Effizienter Algorithmus für das beschränkte zweidimensionale Zuschneideproblem

Efficient algorithm for the constrained two-dimensional cutting stock problem
Bachelor-Thesis von Benedikt Kehr
November 2010

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Mathematik
Fachgebiet Optimierung

Effizienter Algorithmus für das beschränkte zweidimensionale Zuschneideproblem
Efficient algorithm for the constrained two-dimensional cutting stock problem

Vorgelegte Bachelor-Thesis von Benedikt Kehr

1. Gutachten: Prof. Dr. Marco E. Lübbecke
2. Gutachten: Prof. Dr. Stefan Ulbrich

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 1. November 2010

(Benedikt Kehr)

Inhaltsverzeichnis

1 Einführung

In dieser Bachelor-Thesis soll auf das beschränkte zweidimensionale Zuschneideproblem eingegangen werden. Dabei werden verschiedene Algorithmen und deren funktionsweise betrachtet. Insbesondere wird eine tiefergehende Betrachtung des Cutting Stock Algorithm von André R.S. Amaral und Mike Wright (1999) [1] durchgeführt. Zum näheren Verständnis des beschränkten zweidimensionalen Zuschneideproblems betrachten wir zunächst, das entsprechende eindimensionale Problem und einige Anwendungsbeispiele.

Problem 1. Das eindimensionale Zuschneideproblem

Das eindimensionale Zuschneideproblem ist ein Problem aus der Wirtschaft, dass mit Hilfe der ganzzahligen linearen Optimierung modelliert werden kann.

Aus endlich vielen Stücken Material vorgegebener Länge L, im Folgenden $Container$ genannt, sollen kleinere Stücke, sogenannte Güter, der Länge $l_i \in \mathbb{R}^+$, mit $i \in \{1, ..., n\}$ geschnitten werden. Dabei muss ein Gut mit Länge l_i, nach dem Zuschneiden, mindestens $b_i \in \mathbb{N}$ mal vorhanden sein.

Ein Vektor $c \in \mathbb{N}^n$, wobei $c_i \in c$, $i \in \{1, ..., n\}$ die Anzahl des Guts der länge l_i beschreibt, heißt $Muster$. Ein $zulässiges\ Muster$ passt in die vorgegebene Dimension L des Containers, das heißt $\sum_{i=1}^{n} c_i \cdot l_i \leq L$. Die Differenz zwischen der Länge L und der Länge der Güter aus einem Muster heißt Restabfall.
Sei C eine Menge mit zulässigen Mustern. Dann minimiere die Anzahl der verwendeten Muster (und somit der benutzten Materialstücke):

$$\min_{s.t.} |C|$$

$$\sum_{c \in C} c_i \geq b_i, \text{ für alle } i \in \{1, ..., n\}$$

Alternativ kann man das Problem so beschreiben, dass der Restabfall minimal werden soll:

$$\min_{s.t} \sum_{c \in C} (L - \sum_{i=1}^{n} c_i \cdot l_i)]$$

$$\sum_{c \in C} c_i \geq b_i, \text{ für alle } i \in \{1, ..., n\}$$

Das eindimensionale Zuschneideproblem ist $\mathcal{NP}$-schwer[5]. Das heißt es ist durch einen nichtdeterministischen Algorithmus in polynomieller Zeit lösbar. Ein deterministischer Algorithmus der das Problem in polynomieller Zeit löst ist bis jetzt noch nicht bekannt.

Anwendungsbeispiele. *Das eindimensionale Zuschneideproblem findet Anwendung in vielen Industrien:*

- *Holzindustrie*
- *Metallindustrie*
- *Glassindustrie*
- *Textilindustrie*
- *Papierindustrie*
- *Zuschneide- und Packungssoftware*

Dabei ist zu beachten, dass das Zuschneideproblem auch als Packungsproblem interpretiert werden kann. Dazu betrachtet man als Container anstatt Materialstücke bestimmter Länge leere Bodenflächen entsprechender Länge, die durch Einheiten von Waren, den Gütern, mit vorgegebenen Längen aufgefüllt werden müssen. Wobei eine Mindestanzahl von Waren jedes Typs verstaut werden muss.

Problem 2. Das zweidimensionale Packungsproblem

Packungsprobleme können auch zweidimensional betrachtet werden. Das heißt das Problem besteht darin zweidimensionale Güter in Container zu packen, sodass möglichst viele Güter hinein passen, bzw. möglichst wenig Fläche des Containers verschwendet wird.

Das Modell lautet wie folgt:
Sei $R = \{r_1, ..., r_i, ..., r_n\}$ eine endliche Menge von zulässigen Rechtecken, definiert durch die Tupel $r_i = (l_i, b_i)$, für $i \in \{1, ..., n\}$, hierbei heißt l_i Länge und b_i Breite von r_i. Sei weiterhin ein Container $C = (L, H)$ mit Länge L und Breite B gegeben. Nun gilt es eine Anordnung $Z = (L_Z, H_Z, a)$ zu finden, wobei $a = (a_1, ..., a_n)$ mit $a_i \in \mathbb{N}$, sodass gilt:

- $\sum a_i \cdot (l_i \cdot b_i)$ ist minimal.
- $L_Z \leq L$
- $H_Z \leq H$
- Die durch r_i beschriebenen Rechtecke dürfen sich innerhalb der Anordnung nicht überschneiden.

Problem 3. Das zweidimensionale Zuschneideproblem

Im Folgenden erweitern wir das zweidimensionale Packproblem zu dem hier eigentlich betrachteten zweidimensionalen Zuschneideproblem. Beginnen wir zunächst mit einer Defintion:

Definition 1.1. Ein *Guilloutinenschnitt* ist ein Schnitt der durch ein Rechteck parallel zu einer seiner Außenkanten verläuft und das Rechteck in zwei Teile teilt.

Beispiele für Guilloutinenschnitte:

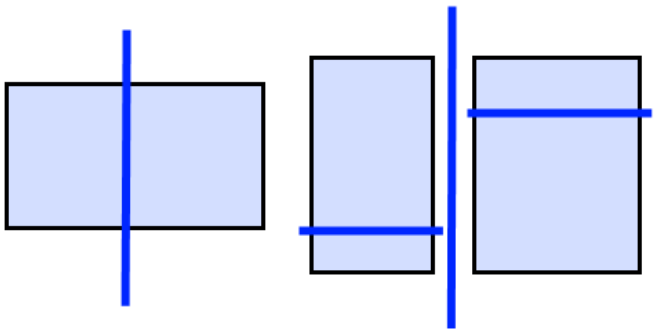


Abbildung 1.1: Guilloutinenschnitt

Wie im eindimensionalen Fall kann man das zweidimensionale Packungsproblem auch als Zuschneideproblem interpretieren. Dabei wird ein rechteckiger Container mit vorgegebenen Dimensionen in zulässige Rechtecke (Güter) zugeschnitten. Die Anzahl der Güter ist beschränkt. Um das Problem zusätzlich an die Praxis anzunähern führen wir eine zusätzliche Nebenbedingung ein:

Jeder Schnitt der beim Zuschneiden des Containers durchgeführt wird, muss ein Guilloutinenschnitt sein.

Dabei entstehen im Allgemeinen Reststücke, aus denen kein Gut mehr geschnitten werden kann. Es gilt die Fläche dieser Reststücke und damit den anfallenden Abfall zu minimieren.
Der Guilloutinen-Schnitt ist der wesentliche Unterschied des Problems zu einem zweidimensionalen Packproblem.
Im Folgenden werden mehrere mögliche Algorithmen zur Lösung des Problems erläutert. Insbesondere wird auf den, im

Verhältnis zu anderen Algorithmen, effizienten Algorithmus „CSA“ (Cutting Stock Algorithm) von André S. Amaral und Mike Wright aus dem Jahr 1999 eingegangen. Dieser Algorithmus wird effizient, durch geschicktes Bewerten und frühes Verwerfen von schlechten Lösungen.

2 Das beschränkte zweidimensionale Zuschneideproblem

Seien die Dimensionen L (Länge) und H (Höhe) eines rechteckigen Containers gegeben. Sei $R := \{r_1, r_2, ..., r_i, ..., r_n\}$ die Menge von rechteckigen Gütern, wobei r_i die Länge $l_i, i \in \{1, ..., n\}$ und die Höhe $h_i, i \in \{1, ..., n\}$ hat. Die Anzahl des Guts r_i die aus dem Container geschnitten werden sei x_i (für $i \in \{1, ..., n\}$). Das Ziel ist es nun den Restmüll, d.h. die Fläche die nicht für ein Gut benutzt wird, zu minimieren:

$$\min L \cdot H - \sum_{i=1}^{n} x_i \cdot l_i \cdot h_i$$

Das Problem wird durch folgende Nebenbedingungen eingeschränkt:

(1) Die Anzahl der Güter muss ganzzahlig und positiv sein.

$$x_i \in \mathbb{N}, (\forall i \in \{1, 2, ..., n\})$$

(2) Die Anzahl der Recktecke eines Typs $r_i \in R$ (für alle $i \in \{1, ..., n\}$) sind durch b_i beschränkt.

$$x_i \leq b_i, (\forall i \in \{1, 2, ..., n\})$$

(3) Weiterhin müssen die Rechtecke so innerhalb des Blechs anordenbar sein, dass alle Schnitte durch Guillotinenschnitte durchgeführt werden können.

Zur Lösung des *beschränkten zweidimensionalen Zuschneideproblems* gab es, laut Amaral und Wright 1999, im wesentlichen 2 Ansätze. Ein Top-Down-Ansatz wurde 1977 durch Christofides und Whitlock, ein Bottom-Up-Ansatz 1983 durch Wang vorgeschlagen. Letzterer wurde als Grundlage für den Cutting Stock Algorithmus verwendet.

3 Der Algorithmus von Wang

Als Grundlage für diesen Abschnitt wurden die Ausführungen von Francesco Kriegel und Matthias Lange[2] von der TU Dresden benutzt.

Im Folgenden werden die Definitionen der Einführung, auf das zweidimensionale Zuschneideproblem und den Algorithmus von Wang angepasst:

Definition 3.1. Ein *Container* ist ein Tupel $C = (L, H)$. Dabei ist L die Länge und H die Höhe des Containers.

Anschaulich beschreibt der Container das ursprüngliche Rechteck, dass durch Guilloutinen-Schnitte zerteilt werden soll.

Definition 3.2. Ein *Gut* ist ein Tripel $g = (l, h, b)$. Dabei sind l und h die Länge und Höhe und b die Beschränkung der Anzahl dieses Guts.

Die Menge $\mathscr{G} = \{g_1, ..., g_n\}$, wobei g_i (für alle $i \in \{1, ..., n\}$) ein Gut ist, heißt *Gtermenge*.

Anschaulich beschreibt ein Gut ein Rechteck, dass als Ergebnis des Zerteilens des Containers angestrebt wird. Die Beschränkung gibt an wie oft ein Gut in einer zulässigen Lösung benutzt werden kann.

Definition 3.3. Ein *Muster* ist eine geometrische Anordnung von Elementen aus einer Menge $\mathscr{G} = \{g_1, ..., g_n\}$, wobei g_i (für alle $i \in \{1, ..., n\}$) ein Gut ist. Das Element c_i des zugehörigen *Mustervektors* $c = (c_1, ..., c_n) \in \mathbb{N}^n$ beschreibt die Anzahl des im Muster enthaltenen Guts g_i.

Definition 3.4. Ein Zuschneideproblem ist ein Tupel $\mathscr{P} = (C, \mathscr{G})$, wobei $C = (L, H)$ ein Container und $\mathscr{G} = \{g_1, ..., g_n\}$ eine Gütermenge ist.

Definition 3.5. Sei M ein Muster mit Mustervektor c, sodass die Länge und Höhe der Anordnung durch l und h definiert sind. Dann heißt das Tupel $Z = (l, h, c)$ *Zusammenstellung*.

Eine Zusammenstellung genügt der Guilloutinen-Bedingung, falls sich das Muster durch Guilloutinenschnitte erzeugen lässt.

Zwei Zusammenstellungen können gleich sein, obwohl sie unterschiedliche Muster enthalten. Im Folgenden ist, zur Vereinfachung, mit einer Zusammenstellung immer eine Zusammenstellung, die der Guilloutinen-Bedingung genügt, gemeint.

Definition 3.6. Eine *triviale Zusammenstellung* ist eine Zusammenstellung $Z = (l, h, c)$ mit $|c| = 1$.

Definition 3.7. Bei gegebenen Zuschneideproblem $\mathscr{P} = (C, \mathscr{G})$, mit $C = (L, H)$ und $\mathscr{G} = \{g_1, ..., g_n\}$, heißt eine Zusammenstellung $Z = (l, h, c)$ *zulässig*, falls

- $l \leq L$ und $h \leq H$ gilt.
- $c \leq m$, d.h. für alle $i \in \{1, ..., n\}$ gilt $c_i \leq m_i$

Die Menge der zulässigen Zusammenstellungen in $\mathscr{P}$ wird mit $Z_{\mathscr{P}}$ bezeichnet.

Definition 3.8. Das *horizontale Zusammenbauen* zweier Zusammenstellungen $X_1 = (l_1, h_1, c_1)$ und $X_2 = (l_2, h_2, c_2)$ ist eine Zusammenstellung $Z = (l, h, c)$ mit $l = l_1 + l_2$, $h = max\{h_1, h_2\}$ und $c = c_1 + c_2$ und wird mit $X_1 H X_2$ bezeichnet.

Abbildung 3.1: horizontaler Zusammenbau

Definition 3.9. Das *vertikale Zusammenbauen* zweier Zusammenstellungen $X_1 = (l_1, h_1, c_1)$ und $X_2 = (l_2, h_2, c_2)$ ist eine Zusammenstellung $Z = (l, h, c)$ mit $l = max\{l_1 + l_2\}$, $h = h_1 + h_2$ und $c = c_1 + c_2$ und wird mit $X_1 V X_2$ bezeichnet.

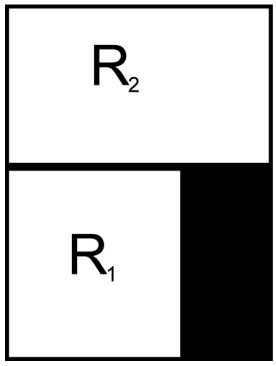


Abbildung 3.2: vertikaler Zusammenbau

Definition 3.10. Sei $Z = (l, h, c)$ eine Zusammenstellung, dann bezeichnet $Z' = (l', h', c')$, mit $l' = h$, $h' = l$ und $c' = c$ die *Rotation der Zusammenstellung Z*.

Wang's Algorithmus basiert auf dem folgenden Lemma:

Lemma 3.11. *Sei A eine Zusammenstellung von Rechtecken, sodass diese der Guilloutinen-Schnittbeschränkung genügt. Dann kann A durch horizontales und vertikales Zusammenbauen von Rechtecken gebildet werden.*

Beweis. Der Beweis ist trivial, da ein Guilloutinen-Schnitt die Umkehroperation des Zusammenbauens ist. □

Definition 3.12. Eine Zusammenstellung, die durch eine Reihe von horizontalem und vertikalem Zusammenbauen von Rechtecken gebildet werden kann heißt *Guilloutinen-Rechteck*.

Definition 3.13. Sei $G = \{g_1 = (l_{g1}, h_{g1}), ..., g_n = (l_{gn}, h_{gn})\}$ Gütermenge und $Z = (l, h, c)$ eine Zusammenstellung, dann heißt

$in = l \cdot h - \sum_{i_1}^{n} (c_i \cdot (l_{gi} \cdot h_{gi}))$

innerer Abfall.

Definition 3.14. Sei $Z = (l, h, c)$ eine Zusammenstellung und $C = (L, H)$ ein Container. Dann ist der äußere Abfall ae von Z in C beschrieben durch:

$ae = (L \cdot H) - (l \cdot h)$

Definition 3.15. Die Summe von innerem und äußerem Abfall heißt *kompletter Abfall*.

Der Algorithmus von Wang löst das folgende Problem:

Sei $\mathscr{P} = (C, \mathscr{G})$, mit $C = (L, H)$ und $\mathscr{G} = \{g_1, ..., g_n\}$, ein Zuschneideproblem.
Finde eine Zusammenstellung $Z \in Z_{\mathscr{P}}$, sodass der komplette Abfall minimal ist.

Data : Container C, Gütermenge G
Result : Zusammenstellung Z mit minimalem kompletten Abfall

Wähle $\beta \in [0, 1]$;
Initialisiere $\Omega_0 = \emptyset$;
Initialisiere $k := 0$;

while $\Omega_k = \emptyset$ **do**
  Erzeuge die trivialen Zusammenstellungen aus der Gütermenge G. (Dies sind alle Zusammenstellungen, die genau ein Gut enthalten). Prüfe die Zusammenstellungen auf Zulässigkeit und füge die zulässigen Ω_0 hinzu.
  if *Zulässig* **then**
    Erzeuge aus den Elementen von Ω_0 paarweise die horizontale sowie die vertikale Zusammenstellung.
  end
  Füge alle erzeugten Zusammenstellungen $Z = (l, h, c)$ mit $l \cdot h - \sum_{i=1}^{n} l_i \cdot h_i \cdot c_i \leq \beta \cdot L \cdot H$ zu Ω_{k+1} hinzu.;
  if $\Omega_{k+1} = \emptyset$ **then**
    Setze $k := k + 1$
  else
    Sonst wähle die Zusammestellung aus $\bigcup_{i=1}^{k} \Omega_i$ mit minimalem kompletten Abfall als Lösung aus.
  end
end

Algorithmus 1: Algorithmus von Wang

Dieser Algorithmus liefert allerdings nur eine optimale Lösung unter den Lösungen für die gilt:

$$l \cdot h - \sum_{i=1}^{n} l_i \cdot h_i \cdot c_i \leq \beta \cdot L \cdot H$$

Diese ist nur dann optimal wenn β groß genug ist. (Für $\beta = 0$ liefert er zum Beispiel gar keine Lösung, obwohl zulässige Zusammenstellungen existieren können). Je größer β ist, desto mehr Berechnungen führt der Algorithmus allerdings durch. Sei $\beta = \beta^*$ der Wert bei dem der Algorithmus eine optimale Lösung liefert, aber möglichst wenig Berechnungen durchführt.
Um diesen Wert möglichst gut zu erreichen können wir $\beta = 0,01$ wählen und nach jedem Durchlauf um $0,01$ erhöhen bis wir die optimale Lösung ausgegeben bekommen.

Ändern wir den Algorithmus also wie folgt ab:

Data : Container C, Gütermenge G
Result : Zusammenstellung Z mit minimalem kompletten Abfall

Wähle $\beta = 0,01$;
Initialisiere $\Omega_0 = \emptyset$;
Initialisiere $k := 0$;

while *Noch keine optimale Lösung gefunden* **do**
  while $\Omega_k = \emptyset$ **do**
    Erzeuge die trivialen Zusammenstellungen aus der Gütermenge G. (Dies sind alle Zusammenstellungen, die genau ein Gut enthalten). Prüfe die Zusammenstellungen auf Zulässigkeit und füge die zulässigen Zusammenstellungen Ω_0 hinzu.

    Erzeuge aus den Elementen von Ω_0 paarweise die horizontale sowie die vertikale Zusammenstellung und prüfe diese auf Zulässigkeit.;

    Füge alle zulässigen erzeugten Zusammenstellungen $Z = (l, h, c)$ mit $l \cdot h - \sum_{i=1}^{n} l_i \cdot h_i \cdot c_i \leq \beta \cdot L \cdot H$ zu Ω_{k+1} hinzu.;

    if $\Omega_{k+1} = \emptyset$ **then**
      Setze $k := k + 1$
    else
      Sonst wähle die Zusammenstellung aus $\bigcup_{i=1}^{k} \Omega_i$ mit minimalem kompletten Abfall a aus.
    end
    ;
  end
  if $a \leq \beta \cdot L \cdot H$ **then**
    Die Lösung ist optimal, der Algorithmus endet.
  end
  Wähle $\beta := \beta + 0,01$ und setze $\Omega_i = \emptyset$ für alle $i \in \{1, ..., k\}$
end

Algorithmus 2: erweiterter Algorithmus von Wang

Dieser Algorithmus liefert eine optimale Lösung, falls ein zulässiges Gut existiert, denn:

Algorithmus 1 liefert eine optimale Lösung unter den Lösungen für die gilt:

$$l \cdot h - \sum_{i=1}^{n} l_i \cdot h_i \cdot c_i \leq \beta \cdot L \cdot H$$

, weiterhin
Annahme: Der Algorithmus liefert keine optimale Lösung.

Sei β^+ der Wert bei dem der Algorithmus eine Lösung Z^+ ausgeworfen hat. Der Algorithmus hat dann die Lösung mit

$$l^+ \cdot h^+ - \sum_{i=1}^{n} l_i^+ \cdot h_i^+ \cdot c_i^+ \leq \beta \cdot L \cdot H$$

(innerer Abfall) berechnet. Für diese Lösung gilt zusätzlich

$$L \cdot H - \sum_{i=1}^{n} l_i^+ \cdot h_i v \cdot c_i^+ \leq \beta \cdot L \cdot H$$

(kompletter Abfall)

Wenn dies nicht die optimale Lösung des Problems ist, muss für die optimale Lösung Z^* gelten:

$$l \cdot h - \sum_{i=1}^{n} l_i \cdot h_i^* \cdot c_i^* > \beta \cdot L \cdot H$$

Dann gilt jedoch auch

$$l \cdot h - \sum_{i=1}^{n} l_i \cdot h_i * \cdot c_i * > L \cdot H - \sum_{i=1}^{n} l_i^+ \cdot h_i^+ \cdot c_i$$

Also übersteigt bereits der innere Abfall der optimalen Lösung Z^* den kompletten Abfall der Lösung Z^+, das ist ein Widerspruch zur Optimalität dieser Lösung, also muss bereits die Lösung Z^+ optimal sein.

Auch wenn dieser Algorithmus ein kleines β^+ findet, für das eine optimale Lösung berechnet wird, kann man davon ausgehen, dass dieses nicht dem kleinsten Wert β^* entspricht, für den dies zutrifft. Allerdings liegen die beiden Werte maximal um $\epsilon \in [0; 0,01)$ auseinander.

4 Der CSA von Amaral und Wright

Amaral und Wright haben in ihrer Arbeit aus dem Jahr 1999/2000[1] ebenfalls einen Bottom-up-Algorithmus mit dem Namen 'Cutting Stock Algorithm' (CSA) entwickelt.

Bei der Entwicklung ihres Algorithmus haben Amaral und Wright eine scheinbar offensichtliche Beobachtung gemacht. Diese ist im folgenden Lemma festgehalten:

Lemma 4.1. *Seien $X = (l, h, c)$, $X_1 = (l_1, h_1, c_1)$ und $X_2 = (l_2, h_2, c_2)$ Zusammenstellungen, dann gilt:*

a) $X'' = X$

b) $(X_1 V X_2)' = X_1' H X_2'$

c) $X_1 V X_2 = (X_1' H X_2')'$

Beweis. a) Es folgt direkt aus der Definition der Rotation $X'' = (l'', h'', c'')$ mit $l'' = h' = l$, $h'' = l' = h$ und $c'' = c' = c$. Also gilt

$$X'' = (l'', h'', c'') = (l, h, c) = X$$

b) Es gilt: $(X_1 V X_2)' = (max\{l_1, l_2\}, h_1 + h_2, c_1 + c_2)' = (h_1 + h_2, max\{l_1, l_2\}, c_1 + c_2) = X_1' H X_2'$

c) Folgt direkt aus a) und b).

□

Damit gilt:

$$\begin{aligned} X_1 V X_2 &= (X_1' H X_2')' \\ X_1' V X_2 &= (X_1 H X_2')' \\ X_1 V X_2' &= (X_1' H X_2)' \\ X_1' V X_2' &= (X_1 H X_2)' \end{aligned}$$

Es gibt insgesamt acht verschiedene Möglichkeiten zwei Zusammenstellungen horizontal bzw. vertikal zusammenzubauen und um 90° zu drehen. Diese können wir mit Hilfe von *Lemma* 4.1 nur durch H und $'$ darstellen

$$\begin{array}{ll} X_1 H X_2 & (X_1 H X_2)' \\ X_1' H X_2 & (X_1' H X_2)' \\ X_1 H X_2' & (X_1 H X_2')' \\ X_1' H X_2' & (X_1' H X_2')' \end{array}$$

Der CSA basiert auf der Branch-and-Bound-Methode. Daher gilt es zunächst die Branching-Strategie zu erklären.

4.1 Branching-Strategie

Sei R die Menge aller zulässigen Zusammenstellungen zu einem beliebigen Zeitpunkt innerhalb des Algorithmus und $X_1 = (l_1, h_1, c_1) \in R$. Teile alle Zusammenstellungen $Z = (l, h, c) \in R$ mit $h \geq h_1$ in die Mengen:

$$B_0(X_1), B_1(X_1), ..., B_k(X_1)$$

auf, sodass für $X = (l_X, h_X, c_X)$ und $Y = (l_Y, h_Y, c_Y)$ gilt:

Aus $X \in B_0(X_1)$ folgt $h_X = h_1$.
$B_i(X_1)$ ist nicht leer. (für alle $i \in \{0, ..., k\}$).
Aus $X, Y \in B_i(X_1)$ folgt $h_X = h_Y$ (für alle $i \in \{0, ..., k\}$).
Aus $X \in B_{i+1}(X_1)$ und $Y \in B_i(X_1)$ folgt $h_X > h_Y$ (für alle $i \in \{0, ..., k-1\}$).
Innerhalb von $B_i(X_1)$ sind die Rechtecke nach aufsteigender Länge sortiert. (für alle $i \in \{0, ..., k\}$)

Bemerkung.

- *Es reicht X_1 nur mit größeren Rechtecken zu kombinieren, da die Kombination mit einem kleineren Rechteck X_2 auftritt wenn die Branching-Strategie auf X_2 angewandt wird.*
- *Es reicht aus horizontale Zusammenstellungen und Rotationen zu betrachten.*
- *Durch das gezielte Sortieren der Zusammenstellungen in der Branching-Strategie können später im CSA Kombinationen die nicht zu einer optimalen Lösung führen früh verworfen werden.*

Nehmen wir ohne Beschränkung der Allgemeinheit an, dass $L \geq H$ gilt. Bauen wir nun das Rechteck X_1 mit Länge und Höhe (l_1, h_1) horizontal mit $X_2 \in B_i(X_1)$ (für alle $i \in \{0, ..., j\}$) mit Länge und Höhe (l_2, h_2) zusammen. Da h_2 größer ist als h_1 ergibt dies ein Rechteck mit Länge und Höhe $(l_1 + l_2, h_2)$. Durch Rotation ergibt sich als Bedingung für die Zulässigkeit des Rechtecks:

$$max\{l_1 + l_2, h_2\} \leq L \text{ und } min\{l_1 + l_2, h_2\} \leq H$$

Weiterhin gilt für die Zulässigkeit:

$$\text{Der innere Abfall ist nicht größer als } \beta \cdot H \cdot L$$

Damit können wir nun den Algorithmus aufstellen:

Algorithmus 4.2. Der CSA

Data : Container C, Gütermenge G
Result : Zusammenstellung Z mit minimalem kompletten Abfall

Wähle $\beta = 0.01$;
while *Optimale Lösung noch nicht gefunden* **do**
  Initialisiere die Menge R als die Menge aller trivialen zulässigen Zusammenstellungen, sortiert nach nicht-absteigender Höhe. Zusammenstellungen gleicher Höhe sind nach nicht-absteigender Länge sortiert.;
  while *Die Schleife wurde nicht durch break; abgebrochen* **do**
    Wähle die erste Zusammenstellung $X_1 \in R$, die noch nicht betrachtet wurde mit Länge und Höhe (l_1, h_1).;
    while *Es existiert eine Zusammenstellung in R, die noch nicht betrachtet wurde* **do**
      Konstruiere die Mengen $B_1(X_1), ..., B_k(X_1)$ nach der oben beschriebenen Branching-Strategie. ;
      Initialisiere $S = \emptyset$ und $z = 0$.;
      while $z<k$ **do**
        while *Es existiert eine Zusammenstellung in* $B_z(X_1)$*, die noch nicht betrachtet wurde;* **do**
          Baue X_1 horizontal mit der ersten (bzw. nächsten) Zusammenstellung $X_2 \in B_z(X_1)$ zu Z zusammen. ;
          if *Der zusätzliche innere Abfall* $(h_2 - h_1) \cdot l_1$ *übersteigt den erlaubten Abfall* $\beta \cdot H \cdot L$*;* **then**
            break;
          end
          if $max\{l_1 + l_2, h_2\} > L$ *oder* $min\{l_1 + l_2, h_2\} > H$ *und* R *enthält nicht* Z*;* **then**
            Füge Z der Menge S hinzu.;
          end
          z++;
        end
      end
      if S *ist nicht leer* **then**
        Setze $R = R \cup S$;
      else
        break;
      end
    end
  end
  Bestimme die Zusammenstellung aus R, die den geringsten kompletten Abfall a hat. ;
  if $a \leq \beta \cdot H \cdot L$ *ist die Zusammenstellung optimal und der Algorithmus endet.* **then**
    Setze $\beta = \beta + 0.01$;
  end
end

Bemerkung.

6.2 *Gilt der zusätzliche innere Abfall* $(h_2 - h_1) \cdot l_1$ *übersteigt den erlaubten Abfall* $\beta \cdot H \cdot L$ *ist diese Zusammenstellung des Problems nicht zulässig. Durch die Konstruktion der betrachteten Mengen gilt dies für alle Rechtecke* $X_2 \in B_y(X_1)$ *für alle* $y \geq z$.

6.3 *Gilt* $max\{l_1 + l_2, h_2\} > L$ *oder* $min\{l_1 + l_2, h_2\} > H$ *ist die Zusammenstellung nicht zulässig. Durch die Sortierung innerhalb der betrachteten Menge gilt dies für alle Rechtecke* $X_2 in B_z(X_1)$.

Beweis. zu 6.2 Sei $X_1 \in R$, $X_2 \in B_z(X_1)$ mit den Längen und Höhen (l_1, h_1) bzw. (l_2, h_2). Bezeichne a den inneren Abfall von X_1 und gelte:

$$(h_2 - h_1) \cdot l_1 + a > \beta \cdot H \cdot L$$

Dann gilt für jedes $X_3 \in B_y(X_1)$ mit $y \geq z$ mit Länge und Höhe (l_3, h_3):

$$h_3 \geq h_1$$

Damit folgt

$$(h_2 - h_1) \cdot l_1 \leq (h_3 - h_1) \cdot l_1$$

und weiter

$$\beta \cdot H \cdot L < (h_2 - h_1) \cdot l_1 + a \leq (h_3 - h_1) \cdot l_1 + a$$

Also ist das Rechteck, das aus dem horizontalen Zusammenbauen von X_1 und X_3 entsteht ebenfalls nicht zulässig.

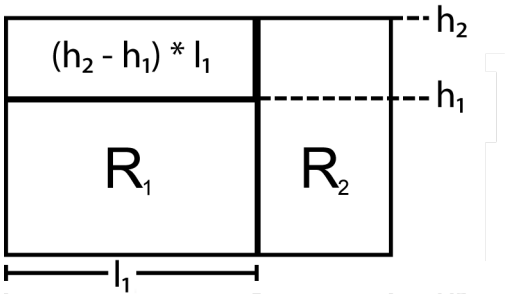


Abbildung 4.1: Wird R_2 durch ein Rechteck mit Höhe $h_3 > h_2$ erstetzt wird der innere Abfall größer.

zu 6.3 Sei $X_1 \in R$, $X_2 \in B_z(X_1)$ mit den Längen und Höhen (l_1, h_1) bzw. (l_2, h_2). Es gelte

$$max\{l_1 + l_2, h_2\} > L \text{ oder } min\{l_1 + l_2, h_2\} > H$$

Für alle Rechtecke $X_3 \in B_z(X_1)$ mit Länge und Höhe (l_3, h_3), wobei X_3 in $B_z(X_1)$ hinter X_2 sortiert ist (also $l_3 > l_2$ gilt) gilt:

$$max\{l_1 + l_3, h_2\} \geq max\{l_1 + l_2, h_2\} > L$$

und

$$min\{l_1 + l_3, h_2\} \geq min\{l_1 + l_2, h_2\} > H$$

Also ist das Rechteck, das aus dem horizontalen Zusammenbauen von X_1 und X_3 entsteht ebenfalls nicht zulässig. □

5 Vergleich zwischen dem Algorithmus von Wang und dem CSA

Der Algorithmus der in diesem Vergleich genutzt wurde ist nicht der klassische Algorithmus von Wang wie hier vorgestellt. Es wird eine Implementierung, die im Jahr 1989 von Vasko vorgeschlagen wurde genutzt. Diese läuft im Durchschnitt mehr als 25 mal so schnell wie der klassische Algorithmus von Wang (für die von Vasko betrachteten Daten, bei nicht angegebener Rechenmaschine). Des Weiteren wird bei Fall 3 eine Methode von Vasko (nach dem Vorbild von Christofides und Whitlock) verwendet um Probleminstanzen zu generieren.[1]

5.1 Generieren von Probleminstanzen nach Vasko

Um den Algorithmus quantitativ bewerten zu können werden möglichst viele Probleminstanzen benötigt. Diese Instanzen sollten nach Möglichkeit jedoch sinnvoll gewählt werden. So macht es beispielsweise keinen Sinn einen Container zu wählen, dessen Ausmaße so klein sind, dass kein zulässiges Gut existiert. Genauso führt ein im Verhältnis zu den Gütern sehr großer Container eventuell zu keinen benutzbaren Ergebnissen, da die Berechnung vieler solcher Probleminstanzen zu lange dauert.

Um möglichst viele Instanzen betrachten zu können und zu repräsentativen Ergebnissen zu gelangen wurde von Vasko folgende Methode vorgeschlagen um die Parameter einer Probleminstanz zu generieren:

Erzeuge für alle Kombinationen der Parameter $n \in \{5, 10, 15, 20\}, H \in \{150, 200, 300\}, Qmax \in \{5, 10\}$ fünf Probleminstanzen $\mathscr{P} = (C, \mathscr{G})$ mit

- $C = (100, H)$
- $\mathscr{G} = \{g_1, ...g_n\}$, wobei $g_i = (l_i, h_i, b_i)$ für alle $i \in \{1, ..., n\}$.
- $|\mathscr{G}| = n$
- $b_i \leq Qmax$ für alle $i \in \{1, ..., n\}$
- Generiere $k \in (0.05, 0.25)$ zufällig und setze $l_i \cdot h_i = k \cdot H \cdot L$.
- Generiere $ratio \in (1, 5)$ zufällig und setze $\frac{l_i}{h_i} = ratio$

Daraus resultieren $|n| \cdot |H| \cdot |Qmax| = 24$ verschiedene Parameterkombinationen. Bei fünf Probleminstanzen pro Parameterkombination gibt es also ingesamt 120 generierte Probleminstanzen.

5.2 Rechenzeiten

Vergleichen wir nun die Rechenzeiten der beiden Algorithmen. Im Folgenden betrachten wir drei Fälle.
Fall 1:

- β wird direkt als optimaler Wert gewählt.
- 10 betrachtete Probleme bestehend aus jeweils 30 Rechtecken.
- Ausgeführt auf einem HP9000/735.
- Durch Schwierigkeiten bei der Messung der Zeit auf dem benutzten Betriebssystem gibt es Messfehler im Bereich von $\pm 20\%$.

Die Ergebnisse sind in folgender Tabelle dargestellt:
Selbst wenn die 20% Fehlertoleranz einbezogen werden, hat der CSA jedes mal schneller eine optimale Lösung gefunden als der Algorithmus von Wang. Es fällt auf, dass der CSA in vielen der Fälle trotzdem mehr Rechtecke generiert und überprüft. Da beide Algorithmen die optimale Lösung finden muss der komplette Abfall gleich sein.

	Zeit		Fehlerbereich				minimale Differenz	Anzahl an generierten Rechtecken		kompletter Abfall	
Probleminstanz	CSA	EWA	CSA (min)	CSA (max)	EWA (min)	EWA (max)	min(EWA) - max(CSA)	CSA	EWA	CSA	EWA
B1	0,032	0,068	0,026	0,038	0,054	0,082	0,016	20	69	1.234	1.234
B2	0,031	0,207	0,025	0,037	0,166	0,248	0,128	101	96	1.586	1.586
B3	0,048	0,428	0,038	0,058	0,342	0,514	0,285	162	158	2.310	2.310
B4	0,047	0,287	0,038	0,056	0,230	0,344	0,173	124	113	1.256	1.256
B5	0,048	0,338	0,038	0,058	0,270	0,406	0,213	116	126	1.053	1.053
B6	0,071	1,149	0,057	0,085	0,919	1,379	0,834	262	242	1.203	1.203
B7	0,104	2,514	0,083	0,125	2,011	3,017	1,886	369	293	571	571
B8	0,022	0,054	0,018	0,026	0,043	0,065	0,017	20	68	1.041	1.041
B9	0,549	32,659	0,439	0,659	26,127	39,191	25,468	1.512	1.033	410	410
B10	0,100	3,264	0,080	0,120	2,611	3,917	2,491	346	350	203	203

Abbildung 5.1: Fall 1[1]

Fall 2:

- β wird 0 gesetzt und anschließend in Schritten von $0,01$ erhöht.
- 8 betrachtete Probleme erstellt von Daza et al. (1995)
- Ausgeführt auf dem gleichen System wie in **1**, daher auch hier Messfehler im Bereich von ±20%.

Die Ergebnisse sind in folgender Tabelle dargestellt:

		Zeit			Fehlerbereich (des jeweils spätesten Wertes)		minimale Differenz	Anzahl an generierten Rechtecken			kompletter Abfall		
Probleminstanz					min	max	min(EWA) - max(CSA)						
		0,00	0,01	0,02				0,00	0,01	0,02	0,00	0,01	0,02
D1	CSA	0,154	-	-	0,123	0,185	9,456	573	-	-	0	-	-
	EWA	12,051	-	-	9,641	14,461		1.758	-	-	0	-	-
D2	CSA	0,044	0,133	0,431	0,345	0,517	29,351	212	573	1.677	520	379	29
	EWA	0,319	3,053	37,335	29,868	44,802		153	499	1.690	520	59	29
D3	CSA	0,041	0,140	0,507	0,406	0,608	18,267	110	497	1.918	556	242	43
	EWA	0,098	1,303	23,594	18,875	28,313		98	406	1.447	556	421	43
D4	CSA	0,044	0,108	0,388	0,310	0,466	18,530	148	445	1.543	280	280	31
	EWA	0,173	1,447	23,745	18,996	28,494		133	398	1.365	280	280	31
D5	CSA	0,030	-	-	0,024	0,036	0,078	101	-	-	0	-	-
	EWA	0,143	-	-	0,114	0,172		162	-	-	0	-	-
D6	CSA	0,337	-	-	0,270	0,404	116,364	1.256	-	-	0	-	-
	EWA	145,961	-	-	116,769	175,153		6.081	-	-	0	-	-
D7	CSA	0,056	0,118	-	0,094	0,142	2,284	239	658	-	288	8	-
	EWA	0,445	3,032	-	2,426	3,638		222	580	-	96	8	-
D8	CSA	0,041	0,068	0,218	0,174	0,262	11,119	131	358	1.044	638	616	34
	EWA	0,178	1,244	14,226	11,381	17,071		131	358	1.045	110	110	34

Abbildung 5.2: Fall 2[1]

Erneut sehen wir, dass der CSA oft genauso viele bzw. mehr Rechtecke generiert als der EWA. Dies lässt darauf schließen, dass nicht die Strategie wann der Algorithmus ein Rechteck verwirft, sondern vielmehr die Branching-Strategie sehr effizient ist. Da mehr Rechtecke schneller erstellt werden können. Insbesondere wenn β oft erhöht werden muss, schneidet der CSA in Bezug auf die Anzahl der generierten Rechtecke schlechter ab.

Fall 3:

- Für β wird direkt ein sehr hoher Wert, hier konkret $\beta = 0,25$ gewählt.
- 120 zufällig generierte Probleme wurden betrachtet (siehe **Generieren von Probleminstanzen nach Vasko**).
- Ausgeführt auf einem Pentium II 350MHz Mikrocomputer.

Bei den folgenden Ergebnissen ist zu beachten, dass diese in der Arbeit von Amaral und Wright inkonsistent sind, da sich einige Durchschnittszahlen nicht nachvollziehen lassen. Diese sind in den folgenden Tabellen rot markiert:
Zunächst betrachten wir die Anzahl der generierten Rechtecke durch den CSA und EWA:

Tabelle für EW

	n =	5	10	15	20	Durchschnitt
H	b_i					EW
150	1-5	384,80	1860,40	2188,80	3481,00	1978,75
	1-10	253,00	2582,00	1535,60	1628,00	1499,65
200	1-5	666,00	917,20	996,80	1800,60	1095,15
	1-10	543,20	745,00	1703,20	2052,40	1260,95
300	1-5	409,00	675,60	2655,00	964,20	1175,95
		306,40	584,60	6062,60	1147,80	2025,35
Durchschnitt		427,07	1227,47	2523,67	1845,67	1505,97

Durchschnitt Amaral/Wright: 1565,26

Amaral/Wright: 433,07

Tabelle für CSA

	n =	5	10	15	20	Durchschnitt
H	b_i					CSA
150	1-5	532,00	2332,60	2754,80	3854,20	2368,40
	1-10	330,00	2864,60	2106,40	1415,20	1679,05
200	1-5	1138,60	2093,00	1565,60	2694,00	1872,80
	1-10	848,20	1708,60	2360,80	3395,00	2078,15
300	1-5	458,20	1367,60	4951,00	1739,80	2129,15
		451,40	1120,40	7655,80	1987,40	2803,75
Durchschnitt		626,40	1914,47	3565,73	2514,27	2155,22

Durchschnitt Amaral/Wright: 1750,05

Amaral/Wright: 636,62

Abbildung 5.3: Fall 3[1]

Erneut sehen wir, dass im Schnitt der CSA deutlich mehr Rechtecke generiert als der Algorithmus von Wang in der Implementierung von Vasko. Doch auch hier ist die Laufzeit des Cutting Stock Algorithm deutlich kürzer. In der folgenden Tabelle erkennen wir, dass der CSA im Durchschnitt über 7 Sekunden schneller ist (dieses Ergebnis erhalten wir auch, wenn die roten Zellen nicht beachtet werden):
Nach zeitlichen Kriterien ist der Cutting Stock Algorithm also deutlich effizienter im Berechnen der optimalen Lösung. Allerdings spricht die hohe Anzahl an generierten Gütern beim CSA dafür, dass dieser einen im Vergleich höheren Speicherbedarf hat.

Wie kann es sein, dass der Algorithmus zwar mehr Güter generiert, jedoch einen geringeren Zeitaufwand hat?
Betrachten wir den Cutting Stock Algorithm stellen wir fest, dass in der Branching-Strategie jedes mal alle möglichen neuen Zusammenstellungen generiert werden ohne darauf zu achten ob diese zulässig sind oder nicht. Im Gegensatz hierzu generiert der Algorithmus von Wang nur die Zusammenstellungen die tatsächlich zulässig sind. Dadurch werden bei Wang aber sowohl die zulässigen Zusammenstellungen, als auch die zu verwerfenden Zusammenstellungen auf Zulässigkeit geprüft.
Beim CSA wird durch das frühe Verwerfen von ganzen, in der Branching-Strategie erzeugten, Teilmengen nur ein geringer Teil der Güter auf Zulässigkeit geprüft. An dieser Stelle hat der CSA durch seine wenigen Vergleiche einen Zeitvorteil der sich in den obigen Tabellen darstellt.
Insbesondere durch die Tests mit zufällig generierten Daten können wir also sagen, dass der Cutting Stock Algorithm in der Anwendung effizienter ist als der Algorithmus von Wang.

5.3 Lösen von großen Problemen mit Hilfe des CSA

Anhand der Berechnungen im vorhergehenden Kapitel kann man erkennen, dass kleinere Probleme vom CSA effizient gelöst werden. Betrachten wir nun, wie der Umgang des Algorithmus mit großen Problemen ist.

Tabelle für EW						
	n =	5	10	15	20	Durchschnitt
H	b_i					EW
150	1-5	0,29	6,99	10,58	28,27	11,53
	1-10	0,14	11,49	6,05	4,49	5,54
200	1-5	1,15	1,49	2,79	6,47	2,98
	1-10	0,72	1,65	6,55	9,68	4,65
300	1-5	0,21	1,16	14,00	5,06	5,11
		0,17	0,56	53,36	4,49	14,65
Durchschnitt		0,45	3,89	15,56	9,74	7,41
Tabelle für CSA						
	n =	5	10	15	20	Durchschnitt
H	b_i					CSA
150	1-5	0,04	0,17	0,20	0,30	0,18
	1-10	0,02	0,24	0,14	0,09	0,12
200	1-5	0,10	0,22	0,13	0,24	0,17
	1-10	0,06	0,15	0,21	0,31	0,18
300	1-5	0,04	0,11	0,77	0,16	0,27
		0,03	0,10	1,17	0,18	0,37
Durchschnitt		0,05	0,17	0,44	0,21	0,22

Abbildung 5.4: Fall 3[1]

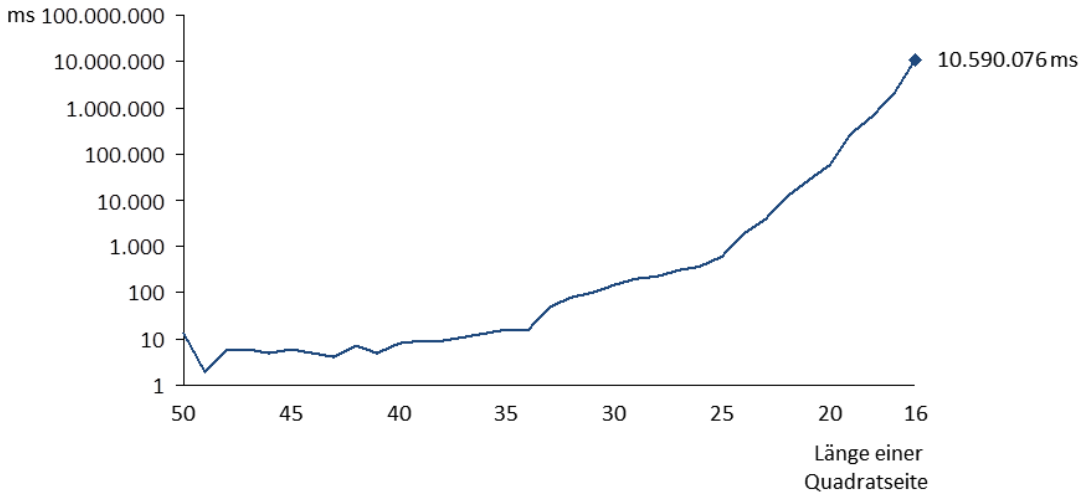
Bei den Ergebnissen in diesem Abschnitt ist jedoch zu beachten, dass diese Berechnungen mit der Implementierung des Algorithmus aus dem Anhang auf einer Rechenmaschine mit folgenden Spezifikationen durchgeführt wurde: AMD Athlon(tm) X2 Dual-Core QL-62 2.00 GHz mit 4,00GB RAM und einem Win7 64-bit OS.
Darüber hinaus haben, laut deren Veröffentlichung, Amaral und Wright noch zusätzliche Routinen zum Zeitsparen implementiert, da sie diese nicht näher beschreiben sind sie in der benutzten Implementierung nicht vorhanden.
Diese Unterschiede haben Auswirkungen auf die Dauer und den Speicherbedarf des Algorithmus, da hier aber nur eine Tendenz aufgezeigt werden soll kann dieser Einfluss vernachlässigt werden.

Betrachten wir zunächst was unter der Größe eines Problems zu verstehen ist. Da der Algorithmus sehr schnell neue Zusammenstellungen generiert macht die ursprüngliche Anzahl der Güter nur einen kleinen Teil der benötigten Ressourcen aus. Wählen wir einen Container mit sehr großen Ausmaßen (bspw. Höhe = 1000 und Länge = 1000), kann es trotzdem passieren, dass bereits ein triviales Gut die optimale Lösung direkt liefert (falls im Beispiel ein Gut mit Höhe und Länge 1000 existiert). Ein großes Problem entsteht also durch das Verhältnis der eigenen Höhe und Länge zu der Höhe und Länge des größten zulässigen Guts bzw. dessen Rotation.

Betrachten wir im Folgenden also Probleminstanzen folgender Form:

- Der Container $C = (L, H)$ ist definiert durch $L = 100$ und $H = 100$.
- Die Gütermenge $\mathcal{G}$ besteht aus einem Gut, d.h. $|\mathcal{G}| = 1$.
- Das Gut $g = (l, h, b) \in \mathcal{G}$ hat gleiche Seitenlängen $l = h$ und es gilt $b = \frac{L}{l} \cdot \frac{H}{h}$.

Das folgende Diagramm zeigt die Berechnungszeit in Millisekunden für die Probleminstanzen mit $l, h \in \{50, 49, ..., 16\}$.



Bereits wenn das betrachtete Gut Seitenlängen hat, deren Länge 15% der Länge der Seitenlängen des Containers ausmachen ist das Problem nicht in angemessener Zeit lösbar. Jedoch kann man davon ausgehen, dass der Algorithmus durchläuft und das Ergebnis danach ausgibt.

Bei Betrachtung eines noch größeren Problems - hier wurde konkret ein Container $C = (1000, 1000)$ mit einem Gut $g = (10, 10)$ gewählt - stößt der Algorithmus jedoch nicht nur zeitlich an seien Grenzen. Durch die bei so großen Problemen extrem schnell wachsende Anzahl an möglichen Zusammenstellungen wurde der Durchlauf des Algorithmus erfolglos durch einen Speicherüberlauf abgebrochen.

Der Cutting Stock Algorithm läuft also, je nach Implementierung und Rechner, bis zu einer bestimmten Grenze der Größe der betrachteten Probleminstanz sehr schnell. Ab dieser Größe verwirft der Algorithmus zwar weiterhin Zusammenstellungen sehr früh und arbeitet dadurch effizient, allerdings wächst die Größe des Problems zu stark an, damit dieser noch nutzbar ist.

6 Ergebnis

6.1 Schlussfolgerungen

Zusammenfassend können wir über den CSA sagen, dass dieser in den meisten Fällen schneller als der Algorithmus von Wang eine optimale Lösung findet. Diesen Zeitvorteil bekommt der Algorithmus durch eine geschickte Branchingstrategie. Diese sortiert die erzeugten Rechtecke geschickt an, sodass früh Lösungen verworfen werden können, die zu keinem optimalen Ergebnis führen.

Als weiteren großen Vorteil ihres Algorithmus führen Amaral und Wright an, dass dieser sehr leicht zu implementieren ist. Eine Implementierung in Java findet sich im Anhang. Dabei ist allerdings zu beachten, dass Amaral und Wright eine Methode um doppelte Rechtecke aus den Zwischenmengen zu entfernen nicht weiter diskutiert wurden und somit in der angehängten Implementierung fehlen. Es ist also davon auszugehen, dass dieses Programm eine langsamere Laufzeit hat als der CSA auf identischen Rechenmaschinen.

Die Betrachtungen der Rechenzeit liefern sehr positive Ergebnisse. So ist der CSA, bei den betrachteten Problemen zeitlich fast immer dem Algorithmus von Wang überlegen. Allerdings generiert der Algorithmus von Amaral und Wright dabei meistens mehr Rechtecke, als der Algorithmus von Wang. Des Weiteren wurden zum Testen nur Probleme mittlerer Größe betrachtet. Ein nächster Schritt ist die Überprüfung wie schnell der CSA für große Probleme eine optimale Lösung findet.

6.2 Mögliche Erweiterungen des Algorithmus

Im Folgenden werden noch einige mögliche Erweiterungen des Problems und Lösungsansätze mit Hilfe des CSA erläutert.

Der CSA betrachtet nur einen einzelnen Container aus dem Güter geschnitten werden können. Es liegt jedoch nahe, dass man mehrere Grundgrößen von Blechen oder ähnlichem zur Verfügung hat. Es ist zu Entscheiden zwischen verschiedenen Größen von Containern um möglichst günstig alle Güter auschneiden zu können.

Zur Lösung diese Problems muss der Algorithmus nur auf jeden verfügbaren Container angewendet werden und das Grundrechteck mit dem geringsten Abfall bzw. der günstigste Container gewählt werden.

Nicht jedes Gut muss für eine Lösung gleich wichtig sein. Daher kann es sein, dass eine Lösung mit mehr Abfall gegenüber einer Lösung mit weniger Abfall bevorzugt wird, falls dafür eine bestimmte Art von Gütern in der Lösung enthalten ist.

Eine Möglichkeit dieses Problem anhand des CSA zu lösen ist das Einführen einer Bewertungsrate $r \in [0, 1]$ für jedes Gut. Dabei steht eine Rate von $r = 1$ für ein besonders unwichtiges und eine Rate vn $r = 0$ für ein besonders wichtiges Gut. Bei der Berechnung des kompletten Abfalls wird dieser nun mit den Bewertungsraten der im Container enthaltenen Güter multipliziert. Dies führt dazu, dass Rechtecke mit einer niedrigen Bewertungsrate gegenüber Rechtecken mit einer hohen Bewertungsrate bevorzugt werden.

7 Das Kreispackproblem

In der Einführung dieser Arbeit wurden bereits einige eng verwandte Probleme kurz erläutert. Im Folgenden soll eine Abwandlung des vorne beschriebenen Packproblems und ein Ansatz für einen Algorithmus zu dessen Lösung vorgestellt werden.

Das Kreispackproblem ist ähnlich wie das zweidimensionale Packproblem. Allerdings werden hier Kreisscheiben betrachtet. Das Problem besteht darin die Scheiben s_i mit Mittelpunkt m_i, $s_i = \{x | ||x - m_i|| \leq r_i\}$ und $r_i = i$ für alle $i \in \{1, ..., n\}$ anzuordnen. Dabei soll der Radius r_0 der Kreisscheibe s_0 für die gilt:

$$x \in s_i \Rightarrow x \in s_0, \text{ für alle } i \in \{1, ..., n\}$$

minimal sein.

Die Scheiben dürfen sich dabei nicht überschneiden, das heißt, es gilt $s_i \cap s_j = \emptyset$ für alle $i, j \in \{1, ..., n\}, i \neq j$.

Für dieses Problem wurde bereits ein Algorithmus durch E. Schömer vorgeschlagen (2009)[3]. Dieser nutzt das sogenannte Verfahren der *simulierten Abkühlung*.

7.1 Exkurs: simulierte Abkühlung[4]

Um einen alternativen Ansatz zum Lösen von Packproblemen vorzustellen wird hier auf die simulierte Abkühlung eingegangen. Es soll jedoch lediglich das grobe Konzept erklärt und nicht der konkrete Algorithmus von E. Schömer zum Lösen des Kreispackproblems erläutert werden. Das Verfahren der simulierten Abkühlung bietet als Top-Down-Algorithmus einen Kontrast zu dem vorgestellten Bottom-Up-Algorithmus, dem CSA.

Die simulierte Abkühlung basiert auf dem Metropolisalgorithmus. Durch diesen können Zustände eines thermodynamischen Systems bei einer Temparatur T bestimmt werden. In einem iterativen Verfahren wird in jedem Schritt mit einer bestimmten Wahrscheinlichkeit ein neuer Zustand angenommen. Diese Wahrscheinlichkeit hängt einerseits von der Energiedifferenz der beiden Zustände, andererseits von der Temperatur T ab. Bei einer negativen Energiedifferenz ändert sich der Zustand immer. Bei einer positiven Differenz ist es umso wahrscheinlicher, dass sich der Zustand ändert wenn T groß und die Energiedifferenz klein ist.
Bei der simulierten Abkühlung wird dieses Verfahren mit einer hohen Temperatur T begonnen und diese dann schrittweise verkleinert. Bei großem T können von einem Zustand aus viele andere Zustände erreicht werden, da die Wahrscheinlichkeit für jeden Zustand höher ist. Ist T hingegen klein können nur wenige Zustände erreicht werden, da Zustandswechsel unwahrscheinlicher sind.

Wenden wir den Algorithmus zur Suche eines Optimums an bei dem wir die Güte mit Hilfe einer Zielfunktion beschreiben können. Dann beschreibt ein Zustand x eine bestimmte Lösung. Die Energiedifferenz von Zustand x_0 zu Zustand x_1 wird als Verhältnis zwischen den Güten der beiden Lösungen interpretiert. Das heißt ist die Lösung x_1 besser als x_0 ist die Energiedifferenz negativ, bei einer schlechteren Lösung x_1 ist die Energiedifferenz positiv. Je schlechter die Lösung x_1 im Verhältnis zu Lösung x_0 ist, desto größer ist die positive Energiedifferenz.
Wenden wir die simulierte Abkühlung auf ein Optimierungsproblem an, werden zu Beginn, bei hoher Temperatur, oft auch schlechtere Lösungen akzeptiert. Mit sinkender Temperatur werden allerdings nur noch Lösungen akzeptiert die sich von der Güte immer weniger von den vorhergehenden Lösungen unterscheiden oder besser sind. Bei ausreichend geringer Temperatur bleibt die Lösung in einer ϵ-Umgebung eines lokalen Optimums (für $\epsilon > 0$).
Im Allgemeinen kann dieser Algorithmus nicht das Finden eines globalen Optimums garantieren. Durch die Geschwindigkeit der Abkühlung, d.h. die Rate mit der die Temperatur T sinkt, der Startgröße von T sowie der Bestimmungsmethode für eine Startlösung kann jedoch die Wahrscheinlichkeit ein globales Optimum zu finden vergrößert werden. Konkrete Methoden müssen allerdings zu bestimmten Problemen einzeln bestimmt werden.

Ein beliebtes Beispiel zur anschaulichen Erklärung der simulierten Abkühlung ist das Suchen eines Minimums in einer hügeligen Landschaft indem eine Kugel an einer beliebigen Position abgesetzt wird. Die Kugel wird den Hügel auf dem sie gestartet ist runterrollen und im Tal zwischen zwei Hügeln liegen bleiben. Diese Stelle beschreibt ein lokales Minimum, muss aber kein globales Minimum sein. Bei der simulierten Abkühlung wird der Kugel erlaubt zeitweise auch nach oben zu rollen. Ihr wird also ein Stoß versetzt durch den die Kugel einen kleinen Hügel überwinden kann, aus einem tiefen Tal, wie dem globalen Minimum aber nur schwerer wieder herauskommt. An diesem Beispiel verdeutlicht sich auch das Problem dieser Methode, da man nicht davon ausgehen kann, dass das globale Minimum sehr

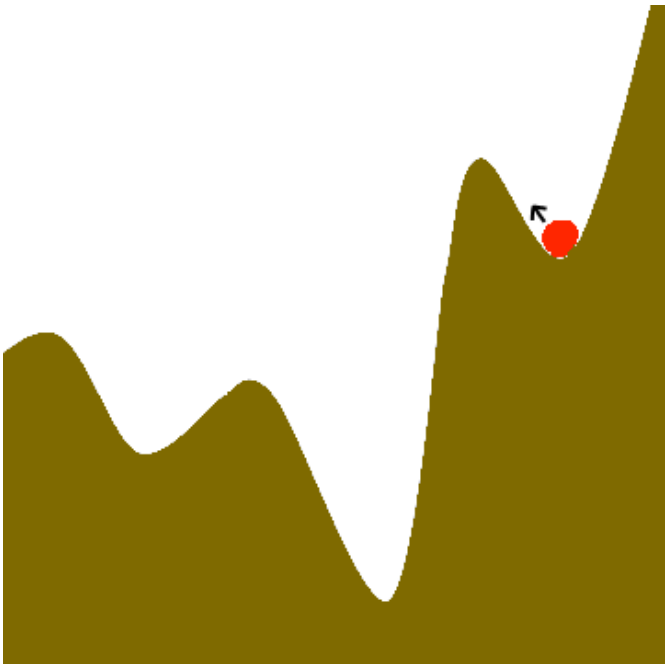


Abbildung 7.1: Simuliertes Abkühlen: Beispiel Kugel

hohe angrenzende Hügel hat sodass die Kugel aus diesem nicht mehr rausrollen kann. Außerdem kann es passieren, dass die Kugel in ein lokales, aber nicht globales, Minimum rollt während die Temperatur schon soweit abgekühlt ist, dass das lokale Minimum nicht mehr verlassen werden kann.

Diese Methode hilft uns auch beim Lösen des Kreispackproblems. Beginnt man mit einer beliebigen Lösung fällt es zunächst noch leicht ähnliche Lösungen zu finden. Insbesondere werden in den ersten Iterationen des Algorithmus auch noch viele Möglichkeiten zur Verbesserung des Zustands und somit mögliche Zustandsänderungen auch bei geringer Temperatur vorhanden sein. Sobald die Kreisscheiben einigermaßen eng zusammen liegen gibt es dieses Verbesserungspotential nur noch in geringem Maße. Das vertauschen zweier Kreisschreiben in einer Lösung kann beispielsweise schon über sehr schlechte Lösungen führen. Um zu umgehen, dass der Algorithmus nach einigen Iterationen an einem lokalen Minimum hängen bleibt wurden im Algorithmus von E.Schömer kurzzeitige Überschneidungen der Kreisscheiben erlaubt. Durch diese Funktionsweise konnten sehr effizient gute Lösungen für das Problem berechnet werden.

8 Vergleich CSA - simulierte Abkühlung

Abschließend wird hier kurz auf die Vorteile der beiden Algorithmen eingegangen. Der ausführlich vorgestellte Cutting Stock Algorithm liefert eine optimale Lösung in, im Verhältnis zu Wang's Algorithmus, bei den betrachteten Beispielen, kurzer Zeit. Es handelt sich um einen Bottom-Up-Algorithmus der durch die vielen verschiedenen Kombinationsmöglichkeiten von Gütern (zwei Güter können auf acht verschiedene Arten kombiniert werden, siehe oben) nur auf kleine und mittelgroße Probleme angewendet werden kann.
Die simulierte Abkühlung findet im Allgemeinen nur ein lokales Optimum und kann nicht sicher entscheiden ob ein globales Optimum gefunden wurde. Da aber nur eine Lösung betrachtet und diese sukzessive verändert wird ist die Größe des betrachteten Problems unkritischer als beim CSA.
Da es sich bei der simulierten Abkühlung um ein Konzept und nicht um einen ausformulierten Algorithmus handelt kann man hier keine konkreten Aussagen zur Rechenzeit dieser machen.

Je nach Problemstellung und Anforderung an die Lösung können also beide Algorithmen dem jeweils anderen vorzuziehen sein.

9 Anhang A: CSA in Java

Die Implementierung des Cutting Stock Algorithm in Java findet sich auf der beigelegten CD.

Bei der Implementierung ist zu beachten, dass diese sich eigene Probleminstanzen generiert und löst. Daher werden die Instanzen und Lösungen aus verschiedenen Durchläufen der Implementierung sich in der Regel unterscheiden. Die Ergebnisse aus Abschnitt **5.3 Lösen von großen Problemen mit Hilfe des CSA** wurden anhand der beigefügten Implementierung mit manuell eingegebenen Probleminstanzen erstellt.
Darüber hinaus wird beim generieren von Problemen nach Vasko ein größerer Faktor k gewählt um die Berechnung zu beschleunigen. Dies ist nötig, da hier im Gegensatz zu den Berechnungen von Amaral und Wright nicht im vorhinein der optimale Wert für β gewählt wird. Damit der Algorithmus zum testen in angemessener Zeit durchläuft werden außerdem zu jeder möglichen Parameterwahl von n, H, $Qmax$ nur eine Instanz generiert (anstatt 5).

Literaturverzeichnis

[1] André R.S. Amaral, Mike Wright *Efficient algorithm for the constrained two-dimensional cutting stock problem*. Intl. Trans. in Op. Res. 8 (2001) 3-13 1999

[2] Francesco Kriegel, Matthias Lange *2-D Guilloutine Zuschnitt Wang-Algorithmus* TU Dresden - Fakultät Mathematik 2009

[3] André Müller, Johannes J. Schneider, Elmar Schömer *Packing a multidisperse system of hard disks in a circular environment* Department of Physics, Mathematics, and Computer Science Johannes Gutenberg University of Mainz 2009

[4] S.Kirkpatrick, C.D. Gelatt, Jr., M.P. Vecchi *Optimization by Simulated Annealing* Science, New Series, Vol.220, No. 4598. (May 13, 1983), pp. 671-680 1983

[5] Sekundärquelle: Foerster, Hildegard *Fixkosten- und Reihenfolgeprobleme in der Zuschnittplanung* Frankfurt am Main 1998 zitiert in Primärquelle: Zajons, Andreas *Zuschnittplanung im Stahlbau* Universität zu Köln 2001