, es wird eine Lösung mit maximalem Wert ausgewählt und ausgegeben.

Definition 2.6

Für jedes $R_i = (l_i, b_i, w_i, m_i) \in \mathcal{R}$ heißt

$$A_i = (l_i, b_i, w_i, e_i)$$

primitive Anordnung. $\mathbb{A}_{prim}$ ist die Menge aller primitiver Anordnungen. Wir erklären den *horizontalen Kombinationsoperator*

$$\begin{array}{rcl} +_h\colon \quad \mathbb{A} \times \mathbb{A} & \to & \mathbb{A} \\ ((l,b,w,a),(l',b',w',a')) & \mapsto & (l \vee l', b + b', w + w', a + a') \end{array}$$

und den *vertikalen Kombinationsoperator*

$$\begin{array}{rcl} +_v\colon \quad \mathbb{A} \times \mathbb{A} & \to & \mathbb{A} \\ ((l,b,w,a),(l',b',w',a')) & \mapsto & (l + l', b \vee b', w + w', a + a'). \end{array}$$

Für $\mathbb{B}, \mathbb{C} \subset \mathbb{A}$ schreiben wir $\mathbb{B} +_h \mathbb{C}$ bzw. $\mathbb{B} +_v \mathbb{C}$ für das elementweise Operieren. Der Operator

$$\begin{array}{rcl} \text{Prüf}\colon \quad \wp(\mathbb{A}) & \to & \wp(\mathbb{A}_P) \\ X & \mapsto & X \cap \mathbb{A}_P \end{array}$$

heißt *Prüfoperator* und gibt aus einer Menge von Anordnungen die Menge der zulässigen Anordnungen zurück. Der Operator

$$\begin{array}{rcl} \text{Opt}\colon \quad \wp(\mathbb{A}_P) & \to & \wp(\mathbb{A}_P) \\ X & \mapsto & \{A = (l,b,w,a) \in X \mid w \text{ ist maximal}\} \end{array}$$

gibt aus einer Menge von Anordnungen die Menge der optimalen Anordnungen zurück.

Mit den eingeführten Operationen und Operatoren schreibt sich der eingangs verbal beschriebene Algorithmus von Wang wie folgt:

Algorithmus 2.7
(1) $\Omega_0 := \mathbb{A}_{prim}$

(2) $\Omega_{i+1} := \text{Prüf}((\Omega_i +_v \bigcup_{j=1}^{i} \Omega_j) \cup (\Omega_i +_h \bigcup_{j=1}^{i} \Omega_j))$

(3) $\Omega_{i+1} = \emptyset \rightarrow$ *(4)*
$\Omega_{i+1} \neq \emptyset \rightarrow$ *(2)*

(4) Ausgabe: $\text{Opt}(\bigcup_{j=1}^{i} \Omega_j)$

2.4 Einschränkung des Suchraums

In [2] beschreibt P. Y. Wang, wie man den Algorithmus effizienter ablaufen lassen kann. Sie beschränkt sich hierbei auf die diejenigen Packungsprobleme, in denen für alle Rechtecke $R = (l, b, w, m)$ die Beziehung $w = l \cdot b$ gilt.

Definition 2.8

Für eine Anordnung (l, b, w, a) heißt

- $v_i := l \cdot b - w$ *innerer Verschnitt*,
- $v_o := L \cdot B - l \cdot b$ *äußerer Verschnitt* und
- $v_t := v_i + v_o = L \cdot B - w$ *(totaler) Verschnitt*.

Es gibt zwei fundamentale Ideen, den Verschnitt zu minimieren:

- *Methode des absoluten Verschnitts:* Verwende Anordnungen (l, b, w, a), für die $\frac{v_i}{L \cdot B} \leq \beta_a$ für festgelegtes $\beta_a \in [0, 1]$ gilt.
- *Methode des relativen Verschnitts:* Verwende Anordnungen (l, b, w, a), für die $\frac{v_i}{l \cdot b} \leq \beta_r$ für festgelegtes $\beta_r \in [0, 1]$ gilt.

Definition 2.9

Der Operator

$$\begin{aligned} \text{PrüfAbs}_{\beta_a}: \quad \wp(\mathbb{A}) &\rightarrow \wp(\mathbb{A}_P) \\ X &\mapsto \left\{ A = (l, b, w, a) \in X \mid A \in \mathbb{A}_P, \frac{l \cdot b - w}{L \cdot B} \leq \beta_a \right\} \end{aligned}$$

heißt *absoluter Prüfoperator* und gibt aus einer Menge von Anordnungen die Menge der zulässigen Lösungen mit hinreichend kleinem absolu-

tem Abfall zurück. Der Operator

$$\begin{aligned} \text{PrüfRel}_{\beta_r}: \quad \wp(\mathbb{A}) &\rightarrow \wp(\mathbb{A}_P) \\ X &\mapsto \left\{ A = (l, b, w, a) \in X \mid A \in \mathbb{A}_P, \frac{l \cdot b - w}{l \cdot b} \leq \beta_r \right\} \end{aligned}$$

heißt *relativer Prüfoperator* und gibt aus einer Menge von Anordnungen die Menge der zulässigen Lösungen mit hinreichend kleinem relativem Abfall zurück.

Mit diesem beiden Prüfoperatoren wird der Suchraum eingeschränkt. Der Algorithmus ändert sich folgendermaßen:

Algorithmus 2.10

(1) $\Omega_0 := \mathbb{A}_{prim}$

(2) $\Omega_{i+1} := \text{PrüfAbs}_{\beta_a}((\Omega_i +_v \bigcup_{j=1}^{i} \Omega_j) \cup (\Omega_i +_h \bigcup_{j=1}^{i} \Omega_j))$

(3) $\Omega_{i+1} = \emptyset \rightarrow$ *(4)*
$\Omega_{i+1} \neq \emptyset \rightarrow$ *(2)*

(4) Ausgabe: $\text{Opt}(\bigcup_{j=1}^{i} \Omega_j)$

Algorithmus 2.11

(1) $\Omega_0 := \mathbb{A}_{prim}$

(2) $\Omega_{i+1} := \text{PrüfRel}_{\beta_r}((\Omega_i +_v \bigcup_{j=1}^{i} \Omega_j) \cup (\Omega_i +_h \bigcup_{j=1}^{i} \Omega_j))$

(3) $\Omega_{i+1} = \emptyset \rightarrow$ *(4)*
$\Omega_{i+1} \neq \emptyset \rightarrow$ *(2)*

(4) Ausgabe: $\text{Opt}(\bigcup_{j=1}^{i} \Omega_j)$

Beachte: Je kleiner β_a bzw. β_r, desto größer die Chance, dass zuviele Anordnungen aussortiert werden und die optimale Anordnung nicht gefunden werden kann.

Definition 2.12

Das kleinstmögliche β_a bzw. β_r, für das das Verfahren die optimale Anordnung zurückgibt, wird mit β_a^* bzw. β_r^* symbolisiert.

Abschließend geben wir noch eine Abschätzung für β_a^* und β_r^* an. Der Beweis ist in [2] zu finden.

Satz 2.13
Bei der Methode des absoluten Verschnitts gilt für die optimale Fehlertoleranz

$$\beta_a^* \leq \frac{v_t}{L \cdot B}$$

und bei der Methode des relativen Verschnitts gilt

$$\beta_r^* \leq \frac{v_t}{l_0 \cdot b_0},$$

dabei ist $v_t = L \cdot B - w$ der Verschnitt einer beliebigen Anordnung (l, b, w, a) und (l_0, b_0, w_0, a_0) ist eine Anordnung, die als horizontale oder vertikale Kombination zweier Grundrechtecke bzw. primitiver Anordnungen entsteht, mit minimaler Fläche $l_0 \cdot b_0$ bzw. minimalem Wert w_0.

Der folgende Algorithmus findet die optimale Fehlerschranke β_a^* für die Methode des absoluten Verschnitts.

Algorithmus 2.14
(1) Wähle eine geeignete kleine Fehlerschranke β_a.

(2) Führe den Algorithmus 2.7 von Wang aus.

(3) Berechne $\beta_a^* = \min_{(l,b,w,n) \in \bigcup_{j=1}^{i} \Omega_j} \frac{L \cdot B - w}{L \cdot B}$.

(4) Falls $\beta_a^* > \beta_a$, erhöhe β_a in Richtung β_a^* und gehe zu (2). Andernfalls fertig!

Der folgende Algorithmus findet die optimale Fehlerschranke β_r^* für die Methode des relativen Verschnitts.

Algorithmus 2.15
(1) Wähle eine geeignete kleine Fehlerschranke β_r.

(2) Führe den Algorithmus 2.7 von Wang aus.

(3) Berechne $\beta_r^{**} = \min_{(l,b,w,n) \in \bigcup_{j=1}^{i} \Omega_j} \frac{L \cdot B - w}{l_0 \cdot b_0}$. (Dabei ist (l_0, b_0, w_0, a_0) eine Anordnung, die als horizontale oder vertikale Kombination zweier Grundrechtecke bzw. primitiver Anordnungen entsteht, mit minimaler Fläche $l_0 \cdot b_0$ bzw. minimalem Wert w_0.)

(4) Falls $\beta_r^{**} > \beta_r$, erhöhe β_r in Richtung β_r^{**} und gehe zu (2).

(5) Setze $\beta_r := \beta_r^{**}$ und $x := \beta_r$.

(6) Führe den Algorithmus 2.7 von Wang aus.

(7) Berechne $\beta_r^* = \min_{(l,b,w,n)\in\bigcup_{j=1}^{i}\Omega_j} \frac{L\cdot B-w}{l_0\cdot b_0}$.

(8) Falls $\beta_r^* = \beta_r^{**}$, setze $x := \beta_r$, verringere β_r und gehe zu (6). Andernfalls setze $\beta_r^* := x$ und fertig!

2.5 Einschränkung durch Ordnung

Definition 2.16

Zwei Anordnungen $A_1 = (l_1, b_1, w_1, m_1)$ und $A_2 = (l_2, b_2, w_2, m_2)$ heißen *äquivalent*, wenn $l_1 = l_2$, $b_1 = b_2$, $w_1 = w_2$ und $m_1 = m_2$ gelten. Wir schreiben dafür $A_1 \approx A_2$ und entsprechend ist $[A] := A\approx$ die Äquivalenzklasse aller Anordnungen, die zu A äquivalent sind.

Bemerkung: Im obigen Falle ist die definierte Äquivalenzrelation $\approx$ schlicht die Gleichheit. Hat man jedoch Anordnungen $A = (l, b, w, m, A_1, A_2, i)$, die aus weiteren Parametern („Dimensionen") A_1, A_2, i bestehen, wie beispielsweise den beiden Vorgängeranordnungen A_1, A_2 und einer Information i, ob diese horizontal oder vertikal kombiniert wurden, so ist $\approx$ eine echte Äquivalenzrelation, d.h. es gilt $\approx \supsetneq =$. Dies wird beispielsweise so in der Implementierung des Algorithmus im dritten Kapitel benutzt.

Weiter setzen wir die Operatoren horizontale Kombination $+_h$ und vertikale Kombination $+_v$ auf die Faktormenge $\mathbb{A}/\approx$ aller Äquivalenzklassen fort, vermöge

$$[A_1] +_h [A_2] := [A_1 +_h A_2]$$

und

$$[A_1] +_v [A_2] := [A_1 +_v A_2].$$

Analog setzen wir noch

$$\text{Prüf}\{[A_i] \mid i \in I\} := \{[A_i] \mid \mathbb{A}_i \in \text{Prüf}\{A_i\}, i \in I\}$$

sowie

$$\text{Opt}\{[A_i] \mid i \in I\} := \{[A_i] \mid \mathbb{A}_i \in \text{Opt}\{A_i\}, i \in I\}$$

und entsprechend auch für PrüfAbs_{β_a} und PrüfRel_{β_r}. Dann können wir den Suchraum weiter einschränken, indem mit den Äquivalenzklassen gerechnet wird.

Algorithmus 2.17

(1) $\Omega_0 := \mathbb{A}_{prim}/\approx$

(2) $\Omega_{i+1} := \text{PrüfAbs}_{\beta_a}((\Omega_i +_v \bigcup_{j=1}^{i} \Omega_j) \cup (\Omega_i +_h \bigcup_{j=1}^{i} \Omega_j))$

(3) $\Omega_{i+1} = \emptyset \rightarrow$ *(4)*
$\Omega_{i+1} \neq \emptyset \rightarrow$ *(2)*

(4) Ausgabe: $\text{Opt}(\bigcup_{j=1}^{i} \Omega_j)$

Algorithmus 2.18

(1) $\Omega_0 := \mathbb{A}_{prim}/\approx$

(2) $\Omega_{i+1} := \text{PrüfRel}_{\beta_r}((\Omega_i +_v \bigcup_{j=1}^{i} \Omega_j) \cup (\Omega_i +_h \bigcup_{j=1}^{i} \Omega_j))$

(3) $\Omega_{i+1} = \emptyset \rightarrow$ *(4)*
$\Omega_{i+1} \neq \emptyset \rightarrow$ *(2)*

(4) Ausgabe: $\text{Opt}(\bigcup_{j=1}^{i} \Omega_j)$

Definition 2.19

Eine Anordnung $A_1 = (l_1, b_1, w_1, m_1)$ mit dem inneren Verschnitt $v_1 = l_1 \cdot b_1 - w_1$ heißt *besser* als eine Anordnung $A_2 = (l_2, b_2, w_2, m_2)$ mit innerem Verschnitt $v_2 = l_2 \cdot b_2 - w_2$, falls $v_1 \leq v_2$ und $m_1 = m_2$ gilt. Symbol: $A_1 \precsim A_2$. Für eine Anordnung A definieren wir ΥA als diejenige Anordnung, für die keine Anordnung A_0 mit $A_0 \precsim \Upsilon A$ existiert.

Wir setzen

$$\Upsilon\{A_i \mid i \in I\} := \{\Upsilon A_i \mid i \in I\}$$

und

$$\Upsilon[A] := [\Upsilon A]$$

sowie entsprechend

$$\Upsilon\{[A_i] \mid i \in I\} := \{[\Upsilon A_i] \mid i \in I\}.$$

Damit lässt sich der Suchraum des Algorithmus weiter einschränken, indem nur die relativ besten Anordnungen verwendet werden. Es ist allerdings nicht mehr sicherzustellen, dass eine optimale Lösung verwendet wird, da eine optimale Anordnung unter Umständen nur aus einer Anordnung A mit $A \neq \Upsilon A$ kombiniert werden kann.

Algorithmus 2.20

(1) $\Omega_0 := \curlyvee \mathbb{A}_{prim}/\approx$

(2) $\Omega_{i+1} := \curlyvee \text{PrüfAbs}_{\beta_a}((\Omega_i +_v \bigcup_{j=1}^{i} \Omega_j) \cup (\Omega_i +_h \bigcup_{j=1}^{i} \Omega_j))$

(3) $\Omega_{i+1} = \emptyset \rightarrow$ *(4)*
$\Omega_{i+1} \neq \emptyset \rightarrow$ *(2)*

(4) Ausgabe: $\text{Opt}(\bigcup_{j=1}^{i} \Omega_j)$

Algorithmus 2.21

(1) $\Omega_0 := \curlyvee \mathbb{A}_{prim}/\approx$

(2) $\Omega_{i+1} := \curlyvee \text{PrüfRel}_{\beta_r}((\Omega_i +_v \bigcup_{j=1}^{i} \Omega_j) \cup (\Omega_i +_h \bigcup_{j=1}^{i} \Omega_j))$

(3) $\Omega_{i+1} = \emptyset \rightarrow$ *(4)*
$\Omega_{i+1} \neq \emptyset \rightarrow$ *(2)*

(4) Ausgabe: $\text{Opt}(\bigcup_{j=1}^{i} \Omega_j)$

3 Implementierung in Java

In diesem Kapitel beschreiben wir eine Implementierung des Algorithmus von P. Y. Wang in Java. Der Suchraum wird durch eine Begrenzung der Anzahlen der Zwischenergebnisse eingeschränkt, um eine vernünftige Laufzeit und Speicherbedarf sicherzustellen.
Sei im folgenden Text stets n die Anzahl der anzuordnenden Rechtecktypen.

3.1 Abstrakter Datentyp: Rechtecktyp

Für die Typen von Rechtecken wird ein abstrakter Datentyp `RectangleType` definiert, der aus den Komponenten

- Länge `int length`
- Breite `int width`
- Anzahl `int number`
- drehbar `boolean rotatable`

sowie entsprechenden Hilfsfunktionen (Konstruktor, Eingabe- und Ausgabefunktionen) besteht. Die Hilfsfunktionen haben die Zeitkomplexität $\mathcal{O}(1)$.

```
1 package wang;
2
3 public class RectangleType {
4
5     private int length;
6     private int width;
7     private int number;
8     private boolean rotatable;
9
```

```
10      public RectangleType(int l, int w, int n,
            boolean r) {
11          this.length = l;
12          this.width = w;
13          this.number = n;
14          this.rotatable = r;
15      }
16
17      public void setLength(int l) {
18          this.length = l;
19      }
20
21      public void setWidth(int w) {
22          this.width = w;
23      }
24
25      public void setNumber(int n) {
26          this.number = n;
27      }
28
29      public void setRotatable(boolean r) {
30          this.rotatable = r;
31      }
32
33      public int getLength() {
34          return length;
35      }
36
37      public int getWidth() {
38          return width;
39      }
40
41      public int getNumber() {
42          return number;
43      }
44
45      public int getArea() {
46          return length * width;
47      }
```

```
48
49      public boolean isRotatable() {
50          return rotatable;
51      }
52
53  }
```

3.2 Abstrakter Datentyp: Lösung

Für die Anordnungen bzw. Lösungen wird ein abstrakter Datentyp `Solution` definiert, der aus den Komponenten

- Länge `int length`,
- Breite `int width`,
- Fläche `int area`,
- Array der Anzahlen der verwendeten Rechtecke `int[] number`,
- Array der Anzahlen der gedrehten Rechtecke `int[] rotate`,
- Vorgänger 1 `Solution pred1`,
- Vorgänger 2 `Solution pred2` und
- horizontale oder vertikale Kombination `boolean horizontal`

sowie entsprechenden Hilfsfunktionen (Konstruktor, Ausgabefunktionen) besteht. Die Hilfsfunktionen haben die Zeitkomplexität $\mathcal{O}(1)$.
Dieser Datentyp hat eine Baumstruktur. Die beiden Vorgänger `pred1`, `pred2` und der Wahrheitswert `horizontal` dienen lediglich zur Darstellung der Lösung (es kann mit einer rekursiven Routine die Lösung dargestellt werden), sie sind zur Ausführung des Algorithmus nicht nötig.

```
1  package wang;
2
3  public class Solution {
4
5      private int length;
6      private int width;
```

```
 7      private int area;
 8      private int[] number;
 9      private int[] rotate;
10      private Solution pred1;
11      private Solution pred2;
12      private boolean horizontal;
13
14      public Solution(int l, int w, int a, int[] n,
            int[] r, Solution p1, Solution p2, boolean
            h) {
15          this.length = l;
16          this.width = w;
17          this.area = a;
18          this.number = n;
19          this.rotate = r;
20          this.pred1 = p1;
21          this.pred2 = p2;
22          this.horizontal = h;
23      }
24
25      public int getLength() {
26          return length;
27      }
28
29      public int getWidth() {
30          return width;
31      }
32
33      public int getArea() {
34          return area;
35      }
36
37      public int getArea0() {
38          return length * width;
39      }
40
41      public int getTrim() {
42          return length * width - area;
43      }
```

```
44
45      public int[] getNumber() {
46          return number;
47      }
48
49      public int[] getRotate() {
50          return rotate;
51      }
52
53      public Solution getPred1() {
54          return pred1;
55      }
56
57      public Solution getPred2() {
58          return pred2;
59      }
60
61      public boolean getHorizontal() {
62          return horizontal;
63      }
64
65  }
```

Die Methode `getArea0` gibt die Fläche der Anordnung zurück, während `getArea` die durch Rechtecke in der Anordnung überdeckte Fläche zurückgibt. Die Funktion `getTrim` liefert genau die Differenz der beiden Werte, also den Verschnitt der Anordnung bzw. Lösung.

3.3 Algorithmus

Der Kern der Implementierung ist in dieser Klasse `Algorithm`.

```
1  package wang;
2
3  import java.util.HashSet;
4  import java.util.LinkedList;
5  import java.awt.Graphics;
6  import java.awt.Color;
```

```
7
8  public class Algorithm {
```

3.3.1 Hilfsfunktionen

Es werden zuerst noch einige Hilfsfunktionen definiert, die zur Ausführung benötigt werden.

```
10      public static LinkedList<Solution> setToList(
            HashSet<Solution> X) {
11          LinkedList<Solution> x = new LinkedList<
                Solution>();
12          for (Solution s : X) x.add(s);
13          return x;
14      }
```

`setToList` wandelt eine Menge von Lösungen (`HashSet<Solution>`) in eine Liste von Lösungen (`LinkedList<Solution>`) um. Die Zeitkomplexität ist $\mathcal{O}(m)$, dabei ist m die Anzahl der Lösungen in der umzuwandelnden Menge.

```
16      public static int[] getNumbers(RectangleType[]
            x) {
17          int[] n = new int[x.length];
18          for (int i = 0; i < x.length; i++) n[i] =
                x[i].getNumber();
19          return n;
20      }
```

`getNumbers` gibt für ein Array von Rechtecktypen (`RectangleType[]`) ein Array von Integern zurück, welches die Anzahlen der Rechtecktypen enthält. Die Zeitkomplexität ist $\mathcal{O}(n)$.

```
22      public static int getMaximumNumber(
            RectangleType x, int l0, int w0) {
23          return Math.min(x.getNumber(), (l0 / x.
                getLength()) * (w0 / x.getWidth()));
24      }
```

`getMaximumNumber` berechnet für einen Rechtecktyp und die Länge und Breite des Grundrechtecks die maximale Anzahl von anordenbaren Rechtecken dieses Typs. Die Zeitkomplexität ist $\mathcal{O}(1)$.

```
26      public static RectangleType[]
            setMaximumNumbers(RectangleType[] x, int l0
            , int w0) {
27          for (int i = 0; i < x.length; i++) x[i].
                setNumber(getMaximumNumber(x[i], l0, w0
                ));
28          return x;
29      }
```

`setMaximumNumbers` setzt mit Hilfe von `getMaximumNumber` die Anzahlen der Rechtecktypen in einem Array auf die größtmögliche Anzahl, die durch eine Anordnung verwendet werden kann, und gibt das Array zurück. Die Zeitkomplexität beträgt $\mathcal{O}(n)$.

```
31      public static int[] add(int[] x, int[] y) {
32          int[] z = new int[x.length];
33          for (int i = 0; i < x.length; i++) z[i] =
                x[i] + y[i];
34          return z;
35      }
36
37      public static boolean leq(int[] x, int[] y) {
38          for (int i = 0; i < x.length; i++) if (x[i
                ] > y[i]) return false;
39          return true;
40      }
41
42      public static boolean alphaGeq(int[] x, int[]
            y) {
43          for (int i = 0; i < x.length; i++) if (x[i
                ] > y[i]) return true; else if (x[i] <
                y[i]) return false;
44          return true;
45      }
46
```

```
47      public static boolean eq(int[] x, int[] y) {
48          for (int i = 0; i < x.length; i++) if (x[i
                ] != y[i]) return false;
49          return true;
50      }
```

Diese vier Hilfsfunktionen haben als Argumente zwei Arrays von Integern. Insbesondere treten zur Laufzeit des Programms nur Arrays der Länge n auf, d.h. die Argument-Arrays sind stets gleich lang. `add` addiert komponentenweise und gibt ein Array von Integern zurück. `leq` vergleicht komponentenweise nach üblicher Ordnung und gibt einen entsprechenden Wahrheitswert zurück. `alphaGeq` vergleicht nach lexikografischer Ordnung und gibt einen Wahrheitswert zurück. `eq` testet komponentenweise auf Gleichheit und gibt einen Wahrheitswert zurück.
Die Zeitkomplexität beträgt $\mathcal{O}(n)$.

3.3.2 Sortieren von Lösungen

Nun folgen drei Sortieralgorithmen, die eine Liste von Lösungen (`LinkedList<Solution>`) nach verschiedenen Kriterien sortieren. Sie verwenden die MergeSort-Methode.

```
52      public static LinkedList<Solution> sort(
            LinkedList<Solution> X) {
53          if (X.size() <= 1) return X;
54          LinkedList<Solution> Y = new LinkedList<
                Solution>();
55          LinkedList<Solution> Z = new LinkedList<
                Solution>();
56          for (int i = 0; i < X.size(); i++) if (i <
                 X.size() / 2) Y.add(X.get(i)); else Z.
                add(X.get(i));
57          return merge(sort(Y), sort(Z));
58      }
59
60      public static LinkedList<Solution> merge(
            LinkedList<Solution> Y, LinkedList<Solution
            > Z) {
```

```
61          LinkedList<Solution> X = new LinkedList<
                Solution>();
62          while (!Y.isEmpty() && !Z.isEmpty()) if (Y
                .getFirst().getTrim() <= Z.getFirst().
                getTrim()) X.add(Y.pollFirst()); else X
                .add(Z.pollFirst());
63          while (!Y.isEmpty()) X.add(Y.pollFirst());
64          while (!Z.isEmpty()) X.add(Z.pollFirst());
65          return X;
66      }
67
68      public static LinkedList<Solution> sort2(
            LinkedList<Solution> X) {
69          if (X.size() <= 1) return X;
70          LinkedList<Solution> Y = new LinkedList<
                Solution>();
71          LinkedList<Solution> Z = new LinkedList<
                Solution>();
72          for (int i = 0; i < X.size(); i++) if (i <
                 X.size() / 2) Y.add(X.get(i)); else Z.
                add(X.get(i));
73          return merge2(sort2(Y), sort2(Z));
74      }
75
76      public static LinkedList<Solution> merge2(
            LinkedList<Solution> Y, LinkedList<Solution
            > Z) {
77          LinkedList<Solution> X = new LinkedList<
                Solution>();
78          while (!Y.isEmpty() && !Z.isEmpty()) if (Y
                .getFirst().getArea() >= Z.getFirst().
                getArea()) X.add(Y.pollFirst()); else X
                .add(Z.pollFirst());
79          while (!Y.isEmpty()) X.add(Y.pollFirst());
80          while (!Z.isEmpty()) X.add(Z.pollFirst());
81          return X;
82      }
83
84      public static LinkedList<Solution> sort3(
```

```
        LinkedList<Solution> X) {
85          if (X.size() <= 1) return X;
86          LinkedList<Solution> Y = new LinkedList<
                Solution>();
87          LinkedList<Solution> Z = new LinkedList<
                Solution>();
88          for (int i = 0; i < X.size(); i++) if (i <
                 X.size() / 2) Y.add(X.get(i)); else Z.
                add(X.get(i));
89          return merge3(sort3(Y), sort3(Z));
90      }
91
92      public static LinkedList<Solution> merge3(
            LinkedList<Solution> Y, LinkedList<Solution
            > Z) {
93          LinkedList<Solution> X = new LinkedList<
                Solution>();
94          while (!Y.isEmpty() && !Z.isEmpty()) {
95              if (leq(Y.getFirst().getRotate(), Z.
                    getFirst().getRotate())) {
96                  if (alphaGeq(Y.getFirst().
                        getNumber(), Z.getFirst().
                        getNumber())) {
97                      if (Y.getFirst().getArea() >=
                            Z.getFirst().getArea()) {
98                          X.add(Y.pollFirst());
99                      } else {
100                         X.add(Z.pollFirst());
101                     }
102                 } else {
103                     if (Z.getFirst().getArea() >=
                            Y.getFirst().getArea()) {
104                         X.add(Z.pollFirst());
105                     } else {
106                         X.add(Y.pollFirst());
107                     }
108                 }
109             } else {
110                 if (alphaGeq(Z.getFirst().
```

```
                    getNumber(), Y.getFirst().
                    getNumber())) {
111                  if (Z.getFirst().getArea() >=
                        Y.getFirst().getArea()) {
112                      X.add(Z.pollFirst());
113                  } else {
114                      X.add(Y.pollFirst());
115                  }
116              } else {
117                  if (Y.getFirst().getArea() >=
                        Z.getFirst().getArea()) {
118                      X.add(Y.pollFirst());
119                  } else {
120                      X.add(Z.pollFirst());
121                  }
122              }
123          }
124      }
125      while (!Y.isEmpty()) X.add(Y.pollFirst());
126      while (!Z.isEmpty()) X.add(Z.pollFirst());
127      return X;
128  }
```

`sort1` mit `merge1` sortiert nach aufsteigenden Verschnitt und `sort2` mit `merge2` sortiert nach absteigender Fläche. Die dritte Sortierfunktion `sort3` mit `merge3` sortiert nach absteigender Fläche (höchste Priorität), nach alphabetischer Ordnung der Arrays der Anzahlen der verwendeten Rechtecke und nach Ordnung der Arrays der Anzahlen der gedrehten Rechtecke. Diese dient am Ende des Algorithmus zum Sortieren aller erhaltenen Lösungen.
Die Zeitkomplexität beträgt $\mathcal{O}(n \log n)$.

3.3.3 Erkennen und Bereinigen von überflüssigen Lösungen

Um den Suchraum während der Laufzeit des Programms einzuschränken, sind einige Vergleichs- und Bereinigungmethoden notwendig. Diese werden nun vorgestellt.

```
130     public static boolean eq(Solution s1, Solution
            s2) {
131         return s1.getLength() == s2.getLength() &&
                s1.getWidth() == s2.getWidth() && eq(
               s1.getNumber(), s2.getNumber()) && eq(
               s1.getRotate(), s2.getRotate());
132     }
```

`eq` testet zwei Lösungen auf Äquivalenz, d.h. ob ihre Längen und Breiten sowie ihre Anzahlen der verwendeten und gedrehten Rechtecke übereinstimmen, und gibt einen entsprechenden Wahrheitswert zurück.

```
134     public static boolean better(Solution s1,
           Solution s2) {
135         return s1.getTrim() <= s2.getTrim() && eq(
               s1.getNumber(), s2.getNumber()) && eq(
               s1.getRotate(), s2.getRotate());
136     }
```

`better` überprüft, ob eine Lösung besser ist als eine andere, d.h. ob ihr Verschnitt kleiner ist als der der anderen und ob die Anzahlen der verwendeten und gedrehten Rechtecke übereinstimmen, und gibt einen Wahrheitswert zurück.

```
138     public static boolean containsEq(Solution s0,
           HashSet<Solution> X) {
139         for (Solution s : X) if (eq(s, s0)) return
                true;
140         return false;
141     }
142
143     public static boolean containsEq(Solution s0,
           LinkedList<HashSet<Solution>> X) {
144         for (HashSet<Solution> Y : X) if (
               containsEq(s0, Y)) return true;
145         return false;
146     }
```

`containsEq` hat als Argumente eine Lösung `Solution s0` sowie eine Menge von Lösungen `HashSet<Solution> X` und überprüft mit `eq`, ob in `X`

eine zu s0 äquivalente Lösung existiert. Die Zeitkomplexität ist $\mathcal{O}(nm)$, dabei ist m die Anzahl der Lösungen in X. Weiter kann containsEq auch mit einer Lösung Solution s0 sowie einer Liste von Menge von Lösungen LinkedList<HashSet<Solution>> X aufgerufen werden. Die Zeitkomplexität beträgt dann $\mathcal{O}(nml)$, dabei ist l die Länge der Liste X und m das Maximum der Anzahlen der Lösungen über alle Mengen aus der Liste X.

```
148     public static boolean containsBetter(Solution
            s0, HashSet<Solution> X) {
149         for (Solution s : X) if (better(s, s0))
                return true;
150         return false;
151     }
152
153     public static boolean containsBetter(Solution
            s0, LinkedList<HashSet<Solution>> X) {
154         for (HashSet<Solution> Y : X) if (
                containsBetter(s0, Y)) return true;
155         return false;
156     }
```

containsBetter funktioniert analog containsEq, jedoch wird statt eq hier better verwendet, d.h. es wird getestet ob in X eine bessere Lösung als s0 existiert. Die Zeitkomplexitäten sind auch hier $\mathcal{O}(nm)$ bzw. $\mathcal{O}(nml)$.

```
158     public static boolean contains(Solution s,
            LinkedList<HashSet<Solution>> X, HashSet<
            Solution> Y, boolean r) {
159         if (r) return containsBetter(s, Y) ||
                containsBetter(s, X); else return
                containsEq(s, Y) || containsEq(s, X);
160     }
```

contains hat als Eingabedaten eine Lösung s0, eine Menge von Lösungen Y, eine Liste von Mengen von Lösungen X und einen Wahrheitswert r. Wenn r wahr ist, dann wird überprüft, ob es in X oder Y eine bessere Lösung als s0 gibt, andernfalls wird überprüft, ob es in X oder Y eine zu

s0 äquivalente Lösung gibt. Die Zeitkomplexität ist $\mathcal{O}(nml)$.

```
162     public static HashSet<Solution> removeWorse(
            Solution s0, HashSet<Solution> X) {
163         HashSet<Solution> Y = new HashSet<Solution
                >();
164         for (Solution s : X) if (s != s0 && better
                (s0, s)) Y.add(s);
165         X.removeAll(Y);
166         return X;
167     }
168
169     public static LinkedList<HashSet<Solution>>
            removeWorse(Solution s0, LinkedList<HashSet
            <Solution>> X) {
170         for (HashSet<Solution> Y : X) Y =
                removeWorse(s0, Y);
171         return X;
172     }
173
174     public static LinkedList<HashSet<Solution>>
            removeWorse(LinkedList<HashSet<Solution>> X
            ) {
175         for (HashSet<Solution> Y : X) for (
                Solution s : Y) X = removeWorse(s, X);
176         return X;
177     }
```

removeWorse hat als Argumente eine Lösung s0 und eine Menge von Lösungen X und entfernt diejenigen Lösungen aus X, für die in X eine bessere Lösung existiert. Entsprechend auch für eine Liste von Mengen von Lösungen als Argument. Das dritte removeWorse entfernt aus einer Liste von Mengen von Lösungen alle schlechten Lösungen, d.h. Lösungen, für die bereits eine bessere existiert. Die Zeitkomplexitäten betragen $\mathcal{O}(nm)$, $\mathcal{O}(nml)$ bzw. $\mathcal{O}(nmlk)$.

```
179     public static HashSet<Solution> removeTooMuch(
            HashSet<Solution> Y, int k) {
```

```
180         HashSet<Solution> X = new HashSet<Solution
                >();
181         if (Y.isEmpty() || k == 0) return X;
182         LinkedList<Solution> Z = sort(setToList(Y)
                );
183         int j = Math.min(k / 2, Z.size() - 1);
184         int trim = Z.get(j).getTrim();
185         for (Solution s : Z) if (s.getTrim() <=
                trim) X.add(s);
186         if (X.size() > k) {
187             Z = sort2(setToList(X));
188             X = new HashSet<Solution>();
189             int area = Z.get(j).getArea();
190             int i = 0;
191             for (Solution s : Z) if (s.getArea()
                    >= area && i < k) {
192                 X.add(s);
193             i++;
194             }
195         }
196         return X;
197     }
```

`removeTooMuch` sorgt dafür, dass der Suchraum nicht zu groß wird. Die Methode hat als Argumente eine Menge von Lösungen `Y` und eine ganze Zahl `k`. Zuerst wird `Y` in eine Liste von Lösungen `Z` umgewandelt und dann nach aufsteigendem Verschnitt geordnet. Alle diejenigen Lösungen aus `Z`, deren Verschnitt kleiner oder gleich dem Verschnitt des $\lfloor \frac{k}{2} \rfloor$-ten Elements von `Z` ist, werden zu einer Menge von Lösungen `X` hinzugefügt. Falls `X` nicht mehr als `k` Elemente hat, so wird `X` zurückgegeben. Andernfalls wird `X` in eine Liste von Lösungen `Z` umgewandelt und dann nach absteigender Fläche geordnet. `X` wird neu initialisiert. Die ersten `k` Lösungen aus `Z`, deren Fläche größer oder gleich der Fläche des $\lfloor \frac{k}{2} \rfloor$-ten Elements von `Z` ist, werden zu einer Menge von Lösungen `X` hinzugefügt. Schließlich wird `X` zurückgegeben.
Die Zeitkomplexität beträgt aufgrund der verwendeten Sortiermethoden hier $\mathcal{O}(n \log n)$.

3.3.4 Erstellen der Basislösungen

Wie in den ersten beiden Kapiteln vorgestellt, kombiniert der Algorithmus von Wang zwei Lösungen zu einer neuen Lösung. Die folgenden Funktionen dienen der Erzeugung von Basislösungen, aus denen dann alle weiteren Lösungen kombiniert werden.

```
199     public static int[] getInitialNumber(int i,
            int n) {
200         int[] e = new int[n];
201         for (int j = 0; j < n; j++) if (j == i) e[
                j] = 1; else e[j] = 0;
202         return e;
203     }
```

`getInitialNumber` hat als Argumente zwei ganze Zahlen `i` und `n` und gibt den `i`-ten Einheitsvektor der Länge `n` zurück, genauer ein Array von Integern der Länge `n`, das an der `i`-ten Stelle eine 1 hat und sonst 0. Die Zeitkomplexität ist $\mathcal{O}(n)$.

```
205     public static int[] getInitialRotate(int i,
            int n, boolean a) {
206         int[] e = new int[n];
207         for (int j = 0; j < n; j++) if (a && j ==
                i) e[j] = 1; else e[j] = 0;
208         return e;
209     }
```

`getInitialRotate` hat als Argumente zwei ganze Zahlen `i` und `n` sowie einen booleschen Wert `a`. Wenn `a` wahr ist, gibt `getInitialRotate` den `i`-ten Einheitsvektor der Länge `n` zurück, also ein Array von Integern der Länge `n`, das an der `i`-ten Stelle eine 1 hat und sonst 0. Andernfalls wird der Nullvektor der Länge `n` zurückgegeben, d.h. ein Array von Integern der Länge `n`, das an jeder Stelle eine 0 hat. Als Zeitkomplexität ergibt sich $\mathcal{O}(n)$.

```
211     public static HashSet<Solution>
            getFirstSolutions(RectangleType[] x, int l,
             int w, double t, boolean a) {
```

```
212         HashSet<Solution> X = new HashSet<Solution
                >();
213         int[] n = getNumbers(x);
214         Solution s;
215         for (int i = 0; i < x.length; i++) {
216             s = new Solution(x[i].getLength(), x[i
                    ].getWidth(), x[i].getArea(),
                    getInitialNumber(i, x.length),
                    getInitialRotate(i, x.length, false
                    ), null, null, true);
217             if (isPermissible(s, l, w, n, t, a)) X
                    .add(s);
218             if (x[i].isRotatable()) {
219                 s = new Solution(x[i].getWidth(),
                        x[i].getLength(), x[i].getArea
                        (), getInitialNumber(i, x.
                        length), getInitialRotate(i, x.
                        length, true), null, null, true
                        );
220                 if (isPermissible(s, l, w, n, t, a
                        )) X.add(s);
221             }
222         }
223         return X;
224     }
```

`getFirstSolutions` erzeugt aus gegebenen Rechtecktypen die Basislösungen. Dazu hat die Methode die Argumente

- Array von Rechtecktypen `RectangleType[] x`,
- Länge des Grundrechtecks `int l`,
- Breite des Grundrechtecks `int w`,
- Fehlerkoeffizient `double t` (um maximalen Verschnitt festzulegen) und
- Auswahl der Methode (relativer Verschnitt / absoluter Verschnitt) `boolean a`.

Für jeden Rechtecktyp wird eine entsprechende Basislösung bzw. Grundanordnung erzeugt, die aus genau einem Rechteck dieses Typs besteht. Ist die Lösung zulässig, so wird sie zu einer Menge von Lösungen `X` hinzugefügt. Wenn es sich um einen drehbaren Rechtecktyp handelt, so wird eine entsprechend gedrehte Basislösung erzeugt (Länge und Breite sind vertauscht) und falls diese zulässig ist, so wird sie auch zu `X` hinzugefügt. Schließlich wird `X` zurückgegeben.
Die Zeitkomplexität dieser Methode ist $\mathcal{O}(n^2)$.

3.3.5 Kombinieren und Prüfen von Lösungen

Es werden zwei binäre partielle Operationen auf den Lösungen definiert.

```
226     public static Solution horizontalCombination(
            Solution s1, Solution s2) {
227         return new Solution(
228                 s1.getLength() + s2.getLength(),
229                 Math.max(s1.getWidth(), s2.
                        getWidth()),
230                 s1.getArea() + s2.getArea(),
231                 add(s1.getNumber(), s2.getNumber()
                        ),
232                 add(s1.getRotate(), s2.getRotate()
                        ),
233                 s1,
234                 s2,
235                 true);
236     }
237
238     public static Solution verticalCombination(
            Solution s1, Solution s2) {
239         return new Solution(
240                 Math.max(s1.getLength(), s2.
                        getLength()),
241                 s1.getWidth() + s2.getWidth(),
242                 s1.getArea() + s2.getArea(),
243                 add(s1.getNumber(), s2.getNumber()
                        ),
```

```
244                 add(s1.getRotate(), s2.getRotate()
                        ),
245                 s1,
246                 s2,
247                 false);
248     }
```

`horizontalCombination` bzw. `verticalCombination` erzeugen für zwei Lösungen die horizontale bzw. vertikale Kombination, entsprechend der mathematische Definition in den vorhergehenden Kapiteln. Zusätzlich werden in der erzeugten Lösung noch die beiden erzeugenden Lösungen gespeichert und ob es sich um eine horizontale oder vertikale Kombination handelt. Beide Methoden haben die Zeitkomplexität $\mathcal{O}(n)$.

```
250     public static boolean isPermissible(Solution s
            , int l, int w, int[] n, double t, boolean
            a) {
251         if (s.getLength() > l) return false;
252         if (s.getWidth() > w) return false;
253         if (!leq(s.getNumber(), n)) return false;
254         if (a) {
255             if (s.getTrim() >= t * l * w) return
                    false;
256         } else {
257             if (s.getTrim() >= t * s.getArea0())
                    return false;
258         }
259         return true;
260     }
```

`isPermissible` überprüft Lösungen auf Zulässigkeit und hat die Argumente

- Lösung `Solution s`,
- Länge des Grundrechtecks `int l`,
- Breite des Grundrechtecks `int w`,
- Array der Anzahlen der Rechtecktypen `int[] n`,

- Fehlerkoeffizient `double t` (um maximalen Verschnitt festzulegen) und
- Auswahl der Methode (relativer Verschnitt / absoluter Verschnitt) `boolean a`.

Die Funktion liefert den Wahrheitswert `false`, wenn die Länge bzw. die Breite der Lösung die Länge bzw. die Breite des Grundrechtecks überschreitet, oder falls die Lösung mehr Rechtecke verwendet, als vorhanden sind, oder falls der Verschnitt der Lösung zu groß ist. Andernfalls gibt sie `true` zurück.
Die Zeitkomplexität ergibt sich zu $\mathcal{O}(n)$.

3.3.6 Algorithmus von P. Y. Wang

Die folgenden Methoden implementieren den Algorithmus von Wang.

```
262     public static LinkedList<HashSet<Solution>>
            wangSolution(Graphics g, RectangleType[] x,
             int l, int w, double t, boolean a, int s,
            int m, boolean r, boolean v) {
263          LinkedList<HashSet<Solution>> X = new
                 LinkedList<HashSet<Solution>>();
264          X.add(new HashSet<Solution>(
                 getFirstSolutions(x, l, w, t, a)));
265          int k = 0;
266          if (s == -1) k = -2;
267          String aa;
268          if (a) aa = "abs"; else aa = "rel";
269          String ss;
270          if (s != -1) ss = ",␣" + String.valueOf(s)
                 .toString() + "stp"; else ss = "";
271          String mm;
272          if (m != -1) mm = ",␣" + String.valueOf(m)
                 .toString() + "sol"; else mm = "";
273          String rr;
274          if (r) rr = ",␣cln"; else rr = "";
275          System.out.print("\n\n[" + aa + ",␣" + t +
                  ss + mm + rr + "]");
```

```
276         while (!X.getLast().isEmpty() && k < s) {
277             X = wangStep(g, x, l, w, t, a, X, m, r
                    , v);
278             if (s != -1) k++;
279         }
280         return X;
281     }
```

`wangSolution` ist der Aufruf des Algorithmus von Wang und hat die Argumente

- Grafikausgabebereich `Graphics g`,
- Array von Rechtecktypen `RectangleType[] x`,
- Länge des Grundrechtecks `int l`,
- Breite des Grundrechtecks `int w`,
- Fehlerkoeffizient `double t` (um maximalen Verschnitt festzulegen),
- Auswahl der Methode (relativer Verschnitt / absoluter Verschnitt) `boolean a`,
- maximale Anzahl von Schritten `int s`,
- maximale Anzahl von Zwischenergebnissen `int m`,
- Möglichkeit der starken Bereinigung `boolean r` (`eq` oder `better`) und
- Möglichkeit der Grafikausgabe `boolean v`.

Für `s` = -1 ist die Schrittzahl unbegrenzt und für `m` = -1 ist die Anzahl der Zwischenergebnisse nicht begrenzt. Es wird eine neue Liste von Mengen von Lösungen `LinkedList<HashSet<Solution>> X` erzeugt. Die Menge aller Grundlösungen wird zu `X` hinzugefügt. Es folgen einige Ausgaben auf der Java-Konsole. Nun wird `X = wangStep(g, x, l, w, t, a, X, m, r, v)` solange aufgerufen (höchstens `s`-mal), bis das letzte Element von `X` leer ist, d.h. bis keine neuen Lösungen mehr erzeugt werden. Schließlich wird `X` zurückgegeben.
Für die Zeitkomplexität ergibt sich $\mathcal{O}(n^2m^2)$, dabei ist m die maximale Anzahl an Zwischenergebnissen.

```
283     public static LinkedList<HashSet<Solution>>
            wangStep(Graphics g, RectangleType[] x, int
             l, int w, double t, boolean a, LinkedList<
            HashSet<Solution>> X, int m, boolean r,
            boolean v) {
284          Solution s;
285          int[] n = getNumbers(x);
286          double f = getZoomFactor(l, w);
287          HashSet<Solution> Y = new HashSet<Solution
                 >();
288          for (Solution s1 : X.getLast()) {
289              for (int i = 0; i < X.size(); i++) {
290                  for (Solution s2 : X.get(i)) {
291                      s = horizontalCombination(s1,
                             s2);
292                      if (isPermissible(s, l, w, n,
                             t, a) && !contains(s, X, Y,
                              r)) {
293                          if (r) Y = removeWorse(s,
                                 Y);
294                          Y.add(s);
295                          if (v) {
296                              g = resetGraphics(g);
297                              g = drawSolution(s, g,
                                     l, w, 0, 0, f);
298                              g = drawBasicRectangle
                                     (g, l, w, f);
299                          }
300                      }
301                      s = verticalCombination(s1, s2
                             );
302                      if (isPermissible(s, l, w, n,
                             t, a) && !contains(s, X, Y,
                              r)) {
303                          if (r) Y = removeWorse(s,
                                 Y);
304                          Y.add(s);
305                          if (v) {
306                              g = resetGraphics(g);
```

```
307                                 g = drawSolution(s, g,
                                        l, w, 0, 0, f);
308                                 g = drawBasicRectangle
                                        (g, l, w, f);
309                             }
310                         }
311                     }
312                 }
313             }
314             System.out.print("\n" + X.size() + ":␣");
315             if (r) X = removeWorse(X);
316             if (m != -1) {
317                 System.out.print(Y.size() + ",␣");
318                 Y = removeTooMuch(Y, m);
319             }
320             System.out.print(Y.size());
321             X.add(Y);
322             return X;
323         }
```

`wangStep` ist ein Iterationsschritt des Algorithmus von Wang und hat die Argumente

- Grafikausgabebereich `Graphics g`,
- Array von Rechtecktypen `RectangleType[] x`,
- Länge des Grundrechtecks `int l`,
- Breite des Grundrechtecks `int w`,
- Fehlerkoeffizient `double t` (um maximalen Verschnitt festzulegen),
- Auswahl der Methode (relativer Verschnitt / absoluter Verschnitt) `boolean a`,
- Liste von Mengen von Lösungen `LinkedList<HashSet<Solution>> X`,
- maximale Anzahl von Zwischenergebnissen `int m`,
- Möglichkeit der starken Bereinigung `boolean r` (`eq` oder `better`) und

- Möglichkeit der Grafikausgabe `boolean v`.

Es wird eine neue Menge von Lösungen `HashSet<Solution> Y` erzeugt. Die Liste `X` bestehe aus den Mengen `X_1`,...,`X_n`. Nun wird jede Lösung aus `X_n` mit jeder Lösung aus $\bigcup_{i=1}^{n}$`X_i` horizontal und vertikal kombiniert, die jeweils entstehende Lösung wird auf Zulässigkeit (`isPermissible`) und Überflüssigkeit(`!contains`) überprüft und entsprechend zu `Y` hinzugefügt oder verworfen. Falls `v` `true` ist, so wird die erzeugte Lösung gezeichnet. Am Ende wird gegebenenfalls die Anzahl der Lösungen in `Y` beschränkt durch den Aufruf der Methode `removeTooMuch` und schließlich wird `Y` zu `X` hinzugefügt und `X` ausgegeben.
Die Zeitkomplexität beträgt $\mathcal{O}(nm^2)$, dabei ist m die maximale Anzahl an Zwischenergebnissen.

```
325     public static int wangTolerance(RectangleType
            [] x, int l, int w, double t, boolean a) {
326         HashSet<Solution> X = getFirstSolutions(x,
                l, w, t, a);
327         HashSet<Solution> Y = new HashSet<Solution
                >();
328         Solution s;
329         int[] n = getNumbers(x);
330         for (Solution s1 : X) for (Solution s2 : X
                ) {
331             s = horizontalCombination(s1 ,s2);
332             if (isPermissible(s, l, w, n, t, a)) Y
                    .add(s);
333             s = verticalCombination(s1, s2);
334             if (isPermissible(s, l, w, n, t, a)) Y
                    .add(s);
335         }
336         int z = l * w;
337         for (Solution y : Y) if (y.getArea0() < z)
                z = y.getArea0();
338         return z;
339     }
```

Diese Methode `wangTolerance` dient der Bestimmung einer oberen Schranke für die optimale Fehlerschranke bei der relativen Methode. Es wird für die Argumente

- Array von Rechtecktypen `RectangleType[] x`,
- Länge des Grundrechtecks `int l`,
- Breite des Grundrechtecks `int w`,
- Fehlerkoeffizient `double t` (um maximalen Verschnitt festzulegen),
- Auswahl der Methode (relativer Verschnitt / absoluter Verschnitt) `boolean a` (Aufruf dieser Methode stets mit `a = false`)

eine ganze Zahl zurückgeliefert, die die minimale Fläche einer zulässigen Kombination zweier Basislösungen angibt. Für die Ausführung des Algorithmus ist diese Funktion nicht nötig.

3.3.7 Grafische Darstellung von Lösungen

Zur grafischen Darstellung von Lösungen werden die nachfolgenden Methoden genutzt.

```
341     public static double getZoomFactor(int l, int
            w) {
342         if (l == 0 || w == 0) return 0;
343         return Math.min(800 / (double) l, 300 / (
                double) w);
344     }
```

`getZoomFactor` bestimmt aus der Länge und Breite des Grundrechtecks einen Skalierungsfaktor, um die Lösung für beliebige Ausmaße vollständig darzustellen.

```
346     public static Graphics drawSolution(Solution s
            , Graphics g, int l, int w, int x, int y,
            double f) {
347         if (!(s.getPred1() == null)) {
348             if (s.getHorizontal()) g =
                    drawSolution(s.getPred1(),
                    drawSolution(s.getPred2(), g, l, w,
                     x + s.getPred1().getLength(), y, f
                    ), l, w, x, y, f);
```

```
349             if (!s.getHorizontal()) g =
                    drawSolution(s.getPred1(),
                    drawSolution(s.getPred2(), g, l, w,
                     x, y + s.getPred1().getWidth(), f)
                    , l, w, x, y, f);
350         } else {
351             g.setColor(Color.black);
352             g.drawRect((int) (f * x), (int) (f * y
                    ), (int) (f * s.getLength()), (int)
                     (f * s.getWidth()));
353             int k = 0;
354             for (int i = 0; i < s.getNumber().
                    length; i++) if (s.getNumber()[i]
                    != 0) k = i;
355             g.drawString(String.valueOf(k).
                    toString(), (int) (f * x + 2), (int
                    ) (f * y + 10));
356         }
357         return g;
358     }
```

`drawSolution` zeichnet eine Anordnung. Dazu dienen die Argumente

- Lösung `Solution s`,
- Grafikausgabebereich `Graphics g`,
- Länge des Grundrechtecks `int l`,
- Breite des Grundrechtecks `int w`,
- horizontale Koordinate `int x`,
- vertikale Koordinate `int y` und
- Skalierungsfaktor `double f`.

Verbal beschrieben verläuft das rekursive Zeichnen einer Lösung $A = (l, w, n, A_1, A_2, h)$ so:

(1) Hat die Lösung Vorgänger (ist also keine Grundlösung), d.h. gilt $A_1 \neq () \neq A_2$, dann ruft die Methode sich selbst auf, mit entsprechend verschobenen Koordinaten x und y, d.h. genauer setze $A = A_1$ und gehe zu (1) sowie setze $A = A_2$ und verschiebe die aktuelle Position

$x := x + l_1$ nach rechts für $h = 1$ (horizontal) bzw. verschiebe die aktuelle Position $y := y + w_1$ nach unten für $h = 0$ (vertikal) und gehe auch zu (1). Andernfalls wird an der Stelle (x, y) die Grundlösung A als ein schwarzes Rechteck mit der Nummer des Rechtecktyps gezeichnet, d.h. zeichne an aktueller Position ein Rechteck der Größe $l \cdot w$.

```
360     public static Graphics drawRectangleType(
            Graphics g, int l0, int w0, int l, int w,
            int n, boolean r, double f) {
361         g = resetGraphics(g);
362         g.setColor(Color.black);
363         g.drawRect(0, 0, (int) (f * l), (int) (f *
                w));
364         if (r) g.drawRect(0, 0, (int) (f * w), (
                int) (f * l));
365         g.drawString(String.valueOf(n).toString(),
                2, 10);
366         g = drawBasicRectangle(g, l0, w0, f);
367         return g;
368     }
```

`drawRectangleType` zeichnet das Grundrechteck und ein Exemplar eines Rechtecktyps bzw. ein normales und ein gedrehtes Exemplar für einen drehbaren Rechtecktyp. Dazu dienen die Argumente

- Grafikausgabebereich `Graphics g`,
- Länge des Grundrechtecks `int l0`,
- Breite des Grundrechtecks `int w0`,
- Länge des Rechtecktyps `int l`,
- Breite des Rechtecktyps `int w`,
- Nummer des Rechtecktyps `int n`,
- drehbar `boolean r` und
- Skalierungsfaktor `double f`.

```
370     public static Graphics drawBasicRectangle(
            Graphics g, int l, int w, double f) {
371         g.setColor(Color.red);
372         g.drawRect(0, 0, (int) (f * l), (int) (f *
                w));
373         return g;
374     }
```

`drawBasicRectangle` zeichnet das Grundrechteck.

```
376     public static Graphics resetGraphics(Graphics
            g) {
377         g.clearRect(0, 0, 801, 301);
378         return g;
379     }
```

`resetGraphics` initialisiert den Grafikausgabebereich.

```
381 }
```

3.4 Eingabe und Ausgabe von Daten

3.4.1 Ausgaben auf der Konsole und im Programm

Während der Laufzeit des Programms werden auf der Java-Konsole Daten ausgegeben: Anfangs (in `wangSolution`) wird ein String `[a, t, s, m, r]` ausgegeben, der die Parameter angibt, genauer ist `a = rel` für die Methode des relativen Verschnitts und `a = abs` für die Methode des absoluten Verschnitts, `t` gibt den Verschnittkoeffizient bzw. Fehlerschranke an, `s` gibt die Anzahl der Schritte an (gefolgt von `stp`), `m` gibt die maximale Anzahl von Zwischenlösungen an (gefolt von `sol`) und falls starke Bereinigung gewählt wurde, dann ist `r = cln`. Dann folgen für jeden Schritt des Algorithmus (in `wangStep`) die Anzahlen der erzeugten Zwischenlösungen, jeweils vor und nach der Bereinigungsfunktion `removeTooMuch`. Eine Lösung wird im Programm im Format `Länge x Breite (Fläche|innerer Verschnitt|gesamter Verschnitt) : Anzahlen der verwendeten Rechtecke` dargestellt.

3.4.2 Datenformat für Rechteck-Datensatz

Das Programm kann Datensätze von Rechtecken lesen und schreiben. Dabei wird für die Dateien folgendes Format verwendet:

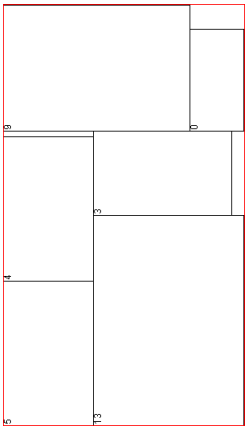
```
-2 Länge Breite des Grundrechtecks
-1 Anzahl von Rechtecktypen
 0 Länge Breite Anzahl drehbar
 1 Länge Breite Anzahl drehbar
 2 Länge Breite Anzahl drehbar
 3 ...
```

4 Beispielprobleme und Lösung

Die folgenden Probleme wurden von dem Programm auf einem PC (Intel Pentium T2370 DualCore 1,73GHz, 2GB RAM, Windows Vista 32bit) gelöst.

4.1 Das Beispiel von P. Y. Wang (70x40, 20 Rechtecke)

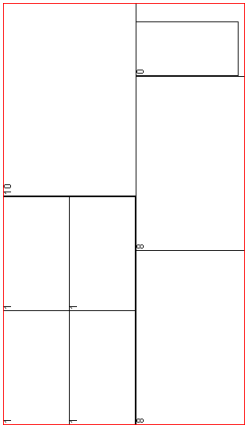
```
-2  70  40
-1  20
 0  17  9  1  false
 1  11  19  4  false
 2  12  21  3  false
 3  14  23  4  false
 4  24  15  1  false
 5  24  15  2  false
 6  25  16  4  false
 7  27  17  2  false
 8  18  29  3  false
 9  21  31  3  false
10  32  22  2  false
11  23  33  3  false
12  34  24  2  false
13  35  25  2  false
14  36  26  1  false
15  37  27  1  false
16  38  28  1  false
17  39  29  1  false
18  41  30  1  false
19  43  31  1  false
```



```
70x40 (2721|79|79) : 1 0 0 1
    1 1 0 0 0 1 0 0 0 1 0 0
  0 0 0 0
[rel, 0.043, 3stp, 68sol,
  cln]
127 solutions in 0.010 s
```

4.2 Das Beispiel von P. Y. Wang (70x40, 20 Rechtecke, drehbar)

```
-2  70  40
-1  20
 0  17  9  1  true
 1  11  19  4  true
 2  12  21  3  true
 3  14  23  4  true
 4  24  15  1  true
 5  24  15  2  true
 6  25  16  4  true
 7  27  17  2  true
 8  18  29  3  true
 9  21  31  3  true
10  32  22  2  true
11  23  33  3  true
12  34  24  2  true
13  35  25  2  true
14  36  26  1  true
15  37  27  1  true
16  38  28  1  true
17  39  29  1  true
18  41  30  1  true
19  43  31  1  true
```

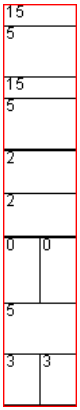


```
70x40 (2737|63|63) : 1 4 0 0
    0 0 0 0 0 2 0 1 0 0 0 0 0 0
    0 0 0 0
[rel, 0.048, 4stp, 60sol,
    cln]
170 solutions in 0.032 s
```

4.3 Datensätze cl1

4.3.1 Datensatz cl1_1 (10x55, 20 Rechtecke)

```
-2   10  55
-1   20
 0   5  9  2
 1   4  2  2
 2   10  6  2
 3   5  7  3
 4   6  3  2
 5   10  7  3
 6   1  5  2
 7   3  5  3
 8   6  9  2
 9   2  4  3
10   6  7  2
11   7  2  3
12   8  3  2
13   4  10  3
14   4  5  2
15   10  3  3
16   8  3  2
17   7  8  3
18   8  3  2
19   8  7  3
```




```
10x55 (550|0|0) : 2 0 2 2 0
    3 0 0 0 0 0 0 0 0 0 2 0 0
      0 0
[rel, 0.0010, 5stp, 34sol,
    cln]
102 solutions in 0.013 s
```

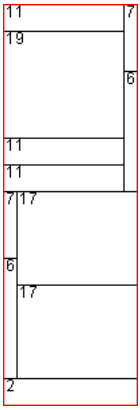
4.3.2 Datensatz cl1_1r (10x55, 20 Rechtecke, drehbar)

```
-2  10  55
-1  20
 0  5  9  2  true
 1  4  2  2  true
 2  10  6  2  true
 3  5  7  3  true
 4  6  3  2  true
 5  10  7  3  true
 6  1  5  2  true
 7  3  5  3  true
 8  6  9  2  true
 9  2  4  3  true
10  6  7  2  true
11  7  2  3  true
12  8  3  2  true
13  4  10  3  true
14  4  5  2  true
15  10  3  3  true
16  8  3  2  true
17  7  8  3  true
18  8  3  2  true
19  8  7  3  true
```


```
10x55 (550|0|0) : 2 0 2 0 0
    3 0 0 0 0 0 0 0 1 0 3 0 0
     0 0
[rel, 0.0010, 6stp, 60sol,
    cln]
272 solutions in 0.079 s
```

4.3.3 Datensatz cl1_2 (10x30, 20 Rechtecke)

```
-2   10  30
-1   20
 0   2  2  2
 1   6  8  3
 2   10  2  2
 3   1  3  3
 4   8  4  2
 5   3  10  3
 6   1  9  2
 7   1  5  3
 8   6  3  2
 9   1  1  3
10   4  2  2
11   9  2  3
12   1  9  2
13   9  5  3
14   4  7  2
15   2  2  3
16   3  4  2
17   9  7  3
18   4  1  2
19   9  8  3
```




```
10x30 (300|0|0) : 0 0 1 0 0
    0 2 2 0 0 0 3 0 0 0 0 0 2
     0 1
[rel, 0.0010, 4stp, 92sol,
    cln]
257 solutions in 0.054 s
```

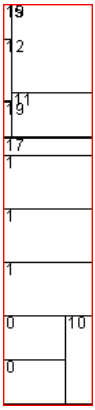
4.3.4 Datensatz cl1_2r (10x30, 20 Rechtecke, drehbar)

```
-2  10  30
-1  20
 0  2  2  2  true
 1  6  8  3  true
 2  10  2  2  true
 3  1  3  3  true
 4  8  4  2  true
 5  3  10  3  true
 6  1  9  2  true
 7  1  5  3  true
 8  6  3  2  true
 9  1  1  3  true
10  4  2  2  true
11  9  2  3  true
12  1  9  2  true
13  9  5  3  true
14  4  7  2  true
15  2  2  3  true
16  3  4  2  true
17  9  7  3  true
18  4  1  2  true
19  9  8  3  true
```


```
10x30 (300|0|0) : 0 0 0 0 0
    1 0 0 0 0 0 3 0 0 0 0 0 0
     0 3
[rel, 0.0010, 4stp, 136sol]
340 solutions in 0.102 s
```

4.3.5 Datensatz cl1_3 (10x45, 20 Rechtecke)

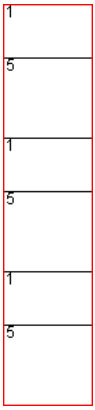
```
-2  10  45
-1  20
 0  7  5  2  false
 1  10  6  3  false
 2  5  6  2  false
 3  7  2  3  false
 4  4  8  2  false
 5  9  10  3  false
 6  8  5  2  false
 7  8  6  3  false
 8  4  9  2  false
 9  9  3  3  false
10  3  10  2  false
11  9  5  3  false
12  1  7  2  false
13  8  9  3  false
14  4  6  2  false
15  3  6  3  false
16  4  3  2  false
17  10  2  3  false
18  6  1  2  false
19  1  4  3  false
```




```
10x45 (450|0|0) : 2 3 0 0 0
    1 0 0 0 0 0 1 1 1 0 0 0 0 0 1
      0 2
[rel, 0.0010, 4stp, 1205sol,
     cln]
2207 solutions in 1:34 m
```

4.3.6 Datensatz cl1_3r (10x45, 20 Rechtecke, drehbar)

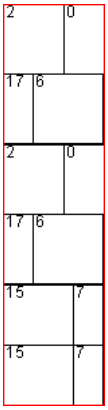
```
-2  10  45
-1  20
 0  7  5  2  true
 1  10  6  3  true
 2  5  6  2  true
 3  7  2  3  true
 4  4  8  2  true
 5  9  10  3  true
 6  8  5  2  true
 7  8  6  3  true
 8  4  9  2  true
 9  9  3  3  true
10  3  10  2  true
11  9  5  3  true
12  1  7  2  true
13  8  9  3  true
14  4  6  2  true
15  3  6  3  true
16  4  3  2  true
17  10  2  3  true
18  6  1  2  true
19  1  4  3  true
```




```
10x45 (450|0|0) : 0 3 0 0 0
    3 0 0 0 0 0 0 0 0 0 0 0 0
     0 0
[rel, 0.0010, 3stp, 6sol]
52 solutions in 0.006 s
```

4.3.7 Datensatz cl1_4 (10x40, 20 Rechtecke)

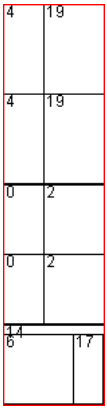
```
-2  10  40
-1  20
 0  4  7  2  false
 1  1  3  3  false
 2  6  7  2  false
 3  8  2  3  false
 4  9  4  2  false
 5  6  2  3  false
 6  7  7  2  false
 7  3  6  3  false
 8  2  7  2  false
 9  1  3  3  false
10  3  8  2  false
11  4  3  3  false
12  1  9  2  false
13  8  1  3  false
14  1  10  2  false
15  7  6  3  false
16  9  5  2  false
17  3  7  3  false
18  8  3  2  false
19  6  9  3  false
```




```
10x40 (400|0|0) : 2 0 2 0 0
    0 2 2 0 0 0 0 0 0 0 2 0 2
     0 0
[rel, 0.0010, 5stp, 34sol]
101 solutions in 0.010 s
```

4.3.8 Datensatz cl1_4r (10x40, 20 Rechtecke, drehbar)

```
-2  10  40
-1  20
 0   4  7  2  true
 1   1  3  3  true
 2   6  7  2  true
 3   8  2  3  true
 4   9  4  2  true
 5   6  2  3  true
 6   7  7  2  true
 7   3  6  3  true
 8   2  7  2  true
 9   1  3  3  true
10   3  8  2  true
11   4  3  3  true
12   1  9  2  true
13   8  1  3  true
14   1 10  2  true
15   7  6  3  true
16   9  5  2  true
17   3  7  3  true
18   8  3  2  true
19   6  9  3  true
```




```
10x40 (400|0|0) : 2 0 2 0 2
    0 1 0 0 0 0 0 0 0 1 0 0 1
     0 2
[rel, 0.0010, 6stp, 58sol,
    cln]
282 solutions in 0.084 s
```

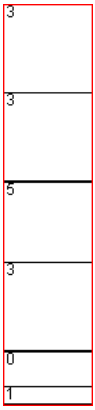
4.3.9 Datensatz cl1_5 (10x45, 20 Rechtecke)

```
-2  10  45
-1  20
 0  10  4  2  false
 1  2  10  3  false
 2  2  4  2  false
 3  10  10  3  false
 4  7  2  2  false
 5  9  10  3  false
 6  5  6  2  false
 7  7  5  3  false
 8  1  7  2  false
 9  5  3  3  false
10  3  9  2  false
11  9  4  3  false
12  2  10  2  false
13  4  3  3  false
14  2  2  2  false
15  4  9  3  false
16  2  8  2  false
17  1  1  3  false
18  1  7  2  false
19  4  4  3  false
```


```
10x45 (450|0|0) : 2 0 0 2 0
    1 0 0 2 2 0 1 0 0 0 0 0 0
     0 0
[rel, 0.0010, 4stp, 84sol]
195 solutions in 0.048 s
```

4.3.10 Datensatz cl1_5r (10x45, 20 Rechtecke, drehbar)

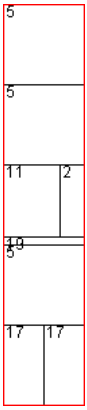
```
-2  10  45
-1  20
 0  10  4  2  true
 1  2  10  3  true
 2  2  4  2  true
 3  10  10  3  true
 4  7  2  2  true
 5  9  10  3  true
 6  5  6  2  true
 7  7  5  3  true
 8  1  7  2  true
 9  5  3  3  true
10  3  9  2  true
11  9  4  3  true
12  2  10  2  true
13  4  3  3  true
14  2  2  2  true
15  4  9  3  true
16  2  8  2  true
17  1  1  3  true
18  1  7  2  true
19  4  4  3  true
```




```
10x45 (450|0|0) : 1 1 0 3 0
    1 0 0 0 0 0 0 0 0 0 0 0 0
     0 0
[rel, 0.0010, 4stp, 6sol]
59 solutions in 0.008 s
```

4.3.11 Datensatz cl1_6 (10x50, 20 Rechtecke)

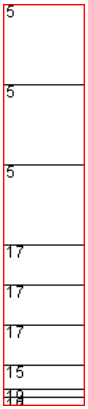
```
-2  10  50
-1  20
 0  2  7  2  false
 1  8  2  3  false
 2  3  9  2  false
 3  9  5  3  false
 4  2  8  2  false
 5  10  10  3  false
 6  6  8  2  false
 7  6  9  3  false
 8  8  7  2  false
 9  5  6  3  false
10  1  6  2  false
11  7  9  3  false
12  3  10  2  false
13  7  9  3  false
14  6  7  2  false
15  3  10  3  false
16  3  7  2  false
17  5  10  3  false
18  3  5  2  false
19  10  1  3  false
```




```
10x50 (500|0|0) : 0 0 1 0 0
    3 0 0 0 0 0 1 0 0 0 0 0 2
     0 1
[rel, 0.0010, 3stp, 22sol,
    cln]
59 solutions in 0.009 s
```

4.3.12 Datensatz cl1_6r (10x50, 20 Rechtecke, drehbar)

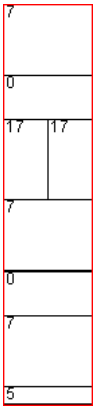
```
-2  10  50
-1  20
 0  2 7 2 true
 1  8 2 3 true
 2  3 9 2 true
 3  9 5 3 true
 4  2 8 2 true
 5  10 10 3 true
 6  6 8 2 true
 7  6 9 3 true
 8  8 7 2 true
 9  5 6 3 true
10  1 6 2 true
11  7 9 3 true
12  3 10 2 true
13  7 9 3 true
14  6 7 2 true
15  3 10 3 true
16  3 7 2 true
17  5 10 3 true
18  3 5 2 true
19  10 1 3 true
```




```
10x50 (500|0|0) : 0 0 0 0 0
    3 0 0 0 0 0 0 0 0 0 1 0 3
     0 2
[rel, 0.0010, 8stp, 1sol]
48 solutions in 0.008 s
```

4.3.13 Datensatz cl1_7 (10x45, 20 Rechtecke)

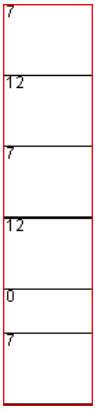
```
-2  10 45
-1  20
 0  10 5 2 false
 1  10 1 3 false
 2  3 7 2 false
 3  4 3 3 false
 4  2 9 2 false
 5  10 2 3 false
 6  3 1 2 false
 7  10 8 3 false
 8  3 8 2 false
 9  7 1 3 false
10  3 7 2 false
11  1 4 3 false
12  8 10 2 false
13  2 7 3 false
14  5 1 2 false
15  4 1 3 false
16  6 8 2 false
17  5 9 3 false
18  5 2 2 false
19  6 9 3 false
```




```
10x45 (450|0|0) : 2 0 0 0 0
    1 0 3 0 0 0 0 0 0 0 0 0 2
     0 0
[rel, 0.0010, 3stp, 16sol]
49 solutions in 0.004 s
```

4.3.14 Datensatz cl1_7r (10x45, 20 Rechtecke, drehbar)

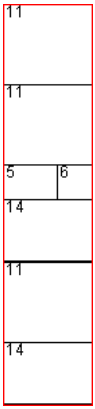
```
-2  10  45
-1  20
 0  10  5  2  true
 1  10  1  3  true
 2  3  7  2  true
 3  4  3  3  true
 4  2  9  2  true
 5  10  2  3  true
 6  3  1  2  true
 7  10  8  3  true
 8  3  8  2  true
 9  7  1  3  true
10  3  7  2  true
11  1  4  3  true
12  8  10  2  true
13  2  7  3  true
14  5  1  2  true
15  4  1  3  true
16  6  8  2  true
17  5  9  3  true
18  5  2  2  true
19  6  9  3  true
```




```
10x45 (450|0|0) : 1 0 0 0 0
    0 0 3 0 0 0 0 2 0 0 0 0 0
     0 0
[rel, 0.0010, 4stp, 4sol]
54 solutions in 0.008 s
```

4.3.15 Datensatz cl1_8 (10x45, 20 Rechtecke)

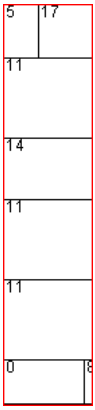
```
-2  10  45
-1  20
 0  9  5  2  false
 1  4  1  3  false
 2  2  8  2  false
 3  9  4  3  false
 4  2  1  2  false
 5  6  4  3  false
 6  4  4  2  false
 7  6  1  3  false
 8  5  1  2  false
 9  5  4  3  false
10  6  2  2  false
11  10  9  3  false
12  2  5  2  false
13  1  3  3  false
14  10  7  2  false
15  4  7  3  false
16  2  5  2  false
17  6  6  3  false
18  2  9  2  false
19  8  7  3  false
```




```
10x45 (450|0|0) : 0 0 0 0 0
    1 1 0 0 0 0 3 0 0 2 0 0 0
     0 0
[rel, 0.01, 3stp, 28sol, cln
    ]
76 solutions in 0.006 s
```

4.3.16 Datensatz cl1_8r (10x45, 20 Rechtecke, drehbar)

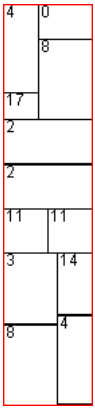
```
-2  10  45
-1  20
 0   9  5  2  true
 1   4  1  3  true
 2   2  8  2  true
 3   9  4  3  true
 4   2  1  2  true
 5   6  4  3  true
 6   4  4  2  true
 7   6  1  3  true
 8   5  1  2  true
 9   5  4  3  true
10   6  2  2  true
11   10  9  3  true
12   2  5  2  true
13   1  3  3  true
14   10  7  2  true
15   4  7  3  true
16   2  5  2  true
17   6  6  3  true
18   2  9  2  true
19   8  7  3  true
```




```
10x45 (450|0|0) : 1 0 0 0 0
    1 0 0 1 0 0 3 0 0 1 0 0 1
     0 0
[rel, 0.0010, 3stp, 86sol,
    cln]
173 solutions in 0.046 s
```

4.3.17 Datensatz cl1_9 (10x45, 20 Rechtecke)

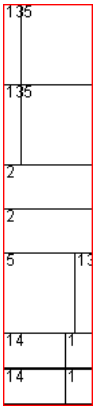
```
-2  10  45
-1  20
 0  6  4  2  false
 1  3  4  3  false
 2  10  5  2  false
 3  6  8  3  false
 4  4  10  2  false
 5  8  9  3  false
 6  7  8  2  false
 7  5  2  3  false
 8  6  9  2  false
 9  9  3  3  false
10  7  9  2  false
11  5  5  3  false
12  1  4  2  false
13  9  2  3  false
14  4  7  2  false
15  4  1  3  false
16  2  10  2  false
17  4  3  3  false
18  8  3  2  false
19  6  4  3  false
```




```
10x45 (450|0|0) : 1 0 2 1 2
    0 0 0 2 0 0 2 0 0 1 0 0 1
     0 0
[rel, 0.0010, 4stp, 106sol,
    cln]
231 solutions in 0.055 s
```

4.3.18 Datensatz cl1_9r (10x45, 20 Rechtecke, drehbar)

```
-2  10  45
-1  20
 0   6  4  2  true
 1   3  4  3  true
 2  10  5  2  true
 3   6  8  3  true
 4   4 10  2  true
 5   8  9  3  true
 6   7  8  2  true
 7   5  2  3  true
 8   6  9  2  true
 9   9  3  3  true
10   7  9  2  true
11   5  5  3  true
12   1  4  2  true
13   9  2  3  true
14   4  7  2  true
15   4  1  3  true
16   2 10  2  true
17   4  3  3  true
18   8  3  2  true
19   6  4  3  true
```

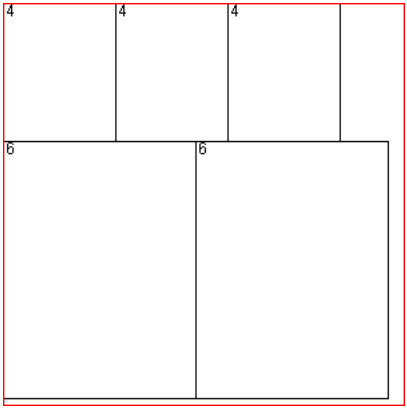



```
10x45 (450|0|0) : 0 2 2 0 0
    3 0 0 0 0 0 0 0 3 2 0 0 0
     0 0
[rel, 0.0010, 694sol]
1366 solutions in 13.137 s
```

4.4 Datensätze gcut

4.4.1 Datensatz gcut1 (250x250, 10 Rechtecktypen)

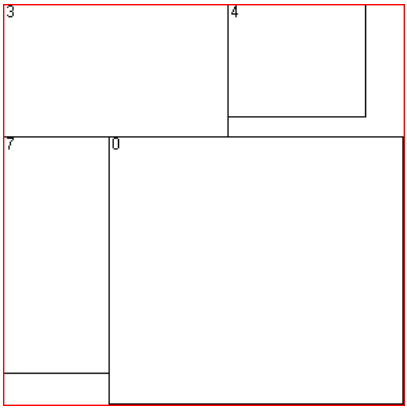
```
-2  250 250
-1  10
 0  167 184 30728 false
 1  114 118 13452 false
 2  167 152 25384 false
 3  83 140 11620 false
 4  70 86 6020 false
 5  143 166 23738 false
 6  120 160 19200 false
 7  66 148 9768 false
 8  87 141 12267 false
 9  69 165 11385 false
```




```
240x246 (56460|2580|6040) :
    0 0 0 0 3 0 2 0 0 0
[rel, 0.044, 3stp, 9sol]
30 solutions in 0.002 s
```

4.4.2 Datensatz gcut1r (250x250, 10 Rechtecktypen, drehbar)

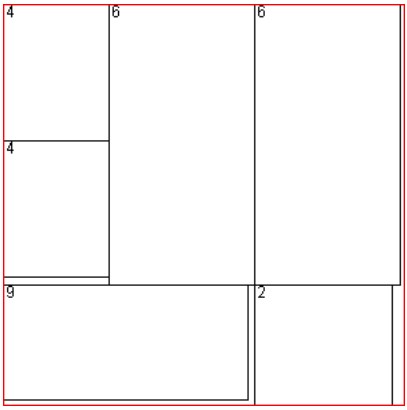
```
-2  250  250
-1  10
 0  167  184  30728  true
 1  114  118  13452  true
 2  167  152  25384  true
 3  83  140  11620  true
 4  70  86  6020  true
 5  143  166  23738  true
 6  120  160  19200  true
 7  66  148  9768  true
 8  87  141  12267  true
 9  69  165  11385  true
```




```
250x250 (58136|4364|4364) :
    1 0 0 1 1 0 0 1 0 0
[rel, 0.07, 2stp, 380sol,
    cln]
283 solutions in 0.017 s
```

4.4.3 Datensatz gcut2 (250x250, 20 Rechtecktypen)

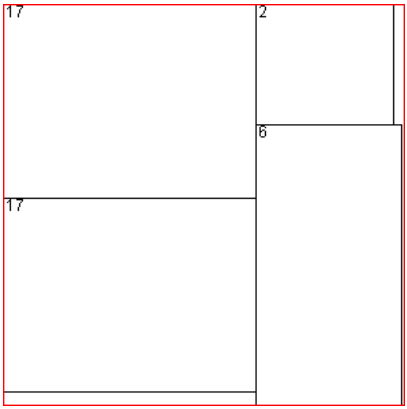
```
-2  250 250
-1  20
 0  120 133 15960 false
 1  135 186 25110 false
 2  86 75 6450 false
 3  103 73 7519 false
 4  66 85 5610 false
 5  135 97 13095 false
 6  91 175 15925 false
 7  131 176 23056 false
 8  71 176 12496 false
 9  153 72 11016 false
10  87 148 12876 false
11  168 107 17976 false
12  118 90 10620 false
13  140 109 15260 false
14  132 159 20988 false
15  152 93 14136 false
16  135 68 9180 false
17  121 158 19118 false
18  68 94 6392 false
19  155 76 11780 false
```




```
248x250 (60536|1464|1964) :
    0 0 1 0 2 0 2 0 0 1 0 0 0
      0 0 0 0 0 0 0
[rel, 0.024, 4stp, 78sol]
146 solutions in 0.019 s
```

4.4.4 Datensatz gcut2r (250x250, 20 Rechtecktypen, drehbar)

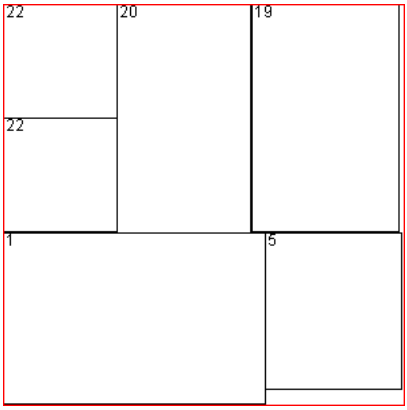
```
-2  250 250
-1  20
 0  120 133 15960 true
 1  135 186 25110 true
 2  86 75 6450 true
 3  103 73 7519 true
 4  66 85 5610 true
 5  135 97 13095 true
 6  91 175 15925 true
 7  131 176 23056 true
 8  71 176 12496 true
 9  153 72 11016 true
10  87 148 12876 true
11  168 107 17976 true
12  118 90 10620 true
13  140 109 15260 true
14  132 159 20988 true
15  152 93 14136 true
16  135 68 9180 true
17  121 158 19118 true
18  68 94 6392 true
19  155 76 11780 true
```




```
249x250 (60611|1639|1889) :
    0 0 1 0 0 0 1 0 0 0 0 0 0
     0 0 0 0 2 0 0
[rel, 0.027, 2stp, 944sol,
    cln]
652 solutions in 0.078 s
```

4.4.5 Datensatz gcut3 (250x250, 30 Rechtecktypen)

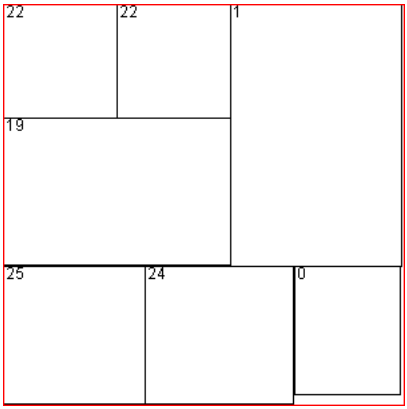
```
-2  250 250
-1  30
 0  66 80 5280 false
 1  164 107 17548 false
 2  64 184 11776 false
 3  121 86 10406 false
 4  163 135 22005 false
 5  85 98 8330 false
 6  81 102 8262 false
 7  103 186 19158 false
 8  152 106 16112 false
 9  176 139 24464 false
10  111 118 13098 false
11  69 169 11661 false
12  146 133 19418 false
13  112 112 12544 false
14  133 160 21280 false
15  63 129 8127 false
16  163 152 24776 false
17  110 155 17050 false
18  96 136 13056 false
19  92 142 13064 false
20  84 143 12012 false
21  119 133 15827 false
22  71 71 5041 false
23  146 84 12264 false
24  93 86 7998 false
25  89 86 7654 false
26  101 146 14746 false
27  172 73 12556 false
28  73 169 12337 false
29  99 99 9801 false
```




```
249x250 (61036|1214|1464) :
    0 1 0 0 0 1 0 0 0 0 0 0 0
      0 0 0 0 0 0 1 1 0 2 0 0
    0 0 0 0 0
[rel, 0.029, 3stp, 546sol]
673 solutions in 0.210 s
```

4.4.6 Datensatz gcut3r (250x250, 30 Rechtecktypen, drehbar)

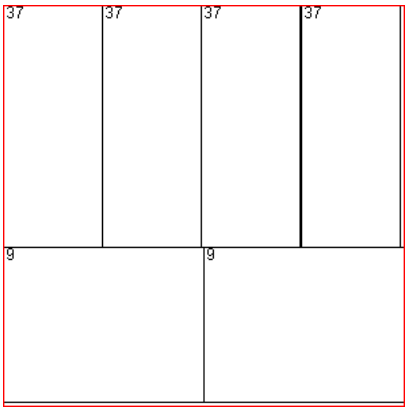
```
-2  250 250
-1  30
 0  66 80 5280 true
 1  164 107 17548 true
 2  64 184 11776 true
 3  121 86 10406 true
 4  163 135 22005 true
 5  85 98 8330 true
 6  81 102 8262 true
 7  103 186 19158 true
 8  152 106 16112 true
 9  176 139 24464 true
10  111 118 13098 true
11  69 169 11661 true
12  146 133 19418 true
13  112 112 12544 true
14  133 160 21280 true
15  63 129 8127 true
16  163 152 24776 true
17  110 155 17050 true
18  96 136 13056 true
19  92 142 13064 true
20  84 143 12012 true
21  119 133 15827 true
22  71 71 5041 true
23  146 84 12264 true
24  93 86 7998 true
25  89 86 7654 true
26  101 146 14746 true
27  172 73 12556 true
28  73 169 12337 true
29  99 99 9801 true
```




```
249x250 (61626|624|874) : 1
    1 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 1 0 0 2 0 1 1
    0 0 0 0
[rel, 0.021, 4stp, 678sol,
    cln]
1310 solutions in 1.794 s
```

4.4.7 Datensatz gcut4 (250x250, 50 Rechtecktypen)

```
-2  250 250
-1  50
 0  139 103 14317 false
 1  166 99 16434 false
 2  88 74 6512 false
 3  174 139 24186 false
 4  128 109 13952 false
 5  175 179 31325 false
 6  62 82 5084 false
 7  178 87 15486 false
 8  121 77 9317 false
 9  125 97 12125 false
10  73 84 6132 false
11  69 97 6693 false
12  122 159 19398 false
13  66 69 4554 false
14  111 118 13098 false
15  165 86 14190 false
16  71 83 5893 false
17  163 162 26406 false
18  67 132 8844 false
19  121 152 18392 false
20  68 150 10200 false
21  138 124 17112 false
22  141 105 14805 false
23  183 122 22326 false
24  171 63 10773 false
25  101 64 6464 false
26  103 95 9785 false
27  110 107 11770 false
28  184 158 29072 false
29  85 97 8245 false
30  86 92 7912 false
31  127 155 19685 false
32  92 147 13524 false
33  106 97 10282 false
34  177 108 19116 false
```

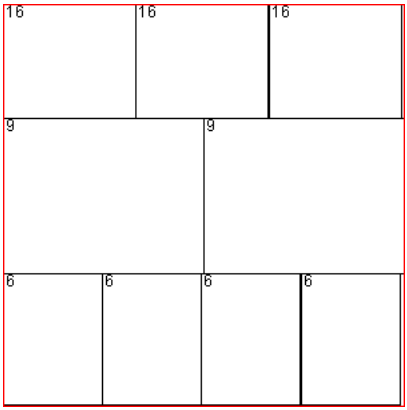



```
250x248 (61698|302|802) : 0
    0 0 0 0 0 0 0 0 2 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 4 0
     0 0 0 0 0 0 0 0 0 0 0
[rel, 0.0050, 3stp, 80sol,
    cln]
196 solutions in 0.059 s
```

```
35  164 150 24600 false
36  68 84 5712 false
37  62 151 9362 false
38  123 134 16482 false
39  158 178 28124 false
40  70 140 9800 false
41  173 117 20241 false
42  147 136 19992 false
43  141 145 20445 false
44  84 164 13776 false
45  144 141 20304 false
46  92 103 9476 false
47  156 98 15288 false
48  160 111 17760 false
49  127 131 16637 false
```

4.4.8 Datensatz gcut4r (250x250, 50 Rechtecktypen, drehbar)

```
-2  250 250
-1  50
 0  139 103 14317 true
 1  166 99 16434 true
 2  88 74 6512 true
 3  174 139 24186 true
 4  128 109 13952 true
 5  175 179 31325 true
 6  62 82 5084 true
 7  178 87 15486 true
 8  121 77 9317 true
 9  125 97 12125 true
10  73 84 6132 true
11  69 97 6693 true
12  122 159 19398 true
13  66 69 4554 true
14  111 118 13098 true
15  165 86 14190 true
16  71 83 5893 true
17  163 162 26406 true
18  67 132 8844 true
19  121 152 18392 true
20  68 150 10200 true
21  138 124 17112 true
22  141 105 14805 true
23  183 122 22326 true
24  171 63 10773 true
25  101 64 6464 true
26  103 95 9785 true
27  110 107 11770 true
28  184 158 29072 true
29  85 97 8245 true
30  86 92 7912 true
31  127 155 19685 true
32  92 147 13524 true
33  106 97 10282 true
```

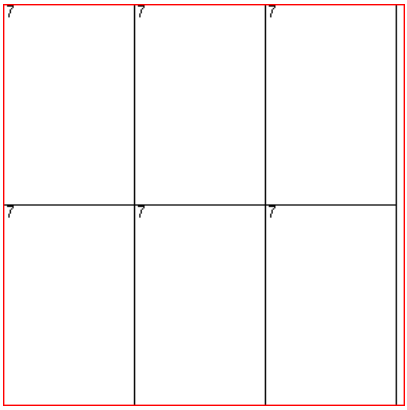



```
250x250 (62265|235|235) : 0
    0 0 0 0 0 4 0 0 2 0 0 0 0
     0 0 3 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 0 0 0 0 0
[rel, 0.0040, 4stp, 172sol,
    cln]
500 solutions in 0.449 s
```

```
34  177 108 19116 true
35  164 150 24600 true
36  68 84 5712 true
37  62 151 9362 true
38  123 134 16482 true
39  158 178 28124 true
40  70 140 9800 true
41  173 117 20241 true
42  147 136 19992 true
43  141 145 20445 true
44  84 164 13776 true
45  144 141 20304 true
46  92 103 9476 true
47  156 98 15288 true
48  160 111 17760 true
49  127 131 16637 true
```

4.4.9 Datensatz gcut5 (500x500, 10 Rechtecktypen)

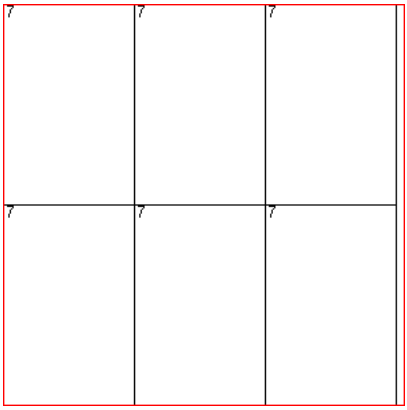
```
-2  500 500
-1  10
 0  198 205 40590 false
 1  179 155 27745 false
 2  364 236 85904 false
 3  272 147 39984 false
 4  352 145 51040 false
 5  343 245 84035 false
 6  132 174 22968 false
 7  164 250 41000 false
 8  282 356 100392 false
 9  342 151 51642 false
```




```
492x500 (246000|0|4000) : 0
    0 0 0 0 0 0 6 0 0
[rel, 0.0010, 3stp, 8sol]
21 solutions in 0.001 s
```

4.4.10 Datensatz gcut5r (500x500, 10 Rechtecktypen, drehbar)

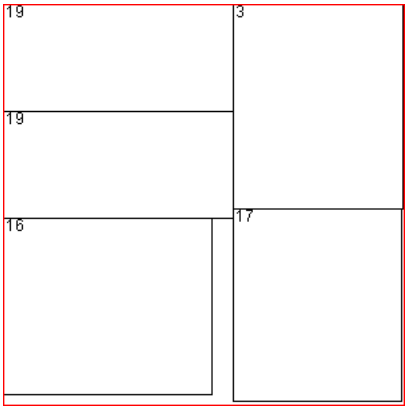
```
-2  500  500
-1  10
 0  198  205  40590  true
 1  179  155  27745  true
 2  364  236  85904  true
 3  272  147  39984  true
 4  352  145  51040  true
 5  343  245  84035  true
 6  132  174  22968  true
 7  164  250  41000  true
 8  282  356  100392  true
 9  342  151  51642  true
```



```
492x500 (246000|0|4000) : 0
    0 0 0 0 0 0 6 0 0
[rel, 0.0010, 3stp, 16sol]
42 solutions in 0.003 s
```

4.4.11 Datensatz gcut6 (500x500, 20 Rechtecktypen)

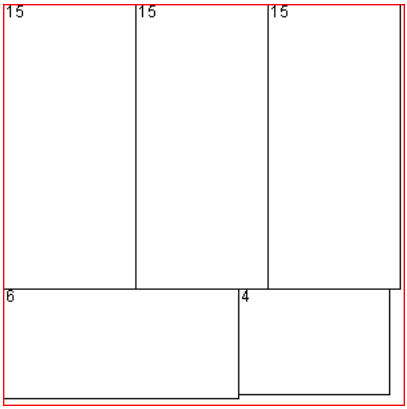
```
-2  500  500
-1  20
 0  313  305  95465  false
 1  292  222  64824  false
 2  330  253  83490  false
 3  212  256  54272  false
 4  132  189  24948  false
 5  149  296  44104  false
 6  294  137  40278  false
 7  225  345  77625  false
 8  345  220  75900  false
 9  337  177  59649  false
10  189  300  56700  false
11  234  321  75114  false
12  335  272  91120  false
13  354  244  86376  false
14  149  169  25181  false
15  355  165  58575  false
16  260  220  57200  false
17  210  241  50610  false
18  194  372  72168  false
19  287  134  38458  false
```




```
499x497 (238998|9005|11002)
    : 0 0 0 0 1 0 0 0 0 0 0 0 0
      0 0 0 0 1 1 0 2
[rel, 0.043, 3stp, 86sol,
    cln]
148 solutions in 0.013 s
```

4.4.12 Datensatz gcut6r (500x500, 20 Rechtecktypen, drehbar)

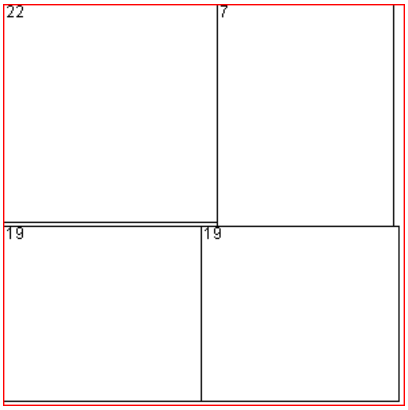
```
-2  500  500
-1  20
 0  313  305  95465  true
 1  292  222  64824  true
 2  330  253  83490  true
 3  212  256  54272  true
 4  132  189  24948  true
 5  149  296  44104  true
 6  294  137  40278  true
 7  225  345  77625  true
 8  345  220  75900  true
 9  337  177  59649  true
10  189  300  56700  true
11  234  321  75114  true
12  335  272  91120  true
13  354  244  86376  true
14  149  169  25181  true
15  355  165  58575  true
16  260  220  57200  true
17  210  241  50610  true
18  194  372  72168  true
19  287  134  38458  true
```




```
495x492 (240951|2589|9049) :
    0 0 0 0 1 0 1 0 0 0 0 0
  0 0 0 3 0 0 0 0
[rel, 0.015, 3stp, 129sol,
  cln]
229 solutions in 0.032 s
```

4.4.13 Datensatz gcut7 (500x500, 30 Rechtecktypen)

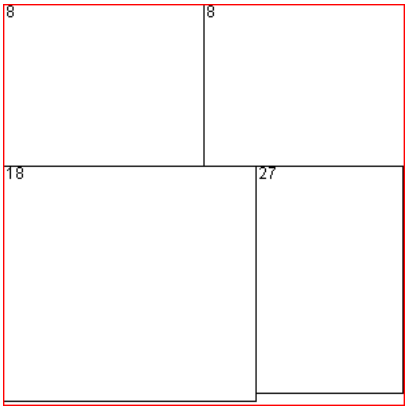
```
-2  500 500
-1  30
 0  348 220 76560 false
 1  255 184 46920 false
 2  191 249 47559 false
 3  212 194 41128 false
 4  348 322 112056 false
 5  171 369 63099 false
 6  206 327 67362 false
 7  221 277 61217 false
 8  250 202 50500 false
 9  205 226 46330 false
10  193 280 54040 false
11  364 311 113204 false
12  264 224 59136 false
13  244 347 84668 false
14  270 338 91260 false
15  224 236 52864 false
16  168 177 29736 false
17  285 347 98895 false
18  315 294 92610 false
19  247 219 54093 false
20  315 302 95130 false
21  129 147 18963 false
22  268 273 73164 false
23  350 352 123200 false
24  186 258 47988 false
25  365 374 136510 false
26  145 259 37555 false
27  284 184 52256 false
28  129 198 25542 false
29  146 351 51246 false
```




```
494x496 (242567|2457|7433) :
    0 0 0 0 0 0 0 1 0 0 0 0
  0 0 0 0 0 0 0 2 0 0 1 0 0
    0 0 0 0 0
[rel, 0.011, 2stp, 70sol,
  cln]
99 solutions in 0.007 s
```

4.4.14 Datensatz gcut7r (500x500, 30 Rechtecktypen, drehbar)

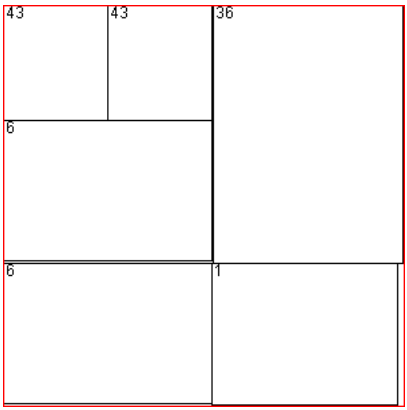
```
-2  500 500
-1  30
 0  348 220 76560 true
 1  255 184 46920 true
 2  191 249 47559 true
 3  212 194 41128 true
 4  348 322 112056 true
 5  171 369 63099 true
 6  206 327 67362 true
 7  221 277 61217 true
 8  250 202 50500 true
 9  205 226 46330 true
10  193 280 54040 true
11  364 311 113204 true
12  264 224 59136 true
13  244 347 84668 true
14  270 338 91260 true
15  224 236 52864 true
16  168 177 29736 true
17  285 347 98895 true
18  315 294 92610 true
19  247 219 54093 true
20  315 302 95130 true
21  129 147 18963 true
22  268 273 73164 true
23  350 352 123200 true
24  186 258 47988 true
25  365 374 136510 true
26  145 259 37555 true
27  284 184 52256 true
28  129 198 25542 true
29  146 351 51246 true
```




```
500x496 (245866|2134|4134) :
    0 0 0 0 0 0 0 0 2 0 0 0
  0 0 0 0 0 0 1 0 0 0 0 0 0
    0 0 1 0 0
[rel, 0.013, 2stp, 366sol,
  cln]
368 solutions in 0.055 s
```

4.4.15 Datensatz gcut8 (500x500, 50 Rechtecktypen)

```
-2  500  500
-1  50
 0  277  242  67034  false
 1  233  177  41241  false
 2  182  216  39312  false
 3  147  134  19698  false
 4  213  127  27051  false
 5  315  212  66780  false
 6  261  175  45675  false
 7  210  281  59010  false
 8  243  256  62208  false
 9  159  314  49926  false
10  217  147  31899  false
11  260  349  90740  false
12  222  199  44178  false
13  296  127  37592  false
14  263  302  79426  false
15  226  250  56500  false
16  264  344  90816  false
17  237  179  42423  false
18  245  137  33565  false
19  137  328  44936  false
20  321  154  49434  false
21  200  174  34800  false
22  270  128  34560  false
23  299  179  53521  false
24  165  370  61050  false
25  198  182  36036  false
26  155  295  45725  false
27  278  219  60882  false
28  184  155  28520  false
29  187  270  50490  false
30  208  293  60944  false
31  261  204  53244  false
32  344  168  57792  false
33  213  135  28755  false
34  362  162  58644  false
```

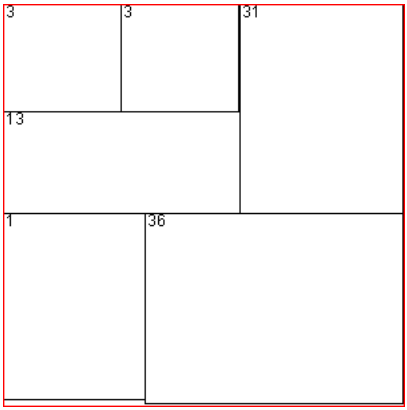



```
499x499 (246633|2368|3367) :
    0 1 0 0 0 0 2 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 1
  0 0 0 0 0 0 2 0 0 0 0 0 0
[rel, 0.01, 4stp, 144sol,
  cln]
299 solutions in 0.128 s
```

```
35  362 374 135388 false
36  237 322 76314 false
37  281 308 86548 false
38  155 344 53320 false
39  210 163 34230 false
40  325 225 73125 false
41  360 225 81000 false
42  346 347 120062 false
43  131 144 18864 false
44  284 284 80656 false
45  262 228 59736 false
46  146 178 25988 false
47  254 164 41656 false
48  332 238 79016 false
49  313 304 95152 false
```

4.4.16 Datensatz gcut8r (500x500, 50 Rechtecktypen, drehbar)

```
-2  500 500
-1  50
 0  277 242 67034 true
 1  233 177 41241 true
 2  182 216 39312 true
 3  147 134 19698 true
 4  213 127 27051 true
 5  315 212 66780 true
 6  261 175 45675 true
 7  210 281 59010 true
 8  243 256 62208 true
 9  159 314 49926 true
10  217 147 31899 true
11  260 349 90740 true
12  222 199 44178 true
13  296 127 37592 true
14  263 302 79426 true
15  226 250 56500 true
16  264 344 90816 true
17  237 179 42423 true
18  245 137 33565 true
19  137 328 44936 true
20  321 154 49434 true
21  200 174 34800 true
22  270 128 34560 true
23  299 179 53521 true
24  165 370 61050 true
25  198 182 36036 true
26  155 295 45725 true
27  278 219 60882 true
28  184 155 28520 true
29  187 270 50490 true
30  208 293 60944 true
31  261 204 53244 true
32  344 168 57792 true
33  213 135 28755 true
```

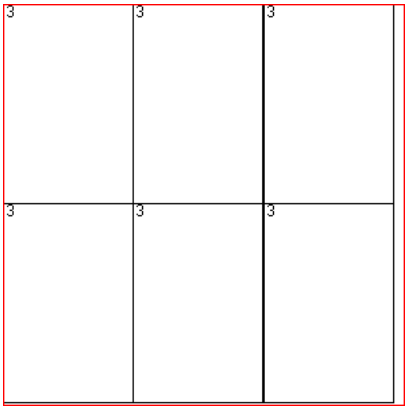



```
500x498 (247787|1213|2213) :
    0 1 0 2 0 0 0 0 0 0 0 0
   0 1 0 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 1 0 0 0 0 1
   0 0 0 0 0 0 0 0 0 0 0 0 0
[rel, 0.0060, 4stp, 384sol,
   cln]
804 solutions in 0.848 s
```

```
34  362  162  58644  true
35  362  374  135388  true
36  237  322  76314  true
37  281  308  86548  true
38  155  344  53320  true
39  210  163  34230  true
40  325  225  73125  true
41  360  225  81000  true
42  346  347  120062  true
43  131  144  18864  true
44  284  284  80656  true
45  262  228  59736  true
46  146  178  25988  true
47  254  164  41656  true
48  332  238  79016  true
49  313  304  95152  true
```

4.4.17 Datensatz gcut9 (1000x1000, 10 Rechtecktypen)

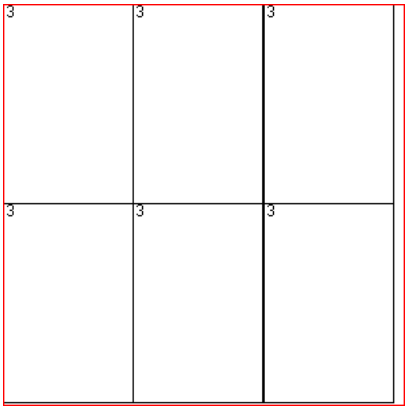
```
-2  1000 1000
-1  10
 0  310 426 132060 false
 1  673 468 314964 false
 2  426 463 197238 false
 3  325 498 161850 false
 4  555 540 299700 false
 5  292 455 132860 false
 6  343 341 116963 false
 7  362 491 177742 false
 8  305 688 209840 false
 9  459 607 278613 false
```




```
975x996 (971100|0|28900) : 0
    0 0 6 0 0 0 0 0 0
[rel, 0.0010, 3stp, 10sol,
    cln]
32 solutions in 0.002 s
```

4.4.18 Datensatz gcut9r (1000x1000, 10 Rechtecktypen, drehbar)

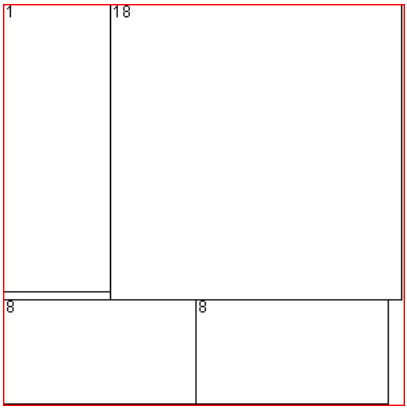
```
-2  1000  1000
-1  10
 0  310  426  132060  true
 1  673  468  314964  true
 2  426  463  197238  true
 3  325  498  161850  true
 4  555  540  299700  true
 5  292  455  132860  true
 6  343  341  116963  true
 7  362  491  177742  true
 8  305  688  209840  true
 9  459  607  278613  true
```




```
975x996 (971100|0|28900) : 0
     0 0 6 0 0 0 0 0 0
[rel, 0.0010, 3stp, 20sol,
    cln]
65 solutions in 0.004 s
```

4.4.19 Datensatz gcut10 (1000x1000, 20 Rechtecktypen)

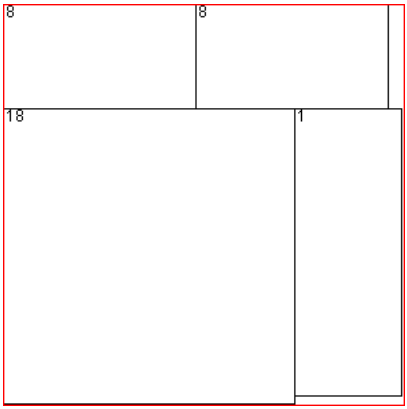
```
-2  1000  1000
-1  20
 0  730  300  219000  false
 1  269  717  192873  false
 2  463  369  170847  false
 3  642  464  297888  false
 4  453  329  149037  false
 5  455  667  303485  false
 6  506  482  243892  false
 7  560  362  202720  false
 8  483  260  125580  false
 9  693  345  239085  false
10  381  510  194310  false
11  456  586  267216  false
12  457  453  207021  false
13  707  658  465206  false
14  639  650  415350  false
15  691  359  248069  false
16  434  700  303800  false
17  576  291  167616  false
18  728  739  537992  false
19  545  742  404390  false
```




```
997x999 (982025|13978|17975)
     : 0 1 0 0 0 0 0 0 2 0 0
    0 0 0 0 0 0 0 1 0
[rel, 0.015, 2stp, 46sol,
    cln]
66 solutions in 0.003 s
```

4.4.20 Datensatz gcut10r (1000x1000, 20 Rechtecktypen, drehbar)

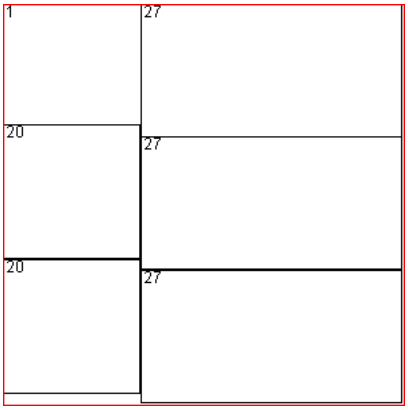
```
-2  1000  1000
-1  20
 0  730  300  219000  true
 1  269  717  192873  true
 2  463  369  170847  true
 3  642  464  297888  true
 4  453  329  149037  true
 5  455  667  303485  true
 6  506  482  243892  true
 7  560  362  202720  true
 8  483  260  125580  true
 9  693  345  239085  true
10  381  510  194310  true
11  456  586  267216  true
12  457  453  207021  true
13  707  658  465206  true
14  639  650  415350  true
15  691  359  248069  true
16  434  700  303800  true
17  576  291  167616  true
18  728  739  537992  true
19  545  742  404390  true
```




```
997x999 (982025|13978|17975)
     : 0 1 0 0 0 0 0 0 2 0 0
    0 0 0 0 0 0 0 1 0
[rel, 0.015, 2stp, 220sol,
    cln]
228 solutions in 0.017 s
```

4.4.21 Datensatz gcut11 (1000x1000, 30 Rechtecktypen)

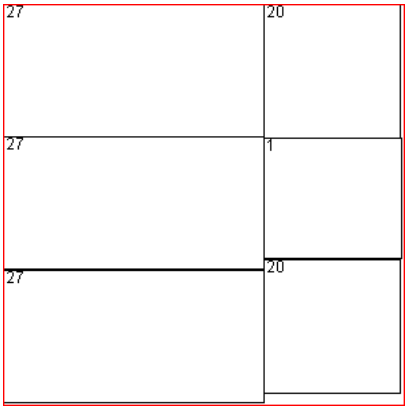
```
-2  1000 1000
-1  30
 0  541 745 403045 false
 1  344 301 103544 false
 2  413 294 121422 false
 3  266 341 90706 false
 4  543 388 210684 false
 5  367 701 257267 false
 6  526 707 371882 false
 7  286 706 201916 false
 8  614 289 177446 false
 9  348 592 206016 false
10  673 431 290063 false
11  475 362 171950 false
12  562 439 246718 false
13  530 638 338140 false
14  609 375 228375 false
15  540 274 147960 false
16  486 634 308124 false
17  272 377 102544 false
18  623 306 190638 false
19  266 714 189924 false
20  341 336 114576 false
21  340 296 100640 false
22  531 479 254349 false
23  571 589 336319 false
24  674 496 334304 false
25  560 497 278320 false
26  509 389 198001 false
27  650 332 215800 false
28  474 285 135090 false
29  419 503 210757 false
```




```
994x996 (980096|9928|19904)
    : 0 1 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 2 0 0 0
    0 0 0 3 0 0
[rel, 0.011, 3stp, 172sol,
    cln]
262 solutions in 0.041 s
```

4.4.22 Datensatz gcut11r (1000x1000, 30 Rechtecktypen, drehbar)

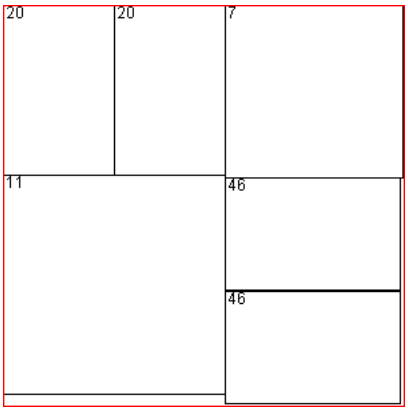
```
-2  1000  1000
-1  30
 0  541  745  403045  true
 1  344  301  103544  true
 2  413  294  121422  true
 3  266  341  90706  true
 4  543  388  210684  true
 5  367  701  257267  true
 6  526  707  371882  true
 7  286  706  201916  true
 8  614  289  177446  true
 9  348  592  206016  true
10  673  431  290063  true
11  475  362  171950  true
12  562  439  246718  true
13  530  638  338140  true
14  609  375  228375  true
15  540  274  147960  true
16  486  634  308124  true
17  272  377  102544  true
18  623  306  190638  true
19  266  714  189924  true
20  341  336  114576  true
21  340  296  100640  true
22  531  479  254349  true
23  571  589  336319  true
24  674  496  334304  true
25  560  497  278320  true
26  509  389  198001  true
27  650  332  215800  true
28  474  285  135090  true
29  419  503  210757  true
```




```
994x996 (980096|9928|19904)
    : 0 1 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 2 0 0 0
    0 0 0 3 0 0
[rel, 0.011, 3stp, 1470sol,
    cln]
1315 solutions in 0.552 s
```

4.4.23 Datensatz gcut12 (1000x1000, 50 Rechtecktypen)

```
-2  1000 1000
-1  50
 0  572 665 380380 false
 1  482 640 308480 false
 2  264 594 156816 false
 3  349 566 197534 false
 4  276 660 182160 false
 5  745 422 314390 false
 6  434 622 269948 false
 7  446 433 193118 false
 8  668 506 338008 false
 9  405 635 257175 false
10  486 320 155520 false
11  554 549 304146 false
12  392 491 192472 false
13  459 528 242352 false
14  426 579 246654 false
15  728 359 261352 false
16  644 465 299460 false
17  382 323 123386 false
18  268 352 94336 false
19  324 512 165888 false
20  277 426 118002 false
21  254 685 173990 false
22  483 532 256956 false
23  678 414 280692 false
24  546 590 322140 false
25  716 640 458240 false
26  542 400 216800 false
27  634 596 377864 false
28  497 696 345912 false
29  417 300 125100 false
30  290 605 175450 false
31  321 520 166920 false
32  543 290 157470 false
33  599 269 161131 false
34  467 679 317093 false
```

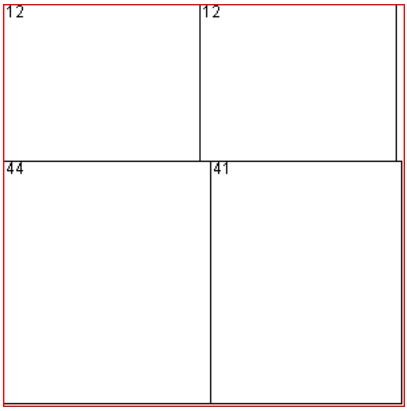



```
1000x995
    (979986|15014|20014) : 0
    0 0 0 0 0 0 1 0 0 0 1 0 0
     0 0 0 0 0 0 2 0 0 0 0 0
    0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0 0 0 0 0 0 2 0 0 0
[rel, 0.016, 3stp, 326sol,
    cln]
511 solutions in 0.246 s
```

```
35  583  421  245443  false
36  729  723  527067  false
37  465  590  274350  false
38  614  585  359190  false
39  446  327  145842  false
40  734  616  452144  false
41  479  606  290274  false
42  656  278  182368  false
43  537  355  190635  false
44  604  519  313476  false
45  746  596  444616  false
46  439  281  123359  false
47  339  405  137295  false
48  273  526  143598  false
49  625  568  355000  false
```

4.4.24 Datensatz gcut12r (1000x1000, 50 Rechtecktypen, drehbar)

```
-2  1000 1000
-1  50
 0  572  665  380380  true
 1  482  640  308480  true
 2  264  594  156816  true
 3  349  566  197534  true
 4  276  660  182160  true
 5  745  422  314390  true
 6  434  622  269948  true
 7  446  433  193118  true
 8  668  506  338008  true
 9  405  635  257175  true
10  486  320  155520  true
11  554  549  304146  true
12  392  491  192472  true
13  459  528  242352  true
14  426  579  246654  true
15  728  359  261352  true
16  644  465  299460  true
17  382  323  123386  true
18  268  352  94336  true
19  324  512  165888  true
20  277  426  118002  true
21  254  685  173990  true
22  483  532  256956  true
23  678  414  280692  true
24  546  590  322140  true
25  716  640  458240  true
26  542  400  216800  true
27  634  596  377864  true
28  497  696  345912  true
29  417  300  125100  true
30  290  605  175450  true
31  321  520  166920  true
32  543  290  157470  true
33  599  269  161131  true
```

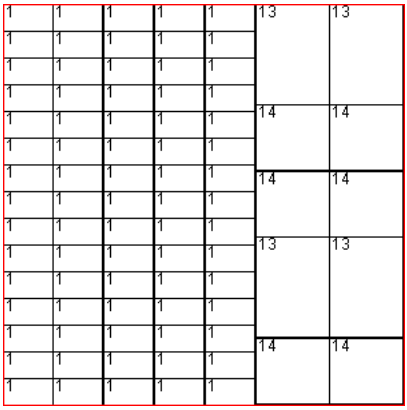



```
998x998 (988694|7310|11306)
    : 0 0 0 0 0 0 0 0 0 0 0 0
      2 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0
       0 0 0 0 1 0 0 1 0 0 0 0
      0
[rel, 0.0080, 2stp, 860sol,
    cln]
764 solutions in 0.290 s
```

34	467	679	317093	true
35	583	421	245443	true
36	729	723	527067	true
37	465	590	274350	true
38	614	585	359190	true
39	446	327	145842	true
40	734	616	452144	true
41	479	606	290274	true
42	656	278	182368	true
43	537	355	190635	true
44	604	519	313476	true
45	746	596	444616	true
46	439	281	123359	true
47	339	405	137295	true
48	273	526	143598	true
49	625	568	355000	true

4.4.25 Datensatz gcut13 (3000x3000, 32 Rechtecktypen)

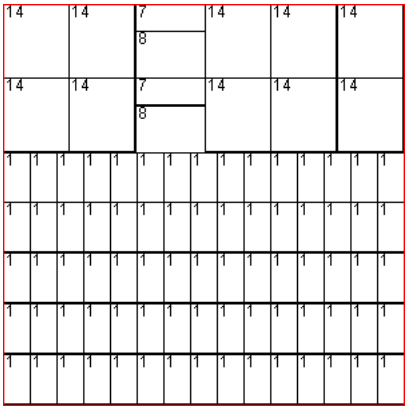
```
-2  3000 3000
-1  32
 0  365 185 67525 false
 1  378 200 75600 false
 2  410 165 67650 false
 3  425 148 62900 false
 4  425 296 125800 false
 5  439 116 50924 false
 6  464 1006 466784 false
 7  520 205 106600 false
 8  520 350 182000 false
 9  540 530 286200 false
10  549 1413 775737 false
11  549 1882 1033218 false
12  553 496 274288 false
13  555 755 419025 false
14  555 496 275280 false
15  555 659 365745 false
16  567 473 268191 false
17  572 592 338624 false
18  572 975 557700 false
19  572 1175 672100 false
20  572 1575 900900 false
21  572 1390 795080 false
22  572 1490 852280 false
23  572 1590 909480 false
24  572 1690 966680 false
25  572 1890 1081080 false
26  610 625 381250 false
27  660 490 323400 false
28  690 447 308430 false
29  949 445 422305 false
30  949 478 453622 false
31  970 463 449110 false
```




```
3000x3000
    (8997780|2220|2220) : 0
    75 0 0 0 0 0 0 0 0 0 0 0
    4 6 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0
[rel, 0.015, 9stp, 1500sol,
    cln]
6986 solutions in 2:39 h
```

4.4.26 Datensatz gcut13r (3000x3000, 32 Rechtecktypen, drehbar)

```
-2  3000 3000
-1  32
 0  365 185 67525 true
 1  378 200 75600 true
 2  410 165 67650 true
 3  425 148 62900 true
 4  425 296 125800 true
 5  439 116 50924 true
 6  464 1006 466784 true
 7  520 205 106600 true
 8  520 350 182000 true
 9  540 530 286200 true
10  549 1413 775737 true
11  549 1882 1033218 true
12  553 496 274288 true
13  555 755 419025 true
14  555 496 275280 true
15  555 659 365745 true
16  567 473 268191 true
17  572 592 338624 true
18  572 975 557700 true
19  572 1175 672100 true
20  572 1575 900900 true
21  572 1390 795080 true
22  572 1490 852280 true
23  572 1590 909480 true
24  572 1690 966680 true
25  572 1890 1081080 true
26  610 625 381250 true
27  660 490 323400 true
28  690 447 308430 true
29  949 445 422305 true
30  949 478 453622 true
31  970 463 449110 true
```



```
3000x3000 (9000000|0|0) : 0
    75 0 0 0 0 0 2 2 0 0 0 0
    0 10 0 0 0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0
[rel, 0.0010, 7stp, 2500sol,
     cln]
7764 solutions in 13:27 m
```

4.4.27 Zusammenfassung

Datensatz	absolute Fläche der optimalen Lösung	prozentualer Verschnitt der optimalen Lösung	Berechnungszeit der optimalen Lösung
Wang	2721	2,821%	0,010s
Wangr	2737	2,250%	0,032s
cl1_1	550	0	0,013s
cl1_1r	550	0	0,079s
cl1_2	300	0	0,054s
cl1_2r	300	0	0,102s
cl1_3	450	0	1:34m
cl1_3r	450	0	0,006s
cl1_4	400	0	0,010s
cl1_4r	400	0	0,084s
cl1_5	450	0	0,048s
cl1_5r	450	0	0,008s
cl1_6	500	0	0,009s
cl1_6r	500	0	0,008s
cl1_7	450	0	0,004s
cl1_7r	450	0	0,008s
cl1_8	450	0	0,006s
cl1_8r	450	0	0,046s
cl1_9	450	0	0,055s
cl1_9r	450	0	13,137s

Datensatz	absolute Fläche der optimalen Lösung	prozentualer Verschnitt der optimalen Lösung	Berechnungszeit der optimalen Lösung
gcut1	56460	9,664%	0,002s
gcut1r	58136	6,982%	0,017s
gcut2	60536	3,142%	0,019s
gcut2r	60611	3,022%	0,078s
gcut3	61036	2,342%	0,210s
gcut3r	61626	1,398%	1,794s
gcut4	61698	1,283%	0,059s
gcut4r	62265	0,376%	0,449s
gcut5	246000	1,600%	0,001s
gcut5r	246000	1,600%	0,003s
gcut6	238998	4,401%	0,013s
gcut6r	240951	3,620%	0,032s
gcut7	242567	2,973%	0,007s
gcut7r	245866	1,654%	0,055s
gcut8	246633	1,347%	0,128s
gcut8r	247787	0,885%	0,848s
gcut9	971100	2,890%	0,002s
gcut9r	971100	2,890%	0,004s
gcut10	982025	1,798%	0,003s
gcut10r	982025	1,798%	0,017s
gcut11	980096	1,990%	0,041s
gcut11r	980096	1,990%	0,552s
gcut12	979986	2,001%	0,246s
gcut12r	988694	1,131%	0,290s
gcut13	8997780	0,025%	2:39h
gcut13r	9000000	0	13:27m

5 Folgerung und Ausblick

Das vorgestellte Programm ist in der Lage, viele verschiedenartige 2D-Guillotine-Zuschnittprobleme in kurzer Zeit zu lösen – die Zeitkomplexität des Algorithmus ist $\mathcal{O}(n^2m^2)$ für die Anzahl n der Rechtecke und die maximale Anzahl m von Zwischenergebnissen.
Die Parameter Fehlerschranke, Anzahl der Schritte des Algorithmus und maximale Anzahl der Zwischenergebnisse bestimmen die Laufzeit des Algorithmus maßgeblich, genauer führt eine Verkleinerung der Parameter zu einer Verringerung der Laufzeit. Durch eine Optimierung des Codes, wie zum Beispiel durch Entfernen der Grafikfähigkeiten oder Drehbarkeitsbehandlungen, könnte eine verbesserte Laufzeit des Programms erreicht werden. Für praktische Zwecke sollte das Programm mit seinem Laufzeitverhalten jedoch ausreichen, gerade dort ist eine Grafikausgabefähigkeit meist auch gewünscht.
Die Bereinigungsfunktion `removeTooMuch` und der Überflüssigkeitstest `!contains` entfernen im Fall der starken Bereinigung (d.h. nicht nur äquivalente Lösungen, sondern auch schlechtere werden entfernt) unter Umständen bei manchen Problemen vermeintlich schlechte (Zwischen-)Lösungen aus dem Suchraum, die aber zur Kombination einer optimalen Lösung benötigt werden. Das Auffinden einer optimalen Lösung ist dann nicht mehr möglich. Dies tritt zum Beispiel im Datensatz gcut3 auf. Ein Ausweg ist, die starke Bereinigung zu deaktivieren.
Eine andere Möglichkeit, den Suchraum einzuschränken, wäre die Bildung von Äquivalenzklassen von Lösungen. Dabei werden äquivalente Lösungen, also Lösungen mit gleicher Länge, Breite, Fläche und Anzahlen der verwendeten Rechtecke, zu Klassen zusammengefasst. Der Algorithmus rechnet dann nur mit einem Repräsentanten der Klasse, der Suchraum ist damit kleiner. Am Ende kann eine Lösung beliebig umgeordnet werden, indem Teillösungen durch äquivalente Lösungen ersetzt werden, die gewisse Anforderungen wie beispielsweise Symmetrie erfüllen. Dies kann durch eine veränderte Datenstruktur erreicht werden: Man definiert einen neuen abstrakten Datentyp für Lösungsklassen mit den Attributen Länge, Breite, Fläche, Anzahlen der verwendeten Recht-

ecke und einer Menge von Lösungen. Eine Instanz dieses Datentyps besteht dann entsprechend aus einer Länge, einer Breite, einer Fläche, einem Array der Anzahlen der verwendeten Rechtecke und einer Menge von Lösungen, die genau diese Attribute hat.

Literaturverzeichnis

[1] G. Scheithauer. *Zuschnitt- und Packungsoptimierung: Problemstellungen, Modellierungstechniken, Lösungsmethoden*. Vieweg+Teubner, 2008.

[2] P. Y. Wang. Two algorithms for constrained two-dimensional cutting stock problems. *Operations Research*, 31:573–586, 1983.