Volumenpackungen nach SAE J1100

Dissertation
zur Erlangung des Grades

"Doktor
der Naturwissenschaften"

am Fachbereich 08 – Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität
in Mainz

Dipl.-Inf. Tobias Baumann
geb. am 25.10.1980 in Schlema

Mainz, im August 2008

Tag der mündlichen Prüfung: 27. November 2008

Zusammenfassung

In dieser Arbeit werden neue Packalgorithmen für die US-Norm SAE J1100 vorgestellt. Das diskrete Packproblem ist NP-schwer und hat hier einen direkten Bezug zur Fahrzeugentwicklung. Die Algorithmen sind exakte Verfahren für das Maximum Weighted Independent Set Problem (MWIS). Wir beschreiben eine Methode, mit der ein großer Anteil der Knoten eines Graphen entfernt werden kann, ohne dass davon die optimale Lösung des MWIS-Problems beeinträchtigt wird. Mit Hilfe dieses Verfahrens wird auch ein kontinuierlicher Graph für das Packproblem definiert.

Das MWIS-Problem selbst wird mit Hilfe eines Aufzählungsalgorithmus exakt gelöst. Dieser Algorithmus nutzt die Vorgaben der US-Norm SAE J1100 zur Berechnung guter oberer Schranken. Wir vergleichen die erzielten Packungen mit denen manueller Packprozesse. Weiterhin untersuchen wir die verwendeten Aufzählungsschemata hinsichtlich ihrer Laufzeit und vergleichen sie mit weiteren Algorithmen aus der Literatur.

Da bei einer gitterbasierten Packung Lücken auftreten können oder Quader sich überschneiden, beschreiben wir einen weiteren Algorithmus, der Quaderpackungen mit beliebigen Platzierungen erzeugen kann. Dieser Packalgorithmus verwendet approximierte Minkowskisummen, die die Vereinigung vieler gleich orientierter und gleich großer Quader darstellen. Hierfür wird ebenfalls ein neuer effizienter Algorithmus vorgestellt, der auch die Verwaltung der Minkowskisummen während des Packens übernimmt.

Diese Algorithmen werden erweitert, so dass beliebige Objekte gepackt werden können.

Abstract

We present new algorithms to approximate the discrete volume of a polyhedral geometry using boxes defined by the US standard SAE J1100. This problem is NP-hard and has its main application in the car design process. The algorithms produce maximum weighted independent sets on a so-called conflict graph for a discretisation of the geometry. We present a framework to eliminate a large portion of the vertices of a graph without affecting the quality of the optimal solution. Using this framework we are also able to define the conflict graph without the use of a discretisation.

For the solution of the maximum weighted independent set problem we designed an enumeration scheme which uses the restrictions of the SAE J1100 standard for an efficient upper bound computation. We evaluate the packing algorithms according to the solution quality compared to manually derived results. Finally, we compare our enumeration scheme to several other exact algorithms in terms of their runtime.

Grid-based packings either tend to be not tight or have intersections between boxes. We therefore present an algorithm which can compute box packings with arbitrary placements and fixed orientations. In this algorithm we make use of approximate Minkowski Sums, computed by uniting many axis-oriented equal boxes. We developed an algorithm which computes the union of equal axis-oriented boxes efficiently. This algorithm also maintains the Minkowski Sums throughout the packing process.

We also extend these algorithms for packing arbitrary objects in fixed orientations.

Ausführliche Zusammenfassung

Bei der Kaufentscheidung für ein neues Auto spielt neben anderen Dingen die Kofferraumkapazität eine große Rolle. Es ist zwar möglich, das kontinuierliche Kofferraumvolumen mit Hilfe von CAD-Systemen einfach zu bestimmen. Allerdings möchte der Kunde normalerweise größere Objekte, so zum Beispiel Koffer oder Getränkekisten, in den Kofferraum laden. Um diesen Sachverhalt abzubilden, gibt es zwei verschiedene Normen, das Kofferraumvolumen zu bestimmen: In Europa ist die DIN 70020 [27] verbreitet, nach der das Volumen mit Hilfe von uniformen Quadern der Größe $200 \times 100 \times 50$ mm bestimmt wird. In den USA hingegen gibt es die Norm SAE J1100 [45]. Diese Norm verlangt, dass Quader verschiedener Größe in einer bestimmten Reihenfolge mit fest vorgegebenen Maximalvorkommen in den Kofferraum geladen werden müssen. Im Gegensatz zum kontinuierlichen Volumen ist das diskrete Packproblem ein NP-schweres Problem.

Bisher hatten Autohersteller nur die Möglichkeit, CAD-Systeme für die Berechnung von Kofferraumpackungen zu nutzen. Allerdings sind diese Methoden sehr zeitraubend und ineffizient, da die Packungen komplett manuell hergestellt werden mussten. Es bestand auch kaum die Möglichkeit, noch in den Entwicklungsprozess des Fahrzeugs einzugreifen, um etwa Regionen mit großen Verschnitt zu modifizieren. Mit Hilfe von effizienten Algorithmen für dieses Packproblem ist es nun möglich, das Kofferraumvolumen frühzeitig in der Entwicklung eines Fahrzeugs zu bestimmen und mit entsprechenden Modifikationen besser auf die Bedürfnisse der Kunden abzustimmen.

In Kooperation mit einem großen deutschen Automobilhersteller haben wir ein Softwarepaket entwickelt, mit dem dreidimensionale Packprobleme, speziell für Quader, effizient gelöst werden können. Die Software kann CAD-Daten verarbeiten und berechnet Packungen nach den oben beschriebenen Normen. Zusätzlich gibt es die Möglichkeit, beliebige Geometrien in einen Kofferraum zu packen, zum Beispiel Getränkekisten oder Koffer.

Diese Arbeit befasst sich mit neuen Packalgorithmen für die US-Norm SAE J1100. Die Algorithmen basieren auf Graphen und berechnen unabhängige Mengen mit maximalem Gewicht (MWIS – Maximum Weight Independent Set). Der so genannte Konfliktgraph kann aus einer Gitter-Diskretisierung des Kofferraums gewonnen werden [48]. Da die Graphen für das gewichtete Packungsproblem sehr groß werden können, beschreiben wir ein Verfahren, mit dem ein großer Anteil der Knoten eines Graphen entfernt werden kann, ohne das Ergebnis der optimalen Lösung zu beeinflussen. Mit Hilfe dieses Verfahrens sind wir ebenfalls in der Lage, einen Konfliktgraphen ohne die Verwendung eines Gitters zu erzeugen.

Zur Lösung des MWIS-Problems verwenden wir einen Aufzählungsalgorithmus, der die Vorgaben der US-Norm ausnutzt, um eine gute obere Schranke zu berechnen. Wir vergleichen die erzielten Packungen mit denen manueller Packprozesse. Weiterhin untersuchen wir die verwendeten Aufzählungsschemata hinsichtlich ihrer Laufzeit und vergleichen sie mit anderen exakten Algorithmen aus der Literatur.

Weiterhin beschreiben wir einen Packalgorithmus mit beliebigen Platzierungen und festen Orientierungen. Innerhalb der US-Norm ist dies ein enormer Fortschritt, da gitterbasierte Packungen entweder Lücken aufweisen oder sich Quader überschneiden

können. Mit beliebigen Platzierungen der Quader wird dies umgangen, und bessere Packungen sind möglich. Dieser Algorithmus verwendet approximierte Minkowskisummen, die die Vereinigung vieler gleich orientierter und gleich großer Quader darstellen. Hierzu beschreiben wir ebenfalls einen neuen Algorithmus, der zusätzlich die effiziente Verwaltung der Minkowskisummen während des Packens übernimmt.

Extended Abstract

The decision for a new car is based amongst others on the capacity of its luggage compartment. Using modern CAD systems, it is only possible to compute the continuous volume. Usually the customer wants to pack some large objects, such as suitcases or bottle crates, into the trunk compartment. For this issue there are two different standards defined to compute the volume of a trunk: First, the German standard DIN 70020 [27], which is used throughout the European Union, uses small boxes of sizes $200 \times 100 \times 50$ mm and demands to pack as many boxes as possible into the trunk. The second standard, SAE J1100 [45], which is used in the United States of America, requires several different box types to be packed. In contrast to the continuous volume it is an NP-hard problem to compute optimal packings with discrete objects.

Up to now, car manufacturers use CAD systems to compute trunk packings manually with time-consuming and inefficient methods. As well, the trunk capacity is computed very late during the car design process. If it is known earlier in the car design process, one can use this knowledge to adapt the trunk in order to use wasted space for other items such as electricity cables. In other places the trunk could be extended in order to place some more items. This of course requires efficient algorithms to produce good packings.

We developed a software package which can handle these three-dimensional packing problems. This package can process CAD input data and computes packings according to the standards mentioned above efficiently. As well, we integrated an algorithm to pack arbitrary geometries into the trunk.

In this dissertation we present new algorithms to compute valid packings according to the US standard SAE J1100. The algorithms are graph-based and produce maximum weighted independent sets on a so-called conflict graph. This graph can be derived from a grid discretisation [48] of the trunk space. We present a framework to eliminate a large portion of the vertices of a graph without affecting the quality of the optimal solution. Using this framework we are also able to define the conflict graph without the use of a discretisation.

For the solution of the maximum weighted independent set problem we designed an enumeration scheme which uses the restriction of the SAE J1100 standard for an efficient bound computation. We evaluate the packing algorithms according to the solution quality compared to manually derived results. Finally, we compare our enumeration scheme to several other exact algorithms in terms of their runtime.

We also present an algorithm which can compute box packings with arbitrary placements and fixed orientations. For the SAE J1100 standard this is an improvement over

the grid discretisation of the trunk volume because all generated packings are tight and valid, in contrast to grid-based packings which either tend to be not tight or have intersections between boxes. In this algorithm we make use of an approximate Minkowski Sum, computed by uniting many axis-oriented equal boxes. We developed an algorithm which computes the union of equal axis-oriented boxes efficiently. This algorithm also calculates changes of the Minkowski Sum created by the addition or deletion of boxes to a packing efficiently.

Danksagung

Zuerst danke ich meinem Doktorvater für seine engagierte Unterstützung und Anleitung. Durch ihn kam ich in das Gebiet der Packprobleme. Seine Anregungen sind immer von großem Nutzen gewesen. Ebenso danke ich meinem Zweitkorrektor für seine Kooperation und einen privaten Packwettbewerb.

Meinen besonderen Dank spreche ich meinen Kollegen vom Institut für Informatik in Mainz aus. Aus der mittäglichen Kaffeerunde habe ich so manche Idee mit zurück ins Büro getragen. Speziell die Mitarbeiter am Trunkpacker-Projekt sowie unsere Vorgänger am MPI für Informatik in Saarbrücken konnten mir in schwierigen Punkten gut weiter helfen.

Nicht zuletzt danke ich meiner Frau und meiner Familie, die mich immer moralisch unterstützt haben.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

The decision for a new car is based amongst others on the capacity of its luggage compartment. Using modern CAD systems, it is possible to compute the continuous volume. This value, of course, does not reflect the special shape of the cargo bay. Usually the customer wants to pack some large objects, such as suitcases or bottle crates, into the trunk compartment. So it would be appropriate to pack solid objects to ensure some sort of comparability depending on the shape and capacity of the compartment. For this issue there are two different standards defined to compute the volume of a trunk: First, the German standard DIN 70020 [27], which is used throughout the European Union, uses small boxes of sizes $200 \times 100 \times 50$ mm and demands to pack as many boxes as possible into the trunk. The second standard, SAE J1100 [45], which is used in the United States of America, requires several different box types to be packed. Both of these standards have their eligibility but for both standards one can discuss their usefulness. For example, one could disagree whether a set of uniform boxes with constant size would reflect a "normal" use of the luggage compartment. On the other hand, would the packing of abstract cuboid boxes without paying attention to orientation requirements be better?

Nevertheless, a comparative standard is needed, and both approaches reduce the complex structure of the trunk geometry to a scalar number which can be easily compared.

In contrast to the continuous volume it is an NP-hard problem to compute optimal packings with discrete objects (see [48] and chapter 2). Up to now, car manufacturers use CAD systems to compute trunk packings. This method is very time-consuming because the user has to find good placements for each item, rotate each object such that it fits optimally, and ensure that the packed objects do not intersect. All of these actions have to be performed manually by an experienced engineer. Afterwards the found packing has to be re-packed in a real car model. The whole procedure takes generally more than a day of valuable engineer's time. Within this time it is only possible to find at most two packings. Of these, none can be guaranteed to be optimal.

Another aspect is the redesign of the luggage compartment according to the found packings. If the trunk capacity is known early in the car design process, one can use this knowledge to adapt the trunk in order to use wasted space for other items such as electricity cables. In other places the trunk could be extended in order to place some more items. This of course requires efficient algorithms to produce good packings.

A large German car manufacturer asked our team to develop a software package which can handle these three-dimensional packing problems. This package has to process CAD input data and to compute packings according to the standards mentioned above efficiently. An additional requirement was to integrate an algorithm to pack arbitrary geometries into the trunk. In this case, some additional constraints must be defined,

such as restrictions to the orientation of some objects.

The first phase of our project dealt with geometrical issues such as the import of CAD data and with packings according to the German standard DIN 70020. Since the boxes have good side length relations $(4 : 2 : 1)$, it is possible to specify a variety of discrete algorithms. A detailed overview on discretised algorithms is shown in [48]. These algorithms include the search for an Independent Set in a so-called conflict graph using different methods like local search, a modified matching algorithm and the formulation of linear programs.

Additionally, there have been approaches with continuous packings, that means the coordinates and orientations of the boxes are not limited to a grid-like structure. These algorithms use Simulated Annealing to improve on an existing packing which may be the solution of a discrete algorithm (see [35, 49] for details).

Using these approaches, it has been possible to meet the specifications made for our cooperation (99% of the volume an experienced engineer can achieve manually). From this project phase several major publications arose [21, 22].

This dissertation is focused on the second part of the project. The existing software had to be extended to be capable of packings according to the SAE J1100 standard. Additionally, it should be possible to pack objects with arbitrary geometries.

As before, discretised and continuous algorithms should be used, and the resulting packings have to be at least as good as the volume found by an experienced engineer manually. The algorithms should find an acceptable solution within at most 24 hours.

1.1 Previous Work

In addition to the publications resulting from our project [21, 22, 35, 48, 49], another algorithm according to the DIN trunk packing problem was developed in [44]. There the trunk is divided into layers of equal width, large enough to fit a box of "unit" size, which is a cube whose side length is the smallest box side length. Then the packing problem is reduced to a two-dimensional problem with additional constraints if boxes are standing upright and therefore occur in two or four adjacent layers.

The US standard SAE J1100 has been explored in [11]. Due to the inhomogeneity of the box sizes, other methods are necessary to find good packings. In [11], an extended pattern search algorithm is used, similar to simulated annealing methods. Some preliminary results of this dissertation have already been published in [1].

In this dissertation, the Trunk Packing Problem will be approximately solved using graph-based algorithms. Therefore, some previous results on graph-related problems are shown below. The problem itself can be transformed into a Weighted Independent Set Problem (MWIS, see section 2.2). Various approximation algorithms are known for this generalisation of the Independent Set Problem (IS). For example, [13, 14, 28, 30, 37, 38] present and analyse several approximation algorithms for the MWIS Problem on various graph classes such as permutation graphs, trees, sparse random graphs, fork-free graphs etc. In [34, 53], approximation algorithms for general weighted graphs are given. Finally, [55, 56] show branch-and-bound algorithms for the MWIS

Type	Allowed number	Size ″ l	w	h	Size mm l	w	h	Volume l	ft^3
A	4	24	19	9	610	483	229	67.47	2.375
B	4	18	13	6.5	457	330	165	24.88	0.880
C	2	26	16	9	660	406	229	61.36	2.167
D	2	21	18	8.5	533	457	216	52.61	1.859
E	2	15	9	8	381	229	203	17.71	0.625
F	2	21	14	7	533	356	178	33.78	1.191
H	20	12.8	6	4.5	325	152	114	5.63	0.200

Table 1.1: Box types

Problem. Closely related to the MWIS problem is the Weighted Clique Problem (MWC). One can obtain a MWC Problem by using the complement graph of an MWIS Problem. For the MWC Problem some algorithms are given in [5, 6, 7, 46]. A detailed analysis of MWIS- and MWC-algorithms is given in section 2.2.

The more-dimensional packing problem is a generalisation of the Knapsack problem [18]. An extensive overview on publications about this and related problems is given in [19]. Many publications deal with packing in two-dimensional space. Fowler et al. show the NP-completeness of packing problems in the plane [25]. A fairly simple packing problem can be found by placing non-overlapping disks of different radii into a circle [61] or squares into squares [23]. A great variety of two-dimensional packing problems can also be found in [26].

Avnaim and Boissonat [3, 4] deal with the simultaneous placement of polygons without translation. Daniels and Milenkovic investigated two-dimensional arrangements of polygons into a polygonal container and present several algorithms for exact and approximate placements of polygonal objects into a polygonal container without rotation [15, 16, 17, 39, 40, 41]. They used Minkowski Sums [42] to determine valid placements of the objects. These approaches are examined closer in chapter 3.

Going back to three dimensions, Cagan et al. present several algorithms for packing or layout problems in [12]. Fekete and Schepers published several approaches for simple container geometries and fixed orientations, such as a cube of unit length [24].

1.2 Specification of SAE J1100

The SAE J1100 standard requires to use boxes of different sizes to determine the baggage capacity of a car trunk. The box sizes are shown in Table 1.1 and displayed in Figure 1.1. There also exists an object of type G which is an irregularly shaped golf bag. This bag is omitted from the SAE packing problem, but it can be included in the packing of arbitrary geometries into a trunk.

For packing these boxes one has to fulfil several requirements. For each box type there is an allowed maximal number of occurrences displayed in the second column. The packing process itself is restricted as follows: At first, the large boxes of types A-F have

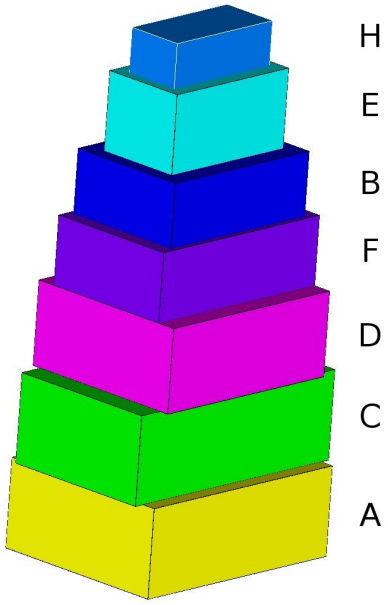


Figure 1.1: The set of SAE box types

to be packed such that no more boxes of these types fit into the trunk. Only afterwards the H-boxes may be added.

If all boxes are used, it is allowed to add another sets of boxes. This case only occurs when packing into vans and large vehicles since the total volume of all boxes sums up to 28 ft^3 which is roughly 3000 litres. For these vehicles the luggage capacity is measured in a different way. Therefore we can restrict the packings to a single set of boxes.

As in all packing problems, the goal is to find a packing covering the largest possible volume with respect to the SAE standard requirements.

1.3 Organisation of this dissertation

Chapter 2 deals with the discretisation of the trunk. In section 2.2, the space discretisation is used to formulate graph-based algorithms to pack boxes of different sizes according to SAE J1100. The advantages and disadvantages of the grid parameters are discussed in section 2.1.2. Afterwards, chapter 3 shows a possibility to omit the space discretisation in order to obtain better packing results. In chapter 4 the implementation of the algorithms presented in chapters 2 and 3 is evaluated and compared to the results obtained by manual packing processes. The packing algorithms of chapter 2 are extended to the packing problem of arbitrary geometries into the trunk in chapter 5. Finally, chapter 6 shows a short survey over the presented algorithms, inaugurates some

possibilities for improvements and states several open problems relating packings.

2 Packing Algorithms on Grids

2.1 Constructing a Grid

When searching for a discretised variant of a geometrical problem, one naturally thinks of discrete positions and fixed orientation of the objects. In the trunk packing problem, the container is divided into a grid of equal cubes. The box positions are now restricted to points of this grid. Additionally, the orientation of the boxes is limited to the six main axis-aligned orientations. So the set of possible placements is reduced from infinitely many to a finite – maybe even rather small – size.

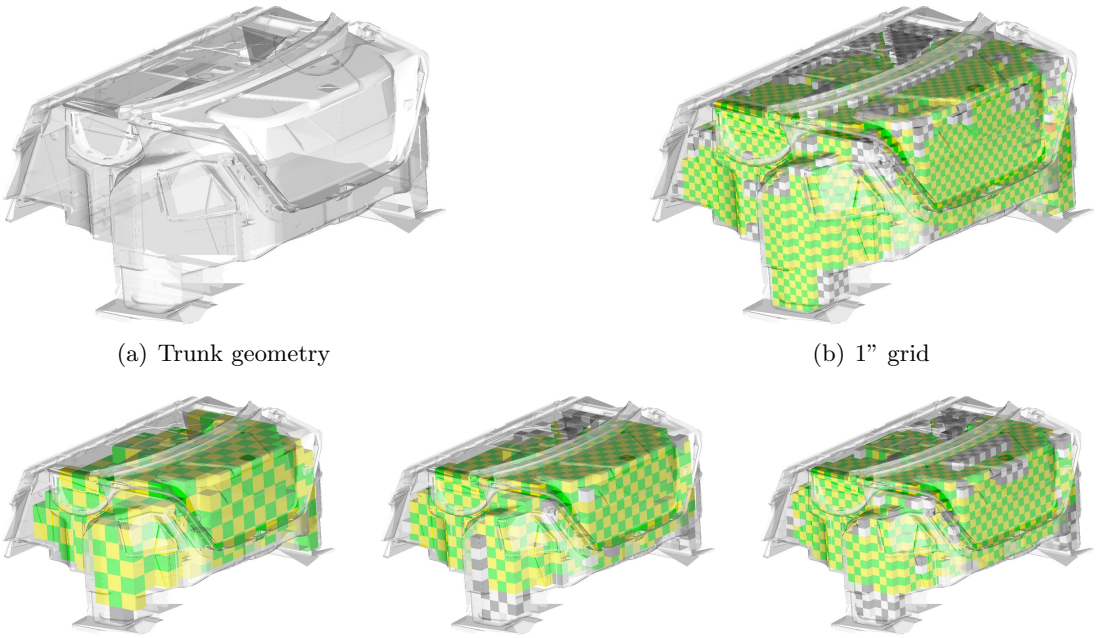
The grid consists of cubes with side length s. These cubes generally occur in three different types: `inside`, `outside` and `boundary`. A box can only be placed to cover exclusively `inside`-cells. Since some of the `inside`-cells can not be covered by a box, we can distinguish between `usable` and `unusable` cubes. Especially all `boundary`- and `outside`-cubes are marked as `unusable`.

Definition 2.1. *Grid-related terms*
A grid is defined by its *origin* $O = (x_0, y_0, z_0) \in \mathbb{R}^3$ and *spacing* $s \in \mathbb{R}$. The orientation of the grid is defined by three orthonormal vectors $\mathbf{x}, \mathbf{y}, \mathbf{z}$. So the space $\mathbb{R}^3$ is divided into cubes with side length s. A cube c can be referred to by its indices (i, j, k), which means the lower left corner of this cube is placed at the coordinates $(i \cdot \mathbf{x} + j \cdot \mathbf{y} + k \cdot \mathbf{z}) \cdot s$ in relation to O. The set of cubes which lie completely within the trunk geometry is denoted `inside`. All cubes intersecting the trunk geometry form the set `boundary`, and finally the cubes completely outside the trunk geometry shall be denoted `outside`. The set `inside` can be divided into the two sets `usable` and `unusable`, qualifying cubes that can be covered resp. can never be covered by a box.

Definition 2.2. *Placing boxes on a grid*
An SAE box on the grid is defined by three parameters: First, the *type* t of the box. Second, the *orientation* o, and at last the position of the box. The orientation of the box is an integer ranging from 1 to 6, indicating the permutation of the three axis coordinates. The position is given as the grid cube c with the smallest coordinates, that means the lower left cube. Hence the box b is completely described by the triple $b = (t, o, c) = (t, o, (i, j, k))$.

The construction and optimisation of the grid was presented first in [22] and is described in detail in [48]. In section 2.1.2 an additional technique for optimising the grid structure is given.

Hence we can get for a given trunk geometry grids with different levels of detail. Figure 2.1 shows grids with different spacings for a given trunk geometry.



(a) Trunk geometry (b) 1” grid

(c) 3” grid (d) 2” grid (e) 1.5” grid

Figure 2.1: For the given trunk geometry, we can create grids with different spacings representing different levels of detail.

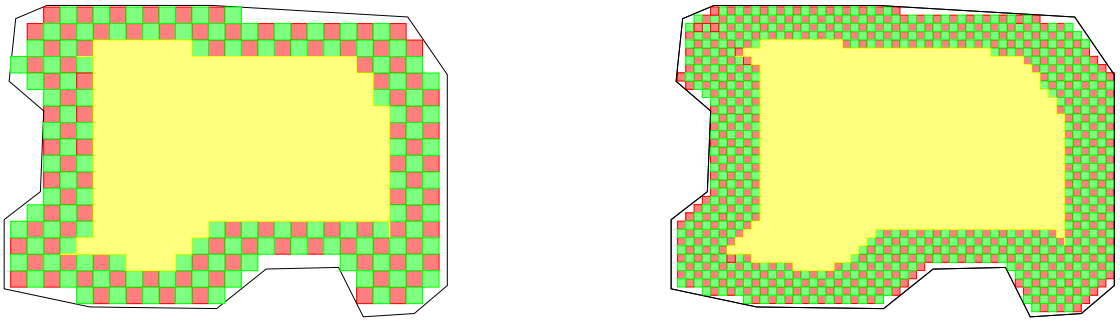


Figure 2.2: Two grids with different spacings for the same trunk

2.1.1 Grid Spacing – 2″, 3″

For constructing the grid, it is necessary to find a grid spacing s. The ideal choice would be such that s is a divisor of all possible side lengths of boxes. Unfortunately, the boxes of the SAE standard have got 17 different side lengths with a greatest common divisor (GCD) of $0.1''$ (see Table 1.1).

The grid spacing determines the problem size. On one hand, a small spacing would represent the trunk geometry very well, but the number of grid cells and therefore the size of the algorithm data structure (a so-called conflict graph – see Definition 2.4) will be raised dramatically. On the other hand, a large grid spacing will result in a small conflict graph, but there are large deficits in the representation of the trunk geometry.

Figure 2.2 illustrates the impact of the grid spacing to the exactness of the geometry representation. As can be seen, the larger grid spacing does not represent the corners of the trunk very well, and boxes that may fit there (according to the smaller grid spacing) can not be represented. This would result in a potentially smaller packing.

As mentioned above, the spacing of the grid is determined to be at most $0.1''$ in order to represent the boxes correctly. If H-boxes are left out, the GCD of the box side lengths is $0.5''$. The trunk of a typical mid-class car would consist of $\approx 2 \cdot 10^5$ grid cells with spacing $0.5''$. Hence the corresponding conflict graph would consist of up to $2.5 \cdot 10^6$ vertices. Even the creation of this conflict graph would exceed the time limit given by our cooperation partner. Table 2.1 shows a survey on the development of the grid spacing compared to the problem size. However, if the cell spacing is increased, some of the boxes will not fit onto the grid cells. Figure 2.3 illustrates this for a two-dimensional grid. In the upper part of the figure the three boxes do not fit onto the large grid spacing. If the grid spacing is halved, one box fits onto the grid. Of the other two objects each has got one side which won't fit, but the other side length does fit onto the smaller grid.

So we have to decide what to do with non-fitting side lengths. If these side lengths are rounded up (the boxes are considered larger than they really are), some space between the packed boxes is wasted. On a rough grid this leads to small gaps between the boxes where nothing can be placed. It turns out that with enlarged boxes the resulting packings are far worse than the comparative packings of our industrial partner.

However, if the side lengths are rounded down (the boxes are considered smaller than

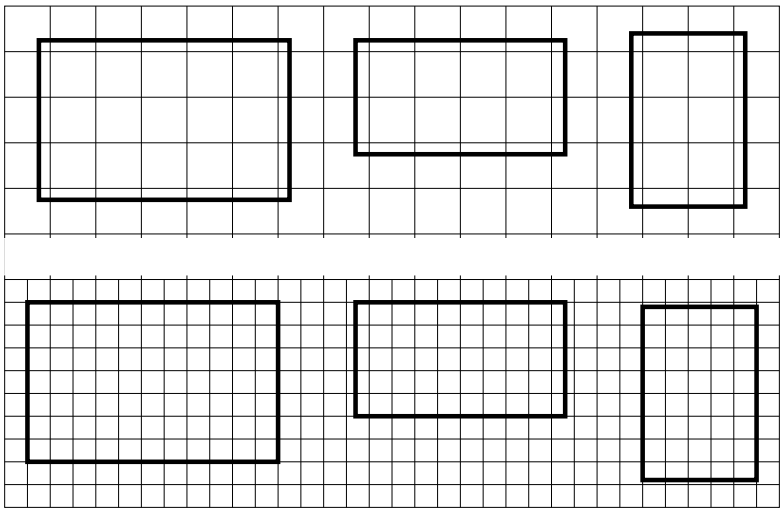


Figure 2.3: If the grid spacing is not a divisor of a box side length, the box will not fit onto the grid. Top: three boxes on a 10 cm grid, bottom: the same boxes on a 5 cm grid.

Grid spacing	# `inside`-cells	# non-fitting side lengths
$3''$	720	11
$2''$	2 517	13
$1.5''$	6 674	10
$1''$	23 737	4
$0.75''$	57 771	10
$0.5''$	$\approx$ 200 000	1
$0.1''$	$\approx$ 25 000 000	0

Table 2.1: Grid spacings: Overview. Seven box types lead to 21 side lengths.

Grid spacing	A-Box	B-Box	C-Box	D-Box
$3''$	$3 \times 6 \times 8$	$2 \times 4 \times 6$	$3 \times 5 \times 8$	$2 \times 6 \times 7$
$2''$	$4 \times 9 \times 12$	$3 \times 6 \times 9$	$4 \times 8 \times 13$	$4 \times 9 \times 10$
$1.5''$	$6 \times 12 \times 16$	$4 \times 8 \times 12$	$6 \times 10 \times 17$	$5 \times 12 \times 14$
$1''$	$9 \times 19 \times 24$	$6 \times 13 \times 18$	$9 \times 16 \times 26$	$8 \times 18 \times 21$
$0.75''$	$12 \times 25 \times 32$	$8 \times 17 \times 24$	$12 \times 21 \times 34$	$11 \times 24 \times 28$
$0.5''$	$18 \times 38 \times 48$	$13 \times 26 \times 36$	$18 \times 32 \times 52$	$17 \times 36 \times 42$
$0.1''$	$90 \times 190 \times 240$	$65 \times 130 \times 180$	$90 \times 160 \times 260$	$85 \times 180 \times 210$

Grid spacing	E-Box	F-Box	H-Box	max. degree
$3''$	$2 \times 3 \times 5$	$2 \times 4 \times 7$	$1 \times 2 \times 4$	27028
$2''$	$4 \times 4 \times 7$	$3 \times 7 \times 10$	$2 \times 3 \times 6$	99642
$1.5''$	$5 \times 6 \times 10$	$4 \times 9 \times 14$	$3 \times 4 \times 8$	268280
$1''$	$8 \times 9 \times 15$	$7 \times 14 \times 21$	$4 \times 6 \times 12$	1021474
$0.75''$	$10 \times 12 \times 20$	$9 \times 18 \times 28$	$6 \times 8 \times 17$	2449096
$0.5''$	$16 \times 18 \times 30$	$14 \times 28 \times 42$	$9 \times 12 \times 25$	8698236
$0.1''$	$80 \times 90 \times 150$	$70 \times 140 \times 210$	$45 \times 60 \times 128$	1134233436

Table 2.2: Box sizes on different grid spacings.

they are), the boxes can intersect. But the grid has got a large spacing s and therefore the deficits in the representation of the trunk geometry are rather large, and maybe the computed solution will nevertheless fit into the real trunk. Anyway, this opportunity should be handled with care since it is not assured that the boxes will fit into the trunk: The intersections can be too severe to be resolvable just by the loss created by the grid spacing. For example, in a grid with $s = 3''$, two E-boxes might be placed next to each other and overlap by $2.5'' = 6.35$ cm. This is not resolvable when an optimised grid is used because of rather small distances between the outer `inside` cubes and the trunk geometry.

So we make use of a trick: We chop the outmost shift of `inside`-cells from the grid. That means we remove for each coordinate pair $(i, j), (i, k), (j, k)$ the `inside`-cells with largest k-, j- or i-coordinate. In most cases this suffices to create a valid packing from the result.

In Table 2.2 the box sizes are shown for several grid spacings. The sizes are measured in grid cubes; the last column shows the theoretical maximum degree of a vertex in a conflict graph for the corresponding grid spacing. A grid with spacing $0.1''$ provides an exact representation of the boxes and the best approximation of the trunk geometry, but it also leads to a large conflict graph and it is not possible to store a representation of this graph, not to mention running an algorithm on this graph. As shown in Table 2.1, a small grid spacing easily leads to more than 10^5 `inside`-cells. In a conflict graph, each cell could be an anchor cube for a box. That means we would have for each cell potentially 42 (seven box types, six orientations) boxes with this cell as anchor cube.

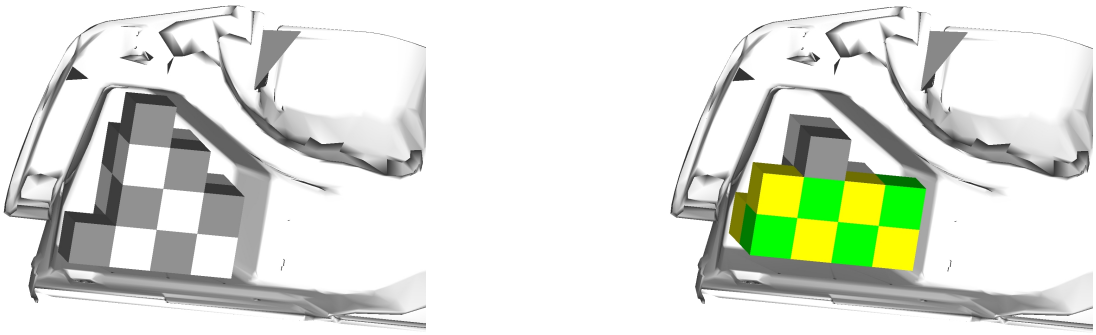


Figure 2.4: A small pocket with two possible grid positions

2.1.2 Optimising the Grid

The quality of the grid-based packing depends on the grid itself. The origin of the grid can be translated, and the grid orientation itself can be modified. According to these parameters, we have to find a grid placement such that a large packing is possible. For example, a small pocket especially designed for an H-box is unusable if the grid does not supply the corresponding `inside` cells. Figure 2.4 shows such a pocket. An H-box on this grid would consist of $1 \times 2 \times 4$ grid cubes. In the left picture, there is no orientation such that an H-box would fit at all. The right picture shows a slightly moved grid where an H-box would fit in two possible orientations.

There are two variables which have an influence on the quality of the grid:

- the position of the grid origin,
- the orientation of the grid axes.

The origin of the grid can be chosen freely within a cube with side length s. The axis orientation is mostly defined by the axes of the car itself. In some cases, however, it might be better to use other orientations in order to align the boxes with a side pocket or the spare wheel compartment. The best choice for the grid origin and orientation is an alignment to the load floor, otherwise a large amount of space will be wasted.

Unfortunately, an unambiguous description of a "good" grid is still missing. As described above, the grid should allow a large packing of boxes. To verify this condition, one would have to compute the optimal packing for each grid and compare the grids according to this criterion. This is impossible since it requires to solve the packing problem itself.

Another measurement is the number of usable `inside`-cells of the grid. This number gives an upper bound for the optimal packing on this grid. Although not all `inside`-cells are used in an optimal solution, a larger number of usable cells increases the probability for a larger solution. This measurement can be easily computed and should suffice for the grid optimisation.

In the following three strategies are described to increase the number of usable `inside`-cells.

Algorithm 2.1 Exhaustive Grid Optimisation

Let $o_0 = (x_0, y_0, z_0) \in \mathbb{R}^3$ be the origin of the grid
for ($x = 0; x \leq s; x = x + \Delta$)
  for ($y = 0; y \leq s; y = y + \Delta$)
    for ($z = 0; z \leq s; z = z + \Delta$)
      $o = o_0 + (x, y, z)$
      Calculate $n_{(x,y,z)}$, the number of usable `inside`-cells
      if ($n_{(x,y,z)} > n_{\max}$)
        $o_{\max} = o$
        $n_{\max} = n_{(x,y,z)}$

Algorithm 2.2 Randomised Grid Optimisation

Let $o_0 = (x_0, y_0, z_0) \in \mathbb{R}^3$ be the origin of the grid.
while (arbitrary constraint, for example a time limit)
  Choose $(x, y, z) \in (0, s)^3$ randomly from a uniform distribution
  $o = o_0 + (x, y, z)$
  Search for a local optimum (x', y', z') with a maximal number of usable `inside`-cells
  if ($n_{(x',y',z')} > n_{\max}$)
    $o_{\max} = o$
    $n_{\max} = n_{(x',y',z')}$

Algorithm 2.3 Exhaustive Grid Optimisation with Local Optimisation

Let $o_0 = (x_0, y_0, z_0) \in \mathbb{R}^3$ be the origin of the grid
for ($x = 0; x \leq s; x = x + \Delta$)
  for ($y = 0; y \leq s; y = y + \Delta$)
    for ($z = 0; z \leq s; z = z + \Delta$)
      $o = o_0 + (x, y, z)$
      Search for a local optimum (x', y', z') with a maximal number of usable `inside`-cells
      Calculate $n_{(x',y',z')}$, the number of usable `inside`-cells
      if ($n_{(x',y',z')} > n_{\max}$)
        $o_{\max} = o$
        $n_{\max} = n_{(x',y',z')}$

Algorithm 2.1 has been discussed in similar shape in [48]. The step size Δ can be chosen according to the grid spacing s. The runtime of this algorithm depends directly on the grid spacing s and on the ratio s/Δ: Clearly the number of `inside` cubes has to be counted ($O(s^{-3})$), and this has to be done for each position ($O(s^2/\Delta)$). Although this approach does not guarantee to find a grid with the maximal number of `inside` cubes, it will be sufficient for most applications.

The second one, Algorithm 2.2, usually finds a grid containing more usable `inside` cubes than Algorithm 2.1, mostly due to the local optimisation step in line 5. For this optimisation step any method can be used. The current implementation does well with a gradient-based optimisation. The runtime for this approach is in general unlimited, but it can be adjusted to the requirements of the task. Normally a fixed runtime of several minutes on a standard office workstation suffices to get a result with similar quality compared to a complete run of Algorithm 2.1. Unfortunately we can not guarantee that Algorithm 2.2 will produce a grid with more usable `inside` cubes than Algorithm 2.1 would, of course due to the randomly chosen initial points.

Algorithm 2.3 is a combination of the first two algorithms. It joins the exhaustive examination of many points within one cube with the local optimisation step of Algorithm 2.2. Due to this additional step the increment Δ can be enlarged such that only few (maybe 10^3 instead of 100^3) starting points are examined. This naturally leads to a trade-off between the runtime of the optimisation method and the iterations.

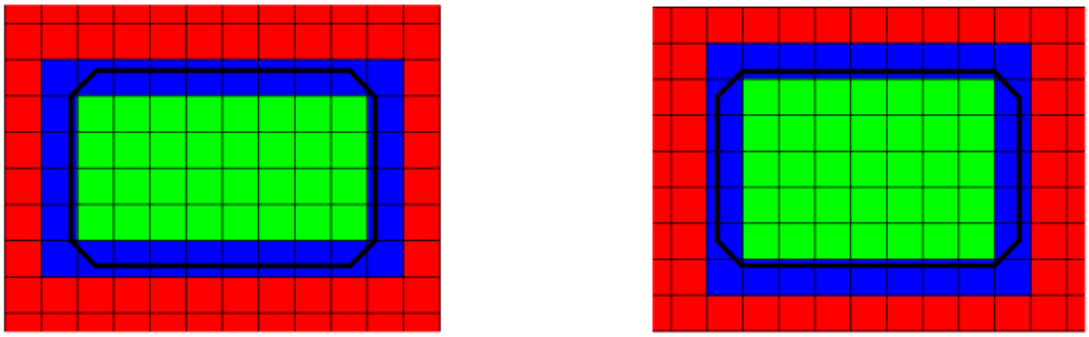
The number of usable `inside` cubes is not necessarily the best indicator for a good grid. One can create examples of trunk geometries where a grid with less `inside`-cells can hold more boxes than a grid with more `inside`-cells. Figure 2.5 shows this aspect for a small trunk geometry, as it has been presented in [48]. The left image shows a grid with 32 `inside`-cells which can hold 5 rectangles of dimension 4×2. The other image shows a grid for the same geometry with 35 `inside`-cells that can hold only 4 rectangles. So maybe other indicators have to be found to reflect the capacity of the grid. These indicators could include the surface of the grid, the compactness etc. Since the optimisation of grid discretisations is not the main topic of this thesis, we only give some possible aspects for further research.

2.2 Design of the Graph Problem

Using the grid, the number of possible placements for the boxes is reduced to a finite number. Hence we can transform the packing problem into a graph related problem:

We can restrict the possible orientations of a box to the permutations of the three coordinate axes. For each box type t, orientation o and position p we create a vertex in the conflict graph (see (2.1)). Here, o is one of the six main axis orientations, and p is the reference point of a cube in the specified grid. Since there are seven different box types, we get up to 42 vertices for each usable inside cell of the grid.

Two vertices are connected within the graph iff the corresponding boxes would overlap in a packing. This means we determine the set $C(v)$ of grid cubes that would be covered by the box specified by vertex v (see (2.2)). Afterwards we find for each grid cube



(a) 32 `inside`-cells, 4 rectangles

(b) 35 `inside`-cells, 3 rectangles

Figure 2.5: The number of usable `inside`-cells is not the only indicator for a good grid!

$c \in C(v)$ all vertices that also would cover c. An alternative formulation is given in (2.3).

Definition 2.3. Types and Orientations
For the SAE trunk packing problem, let $T = \{A, B, C, D, E, F, H\}$ be the set of box types, and $O : \{1, \ldots, 6\}$ be the set of the main axis orientations.

Definition 2.4. Conflict Graph
A Conflict Graph $G = (V, E)$ for the discretised packing problem is defined as follows:

$$V = \{(t, o, p) : t \in T; o \in O; p \in \texttt{usable}; \quad \text{Box of type } t \text{ and orientation } o \text{ fits at } p.\} \tag{2.1}$$

$$C(v) = \{c \in \texttt{inside} : c \text{ is covered by } v\} \tag{2.2}$$

$$E = \{\{u, v\} : u, v \in V; \exists c \in \texttt{inside} : c \in C(u) \wedge c \in C(v)\} \tag{2.3}$$

This graph has been presented in similar form in [48], although it has been called there "Intersection Graph" and was built for only one box type.

A box packing is valid iff there are no intersections between the boxes. Hence a valid packing corresponds directly to an Independent Set on the graph. The DIN case would ask for the maximum number of boxes that would fit into the trunk (Maximum Independent Set problem). Contrary, the SAE standard asks for the maximum *volume* that can be covered by the boxes. This corresponds to the Maximum Weighted Independent Set problem. A brief look at Table 1.1 reveals for example: a single A-box would cover a larger volume than a set of eleven H-boxes. We therefore extend the conflict graph G by a weight function $w : V \to \mathbb{R}$, representing the volumes of the box types.

Definition 2.5. Let $G = (V, E)$ be a conflict graph according to Definition 2.4. Let $w : V \to \mathbb{R}$ be defined as $w(v) = \mathit{volume}(t)$ for all $v = (t, o, p) \in V$. Then $G' = (V, E, w)$ is the conflict graph for the weighted (or SAE) packing problem.

The Maximum Weighted Independent Set problem (MWIS) is defined as follows:

Definition 2.6. Let $G = (V, E, w)$ be a graph with weight function $w : V \to \mathbb{R}$. An Independent Set $I \subseteq V$ is a set of vertices where $\forall u, v \in I : \{u, v\} \notin E$. The *weight* of I, $w(I)$ is defined to be the sum of the weights of all vertices of I: $w(I) = \sum_{v \in I} w(v)$. The task is to find an independent set of G with maximum weight.

Finding an MWIS is known to be an NP-hard problem [29]. The proof can be done by reducing the maximum unweighted independent set problem to the weighted case by setting all vertex weights to 1.

An additional restriction is given by the SAE standard: Every box type has a maximum number of occurrences in a single packing as shown in Table 1.1. This means that an optimal weighted independent set corresponds not necessarily to a valid packing according to the SAE standard. Now there are two possible extensions to the graph problem. First, one could modify the conflict graph such that only independent sets fulfilling the occurrence-requirement are considered as solutions. This construction would be rather complicated and might not keep the size of the original conflict graph. The second possibility is to extend the MWIS problem by a matching problem: For each box type t with n_t occurrences, add n_t vertices to the graph and connect these vertices with all conflict graph vertices of type t. The set of these added vertices is called R. Then a solution I to the MWIS problem would only be valid if there exists a perfect matching in the bipartite graph induced by the vertex set $I \cup R$, which is a matching of size $|I|$, since a trunk where the whole box set would fit would be measured in a different way.

So we can define the extended conflict graph H as follows:

Definition 2.7. Let $G = (V, E, w)$ be a conflict graph to the SAE trunk packing problem. Let $Z = \{a_{1..4}, b_{1..4}, c_{1..2}, d_{1..2}, e_{1..2}, f_{1..2}, h_{1..20}\}$ be a set of additional (type) vertices. Let E' be defined as follows: For each vertex $v \in V$ there exist edges $e \in E'$ connecting v with all its corresponding type vertices. Now, $H = (V \cup Z, E \cup E')$ is the extended conflict graph. We can add a weight function $w' : E' \to \mathbb{R}$ where the weights are identical to w ($w'((v, u)) = w(v)$).

Formally we search for an MWIS $S \subseteq V$ such that there is a matching $M \subseteq E'$ with the condition $\forall v \in S \exists u \in Z : (v, u) \in M$. The combination of these two problems – the finding of an MWIS which induces a matching on the extended conflict graph – is known to be a matroid parity problem [54].

With these properties we can define the combinatorial packing problem:

Definition 2.8. Let k be a nonnegative integer and $H = (V \cup Z, E \cup E')$ be a graph with the following properties:

$$\forall \{u, v\} \in E : u, v \in V$$

$$\forall \{u, v\} \in E' : v \in V, u \in Z$$

The *combinatorial packing problem* asks for an independent set $S \subseteq V$ of size k and a matching $M \subseteq E'$ of size k such that $\forall v \in S : \exists \{u, v\} \in M$.

The weighted version of the combinatorial packing problem gets two weight functions $w : V \to \mathbb{R}$ and $w' : E' \to \mathbb{R}$ and asks for an independent set with weight W and a matching of weight W' for two given real numbers W and W'.

We can extend this problem to the question for a solution with maximum cardinality (weight). In [38] it is shown that the exact MWIS and MWC problems are NP-complete.

Theorem 2.9. *The combinatorial packing problem is* NP*-hard.*

Proof. We first show the NP-hardness for the unweighted case. The hardness of the weighted case can be proven similar to the MWIS-proof. The combinatorial packing problem can be viewed from two sides: Either we want to get an independent set on G satisfying the matching property on H, or we want a matching on H such that we can derive from that an independent set on G.

Hence the proof has to be done in two parts. We define the packing problem on the extended conflict graph for both views:

Definition 2.10. Problem P1:
Input: A graph $H = (V \cup Z, E \cup E')$ and an integer k.
Output: An independent set $S \subseteq V$ with $|S| = k$ such that $\exists$ a matching $M \subseteq E'$ with the following property: $\forall v \in S \exists u \in Z : \{v, u\} \in M$.

Definition 2.11. Problem P2:
Input: A graph $H = (V \cup Z, E \cup E')$ and an integer k.
Output: A matching $M \subseteq E'$ with $|M| = k$ such that $\{v \in V : \{v, u\} \in M\}$ is also an independent set.

We now show the NP-hardness of both problems by reducing the Independent Set problem to both P1 and P2.

Lemma 2.12. *IS* $\leq_p$ *P1.*

Proof. Let $G = (V, E)$ be a graph and k an integer. We want to find an independent set in G of size k. For each vertex $v \in V$ we create one copy $v' \in Z$, and set the edge set $E' = \{(v, v') : v \in V, v' \in Z\}$. Obviously all independent sets of G correspond directly to a solution of $H = (V \cup Z, E \cup E')$ and vice versa. Especially a solution of problem P1 of size k corresponds to an independent set in G of size k. Since the reduction can be done in polynomial time, problem P1 is NP-hard. □

Lemma 2.13. *IS* $\leq_p$ *P2.*

Proof. Let $G = (V, E)$ be a graph and k an integer. As above, we want to find an independent set in G. We use the same construction as in the proof for Lemma 2.12. Now, a solution of P2 to the graph $H = (V \cup Z, E \cup E')$ corresponds directly to an independent set in G. Therefore problem P2 is NP-hard. □

So we can continue the proof to Theorem 2.9: Since both views of the packing problem are NP-hard, the problem itself satisfies this property.

Regarding the weighted version of the combinatorial packing problem, we can reduce the unweighted case to the weighted case by setting all weights (both functions $w(v)$ and $w'(\{u,v\})$) to 1. □

Obviously, the tricky part of the combinatorial packing problem is the finding of an independent set. In our case, the matching edges can be checked as an additional condition to the IS algorithm. Hence we will focus on algorithms for the (MW)IS problem.

For a detailed analysis of the various algorithms for the MWIS problem several definitions are necessary.

Definition 2.14. Let $G = (V, E, w)$ be a weighted undirected graph with positive weight function $w : V \to \mathbb{R}$. Let $G[S]$ denote the subgraph of G induced by a subset $S \subseteq V$. Similarly, let $W(S) = \sum_{v \in S} w(v)$ be the total weight of all vertices of S. The set $N(v)$ denotes the set of vertices adjacent to v, and the set $N^+(v) = N(v) \cup \{v\}$ includes the vertex v. Let $d_G^w(v) = \frac{W(N(v))}{w(v)}$ be the weighted degree of a vertex $v \in V$ within the graph G. The maximum weighted degree shall be denoted as Δ_G^w, the average weighted degree as $\bar{d}_G^w$.

2.3 Previous Algorithms

A detailed overview on algorithms for the (unweighted) IS problem is given in [10]. For the weighted case only few algorithms have been analysed.

2.3.1 Exact Algorithms

The majority of papers proposing exact algorithms for the MWIS and MWC problems deal with the MWC problem. One of the first fast algorithms was presented by Babel [5]. This algorithm uses approximate graph colourings to find good upper and lower bounds for an efficient branching.

Another efficient exact algorithm for the MWC problem was given by Östergård [46]. This algorithm can be adapted straightforward for the MWIS problem by inverting the graph. We also implemented and modified this algorithm for our trunk packing problem (Algorithm 2.4). The analysis is shown in chapter 4.

The algorithm of [46] is an incremental algorithm. It consists of two parts: An outer loop incrementing the subset of vertices S_i, and the recursive clique algorithm `wclique`. At first, a pre-defined order is applied to the graph vertices, such that $V = \{v_1, ..., v_n\}$. The sets $S_i, i = 1, ..., n$ consist of the last vertices of V: $S_i = \{v_i, ..., v_n\}$. Starting with the smallest set S_n, we successively search for the heaviest clique on the induced subgraph $G[S_i], i = n, ..., 1$. The resulting vertex sets resp. clique weights are stored into an array $C[]$, providing an efficient access to upper bounds. We can start the algorithm `wclique` with the set $S_i \cap N(v_i)$ because a weighted clique on $G[S_i]$ is either the same as the weighted clique on $G[S_{i+1}]$, or it contains the vertex v_i. In the first case we already know

the optimal weight and are done. Hence we have to concentrate on the neighbourhood of v_i. Then we also know an upper bound for the heaviest clique by determining the vertex v_j of $S_i \cap N(v_i)$ with the smallest index and apply the weight stored in $C[j]$.

In the end, the entry $C[1]$ displays the weight of the heaviest clique of G.

The upper bounds can even be improved by applying the algorithm from both sides of the ordering, that means simultaneously running a set $T_i = \{v_1, \ldots, v_i\}$, and storing the results into an array $D[]$.

Algorithm 2.4 The MWC-algorithm of Östergård [46]

```
 1: max = 0
 2: for ( i = n downto 1 )
 3:    wclique(S_i ∩ N(v_i), w(v_i))
 4:    C[i] = max
wclique(U, weight)
 1: if ( |U| = 0 )
 2:    if ( weight > max )
 3:       max = weight                          // Save the new record!
 4:    return
 5: while ( U ≠ ∅ )
 6:    if ( weight + W(U) ≤ max )
 7:       return
 8:    i = min{j : v_j ∈ U}
 9:    if ( weight + C[i] ≤ max )
10:       return
11:    U' = U \ {v_i}
12:    wclique(U ∩ N(v_i), weight + w(v_i))
13: return
```

In [55, 56] several branch-and-bound algorithms, including Algorithm 2.4, are compared to each other. Unfortunately, many graph instances, such as those from the DIMACS Implementation Challenge [33], are not suitable for the MWIS problem: Most of these instances are random unweighted graphs, and it is not clear in which distribution the weights have to be set in order to generate an interesting, that means hard to solve, instance. Therefore many instances tested in [55] are graphs with special properties, such as regular graphs, graphs with only few distinct vertex degrees etc. The case with random graphs is somewhat different. At least the edge probability has to be determined. Then it is to be decided whether the vertex weights shall be chosen at random or depend on the number of incident edges. Each of these parameters has a great influence on the solution and on the runtime of the algorithms. In [8] similar effects have been encountered during the evaluation of colouring algorithms: If the edge density is chosen to be linear dependent on $|V|$ (the probability for an edge between two vertices u and v is constant and independent of $|V|$), then a graph becomes quite easy to colour. However, if the edge density is smaller, that is, the probability for an edge between two vertices is

inversely proportional to $|V|$, then the exact colouring becomes difficult. The same effect might be encountered when dealing with (weighted) independent sets. However, since our main interest lies in the field of industrial application, we did not examine those side effects of random graphs.

2.3.2 Approximation Algorithms

The greedy algorithm is presented in section 2.4.1. An analysis of this algorithm can be found in [34]. The main statement there is an approximation ratio of δ_G^w where this denotes the *weighted inductiveness* of the graph:

Definition 2.15. [34] Let $G = (V, E)$ be an undirected graph with positive weight function w. The weighted inductiveness is then given as

$$\delta_G^w = \max_{U \subseteq V} \min_{v \in U} d_{G[U]}^w(v). \tag{2.4}$$

With this definition it can be shown [34] that the greedy algorithm for the MWIS problem has approximation ratio δ_G^w, which is the first result that also considers the weights of the vertices. Another algorithm using linear programming has been given in [34], which has an approximation ratio of $\min(\frac{\bar{d}_G^w+1}{2}, \frac{\delta_G^w+1}{2})$, and an algorithm using semidefinite programming with approximation ratios of $O(\frac{\bar{d}_G^w \log\log \bar{d}_G^w}{\log \bar{d}_G^w})$ and $O(\frac{\delta_G^w \log\log \delta_G^w}{\log \delta_G^w})$.

Other approximation algorithms are designed for special graph types, such as permutation graphs [13], trees [14], sparse random graphs [28], fork-free graphs [37] etc. There is also work on so-called intersection graphs [30], although the notion of intersection refers to the combination of two different graph types.

All of those publications deal only with the approximation of weighted independent sets or cliques. They give rather good approximation ratios for large graphs resp. total solution weights, but these results are not applicable to our problem: As stated in [1], the greedy algorithm for example exceeds its theoretical approximation bound given in (2.4) for every instance we tested. This is caused mainly by the rather small graphs and small weights we use.

2.4 Adaption to the SAE Packing Problem

Algorithms for the MWIS problem have a significant flaw if they are supposed to be applied to the SAE packing problem. An optimal MWIS might consist of several vertices representing the same type, exceeding the maximal number of occurrences for this box type. This inherent feature of the SAE packing problem causes two peculiarities concerning the packing algorithms. On one hand, not all direct approaches will lead to a good solution of the packing problem because several solution ranges will be invalid for our perspective. On the other hand, an algorithm examining only possible solutions systematically draws huge benefits from this search space reduction. So it is necessary to adapt known MWIS algorithms to the special environment of the SAE packing problem.

2.4.1 Greedy-Algorithm

The greedy algorithm is designed in a straightforward manner. First, compute the weighted degree for all vertices. Then successively add vertices to an independent set, starting with vertices having a small weighted degree. The algorithm is shown in detail as Algorithm 2.5.

Algorithm 2.5 Greedy algorithm for finding an MWIS

1: $I = \emptyset$, $U = V$
2: **while** ($U \neq \emptyset$)
3: Let $v \in U$ be the vertex with $d_{G[U]}(v) = \min_{u \in U} d_{G[U]}(u)$
4: $I = I \cup \{v\}$
5: $U = U \setminus N(v)$
6: **if** (Type of v is full)
7: $U = U \setminus \{u \in U : t(u) = t(v)\}$

A detailed analysis of Algorithm 2.5 has been done in [50], together with absolute bounds of the resulting weighted independent sets. The approximation ratio of the greedy algorithm could be established in [34]. It has been computed in terms of the average weighted degree $\bar{d}^w_G$ (Definition 2.14) and the weighted inductiveness δ^w_G (Definition 2.15) of the graph G. With these parameters the approximation ratio of the greedy algorithm is δ^w_G. This ratio can be improved (a smaller approximation ratio is better) to $O(\min\{\frac{\bar{d}^w_G \log\log \bar{d}^w_G}{\log \bar{d}^w_G}, \frac{\delta^w_G \log\log \delta^w_G}{\log \delta^w_G}\})$ by using semidefinite programming [34]

Since the approximation ratio is computed for worst case behaviour and growing graph sizes, it does not give a good start for the quality of the solution in the specialised case of the conflict graph for the trunk packing problem. Here even the size of the conflict graph is rather small, and therefore the approximation ratio is useless. A better solution could be achieved if the algorithm is used for a brute force approach.

2.4.2 Brute Force Approach

A simple way to find an independent set of maximum weight is to enumerate all independent sets. Algorithm 2.6 uses a recursive approach for enumeration.

Two lines bear the crucial elements in Algorithm 2.6. They are discussed in the next two sections.

2.4.3 Computing an Upper Bound

In line 4 of Algorithm 2.6 an upper bound for the resulting independent set is to be computed. This bound has to be strong enough to enable an efficient pruning of the recursion tree. First of all, the upper bound can be computed by summing up all weights of the remaining free vertices. Although this bound can be computed efficiently, it is rather hard to get a value near the real possible weight of the independent set. In almost all cases, the result lies well above a realistic amount.

Algorithm 2.6 The recursive basis of an enumeration algorithm

Recursive enumeration (G, I)

Input: graph G, independent set I

1: Let F be the set of all free vertices of G
2: **if** ($w(I) > w(I_{\max})$)
3: $I_{\max} = I$
4: **while** ($F \neq \varnothing$ and UpperBound$(F, I) > w(I_{\max})$)
5: Choose $v \in F$
6: $I' = I \cup \{v\}$, $F = F \setminus \{v\}$
7: Recursive enumeration (G, I')
8: **Return** $(I_{\max}, w(I_{\max}))$

The other extremal point would be to calculate exactly the size of the best independent set that could be reached within the current branch. Of course, this is impossible because otherwise we would not need to find an algorithm for the MWIS problem. Both criteria drop out of our considerations, the first one because of its inaccuracy and the latter one because of its inefficiency.

Fortunately, the first bound can be lowered significantly by using the restrictions given by the SAE J1100 standard. As shown in Table 1.1, every box can only occur a few times within a single packing. So the maximal reachable volume is bound by 28.7049 ft^3. This is still well above the capacity of a typical trunk.

Nevertheless we can make use of this bound. We can reduce the upper bound if there are no more vertices of a type t left. So if the boxes are grouped per type (see section 2.4.4), the bound can be reduced each time we discard the last vertex of a box type.

In Algorithm 2.4 a third variant of an upper bound is used: The weight of the heaviest independent set containing only vertices from a subset of V is computed directly. Then the subset is increased systematically, thus resulting in the multiple execution of the MWIS algorithm. Although we then have the best possible upper bound, its computation will take substantially longer than the other bounds.

2.4.4 Choosing Vertices

The most important step is choosing the next vertex $v \in F$ (line 5 of Algorithm 2.6). This step is crucial because a good result can be achieved at an early stage of the enumeration, and a good portion of recursion branches can be pruned using the upper bound criterion.

For this step several criteria are possible:

1. The vertex with the smallest *ID*
2. A vertex chosen at random
3. The vertex having the smallest (weighted) degree
4. A vertex with the largest (or smallest) weight

The ID criterion is as well the simplest one. Basically no decision is made at all, and the vertices are sorted by their time of creation. The implementation of criterion 1 is the fastest of all because no computation at all is performed to determine the ordering of the vertices.

The second criterion has got a similar behaviour. No computation is needed to determine the next vertex.

Criterion no. 3 splits into two possibilities. First, the "normal" degree of a vertex is used to find the next vertex of the independent set. This method can be implemented rather efficiently since only the number of neighbours is to be counted. The second approach would be to compute the weighted degree of each vertex, which resembles exactly the greedy Algorithm 2.5. Both possibilities cause the recursion tree to be very wide since the first taken vertices have only few neighbours, thus leaving a wide choice for the next recursion level. Additionally, the degree of a vertex corresponds directly to the size of the respective box. That means, if this strategy is used the smallest boxes are packed first.

Unfortunately, the (weighted) degree of the remaining vertices has to be updated in every recursion step. The computation of $d^w_{G[F]}(v)$ for all vertices would take $O(|F|^2) = O(n^2)$ steps since it is necessary to examine every pair of vertices. Although it would be sufficient to examine all edges, the conflict graph is very dense and almost every pair of vertices is connected by an edge. Additionally, the conflict graph has to be stored efficiently, and due to the density of the graph, an adjacency matrix is appropriate in this case. Another way to update the weighted degrees would be an incremental (or, in this case, a *de*cremental) approach: If vertex v is added to the IS, then all vertices of $N(v)$ would be unusable. Hence, the weighted degrees of all vertices $w \in N(N(v))$ will be decremented by $W(N(w) \cap N(v))$. So all neighbours of all vertices of $N(v)$ have to be examined. However, since the conflict graph is very dense, the update function would still need $O(n^2)$ time.

The last decision criterion makes use of the fact that only few different vertex weights are available in the trunk packing problem. Vertices with small weight also have a small neighbourhood and therefore the resulting independent set will contain more vertices. On the other hand, taking vertices with a large weight will result in few vertices in higher recursion levels.

In fact, a combination of the last two criteria proves to be the best method to select the vertices. First, the remaining free vertices are grouped by their weight, with the largest weight first. This resembles the way one would pack boxes into a trunk in reality: Start with large boxes and add small boxes afterwards. Second, sort the vertices of each group by their weighted degree. There the smallest degree comes first. Translated into the real world, it means to squeeze the largest box into a corner of the free space.

2.5 Conflict Graph Reduction

Since the algorithms discussed in section 2.2 have to examine all edges of the conflict graph, it might prove useful to *reduce* the conflict graph size. To find an independent

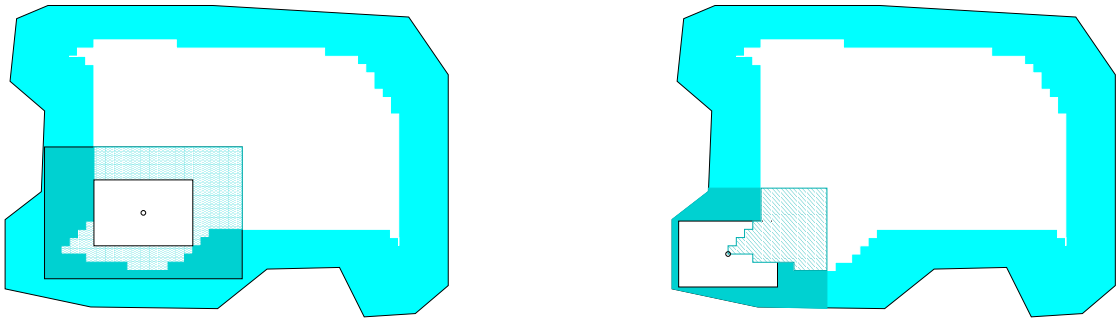


Figure 2.6: If a box is placed somewhere near the boundary, much space is wasted in relation to placing it directly at the boundary

set of maximum size resp. maximum weight, it is necessary to observe some properties of a maximal independent set I. Our goal is to eliminate some vertices of the graph without reducing the size of the largest independent set. In an unweighted graph we can identify some vertices which can be deleted without affecting the size of the maximum independent set, using the following theorem.

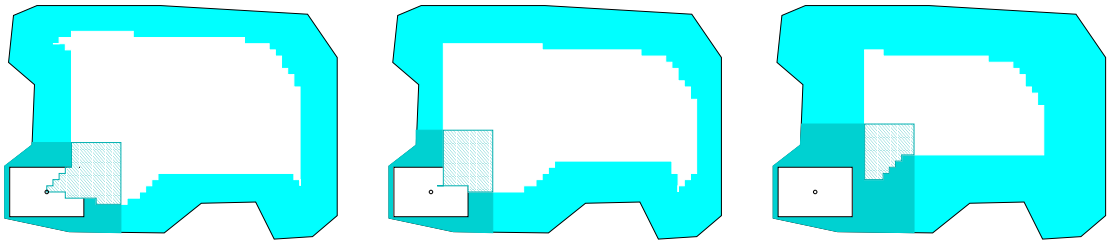
Theorem 2.16. *Let $G = (V, E)$ be an undirected graph, and $u, v \in V$ be adjacent vertices ($\{u, v\} \in E$). If $N^+(u) \subsetneq N^+(v)$, then the following is true:*
For every independent set $I \subseteq V$ containing v, there is also an independent set $I' \subseteq V$ with at least the same cardinality containing u.

Proof. Obviously, $v \in I \Leftrightarrow u \notin I$ since $\{u, v\} \in E$. $N^+(u) \subsetneq N^+(v)$ states that all neighbours of u are also neighbours of v, so v can be replaced by u. That is, $I' = I \setminus \{v\} \cup \{u\}$. □

Theorem 2.16 can be used in an elegant way to reduce the number of vertices in a graph. Since the goal is to find an independent set of maximum size, it is useful to include the vertex u instead of v because v has got a larger neighbourhood and more vertices would be useless for later additions. Since *all* neighbours of u are also neighbours of v, one would rather choose u with its smaller neighbourhood.

What does this mean for the trunk packing problem? If there is a box position near the boundary of the grid, some space between box and boundary would be unusable for other boxes (see Figure 2.6). So it would be better to move the box directly to the boundary, where less space is "wasted". This approach is sometimes called a *left upper justified (LUJ)* packing: All boxes are placed in a way that their bottom, left and front sides touch either other boxes or the surrounding geometry. It has been shown that any axis-oriented packing can be transformed into a LUJ packing [51, 59].

The influence of a single box is shown in Figure 2.7. There a $6'' \times 4''$-box is placed in the lower left corner of the trunk which has been discretised by a $0.5''$-grid. The shaded regions at the trunk boundary mark the unusable cells of the grid whereas the pale region shows the usable cells for different box sizes. The hatched part of the usable cells marks the region where a box of the second size would intersect the already placed box. The



(a) $6'' \times 4''$ vs $6'' \times 4''$ (b) $6'' \times 4''$ vs $4'' \times 6''$ (c) $6'' \times 4''$ vs $10'' \times 7''$

Figure 2.7: Conflict regions

size of the hatched region depends directly on the sizes of the two boxes. Furthermore, the hatched region is an indicator for the weighted degree of the placed box.

The effect of reducible vertices is directly related to the size of the boxes compared to the trunk. Since the SAE boxes are relatively large and the DIN boxes are relatively small, the reduction algorithm is only useful for the SAE case.

Theorem 2.16 also applies to the MWIS problem:

Corollary 2.17. *Let $G = (V, E, w)$ be a weighted undirected graph with weight function w, and $u, v \in V$ be adjacent vertices ($\{u, v\} \in E$). If $N^+(u) \subsetneq N^+(v)$ and $w(u) \geq w(v)$, then the following is true:*
For every weighted independent Set $I \subseteq V$ containing v, there is also a weighted independent set $I' \subseteq V$ with at least the same weight containing u.

Proof. Simply replace v by u. Then $w(I') = w(I) - w(v) + w(u) \geq w(I)$. □

Corollary 2.17 shows the crucial problem of the SAE case: If two different box types are compared, normally the larger box also gets the larger neighbourhood. Then again, a vertex with large weight may contribute to the optimal solution – even if it would have been deleted due to a vertex with small weight. Fortunately, there are only seven (a fixed small number) different box sizes. So an easy way to reduce the number of vertices is to do this *typewise*, that means to compare only vertices of the same weight. In the general case, one could partition the vertices into a few weight classes and perform the reduction on these classes.

Algorithm 2.7 Graph Reduction

```
1: for ( each type t )
2:   for ( each edge {u,v} with Type(u) = Type(v) = t )
3:     if ( N+(u) ⊂ N+(v) )
4:       V = V \ {v}
```

Using this procedure it is possible to reduce the number of vertices of a typical conflict graph of the Trunk Packing problem with cell size $2''$ by 75% and the number of edges by up to 90%. Therefore, the performance of the packing algorithms increases significantly because of the smaller conflict graph.

trunk size	before		after		reduction	
	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	vertices	edges
with H-boxes						
small ($3''$)	815	110963	790	106692	3.1%	3.9%
small ($2''$)	3259	2229450	1857	750735	43.0%	66.3%
small ($1.5''$)	9598	21048262	3475	2921359	63.8%	86.1%
small ($1''$)	34783	264029797	12374	36860456	64.4%	86.0%
mid-size ($3''$)	2282	865263	2243	854566	1.7%	1.2%
mid-size ($2''$)	8211	12985913	4774	4153490	41.9%	68.0%
mid-size ($1.5''$)	22834	107641318	9886	19866786	56.7%	81.5%
large ($3''$)	3280	1635206	3223	1618072	1.7%	1.1%
large ($2''$)	12988	29912476	7590	9534351	41.6%	68.1%
without H-boxes						
small ($3''$)	264	27383	84	2454	68.2%	91.0%
small ($2''$)	1222	580366	235	19759	80.8%	96.6%
small ($1.5''$)	3794	5749578	472	77935	87.6%	98.6%
small ($1''$)	12813	66926604	849	255752	93.4%	99.6%
mid-size ($3''$)	1140	420113	478	63364	58.1%	84.9%
mid-size ($2''$)	4179	5847307	1195	413052	71.4%	92.9%
mid-size ($1.5''$)	11896	48619675	2169	1394084	81.8%	97.1%
mid-size ($1''$)	40368	58349586	5587	10247353	86.2%	98.2%
large ($3''$)	1764	893098	657	109876	62.8%	87.7%
large ($2''$)	7341	16001154	1937	965740	73.6%	94.0%
large ($1.5''$)	22401	144610926	4763	5920031	78.7%	95.9%
large ($1''$)	75566	174874343	11265	36362098	85.1%	98.0%

Table 2.3: Conflict graph sizes for typical problems before and after reduction.

Table 2.3 shows the effects of the conflict graph reduction on instances typical for three trunk sizes. As can be seen, with a large grid spacing the reduction process has only little effect. The reason for this phenomenon is the following: In a grid with large spacing a box covers only few grid cells. Thus the neighbourhood of a vertex is rather small, and Theorem 2.16 has almost no effect.

Similar considerations can be applied to the size and shape of the trunk itself. More precisely, the surface area compared to the volume of the trunk has got a large influence on the amount of removed vertices: Theorem 2.16 has got more influence near the border of the trunk. As described by LUJ packings, a box placed in a corner of the trunk will have only few adjacent vertices. Of these, almost all will share the same neighbourhood, thus causing many vertices to be removed. Since the ratio of surface to the volume is greater at smaller trunks, the reduction has got its greatest effect on the smallest trunks.

Finally it can be seen that the reduction has got a great effect if small boxes are omitted from consideration. Therefore it is unlikely that the reduction algorithm will bring an improvement to the DIN packing algorithms.

2.5.1 Efficient Implementation of the Graph Reduction Algorithm

The reduction algorithm is useful only if it does not impact too much on the runtime of the entire framework. Thus, an investigation of the required running time for this part is necessary.

Naturally, one has to check the inclusion property for the neighbourhoods of every adjacent pair of vertices. That means to calculate two sets of size $O(n)$ and check whether one set is contained in the other, for $O(n^2)$ pairs of vertices.

From our tests (see chapter 6), we get runtimes for the construction of the conflict graphs as shown in Table 2.4.

For measuring the reduction time, we used the same trunk geometries as in chapter 4. Although the geometries for quantifying the graph sizes were the same, we used different (non-optimised) grids for counting vertices and edges. As can be seen in Table 2.4, the reduction time depends on the size of the conflict graph as well as on the reduction ratio. When H-boxes are included, the reduced conflict graph will be larger than a reduced conflict graph without H-boxes in relation to the original conflict graph. Our algorithm is implemented such that the outer loop only iterates over the vertices that are really contained in the reduced graph. So the runtimes fulfilled our expectations that if the reduced graph contains only few vertices in relation to the original graph, then the reduction process will be very fast.

2.5.2 Is Reduction Always Useful?

Now the question arises whether it is in all cases efficient to perform a conflict graph reduction before starting the packing algorithm. In general, the conflict graph represents rather large boxes within a relatively small trunk. So a single vertex is directly connected to a large amount of vertices. If the trunk size increases, many vertex neighbourhoods lie completely on `inside`-cells of the trunk, thus the size of the neighbourhood of some

trunk	$\|V\|$ before	$\|V\|$ after	t_{total}	t_{reduce}
with H-boxes				
I1 ($3''$)	632	599	$0.2s$	$< 0.1s$
I1 ($2''$)	1440	661	$1.9s$	$0.3s$
I1 ($1.5''$)	4630	1526	$8.1s$	$1.0s$
I1 ($1''$)	16694	5351	$1min37s$	$25.2s$
T1 ($3''$)	3036	2993	$1.7s$	$0.5s$
T1 ($2''$)	11368	7780	$20.7s$	$7.1s$
T1 ($1.5''$)	28534	12840	$2min18s$	$48.1s$
T2 ($3''$)	2541	2516	$1.2s$	$0.4s$
T2 ($2''$)	10090	6620	$17.2s$	$5.7s$
T2 ($1.5''$)	24448	10683	$1min47s$	$35.6s$
T2 ($1''$)	85998	34598	$32min43s$	$15min33s$
without H-boxes				
I1 ($3''$)	207	75	$< 0.1s$	$< 0.1s$
I1 ($2''$)	298	41	$0.4s$	$< 0.1s$
I1 ($1.5''$)	1570	159	$2.9s$	$< 0.1s$
I1 ($1''$)	3907	242	$23s$	$0.4s$
T1 ($3''$)	1801	810	$0.9s$	$0.1s$
T1 ($2''$)	6260	2124	$10.2s$	$1.4s$
T1 ($1.5''$)	15316	3747	$1min4s$	$9.0s$
T1 ($1''$)	48222	7408	$14min6s$	$2min11s$
T2 ($3''$)	1529	789	$0.7s$	$0.1s$
T2 ($2''$)	5694	1705	$8.8s$	$1.1s$
T2 ($1.5''$)	13393	2785	$50.6s$	$6.9s$
T2 ($1''$)	39665	5555	$10min51s$	$1min26s$
T3 ($3''$)	2221	982	$1.3s$	$0.2s$
T3 ($2''$)	9934	2780	$19.7s$	$3.5s$
T3 ($1.5''$)	23977	4982	$2min$	$22.8s$
T3 ($1''$)	79748	11488	$30min28s$	$6min25s$

Table 2.4: Conflict graph creation (t_{total}) and reduction (t_{reduce}) times. For the trunk geometries, see chapter 4.

vertices is equal to the sizes shown in Table 2.2. Although these neighbourhoods might overlap, there is no containment relation between the neighbourhoods. Therefore, Theorem 2.16 has no effect within the centre of large trunk geometries. The same statement can be made for very small boxes. Table 2.3 shows that the reduction including H-boxes is not as efficient as the reduction without H-boxes on the same grid. One could imagine the effects on DIN boxes.

3 Packing Without Grids

The previous chapters dealt with axis-oriented packing of boxes into a trunk. The placements of the boxes were restricted to discretised coordinates derived from a grid approximation of the container. Such an approximation causes a severe loss in accuracy and optimality of the results. In addition to this the box sizes have to be rounded in order to fit onto the grid discretisation. This causes invalid packings with overlapping boxes. These packings have to be modified with additional procedures (see section 6.2.2).

So the next logical step is to drop the restriction to discretised coordinates. With arbitrary coordinates we are able to leave the box sizes in their original values, thus creating only non-overlapping packings.

As before, we will approach the packing problem by using a conflict graph.

3.1 Minkowski Sum

3.1.1 Definitions

Definition 3.1. [42] Given two sets $A, B \subseteq \mathbb{R}^3$, the Minkowski Sum $A \oplus B$ is defined as $A \oplus B = \{a + b : a \in A, b \in B\}$.

Using the Minkowski Sum we can define all possible positions of boxes within the trunk. We need these positions to describe the vertices of the conflict graph. As before, a vertex will represent a box of a certain type with a certain orientation. The only difference now is the representation of the coordinates. The position of a box can be referred to by the so-called anchor point analogous to the anchor cube:

Definition 3.2. Let $B = (t, o, p)$ be a box of type $t \in T$ and orientation $o \in O$. Then $p \in \mathbb{R}^3$ is called *anchor point* of this box. Normally the point p is equal to the mass centre of B. A box with $p = (0, 0, 0)$ shall be denoted as $B_{t,o}$.

Definition 3.3. For a type $t \in T$ and an orientation $o \in O$, let $w(t, o) \in \mathbb{R}^3$ be the side lengths of a box of type t in orientation o. Let furthermore be $w_i(t, o)$ be the side length in the i-th dimension.

Definition 3.4. The box $B = (t, o, p)$ can be represented as a subset of $\mathbb{R}^3$:

$$B = \left\{ x \in \mathbb{R}^3 : x_i \in \left[p_i - \frac{w_i(t,o)}{2}, p_i + \frac{w_i(t,o)}{2} \right], i = 1..3 \right\}$$

The description of a box $B_{t,o}$ is equivalent to local coordinates. These local coordinates are used in the computation of the Minkowski Sum to define the set of possible placements.

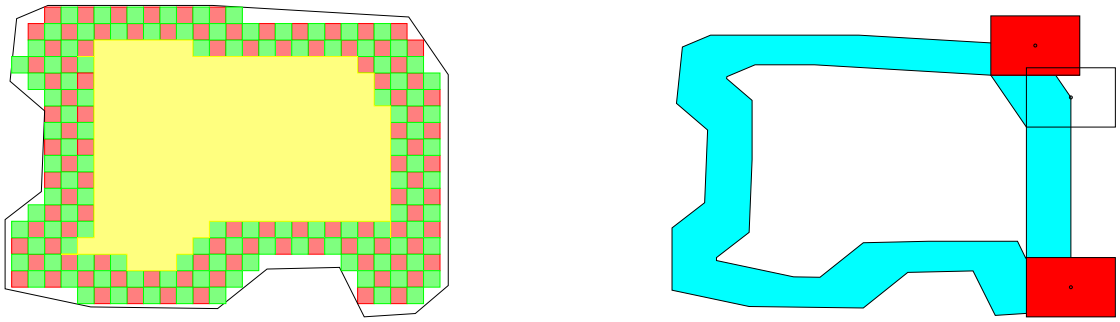


Figure 3.1: A two-dimensional trunk and its Minkowski Sum with a $4'' \times 6''$-box, compared to a $1''$-grid

Now the boxes have no longer to be placed on a grid, but within a space where they will not intersect the geometry of the trunk. Let C be a trunk container, described by a set of triangles. A single box B in a fixed orientation can be placed at various positions within C. The set of possible positions of B can be derived from C as follows: Consider a fixed triangle $d \in C$. Now place B within C such that B only touches d. This is the outmost possible position for box B within the trunk since if the centre of B would be placed nearer to d, the box would intersect the triangle. All points from where B would intersect d can be generated by computing the Minkowski Sum $B_{t,o} \oplus d$. This gives a convex object whose border is the set of all points where B touches d. The Minkowski Sum can be calculated for all triangles with the box B. These convex objects have to be joined. The result is an object shaped like the original trunk but with a shell that has the width of the box. The inner boundary of this object is then the boundary of the desired set of possible placements of Box $B_{t,o}$, from now denoted $P_{t,o}$.

Definition 3.5. Let $C \subset \mathbb{R}^3$ be a trunk container and $\overline{C} = \mathbb{R}^3 \setminus C$ its complement. The set $P_{t,o} = \overline{\overline{C} \oplus B_{t,o}}$ specifies the set of anchor points where a box of type t and orientation o would not intersect the trunk geometry.

The computation of the sets $P_{t,o}$ has to be done for each box type in each desired orientation, that is, for seven box types as in the SAE standard and the six main orientations, there will be 42 sets $P_{t,o}$. The set V of vertices of the conflict graph will therefore consist of vertices $v = (x, y, z, t, o)$ with $(x, y, z) \in P_{t,o}$.

To calculate the set $P_{t,o}$, the box is placed on every point of the trunk boundary and the covered space is marked. The remaining – unmarked – part of the space inside the trunk will be the set $P_{t,o}$.

In the case of trunk packing, the trunk geometry C defines the space *not* to be used by boxes, which includes the boundary and the entire outside of the trunk. The boxes are solid, which means that the Minkowski Sum of trunk and box will be nothing more than a small copy of the trunk, where some details of the geometry are omitted. This effect can be compared to the re-painting of a faint line using a considerably larger pen.

Figure 3.1 illustrates the concept of the set $P_{t,o}$. The trunk boundary is thickened by a box of type t and orientation o which is moved along the boundary. Especially the

corners of the available space are modelled far better than by using a grid.

3.2 Continuous Conflict Graph

After the available space has been defined, we can proceed to the conflict graph itself. For the discretised case we created and stored the entire graph a priori. This allows efficient access operations to determine adjacency and vertex degrees.

Definition 3.6. The continuous conflict graph is a graph $G = (V, E, w)$ with weight function $w : V \to \mathbb{R}$. The set V is defined as $V = \{(x, y, z, t, o) : t \in T, o \in O, (x, y, z) \in P_{t,o}\}$.

Unfortunately, the continuous approach is accompanied by two major problems. First of all, it seems impossible to store all possible vertices of the conflict graph since we only have a description of intervals rather than distinct points. These are necessary to define vertices. At least, one can determine whether a given point will suffice to be a vertex by inclusion tests. Similarly, an adjacency test for two vertices can be performed using interval arithmetics. For each vertex $v = (x, y, z, t, o) \in V$, the set of conflicting points $C_{t',o'}(v)$ for each type t' and orientation o' can be found using the following identity:

$$C_{t',o'}(v) = (x, y, z) \oplus (B_{t,o} \oplus B_{t',o'}) \tag{3.1}$$

So we can continue Definition 3.6 by defining the edges of the continuous conflict graph:

Definition 3.7. In the continuous conflict graph, two vertices $u = (x_u, y_u, z_u, t_u, o_u)$ and $v = (x_v, y_v, z_v, t_v, o_v)$ are adjacent if

$$(x_u, y_u, z_u) \in C_{t_u,o_u}(v).$$

The next problem arises when the vertex degrees are to be calculated. Clearly missing is a notion of how many box placements are made unusable after placing a box. In the discretised case we summed the volumes of the adjacent vertices. The continuous case, however, would now require to compute an integral over the adjacent volume of the box. This value would have to be weighted by the type of the adjacent volume, resulting in the computation of 42 neighbourhoods of each vertex. Since there are infinitely many vertices in the conflict graph, a three-dimensional function computing the vertex degrees would come in handy. This function could be evaluated and local and global minima can be computed. These minima naturally form start points for a packing algorithm.

However, such a degree function would be very complex: it depends mainly on the trunk geometry. This geometry consists of thousands of triangles, sometimes up to $100,000$. So the degree function would be piecewise linear, and the description of this function would take its time. Especially the determination of minima of the degree function will exceed the restricted timeframe. So we will have to determine a priori where a vertex with minimal degree would be located.

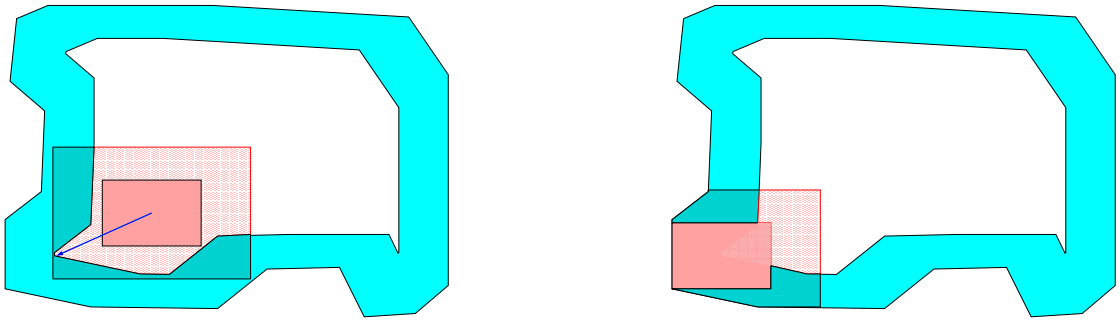


Figure 3.2: The vertex degree is smaller if the box is moved into a corner of $P_{t,o}$

Given a trunk geometry and the set $P_{t,o}$ for a particular box type and orientation, it seems obvious that the point with minimal degree will not be located somewhere in the centre of the available space. On the contrary, the minimal degree will become manifest in one of the corners of $P_{t,o}$. Why? A box $B_{t',o'}$ will always make the same volume unusable: a box with the dimensions of $B_{t,o} \oplus B_{t',o'}$. The vertex degree, however, resembles the portion of $P_{t,o}$ which is made unusable. Now, in a corner of $P_{t,o}$, only a small part of the affected volume lies within $P_{t,o}$, hence the vertex degree will be small compared to the degree of a vertex inside $P_{t,o}$. Figure 3.2 illustrates this for a 2-dimensional trunk.

That means it will suffice to examine only the corners of the minkowski sums to determine the initial vertices of the conflict graph. However, not all boxes within an optimal packing touch the trunk geometry. So it is necessary to add vertices successively during the run of the algorithm. These vertices are created by examining new points resulting from added boxes. So the packing algorithm will have to deal with a finite, but growing graph, depending on the current packing.

3.3 An Algorithm on Continuous Placements with Fixed Orientations

The continuous packing Algorithm 3.1 looks merely the same as Algorithm 2.6. However, there are several differences to be mentioned.

At first, the graph reduction itself is not useful to be computed a priori. It takes too much time and the effects are rather marginal. Hence, the graph reduction is computed implicitly by only using the corners of the usable space. As shown by Schepers and Wottawa [51, 59], every packing can be modified such that all boxes are placed at one of the lower front and left corners of their free space at the time of their addition: Each box has to touch either the trunk geometry or other boxes with their left, lower and front sides. If any box has got a “free” side, we can simply move this box in this direction until it touches an obstacle.

The determination of the lower, front and left point of the free space with minimal vertex degree is rather expensive. We therefore relaxate this condition and place boxes

at an arbitrary corner of their free space. In line 8, the conflicting region of vertex v is cut off each Minkowski Sum. This step creates new corners of each Minkowski Sum (line 9) which are added to the set of vertices in line 10.

Algorithm 3.1 Continuous packing algorithm with axis-oriented boxes

Global variables: $P_{t,o}$ for each type and orientation, I, $I_{\max}$, max

1: **for** (each box type t)
2: **for** (each orientation o)
3: Compute $P_{t,o}$
4: $F_{t,o} = \{v = (p,t,o) \in V : p \in \mathbb{R}^3 : p \text{ is a corner of } P_{t,o}\}$
5: $I = \varnothing$, $I_{\max} = \varnothing$, max = 0
6: $F = \bigcup_{t,o}\{F_{t,o}\}$, $\mathcal{M} = \bigcup_{t,o}\{P_{t,o}\}$
7: Continuous recursive enumeration $(F, I, \mathcal{M}, \max)$ (Algorithm 3.2)

Algorithm 3.2 Continuous recursive enumeration $(F, I, \mathcal{M}, \max)$

Local variables: $F_{t,o}$ for each type and orientation

Input: F – set of free points,
I – current independent set,
$\mathcal{M}$ – current free space,
max – size of best independent set so far

1: **if** ($w(I) > \max$)
2: $\max = w(I)$, $I_{\max} = I$
3: **while** ($F \neq \varnothing$)
4: Choose $v \in F$
5: $I' = I \cup \{v\}$
6: **for** (each box type t)
7: **for** (each orientation o)
8: $P_{t,o} = P_{t,o} \setminus C_{t,o}(v)$
9: $N_{t,o} = \{p \notin F_{t,o} : p \text{ is a new corner of } P_{t,o} \text{ after } v \text{ has been added}\}$
10: $F'_{t,o} = (F_{t,o} \setminus C_{t,o}(v)) \cup N_{t,o}$
11: Continuous recursive enumeration $(F', I', \mathcal{M}, \max)$
12: Restore $P_{t,o}$ for each box type and orientation
13: $F = F \setminus \{v\}$

Return $(I_{\max}, \max)$

The Minkowski Sum itself has got a rather complex structure and uses a large amount of memory. So it is inefficient to store the Minkowski Sums for each recursive call of Algorithm 3.2. Instead we created a backup mechanism which stores only the differences created by the addition of a vertex.

As in Algorithm 2.6, an upper bound can be applied in line 3 of Algorithm 3.2. The choice of the vertex v can also be modified according to different criteria. The most effective choice would be degree-based.

However, several problems occur concerning the design of the algorithm. At first, an exact computation of the Minkowski Sum is not efficient. Further problems include efficient access operations on the Minkowski Sums and the size of the conflict graph. The following sections show how these problems can be bypassed.

3.3.1 Minkowski Sums: Exact Computation

As described in Definition 3.1, the Minkowski Sum of two objects A and B has to be computed by placing object B on every point of A or vice versa. Since the trunk A is divided into many triangles, it suffices to compute the union of the Minkowski Sums of each triangle and object B. It is quite easy to generate such a Minkowski Sum for one triangle: The box B, which is just a cuboid, generates eight points at each triangle corner. The convex hull of these 24 points is exactly the Minkowski Sum of a triangle and a box. The tough part is the union of all these Minkowski Sums. Typically, a trunk consists of thousands of triangles. Since we are interested in the inner hull of the Minkowski Sum, it is necessary to compute many line and plane intersection points of the objects. Numerical instabilities and the number of objects lead to a time-consuming, inefficient process. Although a first efficient algorithm for computing an exact Minkowski Sum has been presented in [31], the size of our trunk geometries leads to a runtime of approximately 40 minutes for only one Minkowski Sum. Since there are seven different box types and six fixed orientations, only the computation of all 42 Minkowski Sums exceeds the desired timeframe of one day.

To provide a good approximate representation of the available space efficiently we use an approximate computation of the Minkowski Sums instead. The following method has been sketched in [52] and was briefly described in [9].

Since the main problem was to compute exact intersection points of lines and planes, we first restricted the coordinates of the triangles to be integer values. This restriction is possible since the triangle coordinates are given as decimal values with an accuracy of at most 0.1 mm. Therefore we place all triangles on a "grid" with 0.1 mm spacing. However, this restriction does not solve the line/plane intersections yet. These intersections could now be solved using rational arithmetics. In order to increase the performance of the Minkowski Sum computation significantly, it is necessary to get rid of these intersection calculations.

It is far easier to calculate the union of several axis-oriented boxes than the union of arbitrary convex polyhedra: This can be done efficiently by using interval comparisons. The idea is the same as in a scanline algorithm for rectangle intersections, but now used in three dimensions. We represent the boxes as sets of rectangles and cut and split them in order to represent the boundary of the union of the boxes. For this operation we need a finite number of boxes, not the infinite set of placements provided by $\mathbb{R}^3$.

Therefore we discretise the trunk geometry again, this time by a dense point cloud. The procedure is sketched in Algorithm 3.3.

In line 3 a threshold has to be defined. This parameter has got elementary influence on the denseness and the size of the point cloud. The points have to be close enough to each other to ensure that no box can tunnel between two points to the outside of the

Algorithm 3.3 Algorithm for creating a dense point cloud

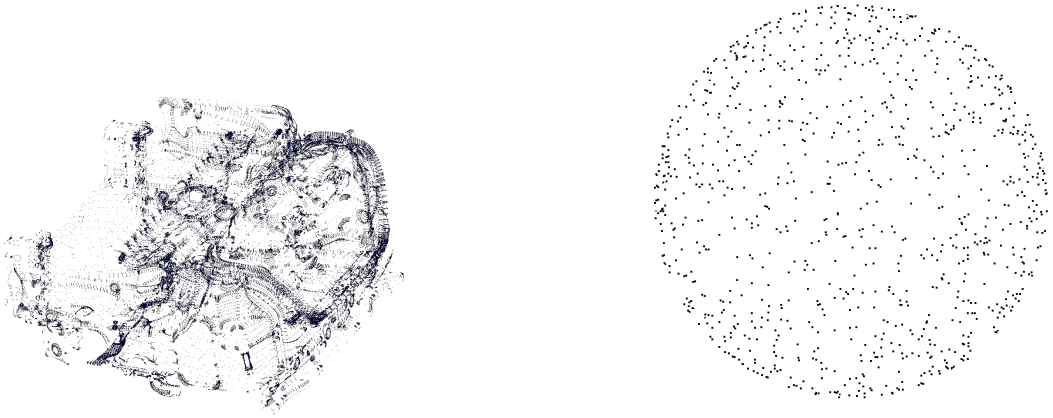
```
1: Store all triangles in a priority queue, sorted by their longest side length
2: d =PQ.top()
3: while ( longest side length of d = ABC is too large )
4:     Create new point D in the middle of the longest side (BC) of d
5:     Create two triangles d' = ABD, d'' = ACD
6:     Remove d and add d', d'' to the priority queue
7:     d =PQ.top()
8: for ( each triangle d )
9:     Store the three points A, B, C.
10: Remove duplicate points
```

trunk. This means that for each point of the point cloud, the nearest point has to be at most the smallest box dimension away. Then again, we want to have as few points as possible. With Algorithm 3.3 it might well be that points are duplicated because triangles may share the same longest side. It is also possible that several long triangles are put together like a fan, thus resulting in points close to each other in circles around the centre of the fan.

We can of course remove such points because they are only added for the density issue. Since in those regions the point cloud is already dense enough, these points can be deleted. However, the points belonging to the original trunk geometry may not be deleted: These points might express distinct features of the geometry, and if they are eliminated, the packing might intersect with the original trunk geometry. Therefore we will get in the end a point cloud containing few points more than the original trunk geometry had.

For runtime comparisons, we generated several artificial point clouds on a spherical geometry. These point clouds have to contain a specified number n of points, and they have to be dense enough for SAE J1100 boxes, and as well the minimal distance between two points has to fulfil the threshold as in Algorithm 3.3. Figure 3.3 shows such dense point clouds. As can be seen, the points in the real trunk geometry are not equally distributed, in contrast to the artificial spherical point cloud: There are many regions with very fine modelled geometry, resulting in a high point density in these regions. Other regions have only been modelled by a single plane. There the thinning procedure could remove many points from the cloud, still preserving the features of the trunk geometry.

Having created the dense point cloud $\mathcal{P}$, we can now compute the discretised Minkowski Sum $P_{t,o} = \mathcal{P} \oplus B_{t,o}$ by uniting equal boxes that are equally oriented. With $|\mathcal{P}| = n$ we get n axis-oriented boxes $B_1, ..., B_n$. We want to compute the union $U_n = P_{t,o} = \bigcup_{i=1}^{n} B_i$ and especially its boundary (the part of the boundary which is inside the trunk) ∂U_n. Since the boxes are axis-oriented, the boundary ∂U_n can be represented as a set of axis-oriented rectangles R_n. The problem of measuring the volume of U_n is known as Klee's measure problem [36]. Yap et al. [47] presented an algorithm for the d-dimensional problem running in $O(n^{d/2} \log n)$ time. Unfortunately, they do not compute



(a) Real trunk geometry

(b) Random point cloud of a sphere

Figure 3.3: These point clouds are dense enough for all SAE J1100 boxes

an explicit representation of U_n. If we have the set R_n characterising the boundary ∂U_n it is easy to compute the volume of U_n in time $O(|R_n|)$ by summing the volumes of all signed parallelepipeds induced by the rectangles parallel to one of the three axis planes.

The algorithm for uniting the boxes $B_1, ..., B_n$ is incremental. This approach gives us the possibility to implement the necessary additional operations in the same way. Suppose we have computed R_{i-1} as boundary of U_{i-1} for the first $i-1$ boxes. We now consider the next box B_i and want to update R_{i-1} to R_i. We have to distinguish two cases.

1. $B_i \cap \partial U_{i-1} = \emptyset$. This means the new box does not intersect the existing boundary. In this case B_i either lies completely in the interior of U_{i-1} and therefore $R_i = R_{i-1}$, or B_i lies completely outside of U_{i-1} and thus $R_i = R_{i-1} \cup \partial B_i$.

2. $B_i \cap \partial U_{i-1} \neq \emptyset$. Now all rectangles $r \in R_{i-1}$ which lie completely inside B_i can be deleted. We denote the set of these rectangles $D = \{r \in R_{i-1} : r \subseteq B_i\}$. Let $S = \{r \in R_{i-1} : r \cap B_i \neq \emptyset\}$ be the set of rectangles of R_{i-1} lying partially within B_i. Each rectangle $r \in S$ has to be trimmed and decomposed into new rectangles which can be done by applying a trimming function t. Figure 3.4 illustrates the trimming and decomposition of rectangles. The set of new rectangles shall be denoted $T = \{t(r, B_i) : r \in S\}$. Parts of the boundary facets of B_i also contribute to the boundary ∂U_i. Therefore we examine the six facets of B_i when intersecting them with the rectangles of R_{i-1}. Each facet F_j is decomposed into regions inside and regions outside of U_{i-1}. A vertical decomposition of the regions outside of U_{i-1} yields a set A_j of new rectangles which have to be added to R_{i-1} in order to get R_i. This is illustrated in Figure 3.5. The complete procedure can be summarised

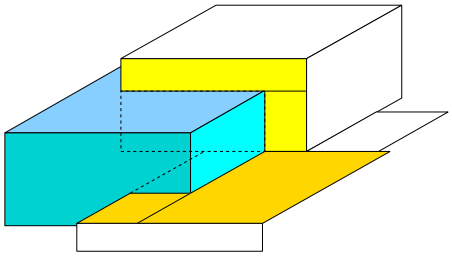


Figure 3.4: Rectangles intersecting B_i have to be trimmed and decomposed.

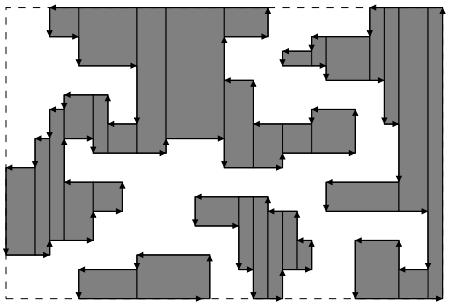


Figure 3.5: The facets of the added box have to be decomposed according to the boundary ∂U_{i-1}.

in the following equation:

$$R_i = R_{i-1} \setminus (D \cup S) \cup T \cup \bigcup_{j=1}^{6} A_j. \tag{3.2}$$

The next step is to determine rectangles of the boundary ∂U_n which do not contribute to the corners of the valid space $P_{t,o}$. This can be solved by shooting a ray starting on each rectangle in axis direction to infinity and analysing the orientation of the first hit rectangle.

If the point cloud has been dense enough, the set R_n consists of at least two connected components – one outer hull and at least one inner boundary of U_n. We can identify the outer and inner boundaries by examining the signs of the volumes over these subsets. The volume of the outer hull will be positive; the volume of the inner boundary will be negative. We can now delete all rectangles of the outer hull: The union U_n represents all box placements which intersect the trunk geometry. As boxes also may not lie outside the trunk geometry, we can extend the union U_n to the outside and therefore remove the outer part of ∂U_n. However, as the algorithm is incremental, we can not remove unnecessary rectangles right from the start, because from the point cloud we can not distinguish between inside and outside of the trunk geometry.

Figure 3.6 shows the approximate Minkowski Sum of a trunk and the smallest box (type H). In Figure 3.7 the Minkowski Sum is compared directly to the trunk geometry.

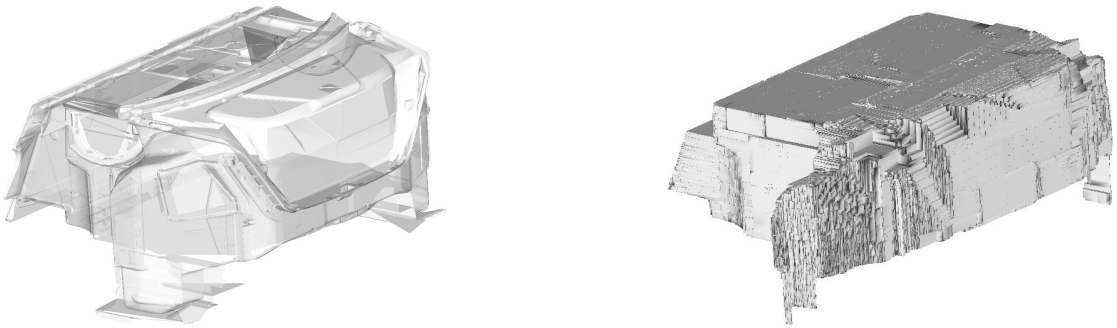


Figure 3.6: The Minkowski Sum of a trunk

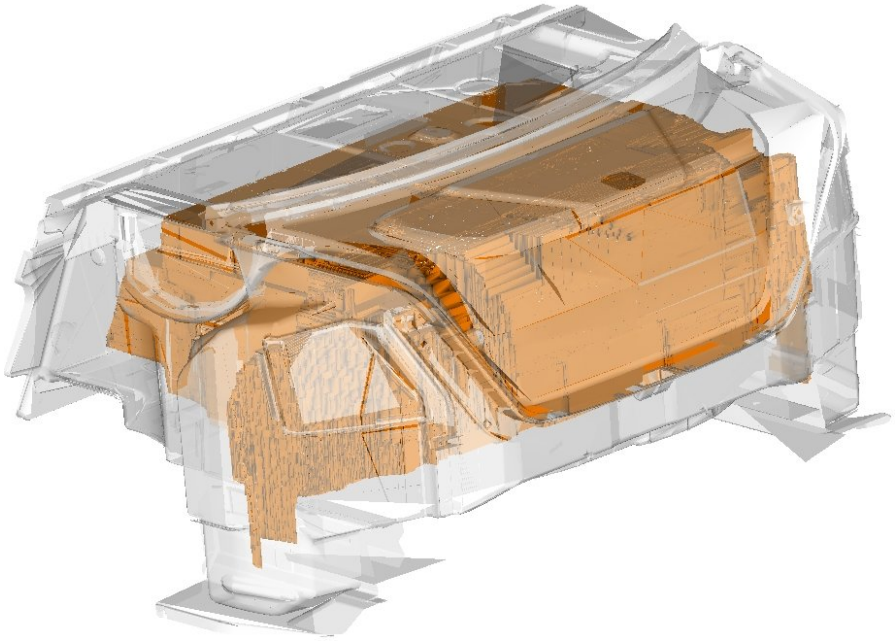


Figure 3.7: The Minkowski Sum compared to the trunk geometry

Having computed the set R_n of rectangles, it is quite easy to get all corners of U_n. We can also use this algorithm for our addition/deletion operation during the run of Algorithm 3.2 (line 8). Now that we can distinguish between necessary and unnecessary rectangles, we are able to remove outer rectangles immediately.

The computation of the Minkowski Sum of a point cloud and a box is repeated below as Algorithm 3.4.

Algorithm 3.4 Minkowski Sum of a point cloud and a box

Input: point cloud $\mathcal{P} = p_1, \ldots, p_n$

1: $U_0 = \emptyset$, $R_0 = \emptyset$
2: **for** (all points $p_i, i = 1..n$)
3: **if** ($B_i \cap \partial U_{i-1} = \emptyset$)
4: **if** ($B_i \subseteq U_{i-1}$)
5: $R_i = R_{i-1}$
6: **else**
7: $R_i = R_{i-1} \cup \partial B_i$
8: **else if** ($B_i \cap \partial U_{i-1} \neq \emptyset$)
9: $D = S = T = A = \emptyset$
10: **for** (all $r \in R_{i-1}$)
11: **if** ($r \subseteq B_i$)
12: $D = D \cup \{r\}$
13: **else if** ($r \cap B_i \neq \emptyset$)
14: $S = S \cup \{r\}$
15: **for** (all $r \in S$)
16: trim and decompose r, add the resulting rectangles to T
17: decompose all facets of B_i according to U_{i-1} and add the resulting rectangles to A.
18: $R_i = R_{i-1} \setminus (D \cup S) \cup T \cup A$ (see Equation (3.2)).

We still have to decide on a data structure which ensures an efficient and easy access to the rectangles.

3.3.2 Minkowski Sums: Efficient Data Access

During the execution of Algorithm 3.2, the Minkowski Sums are repeatedly altered by adding and removing boxes. These operations have to be very efficient because in a brute force algorithm many combinations of boxes are tried. So several thousands of add/delete operations have to be performed per second. The main task during the addition and deletion of boxes is to determine the rectangles affected by the new box B_i (the sets D and S in Algorithm 3.4).

In line 10 of Algorithm 3.4, the rectangles are determined by examining each rectangle for intersection with B_i. Although the intersection itself can be determined quickly for one rectangle, we still have many rectangles in our data structure. Therefore it is obvious to use a spatial organisation of the rectangles. Two types are very promising:

- A tree-like structure – in our case we tried an octree for easier range queries.
- A space partition similar to the partition described in [48].

We first use an octree structure to divide the search space. So we have to deal with two disadvantages against a classical kd-tree: First, the theoretical performance of $O(\log|R_i|)$ cannot be reached because it is impossible to divide the rectangles by their median. Even if this was possible by some criterion (such as the centre of the rectangles), some of the rectangles will stretch over the cell boundaries into other cells. So we have to divide the space instead. This can be done by the octree. Now we still have to deal with the size of the rectangles. Large rectangles may cover many small tree cells, because there are regions with high point density, and this creates many small rectangles as well. There three cases are possible: First, we leave all unchanged and live with the fact that we have to store the large rectangles in many cells. Second, we split the rectangles such that they fit into a cell. This increases the total number of rectangles significantly. Recall that we use the corners of the rectangles as placements in the packing algorithm. However, in the octree we could gain a small advance because not the whole large rectangle is examined during the range query. Third, we can also restrict the cell size to a fixed minimal value, which ensures that one large rectangle will cover only few cells. With this last method we get many rectangles within each tree cell.

We experimented with all three methods and determined the third method to be the best of the three rectangle treatments within the octree. The cell size should be chosen such that the smallest face of an H-box covers not more than four cells.

Still the range query itself remains. For a given box B_i, we have to find all rectangles intersecting this box. This is now done by determining the tree cells intersecting B_i and afterwards checking each rectangle within these cells. Now the cells are still well below the root node of the octree. The tree search is rather time consuming because each tree node has got eight children, if the number of rectangles in it is too large to be stored in the node itself. Hence in the worst case we have a box intersecting all eight children of the root node. Then we have to climb the whole tree height in every child of the root. So if the tree was broader, we could improve the search time. The broadest "octree" one could imagine is the space partition [48]. In it each cell is directly accessible by its position, which increases the search performance significantly. Again we experimented with several cell sizes and finally set the cell spacing to 200 mm. This value ensures that the smallest rectangle – the small face of an H-box – will cover not more than four cells.

By profiling our code, we observed that using the octree with limited cell size around 70% of the total runtime of Algorithm 3.4 is spent on range queries, whereas the space partition only uses 50% of this runtime.

Now that we know about the storage of the rectangles, we have to deal with the addition and deletion of boxes during the packing algorithm.

The addition of a box can generally be done by applying Algorithm 3.4. Deleting a box implies to restore the original Minkowski Sum. This means we have to keep all rectangles that would be deleted during the addition step. The deletion of an arbitrary box leads to a similar computation as in Algorithm 3.4. In our case, however, the deleted

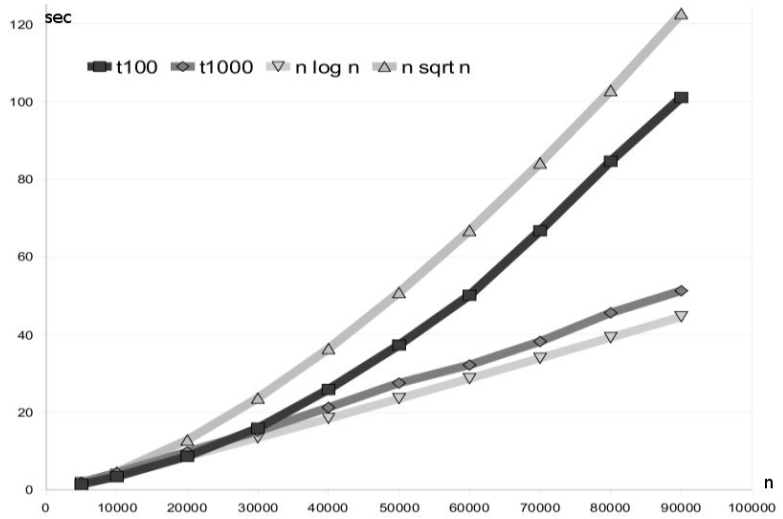


Figure 3.8: The runtime to compute our Minkowski Sum approximation depends on the density of the point set and on the number n of points.

box is not arbitrarily chosen from all (added) boxes, but it is always the most recently added box. This means our deletion step is only an undo process within the packing algorithm, simply restoring the previous state.

Since Algorithm 3.2 performs an enumeration of solutions, boxes are only temporarily added. This enables us to store the changes of an addition process on a stack. An addition of a box then consists of some added and some removed rectangles. If the box is removed, the topmost entry of the `changes`-stack is retrieved and the original rectangle set is restored. This eliminates the effort of copying the whole data structure for the next call.

Still the data structure will be accessed many times, especially in the leaves of the recursion tree. In order to avoid too many access operations in the last level, we store the results of the addition process only temporary into a buffer. The space partition is updated only if the next operation is another addition of a box. If the next operation removes the box, we simply clear the buffer, and the set of rectangles is left untouched. So the add/delete operations on the data structure are reduced to a minimum. Therefore, the search for the intersecting rectangles in the addition step is the main task during the run of Algorithm 3.2.

3.3.3 Minkowski Sums: Complexity

Although the theoretical running time of the incremental algorithm is quadratic in n, its experimental performance is far better at least in our application of computing Minkowski Sums between a point set and a box. The running time strongly depends on the relation between the density of the point cloud and the size of the boxes.

Figure 3.8 shows this dependency for artificial spherical trunk geometries (see Fig-

ure 3.3(b)) of increasing size. Four curves are displayed in this diagram. The two outer curves mark asymptotic bounds ($n\sqrt{n}$ and $n \log n$), the two inner curves show the runtime for two different densities. The density is measured as largest possible distance between one point and its nearest neighbour. A logical consequence is that point sets containing more points in the same density also have a larger diameter. If the density is increased (that means the largest distance between nearest neighbours is reduced) without changing the number of points, we have to shrink the diameter of the point set. As can be seen in Figure 3.8, the asymptotic runtime strongly depends on the density of the point cloud. A high density obviously suggests a runtime of $O(n\sqrt{n})$, whereas the runtime for a low density seems to stay within $\Theta(n \log n)$. The runtime also depends on the size of the boxes. Large boxes lead to more interferences between existing rectangles than small boxes.

Generally, the runtimes are higher for the spherical point clouds than for real trunk geometries. The complete calculation of the approximate Minkowski Sums for a real trunk ($\approx$ 20,000 points) takes around 45 seconds for all 42 combinations of box type and orientation altogether, whereas the runtimes displayed in Figure 3.8 refer only to one box type.

This runtime is possible due to a significant observation. If we compute the Minkowski Sum for the smallest box type, we can determine some points N of the point cloud which do not contribute anything to the inner boundary of the Minkowski Sum. Since this inner volume shrinks with larger boxes, we can be sure that the points of N will indeed never create a rectangle in our desired structure. So we can safely delete these points and continue the computation of the Minkowski Sums using the now smaller point cloud. This effect comes in handy if there are dense regions of points, because there the points are clustered such that only few of them actually have an influence on the Minkowski Sum. Obviously such regions seldom occur in the artificial spherical point clouds, but in our case of real trunk geometries those clusters are quite common.

As well, the runtime also depends on the size of the box type itself: If the box is large, we get many rectangles in the range of each box, thus the runtime rises. Small boxes cause only few interferences with existing rectangles, which has a positive effect on the runtime. So we can kill two birds with one stone: we start with a small box, and after we have computed this Minkowski Sum, we can eliminate many points and continue the computation with a far smaller point set on the large boxes. We therefore use a hierarchical order of the box types. At first we perform a preprocessing on the points by computing the Minkowski Sum with a cube having the smallest side length of an H-box (114 mm). As we may later use this Minkowski Sum again, we shall denote it S. Afterwards we remove the useless points of the point cloud and apply Algorithm 3.4 to the reduced point cloud with the hierarchy displayed in Table 3.1 for each orientation. This partial ordering of the box types ensures that all axis lengths are increasing.

3.3.4 Conflict Graph: Reduction?

The conflict graph for the continuous case is in principle an infinite graph. However, for the algorithm only a finite set of vertices will be needed. Particularly, the impor-

Type	A	B	C	D	E	F	H
Predecessor	D	H	F	F	H	B	–
Successor	–	F	–	A	–	C,D	B,E

Table 3.1: The hierarchy of box types for computing the approximate Minkowski Sums

tant vertices are derived from the corners of the Minkowski Sums. Unfortunately each Minkowski Sum – as we create them in Algorithm 3.4 – can contain up to $10,000$ corners, summing up to $4 \cdot 10^5$ vertices at the boundary. During the run of the packing algorithm additional vertices will be added, mostly around 30 per iteration. So it would be advantageous to reduce the conflict graph size in a similar way as stated in section 2.5.

To accomplish a sufficient reduction we would have to calculate the vertex degrees. This means to add every box to the Minkowski Sum and compute the unusable volume (see section 3.2). Afterwards, we have to examine all pairs of vertices (of the same type) to determine which vertex has to be eliminated. With a total of $4 \cdot 10^5$ vertices, this has a great impact on the preprocessing runtime.

Instead of computing the exact degree, which would include the calculation of 42 Minkowski Sums, we just use the Minkowski Sum S of the previous section. The vertex degree is computed by adding a box to S, computing the disabled volume, and removing the box from S. In fact, since the box will be removed right after the volume calculation, just the newly created and removed rectangles are computed but the data structure is left untouched. This ensures a fast computation of this relaxated version of the vertex degree.

Unfortunately, even this operation takes around 10 milliseconds per box. With $4 \cdot 10^5$ vertices, the complete calculation of the vertex degrees will take over an hour. Including the comparison of the respective neighbourhoods, the graph reduction will likely exceed the runtime benefits of a smaller conflict graph. Instead, the neighbourhoods of two vertices are compared in a more approximate manner, resulting in an even more relaxated graph reduction.

Consider two vertices u, v of the same type t with orientations o_u, o_v of the conflict graph. These two vertices lie on the boundaries of the Minkowski Sums P_{t,o_u}, P_{t,o_v}, and their corresponding boxes would intersect. Now one of the two vertices (u) may be removed if its neighbourhood $N(u)$ contains the complete neighbourhood $N(v)$ of the other vertex.

How can we determine whether this inclusion holds? First of all, the neighbourhood of a vertex u is defined as union over all Minkowski Sums, $N(u) = \bigcup_{t,o} N_{P_{t,o}}(u)$, where

$$N_{P_{t,o}}(u) = \{v = (x, y, z, t, o) \in V : (x, y, z) \in P_{t,o} \wedge \text{ the boxes } u \text{ and } v \text{ intersect}\}.$$

As stated above, the sets $N_{P_{t,o}}$ are disjoint, so the inclusion has to hold for each of the 42 combinations of type and orientation. To avoid this rather expensive test, we make the following considerations:

We already know that S is at least the union of all Minkowski Sums $P_{t,o}$[1]. It is also

[1]Generally, S is derived from a cube with a small side length. Obviously a small cube will fit in more

clear that a vertex u can only be removed if there is another vertex v whose neighbourhood is completely contained in the neighbourhood of u. We can't check the complete inclusion. Instead, we use a sufficient condition for the inclusion: We can easily and quickly check whether all vertices covered by the box of v are also covered by the box of u. If so, then we surely know that the neighbourhood of v is completely contained in the neighbourhood of u and u may be removed. Formally we determine the conflict region $C(u)$ showing all points covered by the box of u, and the region $C(v)$ likewise. These two regions are the same as the corresponding boxes (the dark shaded region in Figure 3.2). Now we can check whether $C(u) \cap S$ is a superset of $C(v) \cap S$. This can be done using interval arithmetics. Since both sets $C(u)$ and $C(v)$ are cut by S, we only have to check for each dimension whether $C(u)$ is on both sides larger than $C(v)$, or whether it reaches the bounds of S on one side and is on the opposite side larger than $C(v)$. In Figure 3.9 the relation between two such placements is illustrated: The dark shaded rectangle is compared to the placement in the lower left corner of the geometry. Since the conflict region of the central box is a superset of the intersection of Minkowski Sum and the other conflict region, we can safely ignore this central placement. If we leave the precise structure of S aside and use only the bounding box of S, then the test will even be faster. It is easy to see that we can forget about the shape of S and assume it to be also of cuboid shape, since if a box reaches the boundary of the bounding box of S it also reaches the boundary of S itself. Let therefore be I_M^d be the projection of all points of $M \subset \mathbb{R}^3$ on the d-th axis. Now it is checked for each dimension d whether

$$I_{C(u)}^d \supseteq (I_{C(v)}^d \cap I_S^d). \tag{3.3}$$

If Relation (3.3) is fulfilled for all dimensions, then u can safely be deleted from the conflict graph by Corollary 2.17.

These are only interval computations and can be checked quickly. With this procedure, the number of start vertices can be reduced by 75%, resulting in a conflict graph with around $25,000$ vertices for a less complex geometry. Nevertheless we have made many relaxations. This means we underestimate the amount of deleted vertices. Therefore the reduction success is rather low compared to the amount of vertices that would have been eliminated if we had the time for an exact inclusion test.

Relation (3.3) can also be applied to the addition of vertices during the run of the algorithm. Normally each step results in the addition of at least four new vertices for each Minkowski Sum, that is, if the cross-section of the Minkowski Sum has got rectangular shape. In fact, the amount of added vertices varies between 30 and 150, combined for all Minkowski Sums per step. Using the reduction mechanism, most of the added vertices can be eliminated, and finally only 200 vertices per $10,000$ steps will be added during the run of the algorithm, which is a fraction of $\approx \frac{1}{4000}$ of the vertices generated during the addition steps. Without this reduction the packing algorithm would get stuck by trying many equivalent placement combinations.

places than a box with two large and only one small side lengths.

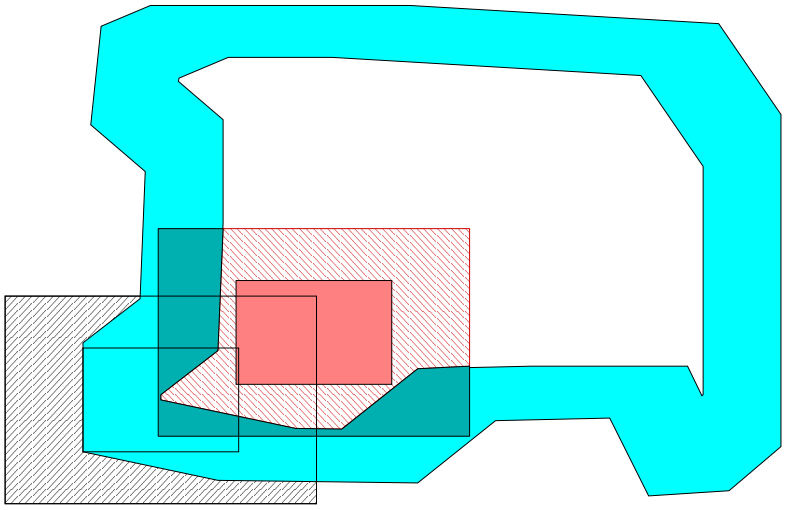


Figure 3.9: The central placement of the rectangle can be safely ignored

3.3.5 Algorithm Details

Now a closer look at Algorithm 3.1 seems appropriate.

Preprocessing

At first we have to compute the Minkowski Sums of the trunk geometry with each box type in every orientation. For the general SAE case, this means to calculate 42 sets. As mentioned above in section 3.3.1, the exact computation of the Minkowski Sum is rather expensive. Experiments with the Minkowski Sum algorithm of [31] have shown a running time of about 40 min for each Minkowski Sum. Therefore the geometry of the trunk is approximated by a dense set of points on integer coordinates, and then the Minkowski Sum is represented as a union of boxes, computed by using Algorithm 3.4.

Once the Minkowski Sums are computed, we need a first set of possible placements of boxes. Generally all points within a Minkowski Sum represent possible placements but not all points are good placements. As stated in [51], all valid packings of convex objects can be transformed into LUJ packings: each object is moved to the lower left and front side of the container without changing the relative positions of the objects. This fact can be relaxated to the following requirement:

Fact 3.8. *Each packing contains at least one object which is located at a corner of its set of possible placements.*

Hence it suffices to check all corners of the 42 Minkowski Sums. A Minkowski Sum of a large box will typically contain around $1,000$ corners, and the Minkowski Sum of a small box will have around $10,000$ corners. So the first set of vertices of the conflict graph can have around $10^5 \dots 10^6$ vertices. According to section 3.3.4, this set can be diminished by comparing the covered volumes of each vertex. Since we assume the boxes to be

axis-oriented, this reduction can be done using interval comparisons. So the starting conflict graph will have only $10^4 \dots 10^5$ vertices, depending on the size and detail of the trunk geometry.

The whole preprocessing, namely the calculation of the Minkowski Sums and the generation and reduction of the conflict graph vertices, can be performed within at most 15 minutes for the majority of our packing instances.

Addition and Deletion of Boxes

If a box B is added to an existing packing, some space is made unusable for more boxes. That means some of the conflict graph vertices have to be removed in the recursion step. On the other hand, according to Fact 3.8, additional boxes may now be placed such that they touch B. These placements, although within their respective Minkowski Sums, have not yet been qualified explicitly.

During the addition of box B, the Minkowski Sums of all box types have to be adapted to the new requirements, especially less available space. In detail, for each Minkowski Sum $P_{t',o'}$ of Box $B_{t',o'}$ we compute the set $P_{t',o'} \setminus (B \oplus B_{t',o'})$ which has all placements of a box of type t' in orientation o' which would intersect B eliminated. Of the new part of the boundary we get an additional set of vertices for the conflict graph. This set can also be reduced in the same way as in the preprocessing. In general, only few vertices will be added to the conflict graph.

On the return of a recursion step the box B has to be removed from the current packing. The Minkowski Sums have to be restored to their previous values as well.

In order to ensure these operations being efficient, we make use of the data structures described in section 3.3.2:

First we use a list *rect* of all rectangles we encounter during the whole algorithm. The rectangles are organised within a space partition T to ensure fast spatial access to the rectangles. For the addition and deletion of boxes we establish a buffer buf of rectangles and two stacks for added (*added*) and removed (*removed*) rectangles. Each of the latter structures use only references in order to save memory and initialisation time.

If a new box B_1 is added to the Minkowski Sum, we call Algorithm 3.4 to determine added and deleted rectangles. These are stored into the buffer, and their corners are returned as new vertices of the conflict graph. The removed corners are not deleted from the conflict graph because after the deletion of box B_1 we may have to use these vertices. If the next step is the deletion of B_1, then simply the buffer is cleared. Otherwise, another box B_2 is added. Then the content of the buffer is transferred to the list of rectangles *rect* and to the two stacks *added* and *removed*, and the space partition T is recomputed. Since the space partition works on a grid, it is sufficient to update the cells covered by the box B_1 only. Some time afterwards, the box B_1 will have to be deleted as well. Then the stacks *added* and *removed* have on top the rectangles which were added (deleted, resp.) during the addition of B_1. These rectangles have now to be restored, which means to eliminate the rectangles of *added* out of the space partition T and afterwards to remove them out of the rectangle list *rect*. The rectangles of *removed* have to be put back into the space partition T and the list *rect*.

Branching

After several recursion steps the first packing has been generated. Now the size of this packing can be used as lower bound for the remainder of the algorithm. For further packings an upper bound can be used to decide whether this branch of the recursion tree should be pursued. This upper bound has to indicate the expected size of the best packing within the branch.

To achieve an upper bound to the best possible packing we have to examine the free vertices of the conflict graph. The fastest bound can be computed by counting the vertices of each box type and reducing the bound according to missing box types in the same way as in Algorithm 2.6. Unfortunately, the remaining vertices of the conflict graph lie in many cases very close to each other, and the bound from Algorithm 2.6 has no measurable effect.

However, the Minkowski Sums give another powerful tool to calculate an upper bound to the current independent set. Now we can use the free space to determine whether two or more boxes can be placed. Typically the extent of the Minkowski Sum does not provide enough space for two boxes.

The simultaneous placement of two boxes can be checked by measuring the bounding boxes of the Minkowski Sums: Let P_{t_1,o_1} and P_{t_2,o_2} be the Minkowski Sums of the box types t_1 and t_2 in orientations o_1 and o_2. These two boxes can only be placed simultaneously if there are points $p_1 \in P_{t_1,o_1}, p_2 \in P_{t_2,o_2}$ such that the box B_{t_1,o_1} placed at point p_1 and B_{t_2,o_2} placed at point p_2 would not intersect. This is fulfilled if the two points differ in one dimension more than the sum of the extents of the boxes. Formally,

$$\exists p_1 \in P_{t_1,o_1}, p_2 \in P_{t_2,o_2} : \exists d : |p_1^d - p_2^d| \geq B_{t_1,o_1}^d + B_{t_2,o_2}^d. \tag{3.4}$$

Theorem 3.9. *Condition (3.4) is equivalent to*

$$\exists d : \max\big(\max P_{t_1,o_1}^d - \min P_{t_2,o_2}^d, \max P_{t_2,o_2}^d - \min P_{t_1,o_1}^d\big) \geq B_{t_1,o_1}^d + B_{t_2,o_2}^d. \tag{3.5}$$

Proof. "(3.4)⇒(3.5)": Let d_0 be the dimension which fulfils (3.4) with points p_1 and p_2. Then

$$|p_1^{d_0} - p_2^{d_0}| = \max\big(p_1^{d_0} - p_2^{d_0}, p_2^{d_0} - p_1^{d_0}\big).$$

Since $\min P_{t_1,o_1}^{d_0} \leq p_1^{d_0} \leq \max P_{t_1,o_1}^{d_0}$ and analogous for p_2 hold, we get immediately

$$|p_1^{d_0} - p_2^{d_0}| \leq \max\big(\max P_{t_1,o_1}^{d_0} - \min P_{t_2,o_2}^{d_0}, \max P_{t_2,o_2}^{d_0} - \min P_{t_1,o_1}^{d_0}\big),$$

and (3.5) holds.

"(3.5)⇒(3.4)": Let d_0 be the dimension which fulfils (3.5). We consider the following four points: $p_1, q_1 \in P_{t_1,o_1}$ fulfilling $p_1^{d_0} = \max P_{t_1,o_1}^{d_0}$ and $q_1^{d_0} = \min P_{t_1,o_1}^{d_0}$, as well as $p_2, q_2 \in P_{t_2,o_2}$ fulfilling $p_2^{d_0} = \min P_{t_2,o_2}^{d_0}$ and $q_2^{d_0} = \max P_{t_2,o_2}^{d_0}$. One of the two pairs (p_1, p_2) and (q_1, q_2) fulfils condition (3.5) and therefore is also an example for (3.4). □

If this condition is violated for all combinations of one type t to any other type t', then the addition of a box of type t reduces the upper bound significantly. However,

this condition is expensive to check for all combinations of types and orientations (42^2 = 1764). Therefore we again try to relaxate this condition and only perform this test for equal types. For the majority of box types, the decision is always between no possible placement ($M_{t,o} = \varnothing$ for all orientations), two boxes may be placed (condition (3.4)), or only one box can be placed. There are still box types (A,B and H) with more than two allowed occurrences according to SAE J1100. The simultaneous placement of more than two boxes is described in section 3.4.2. Unfortunately there are too many possible relative placements of the boxes for an efficient computation.

3.4 Enumerating All Packings

Another way to find an optimal packing is to enumerate all possible combinations of box types and check whether there exists a layout of the boxes that would fit into the trunk geometry. Similar considerations have been made in [1], where the trunk geometry had been approximated by sets of linear inequalities, and the feasibility of a possible layout was checked by an LP solver. A more detailed look into this algorithm is taken in section 3.4.1.

Section 3.4.2 deals with the possibility to calculate a trunk packing directly using only Minkowski Sums.

3.4.1 Directed Layout Graphs

Schepers [51] shows how to compute packings of axis-oriented boxes within a cubic container of "unit" length. This is done by solving several linear programs describing the relative positions of the boxes and the boundary of the container.

This approach can be generalised to convex containers as described in [1]. Again we use the Minkowski Sums of section 3.1. Since the Minkowski Sum is not necessarily convex, we use a convex approximation of the set and insert some convex obstacles. If necessary, the obstacles can be nested such that the Minkowski Sum itself can be represented with an arbitrary level of detail.

Generally it is possible to start the convex approximation of the Minkowski Sum with the convex hull. The faces of the convex hull generate the linear inequalities we need for this approach. These inequalities are far too many for efficient feasibility checks, so we approximate the Minkowski Sum with a small number of planes such that the difference between the volume of the convex hull and the convex approximation is small enough. In most cases it suffices to use 12 inequalities for the convex approximation of the Minkowski Sum. In addition to this, we add nested obstacles to our approximation in order to increase the accuracy of the approximation.

Now we have to create inequalities to enforce non-overlapping boxes. This is done by simple constraints depending on the relative position of the boxes. We have to check the feasibility of a fixed set of boxes for each combination of relative positions: Two boxes are placed validly if they do not overlap. We can describe this forbidden region as a set of linear inequalities as well.

Now we have got all tools for our linear program.

$$\begin{aligned} \text{Maximise} \quad & \textstyle\sum_{i=0}^{k} V(B_i), \quad \text{s.t.} && (3.6)\\ p_i \quad & \in && P_{B_i} \quad (3.7)\\ p_i \quad & \notin && Obst_{B_i} \quad (3.8)\\ p_i - p_j \quad & \notin && B_i \oplus B_j. \quad (3.9) \end{aligned}$$

Line (3.6) expresses the search for a packing with maximal volume. Since the SAE J1100 standard enforces us to restrict the number of occurrences, we will omit the automated search by the external solver, but we will enumerate all possible combinations of box types to define upper and lower bounds. In the end of this section a detailed description of the sequence of box type combinations is given.

The conditions of line (3.7) are described as a set of linear inequalities of which all have to be fulfilled to ensure feasibility. This is equal to a convex region. The inequalities can be directly derived from the planes of the convex approximation. In contrast to this, lines (3.8) and (3.9) can be expressed by sets of linear inequalities of which at least one has to be fulfilled for each obstacle and each pair of placed boxes. This is equivalent to cutting a convex hole into the solution space. Since in a linear program all inequalities have to be fulfilled we have to choose one of the inequalities for each obstacle and pair of boxes.

Condition (3.9) is equal to the existence of a separating plane between the two boxes. If there exists a separating plane H, then the orientation of H can be chosen such that it is parallel to the two boxes [20]. That means we have to enumerate all orientations of the separating planes, that means for each pair B_1, B_2 of boxes, we decide whether the centre of B_2 lies left, right, above (but neither left nor right), below, in front of, or behind B_1 (see Figure 3.10). The same is done for the obstacle condition (3.8). Such a configuration is called "packing pattern".

A packing pattern can also be described by a set of directed layout graphs indicating for each dimension the constraint types. For example, if box A must have a greater x-coordinate than box B, the corresponding x-Graph would contain the edge (A, B). If the box B is required to have a greater z-coordinate, the edge (B, A) would be added to the corresponding z-graph. With this representation, some impossible configurations like loops can be easily detected using a DFS in each graph. Additionally, a general infeasibility is found if there is a chain of boxes which exceeds one dimension of the container.

From the remaining configurations we create linear programs and check these for feasibility. If one of the configurations leads to a solvable linear program, we know a lower bound for the packing program. If otherwise no packing pattern for a specified combination of box types has got a feasible linear program, we know for sure that no optimal packing can contain this box combination as a subset. Hence we can prune the enumeration tree during the run of the packing algorithm.

The complexity of the approach grows rapidly with the number of linear inequalities describing the trunk, especially the obstacles. Therefore we aim to describe a close approximation of the feasible region for the box centres by a small number of obstacles.

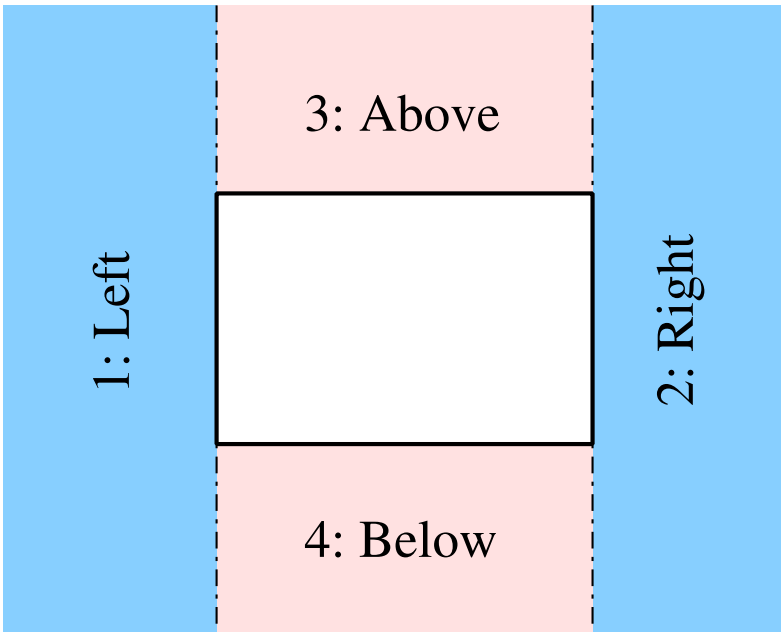


Figure 3.10: Separating "planes" for two-dimensional packing patterns

As the number of feasible packing patterns and therefore the number of linear programs to be solved increases rapidly with the size of the trunk, the enumeration of all packing patterns is very time consuming. For the manually created approximations, we are able to enumerate all packing patterns within a few days.

Algorithm 3.5 Packing Pattern algorithm

```
1: Create a rudimentary linear program from the trunk geometry
2: for ( each valid combination of box types C )
3:   for ( each arrangement of the boxes of C )
4:     Check whether there is a cycle in any of the layout graphs
5:     Check whether there is a chain of boxes which exceeds the trunk measurements
6:     Extend the linear program using the constraints of the layout graphs
7:     Solve the linear program using an external solver
```

In Algorithm 3.5 an external solver for linear programs is used. We have tried two different solvers – a commercial library (CPLEX [32]) and a free one (SOPLEX [60]). We found out that in our application the commercial library is about two times faster than the free library.

When a solution for the LP is found, the next larger combination of box types can be used. We have to arrange the box type combinations in a way such that an unsolvable combination can be used as a branching criterion for the further computation. It may also be possible to store all possible box type combinations (there are only $5^2 \cdot 3^4 \cdot 21 = 42525$ different combinations) into an array and mark all supersets of an unsolvable combination

as unsolvable. Then it is easy to check only combinations not marked as unsolvable.

In order to achieve large lower bounds in the beginning of the algorithm, we start – similar to Algorithm 2.6 – with packings containing large boxes, and increase the packing size by adding small boxes afterwards.

Algorithm 3.5 could also be formulated as an integer linear program. We would need the following additional items:

1. For each box type and orientation, a binary variable $B_{t,o}$
2. Additional equations, expressing the restrictions of the SAE J1100 standard
3. The placement inequalities have to be extended by the variables $B_{t,o}$ indicating whether the inequality has to be used
4. Finally, the pattern inequalities have to be modified, for example by introducing binary variables as well, indicating the packing pattern.

A solution to this ILP consists then of the combination of $B_{t,o}$ and the pattern variables in addition to the placement of the boxes. Due to the large number of additional variables we did not experiment with this approach. Nevertheless, we used a similar ILP formulation on a grid. The computation of a solution, however, did not lead to a runtime and quality improvement over the LP approach of Algorithm 3.5 and the grid based algorithms of chapter 2.

3.4.2 Direct Computation of Solutions

Daniels et al. show in [16] how to calculate the simultaneous placement of two or three convex polygons with fixed orientations within a two-dimensional container directly. However, their approach can also be used for higher dimensions as well as for boxes with axis-aligned orientations. The simultaneous placement of two boxes B_1, B_2 in the free spaces P_1, P_2 is rather easy and can be validated by computing the Minkowski Sum $B_1 \oplus B_2$ and check whether there are two points p_1, p_2 such that

$$p_1 \in P_1, p_2 \in P_2 \text{ and } (p_2 - p_1) \notin (B_1 \oplus B_2). \tag{3.10}$$

This condition can easily be verified because $B_1 \oplus B_2$ is just a box, hence it is only necessary to measure the dimensions of P_1 and P_2: For each dimension d we compute the projections $I^d_{P_1}$ and $I^d_{P_2}$ onto the d-th coordinate axis. Then we compute the differences between the smallest value of $I^d_{P_1}$ and the largest value of $I^d_{P_2}$ and vice versa. If any of these differences is at least as large as the d-extension of $B_1 \oplus B_2$, then we know that the two points p_1 and p_2 holding the condition above exist. If this equation does not hold for any dimension we know that the two boxes B_1 and B_2 do not fit into the container when the box orientations are chosen in this way.

This method is also used in Algorithm 3.1 to determine an upper bound to the current packing.

For three boxes, the determination of a valid placement is far more complicated. Avnaim [2] showed that it is generally impossible to calculate the simultaneous placement of more than two objects just by computing Minkowski Sums. That means we have to put some more computational effort into this algorithm.

According to [16], one has to fix the types and orientations of the three boxes first. Second, the orientations of the separating planes have to be determined. With three boxes there are always three separating planes, one for each pair of boxes. In this algorithm the orientation of the planes is important which means for three spatial dimensions a total of six possible orientations for each separating plane. This corresponds directly to the layout graphs of the previous section, now containing only three vertices.

Now, with fixed box types, orientations and separating planes one can check the feasibility of this layout of boxes within the trunk.

Definition 3.10. For two parameters $w \in \mathbb{R}^3, d \in \mathbb{R}$ let $H(w,d)$ be the half-space determined by $H(w,d) = \{p \in \mathbb{R}^3 : p \cdot w \geq d\}$.

Definition 3.11. Let B_1, B_2, B_3 be the boxes to be packed into a container C. For $1 \leq i \neq j \leq 3$, let w_{ij} be the normal vector of the separating plane between the boxes B_i and B_j. Note that $w_{ij} = -w_{ji}$. The set P_i marks the region of possible placements of box B_i in the container. Let furthermore

$$f_{P,w,w'}(d) = \max_{p \in P, p \cdot w \geq d} p \cdot w'$$

be a function determining the maximum w'-component of the points of $P \in \mathbb{R}^3$ lying in the half-space determined by the plane with normal w and offset d. Finally, let a_{ij} be one of the points of B_i which lies farthest in w_{ij}-direction ($\Leftrightarrow$ is nearest to B_j).

The function $f_{P,w,w'}$ is monotonically decreasing because the set of points in the corresponding half-space shrinks when d rises.

We can now calculate the feasibility of an arrangement of boxes B_1, B_2, B_3 with fixed separating plane orientations w_{12}, w_{13}, w_{23}. At first we have to compute the set D_{23} of pairs (d_{12}, d_{13}) such that there would be a placement of the boxes B_2, B_3 inside the half spaces $H(w_{12}, d_{12}) \cap C$ and $H(w_{13}, d_{13}) \cap C$, respectively, with a separating plane with normal w_{23}. This corresponds to the condition

$$f_{P_3,w_{13},w_{23}}(d_{13} - a_{31} \cdot w_{13}) - f_{P_2,w_{12},w_{32}}(d_{12} - a_{21} \cdot w_{12}) \geq (a_{23} - a_{32}) \cdot w_{23}. \tag{3.11}$$

On the left side of condition (3.11), the maximum components in w_{23}, w_{32}-direction of the valid areas P_2, P_3 of the boxes are determined and checked whether (on the right side) these placements would suffice to separate the lowest point of box B_2 from the highest point of B_3. The coordinates of (d_{12}, d_{13}) are bounded above and to the right by the domains of the functions $f_{P_3,w_{13},w_{23}}$ and $f_{P_2,w_{12},w_{32}}$. Additionally d_{12} depends on d_{13} in a piecewise linear function. Hence the set D_{23} can be described efficiently.

Now the set of valid positions for box B_1 has to be transformed into (w_{12}, w_{13})-space:

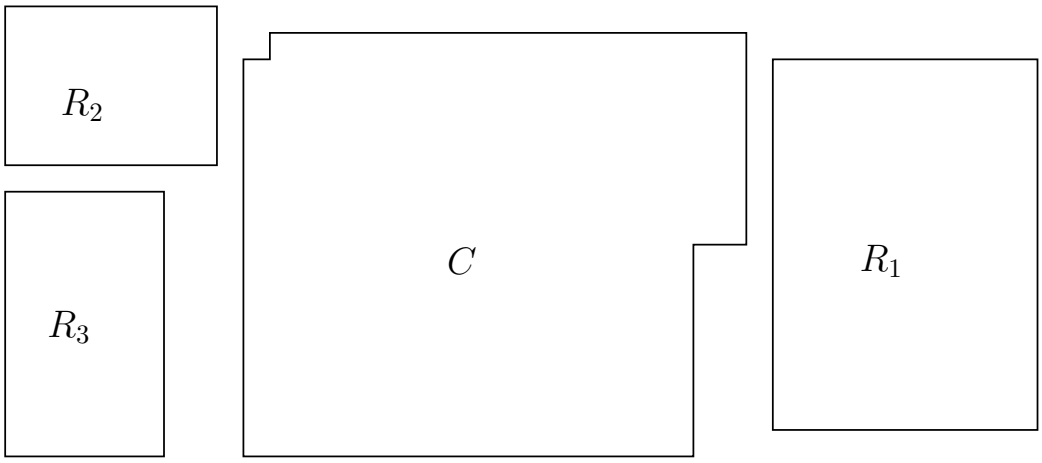


Figure 3.11: The two-dimensional container and the boxes to be packed

$$p \to ((p + a_{12}) \cdot w_{12}, (p + a_{13}) \cdot w_{13}). \tag{3.12}$$

If the transformed set P^T and the set D_{23} have any point p^T in common, then the preimage v of this point corresponds to a placement of box B_1 such that the remaining boxes B_2 and B_3 can be placed without intersections having separating planes with the chosen normal vectors w_{12}, w_{13}, w_{23}.

A small example

Given a container C and three rectangles R_1, R_2, R_3 with the following properties:

$$\begin{aligned}
C &= \overline{\{(a,b) : a < -7 \vee b < -8 \vee a > 12 \vee b > 8 \vee (a < -6 \wedge b > 7) \vee (a > 10 \wedge b < 0)\}} \\
R_1 &= \{(a,b) : -5 \le a \le 5 \wedge -7 \le b \le 7\} \\
R_2 &= \{(a,b) : -4 \le a \le 4 \wedge -3 \le b \le 3\} \\
R_3 &= \{(a,b) : -3 \le a \le 3 \wedge -5 \le b \le 5\}
\end{aligned}$$

Figure 3.11 shows an illustration of the container and the three rectangles to be packed within. The Minkowski Sums of the container complement and the rectangles, P_1, P_2, P_3 respectively, can be computed directly and are shown in Figure 3.12.

$$\begin{aligned}
P_1 &= \overline{\{(a_0 + a_1, b_0 + b_1) : (a_0, b_0) \in \overline{C} \wedge (a_1, b_1) \in R_1\}} \\
&= \overline{\{(a,b) : a < -2 \vee b < -1 \vee a > 7 \vee b > 1 \vee (a < -1 \wedge b > 0) \vee (a > 5 \wedge b < 7)\}} \\
&= \{(a,b) : -2 \le a \le 7 \wedge -1 \le b \le 1 \wedge (a \ge -1 \vee b \le 0) \wedge (a \le 5 \vee b \ge 7)\} \\
&= \{(a,b) : (-1 \le a \le 5 \wedge -1 \le b \le 1) \vee (-2 \le a \le 5 \wedge -1 \le b \le 0)\} \\
&= [(-1,-1),(5,1)] \cup [(-2,-1),(5,0)]
\end{aligned}$$

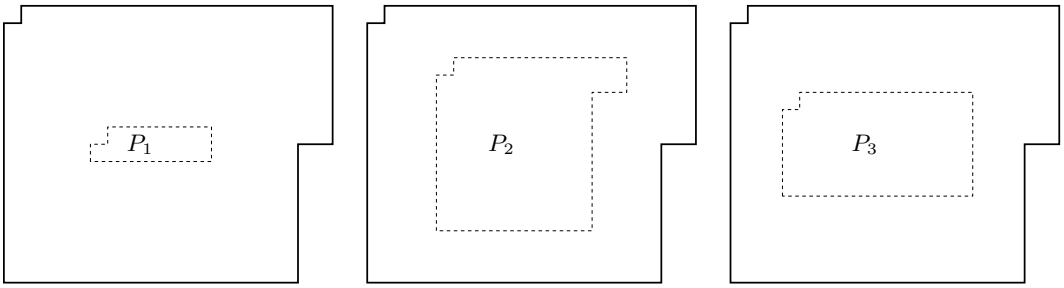


Figure 3.12: The Minkowski Sums P_1, P_2, P_3 of the boxes of Figure 3.11

The same calculations can be applied to determine P_2 and P_3:

$$P_2 = [(-2,3),(8,5)] \cup [(-3,-5),(6,4)]$$
$$P_3 = [(-3,-3),(7,3)] \cup [(-4,-3),(7,2)]$$

Now the different orientations w_{ij} of the separating hyperplanes have to be considered. In this example, we have four possible orientations for each pair of rectangles. For example one could use the triple $w_{12} = w_{13} = e_1$, $w_{23} = e_2$. That means R_1 has to be located to the *left* of both R_2 and R_3, whereas R_2 has to lie *below* R_3.

The points a_{ij} naturally lie at the border of R_i in w_{ij}-direction, so the calculation of these points is quite easy:

$$a_{12} = (5,\cdot), a_{21} = (-4,\cdot), a_{13} = (5,\cdot), a_{31} = (-3,\cdot), a_{23} = (\cdot,3), a_{32} = (\cdot,-5).$$

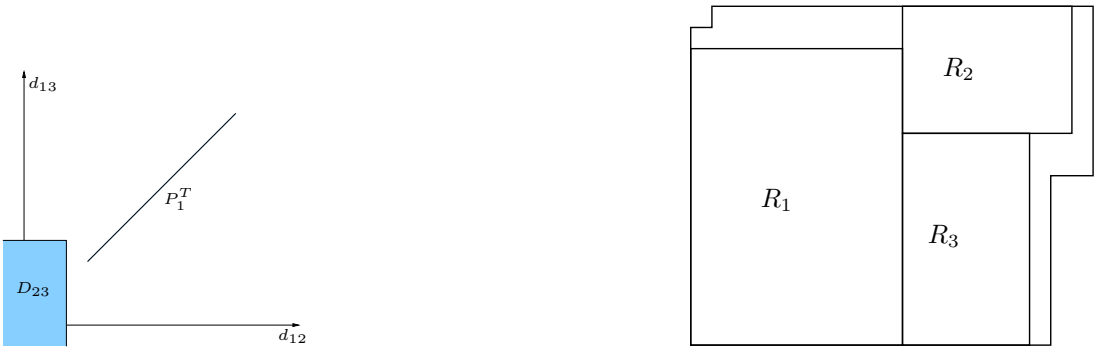
The required distance between the placements of R_2 and R_3 is computed by the difference between the lowest point of R_2 and the highest point of R_3 in w_{32}-direction.

$$(a_{32} - a_{23}) \cdot w_{32} = 8$$

According to condition (3.11), the functions $f_{P_i,w_{1i},w_{ji}}(d)$ have to be computed. In this example,

$$f_{P_3,e_1,e_2}(d) = 3, d \le 7 \Rightarrow d = d_{13} - a_{31} \cdot w_{13} = d_{13} + 3 \Rightarrow d_{13} \le 4$$
$$f_{P_2,e_1,-e_2}(d) = -5, d \le 6; 3, d \le 8 \Rightarrow d = d_{12} - a_{21} \cdot w_{13} = d_{12} + 4 \Rightarrow d_{23} \le 4$$

Finally, the set D_{23} and the transformed set P_1^T can be computed:



(a) The sets D_{23} and P_1^T

(b) The finally packed container

Figure 3.13: A representation of the solution space, and a valid solution to the 2-dimensional packing problem

$$
\begin{aligned}
D_{23} &= \{(d_{12}, d_{13}) : d_{13} \leq 4, d_{12} \leq 2\} \\
P_1^T &= \{((p + a_{12}) \cdot w_{12}, (p + a_{13}) \cdot w_{13}) : p \in P_1\} \\
&= \{(p_x + 5, p_x + 5) : p = (p_x, p_y) \in P_1\} \\
&= \{(a, a) : 3 \leq a \leq 10\} \\
P_1^T \cap D_{23} &= \varnothing
\end{aligned}
$$

The sets D_{23} and P_1^T are visualised in Figure 3.13(a). Since the intersection of these two sets is empty, there is no placement of the three rectangles within the container with the chosen orientations of the separating axes. On the other hand, if the orientation of w_{23} is flipped, that means the rectangle R_2 has to lie *above* R_3, then a solution can be computed. Since w_{12} and w_{13} remain unchanged, the transformed set P_1^T can be adopted as it stands.

$$
\begin{aligned}
a_{23} = (\cdot, -3), a_{32} &= (\cdot, 5) \\
(a_{32} - a_{23}) \cdot w_{32} &= 8
\end{aligned}
$$

$$
\begin{aligned}
f_{P_3, e_1, -e_2}(d) = -3, d \leq 7 &\Rightarrow d = d_{13} - a_{31} \cdot w_{13} = d_{13} + 3 \Rightarrow d_{13} \leq 4 \\
f_{P_2, e_1, e_2}(d) = 5, d \leq 8 &\Rightarrow d = d_{12} - a_{21} \cdot w_{13} = d_{12} + 4 \Rightarrow d_{23} \leq 4 \\
D_{23} &= \{(d_{12}, d_{13}) : d_{13} \leq 4, d_{12} \leq 4\} \\
P_1^T \cap D_{23} &= \{(a, a) : 3 \leq a \leq 4\}
\end{aligned}
$$

The non-empty intersection of P_1^T and D_{23} indicates that there is a placement of R_1 to the left of R_2 and R_3 such that R_2 lies above R_3: Take for example $a = 3$. Then the placement p_1 of the first rectangle has to be on the line $x = -2$. For the placements of the rectangles R_2 and R_3 we have to consider the values of d_{12} and d_{13}. Both have to be

greater or equal to 3. Hence for the x-coordinates of the rectangle placements we have $x(p_2) \geq 7$ and $x(p_3) \geq 6$. The y-coordinates can be directly obtained from the functions $f_{V_i,w_{1i},w_{ji}}(d_{1i} - a_{i1})$. Therefore we get the following solution of this packing problem:

$$p_1 = (-2, -1 \ldots 0), p_2 = (7 \ldots 8, 5), p_3 = (6 \ldots 7, -3).$$

Generally we have to enumerate all possible combinations of orientations w_{ij} of the separating planes. In two dimensions, we have to distinguish between four different orientations for each w_{12}, w_{13}, w_{23}. This gives a total of $4^3 = 64$ orientation combinations. In three dimensions, there are six possible orientations for each separating plane. Therefore we get $6^3 = 216$ different orientation combinations. If we just want to determine whether three fixed boxes with fixed orientations will fit into the remaining space, it is much faster to try to fix the placement of one box and perform the two-box-check.

As a side effect, these orientation combinations correspond to a layout graph as described in section 3.4.1.

An efficient algorithm?

It would be tempting to create an algorithm for placing whole sets of boxes simultaneously. Unfortunately the relative positions of boxes can only be expressed by a binary relation between only two boxes. Hence we have to use the directed layout graphs of section 3.4.1 again. For each possible layout graph (there are $\Theta(4^k)$ of them for k objects), we use an incremental algorithm: After we have found a placement of the first three objects, we transform the set of possible placements of the fourth object into the system of the three separating planes w_{41}, w_{42}, w_{43} to determine whether there is a valid placement of the fourth object with respect to the first three objects. This can be extended to the placement of k objects. However, the crucial point in this algorithm is the exponential number of layout graphs (=packing patterns or relative positions). In addition to this, the orientations of the objects have to be fixed a priori, that means each layout graph is established for every combination of box orientations (6^k). So we have to use the algorithm on $(6^k)^{4^k}$ different layouts if we want to pack k boxes. Of course, we can use similar preprocessing as in Algorithm 3.5, such as eliminating layout graphs with cycles and chains extending the container size. Still the number of layout graphs stays within the same order of magnitude.

If we compare this approach to Algorithm 3.5, we see that for two or three boxes the approach according to [16] provides a direct solution without the use of external LP solvers. Algorithm 3.5 will on the other hand be more efficient when using larger boxsets because not all solutions are returned but only one (which of course suffices for the packing problem).

4 Results

4.1 Implementation and Testsuites

The algorithms have been implemented in the programming language C++, and can be run on Unix- and Windows-based systems. Our industrial partner also uses a graphical interface for a comfortable selection and application of the packing algorithms.

For testing we have got several exemplary car trunks of different sizes and shapes. Unfortunately, most of the test instances are industrial secrets and can't be provided freely. All tests have been run on a 3 GHz Pentium D machine with 1 GB of RAM. Both Unix and Windows platforms have been used but no significant runtime differences have been observed.

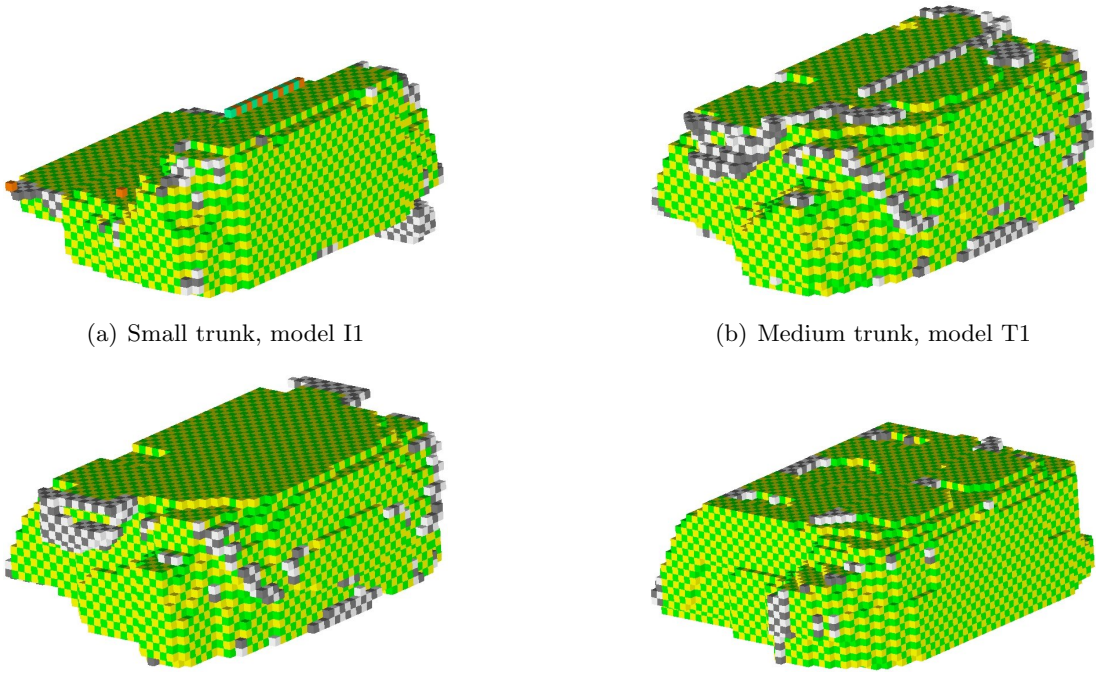
The majority of the tests have been performed on the trunk geometries shown in Figures 4.1 and 4.2. Unfortunately, the geometries themselves are not allowed to be shown freely because the corresponding car models are still in development. Hence we only show the interior of the trunk geometries: at first modelled as a grid with fine spacing (see chapter 2), and second the resulting Minkowski Sum of the trunk geometries and the cube with the smallest H-box length (see chapter 3). In the corresponding chapters the process of generating these structures is described in detail along with some comparing images.

For the mentioned geometries we could obtain "real" manual packings, that means packings (or the sizes of the packings) made by our industrial partner using conventional methods such as CAD tools etc. The reference packing sizes are shown in Table 4.1. Another comparison value, introduced in [11], is the ratio of continuous trunk volume to the packing volume (exploitation ratio), shown in the last column of Table 4.1.

As can be seen, the trunk geometries are almost equally complex in the number of vertices and triangles. The small trunk has got several pockets where only very small boxes can be placed, so the exploitation ratio is quite low. The distribution of the

trunk	vertices	triangles	cont. volume	with H-boxes	without H-boxes	ratio
Model I1	35062	50010	8.64 ft^3	5.382 ft^3	2.382 ft^3	67%
Model T1	38501	72872	14.92 ft^3	10.758 ft^3	9.358 ft^3	73%
Model T2	34901	65701	14.37 ft^3	10.520 ft^3	8.790 ft^3	70%
Model T3	32548	42182	18.53 ft^3	14.037 ft^3	11.837 ft^3	77%

Table 4.1: The continuous interior volumes, geometrical properties and the best manually achieved SAE packings of the tested trunks



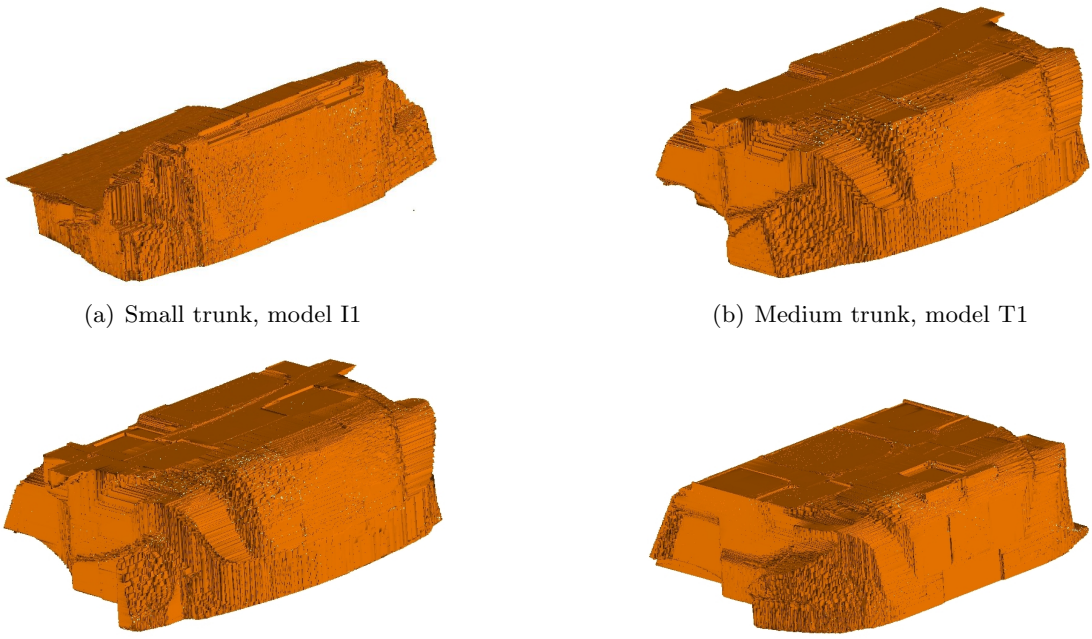
(a) Small trunk, model I1

(b) Medium trunk, model T1

(c) Medium trunk, model T2

(d) Large trunk, model T3

Figure 4.1: The grids of the four main trunk geometries discussed in this chapter



(a) Small trunk, model I1

(b) Medium trunk, model T1

(c) Medium trunk, model T2

(d) Large trunk, model T3

Figure 4.2: The Minkowski Sums of the four main trunk geometries discussed in this chapter

exploitation ratios can be observed throughout all trunk models: large trunks such as trunk T3 always have an exploitation ratio of around 80%.

As in the industrial application, the algorithm runtime – including preprocessing – was limited to at most 24 hours (night job). Packings for very large geometries could be computed within 72 hours (weekend job). Since not all runs could be completed within that timeframe, the results are merely the best packings after 24 (72, resp.) hours, not the optimal solution for this discretisation.

Section 4.2 deals with the quality of the obtained packings. Several criteria like space exploitation and packability are considered there. Nevertheless, we will have a look at runtimes in section 4.3. It is also quite interesting how far different backtracking algorithms can get within a certain time frame. Some implementation details and speedups as discussed in the previous chapters are quantified in section 4.3 as well.

4.2 Differences between Grid Packing and Continuous Packing

Throughout this work, three main packing approaches have been described: First of all, a grid-based approach with its difficulties on rounding different box sizes. A branch and bound strategy has already been presented in [1] and is here shown as Algorithm 2.6. There were four possibilities discussed for choosing a vertex and adapting the pruning conditions (see page 40). Second, there is Algorithm 3.5 using linear inequalities which was as well presented in [1]. The algorithm has been described in section 3.4.1 and is merely added for comparison. For this algorithm external software for solving linear programs is necessary. The third algorithm is another branch and bound algorithm but it is independent of a space discretisation. This algorithm is described in detail as Algorithm 3.1.

Table 4.1 shows the theoretical sizes of each tested trunk and packing sizes of manually obtained SAE packings. As can be seen, the percentage of the used space lies around $65-80\%$. Larger trunks can reach a higher space exploitation whereas some small trunk do not even reach the lower percentage. This is caused by the size of the SAE boxes. Corners of the volume can not be exploited very well, and the different box sizes lead to small gaps between boxes within a packing. It might be interesting to give a general description of the expected packing size in terms of the continuous interior volume and the surface of the trunk. The relationship between the surface/volume ratio and the exploitable volume is quite obvious.

If the H-boxes (see Table 1.1) are excluded from the packing process, the free space of the manual packings could not be re-used for larger boxes. Especially the small trunk has got an entire region that can only be packed with H-boxes. This space is not usable for all algorithms, for example the grid-based algorithms require a grid with the correct position to allow the placement of an H-box in this gap (see also Figure 2.4). Figure 4.3 shows this pocket where the H-boxes even can be placed in only one orientation.

The grid-based algorithms could all be finished within the timeframe of 24 hours, whereas the two continuous algorithms had to be stopped after that time. Table 4.2 shows the sizes of packings obtained using different grid spacings, compared to the

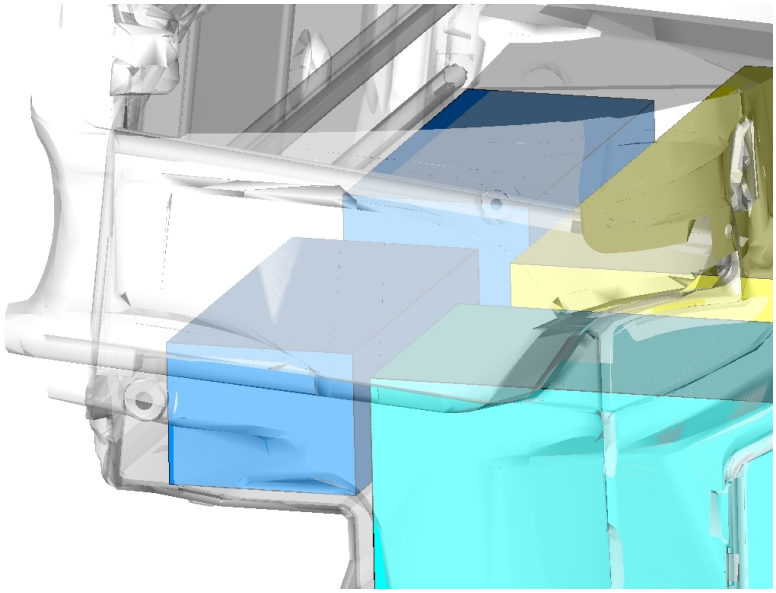


Figure 4.3: A narrow space where only boxes of type H can be placed

continuous algorithm (24 hours) and the manually created packing. The following observations can be made:

- For the grid-based algorithms, the best results could be obtained using the $1.5''$-grids. With the DIN packing problem, we had a strong correlation between grid spacing and solution quality – the smaller the grid spacing, the better the solution. Now the better solutions seem to occur using smaller grids of the $3''$-base.

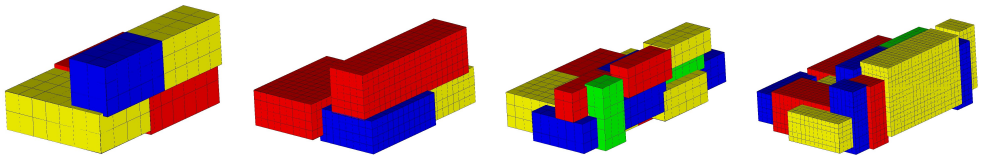
- Except for the large trunk, the continuous algorithm finds better solutions – even within the restricted time frame. Since the search space is much larger than on the grid, it would not have been surprising if it takes too much time to find an adequate solution.

- Almost all solutions without H-boxes are better than the results obtained by manual packings. Possible exceptions can be observed at the largest and the smallest grid spacing. The $3''$-grids provide a too rough approximation of the trunk – the geometry can not be used very well. In the $1''$ case, the problem size is very large, and still the rounding problem occurs. The rounding can obviously be done best for the $1.5''$ case. In addition to this, the manual packings without H-boxes are not optimal. Generally the packings have been created using all box types, and afterwards simply the H-boxes have been removed.

- When H-boxes are included, the conflict graph becomes very large and the reduction process (Algorithm 2.7) has a smaller effect on the conflict graph size. Hence the runtime of the algorithms is increased. Additionally, the manual packings are

trunk	manually	grid-based				Minkowski
		3″	2″	1.5″	1″	
without H-boxes						
Model I1	2.382	3.990	3.887	4.300	4.300	4.816
Model T1	9.358	8.538	9.207	9.418	9.056	9.927
Model T2	8.790	7.658	8.747	8.903	8.490	10.033
Model T3	11.837	10.872	12.465	12.674	12.622	12.561
with H-boxes						
Model I1	5.382	4.965	4.287	4.778	5.565	5.791
Model T1	10.758	10.981	10.526	10.393	10.126	9.782
Model T2	10.520	9.840	9.048	9.048	9.865	10.097
Model T3	14.030	11.486	12.193	12.566	—	13.561

Table 4.2: SAE Packing sizes after at most 24 hrs runtime on the tested trunks (in ft^3)



(a) 3″, no H-boxes (b) 1.5″, no H-boxes (c) 2″, with H-boxes (d) 1″, with H-boxes

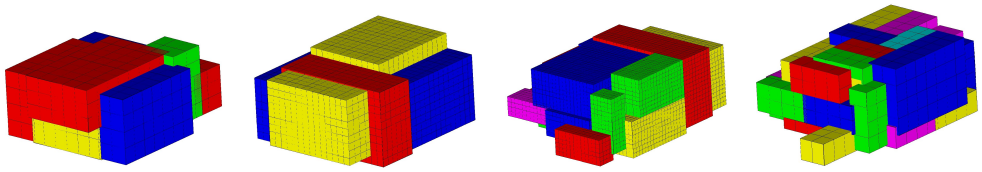
Figure 4.4: Grid packings for model I1

not axis-oriented as our packings. Without these additional degrees of freedom it is rather hard to find better solutions with H-boxes.

The Figures 4.4 through 4.7 show some of the resulting packings, namely the best and the worst grid packing for each geometry.

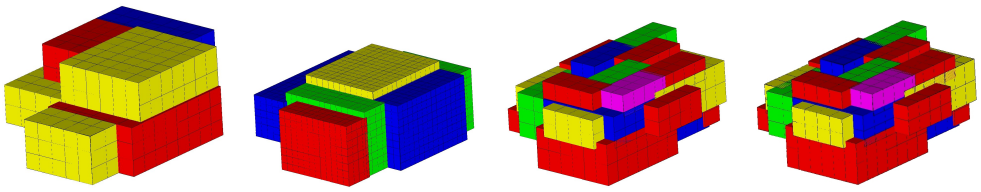
The magnification of a grid based packing (Figure 4.8) shows that intersections between grid boxes are inevitable. In section 6.2.2 we will present a framework to resolve these intersections.

In Figures 4.9 and 4.10 the results of the Minkowski Sum algorithm are shown for the trunk geometries of Figure 4.2. These packings are intersection-free because Algorithm 3.1 has been designed to avoid rounding problems by using arbitrary coordinates.



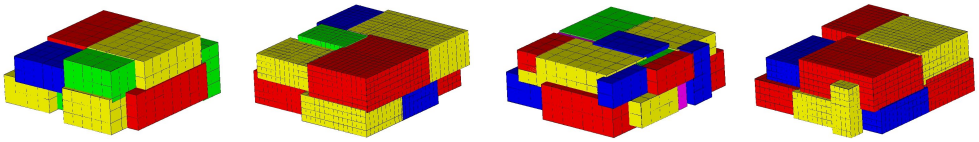
(a) 3″, no H-boxes (b) 1.5″, no H-boxes (c) 1″, with H-boxes (d) 3″, with H-boxes

Figure 4.5: Grid packings for model T1



(a) 3″, no H-boxes (b) 1.5″, no H-boxes (c) 2″, with H-boxes (d) 3″, with H-boxes

Figure 4.6: Grid packings for model T2



(a) 3″, no H-boxes (b) 1.5″, no H-boxes (c) 3″, with H-boxes (d) 1.5″, with H-boxes

Figure 4.7: Grid packings for model T3

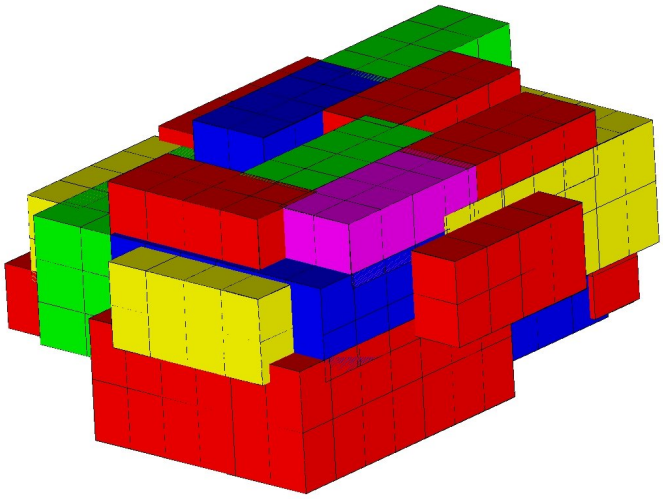
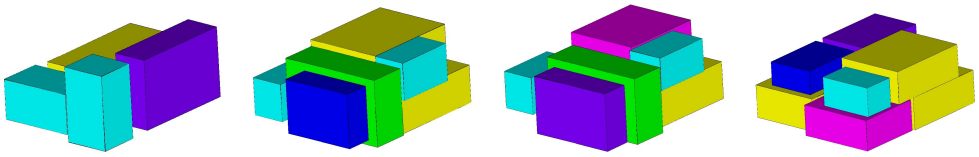
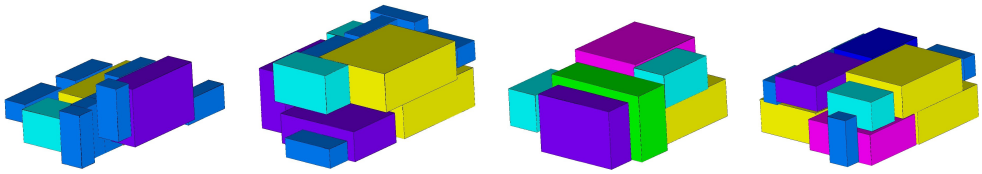


Figure 4.8: In the grid-based packings boxes may intersect



(a) Model I1 (b) Model T1 (c) Model T2 (d) Model T3

Figure 4.9: The solutions of the Minkowski Sum Algorithm 3.1 without H-boxes



(a) Model I1 (b) Model T1 (c) Model T2 (d) Model T3

Figure 4.10: The solutions of the Minkowski Sum Algorithm 3.1 with H-boxes

From now on, we use the best grid-based packing as indicator for comparison to packings achieved with Algorithm 3.1. In Table 4.3 the grid packings of all available trunk geometries are compared to the corresponding Minkowski packings after 24 hours. The models are named in the same way as in [48]. Different letters indicate different model types. If there are numbers behind the letters, there were just minor changes to the model or just parts of the model have been calculated. One exception has been made with the geometries T1, T2 and T3: These trunks have been derived from completely different cars and were not used in [48]. We used these geometries only for the SAE standard. The results are mixed: of overall 33 geometries, 23 Minkowski packings have been better than the grid packings. Mostly this discrepancy originates in the trunk shape. In many cases we could not get manual packings, and we mostly tested Algorithm 3.1 without H-boxes only. However, it should be no problem to run the Minkowski Sum algorithm including H-boxes on the remaining geometries as well.

When looking at the variation of the grid packings, 15 Minkowski packings lie within 5% deviation of the grid packings. As of the absolute deviation, 17 Minkowski packings lie within 0.5 ft^3 deviation from the grid packings. Two trunks have got a huge discrepancy between grid packing and Minkowski packing. The first one – model D – has got a very small continuous volume (less than 3 ft^3) and a special shape where almost only H-boxes can be placed. We managed to place two E-boxes by using Algorithm 3.1, but it was impossible to find a grid where two E-boxes would fit. Hence, by excluding H-boxes, the Minkowski packing exceeds the grid-based packing by 100%. The other extremum can be found as well within very small trunk geometries. In this case, a rather flat trunk with a large surface leads to the differences between grid packing and Minkowski packing. Large geometries – and therefore large optimal packings – lead to more similar results in Minkowski and grid-based packings.

4.3 Runtimes

All algorithms had to be terminated after at most 24 hours, except for large geometries, where 72 hours were allowed. In most cases the Minkowski Algorithm 3.1 could not examine all independent sets, hence the found solution was not guaranteed to be optimal.

Table 4.4 shows the conflict graph sizes for the four trunks of Table 4.1. As can be seen, the grid based conflict graphs are much smaller than the graphs resulting from the Minkowski sums. The conflict graph could be generated rather quickly. Table 4.5 shows

model		with H-boxes			without H-boxes		
	manually	grid	cont.	cont./grid	grid	cont.	cont./grid
A	–	7.04	–	–	7.04	7.4	105.1 %
B	–	9.11	–	–	9.11	9.62	105.6 %
C	–	12.25	–	–	12.25	11.68	95.3 %
D	–	1.6	1.83	114.1 %	0.63	1.25	200 %
G	–	1.96	–	–	1.96	2.07	105.6 %
H1	–	4.05	4.00	98.8 %	3.58	3	83.8 %
H2	6.74	6.51	6.77	104 %	5.65	5.79	102.5 %
I1	5.38	5.48	5.79	105.7 %	4.51	4.82	106.9 %
I2	–	8.06	8.7	107.9 %	7.51	8.02	106.8 %
J	–	8.77	–	–	8.69	8.95	103 %
K	–	8.85	–	–	8.85	9.26	104.6 %
L	–	9.78	–	–	9.78	10.39	106.2 %
M	–	11.64	–	–	11.64	11.22	96.4 %
N1	–	10.4	–	–	10.4	10.59	101.8 %
N2	–	10.3	–	–	10.3	10.81	104.9 %
N3	–	11.53	–	–	11.53	10.7	92.8 %
N4	–	12.16	–	–	12.16	12.56	103.3 %
O	–	12.67	–	–	12.67	11.99	94.6 %
O2	–	11.28	–	–	11.28	11.69	103.6 %
S4	–	12.41	–	–	12.41	12.88	103.8 %
S5	–	4.77	–	–	4.77	6.01	126 %
S6	–	3.99	–	–	3.42	3.67	107.3 %
T1	10.76	10.61	9.93	93.6 %	9.42	9.93	105.4 %
T2	10.52	9.84	10.1	102.6 %	8.9	10.03	112.7 %
T3	14.03	12.67	13.56	107 %	12.67	12.56	99.1 %
U	–	9.11	–	–	9.11	8.68	95.3 %
V1	–	11.64	–	–	11.64	12.52	107.6 %
V2	–	11.53	–	–	11.53	11.38	98.7 %
X	–	14.11	–	–	14.11	15.76	111.7 %
Z	–	6.94	–	–	6.94	7.24	104.3 %
AA	–	7.39	–	–	7.39	7.77	105.1 %
AB	–	12.98	–	–	12.98	13.5	104 %
AC	–	6.63	–	–	6.63	6.98	105.3 %

Table 4.3: Complete overview of all trunks and best packings with grid-based and continuous algorithms (in ft^3)

trunk	3″	2″	1.5″	1″	Minkowski
without H-boxes					
small (I1)	75	41	159	242	10051
medium 1 (T1)	810	2124	3747	7408	58417
medium 2 (T2)	789	1705	2785	5555	55041
large (T3)	982	2780	4982	11488	128634
with H-boxes					
small (I1)	599	661	1526	5351	107996
medium 1 (T1)	2993	7780	12840		166034
medium 2 (T2)	2516	6620	10683	34598	146303
large (T3)					300398

Table 4.4: Sizes of the reduced conflict graphs (# of vertices)

trunk	3″	2″	1.5″	1″	Minkowski
without H-boxes					
small (I1)	$< 0.1s$	$0.4s$	$2.9s$	$23s$	$4min46s$
medium 1 (T1)	$0.9s$	$10.2s$	$1min4s$	$14min6s$	$20min28s$
medium 2 (T2)	$0.7s$	$8.8s$	$50.6s$	$10min50s$	$18min14s$
large (T3)	$1.3s$	$19.7s$	$2min$	$30min$	$53min41s$
with H-boxes					
small (I1)	$0.2s$	$1.6s$	$7.1s$	$1min12s$	$23min45s$
medium 1 (T1)	$1.7s$	$20.7s$	$2min18s$		$50min$
medium 2 (T2)	$1.2s$	$17.2s$	$1min47s$	$32min43s$	$43min10s$
large (T3)					$2h6min$

Table 4.5: Total time to create the reduced conflict graphs

the runtimes needed for computing the conflict graphs. Although the Minkowski sum graphs are much larger than the grid-based graphs, the conflict graph computation does not exceed one hour runtime, except for the largest trunk when H-boxes are included.

An additional aspect is the time needed for reducing the conflict graph. In all instances listed here, the conflict graph reduction is performed as a part of the conflict graph creation. Overall, the graph reduction time is well below the total creation time, mostly between a fraction of $\frac{1}{7}$ to $\frac{1}{5}$ of the total creation time (see Table 2.4).

Now there are two parameters to be considered: First, the total runtime of the algorithms, if they could finish their run, and second, the time when the "final" solution was found. This parameter is important because one could want to kill the run of the algorithm. If an algorithm produces its best solution near to the end of its run, an early termination is not as useful as if the algorithm finds good solutions right in the beginning.

The grid-based algorithms of chapter 2 could be completed in most cases. Table 4.6 shows the runtimes our grid-based algorithms needed to find their best solution for

trunk	grid	naive	randomised	sorted	greedy
small (I1)	$3''$	$< 0.1s$	$< 0.1s$	$< 0.1s$	$< 0.1s$
small (I1)	$2''$	$< 0.1s$	$< 0.1s$	$< 0.1s$	$< 0.1s$
small (I1)	$1.5''$	$< 0.1s$	$< 0.1s$	$< 0.1s$	$< 0.1s$
small (I1)	$1''$	$0.2s$	$0.1s$	$0.1s$	$< 0.1s$
medium 1 (T1)	$3''$	$15.2s$	$1min2s$	$1.3s$	$2.0s$
medium 1 (T1)	$2''$	$33.2s$	$9h24min$	$7.3s$	$6.1s$
medium 1 (T1)	$1.5''$	$5h1min$	$15h2min^*$	$10min35s$	$38min16s$
medium 1 (T1)	$1''$			$41min$	
medium 2 (T2)	$3''$	$7.0s$	$15s$	$0.4s$	$0.7s$
medium 2 (T2)	$2''$	$6.8s$	$43min42s$	$2.1s$	$5.3s$
medium 2 (T2)	$1.5''$	$11.1s$	$8h57min^*$	$5.8s$	$24.7s$
medium 2 (T2)	$1''$			$1min54s$	$7min29s$
large (T3)	$3''$	$2min52s$	$47min58s$	$7.7s$	$1min$
large (T3)	$2''$	$14h3min$	$2min50s^*$	$46.9s$	$3min39s$
large (T3)	$1.5''$			$3min33s$	
large (T3)	$1''$			$3min42s$	

Table 4.6: Runtimes of the grid-based algorithms until the returned solution has been found. An asterisk (*) marks an incomplete run, that means the returned solution might be not optimal

several grid spacings. If the run could be completed, the returned solutions are really optimal for the grid. If not, the entry has been marked with an asterisk(*). In this table, the preprocessing, namely the computation of the conflict graph, is not included.

As can be seen, the fastest vertex ordering – grouped by type and then sorted by weighted degree – also provides its best solution very early in the algorithm run. Therefore it might also be possible to trim the runtime of this algorithm. Using the randomised vertex ordering, the best solution is found very late. The randomised vertex ordering also does not find the optimal solution within the maximal runtime of 24 hours.

Obviously the ordering by decreasing vertex weight gives the best runtimes. The second place is taken in the majority of our test cases by the adapted greedy algorithm. Therefore the decision scheme of the greedy algorithm is still very good. If the recalculation of the weighted degrees can be sped up this decision scheme can compete with the fixed order. If the vertices are taken by their creation time it depends on the conflict graph algorithm whether this scheme provides a good order. In our case the vertices are created typewise but not sorted by their weighted degree. It is quite lucky that the creation of the conflict graph starts with the largest boxes. So the naive algorithm without specific order had similar runtimes to the greedy scheme. Clearly the worst performance was shown by the randomised order. One could also have guessed this because a randomised order does not reflect the special properties of our combined graph problem according to the SAE J1100 standard.

Since all algorithms use basically the same recursion scheme, Table 4.7 is a good

trunk	grid	naive	randomised	sorted	greedy
small (I1)	$3''$	$< 0.1s$	$< 0.1s$	$< 0.1s$	$< 0.1s$
small (I1)	$2''$	$< 0.1s$	$< 0.1s$	$< 0.1s$	$< 0.1s$
small (I1)	$1.5''$	$< 0.1s$	$0.1s$	$< 0.1s$	$< 0.1s$
small (I1)	$1''$	$0.2s$	$0.2s$	$0.1s$	$0.1s$
medium 1 (T1)	$3''$	$18.8s$	$13min35s$	$3.3s$	$19.5s$
medium 1 (T1)	$2''$	$17min26s$	$11h11min$	$20.7s$	$1min53s$
medium 1 (T1)	$1.5''$	$10h14min$	$> 24h$	$11min22s$	$41min49s$
medium 1 (T1)	$1''$			$46min$	
medium 2 (T2)	$3''$	$7.0s$	$2min44s$	$1.6s$	$8.4s$
medium 2 (T2)	$2''$	$3min38s$	$2h10min$	$7.9s$	$51.9s$
medium 2 (T2)	$1.5''$	$44min10s$	$> 24h$	$1min20s$	$7min30s$
medium 2 (T2)	$1''$			$2min23s$	$10min13s$
large (T3)	$3''$	$3min39s$	$2h7min$	$7.7s$	$1min$
large (T3)	$2''$	$22h30min$	$> 24h$	$46.9s$	$6min28s$
large (T3)	$1.5''$			$45min$	
large (T3)	$1''$			$3h55min$	

Table 4.7: Runtimes of the grid-based algorithms needed for verifying the exact solution

indicator for the quality of the chosen vertex sequence. A mapping of the runtimes to the corresponding conflict graph sizes shows the following dependencies:

Generally, the algorithms' runtimes are exponential in the size of the conflict graph and the solution size. However, although a larger trunk with a large grid spacing and a small trunk with a small grid spacing may imply conflict graphs of the same sizes, then the large trunk will mostly produce the faster run of the algorithm. This is because many box combinations might be easily discarded – either there is no space for these too large boxes, or the remaining available boxes can not improve the current best – very large – packing.

Figure 4.11 shows the runtimes of the grid based Algorithm 2.6. However, not all instances are shown in this diagram because the runtime differences are obvious between the best variant – the vertices are sorted first by their weight and then by their degree – and all other variants. In Figure 4.12 the other grid instances are shown only with the best variant. Note for both figures the logarithmic scale. It ranges from 10^{-3} seconds to 10^5 seconds, which is roughly one day. Figure 4.11 shows clearly that the randomised variant takes a whole day for all instances with a grid spacing of less than $2''$, except for the small trunk. The variant called "sorted" (the vertices are grouped by type and then sorted by their weighted degree) is only for three instances not the fastest one. Of these, two runs could still be completed in less than one second. The third instance was finished earlier by the simple vertex ordering. In this case, the simple ordering returned a far smaller packing than the sorted variant. This is caused by the implementation of the SAE standard requirement that boxes may not be removed after H-boxes have been added. Otherwise the runtime of the simple vertex ordering would lie well above the

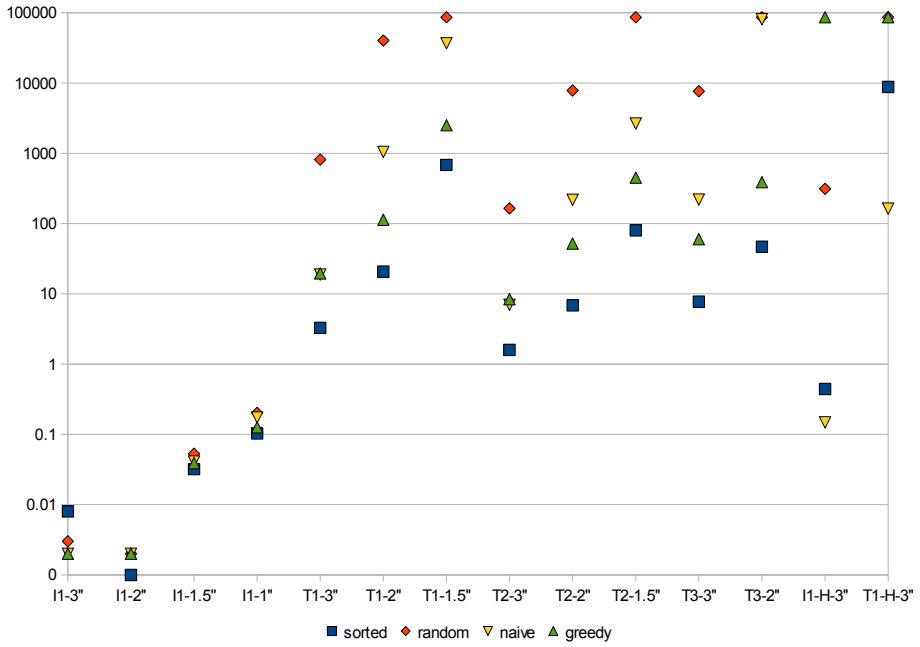


Figure 4.11: The runtimes of the four variants of Algorithm 2.6

best strategy.

We also modified the MWC algorithm of [46]. The algorithm was altered such that it would compute a valid SAE packing according to a given conflict graph. It is presented here as Algorithm 2.4. The runtimes are shown in Table 4.8.

For small instances of the conflict graphs, namely less than 500 vertices, the MWC algorithm performs quite well. This is caused by a very good upper bound to the possible weight of a clique. This bound, however, results from the solution of the same problem with $n-1$ vertices. Hence, the MWC algorithm of [46] is an incremental algorithm which can only use its strength in small conflict graphs. It also has to be taken into account that the MWC algorithm has been designed to find maximum weight cliques in general weighted graphs, in contrast to our Algorithm 2.6 which is used only for the trunk packing problem. Figure 4.13 displays the runtimes of the MWC algorithm for those instances where the algorithm could finish its run. The runtimes are compared to the runtimes of the four variants of Algorithm 2.6. It can be seen that for the latter three instances Algorithm 2.4 has got a similar runtime to the randomised version of Algorithm 2.6.

Finally we examine the runtimes of the Minkowski Sum algorithm.

As before, the important timestamp is the time when the returned solution is found. The runtimes of the first two instances suggest that the best solution will be found rather late in the run of the algorithm. Therefore an early cancellation of the algorithm might

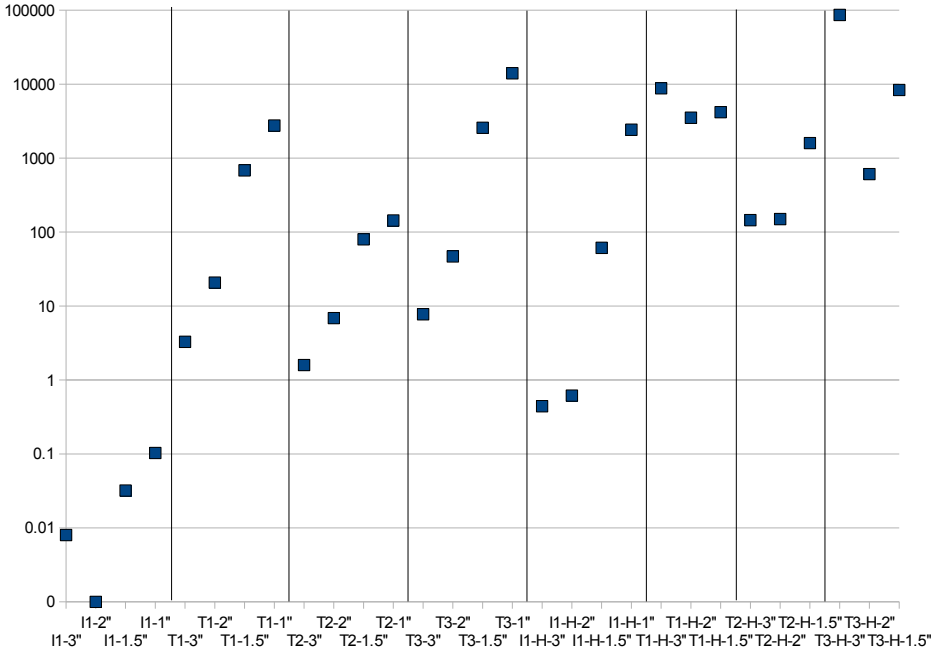


Figure 4.12: The runtimes of the best variant of Algorithm 2.6 for all grids

trunk	grid	$\|V\|$	Algorithm 2.6	MWC-algorithm [46]
small (I1)	$3''$	75	$< 0.1s$	$< 0.1s$
small (I1)	$2''$	41	$< 0.1s$	$< 0.1s$
small (I1)	$1.5''$	159	$< 0.1s$	$< 0.1s$
small (I1)	$1''$	242	$0.1s$	$0.1s$
medium 1 (T1)	$3''$	810	$3.3s$	$7min23s$
medium 1 (T1)	$2''$	2124	$20.7s$	$> 24h$
medium 1 (T1)	$1.5''$	3747	$11min22s$	$> 24h$
medium 1 (T1)	$1''$	7408	$46min$	$> 24h$
medium 2 (T2)	$3''$	789	$1.6s$	$1min48s$
medium 2 (T2)	$2''$	1705	$7.9s$	$> 24h$
medium 2 (T2)	$1.5''$	2785	$1min20s$	$> 24h$
medium 2 (T2)	$1''$	5555	$2min23s$	$> 24h$
large (T3)	$3''$	982	$7.7s$	$4h8min$
large (T3)	$2''$	2780	$46.9s$	$> 24h$
large (T3)	$1.5''$	4982	$45min$	$> 24h$
large (T3)	$1''$	11488	$3h55min$	$> 24h$

Table 4.8: The runtimes of Algorithm 2.6 compared to the MWC-algorithm of [46]

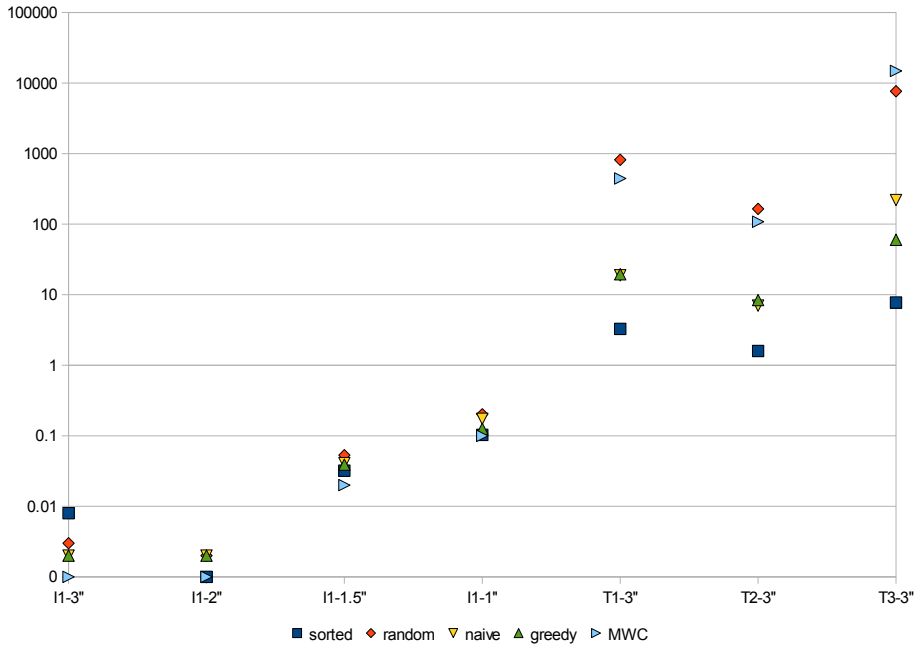


Figure 4.13: The runtimes of the grid based algorithms compared to the algorithm of Östergård [46]

	without H-boxes		with H-boxes	
trunk	T_{ret}	T_{total}	T_{ret}	T_{total}
small (I1)	$2min44s$	$3min31s$	$24.2s$	1 week
medium 1 (T1)	$19h10min$	$24h$	$1min16s$	1 week
medium 2 (T2)	$17min48s$	$24h$	$81h$	1 week
large (T3)	$1min9s$	$24h$	$3min23s$	1 week

Table 4.9: Runtimes of the Minkowski Sum algorithm. T_{ret} denotes the time needed to find the final solution, T_{total} denotes the total runtime of the algorithm

not be appropriate, even if there have been long periods without improvement of the solution.

We extended the runtime for the instances including H-boxes to one week, hoping to get better results. Table 4.9 suggests that this hope could not be fulfilled: The best found results for packings including H-boxes were only found in the beginning and could not be improved throughout the run of the algorithm.

4.4 Matching the Quality Requirements

In most cases the discrete packing algorithms could exceed the required quality to be at least the size of a manual packing. The remaining trunks could be packed within at least 90% of the best known manual packing.

Our software also allows to improve a computed packing by inserting additional boxes, rotating and moving existing boxes, and even changing the type of a box. Throughout these actions, the intersections between boxes and the trunk geometry can be monitored. Though it has been possible to improve on our algorithmic results using these techniques, the resulting packings can not be compared to our results because these packings use an additional degree of freedom, namely the rotation of boxes. This naturally extends the solution space, and an optimal packing might be larger than the best axis-oriented packing. However, it has been possible to find packings more than 20% larger than the previous best known manually derived optimum.

Unfortunately, the addition of H-boxes to the packing algorithm leads to a huge exceedance in the runtime. As shown in Tables 4.2 and 4.7, the found packings do not really justify the extended runtime. Another aspect is the violation of the SAE J1100 standard requirements. Although Algorithm 2.6 chooses vertices according to the volume of the boxes (largest first), a packing with maximum volume would contain as many H-boxes as possible. This contradicts to the requirement that H-boxes may only be added if no other boxes fit anymore. Our algorithm can take this requirement into account but this gives no guarantee to the optimality of a packing.

The general procedure used by our industrial partner is therefore the following: First, a packing without H-boxes is computed on a grid. As seen above, this computation will take no longer than few minutes. Afterwards, the Minkowski Algorithm 3.1 can be used to compute a (mostly better) solution within several hours. The found solution can now be improved manually by inserting H-boxes, restructuring the packing etc. This manual optimisation process might take around one hour. In most cases, the found packing marks a new record for the corresponding trunk.

5 Packing Real Objects into a Trunk

During the car design process not only the luggage capacity according to the SAE and DIN standards is measured. Our industrial partner also uses real geometries (suitcases, bottle crates, golf bags, baby carriages etc.) and packs them into the trunks. These objects are naturally not of cuboid shape. So we are faced with a packing problem using arbitrary geometries.

5.1 Motivation

Given a car trunk and a set of luggage, each modelled with a set of triangles. The model might be incomplete, that means it is possible that a model contains holes or overlapping triangles. Now possible packing questions would be:

- Will all luggage pieces fit into the trunk?
- Given a subset of the luggage pieces, will this set fit into the trunk?
- Find a subset of the luggage pieces with maximum volume that would fit into the trunk!

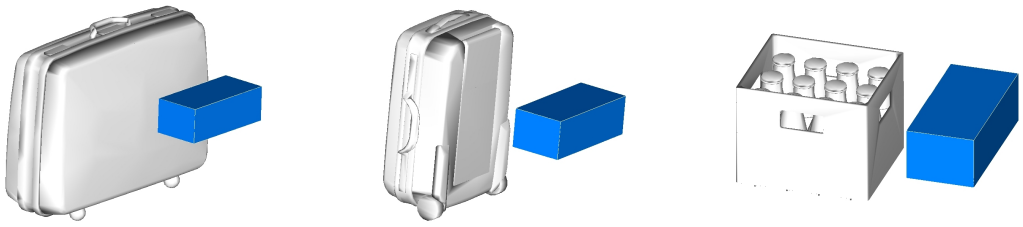
We will concentrate on the second question. That means the user has to select a set of luggage items, and our software will determine whether this set will fit into the trunk.

Some luggage pieces have got additional requirements. For example, bottle crates always have to stand upright. Other requirements include:

- Some luggage pieces may not be stacked on top of each other.
- Some luggage pieces have got a flexible hull and may be squeezed by a certain amount.
- As stated above, bottle crates have to stand upright.

The luggage pieces we are using are usually larger than the boxes of the SAE standard. Some pieces of the luggage set are displayed in Figure 5.1. For comparison, we added a box of type H to each picture.

Now a set of questions arises. At first, the luggage geometries are – as well as the trunk geometry – given as triangular mesh without inside/outside information. There might as well be holes in the geometry and, which is new to our packing problem, triangles in the interior of the luggage items. The first step in a packing algorithm is therefore to create the possibility to check intersections between two objects or between an object and the



(a) Large suitcase (b) Small suitcase (c) Milk crate

Figure 5.1: Some pieces of the luggage set

trunk. Otherwise it might be that a large luggage item is placed completely within another object without any triangle intersection. This can be avoided if the object's bounding boxes are checked first for intersection. This also gives a small speed-up if the objects do not intersect at all because we don't have to compare all triangles of the objects. The second step is of course, the packing algorithm itself.

In this chapter we show that it is possible to use the grid-based Algorithm 2.6 for finding a packing of a subset of luggage pieces. Hence the packing will be axis-oriented as the SAE packings before.

The user may choose arbitrary luggage items and quantify how many items of each type are to be used.

5.2 The Packing Algorithm

First, we discretise the trunk geometry by a uniform grid with spacing s as in chapter 2. This reduces the set of possible placements for a luggage item to a finite number. Now we can check for each luggage item at each position $p = (i \cdot \mathbf{x} + j \cdot \mathbf{y} + k \cdot \mathbf{z}) \cdot s$ defined by a grid cube $c = (i, j, k)$ whether the item fits at this position. We have to perform this check for all possible axis-oriented rotations of the object. Since the luggage items are not symmetrical, we have to use 24 different orientations instead of only six for the box case. For the bottle crates we have the additional requirement that they have to stand upright. That means their local z-axis has to point into the same direction as the global z-axis. This requirement reduces the number of orientations for bottle crates to four. In addition to this we can observe that bottle crates always have a box-like structure. Therefore we can replace the real geometries by the bounding boxes of the bottle crates. There will be only a small disadvantage because of the chamfered edges of the crates.

For each combination (t, o, c) of grid cube c, luggage item t and orientation o we create a conflict graph vertex v if the corresponding item fits into the trunk geometry when its centre is placed at the cube c. Afterwards we generate the conflict graph edges. As before, two vertices u, v are connected by an edge if the corresponding luggage items would intersect. The intersection tests are performed using the real geometries of the luggage items.

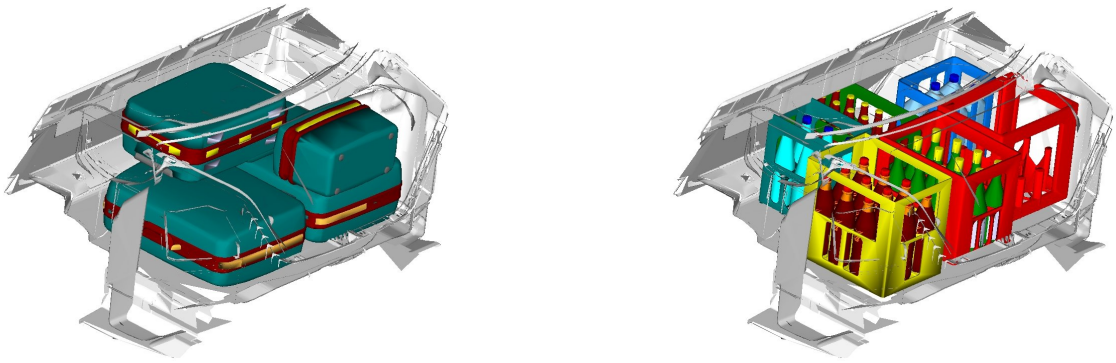


Figure 5.2: Possible solutions for packing real objects into a trunk

Unfortunately the intersection test is rather slow, so we pre-calculate possible intersections for the relative positions of each pair of items. Then we can easily access the intersection information for each relative position.

The graph problem to be solved will essentially be the same as in Definition 2.8. The difference to the SAE J1100 standard is the maximum number of occurrences for each object type. As a simplification we create a vertex type for each object the user has chosen. If the user chose to pack several objects of the same type (for example three crates of milk), we create a type for every occurrence of these objects. So we can set the maximum number of occurrences for each type to 1.

Now the graph reduction process can be done as described in section 2.1.2. Afterwards Algorithm 2.6 is used to find an independent set of maximum weight, with the additional constraint that each vertex type may only occur once in the set.

The resulting packings strongly depend on the discretisation of the trunk and luggage items. Normally we can find a valid packing on a grid with a spacing of 20 mm, if such a packing exists. The conflict graph can be kept rather small, mainly because of the more complex structure of the objects, and because the objects themselves are larger than SAE J1100 boxes.

Figure 5.2 shows two packings of several real objects inside a trunk. As can be seen, the bottle cases are standing upright. If the orientation of the grid is modified, we always use an orientation of the bottle cases such that their z-axis will point in positive global z-direction. This can be easily ensured by examining the global z-coordinate of the object's z-axis.

5.3 Possible Improvements

The quality of the packing depends now on the discretisation of the trunk space. Naturally a large grid spacing is more likely to deny the chosen luggage set whereas with a small spacing we could find a packing. A possible improvement to this restriction could be the use of Minkowski Sums (see chapter 3) to ensure that the objects can always touch. For computing the Minkowski Sums we can use the bounding boxes of the lug-

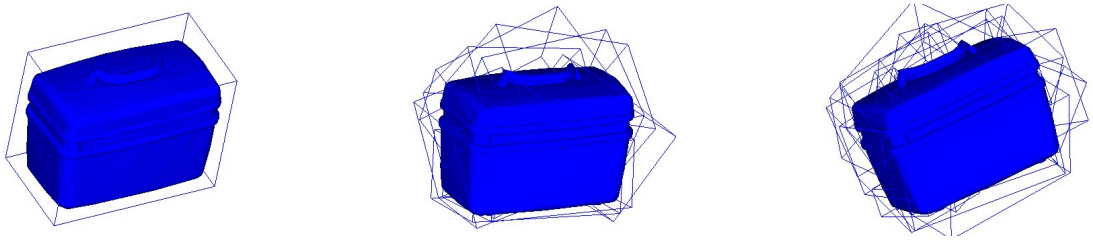


Figure 5.3: Large triangles cause the bounding boxes of the object to have large intersections

gage items. Then the Algorithm 3.1 can be used to find a valid packing: New vertices can be added by using the pre-calculated intersection matrices. For the bottle crates this switch is rather easy because they already have a box-like shape. Only the corners of the crates are chamfered.

Better packing results might also be obtained if the available space is represented by a multi-resolution octree as described in [11]. There a three-dimensional grid structure is used to determine the inside and outside of the trunk and all objects. The resolution of the octree can be adjusted to a desired level of detail. Each octree node gets one of three states, similar to the states of the grid cells: `inside`, `outside` and `boundary`. The first two states are final, that means all child nodes would have the same state as their ancestor node. A `boundary`-cell can have all three types of nodes as children. Intersections between trunk and object can now be detected easily by comparing the octrees of the two items. The objects intersects the trunk geometry, if all `inside` cells of the object octree are also at `inside` cells of the trunk octree. The main open problem remains to implement an efficient octree structure (see section 3.3.2 for similar considerations).

There have also been experiments with a contact simulation similar to the method described in chapter 6. In this case the trunk geometry can be represented as a point cloud (see chapter 3) and the object-trunk interactions can be computed easily as point containment task. Other possible improvements can be made with more efficient intersection tests. These could be achieved by using bounding box hierarchies for the suitcases. Unfortunately the triangles of the suitcase geometries are likely to be very long and thin, which causes the bounding boxes to have large intersections. Figure 5.3 illustrates this effect for the beauty case with three different bounding box levels.

For using bottle crates we also experiment with a shift-based packing (see [44]) combined with a contact simulation [58]. This approach seems very promising.

6 Conclusion and Prospectives

6.1 Algorithm Survey

We have presented several algorithms to compute valid packings according to the US standard SAE J1100. First, we discretised the volume of the luggage compartment into a grid of fixed spacing, according to [48]. We also described techniques to improve the discretisation in order to find larger packings. Then the grid was translated into a so-called conflict graph. On this graph the combinatorial packing problem (see Definition 2.8) is defined.

Using a property of (weighted) independent sets, we could reduce the size of the conflict graph significantly without influence on the solution size (see Theorem 2.16 and Corollary 2.17).

We showed that an approximation algorithm like the greedy algorithm does not suffice to produce an acceptable solution. Therefore we concentrate on exact algorithms, especially for the Maximum Weighted Independent Set problem. For this problem and its complement, the Maximum Weighted Clique problem, several algorithms are given in literature. Experiments showed that the fastest of these algorithms, presented in [46], still does not fulfil the runtime restrictions given by our industrial partner.

In consequence we designed an enumeration scheme to solve the combinatorial packing problem exactly. This scheme is presented as Algorithm 2.6. The most important step is to choose good vertices for a fast execution of the branch-and-bound scheme. For this step we examinated four different orderings. At first, we did not take a specific order of the vertices but used their natural order when they were created in the conflict graph. The second order is a randomised order which is fixed in the beginning of the run and not changed afterwards. The third order was suggested by the greedy algorithm: in each step the weighted vertex degree [34] is computed and the vertices are chosen according to this weighted degree (smallest first). The last vertex order reflects a human packing order: The vertices are first sorted by their weight, with the weighted degree used as tie-breaking.

Since all of these algorithms produce exact solutions to the combinatorial packing problem, we mainly compared the runtimes of these algorithms. Clearly the fastest algorithm used the human packing order. The quality of the discretisation could be measured by comparing the algorithmic solution to a manually packed solution. In most cases our algorithm could find better solutions than the experienced engineer.

However, the discretisation of the trunk volume has got its drawbacks. First, a discretisation can never reflect the complex geometry of a trunk exactly. Second, the standard SAE J1100 uses boxes of different sizes, and the greatest common divisor of these sizes is $0.1'' \approx 2.54$ mm. For comparison, the greatest common divisor within the

German standard DIN 70020 is 50 mm. Due to the runtime requirements a grid with spacing of at least $1''$ is practical. Therefore we had to modify the side lengths with two diametral consequences: Either the boxes are treated as larger boxes, and we get gaps between boxes resulting in a non-optimal packing. Or the boxes are treated as smaller boxes. Then the solution contains overlapping boxes. So a cleaning process is required, such as a contact simulation of boxes and trunk geometry. In most cases this simulation successfully generates a valid packing from our grid-based solution.

These problems lead to further schemes to create axis-oriented packings. In chapter 3, algorithms using the Minkowski Sum are introduced. One algorithm, presented in [1], uses linear programming and a simplified version of the Minkowski Sum. This algorithm still has got the disadvantage of too much user input. In consequence we defined the continuous conflict graph, derived from the Minkowski Sum. Using again Theorem 2.16 and [51], we can find discrete vertices of the Minkowski Sum, namely the corners of this structure.

By extending Algorithm 2.6 to the continuous case and a potentially infinite, dynamic conflict graph, we are able to find valid packings that are of even better quality than the grid-based solutions.

The only difficulty now lies in the computation of the Minkowski Sum. Up to now, the exact computation of the surface of a Minkowski Sum is too expensive for our application, since we have to compute the Minkowski Sum for each box type and each orientation. Therefore we discretise again the trunk geometry. Now we sample the surface of the trunk into a dense point set and compute an approximative Minkowski Sum as the union of boxes. This procedure is done within half a minute for all 42 Minkowski Sums, compared to 40 minutes for only one exact Minkowski Sum.

A second aspect to the trunk packing problem is to pack real objects into the trunk. In our application, the object geometries are given as a triangular mesh. We showed that it is possible to discretise the trunk in the same way as in chapter 2 with a grid, and place the objects only on the grid points. So it is possible to use the same algorithms as for packing boxes. For packing real objects, we have a different goal: Given a trunk and a set of real objects, do all of the objects fit into the trunk? This question does not include the optimisation of an independent set, but we only have to find an independent set of a fixed size.

A few difficulties arise because the real objects have a more complicated structure than boxes: The computation of intersections between objects takes a long time, so we pre-calculate intersection intervals for several relative positions of the objects and can therefore easily access this information. Another aspect is the asymmetry of the objects. This leads to 24 different orientations for each box type.

Since the conflict graph can be kept very small, we can solve many instances directly on grids with small spacings (less than 25 mm).

6.2 Possible Improvements

Nevertheless, all of the presented algorithms are subject to improvements. For the solution of the MWIS problem, there can always arise new, more efficient algorithms. These algorithms might be adapted to our special trunk packing case in order to find solutions for large grids with small spacing.

6.2.1 Using Arbitrary Orientations for the Minkowski Sums

Algorithm 3.1 uses Minkowski Sums. However, the exact calculation of Minkowski Sums is difficult and takes too much time for our application. Therefore we use a discretisation of the trunk by a dense point cloud and compute the Minkowski Sum of the point cloud and a box as union of boxes. This discretisation, however, usually increases the number of corners and with it the start number of vertices of the conflict graph. With a real Minkowski Sum we would reduce the start number of vertices *and* get a more exact solution for axis-oriented packing. The interface of our algorithm is designed such that an arbitrary data structure can be used as geometrical representation of the packing process. This data structure just has to return sets of vertices if a box is added or removed.

With this background it might also be possible to include Minkowski Sums with arbitrary rotation of the boxes. Up to now, the development into this direction has only started, and first Minkowski Sums of 2-dimensional rotatable objects are possible, which as well generates a Minkowski Sum in 3-dimensional configuration space. If the objects become 3-dimensional, the Minkowski Sum will be in 6-dimensional configuration space.

6.2.2 Contact Simulations

As described in chapter 2, the grid-based algorithms suffer from the inaccuracy of the represented box sizes. Currently we solve this problem by a postprocessing step of a contact simulation of boxes and trunk geometry. This simulation shall be described here.

We have developed an efficient and easy-to-implement simulation of contacts between rigid objects. The main idea came during a programming competition where the task was to pack N circles of different radii (1 to N) without overlapping into a larger circle C. The radius of this circle had to be as small as possible [61]. We decided to put our focus to the compression of an existing packing, that means the reduction of the radius of C. Our simulation had to ensure two components. First, the circles had to be non-overlapping. This can easily be ensured by measuring the distance of the circle centres. Second, we had to calculate the movement of the circles if they touch each other. This can be done using impulse propagation. We assume the circles to have equal mass such that even large circles are easy to be moved. Finally, the contact forces can be represented by vectors pointing along the axes connecting the circle centres. So a single moving circle at one side of a packing influences the whole packing. To perform a reduction of the container circle C, we added an external force to the outmost circles.

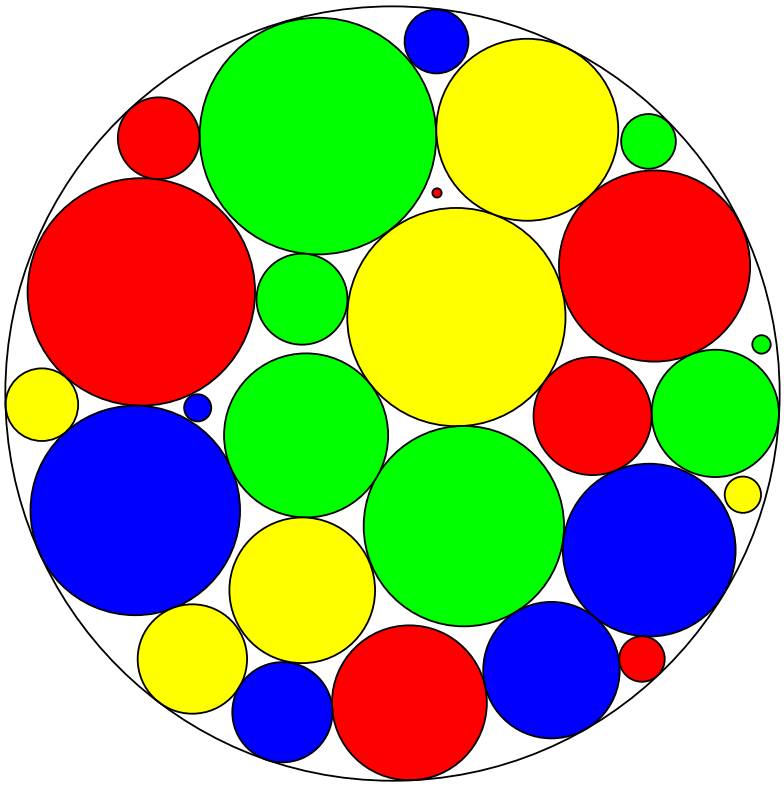


Figure 6.1: The record-holding packing with 26 circles

As an outer framework we used several heuristics to generate different packing patterns (essentially different arrangements of circles). Using these techniques we were able to do quite well within this international contest. Although our hardware ressources were limited, we finally ranked on 8th position of over 150 participants. The contest was divided into 46 instances for $N = 5..50$ circles. We managed to find the best packing for $N = 26$ circles, which is shown in Figure 6.1.

Afterwards we could improve the large instances by using a combination of the contact simulation and simulated annealing methods [43].

Although the contact simulation for circles is quite easy, it is a huge step to the simulation of boxes. Whereas with circles one does only have to consider translational movements, we get in addition the rotation of boxes to our calculations. Therefore we have to define a torque in order to get a physically correct simulation. The simulation is now based on the justification of a packing, namely eliminating all intersections between objects.

Figure 6.2 shows the interaction of two intersecting objects A and B. We have to calculate several parameters for a correct identification of the resulting translational ($\mathbf{v}_a$ and $\mathbf{v}_b$) and rotational (ω_a and ω_b) movements. We have to compute the centre of the intersection of the two objects ($\mathbf{z}$) to get the levers ($\mathbf{r}_a$ and $\mathbf{r}_b$ of the force. The resulting

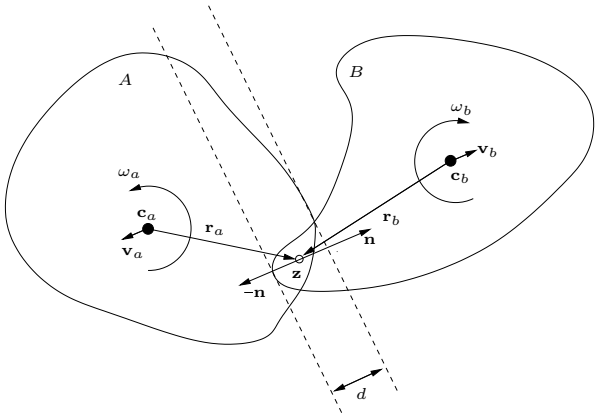


Figure 6.2: The interaction of two objects A, B and their resulting movements

translation of the objects lies parallel to the contact normal $\mathbf{n}$.

We use efficient intersection test routines according to the separating axes theorem [20], alongside with a priority queue that stores the boxes according to their distance. This enables us to perform a real-time collision detection for simple geometries with hundreds of iterations per second. For the packing algorithms, we are able to make grid-based packings intersection-free, usually within less than a minute.

This simulation can also be modified for a contact simulation of real objects as it might be needed for the packing of arbitrary objects as in chapter 5.

6.2.3 Simulated Annealing

In [21, 35, 49], another algorithm for continuous packing of boxes has been presented for the DIN 70020 packing problem. This approach uses Simulated Annealing techniques.

In the Simulated Annealing approach, an objective function f is evaluated for an arbitrary configuration of boxes. The boxes may intersect each other and the trunk geometry. The function f has got several parameters, such as the size of the packing, and penalties for intersections. Additionally a corrective term is given by the "temperature" of the whole system. A high temperature allows more intersections between boxes, whereas a low temperature increases the corresponding penalties. The whole system is cooled down during the simulation in order to get a valid configuration. However, it is possible to heat the system up to avoid getting stuck in local optima.

During the run of the algorithm several moves are allowed:

- Move a box (`TRANSLATE`)
- Rotate a box (`ROTATE`)
- Add a box (`ADD`)
- Delete a box (`DELETE`)

These moves are accepted or discarded according to the evaluation of the objective function. However, the main problem in this approach is that there are many parameters to be considered, and the whole algorithm might tend to be overtuned. Currently the four main moves are performed at random with probabilities depending on the temperature of the system. If the temperature is rather high, more add operations can be allowed, whereas with low temperature it is more likely that boxes are deleted.

The next problem arises due to the randomness of the algorithm: All `TRANSLATE` and `ROTATE` moves have random parameters. So it is likely that many moves are discarded just because intersecting boxes are moved towards each other instead of solving the conflicts directly using the contact simulation of section 6.2.2. Currently we are working on a synthesis of these two approaches.

Finally it is not yet clear where boxes have to be added during the `ADD` move. In the real world one would try to find a gap where a box could be placed. However, this approach is not easy to implement within this algorithm. Instead, the boxes are placed at random within the container, regardless of other boxes. Only afterwards the objective function is evaluated and the move may be discarded. The Minkowski Sums of chapter 3 provide us with a tool to find free spaces for boxes of given measures. We are as well currently experimenting with a synthesis of the Simulated Annealing approach with Minkowski Sums. The third ingredient to this algorithm is a shift-based approach presented in [44]. With these three parts we are able to get competitive packings for the DIN 70020 packing problem [58].

Whereas the previous considerations apply to all packing tasks, we get another problem when regarding the SAE J1100 packing problem. There we have boxes of different sizes, and hence an additional possible move:

- Change the type of a box (`EDIT`)

With this move we can increase or shrink a box in order to get either a packing with more covered volume or a packing with less intersections. However, this algorithm only exists in theory and is still to be implemented and evaluated. Up to now we are just able to perform these moves manually.

As well we are experimenting with a randomised algorithm using the ideas of contact simulation and some of the simulated annealing approach [57]. First results of this algorithm seem rather promising. There it would also be possible to reproduce the requirements of the SAE J1100 standard exactly.

6.2.4 Manual Optimisation of Packings

Using the contact simulation of section 6.2.2, we can also improve existing packings. For this purpose, we can insert new boxes, enlarge or shrink existing boxes and apply the contact simulation to create an intersection-free packing. Our software allows these modifications and checks constantly the validity of the packing, namely the number of boxes according to the standard SAE J1100 and whether there are no box/box- or box/trunk-intersections. This method is applied in the optimisation process used by our industrial partner (see section 4.4).

6.3 Open Problems

Further investigations could focus on the possibility whether the trunk space can be discretised by a tetrahedral grid, which would allow more box orientations. The orientations of the tetrahedra, however, will not suffice to find better packings because the SAE boxes always have cuboid shape and therefore a rectangular grid seems to be better. Nevertheless it might be interesting to know whether the restriction to box placements can be fulfilled while the box orientations can be chosen arbitrarily. This would also result in another type of continuous conflict graph, using rotation angles as edges.

The Minkowski Sums provide a powerful tool for packing algorithms with axis-oriented objects. Still it is an open problem to compute Minkowski Sums of freely rotated objects, that is, including the variable orientation of the boxes (in our case). This results in a high-dimensional search space, for example packing problems in the plane can be transformed into problems in three-dimensional space when objects can be rotated. In three spatial dimensions we get far more degrees of freedom because the orientation of an object has to be given in three additional variables. If the Minkowski Sum can be computed for all orientations as well, Algorithm 3.1 would deliver an exact algorithm for the three-dimensional packing problem.

Nevertheless, we still have to find an efficient algorithm computing the exact Minkowski Sum for fixed orientations. Already this algorithm will provide a huge improvement over our actual computation using a point cloud discretisation: The exact Minkowski Sum will return less points than the point cloud Minkowski Sum. This leads to a smaller conflict graph and to a more efficient computation in Algorithm 3.1.

Bibliography

[1] Ernst Althaus, Tobias Baumann, Elmar Schömer, and Kai Werth. Trunk Packing Revisited. In Camil Demetrescu, editor, *Experimental Algorithms: Proceedings of the 6th International Workshop, WEA 2007*, volume 4525 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2007.

[2] Francis Avnaim. *Placement et déplacement de formes rigides ou articulées.* Thèse de doctorat en sciences, Uni-ver-sité de Franche-Comté, Besançon, France, 1989.

[3] Francis Avnaim and Jean-Daniel Boissonat. The polygon containment problem: Simultaneous containment under translation. Technical report, 1987.

[4] Francis Avnaim and Jean-Daniel Boissonat. Simultaneous containment of several polygons. pages 242–247, 1987.

[5] L. Babel. A fast algorithm for the maximum weight clique problem. *Computing*, 52(1):31–38, 1994.

[6] Egon Balas and Jue Xue. Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs. *SIAM J. Comput.*, 20(2):209–221, 1991.

[7] Egon Balas and Jue Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15(5):397–412, 1996.

[8] Tobias Baumann. Effiziente Färbungsalgorithmen für k-färbbare Graphen. Diploma thesis, Technische Universität Chemnitz, 2004.

[9] Tobias Baumann, Magnus Jans, Elmar Schömer, Christian Schweikert, and Nicola Wolpert. Dynamic Free-Space Detection for Packing Algorithms. 24th European Workshop on Computational Geometry (EuroCG'08), March 2008.

[10] Matthias Baumgart. Effiziente Approximation unabhängiger Mengen in Graphen. Diploma thesis, Technische Universität Chemnitz, 2004.

[11] Jonathan Cagan and Quan Ding. Automated trunk packing with extended pattern search. *Virtual Engineering, Simulation & Optimization*, SP-1779:33–41, 2003.

[12] Jonathan Cagan, K. Shimada, and Sun Yin. A survey of computational approaches to three-dimensional layout problems. *Computer-Aided Design*, 34(8):597–611, 2002.

[13] Maw-Shang Chang and Fu-Hsing Wang. Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs. *Inf. Process. Lett.*, 43(6):293–295, 1992.

[14] G. H. Chen, M. T. Kuo, and J. P. Sheu. An optimal time algorithm for finding a maximum weight independent set in a tree. *BIT Numerical Mathematics*, 28(2):353–356, june 1988.

[15] Karen Daniels. *Containment Algorithms for Nonconvex Polygons with Applications to Layout*. PhD thesis, Cambridge, MA, 1995.

[16] Karen Daniels, Zhenyu Li, and Victor Milenkovic. Multiple containment methods. Technical report, 1994.

[17] Karen Daniels and Victor Milenkovic. Multiple translational containment: Approximate and exact algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 205–214, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

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

[19] Harald Dyckhoff, Guntram Scheithauer, and Johannes Terno. Cutting and packing. pages 393–412, 1997.

[20] David Eberly. *3D Game Engine Design: A Practical Approach to Real Time Computer Graphics*. Morgan Kaufmann, 2001.

[21] Friedrich Eisenbrand, Stefan Funke, Andreas Karrenbauer, Joachim Reichel, and Elmar Schömer. Packing a trunk: now with a twist! In *SPM '05: Proceedings of the 2005 ACM symposium on Solid and physical modeling*, pages 197–206, New York, NY, USA, 2005. ACM Press.

[22] Friedrich Eisenbrand, Stefan Funke, Joachim Reichel, and Elmar Schömer. Packing a trunk. In Giuseppe Di Battista and Uri Zwick, editors, *Algorithms - ESA 2003: 11th Annual European Symposium*, volume 2832 of *Lecture Notes in Computer Science*, pages 618–629, Budapest, Hungary, September 2003. Springer.

[23] P. Erdos and R. L. Graham. On packing squares with equal squares. *Journal of Combinatorial Theory*, 19:119–123, 1975.

[24] S. P. Fekete and J. Schepers. On more-dimensional packing i: Modeling. Technical report, 2000.

[25] Robert J. Fowler, Mike Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are np-complete. *Information Processing Letters*, 12(3):133–137, 1981.

[26] Erich Friedman. Erich's Packing Center. personal homepage, `http://www.stetson.edu/~efriedma/packing.html`, June 2008.

[27] Deutsches Institut für Normung e.V. DIN 70020, Teil 1, Straßenfahrzeuge; Kraftfahrzeugbau; Begriffe von Abmessungen., February 1993.

[28] David Gamarnik, Tomasz Nowicki, and Grzegorz Swirszcz. Maximum weight independent sets and matchings in sparse random graphs. exact results using the local weak convergence method. In *APPROX-RANDOM '04*, pages 357–368, 2004.

[29] Michael R. Garey and David S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., San Francisco, 1979.

[30] Fanica Gavril. Maximum weight independent sets and cliques in intersection graphs of filaments. *Inf. Process. Lett.*, 73(5-6):181–188, 2000.

[31] Peter Hachenberger. Exact minkowksi sums of polyhedra and exact and efficient decomposition of polyhedra in convex pieces. In *ESA*, pages 669–680, 2007.

[32] ILOG. CPLEX. `http://www.ilog.com/products/cplex/`.

[33] D. S. Johnson and M. Trick, editors. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, Providence, RI, USA, 1996.

[34] Akihisa Kako, Takao Ono, Tomio Hirata, and Magnús M. Halldórsson. Approximation algorithms for the weighted independent set problem. In *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG*, pages 341–350, 2005.

[35] Andreas Karrenbauer. Packing boxes with arbitrary rotations. Diploma thesis, Universität des Saarlandes, Saarbrücken, 2004.

[36] Victor Klee. Can the measure of $\cup[a_i, b_i]$ be computed in less than $o(n \log n)$ steps? *American Mathematical Monthly*, 84:284–285, 1977.

[37] Vadim V. Lozin and Martin Milanič;. A polynomial algorithm to find an independent set of maximum weight in a fork-free graph. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete Algorithms*, pages 26–30, New York, NY, USA, 2006. ACM Press.

[38] M. Milanič and J. Monnot. On the complexity of the exact weighted independent set problem for various graph classes. Technical Report 17, DIMACS, 2006.

[39] Victor Milenkovic. Translational polygon containment and minimal enclosure using linear programming based restriction. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 109–118, New York, NY, USA, 1996. ACM Press.

[40] Victor Milenkovic and Karen Daniels. Translational polygon containment and minimal enclosure using geometric algorithms and mathematical programming. Technical report, Cambridge, MA, 1995.

[41] Victor Milenkovic and Karen Daniels. Translational polygon containment and minimal enclosure using mathematical programming. *International Transactions in Operational Research*, pages 6:525–554, 1999.

[42] H. Minkowski. Volumen und Oberfläche. *Mathematische Annalen*, 57:447–495, 1903.

[43] André Müller. Kreispackungen mit Hilfe von Simulated Annealing (working title). Diploma thesis, Johannes Gutenberg-Universität Mainz, Mainz, 2008.

[44] Ulla Neumann. Optimierungsverfahren zur normgerechten Volumenbestimmung von Kofferräumen im europäischen Automobilbau. Diploma thesis, Technische Universität Braunschweig, Braunschweig, 2006.

[45] Society of Automotive Engineers. SAE J1100, Motor Vehicle Dimensions, February 2001.

[46] Patric R. J. Östergård. A new algorithm for the maximum-weight clique problem. *Nordic J. of Computing*, 8(4):424–436, 2001.

[47] Mark H. Overmars and Chee-Keng Yap. New upper bounds in klee's measure problem (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 550–556, 1988.

[48] Joachim Reichel. *Combinatorial approaches for the Trunk packing problem.* PhD thesis, Saarbrücken, 2006.

[49] Jens Rieskamp. Automation and Optimization of Monte Carlo Based Trunk Packing. Diploma thesis, Universität des Saarlandes, Saarbrücken, 2005.

[50] Shuichi Sakai, Mitsunori Togasaki, and Koichi Yamazaki. A note on greedy algorithms for the maximum weighted independent set problem. *Discrete Appl. Math.*, 126(2-3):313–322, 2003.

[51] Jörg Schepers. *Exakte Algorithmen für orthogonale Packungsprobleme.* PhD thesis, Köln, 1997.

[52] Christian Schweikert. Boolesche Operationen auf Quadermengen und Dreiecksnetzen. Diploma thesis, Johannes Gutenberg-Universität Mainz, Mainz, 2007.

[53] D. Shah. Max product for max-weight independent set and matching. *ArXiv Computer Science e-prints*, August 2005.

[54] Zoltán Szigeti. On the graphic matroid parity problem. *J. Comb. Theory, Ser. B*, 88(2):247–260, 2003.

[55] Jeffrey S. Warren and Illya V. Hicks. Combinatorial branch-and-bound for the maximum weight independent set problem, 2006.

[56] Deepak Warrier, Wilbert E. Wilhelm, Jeffrey S. Warren, and Illya V. Hicks. A branch-and-price approach for the maximum weight independent set problem. *Netw.*, 46(4):198–209, 2005.

[57] Kai Werth. *A Streamlined Physics Engine for Packing Issues (working title)*. PhD thesis, Johannes Gutenberg-Universität Mainz.

[58] Dominik Will. Erzeugung dicht gepackter Anordnungen von Quadern in nichtkonvexen polyedrischen Containern (working title). Diploma thesis, Johannes Gutenberg-Universität Mainz, Mainz, 2008.

[59] Michael Wottawa. *Struktur und algorithmische Behandlung von praxisorientierten dreidimensionalen Packungsproblemen*. PhD thesis, Köln, 1996.

[60] Roland Wunderling. SOPLEX. `http://web.bilkent.edu.tr/Online/Soplex/`.

[61] Al Zimmermann. Al Zimmerman's Circle Packing Contest. Al Zimmerman's Programming Contests, `http://www.recmath.org/contest/CirclePacking/`.