Karlsruhe Institute of Technology

Entwicklung einer anwendungsorientierten Packheuristik mit Hohlraumdefragmentierung

Bachelorarbeit
von

Tilman Väth

An der Fakultät für Informatik
Institut für Theoretische Informatik

Erstgutachter:	Prof. Dr. Dorothea Wagner
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuende Mitarbeiter:	Dipl.-Inform. Felix Brandt
	Dipl.-Inform. Benjamin Niedermann
	Dr. Martin Nöllenburg

Bearbeitungszeit: 3. März 2014 – 2. Juli 2014

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und dass alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit als solche gekennzeichnet sind.

Karlsruhe, 2. Juli 2014

Zusammenfassung

In dieser Arbeit wird die Entwicklung einer dreidimensionalen Packheuristik für ein spezielles Containerladeproblem vorgestellt. Das Ganze erfolgt dabei auf Basis von zwei unterschiedlichen wissenschaftlichen Arbeiten, welche miteinander kombiniert werden. Die Grundlage bildet die erste Heuristik, welche auf einer lokalen Suche beruht. Diese optimiert die Güte der Lösung alleine durch die Veränderung der Beladesequenzen, da dabei ein einfacher, deterministischer Belademechanismus verwendet wird. Darauf aufbauend, werden Konzepte der zweiten Arbeit für ein allgemein gefasstes Bin-Packing Problem genutzt. Dieses versucht vor allem Freiräume zu defragmentieren und dadurch eine kompaktere Beladung zu erreichen.

Inhaltsverzeichnis

1. Einführung

In der heutigen Zeit wird es aus wirtschaftlichen Gründen immer wichtiger, Prozesse zu optimieren, um den eigenen Gewinn zu steigern, sowie konkurrenzfähig zu bleiben. Dies ist natürlich auch in der Logistik der Fall, insbesondere bei der Beladung von Containern oder anderen Behältern mit Gütern. Dafür werden immer wieder neue Algorithmen entwickelt, um eine möglichst effiziente und kompakte Beladung zu erhalten, denn jeder freie Raum ist gleichzusetzen mit einem finanziellen Schaden für das betroffene Logistikunternehmen.

Diese Arbeit befasst sich genau mit dieser Problematik und stellt die Entwicklung einer anwendungsorientierten 3D-Packheuristik vor. Die Grundlage bilden zwei unterschiedliche Verfahren, über die hier kurz ein Überblick gewährt wird. In den nachfolgenden Kapiteln werden die einzelnen Komponenten noch detaillierter vorgestellt.

Das Grundgerüst bildet der Algorithmus von Ceschia und Schaerf[1]. Er ist speziell dafür ausgelegt, die unterschiedlichen Zieldestinationen der Boxen zu beachten, um ein aufwändiges Entladen zu vermeiden. Für Boxen die an der momentanen Station entladen werden müssen, soll dies ohne das Verschieben oder Entladen von Boxen, welche zu einem späteren Zeitpunkt entladen werden sollen, möglich sein. Diese Problemeigenschaft wird *Multi-Drop* genannt. Der verwendete Belademechanismus garantiert, dass bei einer nach Zieldestinationen geordneten Beladesequenz, die resultierende Beladung die Multi-Drop Eigenschaft erfüllt. Außerdem erfolgt das Verladen der Boxen immer deterministisch. Aus gleichen Beladesequenzen ergibt sich also immer die gleiche Beladung. Die Lösung hängt somit nur von der Reihenfolge ab, in der die Boxen verladen werden. Diese Eingabesequenzen werden in jedem Lösungsschritt zufällig geringfügig verändert. Ist die daraus resultierende Lösung der neuen Eingabesequenzen besser, als die vorherige Lösung, wird im nächsten Schritt mit dieser Lösung fortgefahren. Nach diesem Schema soll eine lokale Suche am Ende ein bestmögliches Ergebnis liefern.

Der Algorithmus von Zhang et al.[2] befasst sich im Gegensatz dazu, mit einer viel allgemeiner definierten Problembeschreibung für das 3D Bin-Packing. Dabei versucht er freie Positionen in der Ladungsanordnung auszunutzen und dort Boxen einzufügen. Somit lassen sich unter anderem unschöne Hohlräume füllen. Es ist dabei nicht unbedingt nötig, dass die Box klein genug ist, um sie an solch einer Position zu platzieren. Indem andere Boxen beiseite geschoben werden, kann für größere Boxen zusätzlicher Freiraum entstehen. Passt die Box nun in diesen vergrößerten, freien Raum, kann sie eingefügt werden. Danach werden die verschobenen Boxen wieder Richtung Ursprung verschoben. Ist es so trotzdem nicht möglich eine Box einzufügen, gibt es einen weiteren Ansatz. Bei diesem soll eine

Box, durch eine andere mit höherem Gütemaß ersetzt werden. Auch hier wird zusätzlicher Raum geschaffen, indem andere Boxen beiseite geschoben werden. Durch Einfügen an einer freien Position, sowie durch das Ersetzen einer anderen Box, wird jeweils die gewünschte Optimierung(Volumen, Kosten, etc.) erreicht.

Für die entwickelte Packheuristik wurde das Verfahren von Ceschia und Schaerf so gut wie möglich nach implementiert und sich genau an dessen Problembeschreibung und die damit eingehenden Restriktionen gehalten.

Konkret handelt es sich bei dieser Arbeit damit um ein dreidimensionales Containerladeproblem, bei dem Boxen rotiert werden können, jede Seite einer Box nur ein bestimmtes Gewicht tragen kann, die Multi-Drop Eigenschaft einzuhalten ist, Boxen beim Stapeln komplett auf den direkt darunterliegenden Boxen aufliegen müssen und jeder Container nur ein maximales Gesamtgewicht tragen kann.

An diese Problembeschreibung wurden dann die einzelnen Komponenten des Verfahrens von Zhang et al. angepasst und implementiert. Daraus entstanden dann ein serieller und ein verschachtelter Ansatz für eine Packheuristik. Bei der seriellen Variante wird eine Lösung der Heuristik von Ceschia und Schaerf im Anschluss noch von der Heuristik von Zhang et al. verfeinert. Bei der verschachtelten Variante wird direkt während dem Beladevorgang der lokalen Suche probiert, bereits entstandene Hohlräume zu füllen. Beide Varianten werden vorgestellt und miteinander verglichen.

Abschließend sei an dieser Stelle noch erwähnt, dass durch die Multi-Drop Eigenschaft auch ein darunter liegendes Routing Problem existiert. Dies sollte im Hinterkopf behalten werden, ist aber nicht Gegenstand dieser Arbeit.

1.1. Verwandte Arbeiten

Die Kombination der Features dieses Problems wurden bisher einzig von Ceschia und Schaerf[1] behandelt. Allein die Eigenschaft, dass sich die Tragkraft einer Box in Abhängigkeit zu ihrer vertikalen Ausrichtung ändern kann, wird nur von Bischoff [6] beachtet. Er analysiert dabei alle möglichen Kombinationen von Freiräumen und noch nicht geladenen Boxen mit ihren möglichen Orientierungen. Anhand eines Scoringschemas, welches fünf unterschiedliche Kriterien heranzieht, wird die Kombination mit dem höchsten Score gewählt und die Box entsprechend platziert.
Das Prinzip der Freiraumdefragmentierung hat bis jetzt noch keine Rolle gespielt, wenn es darum ging eine Packheuristik für Container zu entwickeln. Selbst für das gewöhnliche Bin-Packing gibt es nur sehr vereinzelt Ansätze. Die Heuristik von Zhang et al. [2], welche in dieser Arbeit Verwendung findet, ist dabei die Neueste und ebenso Beste auf diesem Gebiet.
Eine generelle Bewertung wissenschaftlicher Arbeiten zum Thema Containerbeladung und entsprechende Einteilung in diverse Problemklassen führen Bortfeldt und Wäscher [7] durch. Dadurch kann ein genereller Überblick gewonnen werden und die führenden Strategien für die jeweiligen Problemklassen gefunden werden. Daraus geht auch hervor, dass sich die Forschung auf diesem Gebiet noch in ihren Kinderschuhen befindet, vor allem wenn man die Modellierung von Problemen mit praxisrelevanten Bedingungen betrachtet.

1.2. Gliederung

In diesem Abschnitt soll ein kleiner Überblick über die Gliederung dieser Bachelorarbeit erfolgen.

In Kapitel 2 werden zunächst ein paar Grundlagen geschaffen, auf die in der Folge zurückgegriffen wird.

In Kapitel 3 wird die Packheuristik von Ceschia und Schaerf[1] vorgestellt. Dabei findet zu Beginn eine Problembeschreibung statt und im Anschluss werden die einzelnen Bestandteile des Verfahrens genau vorgestellt. Zu jedem dieser Bestandteile gibt es eine Einführung, in der die zugehörige Funktionalität genauer vorgestellt wird, sowie einen Abschnitt, der sich mit der Implementierung dazu befasst. Zum Ende des Kapitels wird noch erläutert, wie aus den einzelnen Bestandteilen der komplette Algorithmus entsteht und wie eine Lösung auf Einhaltung der Multi-Drop Restriktion geprüft wird.

In Kapitel 4 wird für die Heuristik von Zhang et al.[2] genauso Verfahren wie in Kapitel 3. Zusätzlich werden gleich zu Beginn die Unterschiede zum Problem des vorherigen Kapitels aufgeführt. Für die Implementierung der einzelnen Komponenten wird dabei immer erwähnt, was gegenüber dem Original verändert wurde, um dieses an das Problem aus Kapitel 3 zu adaptieren.

In Kapitel 5 werden nun die beiden Ansätze für eine neue Packheuristik vorgestellt. Der eine ist eine serielle Variante, der andere kombiniert die vorgestellten Verfahren ineinander verschachtelt. Es wird dabei jeweils auf bereits vorgestellte Implementierungen aus den Kapiteln 3 und 4 zurückgegriffen.

In Kapitel 6 werden die umgesetzten Implementierungen gegeneinander evaluiert und dabei auch eine Methode vorgestellt, welche Lösungen validiert.

In Kapitel 7 wird abschließend ein Fazit geschlossen und ein Ausblick gewährt, was noch an den eigens implementierten Packheuristiken verbessert werden könnte.

2. Grundlagen

In diesem Kapitel werden ein paar Grundlagen geschaffen und Definitionen eingeführt, die im weiteren Verlauf dieser Arbeit Verwendung finden.

2.1. Bin-Packing

Das allgemeine Bin-Packing Problem für eine Dimension ist folgendermaßen definiert:

Gegeben ist eine Menge $A = [a_1, ..., a_n]$ von n Elementen mit den Größen g_{a_i} und eine unbegrenzte Anzahl an Behältern B einer bestimmten Größe b.

Gesucht ist eine Beladung der Objekte in eine minimale Anzahl an Behältern. Für die Elemente innerhalb eines Behälters B' gilt dabei stets:

$$\sum_{a_i \in B'} g_{a_i} \leq b$$

Für das 2D und 3D Bin-Packing nehmen die Entitäten dann entsprechend eine zwei- bzw. dreidimensionale Größe an.

Ausgehend von einem 3D Bin-Packing lassen sich dann leicht Containerladeprobleme definieren. Der Container nimmt dann die Rolle eines Behälters ein und eine Box ist gerade ein Objekt, das in dem Container platziert werden soll. Das Verladen der Boxen kann dabei verschiedensten Restriktionen unterliegen, je nachdem welche Problemstellung modelliert werden soll. Für den allgemeinsten Fall wird nur vorgeschrieben, dass sich jede Box innerhalb eines Containers befinden muss und sich keine zwei Boxen schneiden dürfen.

2.2. Lokale Suche

Die lokale Suche ist ein heuristisches Suchverfahren für Optimierungsprobleme. Im Allgemeinen ist sie dadurch gekennzeichnet, dass ausgehend von einer Startlösung näherungsweise eine möglichst gute - im Idealfall die optimale - Lösung, gefunden wird. Durch eine bestimmte Veränderung der aktuellen Lösung, soll hierbei eine bessere Lösung aus der Nachbarschaft gefunden werden. Im welchem Rahmen sich diese Veränderung abspielen darf, ist zu Beginn festzulegen. Die Nachbarschaft ist dabei gerade die Menge aller Lösungen, welche aus der aktuellen Lösung durch solch eine Veränderung gewonnen werden kann. Wird eine bessere Lösung in der Nachbarschaft gefunden, wird diese zur aktuellen Lösung und es geht nach

dem selben Prinzip weiter. Um bewerten zu können, ob eine Lösung besser oder schlechter ist wie eine andere, wird eine Zielfunktion f_{Ziel} verwendet, welche für jede Lösung L_i einen Zielwert $f_{Ziel}(L_i)$ liefert. Es kann natürlich vorkommen, dass die Zielfunktion lokale und globale Optima aufweist und somit die lokale Suche gegen ein lokales Optima konvergiert. Dann gilt für die Lösung L', welche ein lokales Optima liefert: $f_{Ziel}(L') \geq f_{Ziel}(L_j)$ für jeden Nachbarn j des aktuellen Zustandes.

2.2.1. Simuliertes Abkühlen

Die lokale Suche weist das Problem auf, dass sie in lokalen Optima hängen bleibt. Das simulierte Abkühlen[8] ist eine Spezialisierung der lokalen Suche, die dem entgegenwirkt. Sie erlaubt es auch Nachbarn zu akzeptieren, welche keine bessere Lösung liefern, umso den Sprung aus lokalen Optima zu schaffen.

Allgemein wird ein Startwert(Starttemperatur) kontinuierlich abgesenkt(abgekühlt). Für jeden Zwischenwert(Temperaturlevel) wird eine bestimmte Anzahl an Iterationen durchgeführt. Dabei wird für jede Iteration ein Nachbar der aktuell besten Lösung berechnet und mittels einer Zielfunktion f die Güte der Lösung des Nachbarn bestimmt. Ist die neue Lösung besser, wird sie akzeptiert und zur besten Lösung. Ist sie schlechter, wird sie mit einer temperaturabhängigen Wahrscheinlichkeit trotzdem akzeptiert. Dies soll das Entkommen aus einem lokalen Optima ermöglichen. Das Verfahren endet, wenn ein bestimmtes Abbruchkriterium eintritt.

2.3. Definition von Koordinatensystem und den räumlichen Anordnungen

In diesem Abschnitt wird das verwendete Koordinatensystem und die Anordnung der Container und Boxen darin definiert. Dazu werden die räumlichen Anordnungen in Abb. 2.1 betrachtet.

Ein Container ist immer im ersten Quadranten des kartesischen Koordinatensystem angeordnet, mit der linken, unteren Ecke im Ursprung. Die Länge verläuft parallel entlang der x-Achse, die Breite entlang der y-Achse und die Höhe entlang der z-Achse. Die Anordnung der Seiten einer Box erfolgt identisch. Die Tiefe ist dabei equivalent zur Länge. Die Grundfläche ist gerade die Fläche, die sich auf der xy-Ebene, oder parallel dazu befindet. Die x-, y- und z-Achse ist immer genauso wie in Abb. 2.1 definiert. Die Anordnung der Boxen 1-4 zueinander wird folgendermaßen beschrieben: Box 2 befindet sich *über* Box 1, Box 3 befindet sich *neben* Box 1 und Box 4 befindet sich *auf der Grundfläche über* Box 3. Zudem wird ein Stapel, der aus aufeinander gelegten Boxen entlang der z-Achse besteht, in der Höhe in unterschiedliche Ebenen separiert.

Da viele Problemstellungen auftauchen, die viel leichter anhand von zweidimensionalen Beispielen erläutert werden können, wird oft auf zweidimensionale Koordinatensysteme zurückgegriffen. Dies entspricht dann einer Ansicht des Szenarios aus der Vogelperspektive. Um konsistent zur dreidimensionalen Ansicht zu bleiben, verändert sich die Position der y-Achse deshalb zum dreidimensionalen Fall nicht(s. Abb. 2.1). Die x-Achse verläuft dann senkrecht dazu. Darauf muss geachtet werden, da die Achsen bei zweidimensionalen Koordinatensystemen meistens genau andersherum beschriftet sind(x-Achse horizontal, y-Achse vertikal).

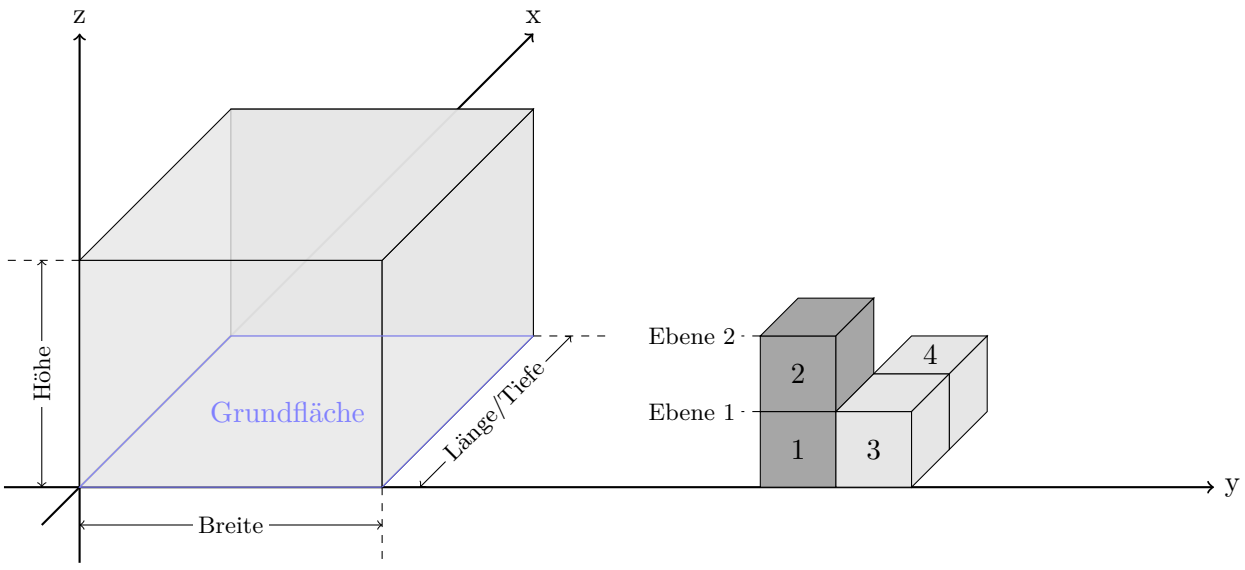


Abbildung 2.1.: Anordnung eines Container / Box im kartesischen Koordinatensystem mit Angabe der zugehörigen Ausmaße. Dazu Anordnung der Boxen zueinander und Unterteilung eines Stapels(dunkel hervorgehoben) in Ebenen.

3. Lokale Suche für das Multi-Drop Problem

3.1. Problembeschreibung

Die zugrunde liegende Arbeit von Ceschia und Schaerf [1] befasst sich mit einer komplexen Variante des praktischen Packproblems für Container, also einem Spezialfall des 3D Bin-Packing Problems. Dabei soll eine Menge an rechteckigen Boxen in eine Menge von rechteckigen Containern geladen werden. Jeder Box und jedem Container sind bestimmte Kosten zugeordnet. Ziel ist es, eine möglichst günstige Beladung der Container zu finden.

Jeder Containertyp führt folgende Informationen mit sich: Eine eindeutige ID, die verfügbare Anzahl, die kompletten Maße bestehend aus $länge \times breite \times höhe$, das tragbare Gesamtgewicht und die anfallenden Nutzungsgebühren.

Jeder Boxtyp führt folgende Informationen mit sich: Eine eindeutige ID, die verfügbare Anzahl, die kompletten Maße bestehend aus $länge \times breite \times höhe$, für jede Achse, ob eine Rotation darum möglich ist, für jede Seite das tragbare Gewicht pro Flächeneinheit, eine Zieldestination und ihre Kosten. Die Kosten geben dabei an, welcher finanzielle Verlust zum Tragen kommt, falls die Box nicht verladen werden kann.

Um eine gültige Lösung zu erhalten, sind bei der Beladung einige Dinge zu beachten, die das Problem noch detaillierter beschreiben:

- Die maximale Tragelast des Containers darf nicht überschritten werden.
- Boxen dürfen nur derart gestapelt werden, dass diese mit ihrer kompletten Grundfläche auf den direkt darunter liegenden Boxen aufliegen.
- Eine Box kann nur ein bestimmtes Gewicht pro Flächeneinheit tragen. Dieses kann je nach Orientierung anders sein.
- Die Multi-Drop Restriktion muss erfüllt sein.

Zusätzlich ist es gestattet Boxen zu rotieren. Je nach Typ ist dies aber nur um bestimmte Achsen möglich.

Kostenfunktion

Um die Kosten einer Lösung zu bestimmen wird eine Kostenfunktion f definiert, die auch später die lokale Suche leiten wird. Diese setzt sich für jeden genutzten Container und

der dazugehörigen Beladesequenz, aus einzelnen Kostenkomponenten $c_1, ..., c_4$ zusammen. Konkret sind die Komponenten wie folgt definiert:

- c_1 = Kosten aller Boxen, die nicht verladen werden konnten.
- c_2 = Nutzungskosten des Containers.
- c_3 = Linear freier Raum auf der x-Achse des Containers.
- c_4 = Entladestopps. Summe der unterschiedlichen Zielorte der Boxen, die im Container verladen sind.

Die Komponente c_3 gibt dabei gerade den Abstand der Box b, welche am tiefsten auf der x-Achse in den Container ragt, zur Containerwand an:

$$c_3 = Containerlänge - (b.position.x + b.boxlänge)$$

Sie soll dichtere Packungen bevorzugen, denn dadurch resultiert mehr verbleibender Freiraum für weitere Boxen. Komponente c_4 zielt auf die Fahrtkosten ab, idealerweise sollte jeder Container nur zu einem Ziel transportiert werden. Die konkreten Kostenangaben wurden der Ceschia Gruppe von ihrem Industriepartner zur Verfügung gestellt und sind in den Beispiel-Instanzen vorhanden.

Insgesamt ergibt sich für die Kostenfunktion f nun:

$$f = w_1 \sum c_1 + w_2 \sum c_2 + w_3 \sum c_3 + w_4 \sum c_4$$

Wobei $w_1, ..., w_4$ die Gewichte der einzelnen Kostenkomponenten darstellen.

3.2. Beladen des Containers

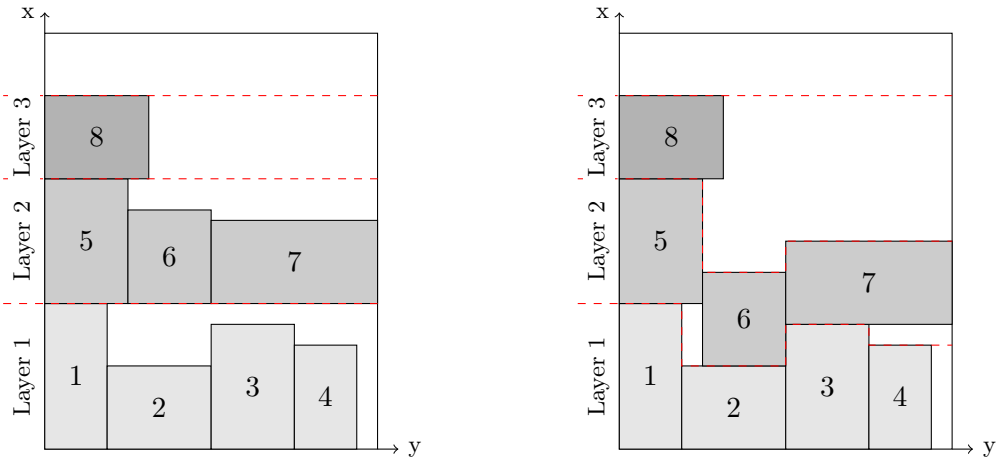
Der Belademechanismus, in der Folge *Loader* genannt, ist einer der zentralen Bestandteile des Algorithmus von Ceschia und Schaerf[1] und damit auch der kompletten Abschlussarbeit. Er ist dafür zuständig, eine Menge von Boxen nach einem bestimmten Schema in einen Container zu laden. Das Vorgehen basiert auf dem "Wall building approach" von George und Robinson[5]. Durch einige Sonderfälle bei diesem Verfahren kommt es auf viele Details an und somit ist eine genaue Beschreibung notwendig, um die Vorgehensweise lückenfrei nachvollziehen zu können. In dieser Arbeit wurde versucht die Implementierung des Loaders möglichst nahe an der Beschreibung durch die Ceschia Gruppe zu halten, dies lässt wie im vorliegenden Paper selbst bereits erwähnt, noch einige Optimierungsmöglichkeiten offen. Anstatt einem verladen nach dem *First Fit* Prinzip, könnte zum Beispiel als nächstes die Box verladen werden, die am besten passt. Jedoch soll die Güte des Resultats der Ausgabe, vor allem durch die lokale Suche erreicht werden und der Loader möglichst schnell und einfach arbeiten. Die Implementierung erwies sich jedoch etwas schwerer als gedacht, da im Paper dieses Modul nicht vollständig beschrieben ist. Die Recherche der Arbeit von George und Robinsion[5], auf der der Loader basiert, sowie der Arbeit von Oliveira[3], deren Verfahren ebenfalls auf dem "Wall building approach" beruht, brachte leider keinen erhofften Fortschritt. Letztlich wurde durch eine Email an die Autorin, sowie die Durchsicht der zur Verfügung gestellten Lösungen klar gestellt, wie das Verfahren komplett abläuft. Die Umsetzung des Verfahren musste jedoch selbst entwickelt werden, denn dazu wurden keine Informationen geliefert. Dies ist durchaus nachvollziehbar, da es sich bei der kompletten Arbeit der Autoren, um ein kommerzielles Softwareprojekt für einen industriellen Partner handelt.

Aus denen in diesem Abschnitt geschilderten Vorkommnissen erschien es besonders wichtig, eine detaillierte und nachvollziehbare Beschreibung des Loaders in dieser Arbeit darzulegen.

3.2.1. Vorgehensweise des Algorithmus

Das Prinzip des Belademechanismus ist relativ einfach: Der Container wird entlang der x-Achse in verschiedene, flexible *Layer* aufgeteilt, welche wiederum entlang der y-Achse in mehrere Stapel unterteilt werden. Dabei wird zuerst der erste Layer aufgebaut und beladen, danach der zweite, usw.

Hierbei sei noch kurz dass Prinzip der Layer, wie in Abbildung 3.1 illustriert, erklärt. Ein Layer wird immer durch eine Box aufgespannt, welche direkt links an der Wand des Containers platziert wird. Der Layer ist nun entlang der x-Achse durch die Länge der Box begrenzt, die Layergrenze. Die nächsten Boxen, welche auf der Grundfläche des Containers platziert werden müssen, werden nun jeweils direkt nebeneinander platziert. Dies geschieht solange, bis eine Box nicht mehr platziert werden kann, da sie ansonsten mit der Außenwand des Containers kollidiert, oder zu weit in die Tiefe ragt, sodass die Layergrenze überschritten wird. In beiden Fällen wird die Box dann auf der Grundfläche direkt über die erste Box des Layers platziert und lässt so einen neuen Layer entstehen, welchen es zu füllen gilt. Bei starren Layern(Abb. 3.1a) werden die Boxen jeweils horizontal direkt an der Layergrenze des vorhergehenden Layers ausgerichtet. Bei flexiblen Layern(Abb. 3.1b), wie es hier der Fall ist, können die Boxen in den vorhergehenden Layer einsinken, was eine kompaktere Beladung ermöglicht. Dieses einsinken wird umgesetzt, indem der übrig gebliebene Freiraum des vorherigen Layers, mit dem neuen, momentanen Layer verschmolzen wird.



(a) Starre Layer

(b) Flexible Layer

Abbildung 3.1.: Layer, die Layergrenzen sind rot eingezeichnet.

Die Beladesequenz für jeden Container besteht aus einer Reihenfolge von mehreren Blöcken. Jeder Block besteht wiederum aus einer bestimmten Anzahl gleicher Boxen, welche alle die gleiche Orientierung im Raum innehaben. Somit ist ein Block charakterisiert durch eine eindeutige ID, einen Boxtyp, die Anzahl der Boxen und deren momentane Orientierung.

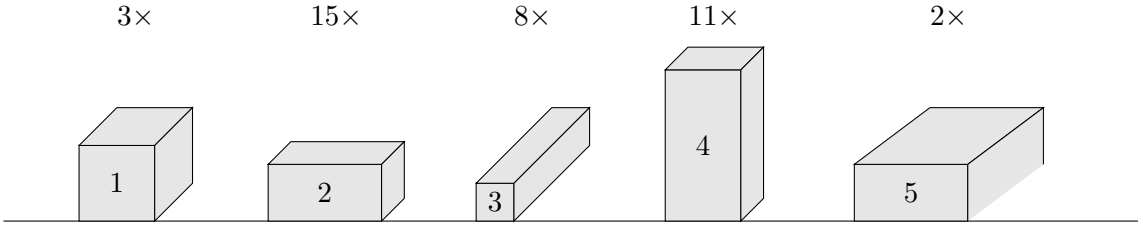


Abbildung 3.2.: Beispiel für eine Beladesequenz eines Containers

Ein Container wird nun mit einer Beladesequenz folgendermaßen beladen: Zu Beginn wird die erste Box b_1 mit Dimension (l, w, h) des ersten Blocks in der linken unteren Ecke des Containers, im Punkt $(x, y, z) = (0, 0, 0)$ platziert. Dadurch wird ein neuer Layer erzeugt, welcher entlang der x-Achse durch die Länge der Box begrenzt ist, die Layergrenze. Entlang der Breite des Containers wird der Layer in natürlicher Weise durch die parallel liegende, rechte Außenwand des Containers begrenzt.
Nun wird zuerst versucht die nächsten Boxen aus der Eingabesequenz auf die eben eingefügte Box aufzulegen und so einen Stapel zu bilden, wobei Größen- und Gewichtsrestriktionen einzuhalten sind. Es ist dabei möglich mehrere Boxen nebeneinander auf die vorherige Box aufzulegen. Dies ist aber nur Boxen des gleichen Typs gestattet, damit eine Ebene mit gleicher Höhe erhalten bleibt. Ist die letzte Box auf der aktuellen Ebene des Stapels platziert, bilden die eben aufgelegten Boxen die neue Auflagefläche, auf der der Stapel weiter aufgebaut werden kann. Sollte es aus den bereits genannten Gründen nicht mehr möglich sein, Boxen auf diesen Stapel zu legen, gilt dieser als abgeschlossen und es wird in der Folge nicht mehr versucht weitere Boxen darauf aufzulegen.
Als nächstes wird probiert die Box entlang der y-Achse auf dem Grund des Containers zu positionieren. Dies geschieht immer von links nach rechts, solange keine Box die Layergrenze überragt, oder zu breit ist für den übrig geblieben Platz. Kann auf diese Weise eine Box platziert werden, wird zuerst wieder versucht einen größtmöglichen Stapel auf ihr zu bilden. Danach kann dann die nächste Box entlang der y-Achse positioniert werden. Es gibt dabei jedoch eine Ausnahme: Ist die darauf folgende Box, die eingefügt werden soll identisch (beachte auch Orientierung), wird nicht sofort versucht sie zu stapeln, sondern es wird geprüft, ob sie innerhalb der Layergrenze entlang der x-Achse, direkt über die aktuelle Box auf der Grundfläche des Containers gelegt werden kann. Ist dies der Fall, wird sie dort platziert. Auf diese Weise können mehrere gleiche Boxen direkt übereinander auf der Grundfläche platziert werden(s. Abb. 3.3). Diese bilden nun zusammen die Grundfläche für einen neuen Stapel. Die Ausnahme hat den leicht ersichtlichen Vorteil, dass der freie Raum im Container weniger stark fragmentiert wird. Zudem ergibt sich eine größere Auflagefläche für die nächsten Boxen.

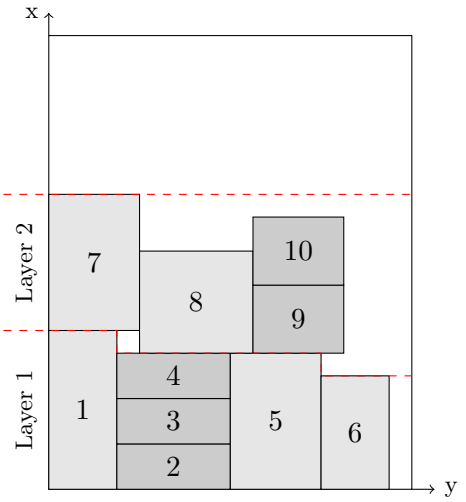


Abbildung 3.3.: Beladung eines Containers aus der Vogelperspektive. Die dunkleren Boxen [2-4 und 9-10] werden ausnahmsweise direkt übereinander auf der Grundfläche platziert.

Würde eine Box nach dem Einfügen die Layergrenze entlang der x-Achse überschreiten, wird diese direkt auf der Grundfläche über die erste Box des aktuellen Layers eingefügt und bildet selbst einen neuen Layer. Jetzt wird wiederum versucht Stapel entlang der y-Achse aufzubauen. Da wie bereits erwähnt flexible Layer(Abb. 3.1b) verwendet werden,

können auch Boxen in den vorhergehenden Layer rutschen, damit möglichst wenig Freiraum zwischen den Boxen entsteht.
Das Verfahren endet sofort, wenn keine Boxen mehr in der Eingabesequenz vorhanden sind, oder die nächste Box nicht mehr in den Container passt.

Algorithmus 3.1 : LOAD(seq, C) - Ablauf des Belademechanismus

Eingabe : Beladesequenz seq; Container C

```
1  abbruchkriterium = true;
2  neuer_layer = true;
3  while abbruchkriterium do
4      if neuer_layer then
5          abbruchkriterium = platzierbar()
6          if abbruchkriterium then
7              Füge Box ein und erstelle neuen Layer
8              neuer_layer = false
9      else
10         if stapelbar() then
11             STAPELN()
12         else
13             neuer_layer = BELADELAYER()
14 return Beladung
```

3.2.2. Implementierung

In diesem Abschnitt soll die Implementierung des Loaders beschrieben werden. Die Umsetzung der gewünschten Funktionalität wurde dabei komplett selbst entwickelt, da das Paper hierzu keine genaueren Informationen, wie zum Beispiel Pseudocodeabschnitte liefert. Zuerst wird darauf eingegangen, wie die Boxen auf der Grundfläche des Containers platziert werden und danach beschrieben, wie das Stapeln der Boxen abläuft.

Freiräume

Zuerst soll das Prinzip der Freiräume vorgestellt werden. Sie werden verwendet, um dem Algorithmus anzuzeigen, wo auf der Grundfläche des Containers verfügbarer Platz für die zu ladenden Boxen ist. Für das Stapeln werden sie nicht verwendet. Jeder Freiraum besitzt eine x- und y- Koordinate, welche gleichzeitig die Einfügepostion darstellen. Entlang der x-Achse wird dieser Freiraum durch die aktuelle Layergrenze eingeschränkt. Damit ist die Länge des Freiraumes w_x entlang der x-Achse, gleich dem Abstand zur Layergrenze. Entlang der y-Achse wird die Einfügeposition des Freiraumes auf die nächste Box, oder die Containerwand projiziert. Dadurch wird festgestellt, welche Breite w_y eine einzufügende Box maximal aufweisen darf. Entlang der z-Achse ist der Freiraum einfach durch die Containerhöhe w_z begrenzt.

Somit ist ein Freiraum F komplett definiert durch das 5-Tupel (x, y, w_x, w_y, w_z). Die Werte w_x und w_z müssen aber nicht mitgeführt werden, denn die momentane Layergrenze und die Höhe des Containers stehen während dem kompletten Beladevorgang zur Verfügung und sind für alle Freiräume gleich. In Abbildung 3.4 ist beispielhaft eine Beladung dargestellt, die drei Freiräume aufweist.

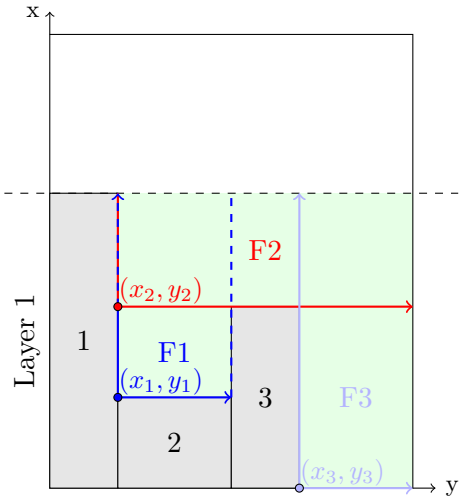
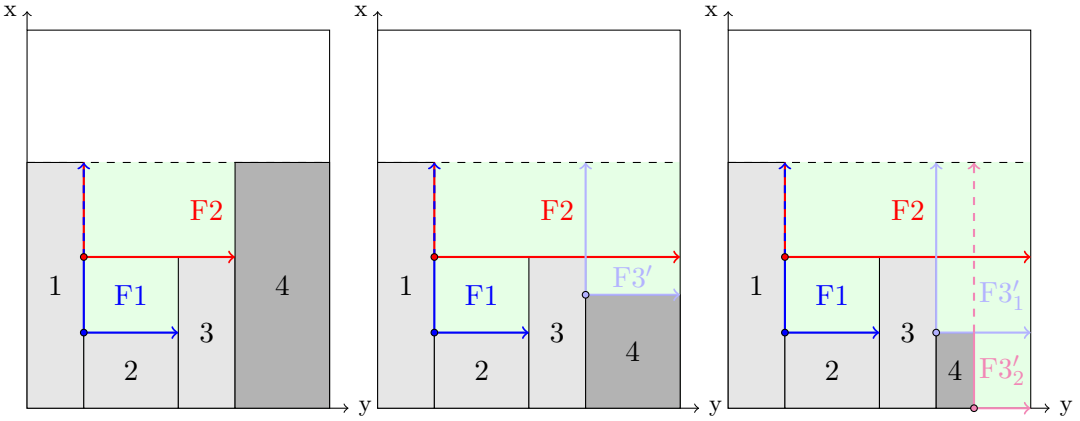


Abbildung 3.4.: Freiräume F1, F2, F3

Wird eine Box in einen Freiraum eingefügt, verändert sich dieser. Es trifft dabei einer der drei folgenden Fälle ein:

1. Die Box passt perfekt hinein. Es ist somit kein weiterer Platz in diesem Freiraum verfügbar und er wird im Anschluss gelöscht, vgl. Abb. 3.5a.
2. Die Box hat die gleiche Höhe/Breite wie der Freiraum. Es muss nur die x/y-Koordinate angepasst werden, vgl. Abb. 3.5b.
3. Die Box ist kürzer und schmaler als der Freiraum. Dann wird der Freiraum in zwei neue, separate Freiräume zerteilt. In einen Oberen, welcher entsteht in dem die x-Koordinate des Freiraums um die Länge der Box erhöht wird und einen Rechten, welcher entsteht in dem die y-Koordinate des Freiraumes um die Breite der Box erhöht wird, vgl. Abb. 3.5c.



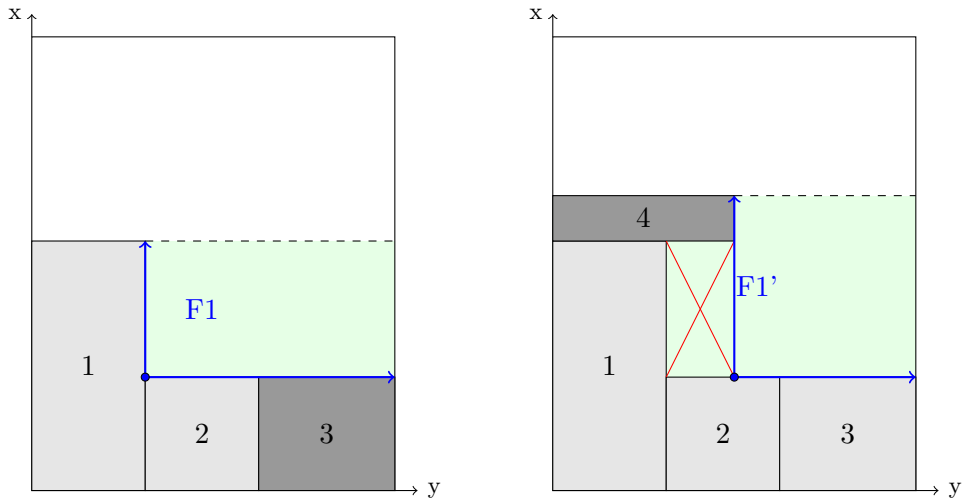
(a) 1.Fall (b) 2.Fall (c) 3.Fall

Abbildung 3.5.: Drei unterschiedlich große Boxen werden in Freiraum F3 aus Abb. 3.4 eingefügt. Je nach Ausmaß der Box entsteht aus F3 0,1, oder 2 neue Freiräume.

Wie man in den Abbildungen 3.4 und 3.5 sehen kann, überschneiden sich oft einige Freiräume, dies stellt aber kein Problem dar. Wird wie in Abb.3.5a eine Box derart eingefügt, dass sie danach auch teilweise in einem anderen Freiraum liegt(Box 4 liegt zum Teil in $F2$), muss dieser danach natürlich angepasst werden. In diesem Fall ist die Breite w_y von $F2$

entlang der y-Achse zu reduzieren. Dadurch dass sich Freiräume überschneiden können, entsteht der Vorteil, dass jeder Freiraum an seiner Position die maximale Größe innerhalb des Layers einnimmt.

Verändert sich einmal ein Freiraum, oder es entstehen neue Freiräume, ist es sehr gut möglich, dass es danach identische Freiräume gibt, oder Freiräume die einen Unterraum eines anderen Freiraumes darstellen. In ersterem Fall werden alle identischen Freiräume bis auf einen gelöscht. In zweiterem Fall wird nur der größere Freiraum erhalten. Unterräume oder identische Freiräume entstehen, falls eine Box auf der Grundfläche über einer anderen platziert wird, sodass die rechte Außenkante beider Boxen auf einer Linie liegt, ohne dass sich dazwischen eine Box befindet. Der analoge Fall tritt auf für Boxen, die links der neuen Box liegen. Nur muss hier die obere Außenkante betrachtet werden (s. Abb. 3.6a). Außerdem kann es vorkommen, dass eine neu eingefügte Box einen anderen Freiraum ganz oder teilweise überdeckt. Wird er ganz überdeckt, wird er gelöscht. Wird er nur teilweise überdeckt, muss dessen y-Koordinate entsprechend nach rechts verlagert werden (s. Abb. 3.6b). In diesem Fall ist zudem der neue Freiraum, der rechts der neu eingefügten Box entsteht, ein Teilraum dieses teilweise überdeckten Freiraumes. Somit kann der Teilraum verworfen werden.



(a) Durch Platzieren von Box 3 würde ein zu F1 identischer neuer Freiraum entstehen. Dieser muss verworfen werden.

(b) Ein Freiraum wird von Box 4 überdeckt. Dieser schrumpft dadurch entlang der y-Achse. Da durch Box 4 auch ein neuer Layer induziert wird, wächst der Freiraum gleichzeitig entlang der x-Achse, da sich die Layergrenze verschiebt.

Abbildung 3.6.: Umgang mit Freiräumen

Alle Freiräume werden aufsteigend nach ihren y-Positionen in einer Liste gehalten. Dies ermöglicht ein leichteres Einfügen der Boxen entlang der y-Achse, von links nach rechts innerhalb eines Layers. Außerdem liegen redundante Freiräume mit gleichen y-Positionen direkt nebeneinander und werden so schnell gefunden.

3.2.3. Erstellen eines neuen Layers

Zu Beginn, wenn noch keine Box eingefügt ist, besteht noch kein Freiraum. Jede Box welche einen neuen Layer beginnt, wird nicht in einen Freiraum, sondern einfach direkt auf der Grundfläche des Containers über der ersten Box des vorherigen Layers, links an der Containerwand eingefügt(s. Box 4 in Abb.3.6b).
Ist der Container noch leer, wird durch Einfügen einer neuen Box b' mit Dimension (l, w, h)

im Ursprung ein initialer Freiraum $F1$ mit $(x, y, w_y) = (0, b'.w, Containerbreite)$ generiert. Ist der Container bereits teilweise beladen, wird der noch aktuelle Layer abgeschlossen und b' erzeugt einen neuen Layer, wie in Abb.3.6b mit b' = Box 4. Es wird dabei nur ein neuer Freiraum erzeugt, falls für die erste Box b_{L1}(in Abb.3.6b wäre das Box 1) mit Dimension (l, w, h) des vorherigen Layers gilt : $b_{L1}.w > b'.w$. Der resultierende Freiraum ist dann beschrieben durch $(x_L, b'.w, Containerbreite)$, wobei x_L die bis dato gültige Layergrenze darstellt. Für $b_{L1}.w \leqslant b'.w$ überlappt b' die Box b_{L1} auf der y-Achse, oder ist gleich breit und es wird kein neuer Freiraum erzeugt, sondern stattdessen die bereits bestehenden Freiräume angepasst oder auch gelöscht, wie es in Abb. 3.6b der Fall ist. Durch das Verschieben der Layergrenze vergrößern sich dabei die Freiräume entlang der x-Achse. Dadurch kann auf einfache Weise eine komplette Neuberechnung der Freiräume eingespart werden und es wird durch diese Handhabung das Prinzip der flexiblen Layer umgesetzt.
Anschließend wird die Layergrenze auf $x_L = x_L + b'.l$ aktualisiert und ein Marker für den momentanen Layer $y_{Next} = b'.w$ definiert. Dieser ist für das Platzieren entlang der y-Achse notwendig, da die Boxen dort immer direkt nebeneinander platziert werden sollen. Dieser Marker gibt dann gerade die Position der rechten Außendwand der vorherigen Box und gleichbedeutend dazu die nächste Einfügeposition auf der y-Achse an. Durch das beschriebene Prinzip der Freiräume gibt es immer einen Freiraum dessen y-Wert mit y_{Next} übereinstimmt, solange es möglich ist eine Box entlang der y-Achse zu platzieren.

3.2.4. Füllen eines Layers

Das Befüllen des Layers, bzw. das Einfügen von Boxen in die entstandenen Freiräume, verläuft ausgehend vom Marker y_{Next} entlang der y-Achse. Hierbei wird probiert die nächste Box b' mit Dimension (l, w, h) in einem Freiraum mit Position y_{Next} auf der y-Achse zu platzieren. Ist dies möglich wird die Box dort platziert. Ist zudem die nächste Box in der Beladesequenz identisch zur Momentanen, wird versucht diese ebenfalls in den gleichen Freiraum, auf der Grundfläche über ihr zu platzieren(s. Abb. 3.3). Dies wird solange wiederholt, bis die nächste einzufügende Box nicht gleich der Momentanen ist, oder eben der Freiraum nicht mehr genug Platz bietet, da sonst die Layergrenze überschritten wird. Wie bereits beschrieben können nun nach dem Einfügen der Box(en) in den Freiraum, neue Freiräume entstehen(s. Abb.3.5). Diese werden in die zugehörige Liste eingefügt. Zudem wird der Marker aktualisiert: $y_{Next} = y_{Next} + b'.w$. Damit wird sichergestellt, dass das Füllen des Layers von links nach rechts verläuft. Bevor die nächste Box in einem Freiraum auf der Grundfläche des Containers platziert werden kann, wird aber erst wieder ein Stapel gebildet. Das Platzieren der Boxen entlang der y-Achse geht solange weiter, bis die nächste einzufügende Box in keinen Freiraum mit Position y_{Next} auf der y-Achse passt. Ist dies der Fall, wird wieder versucht einen neuen Layer zu erstellen.

3.2.5. Stapelbildung

Wurde auf der Grundfläche des Containers eine Box, oder mehrere gleiche entlang der x-Achse platziert, wird danach ein Stapeln in die Höhe angestrebt, bevor es mit der Beladung der Grundfläche weitergeht.
Für die Platzierung einer Box auf einer anderen Box, werden indes keine Freiräume verwendet. Hier bildet die Box/bilden die Boxen, auf dem Boden des Containers einen Stapel mit der Grundfläche: $A = Anzahl * Boxlänge * Boxbreite$. Auf dieser Grundfläche können nun weitere Boxen platziert werden. Wie in der Problembeschreibung dargelegt, ist es nur erlaubt Boxen gleichen Typs auf einer Ebene eines Stapels zu platzieren. Zuerst wird ermittelt, wie viele Boxen des momentanen Blocks überhaupt auf den Stapel passen, unter Berücksichtigung aller Einschränkungen. Um überhaupt eine Box darauf platzieren zu können, muss das Gewicht einer Box für die darunterliegenden Boxen tragbar sein. Um dies festzustellen, wird das tragbare Gewicht pro Flächeneinheit betrachtet. Dabei

Algorithmus 3.2 : BeladeLayer - Platziere Box b' in Layer

Eingabe : Liste $FSList$ mit Freiräumen aufsteigend sortiert nach y-Werten;
Marker y_{Next}; Layergrenze x_L auf x-Achse

```
1  foreach Freiräume F ∈ FSList do
2      if F.y == y_Next and F.x + b'.l ⩽ x_L and b'.w ⩽ F.w then
3          einfügen(b', F)
4          while b' == b_Next and einfügbar(b_Next, F) do
5              einfügen(b_Next, F)
6          y_Next += b'.w
7          Füge mögliche neue Freiräume zu FSList hinzu und passe Alte an;
8          initialisiere_stapel()
9          return true
10 return false
```

ist das tragbare Gewicht pro Flächeneinheit immer das minimal tragbare Gewicht pro Flächeneinheit einer Box im Stapel. Nun wird berechnet, wie viele Boxen der Länge und Breite nach auf die Grundfläche des Stapels passen. Danach können die Boxen platziert werden. Dies erfolgt immer erst der Länge nach, d.h. sind fünf Boxen verfügbar und es passen jeweils drei Boxen der Länge, sowie der Breite nach auf den Stapel, werden zuerst drei Boxen der Länge nach positioniert, dann wird die Einfügeposition auf der y-Achse um eine Boxbreite erhöht und dort die verbliebenen zwei Boxen positioniert, wie in gleicher Art und Weise auf Ebene 2 in Abb. 3.7 zu sehen ist. Die platzierten Boxen auf dieser Ebene bilden nun die neue, in den meisten Fällen kleiner werdende Grundfläche des Stapels.

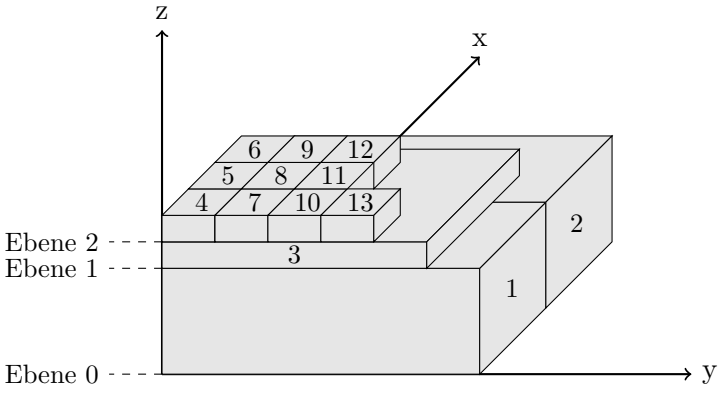


Abbildung 3.7.: Stapel

Im Anschluss daran können die nächsten Boxen darauf platziert werden. Außerdem muss noch eine tragbare Gesamtlast des Stapels mitgeführt werden, um die Gewichtsrestriktionen einzuhalten. Um das klar zu machen, soll folgendes Beispiel dienen: Es sollen drei gleiche Boxen direkt aufeinander platziert werden. Die erste Box kann die Zweite gerade so tragen. Nun soll die Dritte auf der Zweiten platziert werden. Die zweite Box könnte die Dritte nun ebenfalls tragen. Die erste Box würde aber unter dem Gewicht von zwei Boxen einbrechen. Kann keine Box mehr auf einem Stapel platziert werden, gilt dieser als abgeschlossen und wird im weiteren Beladevorgang nicht mehr betrachtet. Als nächstes wird wieder versucht, eine Box entlang der y-Achse auf der Grundfläche des Containers zu platzieren.
Zusammenfassend besteht ein Stapel aus vier Informationen: Stapellänge, Stapelbreite, tragbares Gewicht pro Flächeneinheit und tragbare Gesamtlast.

Algorithmus 3.3 : STAPELN - Auflegen gleicher Boxen auf einer Ebene des Stapels

Voraussetzung : Die aktuelle Box ist auf dem Stapel platzierbar

```
1  anzahl_l = stapellänge / boxlänge
2  anzahl_b = stapelbreite / boxbreite
3  limit = berechnete Gesamtzahl platzierbarer Boxen
4  pos = einfügeposition
5  for i = 0 to anzahl_l and limit > 0 do
6      for j = 0 to anzahl_b and limit > 0 do
7          einfügen(box, einfügeposition)
8          einfügeposition.x += boxlänge
9          limit- -
10     einfügeposition.y += boxbreite
11     einfügeposition.x = pos.x
12 update_stapelparameter()
13 einfügeposition = pos
14 einfügeposition.z += boxhöhe
```

3.3. Lokale Suche

Das Verfahren von Ceschia und Schaerf verwendet eine Spezialform der lokalen Suche, das simulierte Abkühlen, um ein möglichst gutes Ergebnis zu erzielen. Bei dem vorliegenden Problem ist der Zustand einer Instanz gegeben durch die Beladesequenzen der Container. Da der Beladem echanismus deterministisch abläuft, repräsentiert jeder Zustand wiederum eine Lösung. Ein Nachbar des aktuellen Zustandes wird generiert, indem ein Block, oder ein Teil davon, an eine andere Position versetzt wird. Dies kann auch von einer Sequenz in eine andere passieren. Auf die Nachbarschaftsbeziehung wird gleich noch näher eingegangen. Die bereits in 3.1 vorgestellte Kostenfunktion f stellt hier die Zielfunktion dar.

3.3.1. Nachbarschaftsbeziehung

Wie im vorherigen Kapitel bereits erwähnt, müssen für die lokale Suche Nachbarn des aktuellen Zustandes betrachtet werden. Ein Nachbar des aktuellen Zustandes, also der aktuellen Beladesequenzen entsteht, indem ein Block, oder ein Teil davon, aus einer Sequenz entfernt wird und anschließend in eine andere Sequenz, oder an eine andere Position in der gleichen Sequenz, wieder eingefügt wird. Welcher Block ausgewählt und wo er eingefügt wird, entscheidet der Zufall. Zudem ist eine zufällige Änderung der Orientierung möglich. Eine Versetzung besteht also immer aus folgenden Attributen: Ursprungscontainer, Block ID, Anzahl Boxen, Orientierung, Zielcontainer und Zielposition in der Beladesequenz des Zielcontainers. Es würde auch ausreichen auf die Angabe des Ursprungscontainers zu verzichten, wie im Ceschia Paper[1] vorgeschlagen und einfach alle Container nach dem Block mit der gegebenen ID zu durchsuchen. Dies entspreche aber einem deutlich höherem Aufwand, dem einfach durch die Zugabe der Information des Ursprungscontainer entgangen werden kann.

Implementierungsdetails

Ein paar Dinge gilt es bei einer Versetzung zu beachten. Wird der komplette Block entfernt, kann der Block davor und danach verschmolzen werden, falls beide die gleichen Boxen mit derselben Orientierung beinhalten. Wird der ausgewählte Block eingefügt und es befindet sich ein Block mit den gleichen Boxen und derselben Orientierung vor, oder hinter ihm, wird er mit diesem ebenfalls verschmolzen. Bei diesem Verschmelzen zweier Blöcke muss

einfach nur die Anzahl der Boxen eines Blockes, auf die Anzahl der Boxen des anderen Blockes addiert werden. Somit reicht es aus, wenn ein Block fortbesteht.

Um sicher zu stellen, dass ein Fortschritt durch einen Nachbar entstehen kann, werden folgende, nutzlose Veränderungen verboten:

- Das Versetzen eines Blocks, welcher zuvor nicht verladen werden konnte, an eine Position in einer Sequenz, an der er wieder nicht verladen werden kann. Die Lösung würde sich nicht ändern.
- Das Versetzen eines kompletten Blocks direkt hinter dem folgenden Block. Dies wird bereits dadurch erreicht, indem der folgende Block eine Position nach vorne versetzt wird.
- Wenn durch das Versetzen der gleiche Zustand entsteht wie zuvor. Zum Beispiel indem ein Teil des Blocks direkt hinter dem Block selbst wieder eingefügt wird, oder der Zielcontainer, die Zielpostion und Zielorientierung die gleiche sind wie zuvor.

Zusätzlich muss nach dem Versetzten die Multi-Drop Restriktion erfüllt sein, um keine ungültige Lösung zu produzieren. Dies wird sichergestellt, indem eine Box mit Zieldestination d_{Ziel} an eine Position in einer Beladesequenz eingefügt wird, an der alle Boxen welche zuvor an der Reihe sind, eine Zieldestination $d'_{Ziel} \leq d_{Ziel}$ aufweisen und alle Boxen welche danach an der Reihe sind, eine Zieldestination $d'_{Ziel} \geq d_{Ziel}$ aufweisen. Außerdem können Container für Boxen bestimmter Zieldestinationen *reserviert* sein. Dies tritt ein, wenn ein Container voll beladen ist, d.h. der linear freie Raum(s. Kostenkomponente c_3 in Abschnitt 3.1) auf der x-Achse ist gleich Null. In der Folge ist der Container dann für die Zieldestinationen der geladenen Boxen reserviert. Daraus folgt, dass Boxen nur in die Beladesequenzen von Containern versetzt werden dürfen, falls ihre Zieldestination in der Reservationsliste des Containers ist, oder der Container nicht reserviert ist.

3.3.2. Simuliertes Abkühlen

Algorithmus 3.4 : SimuliertesAbkühlen

```
1  c_min = f(initial_solution)
2  best_solution = initial_solution
3  T = T_0
4  while T > T_min do
5      for i = 1 to σ_N do
6          neighbor = generate_neighbor(best_solution)
7          solution = calculate_solution(neighbor)
8          c_N = f(solution)
9          if c_N < c_min or get_true(e^(-c_N/T)) then
10             best_solution = solution
11             c_min = c_N
12         T = T * τ
13 return best_solution
```

Von der Ceschia Gruppe wurde das simulierte Abkühlen wie folgt umgesetzt: Es wird ein Startwert T_0 mittels eines Faktors $0 < \tau < 1$ abgesenkt, bis der Grenzwert T_{min} erreicht ist, welcher das Abbruchkriterium darstellt. In jedem Schritt findet eine feste Anzahl von σ_N Iterationen statt, in denen jeweils ein Nachbar ermittelt wird. Die Wahrscheinlichkeit p mit der ein Nachbar akzeptiert wird, der eine schlechtere Lösung liefert, nimmt hierbei exponentiell mit der Zeit ab. Diese Wahrscheinlichkeit berechnet sich konkret mit $p = e^{-c_N/T}$,

wobei $c_N > 0$ die Kosten des Nachbarn sind. Diese Umsetzung ist in Algorithmus 3.4 dargestellt. In Zeile 7 reicht es dabei aus, nur die Sequenzen welche von der Versetzung betroffen sind, mittels LOAD(C_i, seq_i) neu in den Container zu laden.

3.4. Berechnung der Gesamtlösung

Nun ist es an der Zeit zu erläutern, wie die Gesamtlösung einer Probleminstanz berechnet wird.

Zu aller erst findet eine Vorberechnung statt, in der überschüssig vorhandene Container entfernt werden, um nicht unnötig viele davon zu nutzen. Falls also ausreichend Container vorhanden sind, werden nur die ersten k Container verwendet, sodass deren summiertes Volumen größer gleich dem 1,5-fachen des Gesamtvolumens aller zu verladenden Boxen entspricht. Für jeden Container wird eine leere Beladesequenz erstellt. Die restlichen Container werden verworfen.
Im Anschluss werden die Boxen nach ihrer Zieldestination d_i mit $i \in \{1, ..., n\}$ geordnet, denn die Gesamtlösung wird schrittweise aus n Teillösungen berechnet. Dabei werden für eine Teillösung i mit $i = 1, ..., n$ nur Boxen verladen, welche einer Zieldestination d_j mit $j \leqslant i$ angehören. Die Teillösung i wird aus Teillösung $i-1$ berechnet, indem zuerst alle Boxen mit Zieldestination d_i zufällig an das Ende einer Eingabesequenz eines nicht reservierten Containers angehängt werden. Danach wird die Teillösung i durch SIMULIERTESABKÜHLEN gelöst, wie im vorhergehenden Abschnitt beschrieben. Für die Teillösung 0 werden die Boxen zufällig auf die bis dato leeren Beladesequenzen der Container verteilt. Teillösung n stellt dann die gesuchte Gesamtlösung dar.

Algorithmus 3.5 : CESCHIA_LOCALSEARCH(M_C, M_B)

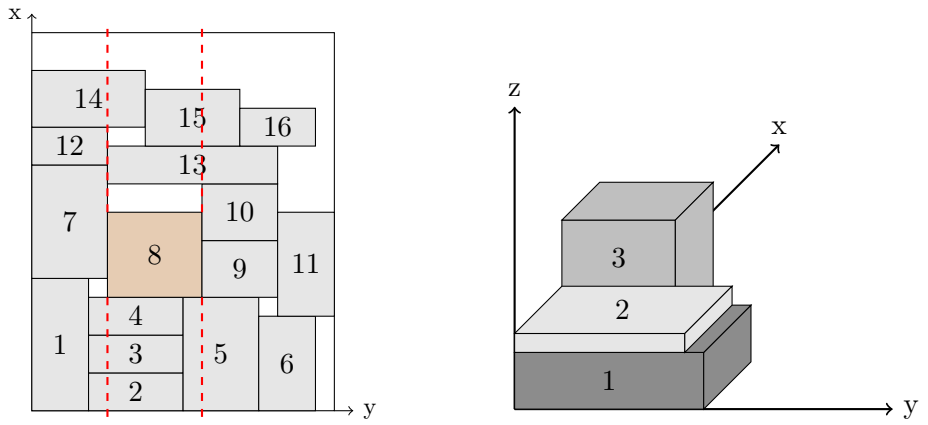
Eingabe : M_C = verfügbare Container; M_B = Menge an Boxen

```
1 container = vorberechnung(M_B, M_C)
2 initialisiere_sequenzen()
3 sortiere_boxen()                          // aufsteigend nach Zieldestinationen
4 foreach Zieldestination d_i do
5 |   Füge alle Boxen mit Ziel d_i an das Ende einer zufälligen Beladesequenz an
6 |   L = SimuliertesAbkühlen()
7 return L
```

3.5. Prüfen der Multi-Drop Eigenschaft

Der Loader garantiert das bei einer nach Zieldestinationen geordneten Beladesequenz die Multi-Drop Restriktion erfüllt ist, da die Boxen einfach in umgekehrter Reihenfolge entnommen werden können. Für das im folgenden Kapitel vorgestellte Verfahren und die Komponenten, welche daraus vorgestellt werden, wird allerdings kein Multi-Drop berücksichtigt. Es ist also notwendig prüfen zu können, ob durch eine Veränderung der Beladungsanordnung die Multi-Drop Restriktion weiterhin erfüllt ist. Zudem ist es nach der Berechnung einer Lösung ebenfalls notwendig, diese zu validieren. In diesem Abschnitt soll nun eine selbst entwickelte Funktion vorgestellt werden, welche diese Aufgabe erledigt. Für eine komplette Beladung eines Containers mit n Boxen, kann damit in $\mathcal{O}(n^2)$ geprüft werden, ob sie die Multi-Drop Eigenschaft erfüllt. Für eine einzige Box, die an einer beliebigen, gültigen Position in eine Ladung eingefügt wird, welche die Multi-Drop Eigenschaft erfüllt, ist dies in linearer Laufzeit möglich.

Zunächst gilt für eine Menge an Zieldestinationen $D = [d_1, ..., d_n]$, dass ein Ziel $d_i \in D$ vor einem Ziel $d_j \in D$ angefahren wird, falls $i < j$. Gilt $i > j$ wird folgerichtig d_i nach d_j



(a) Für Box 8 soll die Multi-Drop Eigenschaft geprüft werden. Dazu müssen alle Boxen betrachtet werden, welche innerhalb des roten Schlauches liegen.

(b) Verletzung der Multi-Drop Eigenschaft. Höherer Grauwert bedeutet hier eine spätere Zieldestination. Box 2 kann nicht vor Box 3 entladen werden.

Abbildung 3.8.: Multi-Drop Restriktion

angefahren. Um nun eine komplette Beladung eines Containers mit Boxen zu überprüfen, denen alle eine Zieldestination $d_i \in D$ zugeordnet ist, wird nacheinander jede Box betrachtet. Es wird dabei wieder zwischen Boxen auf der Grundfläche des Containers und Boxen auf einem Stapel unterschieden.

Für eine Box b' mit Dimension (l, w, h) an Position $(x, y, 0)$ und Zieldestination $d_{k1} \in D$ wird dafür ein imaginärer Schlauch erzeugt, der durch die beiden Außenwände $\gamma_1 = y$ und $\gamma_2 = y + w$ induziert wird. Die Zieldestination aller Boxen, die ganz oder teilweise innerhalb des Schlauches liegen, werden mit der Zieldestination von b' verglichen. Die Idee dahinter soll anhand von Abbildung 3.8a erklärt werden, in der Box 8 die betrachtete Box b' darstellt. Werden alle Boxen, die auf der Grundfläche des Containers über Box 8 innerhalb des Schlauches liegen, also Box 13, 14 und 15 vor, oder zur gleichen Zeit wie Box 8 entladen, kann Box 8 ohne Behinderung entladen werden. Müsste aber zum Beispiel Box 13 erst nach Box 8 entladen werden, würde Box 13 im Weg stehen und die Multi-Drop Restriktion wäre verletzt. Für alle Boxen, die auf der Grundfläche unterhalb von Box 8 innerhalb des Schlauches liegen, also Box 2 - 5, muss Box 8 zur gleichen Zeit, oder zuvor entladen werden, sonst würde sie im Weg stehen um diese Boxen zu entladen. Dies würde ebenfalls zu einer Verletzung der Multi-Drop Restriktion führen. Diese Vorgehensweise wird für jede Box auf der Grundfläche des Containers durchgeführt. Boxen welche bereits erfolgreich getestet wurden, spielen dann für die anderen, die es noch zu betrachteten gilt, keine Rolle mehr.
Formal muss also für eine Box b_2 mit Zieldestination $d_{k2} \in D$, welche auf der Grundfläche des Containers über b' innerhalb des Schlauches liegt, $k2 \leq k1$ gelten. Analog dazu, müsste $k2 \geq k1$ erfüllt sein, falls b_2 auf der Grundfläche des Containers unterhalb von b' liegt. Für den ersten Fall wird die Zieldestination d_Z der Box, welche auf der $x - Achse$ den geringsten Abstand zu b' aufweist gespeichert, um im Anschluss noch die Kontrolle des Stapels durchführen zu können.

Für Stapel gilt, dass diese immer von oben nach unten entladen werden. Also muss für die Zieldestination d_{k1} einer Box, die auf einer anderen Box mit Zieldestination d_{k2} aufliegt, gelten: $k1 \leq k2$. Da ein Stapel wie in Algorithmus 3.3 beschrieben aufgebaut wird, reicht es für eine Box auf dem Stapel aus, nur die in der Beladesequenz vorherige Box zu betrachten. Diese muss eine gleiche, oder spätere Zieldestination aufweisen. Für die letzte auf dem

Stapel platzierte Box, wird abschließend noch geprüft, ob deren Zieldestination kleiner als d_Z ist. Damit kann eine ungültige Beladung wie in Abb. 3.8b dargestellt, erkannt werden.

Algorithmus 3.6 : TestMultiDrop(seq)

Eingabe : seq = berechnete Beladung eines Containers

```
1  b_0 = seq.get_front()
2  seq.pop_front()
3  while seq nicht leer do
4      if b_0.pos.z == 0 then
5          d_Z = MAX_INT
6          foreach b ∈ seq do
7              Falls b in Schlauch von b_0 liegt, vergleiche Zieldestinationen
8              if Multi-Drop verletzt then
9                  return false
10             else
11                 Aktualisiere gegebenenfalls d_Z
12     else
13         if b_0.d < b_vor.d then
14             return false
15         if b_0 ist letzte Box auf Stapel then
16             if b_0.d > d_Z then
17                 return false
18     b_vor = b_0
19     b_0 = seq.get_front()
20     seq.pop_front()
21 return true
```

4. Freiraumdefragmentierung

4.1. Problembeschreibung

Die Defragmentierungsheuristik von Zhang et al.[2] befasst sich mit dem dreidimensionalen Bin-Packing Problem. Dabei soll eine Menge von Boxen, komplett in eine unbegrenzte Anzahl identischer Behälter gepackt werden. Ziel ist es, die Boxen möglichst dicht zu packen, um eine minimale Anzahl an Behältern verwenden zu müssen. Jede Box und jeder Behälter wird komplett beschrieben durch die zugehörige Dimension aus $länge \times breite \times höhe$. Boxen können nicht rotiert, aber innerhalb eines Behälters beliebig im Raum platziert werden, solange sich keine zwei Boxen überschneiden. Insgesamt ist das Platzieren der Boxen nur folgenden Restriktionen unterworfen:

- Jede Box muss sich komplett innerhalb eines Behälters befinden.
- Die Boxen müssen entlang der Achsen ausgerichtet sein, d.h. die Länge verläuft parallel zur x-Achse, etc.
- Es dürfen sich keine Boxen innerhalb eines Behälters überschneiden.

Um eine einheitliche Wortwahl beizubehalten, werden bei diesem Problem in der Folge die Behälter ebenfalls Container genannt.

Unterschiede zur Heuristik von Ceschia und Schaerf

Da einzelne Komponenten dieses Verfahrens verwendet werden sollen, um die Heuristik aus Kapitel 3 aufzuwerten, müssen diese bei der Umsetzung auch an die dortige Problembeschreibung angepasst werden. Deshalb werden in Tabelle 4.1 alle elementaren Unterschiede übersichtlich aufgeführt.

Ceschia und Schaerf[1]	**Zhang et al.**[2]
Boxen rotierbar	Boxen fest im Raum ausgerichtet
Stapeln unterliegt Einschränkungen	Boxen sind beliebig im Raum platzierbar
Multi-Drop	Boxen sind keine Zieldestinationen zugeordnet
Gewichtsrestriktionen	Gewichte von Boxen werden nicht beachtet
Optimierung nach Kosten	Optimierung nach Volumenauslastung
Heterogene Container	Homogene Container

Tabelle 4.1.: Unterschiede beider Heuristiken

Finden in der Folge aufgrund von Unterschieden Anpassungen statt, um eine gültige Umsetzung zu garantieren, wird dies immer erwähnt.

Vorbereitend soll schon einmal die komplette Prozedur für die Freiraumdefragmentierung in Algorithmus FreiraumDefrag vorgestellt werden:

Algorithmus 4.1 : FreiraumDefrag(M_C, M_B, BS)

Eingabe : Menge Container M_C; Beladungen M_B; Einzufügende Boxen BS

```
1  Berechne für alle Beladungen B ∈ M_B die Extrempunkte und die Positionen der Boxen
   nach einem Push-Out
2  foreach b ∈ BS do
3      placed = Extrempunkt-Einfügen(M_C, M_B, b)
4      if placed == true then
5          continue und versuche nächste Box zu platzieren
6      else
7          placed = Ausdehnen-Ersetzen(M_C, M_B, b)
8          if placed == false then
9              Füge b am Ursprung eines neuen Containers C_Neu ein
10             Berechne für C_Neu die Extrempunkte
```

Die einzufügenden Boxen werden dabei zuerst versucht an einer freien Einfügeposition, einem sogenannten *Extrempunkt* im Container zu platzieren (Zeile 3). Dabei können auch andere Boxen verschoben werden, welche potentiell im Weg stehen können. Kann die Box b so aber in keinem Container positioniert werden, wird ausprobiert eine kleinere Box b' in einem der Container mit b auszutauschen (Zeile 7). Schlägt auch das fehl, wird b in einem neuen, leeren Container platziert.

4.2. Extrempunkte

Die Grundlage für die Defragmentierungsheuristik nach Zhang et al.[2] liegt in den sogenannten Extrempunkten, welche in der wissenschaftlichen Arbeit von Crainic et al.[4] ausführlich vorgestellt werden. Diese Extrempunkte stellen einfach potentielle neue Einfügepositionen dar.

Bei einem leeren Container existiert genau ein Extrempunkt, und zwar der Ursprung. Durch einfügen einer Box mit Dimension (l, w, h) an einem Extrempunkt (x, y, z) entstehen sechs neue Extrempunkte (s. Abb. 4.1). Diese erhält man durch Projektion von:

- Eckpunkt $(x + l, y, z)$ entlang y- und z-Achse
- Eckpunkt $(x, y + w, z)$ entlang x- und z-Achse
- Eckpunkt $(x, y, z + h)$ entlang x- und y-Achse

Der jeweilige Eckpunkt wird dabei auf die nächst gelegene Box, oder die Containerwand projiziert.

4.2.1. Extrempunkte für das Ceschia Verfahren

Soll das Prinzip der Extrempunkte nun für das Ceschia Verfahren angewandt werden, müssen dessen Restriktionen beachtet werden. Vorneweg werden die Extrempunkte anhand ihrer Position z auf der z-Achse in zwei Typen eingeteilt:

1. Extrempunkte mit $z = 0$. Hier reicht es aus nur den Extrempunkt selbst zu speichern.

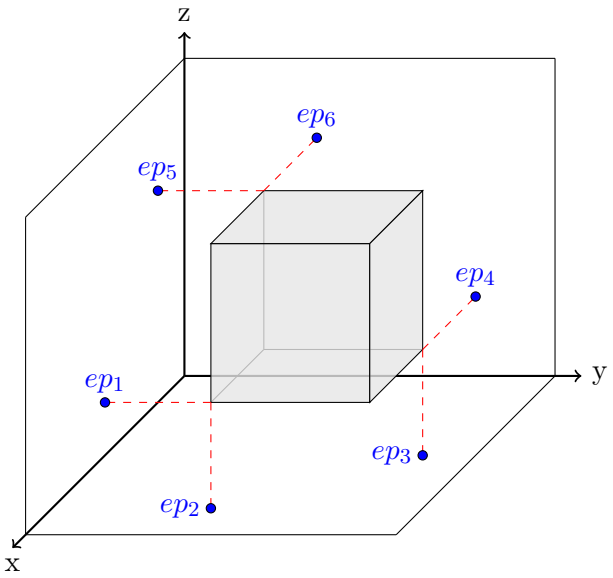


Abbildung 4.1.: Extrempunkte $ep_1 - ep_6$

2. Extrempunkte mit $z > 0$. In diesem Fall müssen zusätzliche Informationen, die für das Platzieren auf einem Stapel nötig sind mit gespeichert werden: wie lang, breit und schwer darf eine Box sein, die dort platziert werden soll und wie viel Gewicht pro Fläche kann der Stapel an dieser Position tragen. Vgl. dazu Kapitel 3.2.5. Zudem muss erkenntlich sein, welche Zieldestination die Box haben darf, um das Multi-Drop einzuhalten.

Welche und wie viele Extrempunkte entstehen können, hängt dann wieder davon ab wo die Box eingefügt wurde, die diese erzeugt. Hierbei wird nun in zwei unterschiedliche Fälle aufgeteilt.

1. Die neue Box wird auf dem Boden des Containers eingefügt

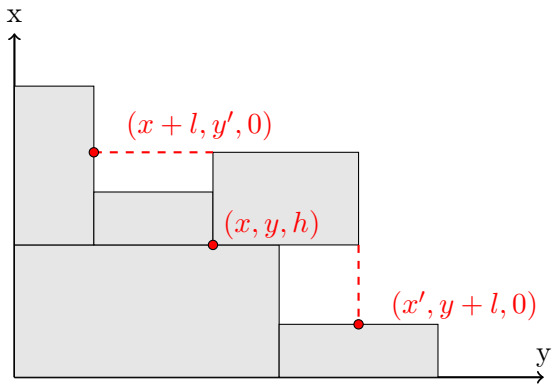


Abbildung 4.2.: Extrempunkte für auf Boden eingefügte Box aus Vogelperspektive

Unter Betrachtung von Abb. 4.1 würde durch ein Platzieren auf der Grundfläche die beiden Extrempunkte ep_2 und ep_3 gerade ihren Eckpunkten entsprechen, von denen sie ausgehen. Da sie dann keinen Mehrwert ggü. den Extrempunkten ep_1 und ep_4 bieten, entfallen beide. Da wie bereits erwähnt Boxen, welche nicht auf der Grundfläche platziert werden, in komplettem Umfang auf einer, oder mehreren anderen Boxen aufliegen müssen, sind die Extrempunkte ep_5 und ep_6 in den allermeisten Fällen keine gültigen Einfügepositionen. Deshalb wird der Eckpunkt, aus dem diese beiden Extrempunkte entstehen, nicht auf

andere Boxen oder die Containerwand projiziert, sondern bildet selbst einen Extrempunkt. Somit ergeben sich für diesen Fall folgende drei Extrempunkte:

- Eckpunkt (x, y, h)
- Punkt $(x + l, y', 0)$ durch Projektion von Eckpunkt $(x + l, y, 0)$ entlang der y-Achse
- Punkt $(x', y + w, 0)$ durch Projektion von Eckpunkt $(x, y + w, 0)$ entlang der x-Achse

2. Die neue Box wird auf einem Stapel platziert

Für das Platzieren auf einem Stapel, gilt noch eine weitere Einschränkung. Es sollen sich nur gleiche Boxen auf einer Ebene eines Stapels befinden. Aus diesem Grund wird für eine Box, die auf einem Stapel platziert wird, nur der Eckpunkt $(x, y, z + h)$ als Extrempunkt verwendet. Das reicht aus, da die Annahme getroffen wird, dass keine weiteren Boxen des gleichen Typs mehr auf dieser Ebene platziert werden können, da dass Ceschia Verfahren sie dann dort schon platziert hätte. Des weiteren wird nur der Extrempunkt, den die erste Box auf dieser Ebene liefert genutzt, da alle anderen Boxen der Ebene schlechtere Extrempunkte liefern würden, hinsichtlich der Frage wie lange und breit eine dort zu platzierende Box sein dürfte.

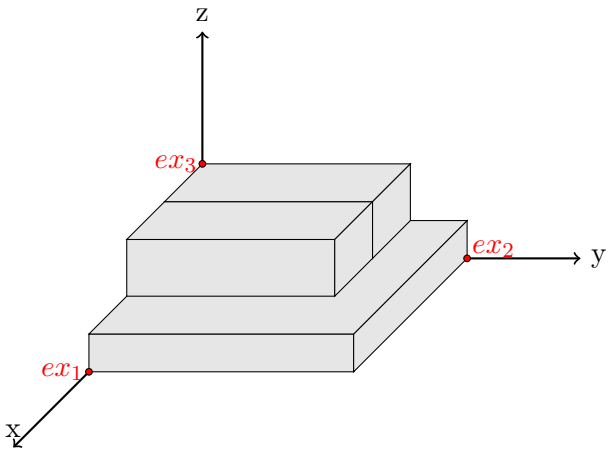


Abbildung 4.3.: Extrempunkte eines Stapels

Abgesehen von diesen beiden unterschiedlichen Fällen, können mehrere gleiche Boxen, welche direkt nebeneinander platziert sind (s. Abb. 3.3 und 3.7), wie eine einzelne, längere Box behandelt werden. Somit müssen dann nur für eine Box Extrempunkte berechnet werden.

4.2.2. Implementierung

Bei der Implementierung soll im Vorhinein wieder zwischen der Berechnung von Extrempunkten mit $z = 0$ und $z > 0$ unterschieden werden.

Für die allgemeine Berechnung der Extrempunkte aus Abb. 4.1 liefert Crainic et al.[4] einen Pseudocodeabschnitt. Dieser wird als Grundlage für die Berechnung der Extrempunkte mit $z = 0$ herangezogen. Dabei werden jedes Mal, wenn eine neue Box im Container platziert wird, die Extrempunkte aktualisiert. Ist nun also eine neue Box mit Dimension (l, w, h) an einem Extrempunkt (x, y, z) positioniert worden, wird geprüft, ob die Eckpunkte $(x + l, y, 0)$ und $(x, y + w, 0)$ auf eine der bereits im Container befindlichen Boxen projiziert werden können. Dafür muss jede Box einmal betrachtet werden. Die Projektionen mit dem geringsten Abstand zum Eckpunkt selbst, sind die neuen Extrempunkte. Dabei kann natürlich der Eckpunkt selbst ein Extrempunkt sein, falls eine andere Box direkt anliegt.

Diese Extrempunkte werden alle in einer Liste gespeichert, die absteigend nach x und y Positionen geordnet ist. Ein neuer Extrempunkt wird dabei an die entsprechende Position in die Liste eingefügt.

Für den neuen Extrempunkt $(x, y, z + h)$ muss der komplette Stapel betrachtet werden, an dessen Spitze er liegt. Dies liefert dann - bis auf die Zieldestination, die die Box maximal haben darf - die zusätzlich nötigen Informationen um eine neue Box auf einem Stapel platzieren zu können. Bis auf die Festlegung der minimalen und maximalen Zieldestination kann dies dem Kapitel 3.2.5 entnommen werden. Die minimale Zieldestination entspricht der Zieldestination, welche die Box hat, die diesen Extrempunkt induziert hat, sprich die Box genau darunter. Die Berechnung der maximalen Zieldestination erfolgt wie bei dem Test zur Einhaltung der Multi-Drop Restriktion für eine Box auf dem Stapel. Ein Objekt für solch einen Extrempunkt besteht also aus folgenden Informationen:

- Position (x, y, z)
- Stapellänge und Stapelbreite auf Höhe z
- Gesamtgewicht, welches der Stapel noch tragen kann
- Gewicht pro Fläche, welches der Stapel tragen kann
- Minimale und maximale Zieldestination einer einzufügenden Box

Diese Extrempunkte werden separiert von den anderen in einer eigenen Liste abgespeichert, absteigend geordnet nach z, x und y Position. Der Extrempunkt, an dem die Box eingefügt wurde, muss dann natürlich wieder aus der Liste gelöscht werden.

Auf die gleiche Weise können auch die Extrempunkte einer bereits bestehenden Beladung eines Container konstruiert werden. Die Boxen werden einfach entsprechend ihrer Einfügereihenfolge neu in den Container geladen und jeweils wie eben besprochen die Extrempunkte aktualisiert.

4.3. Verschieben der Boxen

Die berechneten Extrempunkte geben an wo neue Boxen platziert werden können, die Information wie groß diese Boxen sein dürfen fehlt allerdings noch. Wünschenswert ist es natürlich, wenn möglichst viel Platz an einem Extrempunkt vorhanden ist, um auch große Boxen platzieren zu können. Teilweise kann der durch die aktuelle Beladung induzierte freie Raum an diesem Extrempunkt bereits ausreichen, um die neu einzufügende Box zu platzieren. Ist dies aber nicht der Fall, soll mehr Freiraum generiert werden, indem andere Boxen möglichst weit beiseite geschoben werden. Dieses Vorgehen bildet die Basis der Raumdefragmentierungsheuristik, die die Gruppe um Zhang in ihrer Arbeit vorstellt. Das Wegschieben der Boxen erfolgt dabei immer entlang der Achsen, weg vom Ursprung und wird in der Folge PushOut genannt.

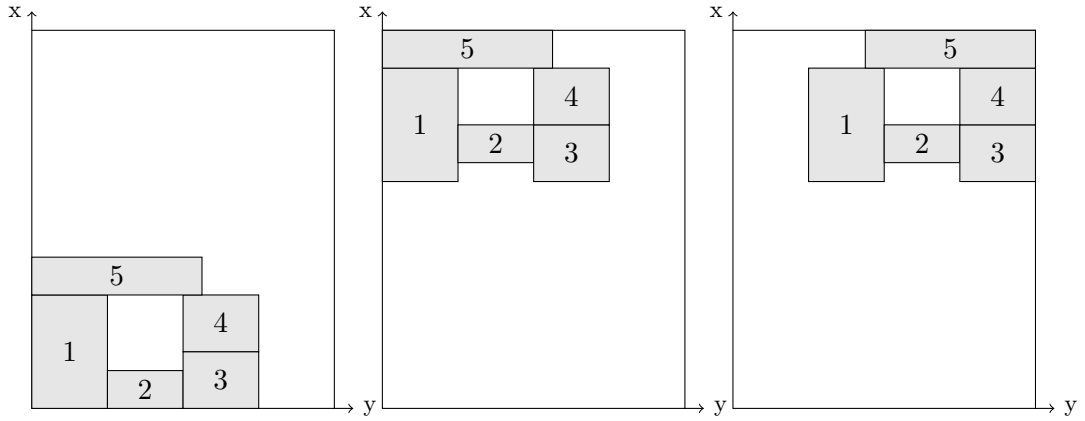
Um effizient zu bestimmen, wie weit alle Boxen entlang einer Achse verschoben werden können - natürlich immer unter Berücksichtigung aller anderen Boxen - wird im Paper[2] Algorithmus 4.2 vorgestellt, der dies in $\mathcal{O}(nlogn)$ berechnet. Der Algorithmus sieht dabei für jede Achse gleich aus, für die x-Achse wie folgt: Zuerst werden alle linken Endpunkte $\underline{x}_i$ und rechten Endpunkte $\overline{x}_i$ der Boxen auf der x-Achse absteigend sortiert, bei Gleichheit erhält ein linker Endpunkt Vorrang. Danach wird ein Marker x_0 mit der Länge des Container initialisiert. Dieser soll angeben, bis zu welchem Punkt auf der x-Achse die Box verschoben werden kann. Die Berechnung ahmt nun mittels des Markers x_0 eine Sweep Line nach, welche vertikal zur x-Achse, von rechts nach links läuft. Trifft sie auf einen rechten Endpunkt, ergibt sich die Distanz $\triangle X_i$, um die die Box i verschoben werden kann, aus der Differenz von x_0 und der Position des Endpunktes auf der x-Achse. Trifft sie auf einen linken Endpunkt,

wird der Marker geupdated: $x_0 = min\{x_0, \triangle X_i + \underline{x}_i\}$. Aus den berechneten $\triangle X_i$ lässt sich dann einfach für jede Box die Position $\overline{x}_i^r$ des rechten Endpunktes nach PushOut bestimmen.

Algorithmus 4.2 : Berechne $\overline{x}_i^r$ für jede Box i im Behälter b

```
1  P = Liste mit Endpunkten aller i ∈ b
2  Sortiere P absteigend, bei Gleichheit hat linker Endpunkt Vorrang
3  x_0 = behälterlänge
4  forall the x ∈ P do
5      if x ist rechter Endpunkt x̄_i für ein i then
6          △X_i = x_0 − x̄_i
7      if x ist linker Endpunkt x_i für ein i then
8          x_0 = min(x_0, △X_i + x_i)
9  forall the i ∈ b do
10     x̄_i^r = △X_i + x̄_i
```

Die Berechnung für jede Achse kann dabei in beliebiger Reihenfolge durchgeführt werden, die resultierenden Positionen sind immer die selben. Diese effiziente Methode hat allerdings den Nachtteil, dass die berechnete Distanz nicht immer exakt so groß, wie die maximalen Distanz ist, sondern ab und zu etwas kleiner, siehe dazu Abb. 4.4, Box 2. Aus Effizienzgründen wurde dies allerdings von den Autoren in Kauf genommen.

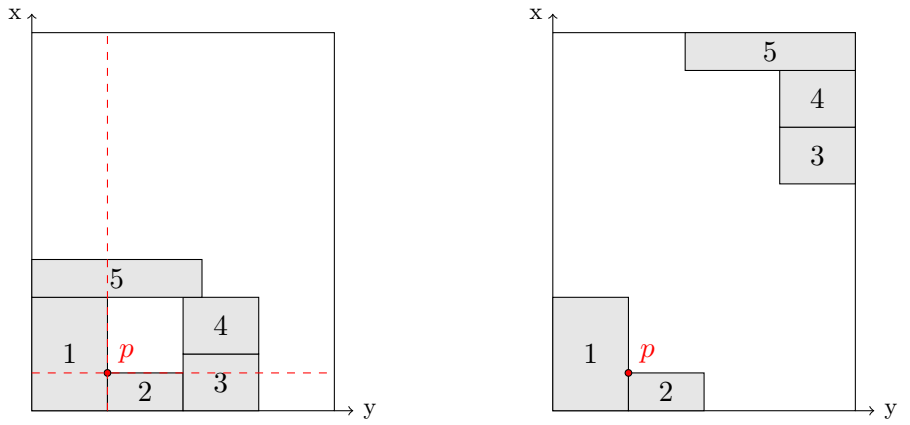


(a) Ausgangszustand (b) PushOut$_X$ (c) PushOut$_Y$

Abbildung 4.4.: PushOut im Zweidimensionalen

Soll nun eine Box an einem Extrempunkt eingefügt werden, müssen in den allermeisten Fällen gar nicht alle Boxen verschoben werden. Betrachten wir hierzu Abb. 4.5 fällt auf, dass die Boxen 1 und 2 nicht bewegt werden müssen, da sie keinen Einfluss auf eine Vergrößerung des Freiraumes über und rechts des Extrempunktes p haben. Sie können also an ihrer Position verbleiben. Die Operation PushOut(p) verschiebt dann nur Boxen, für die es auch nötig sein kann sie zu verschieben.

Die vollständige Vorgehensweise sieht nun wie folgt aus: Zuerst werden alle Extrempunkte berechnet und dann die Positionen aller Boxen nach einem PushOut. Im Anschluss wird über die Extrempunkte iteriert. Nun wird der aktuelle Extrempunkt ep und die zu platzierende Box b_{neu} betrachtet. Jetzt soll festgestellt werden, ob b_{neu} an ep nach der Operation PushOut(ep) platziert werden kann. Dafür werden die resultierenden Positionen der durch PushOut(ep) betroffenen Boxen begutachtet und geprüft, ob eine dieser Boxen b_{neu} schneiden würde, wenn b_{neu} an ep positioniert wird. Ist dies nicht der Fall, wird



(a) Ausgangszustand (b) nach $\textsc{PushOut}_X(p)$ und $\textsc{PushOut}_Y(p)$

Abbildung 4.5.: PushOut an Punkt p

PushOut(ep) durchgeführt und b_{neu} tatsächlich an ep positioniert. Danach müssen die verschobenen Boxen noch normalisiert und Extrempunkte, sowie die Position aller Boxen nach einem PushOut, aktualisiert werden. Falls die Box jedoch nicht positioniert werden kann, wird geprüft ob dies am nächsten Extrempunkt möglich ist. Algorithmus 4.3 zeigt noch einmal übersichtlich, wie versucht wird eine Box an einem Extrempunkt einzufügen.

Algorithmus 4.3 : Extrempunkt-Einfügen(M_C, M_B, b)

Eingabe : Menge Container M_C; Beladungen M_B; Einzufügende Box b

```
1  foreach C ∈ M_C do
2      foreach Extrempunkte ep in C do
3          if b kann an ep nach PushOut(ep, b) platziert werden then
4              PushOut(ep, b)
5              einfügen(b, ep)
6              Normalisieren(C)
7              Aktualisiere Extrempunkte in C
8              Berechne die Positionen nach einem Push-Out aller Boxen in C neu
9              return true
10 return false
```

4.3.1. Implementierung

Für die Implementierung von Operation PushOut für eine Probleminstanz der Ceschia Gruppe, muss die ursprünglichen Funktionalität aus dem vorherigen Abschnitt angepasst werden.

PushOut wird dabei nur für Extrempunkte auf der Grundfläche des Container ausgeführt. Extrempunkte mit Position $z > 0$ auf der z-Achse liegen ja immer auf der Spitze des Stapels und da sie komplett auf Boxen der darunter liegenden Ebene aufliegen müssen, ist kein PushOut nötig. Die Information wie groß die Box an solch einem Extrempunkt sein darf ist auch bereits berechnet, siehe dazu Kapitel 4.2. Für Extrempunkte mit Position $z = 0$ auf der z-Achse wird ein PushOut auf die x- und y-Achse beschränkt. Dabei werden nur die Boxen, welche auf der Grundfläche des Containers liegen verschoben und für Stapel alle aufliegenden Boxen entsprechend mitgezogen. Somit sind alle Stapel auch nach einem PushOut noch in gleicher Form vorhanden.

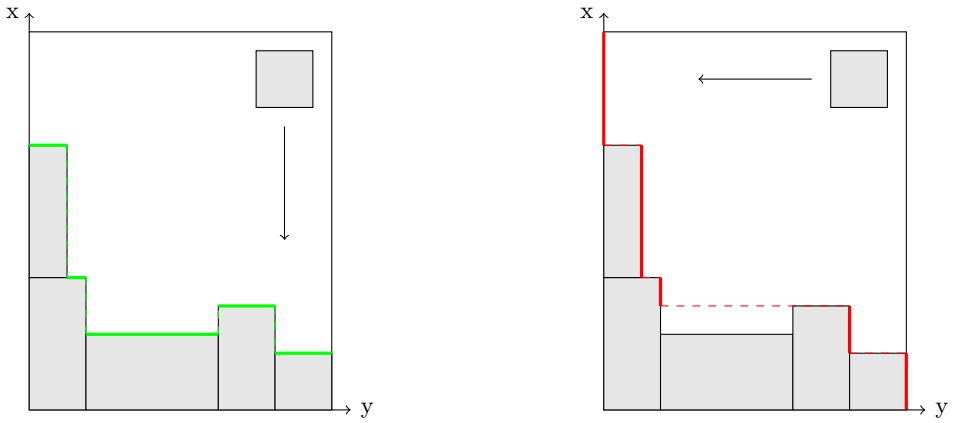
Folglich können die Positionen nach einem PushOut auf x- und y-Achse analog zu Algorithmus 4.3 berechnet werden. Diese werden in einem Vektor abgespeichert und können abgefragt werden. Nach einem durchgeführten PushOut(p) und anschließender Normalisierung, müssen die Boxen dann natürlich noch auf die Einhaltung der Multi-Drop Restriktion geprüft werden, da sich deren Anordnung verändert hat. Um möglichst wenig Veränderungen in der Anordnung der Boxen zu schaffen, wird kein PushOut(p) durchgeführt, falls die Box auch ohne diesen platziert werden könnte.

4.4. Normalisierung

Nach einem PushOut sind einige Boxen verschoben und liegen nicht mehr an der restlichen Ladung an. Um die Beladung wieder zu komprimieren, wird eine Normalisierungsfunktion benötigt. Diese verschiebt die Boxen so weit es geht abwechselnd entlang der x-Achse, dann der y-Achse und zuletzt der z-Achse in Richtung Ursprung. Dies wird solange wiederholt, bis keine Box entlang einer Achse weiter normalisiert werden kann.

4.4.1. Implementierung

Die Gruppe um Zhang et al. geht nur so weit auf diese Prozedur ein, wie im vorherigen Abschnitt beschrieben. Deshalb wurde selbst eine Methodik zur Umsetzung der erforderlichen Funktionalität entwickelt.



(a) Entlang x-Achse (b) Entlang y-Achse

Abbildung 4.6.: Normalisierung. Die durchgezogenen farbigen Linien geben an, wie weit eine Box entlang der Achse normalisiert werden kann.

Da in unserem Fall ein PushOut nur entlang der x- und y-Achse erfolgt, wird auch nur eine Normalisierung entlang dieser beiden Achsen benötigt. Dabei werden nur Boxen normalisiert, welche zuvor auch durch einen PushOut verschoben wurden. Um eine Normalisierung durchführen zu können, muss bekannt sein, wie weit man eine Box entlang der jeweiligen Achse zum Ursprung verschieben kann, ohne in Kollision mit einem Hindernis zu treten. Ein Hindernis ist hierbei entweder eine andere Box, oder die Containerwand. Deshalb soll eine Hindernis- bzw. Oberflächenstruktur für die platzierten Boxen berechnet werden, wie sie in grün und rot in Abb. 4.6 zu sehen ist.

Für das Normalisieren entlang der x-Achse wird das Problem gelöst, indem für jeden Punkt auf der y-Achse ein Wert festgehalten wird, der angibt an welcher Position auf der x-Achse das erste Hindernis auftaucht, d.h. die x-Achse des Containers wird diskretisiert. Zu diesem Zweck wird ein Vektor V_y verwendet, der gerade so viele Einträge besitzt, wie der Container breit ist. Zuerst werden alle Werte von V_y mit 0 initialisiert, was die Containerwand als

Hindernis darstellt. Danach werden nacheinander die bereits fest platzierten Boxen darauf inspiziert, wie weit sie in die Tiefe entlang der x-Achse ragen. Für eine Box b' mit Dimension (l, w, h) an Position (x, y, z) wird für alle i mit $b'.pos.y \leq i \leq b'.pos.y + b'.w$ geprüft, ob $b'.pos.x + b'.l > V_y[i]$. Ist dies der Fall, wir jedes Mal der Vektor mit $V_y[i] = b'.pos.x + b'.l$ aktualisiert. Danach ist die Vorbereitung abgeschlossen. In Abbildung 4.6 (a) kennzeichnen die durchgezogenen grünen Linien die Werte des Vektors für die y-Achse, nachdem alle nicht verschobenen Boxen abgearbeitet worden sind.
Soll nun eine Box b^* entlang der x-Achse normalisiert werden, wird der maximale Wert mit $y_{max} = max(V_y[b^*.pos.y], ..., V_y[b^*.pos.y + b^*w])$ abgefragt. Der Wert von y_{max} gibt nun an, wo sich das erste Hindernis befindet. Bis zu dieser Position lässt sich nun also b^* normalisieren. Nachdem das geschehen ist, werden die abgefragten Werte neu gesetzt und zwar gerade auf $y_{max}+b^*.l$. Es darf nicht vergessen werden, dass diese Werte eventuell wieder zurückgesetzt werden müssen, falls die Box im Anschluss noch in y-Richtung normalisiert wird, dafür wird dann ein zweiter Vektor verwendet, der jeweils die vorherigen Werte speichert.
Kann eine Box entlang der aktuell beobachteten Achse nicht weiter normalisiert werden, wird sie aus der Liste der verschobenen Boxen entfernt und ans Ende der Liste der platzierten Boxen angefügt, inklusive eventueller aufgelegter Boxen. Für die y-Achse funktioniert dies analog (s. Abb. 4.6 (b)).

Auf diese Art werden die verschobenen Boxen abwechselnd in x- und y-Richtung normalisiert, bis kein Fortschritt mehr erreicht wird. Bei Stapeln werden die jeweiligen aufliegenden Boxen entsprechend der Box(en) auf der Grundfläche des Containers mit verschoben.

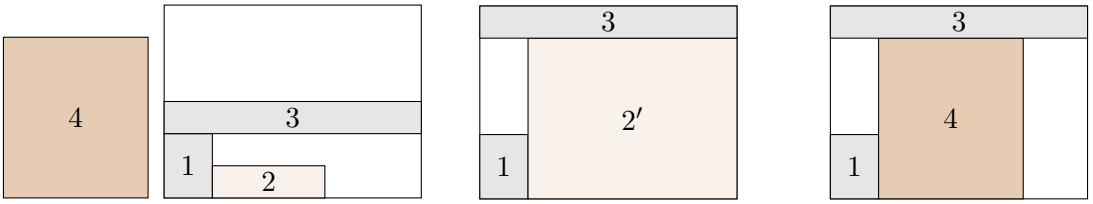
4.5. Ausdehnen und Ersetzen

Neben dem Einfügen einer Box an einem Extrempunkt wird im Paper von Zhang et al.[2] noch eine weitere Möglichkeit vorgestellt, um die Güte der Beladung zu verbessern. Diese wird Ausdehnen-Ersetzen genannt, und hat zum Ziel, eine Box durch eine andere Box zu ersetzen. Da sie das klassische Bin-Packing Problem behandeln, ist in ihrem Fall das Ziel, eine kleinere Box durch eine Größere zu ersetzen. Das Verfahren ist aber ohne Probleme adaptierbar. Es können zum Beispiel auch wie bei uns gewünscht die Kosten, oder aber das Gewicht einer Box betrachtet werden, anstatt der Größe.

Bei diesem Verfahren werden nacheinander die bereits positionierten Boxen betrachtet und jeweils mit der nächsten, noch nicht platzierten Box verglichen. Ist die bereits an Position p_b platzierte Box b kleiner als die neue Box b_{neu}, wird in Betracht gezogen, beide gegeneinander auszutauschen. Ist direkt genug Platz für b_{neu} an der Einfügepostion p_b, können beide einfach direkt ausgetauscht werden. Andernfalls wird b soweit wie möglich ausgedehnt. Der Box ist es dabei auch möglich, andere Boxen soweit zu verschieben, wie die berechnete Positionen nach einer PushOut Operation zulassen. Ist im Anschluss die ausgedehnte Box größer als b_{neu}, kann b_{neu} anstelle von b an Position p_b platziert werden. Dies ist in Abb. 4.7 illustriert. Danach sind die beiseite geschobenen Boxen zu Normalisieren. Die ersetzte Box b wird nun zu den noch nicht platzierten Boxen hinzugefügt. Das Ausdehnen einer Box kann dabei einfach durch ein PushOut(p_b) an der Einfügeposition realisiert werden. Sollte nach dem Ausdehnen einer Box dennoch kein Austausch möglich sein, wird dies rückgängig gemacht und die nächste platzierte Box betrachtet.

4.5.1. Implementierung

Da es im Ceschia Paper[1] um die Minimalisierung der Kosten geht, werden die zwei Boxen, welche gegeneinander ausgetauscht werden sollen, auch anhand ihrer Kosten verglichen,



(a) Box 4 ist zu groß (b) Ausdehnen (c) Ersetzen

Abbildung 4.7.: Ausdehnen und Ersetzen

anstatt ihres Volumens. Das Ausdehnen einer Box ist deshalb aber trotzdem noch nötig. Es sollen ja auch größere, teurere Boxen zum Einfügen in Betracht gezogen werden. Da je nach Position der zu ersetzenden Box andere Sachen beachtet werden müssen, teilen wir in vier verschiedene Fälle ein:

1. Die Box befindet sich auf dem Grund des Containers und es befinden sich keine weiteren Boxen auf ihr.

   In diesem Fall muss am wenigsten beachtet werden. Die Box wird soweit wie möglich ausgedehnt. Falls die einzufügende Box kleiner oder gleich groß der ausgedehnten Box ist, kann man die beiden Boxen austauschen.

2. Die Box befindet sich auf der Grundfläche des Containers und es befinden sich weitere Boxen auf ihr.

   Zusätzlich zum ersten Fall muss die neue Box noch zusätzliche Kriterien erfüllen. Die Grundfläche muss ausreichen groß sein, damit die Boxen, die darauf gestapelt werden sollen auch vollständig darauf passen. Außerdem muss sie das Gewicht dieser Boxen auch tragen können. Falls die neue Box höher ist als jene welche ersetzt werden soll, darf die Stapelhöhe die Containerhöhe nicht überschreiten.

3. Die Box befindet sich auf einem Stapel und es befinden sich keine weiteren Boxen auf ihr.

   Durch die Tatsache das die Box sich auf anderen befindet, darf die Grundfläche der neuen Box, die der Boxen direkt unter der zu ersetzenden Box, nicht überschreiten. Außerdem darf die neue Box nicht zu schwer für die Tragkraft des Stapels sein. Da es nur möglich ist Boxen, welche alleine auf einer Ebene des Stapels sind zu ersetzen, kann die Box gerade soweit ausgedehnt werden, wie die Grundfläche der Boxen der Ebene direkt darunter.

4. Die Box befindet sich auf einem Stapel und es befinden sich weitere Boxen auf ihr.

   Zusätzlich zum dritten Fall müssen noch die Probleme aus Fall zwei beachtet werden.

Das Ausdehnen der Box macht sich die bereits berechneten Positionen nach einem PushOut zu nutze. Hierbei wird gerade ein PushOut um die aktuelle Position der momentan betrachtenden Box ausgeführt. Die Box selbst wird also auch soweit es geht verschoben und die resultierenden x_r und y_r Werte geben an, wieweit die Box entlang x- und y-Achse ausgedehnt werden kann. Wie weit die Box entlang der z-Achse ausgedehnt werden kann, wird berechnet, indem von der Höhe des Containers, die summierten Höhen der aufliegenden Boxen abgezogen werden.
Ist die Box nach dem Ausdehnen mindestens so groß, wie die neue, teurere Box, kann sie durch diese in der Lösungssequenz ersetzt werden. Die ersetzte Box wird dann ans Ende der Liste der noch nicht eingefügten Boxen angehängt. Da einige Boxen verschoben wurden um Platz für das Ausdehnen zu schaffen, muss eine Normalisierung entlang der x- und

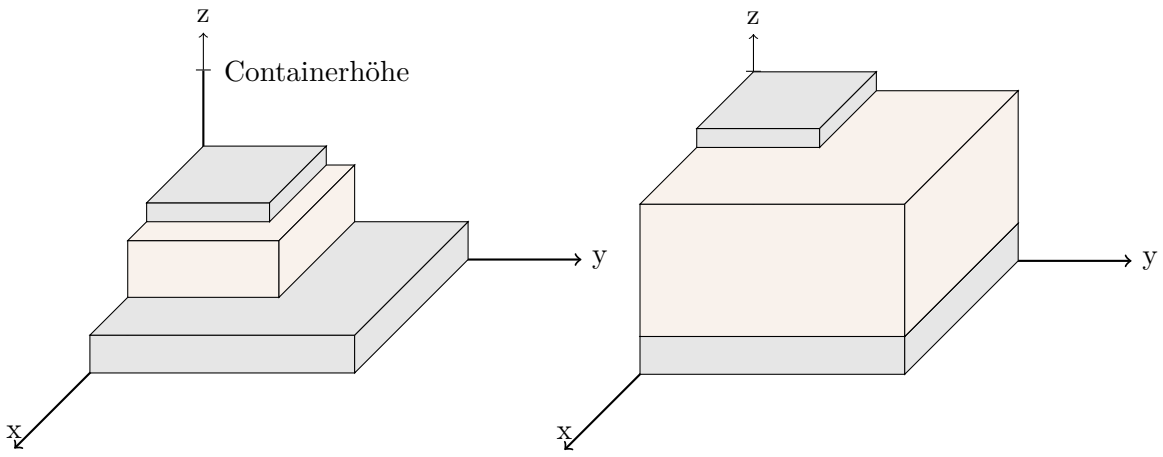


Abbildung 4.8.: Ausdehnen einer Box auf einem Stapel

y-Achse in Richtung Ursprung erfolgen, wie in Kapitel 4.4 beschrieben. Ebenso müssen neue Extrempunkte und PushOut Positionen berechnet werden. Natürlich muss auch die Multi-Drop Restriktion nach dem Austausch erfüllt sein. Wurden Boxen normalisiert, wird deshalb im Anschluss TestMultiDrop(seq) ausgeführt. Ist alles in Ordnung, war der Austausch gültig, ansonsten muss er rückgängig gemacht werden.

Zusammenfassend gilt: Es findet nur eine Ersetzung statt, falls die Kosten der neuen Box, die der alten übersteigen. Außerdem muss die Multi-Drop Eigenschaft weiterhin eingehalten werden, sowie die Gewichtsrestriktionen für Container und Stapel erfüllt sein.

Algorithmus 4.4 : Ausdehnen-Ersetzen(M_B, M_C, b)

Eingabe : Container M_C; Beladungen M_B; Einzufügende Box b

```
1  foreach Seq_C ∈ M_B do
2      foreach b' ∈ Seq_Ci mit b'.Kosten < b.Kosten do
3          Ausdehnen von b'
4          if b' kann durch b ersetzt werden then
5              Ersetze b' durch b
6              Normalisieren(C_i)
7              Aktualisiere Extrempunkte in C_i
8              Berechne die Positionen nach einem Push-Out aller Boxen in C_i neu
9              Füge b' zu noch nicht platzierten Boxen hinzu
10             return true
11 return false
```

4.6. Berechnung der Gesamtlösung mittels BinShuffling

Zu guter Letzt soll aus den bisherigen Komponenten dieses Kapitels eine Gesamtlösung berechnet werden. Dafür stellen Zhang et al.[2] abschließend noch eine *Bin Shuffling* Methode vor, welche die Lösung schrittweise verbessern soll.

Um zunächst eine Startlösung zu generieren, werden alle Boxen absteigend nach ihrem Volumen sortiert und mittels FreiraumDefrag in die benötigte Anzahl an Containern verladen. Die Beladesequenz Seq_{Ci} , sprich die Reihenfolge in der die Boxen platziert worden sind, wird für jeden Container C_i festgehalten. Die Gesamtbeladesequenz BS ist gerade $BS = [\text{Seq}_{C1}+\text{Seq}_{C1}+...+\text{Seq}_{Cn}]$. Im Anschluss wird der Container C_{Vmin} mit

der geringsten Volumenauslastung komplett entladen und die Boxen absteigend nach ihrem Volumen sortiert und in einer Liste L abgespeichert. Dazu gehört auch, dass Seq_{CVmin} aus BS entfernt wird. Nun kommt das Bin Shuffling aus Abb. 4.9 ins Spiel. Dabei wird nun wieder auf die gespeicherten Beladesequenzen $\text{Seq}_{Ci} \in BS^* = BS \setminus \{\text{Seq}_{CVmin}\}$ zurückgegriffen. Das Bin Shuffling sorgt nun dafür, dass die Reihenfolge der Beladesequenzen vertauscht werden. Die Beladesequenzen selbst bleiben unverändert. Für das Beispiel aus Abb. 4.9 gilt: $BS' = \text{BinShuffling}(BS^*) = [\text{Seq}_{C3}+\text{Seq}_{C1}+\text{Seq}_{C2}]$.

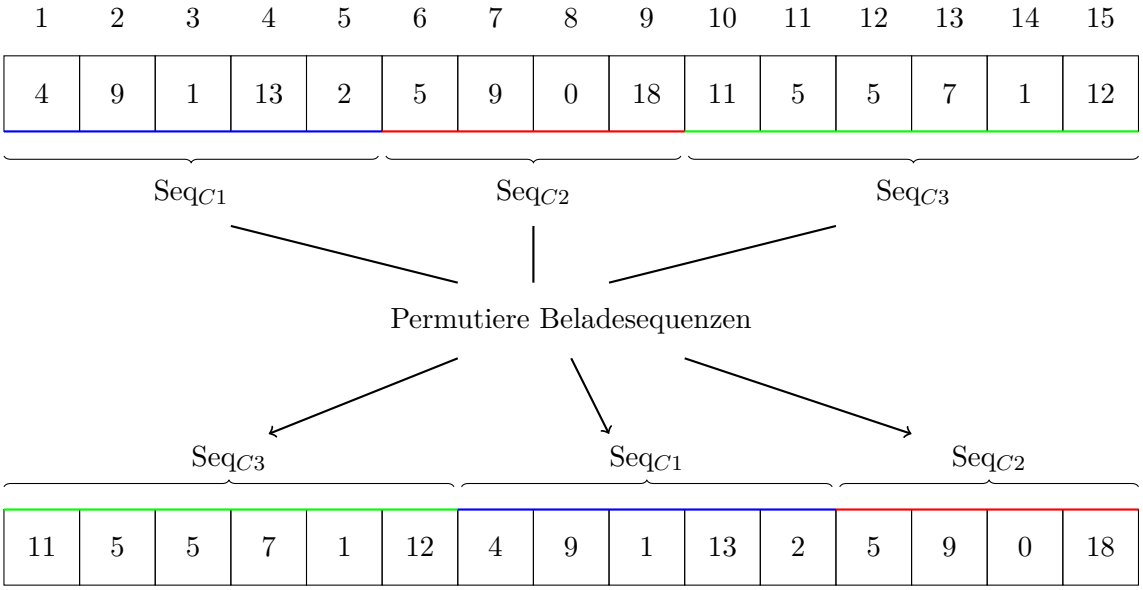


Abbildung 4.9.: Prinzip der BinShuffling Operation

Als nächstes wird die erste Box b_0 aus L an eine zufällige Position in einer der Beladesequenzen in BS' eingefügt und BS' dann als neue Eingabe für FreiraumDefrag verwendet. Falls die generierte Lösung keinen neuen Container benötigt, scheint die Beladesequenz vielversprechend zu sein und es wird versucht so viele weitere Boxen aus L wie möglich, mittels FreiraumDefrag in einen der bereits beladenen Container zu positionieren. Wenn eine weitere Box aus L, die Nutzung eines weiteren Containers erfordern würde, wird die Lösung zurückgegeben. Die in L verbliebenen Boxen werden dabei wieder an die gleichen Positionen, an denen sie zu Beginn waren, in C_{Vmin} eingefügt.
Falls die generierte Lösung stattdessen b_0 in einen neuen Container verladen muss, wird die Operation BinShuffling wiederholt. Treten solche Misserfolge eine bestimmte Anzahl oft hintereinander auf, wird eine zufällige Box aus $BS*$ entfernt und in L eingefügt. Nun wird wiederum damit das Prozedere erneut durchgeführt.

Mit dieser Verfahrensweise wird immer wieder probiert die Lösung zu verbessern, bis schließlich ein Container weniger benötigt wird, um die Boxen komplett zu verladen. Nach einem festgelegten Zeitlimit wird die Optimierung der Lösung abgebrochen.

5. Packheuristik mit lokaler Suche und Freiraumdefragmentierung

In diesem Kapitel werden zwei Ansätze für eine neue Packheuristik auf Grundlage der in Kapitel 3 und 4 vorgestellten Verfahren beschrieben. Zum einen gibt es einen seriellen Ansatz, in dem beide Verfahren strikt seriell nacheinander ausgeführt werden. Zum anderen wird ein verschachtelter Ansatz vorgestellt, in dem einzelne Komponenten bzw. Ideen der Freiraumdefragmentierung, während dem Beladeprozess des Loaders eingesetzt werden. Es gilt jeweils die Problembeschreibung aus Kapitel 3.1.

5.1. Serieller Ansatz

Der serielle Ansatz führt zuerst für eine Eingabe den implementierten Algorithmus von Ceschia und Schaerf aus, so wie in Algorithmus 3.5 aufgeführt. Falls noch nicht alle Boxen verladen werden konnten, soll die Lösung mit der FreiraumDefrag Operation kontinuierlich verbessert werden, wie ab Zeile 7 von Algorithmus 5.1 zu sehen ist. Die Beladesequenzen der Container werden in jeder Iteration permutiert und alle noch nicht platzierten Boxen in eine zufällige, gültige Ausrichtung rotiert. Nun werden so viele Boxen wie möglich mittels FreiraumDefrag verladen, oder kostenreduzierend ausgetauscht. Hat es geklappt die Kosten der Lösung zu senken, wird mit dieser Lösung fortgefahren. Andernfalls wird mit der alten Lösung weitergemacht. Dies entspricht gerade wieder einer lokalen Suche. Das Ganze bricht nach K Iterationen ab, oder falls keine Boxen mehr zu verladen sind. Das Resultat von Algorithmus 5.1 entspricht dann der Gesamtlösung des seriellen Ansatzes.

5.2. Verschachtelter Ansatz

Bei dem verschachtelten Ansatz wird während dem Beladen des Containers durch den Loader eine Freiraumdefragmentierung durchgeführt. Jedes Mal nachdem eine Box auf der Grundfläche des Containers positioniert wurde und der dazugehörige Stapel abgeschlossen ist, wird versucht die nächste Box in einem entstandenen Hohlraum zu platzieren. Die Hohlräume werden dabei durch einen von der momentanen Beladung verdeckten Extrempunkt repräsentiert(s. Abb. 5.1). Zusätzlich werden bestimmte Boxen zur Seite geräumt, um etwas mehr Platz zu schaffen. Kann die Box anschließend platziert werden, gilt es die verschobenen Boxen zu normalisieren und zu Letzt noch die Multi-Drop Eigenschaft zu prüfen. Ist diese erfüllt, war die Aktion erfolgreich und es müssen abschließend die Freiräume

Algorithmus 5.1 : Serielle Packheuristik

Eingabe : Menge an Containern M_C; Menge an Boxen M_B

```
1  L = CESCHIA_LOCALSEARCH(M_C, M_B)
2  C = verwendete Container
3  B = noch nicht verladene Boxen
4  if B ist leer then
5  |  return L
6  else
7  |  k = 0
8  |  while k < K and B nicht leer do
9  |  |  L' = BINSHUFFLING(L)
10 |  |  B' = rotiere Boxen in B zufällig
11 |  |  L' = FREIRAUMDEFRAG(C, L', B')
12 |  |  if Kosten(L') < Kosten(L) then
13 |  |  |  B = B'
14 |  |  |  L = L'
15 |  |  k = k + 1
16 |  return L
```

der Ladung neu berechnet werden. Danach fährt der Loader mit seiner gewohnten Arbeit fort, wobei als nächstes das Stapeln auf der gerade positionierten Box an der Reihe ist.

5.2.1. Implementierung

Um die verdeckten Extrempunkte auf der Grundfläche zu finden, müssen zuerst alle Extrempunkte wie bekannt berechnet werden. Anschließend wird eine Oberflächenstruktur gleich der Hindernisstruktur der Normalisierungsfunktion aus 4.4 erstellt und für die berechneten Extrempunkte geprüft, ob sie durch die Oberflächenstruktur verdeckt werden. Falls ja, stellt solch eine Extrempunkt einen gesuchten Hohlraum dar.
Der nächste Schritt besteht nun darin, die Positionen aller Boxen nach einem PUSHOUT zu berechnen. Eine Box soll dabei immer nur in eine Richtung verschoben werden, entweder entlang der x-Achse, oder entlang der y-Achse. Das hat den Grund, dass so die momentane Konstellation der Boxen weniger stark verändert wird und dadurch ebenfalls eine verminderte Neudefragmentierung erzielt werden kann. Um das Layerprinzip des Loaders später beibehalten zu können, wird zudem die maximale Verschiebung entlang der x-Achse, durch die aktuelle Layergrenze beschränkt. Entlang der y-Achse wird wie gewohnt bis zur Containerwand verschoben. Ausgehend von der momentanen Beladung können nun also die Positionen aller Boxen nach einer Verschiebung entlang der x-Achse / y-Achse mittels Algorithmus 4.2 getrennt voneinander berechnet werden.
Nachdem die Vorarbeit abgeschlossen ist, wird nun über die verdeckten Extrempunkte $ep = (x, y, 0)$ iteriert und geprüft, ob die nächste Box der Sequenz an dieser Position eingefügt werden kann. Dafür müssen Boxen, die mit der neuen Box kollidieren würden verschoben werden, natürlich inklusive der darauf gestapelten Boxen. Es stellt sich dabei die Frage, entlang welcher der beiden Achsen dies geschehen soll, falls in beiden Fällen die Kollision aufgehoben wird? Für eine einfache Entscheidungsfindung wurde deshalb eine Trennlinie $t = ep.x + \frac{L^*}{2}$ definiert, wobei L^* die Distanz zum nächsten Hindernis auf der x-Achse ist. Anhand von t kann nun eine zu verschiebende Box b' an Position $(x, y, 0)$ eingeteilt werden: Gilt $b'.x \geq t$, dann wird b' entlang der x-Achse verschoben, andernfalls entlang der y-Achse. Dies kann dann wieder rekursiv weitere Verschiebungen auslösen. Würde zum Beispiel in Abb. 5.1 direkt über Box 5 eine weitere Box liegen,

müsste diese natürlich dann auch verschoben werden. Dabei muss natürlich immer auch der Ausnahmefall, dass mehrere identische Boxen auf der Grundfläche übereinander liegen, beachtet werden. Diese Boxen werden wie eine größere Box behandelt und wenn dann gemeinsam versetzt, so dass die Basis für die darauf gestapelten Boxen erhalten bleibt.

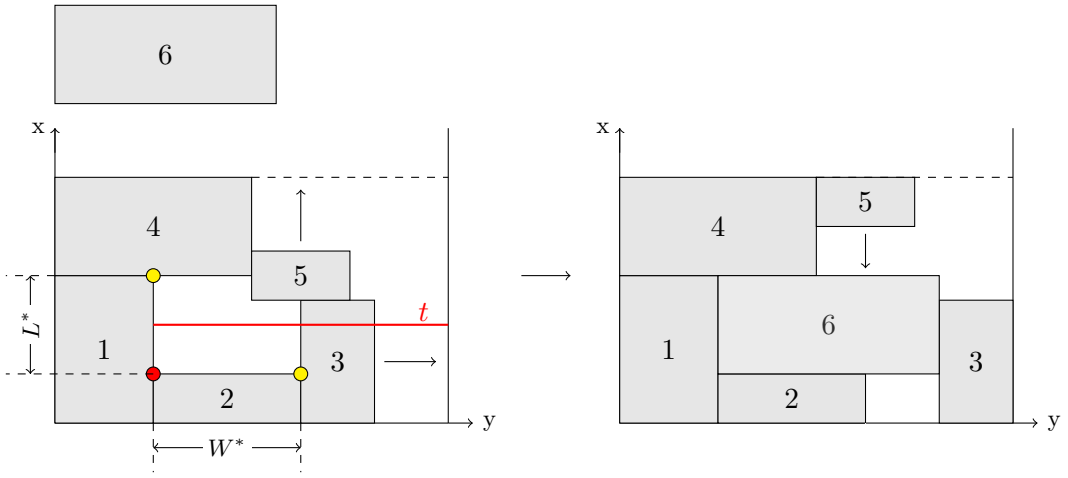


Abbildung 5.1.: Box 6 soll an dem roten Extrempunkt im Hohlraum platziert werden. Um Platz zu schaffen werden Box 3 und 5 beiseite geräumt. Nun kann die Box eingefügt werden.

Zusätzlich soll verhindert werden, dass beliebig große Boxen, in einem beliebig kleinen Hohlraum platziert werden, denn meist entsteht dadurch ein deutlich größerer Hohlraum. Deshalb darf die Länge und Breite einer Box, die Länge und Breite eines Hohlraum nur in einem begrenzten Maße überschreiten. Der Punkt $ep.x$ wird dafür entlang der y-Achse, sowie der Punkt $ep.y$ entlang der x-Achse, auf das nächste Hindernis projiziert, dargestellt durch die gelben Punkte in Abb. 5.1. Der jeweilige Abstand zu ep ergibt dann die Werte L^* und W^*. Eine neue Box b mit Dimension (l, w, h) soll nur dann an ep platziert werden können, falls gilt: $b.l \leq \lambda * L^* \wedge b.w \leq \lambda * W^*$, wobei hier $\lambda = 2$ gewählt wurde.

Algorithmus 5.2 : FreiraumDefrag_Loader(b)

Eingabe : Einzufügende Box b

```
1  EpList = berechne verdeckte Extrempunkte
2  foreach ep ∈ EpList do
3      if !(Größe b >> Größe Hohlraum) then
4          PushOut(ep) für im Weg stehende Boxen
5          if b platzierbar an ep then
6              einfügen(b,ep)
7              normalisieren()
8              if Multi-Drop erfüllt then
9                  Neuberechnung der Freiräume
10                 initialisiere_stapel()
11                 break und fahre fort mit Beladung
```

Können alle potentiellen Kollisionen aufgehoben werden und ist die Box nicht überverhältnismäßig groß, wird sie platziert. Im Anschluss sind die beiseite geräumten Boxen in gewohnter Art und Wiese zu normalisieren. Zu guter Letzt muss noch mit Algorithmus 3.6 die Multi-Drop Eigenschaft getestet werden. Ist diese erfüllt, werden abschließend noch die

Freiräume berechnet und der Loader kann mit der momentanen Beladung fortfahren. Der nächste Schritt wäre dann, auf der eben platzierten Box einen Stapel zu bilden. Kann die Box stattdessen nicht erfolgreich positioniert werden, wird der nächste Hohlraum betrachtet. Die in diesem Abschnitt beschriebene Vorgehensweise kann in Algorithmus 5.2 noch einmal nachvollzogen werden.

5.2.2. Neuberechnung von Freiräumen

In diesem Abschnitt soll es darum gehen, wie Freiräume aus einer Beladung konstruiert werden können. Ändert sich während der Beladung eines Containers durch den Loader plötzlich die Konstellation der Boxen durch Verschieben und Normalisieren, verlieren die bis dato existierenden Freiräume ihre Gültigkeit. Damit der Loader im Anschluss seiner gewohnten Arbeit nachgehen kann, ist es deshalb erforderlich, die durch die Ladung induzierten Freiräume neu zu berechnen. Doch wie findet das Programm die Freiräume? Dafür wurde eine Funktion entwickelt, die dies erledigt.

Das Verfahren wird in der Folge an einem konkreten Beispiel erläutert, welches durch die Abbildungen der Zustände $Z0$ bis zum Endzustand dargestellt ist.

Zuerst wird die Oberflächenstruktur OS einer aktuellen Beladung inspiziert. Diese wird genauso berechnet wie die Hindernisstruktur für das Normalisierungsverfahren aus Kapitel 4.4. Ziel es nun die Oberflächenstruktur entlang der y-Achse bis zur Containerwand nach Freiräumen $F = (x, y, w)$ zu erkunden, siehe dazu Abb.5.2. Der grüne Punkt(s. Abb.5.2, $Z0$) zeigt dabei jeweils die aktuelle Position auf der blau hervorgehobenen Oberflächenstruktur an. Die Ausgangssituation wird durch Startzustand $Z0$ dargestellt. Das Verfahren endet im Endzustand. Gelbe Punkte stellen Freiräume dar, bei denen noch die Information über ihre Breite w_y entlang der y-Achse fehlt. Rote Punkte stellen abgeschlossene Freiräume dar, welche mit ihrer vollständigen Information (x, y, w_y) konstruiert werden konnten. Die Position (x, y) eines Freiraumes auf der Grundfläche des Containers wird dabei erkannt, falls $OS(y_{i-1}) \neq OS(y_i)$. Dann ist die Position des Freiraumes $(x, y) = (OS(y_i), y_i)$, wobei y_i die aktuelle Position des Beobachtungspunktes auf der Oberflächenstruktur darstellt. Ein Sprung nach unten wird erkannt durch $OS(y_{i-1}) > OS(y_i)$ und ein Sprung nach oben durch $OS(y_{i-1}) < OS(y_i)$.

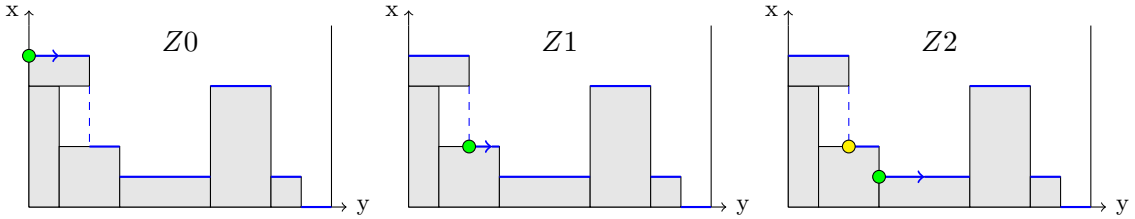


Abbildung 5.2.: Zustände $Z0 - Z2$ der Erkundung von OS (blau)

Zu Beginn wird also ausgehend von Position $(OS(0), 0)$ in $Z0$ die Oberflächenstruktur erkundet. In $Z1$ wird an der aktuellen Position, die Position (x_1, y_1) eines Freiraumes erkannt, da ein Sprung nach unten in OS detektiert wurde. Diese Position wird in einer Liste LF_{pos} gespeichert. Später muss noch die Breite w_y des Freiraumes entlang der y-Achse hinzugefügt werden. Außerdem wird in einer weiteren Oberflächenstruktur OS_L (s. Abb. 5.4b), die erkundete Hindernisstruktur links des Erkundungspunktes festgehalten. Darauf muss später für andere Freiräume noch zurückgegriffen werden.
Der gleiche Fall tritt in $Z2$ auf. In Abb. 5.4b ist OS_L zum Zeitpunkt des Zustandes $Z2$, durch die Blöcke $Z1$ und $Z2$ dargestellt.

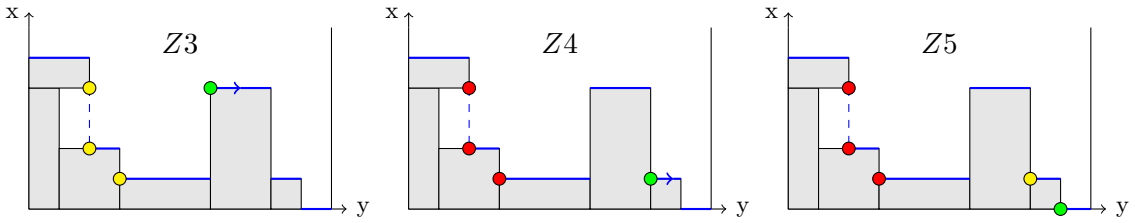
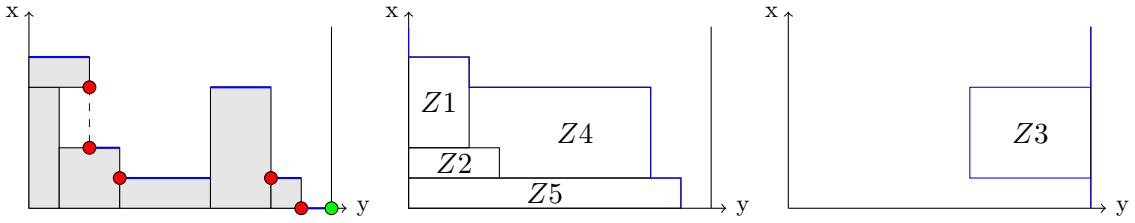


Abbildung 5.3.: Zustände $Z3 - Z4$ der Erkundung von OS

In Zustand $Z3$ wird nun wieder ein Sprung detektiert. Diesmal stellt der Sprung aber eine Steigung dar. Die aktuelle Position des Erkundungspunktes (x_E, y_E) ist dann nicht direkt die Position eines Freiraumes, nur die die x-Koordinate ist passend. Die dazugehörige y-Koordinate entspricht nun $OS_L(x_E)$. Damit ist die Position des nächsten Freiraumes gleich $(x_E, OS_L(x_E))$. Wird ein Sprung nach oben aufgefunden, werden in einer dritten Oberflächenstruktur OS_R (s. Abb. 5.4c) Hindernisse festgehalten, die die Breite bereits entdeckter Freiräume einschränken könnten. Tritt nach einer Anzahl von aufeinander folgenden Anstiegen, wieder ein Abstieg auf, wie es in $Z4$ der Fall ist, werden die Positionen $p \in FL_{pos}$ durchlaufen und nacheinander die Breite der Freiräume abgefragt mit $w_y = OS_R(p.x)$. Somit können zu diesem Zeitpunkt alle in FL_{pos} gespeicherten Einträge fs mit $fs.x < OS[i-1]$ um die letzte Information ergänzt werden und man erhält jeweils die vollständig beschriebenen Freiräume. Außerdem wird in $Z4$ an der aktuell beobachteten Position wieder die Position (x, y) eines Freiraumes entdeckt, sowie OS_L aktualisiert. Gleiches gilt für $Z5$.



(a) Endzustand (b) OS_L (c) OS_R

Abbildung 5.4.: OS_L wird bei jedem detektierten Abfall in OS aktualisiert, OS_R bei jedem detektierten Anstieg. Hier sind beide jeweils nach Erreichen des Endzustandes abgebildet. Die Zustände darin geben an, wann die jeweilige Aktualisierung stattfand.

Das Verfahren geht solange weiter, bis der Endzustand erreicht ist, bei dem die Containerwand erreicht wird. Alle noch in FL_{pos} enthaltenen Positionen werden dann mit $w_y =$ Containerbreite ergänzt und zu den bereits abgeschlossenen Freiräumen hinzugefügt. Somit sind alle durch die Beladung induzierten Freiräume aufgebaut.

5.2.3. Vergleich mit dem ursprünglichen Loader

Zum Abschluss dieses Kapitels soll nun noch ein Vergleich des Loaders aus Kapitel 3.2 mit der eben vorgestellten Variante mit Freiraumdefragmentierung erfolgen.

Die wohl größte Stärke und zugleich größte Schwäche des originalen Loaders ist seine Einfachheit. Die Boxen werden simpel in der vorgegebenen Reihenfolge nach dem *First In - First Out* Prinzip in die Freiräume geladen, egal wie gut die Box hinein passt. Dadurch ist das Verfahren sehr schnell und es können über eine große Anzahl unterschiedlicher

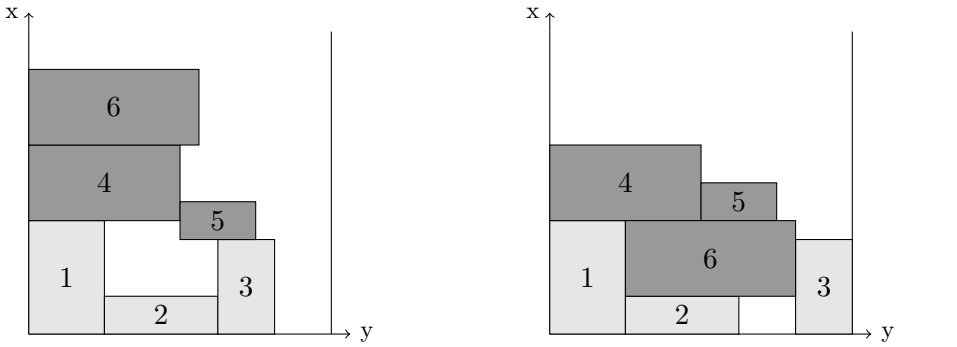
Algorithmus 5.3 : KONSTRUKTION-FREIRÄUME

Eingabe : Aktuelle Beladung des Containers

```
1  initialisiere(OS, OS_L, OS_R)
2  indikator = false;
3  for i = 1 to Containerbreite do
4      if OS[i] > OS[i − 1] then
5          neuer Freiraum mit Position (x, y) = (OS[i], OS_L[OS[i]]) zu Liste LF_pos hinzu
6          aktualisiere OS_R
7      if OS[i] < OS[i − 1] then
8          foreach fs ∈ LF_pos do
9              if fs.x < OS[i − 1] then
10                 Breite von fs = OS_R[fs.x]
11                 Füge fs zur Liste der vollständig berechneten Freiräume LF* hinzu
12         füge neuen Freiraum mit Position (x, y) = (OS[i], i) zu Liste LF_pos hinzu
13         aktualisiere OS_L
14 return LF*
```

Sequenzen gute Lösungen generiert werden. Der Nachteil ist allerdings, dass dadurch direkt sehr viele Sequenzen eine schlechte Lösung liefern und verworfen werden. Möglicherweise entsteht durch eine Sequenz ein riesiger Freiraum nachdem eine Box eingefügt wurde, wenn man es aber schafft diesen mit den verbleibenden Boxen gut aufzufüllen, kann die Sequenz dennoch wertvoll sein. Womöglich können sogar auf diese Weise noch bessere Lösungen erstellt werden. Genau an diesem Problem setzt die verschachtelte Packheuristik an und soll den Loader verfeinern. Zum einen können Lösungen generiert werden, die ursprünglich nicht möglich sind. Zum anderen kann aus einer für den Loader schwächeren Sequenz, direkt die gleiche Lösung erstellt werden, wie für eine Stärkere. Damit kann die Anzahl der betrachteten Nachbarn verringert werden, ohne an Qualität einzubüßen. Diese Vorteile werden aber durch einen viel höheren Aufwand erkauft, da ständig Extrempunkte berechnet und Boxen verschoben werden müssen, was sich deutlich auf die Laufzeit auswirkt.

Die beiden Vorteile sollen jetzt jeweils noch einmal, an einem einfachen zweidimensionalen Beispiel gezeigt. Im ersten Beispiel in Abbildung 5.5 werden die Boxen der Nummerierung nach von 1 bis 6 von beiden Loadern verladen. Die Zieldestinationen sind durch die Graustufe der Boxen dargestellt, dunkle Boxen müssen dabei vor den hellen Boxen entladen werden.

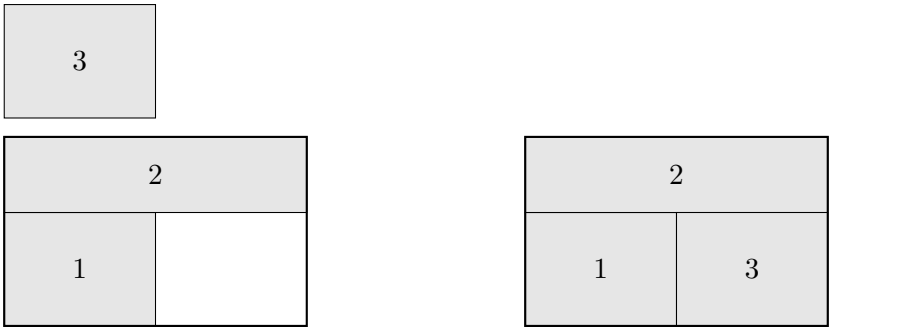


(a) Loader Ceschia (b) Loader mit Freiraumdefragmentierung

Abbildung 5.5.: Vergleich der Loader (1)

Eine Beladung wie in Abb.5.5b kann mit dem gewöhnlichen Loader nicht erreicht werden. Box 1 und 2 können noch identisch platziert werden, Box 3 wird dann aber wie in 5.5a positioniert. Somit ist dann bereits zu diesem Zeitpunkt der Plan aus (b) nicht mehr nachbildbar. Ist einmal eine Box platziert, verbleibt sie auch bis zum Schluss an dieser Position. Im Gegensatz dazu kann die weiterentwickelte Variante Boxen noch verschieben. Im Beispiel werden die Boxen 5 und 3 nach außen geschoben, um für Box 6 Platz zu machen, damit diese in den Hohlraum passt.

Für das zweite Beispiel in Abbildung 5.6 sind alle Boxen der gleichen Zieldestination zugeordnet. Der gewöhnliche Loader kann mit der Sequenz [1,2,3] nur zwei Boxen in den Container verladen. Findet die lokale Suche Sequenz [1,3,2], kann er alle Boxen passend platzieren. Durch die Freiraumdefragmentierung wird dagegen direkt mit der ursprünglichen Sequenz die optimale Lösung gefunden.



(a) Loader Ceschia (b) Loader mit Freiraumdefragmentierung

Abbildung 5.6.: Vergleich der Loader (2)

An letzter Stelle dieses Kapitels soll nun noch eine gemeinsame Schwäche beider Loader aufgezeigt werden. Bei beiden ist die Eingabesequenz immer strikt nach Zieldestinationen geordnet. Dadurch werden Boxen früherer Zieldestinationen, auch wenn sie schlechte Lösungen liefern, immer vor Boxen späterer Zieldestinationen platziert. Somit ist es nicht möglich, diese "schlechten" Boxen einfach überhaupt nicht zu verladen, um dafür mehr Platz für Boxen späterer Zieldestination zu schaffen, die bis jetzt nicht mehr in den Container gepasst haben. Für Instanzen mit mehreren Containern kann dies teilweise noch umgangen werden, indem diese Boxen in die Sequenz eines anderen Containers versetzt werden. Für Instanzen in denen nur ein Container beladen wird, ist dieses Problem aber unumgänglich. Dazu folgendes minimalistisches Beispiel, bei dem die Zieldestinationen wieder durch die Graustufen dargestellt sind und die Kosten der Boxen im Verhältnis zu ihrer Größe stehen:

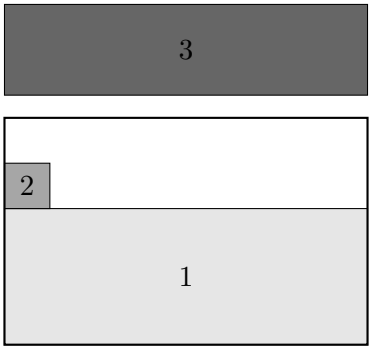


Abbildung 5.7.: Beide Loader können hier nicht die beste Lösung finden.

Da die Sequenz [1,2,3] aufgrund der vorgeschriebenen Anordnung nach Zieldestinationen unveränderlich ist, ist es nicht möglich Box 3 anstatt Box 2 zu platzieren. Somit entsteht

immer die Lösung aus Abbildung 5.7. Abhilfe könnte hier ein gezielter Einsatz einer Ersetzungsstrategie schaffen, ähnlich der Ausdehnen-Ersetzten Operation aus Kapitel 4.5.

6. Evaluation

In diesem Kapitel geht es zu Beginn um die Validierung von Lösungen. Im Anschluss daran werden die Ergebnisse der Implementierungen vorgestellt und gegeneinander verglichen werden. Die Ergebnissen von Ceschia und Schaerf dienen dabei als Referenzwerte.

6.1. Validierung einer Lösung

Bevor geprüft wird, wie gut eine Lösung ist, muss zuerst geprüft werden, ob diese überhaupt gültig ist. Zu diesem Zweck wurde eine Funktion implementiert, welche eine Lösung validiert. Dabei wird die Lösung auf folgende Kriterien geprüft:

- Alle platzierten Boxen müssen sich innerhalb des zugehörigen Container befinden
- Es dürfen sich keine zwei platzierten Boxen überschneiden
- Jede Box muss eine gültige Orientierung vorweisen, denn nicht jede Box darf auf jeder Seite platziert werden
- Jede Box die auf einem Stapel platziert ist, darf nicht über die darunterliegende Ebene hinausragen
- Die Multi-Drop Restriktion muss erfüllt sein
- Die maximale Tragelast des Container darf nicht überschritten werden
- Die maximale Tragelast eines Stapels darf nicht überschritten werden

Generell sollte eine berechnete Lösung stets gültig sein, jedoch kann immer mal an einer Stelle ein Fehler auftreten, der übersehen wurde. Somit eignet sich diese Funktion hervorragend zum Bekämpfen von Fehlern. Außerdem können damit ebenfalls die Lösungen der Ceschia Gruppe validiert werden.

Validierung der Lösungen von Ceschia und Schaerf

Da bei den Vergleichen zwischen eigenen Ergebnissen und denen der Ceschia Gruppe, teilweise große Diskrepanzen auftraten, wurden die Lösungen genauer betrachtet. In den zur Verfügung gestellten Lösungen wurde für jede Box nur ID, Ausmaße und Position im Container angegeben, daraus kann die Lösung nicht komplett aus den Eingabedaten rekonstruiert werden. Eine eindeutige Zuordnung von Rotationseigenschaft und tragbarem

Gewicht pro Fläche für gleichlange Seiten ist nämlich nicht möglich. In solchen Fällen wurde angenommen, dass die vorhandene Orientierung der Box gültig ist und die Seite mit der größten Tragkraft pro Fläche oben liegt. Alle anderen Kriterien wurden in normaler Weise für die Lösungen der Ceschia Gruppe validiert.
Insgesamt sind gerade einmal 16 von 117 Lösungen gültig. Das Problem ist immer das gleiche, die Beladung übersteigt das maximale Tragegewicht des Containers. Entweder wurde ein unerwähnter Faktor eingerechnet, oder dieses Kriterium wurde schlichtweg vergessen. In Tabelle 6.1 sind alle gültigen Instanzen aufgeführt.

CS2000, CS2001, CS2002, CS2003, CS2822, CS2914, CS2925, CS2943
CS3056, CS3079, CS3220, CS3695
CS4968, CS4970, CS4971
CS5005

Tabelle 6.1.: Instanzen für die gültige Lösungen vorlagen (16/117)

Da es schwierig ist zwei Verfahren anhand von nur 16 Probleminstanzen zu vergleichen, wurde bei den Tests der eigenen Implementierungen, die maximale Tragelast des Containers ebenfalls ignoriert.

6.2. Testfälle

Für die experimentellen Tests wurden 117 Realwelt-Instanzen verwendet, die von einem Industriepartner der Ceschia Gruppe stammen. Diese variieren sehr durch Unterschiede der Containertypen, Boxtypen, Anzahl Container pro Typ, Anzahl Boxen pro Typ, Gesamtzahl Container, Gesamtzahl Boxen. Für ungefähr 75% der Instanzen gilt, dass das verfügbare Containervolumen relativ deutlich über dem Gesamtvolumen der zu verladenden Boxen liegt und somit eine vollständige Beladung dieser in die Container erwartet wird. Dies hat zur Folge, dass die Kostenkomponente c_1 für jeden genutzten Container den Wert 0 annimmt und $c3$ die Güte der Lösung dominiert. Deshalb wird in zwei Testklassen unterteilt: Eine Klasse TK1 für Instanzen, für die die Implementierung des Ceschia Verfahrens nicht alle Boxen verladen kann und eine Klasse TK2, für die das Gegenteil der Fall ist. Dies macht vor allem deshalb Sinn, da das serielle Verfahren ausschließlich für Instanzen der Klasse TK1 Lösungen berechnet und dadurch nur anhand dieser evaluiert werden kann.

6.3. Testumgebung

Alle Test wurden auf einem Rechner mit zwei Dual-Core AMD Opteron(tm) Prozessoren vom Modell 2218 mit 2,6 GHz durchgeführt. Hierbei standen 32 GB Arbeitsspeicher zur Verfügung. Die vorgestellten Algorithmen wurden alle in C++ Version 11 programmiert. Da es sich dabei fast ausschließlich um serielle Vorgänge handelt, wurde keine Parallelität implementiert.

6.4. Parametereinstellung

Die Gewichte für die Komponenten der Kostenfunktion f wurden von Ceschia und Schaerf übernommen : $[w_1, w_2, w_3, w_4] = [0.00005, 0.05, 0.1, 1]$.

Für das Simulierte Abkühlen wurden die Parameter auf folgende Werte eingestellt: $T = 50$, $T_{min} = 0.01$,$\sigma_N = 750, \tau = 0.95$. Daraus folgt eine Gesamtzahl von $I \approx 1,245 * 10^5 * |D|$ Iterationen. Ceschia und Schaerf verwenden dagegen eine etwas andere Konfiguration, welche ca. das 50-fache an Iterationen mit sich bringt. Die Änderung beruht auf einer

gewünschten kürzeren Laufzeit der Tests. Bei gleicher Konfiguration würde ein Testset von 5 Test pro Testinstanz über einen Tag veranschlagen. Dies ermöglicht im Zeitrahmen einer Bachelorarbeit allerdings kein effizientes Testen.

Für die verschachtelte Packheuristik aus Kapitel 5.2 entsteht ein erheblicher Mehraufwand ggü. der einfachen Implementierung des Ceschia Verfahrens. Deshalb wurden hierfür die Werte der Parameter nochmals angepasst: $T = 50$, $T_{min} = 0.01$, $\sigma_N = 100$, $\tau = 0.9$. Daraus folgt $I \approx 8084 * |D|$.

6.5. Testergebnisse

In diesem Abschnitt sollen nun die Testergebnisse für die Testklasse TK1 genauer vorgestellt und die jeweiligen Verfahren miteinander verglichen werden. Die Lösungen der Heuristiken für die Instanzen von TK2 sind alle sehr gleichwertig und unterscheiden sich kaum. Dies liegt vor allem daran, dass es dabei kaum schwer fällt, alle Boxen in die verfügbaren Container zu verladen und somit die Ergebnisse auch nicht viel Aufschluss darüber geben, wie gut die einzelnen Verfahren sind. Aus diesem Grund soll auf eine nähere Betrachtung der Ergebnisse von TK2 verzichtet werden. Die Ergebnisse können aber in den Tabellen A.1 - A.3 im Anhang begutachtet werden.

Für TK1 wurden nun die folgenden Verfahren getestet: Die eigene Implementierung des Ceschia Verfahrens (**CS_TV**) aus Kapitel 3, die Packheuristik, die aus dem seriellen Ansatz (**TV1**) aus Kapitel 5.1 hervorgeht und die verschachtelte Packheuristik (**TV2**) aus Kapitel 5.2. Zusätzlich werden dazu die Ergebnisse der Ceschia Gruppe (**CS**) als Referenz angegeben.

In Tabelle 6.2 sind in der ersten Zeile die Mittelwerte für die Kosten der einzelnen Lösungen dargestellt. In Zeile 2, die prozentuale Abweichung des Mittelwertes aus Zeile 1, zu dem Mittelwert der Gesamtkosten aller Instanzen. In Zeile 3 wird schließlich noch die durchschnittliche Laufzeit angegeben. Die Gesamtkosten einer Lösungsinstanz setzen sich dabei aus den maximalen Werten zusammen, die die Kostenkomponenten $c_1, ..., c_4$ annehmen können. Also konkret aus Nutzungsgebühren aller genutzten Container, Kosten der Boxen, minimaler linear freier Raum entlang der x-Achse und für jeden Container die maximale Anzahl an Stopps.

Die exakt berechneten Kosten jeder einzelnen Instanz, können in Tabelle 6.3 nachgesehen werden.

∅ - Werte	**CS**	**CS_TV**	**TV1**	**TV2**
Kosten(€)	3152,13	3741,93	3568,38	3963,91
Abweichung(%)	0,00	3,86	2,73	5,31
Laufzeit(s)	39,41	22,15	24,39	52,53

Tabelle 6.2.: Vergleich der unterschiedlichen Implementierungen und Heuristiken für TK1

Wie in Tabelle 6.2 zu sehen ist, erzielt die eigene Implementierung der Heuristik von Ceschia und Schaerf ein im Schnitt um 3,86% schlechteres Ergebnis als das Original. Da beim Original in jedem Schritt der lokalen Suche 250 mehr Nachbarn des aktuellen Zustandes betrachtet und insgesamt das 43-fache an Gesamtiterationen durchgeführt werden, stellt dies gerade noch ein gutes Ergebnis dar. Vereinzelt konnten auch für ein paar Instanzen bessere Ergebnisse erzielt werden. Betrachtet man die Laufzeit von 22,15 Sekunden ist noch deutlicher Optimierungsbedarf zu erkennen, vor allem da erheblich weniger Iterationen durchgeführt werden.

Für die serielle Kombination beider Verfahren TV1 wurden die erzielten Ergebnisse von CS_TV als Eingabe genutzt. Dadurch konnte die Abweichung von 3,86% auf 2,73% abgesenkt werden. Schaut man sich die einzelnen Ergebnisse näher an, erkennt man dass für 16 der 42 Lösungen keine Verbesserung mehr erzielt werden konnte. In Abbildung 6.1 ist für jede Instanz von TK1 die prozentuale Abweichung von CS_TV und TV1 zu CS abgebildet. Die Kostenreduktion, die durch TV1 erzielt werden kann, liegt dabei im Bereich von $0\% - 8\%$. Die Gesamtverbesserung für alle Testinstanzen von 1,13% fällt insgesamt

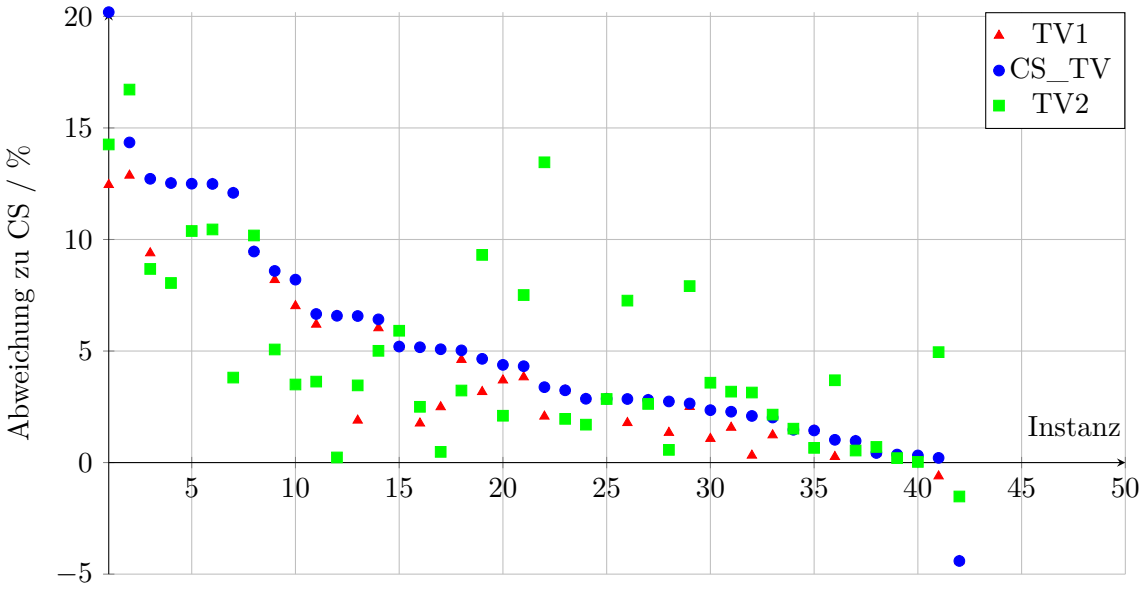


Abbildung 6.1.: Darstellung der prozentualen Abweichung von TV1, TV2 und dem seriellen Ansatz CS_TV zu CS. Die roten Dreiecke geben zu erkenne, wenn durch TV1 noch eine Verbesserung der Lösungen von CS_TV erzielt werden konnte.

eher gering aus. Das hat sehr wahrscheinlich den Grund, dass die Boxen bei den Lösungen von CS_TV meist schon relativ kompakt auf der Grundfläche des Container gepackt sind und sich somit an Extrempunkten in der Ladung nicht mehr viele Boxen platzieren lassen. Zudem muss auch nach dem Einfügen die Multi-Drop Restriktion weiterhin erfüllt sein, was einige passende Boxen vor dem dortigen Positionieren ausschließt. Große Abweichungen von über 10% lassen auf das Festhängen in einem lokalen Minima schließen. Es kommt ebenfalls vor, dass für Instanzen mit geringen Gesamtkosten, sich dass nicht Verladen von nur sehr wenigen Boxen signifikant auf die Kosten der Lösung auswirkt. Dies ruft dann direkt eine hohe Abweichung zu einer Lösung mit etwas mehr geladenen Boxen hervor. Die Gesamtlaufzeit für TV1 ergibt sich aus der Laufzeit von CS_TV und der Laufzeit für das nachträgliche verbessern der Lösung.
Die Heuristik TV2 mit dem optimierten Beladeмechanismus fällt mit einer Abweichung von $5,31\%$ etwas ab. Die Gründe für die schlechteren Ergebnisse sind aber sehr wahrscheinlich die gleichen wie eben erwähnt. Nur werden hier nochmals deutlich weniger Iterationen durchgeführt, genau genommen führt das CS Verfahren ungefähr das 667-fache an Gesamtiterationen durch. Schaut man sich jedoch in Abb.4.1 noch die einzelnen prozentualen Abweichungen für jede Instanz an, fällt auf das die Ergebnisse meistens sogar besser sind, als die von CS_TV und TV1. Das schlechtere Gesamtergebnis lässt sich dann so erklären, dass die Instanzen für die TV2 etwas schlechtere Lösungen liefert, sich stärker auf den Durchschnitt der Gesamtkosten auswirken. In Anbetracht dessen erscheint das Ergebnis dann doch nicht ganz so schlecht zu sein. Allerdings fällt auch die enorm hohe Laufzeit auf, die bedingt durch den großen Mehraufwand der Freiraumdefragmentierung auftritt.

Instanz	CS	CS_TV	TV1	TV2
CS2000	372.12	240.20	240.20	326.79
CS2001	1503.68	1916.83	1761.48	2556.25
CS2003	539.95	742.03	742.03	749.33
CS2803	287.70	292.40	292.40	289.20
CS2805	3064.72	3418.39	3185.04	3235.62
CS2822	284.03	386.76	330.77	440.80
CS2899	6239.67	6942.25	6900.30	6787.86
CS2943	41558.90	45516.80	42171.50	47513.67
CS2966	11122.40	12345.50	11871.00	15994.20
CS2975	2150.40	2349.81	2248.11	2192.16
CS3074	432.05	992.47	992.47	608.85
CS3079	1537.71	2304.79	2268.95	1990.64
CS3084	5055.44	5399.02	5399.02	5262.97
CS3115	5662.11	5900.26	5900.26	5770.51
CS3122	940.64	1338.52	1281.25	1110.42
CS3136	403.98	435.86	435.86	422.89
CS3142	356.40	764.07	764.07	618.47
CS3151	2284.95	2544.93	2544.93	2580.32
CS3152	171.80	337.31	253.09	187.40
CS3182	186.60	1035.69	710.30	786.14
CS3203	17026.90	17591.60	17592.00	17557.10
CS3207	1115.79	1727.09	1566.93	1532.65
CS3240	1093.24	1527.41	1490.97	1372.35
CS3314	269.60	631.67	605.89	466.69
CS3432	3487.85	3776.87	3731.52	3626.38
CS3571	1565.60	3498.81	3498.81	3646.01
CS3605	345.10	480.60	427.93	489.57
CS3653	368.30	425.87	382.85	576.74
CS3654	699.50	988.25	898.09	1102.82
CS3805	2535.91	3364.62	3270.42	3974.69
CS3810	821.94	883.34	883.34	920.76
CS3836	6182.73	6223.94	6061.66	7171.20
CS3870	349.20	785.75	475.01	579.53
CS3889	2691.55	3124.89	3099.81	3983.59
CS3915	930.77	1179.23	1179.23	939.57
CS3928	1715.10	4817.81	4497.69	5329.93
CS3933	3618.50	6644.41	5683.68	9678.18
CS3941	173.80	290.11	290.11	289.81
CS3967	850.20	997.62	997.62	931.80
CS4747	2126.08	2149.32	2146.98	2139.30
CS4970	133.40	424.26	375.02	375.02
CS4971	133.00	423.56	423.56	376.22

Tabelle 6.3.: Ergebnisse für die Instanzen der Klasse TK1

7. Fazit

In dieser Arbeit wurde zunächst die Packheuristik von Ceschia und Schaerf[1] vorgestellt, die ein komplexes Containerladeproblem behandelt. Sie findet mit Hilfe einer angepassten Variante der lokalen Suche, dem simulierten Abkühlen und einem relativ simplen und schnellen Belademechanismus, sehr gute Lösungen. Um später aufbauend auf diesem Verfahren eine Packheuristik zu entwickeln, wurde das Verfahren Schritt für Schritt nach implementiert. Im Anschluss daran wurde die Packheuristik von Zhang et al.[2] eingeführt, welche mit einem allgemeiner definierten Packproblem umgeht. Das Verfahren versucht Boxen an attraktiven Positionen im Container, dazu gehören auch Hohlräume inmitten der Beladung, zu platzieren. Außerdem werden kleinere, bereits verladene Boxen, durch Größere ersetzt. Um nun das Verfahren von Ceschia und Schaerf um ein paar dieser Konzepte zu erweitern, wurden die einzelnen Bestandteile der Heuristik von Zhang et al. an dessen Problembeschreibung angepasst, was sich durch die viel größere Anzahl an zu beachtenden Restriktionen nicht immer einfach gestaltete. Daraus ergaben sich dann zwei verschiedene Ansätze für eine Packheuristik, einen in dem beide Verfahren strikt seriell nacheinander ausgeführt werden und einen in dem während des Belademechanismus des Ceschia Verfahrens bereits entstandene Freiräume gefüllt werden. Abschließende Tests zeigten, dass durch beide Ansätze vereinzelt bessere Lösungen generiert werden können, insgesamt aber nicht die Ergebnisse der Ceschia Gruppe erreicht werden.
Es gibt dabei natürlich noch einige Dinge, die für die entwickelten Verfahren optimiert werden könnten, aber durch den begrenzten Zeitrahmen innerhalb dem diese Bachelorarbeit stattfand, nicht mehr umgesetzt werden konnten. Einerseits wäre eine Optimierung der Verfahren zugunsten deren Laufzeit sinnvoll, um schneller Tests abhandeln zu können und somit auch in der gleichen Zeitspanne wie zuvor, der lokalen Suche mehr Suchschritte zu genehmigen. Dadurch könnten eventuell noch bessere Lösungen gefunden werden. Vor allem die verschachtelte Kombination ist noch sehr langsam. Andererseits gibt es auch noch die eine oder andere funktionelle Anpassung die getroffen werden kann. Zum Beispiel könnte für den Loader die nächste zu verladende Box nach einem *Best-Fit* Prinzip ausgewählt werden, oder für den verschachtelten Ansatz ein gezieltes Ersetzen einer Box, nach dem Ausdehnen und Ersetzten Prinzip aus der Arbeit von Zhang et al., umgesetzt werden. Des Weiteren bietet die Stapelbildung noch viel Potential, da die Grundfläche des Containers bereits sehr dicht beladen wird. Hier wäre das Platzieren unterschiedlicher Boxen auf einer Ebene des Stapels denkbar, oder das Verwenden von Freiräumen auch auf Stapeln, um zu einem späteren Zeitpunkt dort noch Boxen auflegen zu können.

Abbildungsverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] S. Ceschia, A. Schaerf *: Local search for a multi-drop multi-container loading problem.* Journal of Heuristics 19:275-294 April 2013

[2] Z. Zhang, S. Guo, W. Zhu, W. Oon, A. Lim *: Space Defragmentation Heuristic for 2D and 3D Bin Packing Problems.* Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Spanien, 16-22 Juli 2011

[3] A. Moura, J. F. Oliveira *: A GRASP Approach to the Container Loading Problem.* IEEE Intelligent Systems 20:50-57, 2005

[4] T.G. Crainic, G. Perboli, R. Tadei *: Extreme Point-Based Heuristics for Three-Dimensional Bin Packing.* INFORMS Journal on Computing 20:368–384, Juni 2008

[5] J.A. George, D.F. Robinson *: A heuristic for packing boxes into a container.* Computers & Operations Research 7:147–156, 1980

[6] E.E. Bischoff *: Three-dimensional packing of items with limited load bearing strength* European Journal of Operational Research 168:952-966, Februar 2006

[7] A. Bortfeldt, G. Wäscher *: Constraints in container loading - A state-of-the-art review* European Journal of Operational Research 229:1-20, Dezember 2012

[8] S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi *: Optimization by Simulated Annealing.* Science 220:671-680, 13. Mai 1983

8. Anhang

A. Ergebnisse der Testklasse TK2

Instanz	CS	CS_TV	TV2
CS2002	629.49	181.60	182.50
CS2843	127.90	141.30	143.50
CS2914	1171.30	1228.10	1218.50
CS2925	88.80	92.60	91.50
CS2988	518.90	537.80	520.30
CS3043	3516.90	3522.90	3509.30
CS3048	144.50	164.30	156.60
CS3056	728.95	552.00	552.50
CS3107	176.40	187.80	176.40
CS3220	1048.10	1040.60	1039.70
CS3253	451.60	514.00	500.40
CS3255	89.30	89.30	89.30
CS3258	736.37	738.80	1131.78
CS3291	180.00	178.60	182.70
CS3293	458.50	350.30	347.90
CS3296	250.80	273.70	274.80
CS3388	120.60	133.00	130.90
CS3403	109.70	122.60	120.10
CS3425	275.80	301.70	294.30
CS3469	169.90	253.60	247.00
CS3518	265.10	301.40	292.70
CS3556	162.30	176.90	171.30
CS3581	480.00	546.40	508.50
CS3583	620.90	700.50	689.40
CS3595	298.90	321.50	318.90
CS3598	356.70	1324.00	1318.60
CS3600	848.00	1186.38	1443.03
CS3606	717.00	826.60	791.70

Tabelle A.1.: Ergebnisse für die Instanzen der Klasse TK2 - Teil 1

Instanz	CS	CS_TV	TV2
CS3610	414.10	465.10	457.00
CS3625	1487.70	2057.20	2054.10
CS3636	243.10	269.70	271.10
CS3641	263.10	289.20	273.30
CS3651	1004.52	1112.60	1114.90
CS3659	414.80	457.30	448.40
CS3673	458.43	509.70	514.10
CS3695	131.80	145.40	147.60
CS3703	2064.80	2082.20	2080.60
CS3715	280.70	291.70	296.50
CS3719	367.70	1371.10	1381.40
CS3721	957.50	1029.60	1020.70
CS3727	575.00	637.20	633.90
CS3764	308.00	335.90	328.30
CS3766	303.00	313.60	322.50
CS3767	865.90	914.90	1111.72
CS3773	782.04	1554.10	1554.30
CS3778	942.20	1061.50	1057.80
CS3794	1597.30	2252.26	2614.60
CS3800	616.00	692.20	672.20
CS3815	1036.90	1072.20	1054.20
CS3849	442.40	480.80	483.80
CS3855	295.80	311.30	311.30
CS3857	777.90	902.90	902.20
CS3860	5284.20	5319.90	1443.03
CS3876	316.80	350.00	353.50
CS3884	547.60	665.40	684.10
CS3885	816.50	849.70	859.60
CS3894	485.50	545.10	537.60
CS3961	670.08	821.82	823.42
CS3962	797.60	889.00	880.10
CS3966	509.30	512.30	518.70
CS4765	305.50	340.20	332.10
CS4769	104.55	130.35	122.05
CS4924	84.15	120.55	115.65
CS4939	85.75	120.25	114.95
CS4960	92.35	108.65	103.25
CS4968	11.56	23.35	26.25
CS4972	49.45	66.45	68.65
CS4974	209.70	247.50	250.70
CS4975	214.20	264.10	322.82
CS4980	376.80	408.90	407.20
CS4981	81.45	92.95	92.45
CS4998	29.35	48.55	46.45
CS4999	150.10	187.40	174.10
CS5005	57.85	57.85	57.85
CS5006	155.50	226.20	232.95

Tabelle A.2.: Ergebnisse für die Instanzen der Klasse TK2 - Teil 2

∅ - Werte	**CS**	**CS_TV**	**TV2**
Kosten(€)	570,79	653,26	614,91
Abweichung(%)	0,00	1,12	0,60
Laufzeit(s)	61,65	15,93	51,97

Tabelle A.3.: Vergleich der unterschiedlichen Implementierungen und Verfahren für TK2