Combinatorial Approaches for the Trunk Packing Problem

Dissertation

zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von

Joachim Reichel

Saarbrücken
2006

Tag des Kolloquiums: 10. Juli 2006

Dekan der Naturwissenschaftlich-Technischen Fakultät I:
Prof. Dr.-Ing. Thorsten Herfet

Vorsitzender des Prüfungsausschusses:
Prof. Dr.-Ing. Gerhard Weikum

Gutachter:
Prof. Dr. Elmar Schömer
Prof. Dr. Kurt Mehlhorn

Promovierter akademischer Mitarbeiter:
Dr. Stefan Funke

Abstract

In this thesis we consider a three-dimensional packing problem arising in industry. The task is to pack a maximum number of rigid boxes with side length ratios of $4 : 2 : 1$ into an irregularly shaped container. Motivated by the structure of manually constructed packings so far, we pursue a discrete approach. We discretize the shape of the container as well as the set of possible box placements. This discrete packing problem can be reduced to a maximum stable set problem.

First we formulate the problem as an integer linear program, which admittedly can only be solved to optimality within reasonable runtime for very small instances. Therefore, we present several heuristics based, for example, on the linear programming relaxation or on local search. Other heuristics generate tight packings for the core of the container, thereby reducing the problem to a set of smaller subproblems.

We compare all presented algorithms on real data sets. We achieve very good results for the majority of instances and for some instances we even surpass the manually constructed solutions.

Zusammenfassung

In dieser Arbeit behandeln wir ein dreidimensionales Packungsproblem aus der Industrie. Die Aufgabe besteht darin, möglichst viele starre Quader mit einem Seitenverhältnis von $4 : 2 : 1$ in einen unregelmäßig geformten Container zu packen. Motiviert durch die Struktur der bisher manuell erstellten Packungen verfolgen wir einen diskreten Lösungsansatz. Dazu diskretisieren wir sowohl die Form des Containers als auch die Platzierungsmöglichkeiten der Quader. Dieses diskrete Packungsproblem lässt sich auf die Berechnung einer größtmöglichen unabhängigen Knotenmenge reduzieren.

Wir formulieren das Problem zunächst als ganzzahliges lineares Programm, das allerdings nur für sehr kleine Instanzen mit angemessenem Rechenaufwand beweisbar optimal gelöst werden kann. Daher stellen wir verschiedene Heuristiken vor, die zum Beispiel auf einer Relaxierung des ganzzahligen linearen Programms oder lokaler Suche basieren. Andere Heuristiken generieren zunächst dichte Packungen für den Kern des Containers und reduzieren so das Problem auf eine Reihe kleinerer Teilprobleme.

Wir vergleichen alle vorgestellten Algorithmen an Hand realer Datensätze. In der Mehrzahl der Fälle erreichen wir sehr gute Resultate, bei einigen Instanzen übertreffen wir sogar die manuell erstellten Packungen.

Ausführliche Zusammenfassung

In der vorliegenden Arbeit behandeln wir ein dreidimensionales Packungsproblem aus der Automobilindustrie. Auf dem europäischen Markt sind Automobilhersteller dazu verpflichtet, das Gepäckraumvolumen eines PKWs entsprechend den Regelungen in der Norm DIN 70020 zu bestimmen und zu veröffentlichen. Diese Norm schreibt vor, den Kofferraum mit starren Quadern der Größe 200mm $\times$ 100mm $\times$ 50mm zu packen. Das Gepäckraumvolumen des Kofferraums wird dann als das durch die Quader überdeckte Volumen definiert. Das so ermittelte Gepäckraumvolumen ist ein wichtiges Verkaufsargument; dementsprechend wird von den Automobilherstellern viel Aufwand betrieben, um einen möglichst hohen Wert zu erreichen.

Das Ziel des dieser Arbeit zu Grunde liegenden Projektes mit einem internationalen Automobilhersteller ist es, ein Softwarepaket zu entwickeln, um das Gepäckraumvolumen eines PKWs in Übereinstimmung mit der Norm DIN 70020 zu bestimmen. Die Anwendung soll abgesehen von einer Vorbereitungphase keine Benutzerinteraktion erfordern und muss ohne Expertenwissen zu bedienen sein. Die Güte der berechneten Lösungen soll von manuell durch Experten konstruierten Packungen um nicht mehr als ein Prozent bzw. zehn Litern nach unten abweichen.

Die unregelmäßige Form des Kofferraums ist ein wesentlicher Unterschied zu der Mehrzahl der in der Literatur betrachteten Packungsprobleme. Motiviert durch die bisher manuell erstellten Packungen verfolgen wir in dieser Arbeit einen diskreten Lösungsansatz. Wir zeigen, dass bereits eine diskrete Variante des eigentlichen Packungsproblems *NP*-vollständig ist. Für diese diskrete Variante existiert ein Approximationsschema mit polynomieller Zeitkomplexität, das für die in der Praxis auftretenden Problemgrößen allerdings nicht geeignet ist.

Die Diskretisierung erfolgt in zwei Schritten: Zunächst approximieren wir das Innere des Kofferraums durch ein dreidimensionales, kubisches Gitter. Diese Approximation erfordert besondere Sorgfalt, da die geometrische Beschreibung des Kofferraums verschiedene Arten von Mängeln aufweist. Weiterhin schränken wir in der diskreten Problemstellung die möglichen Positionen und Orientierungen der Quader ein, so dass alle Quader an den Zellen des Gitters ausgerichtet sind. Wir verwenden effiziente Implementierungen der notwendigen geometrischen Prädikate und beschreiben Algorithmen, um die Informationen einer vorhandenen Approximation bei einer Veränderung ihrer Parameter zu aktualisieren. Diese beiden Komponenten verwenden wir, um eine möglichst gut geeignete Approximation des Kofferraums zu berechnen.

Das diskrete Packungsproblem lässt sich zurückführen auf die Berechnung einer möglichst großen unabhängigen Knotenmenge (stable set) in dem zugehörigen Konfliktgraphen. Wir formulieren das Problem zunächst als ganzzahliges lineares Programm und erhalten damit einen Algorithmus,

der das Packungsproblem optimal lösen kann. In der Praxis stellt sich jedoch heraus, dass selbst mit marktführenden Softwarebibliotheken für ganzzahlige lineare Programme nur kleine Probleminstanzen mit angemessenem Rechenaufwand beweisbar optimal gelöst werden können. Dennoch ist ein solcher exakter Algorithmus nützlich für kleinere Teilprobleme.

Aus diesem Grund präsentieren wir verschiedene Heuristiken. Diese lassen sich in zwei Kategorien unterteilen: direkte Ansätze, die das Packungsproblem in seiner Vollständigkeit lösen können, und indirekte Ansätze, die das Packungsproblem auf eine Menge kleinerer Teilprobleme reduzieren.

Wir stellen zunächst eine einfache Greedy-Heuristik vor, deren Ergebnisse allerdings hinter den Erwartungen zurückbleiben. Weiterhin präsentieren wir einen Algorithmus, der iterativ die Informationen des wesentlich einfacher zu lösenden kontinuierlichen linearen Programms nutzt um die Problemgröße zu reduzieren. Sobald die Problemgröße hinreichend reduziert wurde, wird das verbleibende Teilproblem durch ein ganzzahliges lineares Programm optimal gelöst. Schließlich stellen wir einen auf lokaler Suche basierenden Algorithmus vor. Dieser Algorithmus beinhaltet einen Rückkopplungsmechanismus zur Vermeidung von Zyklen. Regelmäßige Neustarts helfen dabei eine breite Abdeckung des Suchraums zu erreichen. Die Liste der direkten Ansätze wird durch eine einfache Heuristik ergänzt, die sich an der geometrischen Form des Kofferraums orientiert.

Als indirekte Ansätze stellen wir zwei einfache Heuristiken vor, die zunächst das Innere des Kofferraums mit dichten, regelmäßigen Packungen füllen. Dadurch entstehen Teilprobleme unterschiedlicher Zahl und Größe, die wir durch einen der zuvor vorgestellten direkten Ansätze lösen. Ein dritter indirekter Ansatz unterteilt den Kofferraum in mehrere Regionen, die sequentiell gepackt werden. Um den durch die Unterteilung entstehenden Verschnitt zu reduzieren, ist es dabei notwendig, die Regionen nicht unabhängig voneinander zu behandeln und die Platzierung der Quader innerhalb der Regionen gezielt zu beeinflussen.

Wir untersuchen an Hand realer Datensätze aus der Praxis zunächst verschiedene Varianten und Implementierungsdetails der vorgestellten Algorithmen. Die so gewonnenen Erkenntnisse verwenden wir, um die Algorithmen auf die typischen Instanzen abzustimmen. Wir zeigen an einem praktischen Beispiel, dass es oft vorteilhaft ist, bestimmte kleinere Bereiche des Kofferraums zunächst getrennt zu behandeln (inklusive eigener Approximation). Schließlich vergleichen wir verschiedene Kombinationen der vorgestellten Algorithmen hinsichtlich der Qualität und Struktur der Lösungen als auch der dazu benötigten Rechenzeit. In der Mehrzahl der Fälle erreichen wir die geforderte Lösungsgüte, in manchen Fällen übertreffen wir sogar die manuell von Experten zur Verfügung gestellten Packungen.

Alle in dieser Arbeit beschriebenen Algorithmen wurden in einem einfach zu bedienenden Softwarepaket integriert. Dieses Softwarepaket ist in der Entwicklungsphase neuer Fahrzeuge bei unserem Kooperationspartner im Einsatz.

Acknowledgments

First of all, I thank my advisor Prof. Dr. Elmar Schömer for his support and guidance. He rose my interest in the field of packing problems. The discussions with him were always a source of inspiration and motivation. I am thankful to Prof. Dr. Kurt Mehlhorn for the opportunity to work in his excellent research group at the Max-Planck-Institut für Informatik in a very pleasent atmosphere. I want to express my thanks to all colleagues involved in the trunk packing project for a productive and uncomplicated collaboration: Friedrich Eisenbrand, Stefan Funke, Andreas Karrenbauer, Jens Rieskamp and Kai Werth.

Furthermore, I want to thank all my friends and colleagues at the MPI. In particular, I would like to mention my room mates Peter Hachenberger and Joachim Ziegler, as well as the 11:30am lunch group. I will always keep my time at the MPI in good remembrance.

Finally, I thank my parents and my sister, who supported me all the time.

Contents

List of Figures

List of Tables

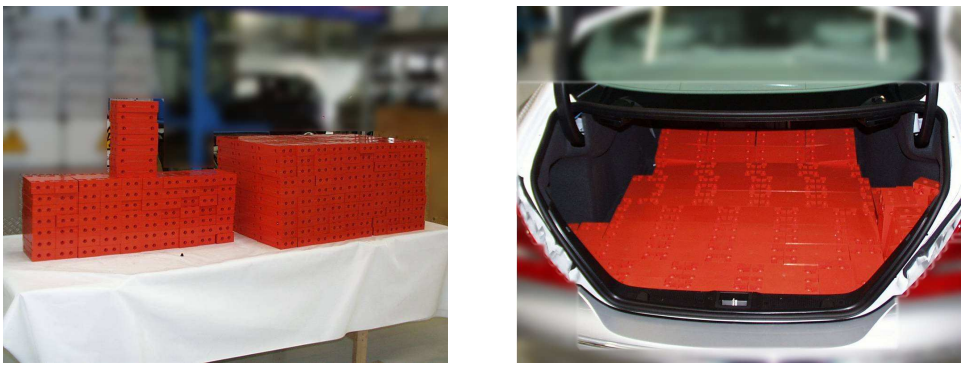
List of Algorithms

Chapter 1

Introduction

In this thesis we present a combinatorial approach to a real-world packing problem which arises in car industry. The task is to pack a maximum number of uniform rigid boxes into the interior of a car trunk. This problem is of practical importance due to a German standard which requires car manufactures to state the trunk volume according to this measure.

The details are set forth in the German standard DIN 70020 [Deu93], which is also used in several other European countries. This standard defines the size of such a box as $200\text{mm} \times 100\text{mm} \times 50\text{mm}$, i.e., each box has a volume of exactly one liter. A set of such boxes and a partial packing of a car trunk can be seen in Figure 1.1.



(a) boxes (b) partial packing

Figure 1.1: Measurement of the trunk volume according to DIN 70020

The standard defines the volume of the trunk (to be more precise, its *luggage capacity*) as the volume that is covered by the boxes packed into the trunk. The volume obtained by this discrete measurement is significantly smaller than the continuous volume of the trunk. The motivation for this standard is the fact that the luggage usually to be stored is also discrete

and the continuous volume of the trunk does not reflect its effective storage capacity. The American standard SAE J1100 actually uses a standardized luggage set to define the luggage capacity of the trunk (see Section 1.2).

European car manufactures are required to publish the trunk volume according to DIN 70020. This discrete volume is an important sales argument, and therefore much time is spent on achieving a high volume.

Up to now, this problem has been manually solved with a lot of effort, either with a real-world model (*physical mockup*) or virtually within a CAD system (*digital mockup*). In both cases, experienced engineers spend one or two days to find a satisfying solution for this packing problem. Large models, e.g., minivans, require even up to a week of manpower.

A measurement of the trunk volume is not performed only once, but multiple times during the design phase of a new car. Any changes of the geometry need to be evaluated with respect to their impact on the trunk volume. Therefore, an automated solution is highly desirable.

The goal of our project in cooperation with a large international car manufacturer is to develop an industrial-strength system that allows to compute the trunk volume according to DIN 70020. The key requirements of the system are:

- **Validity** The boxes must not intersect each other. Additionally, the boxes must not pierce the boundary of the trunk by more than a predefined threshold, which models the deformability of the trunk.
- **Quality** The number of boxes should be as high as possible. It should never fall short of the solution of an expert by more than 1% or 10 liters.
- **Usability** Ease-of-use is important; expert knowledge should not be necessary for operation. With the exception of an initial preprocessing phase, the system must work non-interactively.
- **Runtime** The solution for a problem instance should be computed within a time-frame of about one day.
- **Data Import** The system has to cope with the data exported from the CAD system. The geometry of the trunk is given as a set of triangles, which exhibits different types of deficiencies and in general does not form a manifold. There is no notion of *inside* and *outside the trunk* in the data.

Apart from the main objective of maximizing the number of packed boxes, there are two additional objectives of minor importance. First, among all packings of the same cardinality, solutions with an easily recognizable structure are preferred. If possible, the boxes (or a majority thereof) should be aligned with the axes of a common coordinate system. This requirement helps to reproduce a computed solution in reality with a physical mockup.

Second, if possible, packings should be *tight*, i.e., there should be no uncovered space (*gaps*) in between of two boxes. Gaps should be avoided because they do not contribute to the discrete volume and cannot be used otherwise. If the packing is tight, and hence the uncovered space is close to the boundary of the trunk, one can easily identify regions of the trunk geometry suited for modifications. For example, one can identify regions where a small enlargement of the trunk allows to pack additional boxes.

1.1 Previous Work

Packing problems arise in many different fields of application. Consequently, a large number of terms related to packing are known in the literature, e.g., knapsack, bin packing, scheduling, pallet loading, stock cutting, strip packing, circuit board layout, nesting, VLSI design, map labeling, packaging, container stuffing, FPGA configuration, vehicle loading, and spatial arrangement. In general, a packing problem deals with the placement of components in an available space while optimizing a set of objectives and satisfying optional constraints.

Dyckhoff presents in [Dyc90] a typology for a large variety of cutting and packing problems. In such a general setting, the term *small item* denotes the bodies that are to be packed (in our case: the boxes). The term *large objects* (also called *container*) is used to denote the set of bodies in which the small items are packed. In our case, there is a single large object, the trunk. In this typology, our problem can be classified as 3/B/O/C, i.e., it is a three-dimensional problem (3), a subset of the small items is to be packed into a given large object (B), there is a single large object (O), and the items have congruent shape (C). Dyckhoff also discusses other criteria not covered by this classification string, e.g., quantity measurement (discrete, continuous), shapes of the items and objects (form, size, orientation), pattern restrictions and objectives. In our case, the container has an irregular shape and varies in size. The items are identical and have a regular, cuboid shape of fixed side lengths. There are no restrictions on the orientation of the items.

Extensive bibliographies of cutting and packing problems can be found in [SP92] and [DST97]. A review of solutions for practical packing problems was given by Dowsland and Dowsland [DD92]. The survey of Cagan et. al [CSY02] focuses on three-dimensional packing problems with complex, non-cuboid shapes. They discuss different types of algorithms, e.g., heuristics, branch-and-bound approaches, genetic algorithms, simulated annealing, and extended pattern search methods. The survey also covers the representation of geometric objects and predicate evaluation.

Fowler et al. [FPT81] have shown that already the two-dimensional problem of packing unit squares into a polygon with holes is *NP*-complete. A polynomial-time approximation scheme for this problem was given by

Hochbaum and Maas [HM85]. It was conjectured by Baur and Fekete in [BF01] that the simplified problem for simple polygons without holes is in P, but until now, this question is still open. A list of open problems in computational geometry is maintained by the Open Problems Project [DMO]. In particular, see items 55 (pallet loading) and 56 (packing unit squares in a simple polygon).

It is unknown, whether the unrestricted pallet loading problem, i.e., packing small $(a \times b)$-rectangles orthogonally into a large $(A \times B)$-rectangle, is contained in NP at all [Nel93]. This is due to the fact that the representation of an optimal solution might be very complex compared to the terse representation of the input. The restricted pallet loading problem allows only orthogonal guillotine patterns, which are obtained by a series of guillotine cuts, i.e., cuts from one edge of a previously cut rectangle to the opposite edge. A polynomial-time algorithm for the restricted pallet loading problem has been given by Tarnowski et al. [TTS94].

Closely related to packing problems are maximum stable set and maximum clique problems. A broad survey on these problems was written by Bomze et al. [BBPP99]. They state integer programming as well as continuous problem formulations. The survey contains an extensive discussion of exact algorithms as well as heuristics. The maximum stable set problem is NP-complete [GJ79]. An exact algorithm with time complexity $O(2^{0.276n})$ was given by Robson [Rob86], improving an earlier result by Tarjan and Trojanowski [TT77].

Schepers [Sch97] and Fekete et al. [FKT01] consider a three-dimensional bin packing problem with non-congruent cuboid items. They present a graph theoretic characterization (*packing classes*) of feasible packings and develop efficient lower bounds. These techniques allow a drastic reduction of the search space.

1.2 Related Work

In this section we present two closely related fields of work: a continuous approach for our packing problem, and a different trunk packing problem resulting from a standard of the Society of Automotive Engineers (SAE).

A Continuous Approach A completely different approach to our packing problem was pursued by Eisenbrand et al. [EFK+05]. This approach termed *Specialized Grand Canonical Simulated Annealing (SGCSA)* allows arbitrary position and orientation of the boxes. Later, the approach was further improved and automated by Rieskamp [Rie05].

Approaches based on simulated annealing [KGV83] use a *potential function* to evaluate configurations, i.e., the position and orientation of the boxes. Valid configurations should result in a low potential, whereas invalid config-

urations should be assigned high potential values. In this application, the potential function consists of three factors, the *interpenetration depth* and the *interpenetration volume* of box pairs, as well as the *wall potential*, i.e., the penetration of the exterior. The goal is to find a global minimum of the potential function.

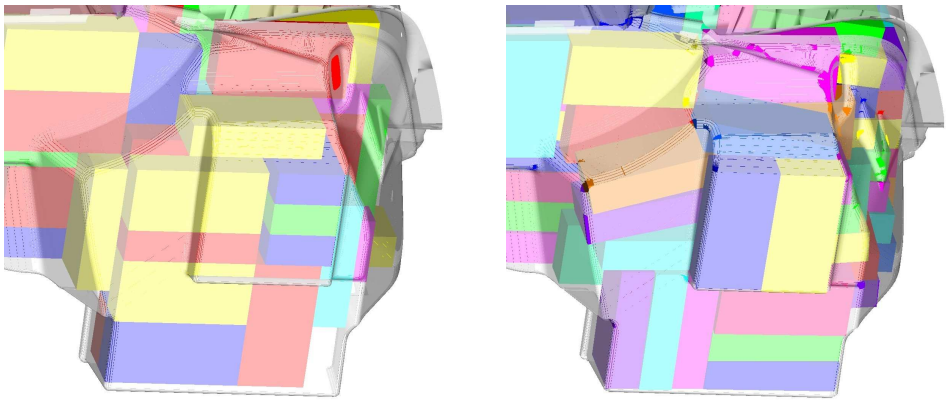
The SGCSA approach uses the Monte Carlo importance sampling algorithm [MRR+53] to explore the configuration space. Given a configuration, the algorithm performs a *trial move* to a nearby configuration. If the potential decreases, the new configuration is accepted. Otherwise, the new configuration is accepted with probability $p = e^{-\beta \cdot \Delta U}$, where ΔU denotes the increase in the potential. The acceptance of worse configurations is crucial to escape from local minima. The parameter $\beta > 0$ is called *inverse temperature* and controls the probability of moves increasing the potential. The value of β starts at a low value and is increased during the simulation according to a given schedule, thereby reducing the probability of uphill moves towards the end of the simulation.

The basic simulated annealing approach has been extended to handle the creation of boxes at promising positions as well as the destruction of boxes with a high penetration. Different classes of moves specifically tailored to the problem have been proposed to improve the performance.

While the continuous approach on its own yields unsatisfying results, it demonstrates its strength in conjunction with the discrete approach. Using a good combinatorial packing as initial solution drastically reduces the runtime of the continuous approach. Since the current implementation of SGCSA handles only small to mid-size instances in the given time limit, it is best applied to unsatisfying parts of a combinatorial solution. The benefit of arbitrary box placements can be most prominently perceived in regions with a complicated local geometry. Examples include small side-compartments holding tightly a few boxes (see Figure 1.2) or models with a curved lid.

Both the discrete as well as the continuous approach are implemented in the software system used by our industrial partner. Using both approaches we meet all prescribed quality requirements.

The SAE packing problem Ding and Cagan [DC03] consider a closely related packing problem: The American standard SAE J1100 [Soc01] defines the *interior volume index*, which is used to classify cars as subcompact, compact, mid-size or full-size cars. The volume of the trunk is one of many factors influencing this index. The standard defines a *standard luggage set* containing different types of luggage and a golf bag. With the exception of the golf bag, all luggage types are modeled as boxes of fixed size (see Table 1.1 for details). Wood replicas of the boxes and a partial packing are shown in Figure 1.3. The luggage capacity of the trunk is defined as the volume covered by items from the standard luggage set that can be packed



(a) discrete solution (b) continuous solution

Figure 1.2: Improvement due to arbitrary placements. In regions with a complicated geometry a discrete packing can often be improved by allowing boxes with arbitrary positions and orientations.

into the trunk. To make the distinction, we refer to this packing problem as *SAE packing problem* and to the boxes as *SAE boxes*.

Ding and Cagan [DC03] use an extended pattern search algorithm to attack the SAE packing problem. The basic pattern search algorithm is a deterministic direct search algorithm, which was first introduced by Hooke and Jeeves [HJ61] (see also [TT97]). It allows the local minimization of a real-valued function in n variables using only function evaluations. Given an initial point, a step length, and a set of pattern directions, the algorithm performs a series of exploratory moves corresponding to the pattern directions. If a new minimum is found, it is adopted as new base point from which the next iteration of exploratory moves starts. If no better solution is found, the step length is reduced.

luggage	size [mm]	volume [m^3]	letter	no.
Men's 2-suiter	$229 \times 483 \times 610$	0.067	A	4
Women's overnight	$165 \times 330 \times 457$	0.025	B	4
Women's pullman	$229 \times 406 \times 660$	0.061	C	2
Women's wardrobe	$216 \times 457 \times 533$	0.053	D	2
Women's train case	$203 \times 229 \times 381$	0.018	E	2
Men's overnight	$178 \times 356 \times 533$	0.034	F	2
Golf bag	$\approx 1143 \times 204 \times 204$	0.043	G	2
H-boxes	$152 \times 114 \times 325$	0.006	H	20

Table 1.1: Standard luggage set according to SAE J1100. The golf bag has non-cuboid shape; hence the given values are the sizes of the bounding box.

(a) standard luggage set (selected subset) (b) partial packing

Figure 1.3: Measurement of the trunk volume according to SAE J1100

Yin and Cagan [YC00] use the pattern search algorithm for three-dimensional packing problems and propose several extensions for performance improvement. The obtained results are superior to those of a simulated annealing approach. Ding and Cagan [DC03] apply the extended pattern search algorithm to the SAE packing problem. They incorporate problem-specific enhancements to address the difficulties. Unfortunately, the results are not compared with solutions obtained by human experts.

1.3 Our Contribution

In this thesis we present a combinatorial approach to a three-dimensional real-world packing problem. The task is to pack a maximum number of rigid boxes with side length ratios $4 : 2 : 1$ into the trunk of a car. Apart from the work described in Section 1.2, no fully automated solutions for this problem are known.

In contrast to many other packing problems studied in the literature, our container has an irregular, non-cuboid shape. Moreover, the number of items to be packed is very high.

As a theoretical result we show that our packing problem is *NP*-complete and present a polynomial-time approximation scheme. Unfortunately, this approximation scheme is not suited for instances of typical size, but it inspires a similar heuristic.

Motivated by packings produced by human experts so far, we discretize the problem in a two-fold way: We approximate the shape of the trunk by a three-dimensional cubic grid. Furthermore, we restrict the positions and orientations of the boxes to those aligned with the grid. This discrete packing problem can be reduced to a maximum stable set problem.

The approximation of the trunk requires special care, since the description of the trunk geometry contains several kinds of deficiencies, e.g., holes, dangling faces, and self-intersections. There is also no notion of inside or outside the trunk in the input data. We present an approach to approximate the shape of the trunk despite these difficulties.

We present different algorithms for the discrete packing problem, exact ones as well as heuristics. First we formulate the problem as an integer linear program (ILP). Typical problem instances are too complex for state-of-the-art ILP solvers, but an exact algorithm is still important for smaller subproblems. We also present a greedy algorithm as well as heuristics based on linear programming and local search. Other heuristics generate tight packings for the core of the trunk, thereby reducing the problem to a set of smaller subproblems.

Real-world data sets are used to study the performance of our algorithms with respect to algorithmic variants and implementation details. We compare our results with packings manually constructed by human experts. In most cases, our results meet the prescribed quality bounds. For some instances, we significantly outperform the expert solutions. All presented algorithms are implemented in an industrial-strength software system which is used by our project partner in the design process of new cars.

The remainder of this work is structured as follows: In Chapter 2 we define the packing problem from a formal point of view. We proof that the problem is *NP*-complete and present a polynomial-time approximation scheme.

Chapter 3 deals with several preprocessing steps. We explain the data import and point out several types of deficiencies in the input data. We present a simple heuristic to reconstruct the face normals and introduce a space partition for performance improvements.

We discuss the discretization process in Chapter 4. First, we present efficient algorithms for the fundamental geometric predicates. We explain how to compute an inner approximation of the trunk in the presence of deficiencies in the input data. Given such an approximation by a cubic grid, we discuss how to efficiently update the computed information after a change of the geometric parameters of the grid. Based on this, we discuss how to obtain a *good* discretization.

In Chapter 5 we present different combinatorial algorithms for our discrete packing problem. The set of algorithms is divided into two classes, the direct approaches, which can solve an instance on its own, and the divide-and-conquer approaches, which need a direct approach to solve smaller subproblems.

We evaluate the presented algorithms on real-world instances in Chapter 6. We use a small set of models to discuss the impact of several algorithm variants and implementation details. We also present a simple technique to reduce the effect of the restrictions due to the discretization. A larger set

of instances is used to compare the results of our algorithms to packings manually constructed by human experts.

Finally, we give a conclusion and present some directions for further work.

1.4 Notation

In this section we briefly introduce the notation that is used in this work.

- **Scalars** are written as lower-case letters, such as m, n, x, y, z and λ. The letters i, j, k, l, m and n are used for integral values.
- **Vectors** are denoted by lower-case bold-face letters, such as $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$. A vector $\boldsymbol{a} \in \mathbb{R}^n$ is usually viewed as a column vector

$$\boldsymbol{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}.$$

  Row vectors are written as transposed column vectors, such as $\boldsymbol{a}^{\mathrm{T}}$. The scalar product of two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is written as $\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b}$.
- **Matrices** are written as bold-face upper-case letters, such as $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$. The transposed matrix of $\mathbf{A}$ is denoted as $\mathbf{A}^{\mathrm{T}}$. The identity matrix is written as $\mathbf{I}$.
- **Sets** are denoted by upper-case letters, such as A, B and C. The script letter $\mathcal{C}$ denotes sets of cliques, i.e., sets of node sets. The script letter $\mathcal{G}$ is used to denote grids (which can also be seen as sets of cells), in contrast to the letter G, which denotes graphs.

Chapter 2

Theoretical Results

2.1 Problem Definition

A first definition of the central problem from an industrial point of view has already been presented in Chapter 1. In this section, we define the problem formally. We also introduce a few other important terms.

The central problem considered in this work is the Continuous-Box-Packing problem:

Definition 2.1 (Continuous-Box-Packing–optimization variant). *Given a polyhedral domain $P \subseteq \mathbb{R}^3$ homeomorphic to a ball, compute a maximum packing of boxes of size $4 \times 2 \times 1$.*

Definition 2.2 (Continuous-Box-Packing–decision variant). *Given a polyhedral domain $P \subseteq \mathbb{R}^3$ homeomorphic to a ball, and $k \in \mathbb{N}$, decide whether there is a packing of at least k boxes of size $4 \times 2 \times 1$.*

The volume to be packed is given by a polyhedral domain, i.e., a connected region in three-dimensional space bounded by linear elements. Note that this domain is not necessarily convex nor star-shaped. For typical instances, the domain is homeomorphic to a ball, but we also consider subproblems for which this is not the case.

For the purpose of tight packing patterns, we allow lower-dimensional intersections among the boxes. Alternatively, one could consider the boxes as open sets.

By scaling the input we can replace the fixed side lengths of 200mm, 100mm and 50mm by small integers. The scaling maintains the integral side length ratios which are very important for our discretization process later. Most of the presented algorithms work for arbitrary integral side length ratios, while one algorithm is specifically designed for the given ratios $4 : 2 : 1$.

Note that Definition 2.1 contains a single objective, namely maximization of the number of boxes. Additional objectives like orientation and relative

position of the boxes have been omitted. In this work, we concentrate on maximizing the size of the packing. We keep the additional objectives in mind and evaluate the presented algorithms also with respect to those objectives.

In this thesis we focus on discrete approaches for the trunk packing problem. Hence we next introduce the DISCRETE-BOX-PACKING problem, which is a discrete variant of the CONTINUOUS-BOX-PACKING problem.

Definition 2.3 (DISCRETE-BOX-PACKING–optimization variant). *Given an orthogonal, cubic grid $\mathcal{G}$ with unit spacing, a subset I of the grid cells and $n \in \mathbb{N}$, compute a maximum packing of cell-aligned boxes with side lengths $4n$, $2n$ and n such that only cells in I are covered and each cell in I is covered by at most one box.*

Definition 2.4 (DISCRETE-BOX-PACKING–decision variant). *Given an orthogonal, cubic grid $\mathcal{G}$ with unit spacing, a subset I of the grid cells and $k, n \in \mathbb{N}$, decide whether there is a packing of at least k cell-aligned boxes with side lengths $4n$, $2n$ and n such that only cells in I are covered and each cell in I is covered by at most one box.*

The DISCRETE-BOX-PACKING problem restricts the original problem in a two-fold way. The polyhedral domain P is restricted to the union of cells in I, i.e., to a union of unit cubes. Furthermore, the allowed orientations and positions of the boxes have been drastically reduced. In the discrete setting, there are only six possible orientations per box. The set of possible positions has been restricted to those for which the boundary of the box is aligned with cell boundaries.

Note that all grids considered in this work have orthogonal axes. In the following, we omit the adjective orthogonal for simplicity.

We show in Section 2.2 that the CONTINUOUS-BOX-PACKING problem is *NP*-hard and that the DISCRETE-BOX-PACKING problem is *NP*-complete. We refer to Section 4.1 for further discussion of the DISCRETE-BOX-PACKING problem.

In the remainder of this section we want to introduce the maximum stable set problem which is closely related to the DISCRETE-BOX-PACKING problem.

Definition 2.5 (STABLE SET). *A subset S of the nodes V of a graph $G = (V, E)$ is called* stable set *(or* independent set*) if S does not contain a pair of adjacent nodes.*

Definition 2.6 (MAXIMUM STABLE SET–optimization variant). *Given a graph $G = (V, E)$, compute a stable set of G of maximum size.*

Definition 2.7 (MAXIMUM STABLE SET–decision variant). *Given a graph $G = (V, E)$ and $k \in \mathbb{N}$, decide whether G has a stable set of size k.*

Maximum stable set problems are well-studied (see [BBPP99] for a broad survey). The maximum stable set problem is *NP*-complete [GJ79].

Stable set problems are closely related to discrete packing problems. This relation becomes apparent in the concept of the *conflict graph* (also called *intersection graph*).

Definition 2.8 (Conflict Graph). *Let X be a finite set of geometric objects with fixed positions. The* conflict graph $G = (V, E)$ *of X is defined as follows: For each object $x_i \in X$, there is a corresponding node $v_i \in V$. Two nodes $v_i, v_j \in V$ are adjacent if the interiors of the objects x_i and x_j intersect.*

Conflict graphs are useful as an alternative representation of discrete packing problems. Actually, we can reduce the Discrete-Box-Packing problem to a maximum stable set problem.

Proposition 2.9. *The* Discrete-Box-Packing *problem can be reduced to a maximum stable set problem in the corresponding conflict graph.*

Proof. Let $(\mathcal{G}, n, I)$ denote an instance of the Discrete-Box-Packing problem. Let X denote the set of feasible box placements, i.e., the set of cell-aligned boxes of size $4n \times 2n \times n$ covering only cells in I. Define G as the conflict graph of X.

There is a trivial one-to-one relation between solutions of the Discrete-Box-Packing problem $(\mathcal{G}, n, I)$ and the stable sets in the conflict graph G, in particular every packing of $(\mathcal{G}, n, I)$ corresponds to a stable set in G of the same size and vice versa. □

2.2 *NP*-completeness

In this section, we prove that the Continuous-Box-Packing problem is *NP*-hard and that the Discrete-Box-Packing is *NP*-complete. In a first step, we show that a two-dimensional, discrete version of these problems is *NP*-complete.

Definition 2.10 ($m \times n$-Rectangle-Packing). *Given integers k, m, $n \in \mathbb{N}$, $m \geq n$ and finite sets $H \subseteq \mathbb{Z}^2$ and $V \subseteq \mathbb{Z}^2$, decide whether it is possible to pack at least k axis-aligned rectangles of size $m \times n$ in a way such that the lower left corner of a horizontal or vertical rectangle coincides with a point in H or V, respectively.*

This problem is trivial for $m = n = 1$. For $m = 2, n = 1$, the problem is equivalent to a matching problem for the corresponding conflict graph, and hence can be solved in polynomial time [GJ79]. We now turn to the cases $m = n = 3$ and $m = 6$, $n = 3$.

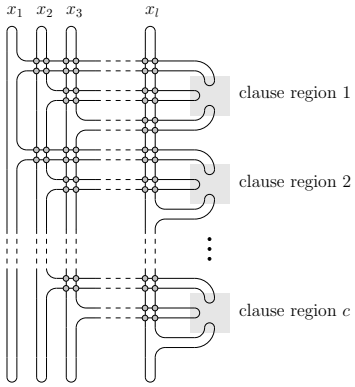


Figure 2.1: High-level view of the conflict graph constructed from a boolean formula with l variables and c clauses. For each variable there is a node cycle of even length. These node cycles cross each other at crossover regions (marked with dots). The clauses of the formula are represented by clause regions.

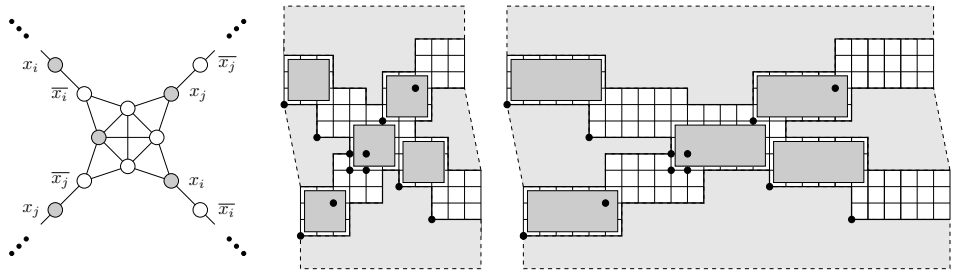
Proposition 2.11. 3×3-Rectangle-Packing *and* 6×3-Rectangle-Packing *are NP-complete.*

Proof. The case $m = n = 3$ has been shown by Fowler et al. in [FPT81] using a reduction of *3-SAT* to this problem. We use the same technique here for $m = 6, n = 3$.

Suppose we are given a boolean formula in conjunctive normal form (*CNF*) with three literals per clause. First we construct a graph and compute a number k, such that the graph has a maximum stable set of size k iff the formula is satisfiable. Then we show how to compute the sets H and V such that the conflict graph of the corresponding packing problem equals the previously constructed graph. Thus the packing problem has a solution of size k iff the boolean formula is satisfiable.

Given a boolean formula F in CNF, let l denote the number of variables and c the number of clauses of F. We construct a graph as follows: For each variable $x_i, 1 \leq i \leq l$ there is a cycle of nodes of even length. Such a cycle has exactly two stable sets of maximum cardinality which correspond to the two possible assignments of x_i. A high-level view of the graph is depicted in Figure 2.1.

A cycle of nodes consists of intersection-free segments of even length and *crossover regions* where the paths of two cycles cross each other. These regions have the property that the size of a stable set can be increased by one iff the variable assignments encoded by the stable set are consistent for all branches of the crossover region. In each maximum stable set the assignment of a variable is propagated to the opposite branch of the crossover region, independently of the assignment of the other variable. See Figure 2.2(a) for the shape of the graph at crossover regions.



(a) Conflict graph (b) $m = n = 3$ (c) $m = 6, n = 3$

Figure 2.2: Crossover region for variables x_i and x_j. Note that the segments between two adjacent crossover regions have even length. Each maximum stable set contains exactly one of the four center nodes, and hence the assignment of a variable is consistent in both branches, independently of the assignment of the other variable. In figures (b) and (c), the lower left corners of feasible placements are marked with a dot. The set of feasible placements is constructed such that its conflict graph equals the graph in figure (a).

For each clause of the formula there is a *clause region* where the cycles of the three involved variables are brought into proximity. These regions have the property that the size of a stable set can be increased by one iff the corresponding clause is satisfied. See Figure 2.3(a) for the shape of the graph at clause regions.

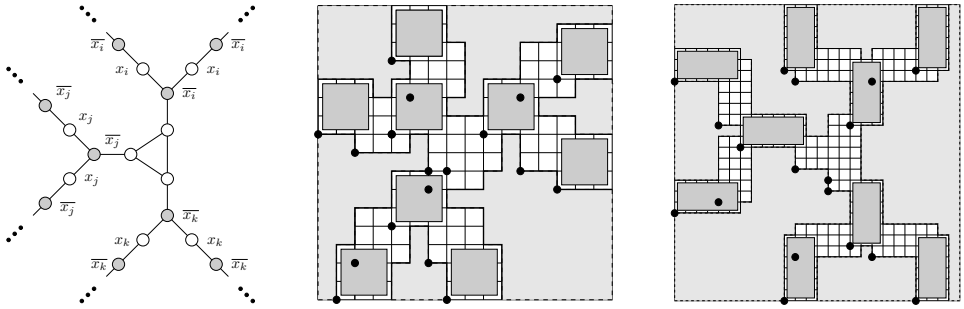
Let k' be the number of all nodes, excluding the four inner nodes of crossover regions and the three inner nodes of clause regions. Let

$$k := \frac{k'}{2} + c + \#\text{crossover regions}. \tag{2.1}$$

Now it holds that there exists a stable set of size k iff the formula is satisfiable. Each variable assignment that satisfies the formula induces a stable set of size k. On the other hand, each stable set of size k corresponds to a variable assignment that satisfies the formula.

It is straightforward (but tedious) to compute the sets H and V such that the conflict graph of the corresponding packing problem is the graph we just constructed. Details for crossover and clause regions can be seen in Figure 2.2 and 2.3. This construction can be carried out in $O(cl)$ time using $O(cl)$ space.

It holds that the given boolean formula F is satisfiable iff the constructed instance (H, V, k) of 3×3-Rectangle-Packing respectively 6×3-Rectangle-Packing has a solution. The $m \times n$-Rectangle-Packing is in NP, since a given solution can be verified in quadratic time. Hence 3×3-Rectangle-Packing and 6×3-Rectangle-Packing are NP-complete. □



(a) Conflict graph (b) $m = n = 3$ (c) $m = 6, n = 3$

Figure 2.3: Clause region for $x_i \vee x_j \vee x_k$. If the clause is satisfied, i.e., the assignment of at least one variable is changed, it is possible to add one of the three center nodes to the stable set. In figures (b) and (c), the lower left corners of feasible placements are marked with a dot. The set of feasible placements is constructed such that its conflict graph equals the graph in figure (a).

Note that in the previous proof the values of m and n are crucial for the shape of the intersection and clause regions. Similar constructions are possible for all $m \geq n$ and $n \geq 3$. But such a construction does not work for the case $m = n = 2$.

An alternative proof can be obtained from a result of Berman et al. [BJL+90] on generalized planar matchings. Their technique can also be applied to the remaining cases $m \geq 3$, $n = 1$ and $m \geq 2$, $n = 2$.

Now we are ready to state our main result about the *NP*-hardness of the Continuous-Box-Packing problem.

Theorem 2.12. Continuous-Box-Packing *is NP-hard.*

Proof. We reduce the 6×3-Rectangle-Packing problem to Continuous-Box-Packing.

Let (k, H, V) denote an instance of 6×3-Rectangle-Packing. Intuitively, our approach works as follows: We scale the shape induced by the sets H and V by $\frac{2}{3}$ and extrude it into the third dimension with z-coordinates ranging from 0 to 1. On the bottom of this object we glue a sufficiently large box of height $\frac{1}{2}$. The size of the box is chosen such that the result of the construction is homeomorphic to a ball. More formally, $P \subseteq \mathbb{R}^3$ is constructed as follows:

$$P_H := \bigcup_{(x,y)\in H} \left[\frac{2}{3}x,\ \frac{2}{3}x+4\right] \times \left[\frac{2}{3}y,\ \frac{2}{3}y+2\right] \times [0,\ 1]\,, \tag{2.2}$$

$$P_V := \bigcup_{(x,y)\in V} \left[\frac{2}{3}x,\ \frac{2}{3}x+2\right] \times \left[\frac{2}{3}y,\ \frac{2}{3}y+4\right] \times [0,\ 1]\,, \tag{2.3}$$

$$P := P_H \quad \cup \quad P_V \quad \cup \quad X \times Y \times \left[-\frac{1}{2}, 0\right], \tag{2.4}$$

where X and Y denote the projection of $P_H \cup P_V$ onto the first and second coordinate, respectively.

This construction can be carried out in polynomial time. It is clear that a projection of a solution for the CONTINUOUS-BOX-PACKING problem onto the first two coordinates corresponds to a solution for the 6×3-RECTANGLE-PACKING of the same size and vice versa. □

It is unclear whether the CONTINUOUS-BOX-PACKING problem is contained in *NP*, because there are instances that can be encoded very efficiently, e.g., simple geometric shapes like cuboids. This is similar to the unrestricted pallet loading problem, i.e., packing a maximum number of small $(a \times b)$-rectangles into a large $(A \times B)$-rectangle. The representation of an optimal solution might be very complex compared to the terse representation of the input [Nel93].

The *NP*-completeness of the DISCRETE-BOX-PACKING problem follows directly from Proposition 2.11.

Theorem 2.13. DISCRETE-BOX-PACKING *is NP-complete.*

Proof. The proof is very similar to the proof of Theorem 2.12. Given an instance (k, H, V) of 6×3-RECTANGLE-PACKING, we define the polyhedral domain P as described in that proof. Now we construct an instance of DISCRETE-BOX-PACKING as follows: First we scale the domain P by a factor of two. We choose $n = 2$, i.e., the boxes have a size of $8 \times 4 \times 2$. The grid $\mathcal{G}$ is centered at the origin and aligned with the axes of the coordinate system. Now the domain P can be decomposed into a set of grid cells. Define I to be this set.

Given the sets H and V of the 6×3-RECTANGLE-PACKING problem, the set I can be constructed directly in polynomial time without the explicit computation of P. A projection of a solution for the DISCRETE-BOX-PACKING problem onto the first two coordinates corresponds to a solution for the 6×3-RECTANGLE-PACKING of the same size and vice versa.

The DISCRETE-BOX-PACKING problem is *NP*, since a given solution can be verified in quadratic time. Hence the DISCRETE-BOX-PACKING problem is *NP*-complete. □

2.3 An Approximation Scheme

In this section we present a polynomial-time approximation scheme for the DISCRETE-BOX-PACKING problem. We present the approximation scheme for a special case, namely for cubes. It can be easily generalized for boxes.

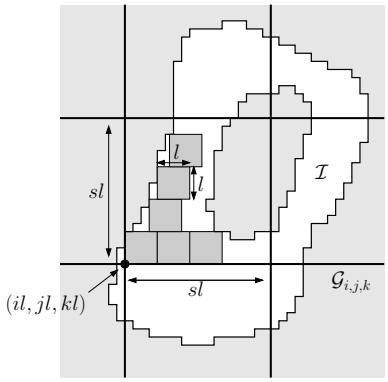


Figure 2.4: The approximation scheme uses grids with spacing sl to partition the set $\mathcal{I}$ into subregions. Each subregion is optimally packed, e.g., by complete enumeration.

Definition 2.14 (Cube-Packing). *Given integers k, $l \in \mathbb{N}$ and a finite set $I \subseteq \mathbb{Z}^3$, decide whether it is possible to pack at least k axis-aligned cubes of side length l in a way such that the lexicographic smallest vertex of each cube coincides with a point in I.*

It follows from Proposition 2.11 and the subsequent remarks that the Cube-Packing problem is NP-complete for $l \geq 2$. However, there is a polynomial-time approximation scheme.

Theorem 2.15. *There exists a polynomial-time approximation scheme for* Cube-Packing.

The approximation scheme is based on the *shifting strategy*, which was first presented by Hochbaum and Mass [HM85]. Our proof is an extension of a proof for the two-dimensional analogon of Theorem 2.15 by Baur and Fekete [BF01].

Proof of Theorem 2.15. We show that for any fixed $\varepsilon > 0$ there is a polynomial-time algorithm that computes an $(1 - \varepsilon)$-approximation.

Consider a cubic grid with spacing sl centered at the origin. The parameter $s \in \mathbb{N}$ is a constant (depending on ε) and will be determined later. The grid subdivides the space into subregions, whose number is linear in the size of I (see Figure 2.4). For each such subregion, the heuristic computes an optimal packing, which can be computed in constant time, e.g., by complete enumeration of all possible packings for this subregion. The packings of all subregions are joined and form a solution corresponding to the given grid.

For $0 \leq i, j, k < s$, let $\mathcal{G}_{i,j,k}$ denote the grid that is obtained by a translation of the initial grid by the vector (il, jl, kl). For each of the s^3 grids, we compute a corresponding solution. Finally, we report the best solution found.

For the analysis, we consider an optimal solution of the problem and compare it to a heuristic solution for some grid $\mathcal{G}_{i,j,k}$. Within each subregion of the grid, the solution of our heuristic is not worse than the corresponding subset of the optimal solution. Cubes of an optimal solution that do not entirely fall into a single subregion can never be part of the heuristic solution. It can be easily seen that for any axis-aligned cube this can happen for at most $s^3 - (s-1)^3$ grids: If a cube of size l intersects two or more subregions of $\mathcal{G}_{i,j,k}$, then it never does so for grids $\mathcal{G}_{i',j',k'}$ with $i \neq i' \wedge j \neq j' \wedge k \neq k'$. There are exactly $(s-1)^3$ such tuples (i', j', k'), hence a cube intersects two or more subregions in at most $s^3 - (s-1)^3$ cases.

Let OPT denote the cardinality of an optimal solution. By the previous observation, the sum of all s^3 heuristic solutions is at least

$$s^3 OPT - (s^3 - (s-1)^3)\, OPT = (s-1)^3\, OPT.$$

This implies that the best of all heuristic solutions has cardinality at least $\left(\frac{s-1}{s}\right)^3 OPT$. Hence the desired approximation ratio of $1-\varepsilon$ is reached for $s \geq \left(1 - \sqrt[3]{1-\varepsilon}\right)^{-1}$. □

The theorem does not only hold for cubes of side length l, but can also be generalized for boxes of side length at most l. No further changes in the proof are required.

Unfortunately, the presented approximation scheme is not useful for our problem. The reason lies in the size of the subregions that are to be optimally solved. The volume of these subregions can be up to $V = (sl)^3$. For example, for $l = 200$mm and $\varepsilon = 0.1$, we have $s = 29$ and $V = 195112\,\mathrm{l}$, which is two to three orders of magnitude larger than our instances.

Nevertheless, we pursue the idea of subdividing the space by axes-parallel planes in the Partition algorithm (see Section 5.2.3). In contrast to the presented approximation scheme, this algorithm handles resulting subregions not independently of each other.

Chapter 3

Preprocessing

This chapter deals with the preprocessing of the input data. The given CAD data sets describe the volume to be packed. Unfortunately, this description often is defective or incomplete. The preprocessing includes conversion of the CAD data, modification according to our needs, and precomputation of accelerating data structures like space partitions.

3.1 Data Import

Our input data originate from commercial CAD systems. In such a system the geometry of a car is modeled by free form surfaces. However, we do not work on this high-level description, but require a triangulation of the faces.

The data is exported as VRML 1.0 (see [BPP96, ANM96] for a definition of the VRML 1.0 file format). The following VRML nodes are of major importance:

- `Coordinate3` - defines vertices in three-dimensional space
- `Normal` - defines vectors in three-dimensional space
- `IndexedFaceSet` - defines faces in three-dimensional space by referencing the vertices and normal vectors defined before
- `Material` - defines material attributes, e.g., colors
- `MatrixTransform` - defines coordinate system transformations

The data also contains some other node types that are not relevant in our case, e.g. `IndexedLineSet` or `ShapeHints`.

For efficiency reasons, we do not use existing libraries that are offering VRML import, but implemented our own VRML reader. A compilation of supported and unsupported nodes is shown in Table 3.1. The VRML formatted data are converted into a custom file format specially designed for

Supported nodes	Unsupported nodes
Cone	AsciiText
Coordinate3	DirectionalLight
Cube	FontStyle
Cylinder	IndexedLineSet
DEF	Info
Group	OrthographicCamera
IndexedFaceSet*	PerspectiveCamera
LOD*	PointLight
Material*	PointSet
MaterialBinding	ShapeHints
MatrixTransform	SpotLight
Normal	Texture2
NormalBinding	TextureCoordinate2
Rotation	Texture2Transform
Scale	WWWAnchor
Separator*	WWWInline
Sphere	
Switch	
Transform	
TransformSeparator	
Translation	
USE	

Table 3.1: Supported and unsupported VRML nodes. Subtrees rooted at unsupported nodes are ignored by the parser. Nodes marked with an asterisk (*) are only partially supported.

our needs. This file format basically encodes an indexed face set enriched by some additional information. Each file consists of some header data and three major parts: a list of colors, a list of vertices and a list of triangles. Each triangle consists of three vertex indices, a color index and some flags. The normal of a triangle is implicitly given by the order of its vertices. Colors are used to encode additional information for each triangle (see Section 3.3).

Note that no adjacency information is stored, i.e., no information about the local neighborhood of a face or vertex is available. Such information is not stored for three reasons:

- The information is of limited use (none of the major algorithms would benefit from it).
- The information is only partly available (namely, only for small groups of triangles belonging to one `IndexedFaceSet`).
- Reconstruction of the information is difficult due to deficiencies in the input data (see Section 3.2).

In order to reduce the size of the data sets and to get rid of redundancies, we perform some simple modifications of the data during the conversion

process. The following four modifications are performed:

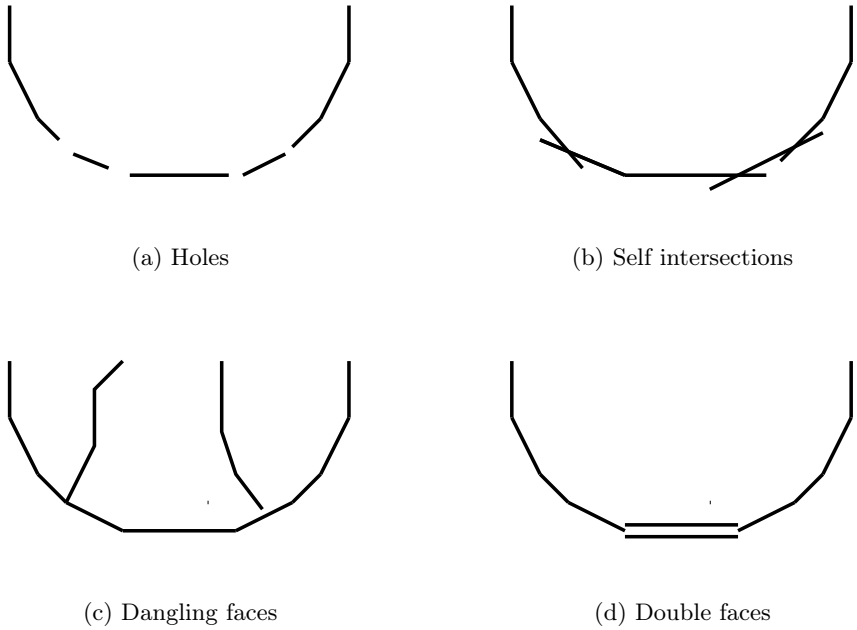
- **Clipping** The provided data sets often contain features that lie far away from the region of interest. Therefore, the user may define a box that denotes the region of interest. All triangles are clipped against this box.
- **Removal of tiny triangles** The triangulation process produces an immoderate fine triangulation for regions with a high curvature. We prune all triangles with an area below a threshold. The resulting holes are usually quite small. The formation of such small holes is not an additional burden since we have to deal with them anyway (see Section 3.2). A better, but more costly approach is to perform some kind of mesh simplification.
- **Removal of unused vertices** The input data might contain vertices that are not referenced by any face. Further vertices can become unreferenced after the steps above. All such vertices are pruned.
- **Unification of vertices** The input data contains multiple vertices at the same location. All instances of a vertex are identified and collapsed into one instance.

3.2 Deficiencies in the Input Data

The shape of a car trunk can be modeled as a polyhedral domain, i.e., a connected region in three-dimensional space bounded by linear elements. A natural way to describe such a polyhedral domain is to specify the structure of its boundary, which is a two-dimensional manifold consisting of polygonal faces, edges and vertices. Often, polygons are decomposed into a set of planar triangles. Such a boundary description is called *triangular mesh*.

Unfortunately, the given input data does not describe a two-dimensional manifold. There are several kinds of deficiencies in the structure of the triangles (see Figure 3.1):

- **Holes** The input data contains holes, i.e., the description of the boundary of the volume to be packed is incomplete. This is most likely caused by data sets that are not available, e.g., parts that are not yet designed. Smaller holes may also arise from triangulation errors. Another source of holes are surface patches that do not fit together exactly and cause holes that are often long and skinny. We discuss the difficulties arising from holes in more detail in the remainder of this section.
- **Self Intersections** The set of triangles that forms the boundary of the volume to be packed intersects itself. This often happens in regions where different surface patches come into proximity. In the early



(a) Holes (b) Self intersections

(c) Dangling faces (d) Double faces

Figure 3.1: Various kinds of deficiencies in the triangular mesh (cross sections)

design process, the geometry of the trunk boundary is only vague and is gradually refined. It is not uncommon, that different parts that constitute the boundary do not fit together very well and cause such intersections.

- **Dangling Faces** There are three (or more) faces that share a common edge, or alternatively, an edge of a face is contained in (the interior of) another face. Such situations are also caused by different surface patches that come into close proximity. However, this is often by design and not an error.

- **Double Faces** The term *double face* denotes the description of the very same boundary element by two parallel faces that differ by a small offset in normal direction. These faces originate from sheets that —starting with a certain thickness— are modelled by their surface, resulting in two parallel faces. Both faces may be triangulated in different ways. Note that such a double face itself does not necessarily cause the non-manifoldness of the structure.

In summary, one can say that the set of triangles has several kinds of errors such that it does not constitute a two-dimensional manifold. If the

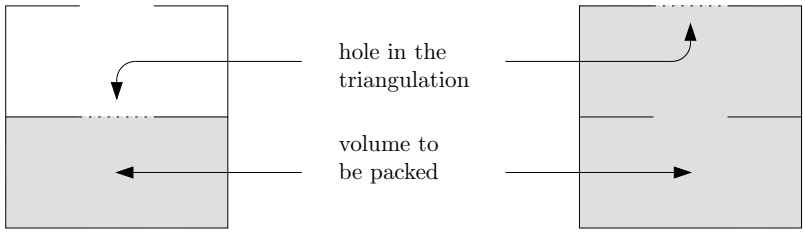


Figure 3.2: The location of the hole depends on the volume to be packed. Both cases cannot be distinguished solely based on the geometry of the trunk.

term *triangular mesh* is used in the following, it is always to be understood as a triangular mesh that contains these kinds of errors.

Looking again at the classification of the errors above, there is an important difference between holes and the other three kinds of errors. If there are no holes, the set of triangles still separates the volume to be packed from the remaining space. (It might happen, that the remaining space is divided into several unconnected components.) The containment test can still be performed by computing a path to a known reference point. But if the boundary has holes, this is no longer possible, since the path might unwittingly pass through such a hole.

On the other hand, the absence of other kinds of errors except holes allows the elimination of the holes. It is possible to identify and close them with additional triangular patches. This procedure might misrepresent the (unknown) local geometry, however, the structure of the triangles is altered towards a two-dimensional manifold. But such an approach is not possible in the presence of other errors. Figure 3.2 shows an example in which the location of a hole depends on the volume to be packed. Both cases cannot be distinguished solely based on the given geometry data. It is not possible to locate holes and fix them without user interaction.

Since holes cannot be eliminated, we have to deal with them. Looking closer at our application, we actually do not need to decide the containment problem for points, but for boxes of a given size. In particular, we want to avoid that a box located in the interior in the trunk can be moved along a path leading to a point far outside without intersecting the boundary of the trunk.

We distinguish two kinds of holes: *large holes* and *small holes*. Holes are called *large* if a box of fixed size can pass through the hole, otherwise they are called *small*. In Chapter 4 we explain how to classify regions of the space as interior or exterior. This approach allows the handling of small holes.

We demand that there are no large holes present. Otherwise, they have to be eliminated interactively by the user. This is rather easy because we already handle small holes and other kinds of errors. There is no need to construct a perfect fitting surface patch; an approximate placement of few sufficiently large triangles often suffices.

3.3 Face Normals

A further deficiency of the input data not mentioned in the previous section is the missing or incorrect information about face normals. We call a face normal *correct* if it points towards the outside of the trunk, i.e., away from the region to be packed. If one considers the interior of the trunk as a rigid body, our definition of correct face normals coincides with the convention for rigid bodies. We do not care about faces that do not bound the region to be packed. Their face normals and orientations do not matter in our application.

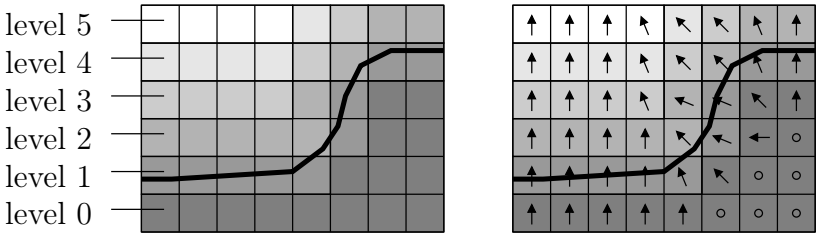
Why are face normals needed at all? Why is the orientation of the faces that bound the trunk important?

The main reason is the deformability of the trunk. The boundary of the trunk is not rigid, but rather deformable, at least in parts. For example, boundary parts made of plastic give way if there is no rigid support behind them. Or the fabric that covers the interior trim can often be slightly compressed. Thus one gains additional space ranging from fractions of a millimeter to a few centimeters. This additional space is heavily exploited in practice and has also to be taken into account by our software. In order to increase the region to be packed, a triangle may be shifted up to a specified distance in its normal direction. Since VRML 1.0 does not allow additional user-defined attributes, material attributes like color are used to encode this information.

Moreover, correct face normal are valuable for visualization. The term *front-face culling* refers to a rendering mode where all faces whose normal points towards the viewer (rather than away from the viewer) are ignored. Similarly, *back-face culling* ignores all faces whose normal points away from the viewer.

Back-face culling is often used to speed up visualization. In the case of two-dimensional manifolds, it is safe to skip all faces whose normal points away from the viewer since such faces are always concealed by other faces (provided that the viewer is located outside of the manifold). In our case, front-face culling is a much more useful feature, since it allows to look *into* the trunk. Face normals are also needed for computing lighting effects, e.g., face shading.

We have seen that correct face normals are mandatory for the packing process and useful for visualization. Thus we have to find a way to reconstruct them. Known algorithms do not work in our setting. If the boundary is a two-dimensional manifold, all face normals can be reconstructed given a single correct face normal. If the region to be packed is convex, it suffices to specify a point contained in it. But these preconditions are not fulfilled in our case. In the following we describe a simple heuristic that works sufficiently well.



(a) Cell levels (b) Flow directions

Figure 3.3: First two phases of the heuristic to fix the face normals. First the cell levels are computed based on the distance to the next cell completely contained in the interior of the trunk. Then the flow is computed based on these cell levels.

First we give an informal description of the heuristic and the intuition behind it. Imagine a flow that originates from the interior of the trunk, or to be more precise, in the region of the trunk that is to be packed. The flow passes through the faces that bound the interior and runs towards a sufficiently large sphere. Given a triangle of the boundary, one observes that the direction of the flow at this triangle is often similar to the face normal. More precisely, the angle between flow direction and face normal is often much smaller than 90 degrees. Thus the direction of the flow can be used to determine the orientation of the faces.

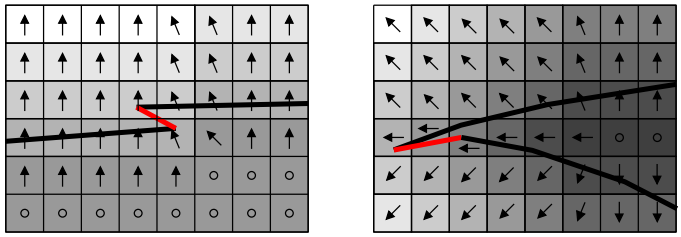
A more formal description of the algorithm is presented in Algorithm 3.1. The algorithm takes two parameters: the set T of triangles and a grid $\mathcal{G}$. The purpose of the grid is twofold: first it specifies an approximation of the region to be packed, and second it provides a discretization of the space on which the flow computation is based. The details of obtaining such a grid are depicted in Chapter 4. Here we simply state the essentials that are needed for the algorithm.

The grid $\mathcal{G}$ consists of uniform, cubic cells with side length l. The variable $\mathcal{G}$ is also used to denote the set of all cells of the grid $\mathcal{G}$. The cells are identified by a three-dimensional integral vector called *index*. More precisely, a cell $c \in \mathcal{G}$ with index $(x, y, z) \in \mathbb{Z}^3$ comprises

$$\boldsymbol{o} + [xl, xl + l) \times [yl, yl + l) \times [zl, zl + l) \subseteq \mathbb{R}^3,$$

where $\boldsymbol{o} \in \mathbb{R}^3$ denotes the origin of the grid. Two cells are called *neighbors* iff their indices differ in exactly one component by 1. The set $I \subseteq \mathbb{Z}^3$ contains exactly the cells that are completely contained in the interior of the trunk.

The algorithm consists of three phases (see Algorithm 3.1). In the first phase we use breadth first search (BFS) to compute for each grid cell $c \in \mathcal{G}$ its minimum distance to the set I. This distance is stored in $level[c]$. An example can be seen in Figure 3.3(a).



(a) Small features (b) Cusps

Figure 3.4: Examples in which the heuristic fails to reconstruct the correct face normals. Red faces indicate incorrectly computed face normals.

The second phase computes the direction of the flow for each cell $c \in C$. The direction is based on the levels of the corresponding cell and its six neighbors. Note that the difference of the levels of two neighboring cells is at most 1. Neighbors of c with the same level do not contribute to the flow direction at all, and to the same extent otherwise. More precisely they contribute a directional vector that connects its center and the center of the cell c. The orientation of this vector is chosen such that it points from the cell with lower level to the cell with higher level. Finally, the sum of all contributing vectors is normalized. The flow directions for the previous example are shown in Figure 3.3(b).

The last phase looks at each triangle $t \in T$ in turn and computes the average flow direction of all cells intersected by that triangle. The orientation of t is reversed if the angle between the normal of the triangle and the average flow direction is larger than $\frac{\pi}{2}$.

The heuristic presented above works pretty well in our case. Nevertheless there are situations in which it fails. Two such examples are shown in Figure 3.4. The left figure shows a small face (compared to the spacing of the grid) whose normal deviates much from the normals of the surrounding mesh. Such small local features are not recognized due to the discretization imposed by the grid. As a result, computed face normals for small features may be wrong.

Figure 3.4(b) shows a cusp that bounds the region to be packed. Consider the lower arc at the tip of the cusp. The computed flow and the (correct) face normal enclose an angle larger than $\frac{\pi}{2}$. Hence the heuristic computes a wrong face normal.

Such errors are a minor annoyance for visualization, but they are not critical for the following computations. The deformability of the trunk in such regions is quite small and often zero. Thus the effects of a wrong face normal are very limited. Additionally, tiny regions as the interior of the cusp are far too small to be packed with boxes. The extent of such errors can be further reduced by using grids with finer spacing.

Algorithm 3.1 Reconstruction of face normals

FaceNormals $(T, \mathcal{G})$

$Q \leftarrow$ empty FIFO queue ▷ compute $level[c]$
for all cells $c \in C$ **do**
  if $c \in I$ **then**
    $level[c] \leftarrow 0$
    Enqueue (Q, c)
  else
    $level[c] \leftarrow \infty$
  end if
end for
while $Q \neq \emptyset$ **do**
  $c \leftarrow$ Dequeue (Q)
  for all neighboring cells c' of c **do**
    if $level[c'] = \infty$ **then**
      $level[c'] \leftarrow level[c] + 1$
      Enqueue (Q, c')
    end if
  end for
end while

for all cells $c \in C$ **do** ▷ compute $direction[c]$
  $direction[c] \leftarrow (0, 0, 0)^T$
  for all neighboring cells c' of c **do**
    $direction[c] \leftarrow direction[c]$
      $+ (level[c'] - level[c]) \cdot (index(c') - index(c))$
  end for
  normalize $direction[c]$
end for

for all triangles $t \in T$ **do** ▷ compute normal of t
  $d \leftarrow (0, 0, 0)^T$
  for all cells $c \in C$ with $t \cap c \neq \emptyset$ **do**
    $d \leftarrow d + direction[c]$
  end for
  if $d \cdot$ Normal $(t) < 0$ **then**
    reverse orientation of t
  end if
end for

3.4 Space Partition

In our application we are often faced with the following problem: Given a box in three-dimensional space, decide whether this box can be placed without intersecting the boundary of the trunk or other already placed boxes. This is a very important and fundamental test and its runtime strongly influences the overall runtime of the discretization process. We cannot afford to consider all triangles of the boundary or all already placed boxes. Hence we use a space partition to reduce the computational effort.

Sophisticated approaches for space partitions are known in the literature, e.g., hierarchical space partitions like kd-trees [BKOS00]. As we shall see in Chapter 4, the vast majority of queries involves objects closely located to the boundary of the trunk. Correspondingly, it is often necessary to consider the leaves in the kd-tree. Hence we do not expect faster query processing times by using a kd-tree. On the other side, a kd-tree should be more space-efficient than our simple approach.

Our space partition uses a uniform, axis-aligned, cubic grid $\mathcal{G}$ that partitions the space into cells. The spacing of the grid is based on experiments. This data structure provides two major functions: insertion of new objects and query of the stored information. In case of moving objects it is also possible to support efficient updates, but this functionality is not needed in our application. In our setting, the objects that are stored in the space partition are the triangles of the trunk. We also use a second space partition for already placed boxes.

For each of its cells $c \in \mathcal{G}$ the space partition maintains a list l_c of objects that intersect this cell. We maintain the invariant

$$\forall o \in O \; \forall c \in \mathcal{G} : o \cap c \neq \emptyset \Leftrightarrow o \in l_c, \tag{3.1}$$

where O denotes the set of all objects. In other words, if the list l_c for grid cell c does not contain an object $o \in O$, then $o \cap c = \emptyset$ holds.

The insertion of new objects into the space partition is described in Algorithm 3.2. The algorithm takes an object $o \in O$ as input and updates the data structures l_c of the space partition accordingly such that the invariant (3.1) is maintained.

The query process for an object o' is described in Algorithm 3.3. Note that $o' \in O$ is not mandatory. The algorithm takes the query object o' as input and returns a list l of objects that potentially intersect o'. Moreover, it follows from (3.1) that

$$\forall o \in O : o \notin l \Rightarrow o' \cap o = \emptyset. \tag{3.2}$$

In other words, objects that are not present in the computed list l have empty intersection with the query object o' and do not need to be taken into account.

Algorithm 3.2 Insertion of objects into the space partition

INSERT (o)
1: $B \leftarrow$ axis-aligned bounding box of object o
2: $C \leftarrow$ set of grid cells intersecting B
3: **for all** cells $c \in C$ **do**
4: **if** $o \cap c \neq \emptyset$ **then**
5: $l_c := l_c \cup \{o\}$
6: **end if**
7: **end for**

Algorithm 3.3 Querying the space partition

QUERY (o')
1: $B \leftarrow$ axis-aligned bounding box of object o'
2: $\tilde{C} \leftarrow$ set of grid cells intersecting B
3: **return** $\cup_{c \in \tilde{C}} l_c$

A variant of Algorithm 3.3 works as follows: Instead of first computing the bounding box B for a query object o' and then the set C of grid cells that intersect this bounding box B, one can also directly compute the set $\tilde{C}$ of grid cells that intersect the object o'. Thus in general the list of candidate objects for an intersection gets smaller. On the other side, the computation of $\tilde{C}$ is usually more expensive. The benefit of this modification depends on the setting.

The following basic operations are needed for both algorithms:

- computation of bounding boxes for stored objects and query objects
- deciding non-empty intersection for axis-aligned boxes
- deciding non-empty intersection for stored objects and axis-aligned boxes
- deciding non-empty intersection for query objects and axis-aligned boxes

The last operation is only needed in the above presented variation of Algorithm 3.3. The realization of these basic operations is discussed in Section 4.2.

Chapter 4

Discretization

Prior to the begin of this work, our customer constructed all packings manually. All solutions that they have provided have a certain structure in common. Examples for such packings can be seen in Figure 4.1.

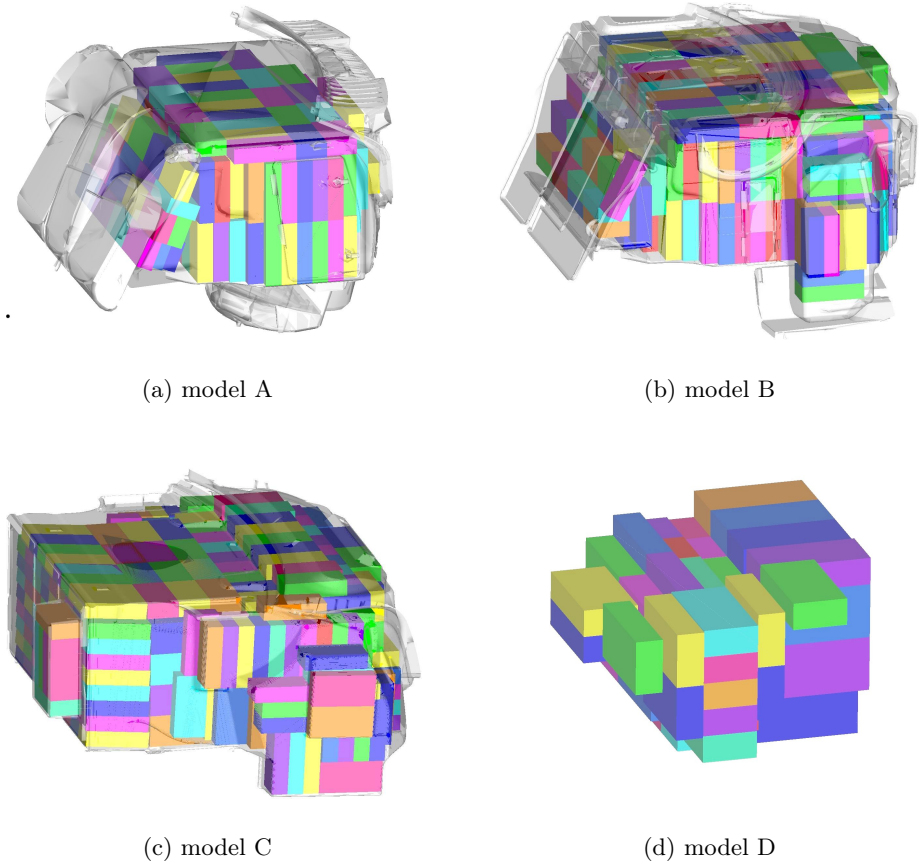
One observation is that the axes of almost all boxes are aligned to the axes of some coordinate system. Usually, less than 1% of the boxes have an independent orientation. These boxes are found at the boundary of the packing and their orientations are caused by the local geometry of the trunk.

Moreover, many boxes are aligned with each other and are not displaced arbitrarily. The extensions of the boxes imply side length ratios of $4 : 2 : 1$. Therefore it is possible to align the boxes without ruling out a tight packing, even if some of the boxes have been rotated by 90 degrees around one of their axes. Again, a small number of boxes at the boundary of the packing are displaced with respect to the core. Often their position is directly influenced by the nearby boundary of the trunk.

These observations motivate a discretization approach that restricts the solution space to boxes that are placed in a grid-like fashion. Such a restriction rules out arbitrary positions and orientations. Because such situations are rare, there is reason to believe that the restriction of the solution space imposed by the discretization is not too severe.

Our discretization approach creates an approximation of the space, and in particular of the region to be packed. The quality of the approximation depends on the geometric parameters of the grid, namely its origin, orientation and spacing. The proper choice of the spacing is crucial to be able to allow tight packings. A smaller spacing leads to a finer approximation of the region to be packed. On the other hand, a smaller spacing also increases the complexity of the resulting discrete packing problem.

The discretization allows us to classify regions of the space. Naturally, we are interested in distinguishing the region to be packed from its surrounding. We want to classify each cell of the grid with respect to the geometry of the



(a) model A (b) model B

(c) model C (d) model D

Figure 4.1: Manually constructed packings. These packings have a very regular structure. The majority of the boxes are aligned with the axes of a common coordinate system. Only few boxes have an independent orientation.[1]

trunk. It is our goal to identify the cells that are entirely, respectively in parts, covered by the region to be packed.

In Chapter 3 we explained that the input data does not precisely define the region to be packed. The introduction of a new data structure for the discrete problem allows us to restrict the effect of those problems to the construction phase of the grid. The packing algorithms can work on a well-defined problem instance and do not need to cope with the problems caused by the deficiencies in the boundary description of the trunk.

This chapter is organized as follows: In the first section we concentrate on the theoretical foundation of the discretization step. We describe the fundamental routines that are used for intersection tests in Section 4.2. These

[1]For legal reasons we may not publish the geometry of model D.

routines are needed to detect grid cells that intersect the boundary of the trunk or given boxes. In Section 4.3 we explain the classification process in detail. Section 4.4 deals with transformations of grids reusing existing information. We discuss the problem of optimal grids in Section 4.5.

4.1 Theoretical Aspects

In this section we consider the theoretical foundation of our discretization step. Actually, the discretization consists of two parts. First, we discretize the space and obtain an inner approximation of the region to be packed. We formally define the cubic grid that is used for this approximation. Second, we restrict the set of potential box placements. Finally, we revisit the notion of the conflict graph and the maximum stable set problem.

4.1.1 Discretization of the Space

We discretize the space with a uniform cubic grid. The geometry of the grid is described by three parameters: the origin $\boldsymbol{o} \in \mathbb{R}^3$, the orientation (denoted by a rotation matrix $\mathbf{R} \in \mathbb{R}^{3\times 3}$) and the spacing $l > 0$. We use integer triples to identify the grid cells. Consider the function

$$f\colon \mathbb{R}^3 \to \mathbb{R}^3,\ (i,j,k) \mapsto \boldsymbol{o} + l \cdot \mathbf{R} \cdot (i,j,k)^{\mathrm{T}}. \tag{4.1}$$

The function f maps the elements of $\mathbb{Z}^3$ (called *cell indices*) to the vertices of the grid with origin $\boldsymbol{o}$, orientation $\mathbf{R}$ and spacing l. This mapping leads to the definition of the grid cells as follows:

$$\mathit{cell}\colon \mathbb{Z}^3 \ni (i,j,k) \mapsto \bigcup_{\boldsymbol{x}\in[0,1)^3} f\,(\,(i,j,k) + \boldsymbol{x}\,) \subseteq \mathbb{R}^3. \tag{4.2}$$

The function *cell* maps integer triples to oriented cubes with side length l. Note that a cell is a continuous image of a cartesian product of three half open intervals, and thus it is a half-open cube. Conversely, with each point $\boldsymbol{p} \in \mathbb{R}^3$ we can associate its cell index:

$$\mathit{index}\colon \mathbb{R}^3 \to \mathbb{Z}^3,\ \boldsymbol{p} \mapsto \lfloor f^{-1}(\boldsymbol{p}) \rfloor, \tag{4.3}$$

where the operator $\lfloor \cdot \rfloor$ is to be applied componentwise.

Notice that the representation of a grid by the parameters $\boldsymbol{o}$, $\mathbf{R}$ and l is unique in the sense that there are no two distinct parameter triples that denote the same function f (and consequently, *cell* and *index*). But in the end, we are interested in the subdivision of the space into cells, and not in the index that is actually assigned to a given cell; the cell indices are merely a means of addressing the cells. In this context, the subdivisions of the space resulting from such grids are no longer uniquely represented by the

parameters $\boldsymbol{o}$, $\mathbf{R}$ and l. For example, a translation of the origin $\boldsymbol{o}$ by any element of $l \cdot R \cdot \mathbb{Z}^3$ results in the same subdivision of the space. We defer the discussion how to choose those parameters to Section 4.5.

We classify each cell with respect to the geometry of the trunk and boxes of a given packing. For now, we restrict ourselves to two states: *usable* and *unusable*[2]. A grid cell is called *usable* if it lies completely in the region to be packed and does not intersect boxes of a given (partial) packing. Otherwise, the cell is called *unusable*. The latter state includes cells that lie in the exterior of the trunk, cells that lie outside the region to be packed or cells that intersect given boxes or the boundary of the trunk. The set of usable cells corresponds to the subset I in the definition of the DISCRETE-BOX-PACKING problem (see Definition 2.3).

Thus the set of usable cells is an inner approximation of the region to be packed. In other words, the space occupied by usable cells is entirely available for packing with boxes. A packing that covers only usable cells does never interfere with the boundary or given boxes and is always feasible. Figure 4.2 shows the set of usable grid cells for four grids with different spacing.

4.1.2 Discretization of Box Placements

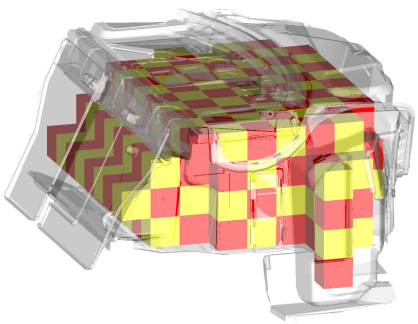
We described the discretization of the space in the previous subsection. Now we restrict the solution space of our problem even more by discretizing the possible box placements.

The boxes to be packed have a length, width, and height of 200mm, 100mm, and 50mm. We restrict ourselves to grids with spacing l such that $n \cdot l = 50\text{mm}$ for some integer n. Thus a box has the same shape as the union of $4n \times 2n \times n$ grid cells. The above condition for the choice of l is necessary in order to obtain tight packings.

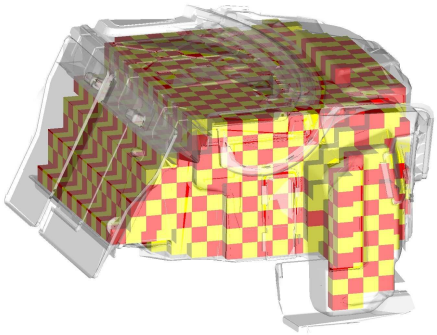
Furthermore, we restrict the position and orientation of boxes as follows: We demand that the axes of a box coincide with the axes of the grid. This implies that there are six different orientations for each box. Furthermore, we enforce that a box is aligned with the grid cells. That means that a box does not only have the same shape as, but also coincides with a set of $4n \times 2n \times n$ grid cells.

Thus the position and orientation of a box is defined by six parameters (x, y, z, w, h, d). The triple $(x, y, z) \in \mathbb{Z}^3$ denotes the so-called *anchor cell*. This cell has the (componentwise) smallest index among all cells covered by that box. The triple (w, h, d) denotes the orientation of the box and specifies its extension in width, height, and depth (with respect to the axes of the grid, i.e., the columns of $\mathbf{R}$, and measured in grid cells). Any permutation of the set $\{4n, 2n, n\}$ is valid for (w, h, d). The box defined by

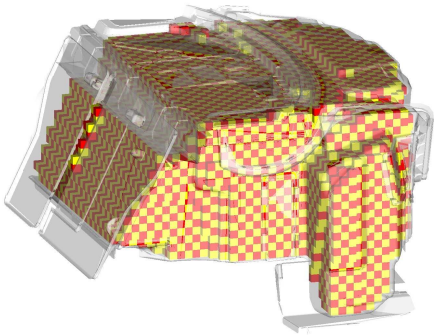
[2]We shall introduce a finer grained classification in Section 4.3.



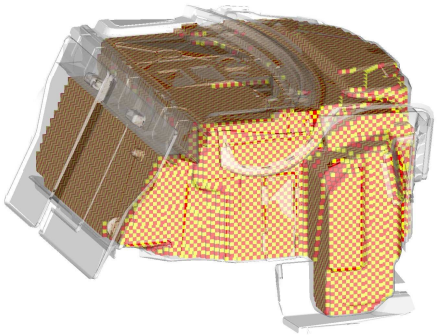
(a) spacing: 100mm



(b) spacing: 50mm



(c) spacing: 25mm



(d) spacing: 12.5mm

Figure 4.2: Discretization of the space with grids of different spacing. Only usable grid cells are shown.

the parameters (x, y, z, w, h, d) consists exactly of the grid cells with index in the set

$$[x, x+w) \times [y, y+h) \times [z, z+d) \quad \cap \quad \mathbb{Z}^3.$$

4.1.3 Formulation as a Stable Set Problem

It is straightforward to formulate the Discrete-Box-Packing problem as a stable set problem. We repeat the definition of the conflict graph (see Definition 2.8), now adapted to the terminology of this chapter.

Definition 4.1 (Conflict Graph). *Let I denote the set of usable cells of a grid $\mathcal{G}$ with parameters $(\boldsymbol{o}, \mathbf{R}, l)$. The* conflict graph $G = (V, E)$ *of $(\mathcal{G}, n, I)$ is defined as follows: There is a node $v_{x,y,z,w,h,d} \in V$ iff the box with anchor cell (x, y, z) and orientation (w, h, d) covers only usable cells, i.e., cells in I. Two nodes $v_i, v_j \in V$ are adjacent iff the corresponding boxes i and j intersect.*

n	spacing [mm]	#cells / box	max. size of cliques in G	maximal degree of G
1	50.00	8	48	201
2	25.00	64	384	2609
4	12.50	512	3072	25737
8	6.25	4096	24576	227225
n	$50/n$	$8n^3$	$48n^3$	$488n^3 - 364n^2 + 84n - 7$

Table 4.1: Characteristic numbers for conflict graphs

We have seen in Proposition 2.9 that any DISCRETE-BOX-PACKING problem can be reduced to a stable set problem for the corresponding conflict graph. Note that the conflict graph G has some regular structure that originates from the grid $\mathcal{G}$. However, it is difficult to exploit this structure if only the graph itself is given. On the other hand, this structure is easily accessible in the original grid $\mathcal{G}$.

Table 4.1 gives an impression about the size and complexity of the conflict graphs, depending on the parameter n. The number of cells per box equals $8n^3$. For example, a grid with a spacing of 25mm for a trunk of a typical size of 400 liters consists of approximately 25600 usable cells (depending on the orientation and translation of the grid). Since there are six different orientations, the maximum size of cliques in the conflict graph is six times the number of cells per box, i.e., $48n^3$. The maximum degree of the conflict graph is almost a magnitude larger. It can be determined by a simple enumeration of possible configurations of box pairs. The given function $488n^3 - 364n^2 + 84n - 7$ has been experimentally verified for powers of two up to $n = 1024$. For typical instances more characteristic values of the conflict graph are given in Table 6.2.

4.2 Fundamental Intersection Routines

In this section we describe the fundamental routines used in all intersection computations. These routines are used to decide whether two geometric primitives have a non-empty intersection or not.

We use these routines in the construction of the cubic grid. We need to identify all cells that intersect either the boundary of the trunk or boxes of given (partial) packings. The boundary of the trunk is given as a set of triangles. The given packings consist of a set of boxes. Therefore, we need to decide the intersection problem for the object pairs *triangle–cube* and *box–cube*.

We are also interested in the more general cases *triangle–box* and *box–box*, which are useful for verification of solutions obtained by other means. This includes solutions of other (non-grid-based) algorithms as well as solutions that are imported from a CAD system.

In this section we describe the implementation of the general cases *triangle–box* and *box–box*. The simplification for the cases *triangle–cube* and *box–cube* is straightforward.

Our intersection tests are based on a concept called *Separating Axis Theorem*. This concepts works as follows: Both objects in question are projected onto a fixed axis. Assume there is an axis such that the projections of both objects are disjoint. This implies the existence of a orthogonal plane that separates both objects. Such an axis is called a *separating axis* and it is a witness that both objects are disjoint.

This observation leads to some questions: Is there an axis such that the projection intervals are disjoint? If it exists, how does one find it? And if there is no such axis, how can one proof that? The *Separating Axis Theorem* stated in [GLM96a, GLM96b] answers these questions for the case of two oriented boxes.

Theorem 4.2 (Separating Axis Theorem). *If two oriented boxes are disjoint, then there exists a separating axis* $\boldsymbol{l} = \boldsymbol{a} \times \boldsymbol{b}$*, where* $\boldsymbol{a}$ *and* $\boldsymbol{b}$ *are taken from the six box axes.*

This theorem says that it is sufficient to look at a set of fifteen axes in total (three axes from one box, three axes from the other box, and nine axes resulting from pairwise cross products). Both boxes are disjoint iff one of those fifteen axes is a separating axis. A similar statement is valid for the case of a triangle and a box. Both cases are discussed in [Ebe01]. Our implementation is based on the routines described in this book.

4.2.1 Intersection Test for a Triangle and a Box

Let the triangle have vertices $\boldsymbol{u_0}$, $\boldsymbol{u_1}$ and $\boldsymbol{u_2} \in \mathbb{R}^3$. Define the edges of the triangle as $\boldsymbol{e_0} = \boldsymbol{u_1} - \boldsymbol{u_0}$, $\boldsymbol{e_1} = \boldsymbol{u_2} - \boldsymbol{u_0}$ and $\boldsymbol{e_2} = \boldsymbol{u_2} - \boldsymbol{u_1}$. The triangle is described by the set

$$\Big\{ \boldsymbol{u_0} + \lambda \boldsymbol{e_0} + \mu \boldsymbol{e_1} \,\Big|\, 0 \le \lambda \le 1,\, 0 \le \mu \le 1,\, \lambda + \mu \le 1 \Big\}.$$

Let the box have center $\boldsymbol{c} \in \mathbb{R}^3$, normalized axes $\boldsymbol{a_0}$, $\boldsymbol{a_1}$, $\boldsymbol{a_2} \in \mathbb{R}^3$, and side lengths $2a_0$, $2a_1$, $2a_2 > 0$. Thus the box is given by the set

$$\Big\{ \boldsymbol{c} + \sum_{i=0,1,2} \lambda_i \boldsymbol{a_i} \,\Big|\, -a_i \le \lambda_i \le a_i \Big\}.$$

Furthermore, define $\boldsymbol{d} = \boldsymbol{u_0} - \boldsymbol{c}$ and let $\boldsymbol{n} \in \mathbb{R}^3$ be an oriented normal of the triangle, for example $\boldsymbol{n} = \boldsymbol{e_0} \times \boldsymbol{e_1}$. The normal $\boldsymbol{n}$ does not need to have unit length.

The intersection test for a triangle and a box is described in Algorithm 4.1. The algorithm computes the quantities p_0, p_1, p_2 and r for each

Algorithm 4.1 Intersection test for a triangle and a box

IntersectionTriangleBox (t, b)
1: **for all** potential separating axes $\boldsymbol{l}$ **do**
2: compute quantities p_0, p_1, p_2 and r
3: **if** $\min\{p_0, p_1, p_2\} > r$ **or** $\max\{p_0, p_1, p_2\} < -r$ **then**
4: **return** false ▷ $\boldsymbol{l}$ is a separating axis, no intersection
5: **end if**
6: **end for**
7: **return** true ▷ no separating axis found, intersection

potential separating axis $\boldsymbol{l}$ (we identify an axis $\boldsymbol{c} + \lambda\boldsymbol{l}, \lambda \in \mathbb{R}$ with its directional vector $\boldsymbol{l}$). The values p_i, $0 \leq i \leq 2$ correspond to the projection of the triangle vertices $\boldsymbol{u_i}$, $0 \leq i \leq 2$ onto the axis $\boldsymbol{l}$ (see Figure 4.3). The value r describes the width of the projection interval of the box. The computation of those four quantities is summarized in Table 4.2. If $\min\{p_0, p_1, p_2\} > r$ or $\max\{p_0, p_1, p_2\} < -r$ holds, the projections of the box and the triangle are separated, $\boldsymbol{l}$ is a separating axis for both objects and the algorithm immediately returns false. Otherwise, $\boldsymbol{l}$ is not a separating axis. If there is no separating axis, both objects intersect and the algorithm returns true.

Under certain conditions, the separating axis test for a triangle and box can be accelerated. The tests described in lines 2 to 4 in Table 4.2 check whether the box intersects the bounding box of the triangle (the orientation of the bounding box coincides with that of the given box). Such a test might have already been performed in advance in order to reduce the number of

	$\boldsymbol{l}$	p_0	p_1	p_2	r
1	$\boldsymbol{n}$	$\boldsymbol{n}^{\mathrm{T}}\boldsymbol{d}$	p_0	p_0	$\sum_{i=0,1,2} a_i \lvert\boldsymbol{n}^{\mathrm{T}}\boldsymbol{a_i}\rvert$
2	$\boldsymbol{a_0}$	$\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{d}$	$p_0 + \boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{e_0}$	$p_0 + \boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{e_1}$	a_0
3	$\boldsymbol{a_1}$	$\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{d}$	$p_0 + \boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{e_0}$	$p_0 + \boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{e_1}$	a_1
4	$\boldsymbol{a_2}$	$\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{d}$	$p_0 + \boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{e_0}$	$p_0 + \boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{e_1}$	a_1
5	$\boldsymbol{a_0} \times \boldsymbol{e_0}$	$(\boldsymbol{a_0} \times \boldsymbol{e_0})^{\mathrm{T}}\boldsymbol{d}$	p_0	$p_0 + \boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{n}$	$a_1 \lvert\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{e_0}\rvert + a_2 \lvert\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{e_0}\rvert$
6	$\boldsymbol{a_0} \times \boldsymbol{e_1}$	$(\boldsymbol{a_0} \times \boldsymbol{e_1})^{\mathrm{T}}\boldsymbol{d}$	$p_0 - \boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{n}$	p_0	$a_1 \lvert\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{e_1}\rvert + a_2 \lvert\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{e_1}\rvert$
7	$\boldsymbol{a_0} \times \boldsymbol{e_2}$	$(\boldsymbol{a_0} \times \boldsymbol{e_2})^{\mathrm{T}}\boldsymbol{d}$	$p_0 - \boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{n}$	p_1	$a_1 \lvert\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{e_2}\rvert + a_2 \lvert\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{e_2}\rvert$
8	$\boldsymbol{a_1} \times \boldsymbol{e_0}$	$(\boldsymbol{a_1} \times \boldsymbol{e_0})^{\mathrm{T}}\boldsymbol{d}$	p_0	$p_0 + \boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{n}$	$a_0 \lvert\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{e_0}\rvert + a_2 \lvert\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{e_0}\rvert$
9	$\boldsymbol{a_1} \times \boldsymbol{e_1}$	$(\boldsymbol{a_1} \times \boldsymbol{e_1})^{\mathrm{T}}\boldsymbol{d}$	$p_0 - \boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{n}$	p_0	$a_0 \lvert\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{e_1}\rvert + a_2 \lvert\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{e_1}\rvert$
10	$\boldsymbol{a_1} \times \boldsymbol{e_2}$	$(\boldsymbol{a_1} \times \boldsymbol{e_2})^{\mathrm{T}}\boldsymbol{d}$	$p_0 - \boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{n}$	p_1	$a_0 \lvert\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{e_2}\rvert + a_2 \lvert\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{e_2}\rvert$
11	$\boldsymbol{a_2} \times \boldsymbol{e_0}$	$(\boldsymbol{a_2} \times \boldsymbol{e_0})^{\mathrm{T}}\boldsymbol{d}$	p_0	$p_0 + \boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{n}$	$a_0 \lvert\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{e_0}\rvert + a_1 \lvert\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{e_0}\rvert$
12	$\boldsymbol{a_2} \times \boldsymbol{e_1}$	$(\boldsymbol{a_2} \times \boldsymbol{e_1})^{\mathrm{T}}\boldsymbol{d}$	$p_0 - \boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{n}$	p_0	$a_0 \lvert\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{e_1}\rvert + a_1 \lvert\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{e_1}\rvert$
13	$\boldsymbol{a_2} \times \boldsymbol{e_2}$	$(\boldsymbol{a_2} \times \boldsymbol{e_2})^{\mathrm{T}}\boldsymbol{d}$	$p_0 - \boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{n}$	p_1	$a_0 \lvert\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{e_2}\rvert + a_1 \lvert\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{e_2}\rvert$

Table 4.2: Values for the separating axis test for a triangle and a box

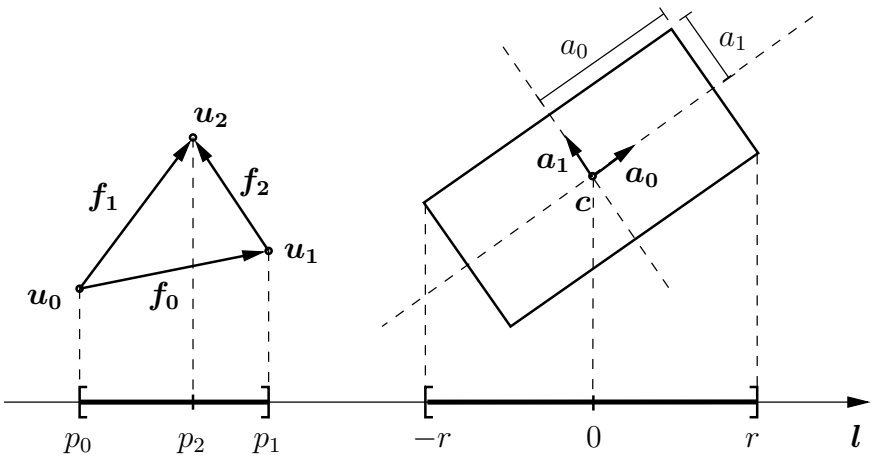


Figure 4.3: Separating axis test for a triangle and a box. The axis $\boldsymbol{l}$ is a separating axis if the interval spanned by p_0, p_1, p_2 and the interval $[-r, r]$ are disjoint.

candidate pairs for the intersection test. In this case, the corresponding tests can be skipped here. Thus the maximum number of axes that need to be tested can be reduced from 13 to 10.

4.2.2 Intersection Test for Two Boxes

Let the first box have center $\boldsymbol{c_0} \in \mathbb{R}^3$, normalized axes $\boldsymbol{a_0}$, $\boldsymbol{a_1}$, $\boldsymbol{a_2} \in \mathbb{R}^3$, and side lengths $2a_0$, $2a_1$, $2a_2 > 0$. Similarly, let the second box have center $\boldsymbol{c_1} \in \mathbb{R}^3$, normalized axes $\boldsymbol{b_0}$, $\boldsymbol{b_1}$, $\boldsymbol{b_2} \in \mathbb{R}^3$, and side lengths $2b_0$, $2b_1$, $2b_2 > 0$. Thus the boxes are given by the sets

$$\Big\{\boldsymbol{c_0} + \sum_{i=0,1,2} \lambda_i \boldsymbol{a_i} \;\Big|\; -a_i \le \lambda_i \le a_i\Big\} \text{ and } \Big\{\boldsymbol{c_1} + \sum_{i=0,1,2} \mu_i \boldsymbol{b_i} \;\Big|\; -b_i \le \mu_i \le b_i\Big\}.$$

Let $\boldsymbol{d} = \boldsymbol{c_1} - \boldsymbol{c_0}$ and $\mathbf{C} = \mathbf{A}^{\mathrm{T}}\mathbf{B}$, where $\mathbf{A} = (\boldsymbol{a_0}, \boldsymbol{a_1}, \boldsymbol{a_2})$ and $\mathbf{B} = (\boldsymbol{b_0}, \boldsymbol{b_1}, \boldsymbol{b_2})$. The matrix $\mathbf{C}$ denotes the orientation of the second box relative to the orientation of the first box.

The intersection test for two boxes is described in Algorithm 4.2. The algorithm computes the quantities r_0, r_1 and r for each potential separating axis $\boldsymbol{l}$. The values r_0 and r_1 correspond to the projection intervals of both boxes (see Figure 4.4), r describes the projection of the center of the second box. The computation of those three quantities is summarized in Table 4.3. If $r > r_0 + r_1$ holds, the projections of both boxes are separated, $\boldsymbol{l}$ is a separating axis for both objects and the algorithm immediately returns false. If there is no separating axis, both objects intersect and the algorithm returns true.

Algorithm 4.2 Intersection test for two boxes

IntersectionBoxBox (b_1, b_2)
1: **for all** potential separating axes $\boldsymbol{l}$ **do**
2: compute quantities r_0, r_1 and r
3: **if** $r > r_0 + r_1$ **then**
4: **return** false ▷ $\boldsymbol{l}$ is a separating axis, no intersection
5: **end if**
6: **end for**
7: **return** true ▷ no separating axis found, intersection

Similar to the case of a triangle and a box, the separating axis test for two boxes can be accelerated under certain conditions. The tests described in lines 1 to 6 in Table 4.3 check whether the first box intersects the bounding box of the second box and vice versa (the orientation of a bounding box coincides with the orientation of the other given box). Such a test might have already been performed before in advance in order to reduce the number of candidate pairs for the intersection test. In this case, the corresponding tests can be skipped here. Thus the maximum number of axes that need to be tested can be reduced from 15 to 12 or 9.

4.3 Generation of Grids

Suppose we are given the geometry of the trunk and the geometric parameters of the desired grid, that is, its origin $\boldsymbol{o}$, its orientation $\mathbf{R}$ and the spac-

	$\boldsymbol{l}$	r_0	r_1	r
1	$\boldsymbol{a_0}$	a_0	$b_0\,\lvert c_{00}\rvert + b_1\,\lvert c_{01}\rvert + b_2\,\lvert c_{02}\rvert$	$\lvert\,\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
2	$\boldsymbol{a_1}$	a_1	$b_0\,\lvert c_{10}\rvert + b_1\,\lvert c_{11}\rvert + b_2\,\lvert c_{12}\rvert$	$\lvert\,\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
3	$\boldsymbol{a_2}$	a_2	$b_0\,\lvert c_{20}\rvert + b_1\,\lvert c_{21}\rvert + b_2\,\lvert c_{22}\rvert$	$\lvert\,\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
4	$\boldsymbol{b_0}$	$a_0\,\lvert c_{00}\rvert + a_1\,\lvert c_{10}\rvert + a_2\,\lvert c_{20}\rvert$	b_0	$\lvert\,\boldsymbol{b_0}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
5	$\boldsymbol{b_1}$	$a_0\,\lvert c_{01}\rvert + a_1\,\lvert c_{11}\rvert + a_2\,\lvert c_{21}\rvert$	b_1	$\lvert\,\boldsymbol{b_1}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
6	$\boldsymbol{b_2}$	$a_0\,\lvert c_{02}\rvert + a_1\,\lvert c_{12}\rvert + a_2\,\lvert c_{22}\rvert$	b_2	$\lvert\,\boldsymbol{b_2}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
7	$\boldsymbol{a_0} \times \boldsymbol{b_0}$	$a_1\,\lvert c_{20}\rvert + a_2\,\lvert c_{10}\rvert$	$b_1\,\lvert c_{02}\rvert + b_2\,\lvert c_{01}\rvert$	$\lvert\, c_{10}\,\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{d} - c_{20}\,\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
8	$\boldsymbol{a_0} \times \boldsymbol{b_1}$	$a_1\,\lvert c_{21}\rvert + a_2\,\lvert c_{11}\rvert$	$b_0\,\lvert c_{02}\rvert + b_2\,\lvert c_{00}\rvert$	$\lvert\, c_{11}\,\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{d} - c_{21}\,\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
9	$\boldsymbol{a_0} \times \boldsymbol{b_2}$	$a_1\,\lvert c_{22}\rvert + a_2\,\lvert c_{12}\rvert$	$b_0\,\lvert c_{01}\rvert + b_1\,\lvert c_{00}\rvert$	$\lvert\, c_{12}\,\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{d} - c_{22}\,\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
10	$\boldsymbol{a_1} \times \boldsymbol{b_0}$	$a_0\,\lvert c_{20}\rvert + a_2\,\lvert c_{00}\rvert$	$b_1\,\lvert c_{12}\rvert + b_2\,\lvert c_{11}\rvert$	$\lvert\, c_{20}\,\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{d} - c_{00}\,\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
11	$\boldsymbol{a_1} \times \boldsymbol{b_1}$	$a_0\,\lvert c_{21}\rvert + a_2\,\lvert c_{01}\rvert$	$b_0\,\lvert c_{12}\rvert + b_2\,\lvert c_{10}\rvert$	$\lvert\, c_{21}\,\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{d} - c_{01}\,\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
12	$\boldsymbol{a_1} \times \boldsymbol{b_2}$	$a_0\,\lvert c_{22}\rvert + a_2\,\lvert c_{02}\rvert$	$b_0\,\lvert c_{11}\rvert + b_1\,\lvert c_{10}\rvert$	$\lvert\, c_{22}\,\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{d} - c_{02}\,\boldsymbol{a_2}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
13	$\boldsymbol{a_2} \times \boldsymbol{b_0}$	$a_0\,\lvert c_{10}\rvert + a_1\,\lvert c_{00}\rvert$	$b_1\,\lvert c_{22}\rvert + b_2\,\lvert c_{21}\rvert$	$\lvert\, c_{00}\,\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{d} - c_{10}\,\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
14	$\boldsymbol{a_2} \times \boldsymbol{b_1}$	$a_0\,\lvert c_{11}\rvert + a_1\,\lvert c_{01}\rvert$	$b_0\,\lvert c_{22}\rvert + b_2\,\lvert c_{20}\rvert$	$\lvert\, c_{01}\,\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{d} - c_{11}\,\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{d}\,\rvert$
15	$\boldsymbol{a_2} \times \boldsymbol{b_2}$	$a_0\,\lvert c_{12}\rvert + a_1\,\lvert c_{02}\rvert$	$b_0\,\lvert c_{21}\rvert + b_1\,\lvert c_{20}\rvert$	$\lvert\, c_{02}\,\boldsymbol{a_1}^{\mathrm{T}}\boldsymbol{d} - c_{12}\,\boldsymbol{a_0}^{\mathrm{T}}\boldsymbol{d}\,\rvert$

Table 4.3: Values for the separating axis test for two boxes

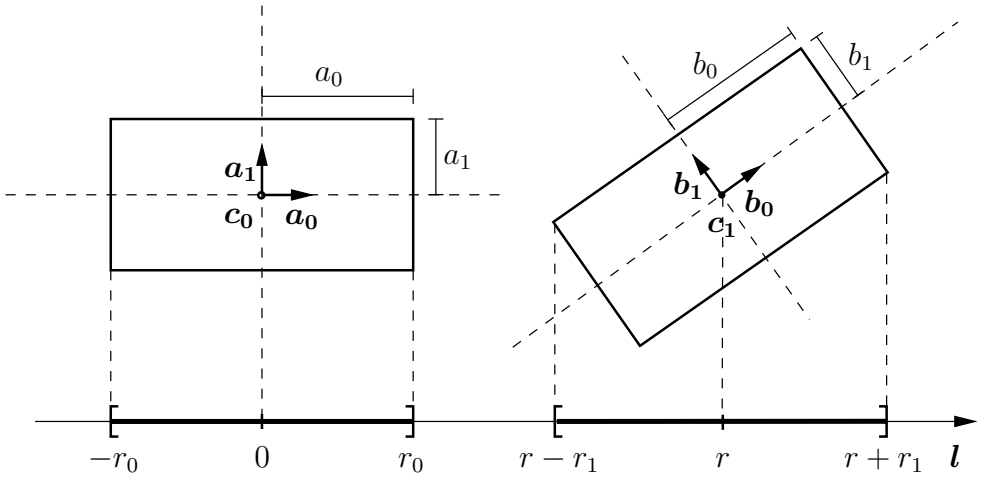


Figure 4.4: Separating axis test for two boxes. The axis $\boldsymbol{l}$ is a separating axis if the intervals $[-r_0, r_0]$ and $[r - r_1, r + r_1]$ are disjoint.

ing l. Optionally, a set of boxes that represents a partial packing might also be given. The goal is to identify all the cells that are completely contained in the region to be packed.

In order to accomplish this goal, we introduce the following classification for grid cells:

- OUTSIDE: The cell lies in the exterior of the trunk.
- BOUNDARY: The cell intersects the boundary of the trunk or intersects a given box.
- INSIDE: The cell lies in the interior of the trunk.
- UNKNOWN: The state of the cell is not yet determined.

We shall see in Section 4.3.2 that some INSIDE and UNKNOWN cells can never be covered by a box. These cells are important in the classification process. Therefore we extend the above set of states as follows:

- INSIDE*: The cell lies in the interior of the trunk and can never be covered by a box.
- UNKNOWN*: The state of the cell is not yet determined, but it can never be covered by a box.

In figures we use different colors to distinguish the various cell states (see Figure 4.5). Cells labeled as INSIDE* and UNKNOWN* are additionally marked by an X-shaped cross.

Initially, all cells of the grid are labeled as UNKNOWN. Our goal is to label all cells according to the classification above. In particular, we want to

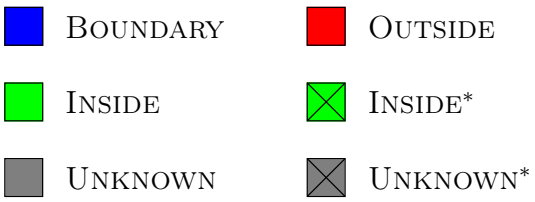


Figure 4.5: Visual representation of cell states

reduce the number of UNKNOWN cells as much as possible. All INSIDE cells constitute an inner approximation of the region to be packed. All other cells, except those labeled as UNKNOWN, can definitely not be used for packing.

The algorithms used for the classification are also used in a different context (see Section 4.4 and 4.5). In this context, the setting is as follows: Only a subset of the cells is labeled as UNKNOWN, the remaining cells are already properly classified. The goal of classification of all cells remains the same. Because only a subset of all cells is left to be classified, we added in the following algorithms some statements for the purpose of optimization. For simplicity, we replace all information about INSIDE* and UNKNOWN* cells by INSIDE and UNKNOWN, respectively.

The classification process consists of several steps that are presented in the following subsections.

4.3.1 Boundary Cells

In the first step of the classification process we identify all cells that should be labeled as BOUNDARY. We compute the set of cells that intersect at least one triangle of the trunk boundary. Similarly, we calculate the set of cells that intersect at least one box.

ZOMORODIAN and EDELSBRUNNER [ZE00] give an efficient algorithm to compute all intersection pairs of two sets of objects. First, the objects are enclosed in axis-aligned bounding boxes which reduces the problem to finding intersecting intervals. A hybrid algorithm involving range trees and scanning is used to generate the set of intersection candidates. Primitive intersection tests are needed to compute the list of actual intersection pairs.

In our setting, a much simpler approach suffices. We do not need to compute all intersection pairs, it suffices to decide for a given cell whether it is intersected by any triangle (or any box). Moreover, given the special structure of the set of cells, it is straightforward to compute the set of intersection candidates for a given triangle or box.

First, we explain how to compute the set of cells intersected by the boundary of the trunk. The pseudo code for our implementation is presented in

Algorithm 4.3 Identification of BOUNDARY cells (part 1)

```
BOUNDARYCELLSTRIANGLES (G, T)
 1: for all triangles t ∈ T do
 2:     B ← grid-aligned bounding box of t
 3:     for all cells c ∈ G that intersect B do
 4:         if c is labeled as UNKNOWN then                    ▷ speed-up
 5:             if INTERSECTIONTRIANGLEBOX (t, c) then
 6:                 label c as BOUNDARY
 7:             end if
 8:         end if
 9:     end for
10: end for
```

Algorithm 4.3. The algorithm takes a grid $\mathcal{G}$ and a set of triangles T as input. The outer loop iterates over all triangles $t \in T$. The inner loop does *not* iterate over all grid cells $c \in \mathcal{G}$. It is restricted to those cells that intersect the bounding box of t. Note that the set of cells satisfying this condition (see line 3) can be efficiently determined. This set has a cuboid shape and the indices of both extremal cells can be obtained by applying the *index* function of the grid (see function (4.3)) to both extremal vertices of the bounding box B. The intersection test in line 5 is skipped if a cell is not labeled as UNKNOWN.

A second implementation that reverses the order of both loops is given in Algorithm 4.4. The outer loop iterates over all UNKNOWN cells $c \in \mathcal{G}$. We compute an axis-aligned bounding box B of a cell c and use the precomputed space partition of the set of triangles T to obtain a set $S \subseteq T$ of intersection candidates. The inner loop iterates over this set S and is terminated as soon as an intersection is found.

The relative runtime of both variants depends on many parameters, e.g., on the number of the triangles and the cells, on the size and distribution of the triangles, on the spacing of the grid, and on the number of UNKNOWN cells. We performed various tests with different models and grids with different spacing. If no further information about the cell labels is given, i.e., all cells are initially labeled as UNKNOWN, the first implementation is up to factor two faster than the second one. However, often only the labels for cells close to the boundary need to be recomputed, e.g., during a translation of the grid (see Section 4.4). In such a situation, the runtime of both implementations is roughly the same.

The boxes of a given (partial) packing are handled in a similar way. The pseudo code for this case is shown in Algorithm 4.5, which is analog to Algorithm 4.3. Naturally, an implementation analog to Algorithm 4.4 is also possible. Because the runtime spent on the boxes B is significantly

Algorithm 4.4 Identification of BOUNDARY cells (part 1, alternative implementation)

```
BOUNDARYCELLSTRIANGLES (G, T)
 1: let SP denote the space partition of T
 2: for all cells c ∈ G do
 3:     if c is labeled as UNKNOWN then                    ▷ speed-up
 4:         B ← axis-aligned bounding box of c
 5:         S ← SP.QUERY(B)
 6:         for all triangles t ∈ S do
 7:             if INTERSECTIONTRIANGLEBOX (t, c) then
 8:                 label c as BOUNDARY
 9:                 break                                  ▷ speed-up
10:             end if
11:         end for
12:     end if
13: end for
```

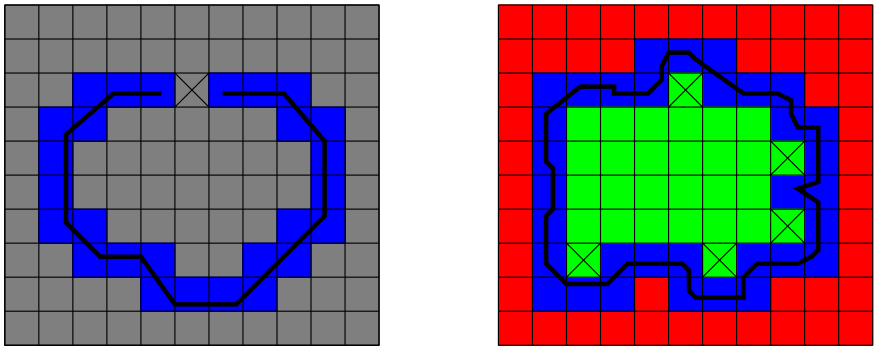
smaller than the time spent on the triangles T, a gain of an alternative implementation would not improve the overall runtime very much. Therefore we did not study this variant.

Algorithm 4.5 Identification of BOUNDARY cells (part 2)

```
BOUNDARYCELLSBOXES (G, B)
 1: for all boxes b ∈ B do
 2:     B ← grid-aligned bounding box of b
 3:     for all cells c ∈ G that intersect B do
 4:         if c is labeled as UNKNOWN then                ▷ speed-up
 5:             if INTERSECTIONBOXBOX (b, c) then
 6:                 label c as BOUNDARY
 7:             end if
 8:         end if
 9:     end for
10: end for
```

4.3.2 Unusable Cells

In the second step we identify the set of UNKNOWN* and INSIDE* cells. The purpose of UNKNOWN* cells is to ease the classification of INSIDE and OUTSIDE cells in the presence of small holes in the boundary description of the trunk (see Section 3.2). Consider the example depicted in Figure 4.6(a). Assume that we want to pack rectangles of size 4×2 cells. All cells that



(a) UNKNOWN* cells (b) INSIDE* cells

Figure 4.6: Examples of unusable cells. The cells that are marked by an X-shaped cross can never be covered by a rectangle of 4×2 cells.

intersect the boundary of the trunk are already labeled as BOUNDARY. But they do not separate the region to be packed from the exterior because there is a hole in the trunk boundary.

On closer examination, we recognize that the marked cell can never be covered by a rectangle. The nearby BOUNDARY cells rule out any position of a box covering this cell. Therefore we can classify it as UNKNOWN*. The combined set of BOUNDARY and UNKNOWN* cells separates the region to be packed and the exterior.

Cells labeled as UNKNOWN* are always in close proximity to BOUNDARY cells. In fact, both kinds of cells play a similar role: They belong to a set of cells that (hopefully) separates the region to be packed from the exterior. The difference is that BOUNDARY cells are intersected by the boundary of the trunk (or given boxes), while UNKNOWN* cells are not intersected by those objects.

The purpose of INSIDE* cells is quite different. Assume that some cells are already labeled as INSIDE. We shall later compare different grids with respect to their potential for a packing of high cardinality (see Section 4.5). A fast and easy way is to compare the number of INSIDE cells. However, this measure neglects that some INSIDE cells can never be covered by any box. An example with such cells is shown in Figure 4.6(b).

Therefore, we label such cells as INSIDE*. The number of remaining INSIDE cells gives a more precise measure of the cells that can be covered in a packing. The algorithm that identifies UNKNOWN* cells can be easily extended to detect also INSIDE* cells.

The algorithm used to determine UNKNOWN* and INSIDE* cells is depicted in Algorithm 4.6. For each cell $c \in \mathcal{G}$ we have a flag $covered[c]$ that is initialized with false. We iterate over all possible box positions and orientations and mark all cells that can be covered by a box. Finally we label all

Algorithm 4.6 Identification of unusable cells

```
UnusableCells (G)
 1: for all cells c ∈ G do
 2:     covered[c] ← false
 3: end for
 4:
 5: for all cells c ∈ G do
 6:     for all six orientations o do
 7:         let b denote a box with orientation o anchored at cell c
 8:         if b covers only cells labeled as Inside or Unknown then
 9:             for all cells c' covered by b do
10:                 covered[c'] ← true
11:             end for
12:         end if
13:     end for
14: end for
15:
16: for all cells c ∈ G do
17:     if covered[c] = false then
18:         if c is labeled as Inside then
19:             label c as Inside*
20:         end if
21:         if c is labeled as Unknown then
22:             label c as Unknown*
23:         end if
24:     end if
25: end for
```

Inside and Unknown cells that cannot be covered by any box as Inside* and Unknown*, respectively.

4.3.3 Inside and Outside Cells

In the last step we identify Inside and Outside cells. We have already labeled the sets of Boundary and unusable cells. Some of the cells might already be labeled as Inside or Outside. The goal is to classify the remaining Unknown (Unknown*) cells as either Inside (Inside*) or Outside.

First we label the outmost layer of cells as Outside (we assume that the grid is large enough, such that the cells of the outmost layer either are labeled as Unknown or are already labeled as Outside). Next we look at maximal connected components of Unknown and Unknown* cells. We examine the labels of all cells adjacent to such a component. Based on the set of these labels we classify the cells in the component. In the end, if no

Algorithm 4.7 Identification of INSIDE, INSIDE* and OUTSIDE cells

INSIDEANDOUTSIDECELLS ($\mathcal{G}$)
1: $I \leftarrow \{$INSIDE, INSIDE*$\}$
2: $O \leftarrow \{$OUTSIDE$\}$
3: $I' \leftarrow I \cup \{$UNKNOWN$\}$
4: $O' \leftarrow O \cup \{$UNKNOWN$\}$
5:
6: **for all** maximal connected components C of UNKNOWN cells **do**
7: $L \leftarrow$ set of labels of cells adjacent to C
8: **if** $L \cap I \neq \emptyset$ **and** $L \cap O = \emptyset$ **then**
9: label all cells of C as INSIDE
10: **end if**
11: **if** $L \cap O \neq \emptyset$ **and** $L \cap I = \emptyset$ **then**
12: label all cells of C as OUTSIDE
13: **end if**
14: **end for**
15:
16: **for all** maximal connected components C of UNKNOWN* cells **do**
17: $L \leftarrow$ set of labels of cells adjacent to C
18: **if** $L \cap I \neq \emptyset$ **and** $L \cap O' = \emptyset$ **then**
19: label all cells of C as INSIDE*
20: **end if**
21: **if** $L \cap O \neq \emptyset$ **and** $L \cap I' = \emptyset$ **then**
22: label all cells of C as OUTSIDE
23: **end if**
24: **end for**

cells are labeled as INSIDE and there is exactly one component of UNKNOWN cells left, we label those cells as INSIDE.

The central part of the algorithm is outlined in Algorithm 4.7. For each maximal connected component C of UNKNOWN cells we compute the set L of labels of adjacent cells. If the component is adjacent to an INSIDE or INSIDE* cell, and is not adjacent to OUTSIDE cells, we know that the component belongs to the interior and label all cells in the component as INSIDE. Similarly, if the component is adjacent to a OUTSIDE cell, and is not adjacent to INSIDE or INSIDE* cells, we assume that the whole component lies in the exterior and all cells in the component are labeled as OUTSIDE. We also look at maximal connected components of UNKNOWN* cells and classify them according to similar rules.

These conditions ensure that we do not end up with OUTSIDE cells adjacent to INSIDE (or INSIDE*) cells. Note that we do not distinguish between BOUNDARY and UNKNOWN* cells in the first loop of the algorithm. In the

second loop, we act conservatively and treat UNKNOWN cells as OUTSIDE if the component is adjacent to an INSIDE cell and vice versa.

For efficiency reasons, we do not explicitly compute the maximal connected components C. We rather directly compute the sets L of labels of adjacent cells.

There are three cases in which UNKNOWN cells remain or the classification of INSIDE, INSIDE* and OUTSIDE cells is incorrect:

- There are two or more components of UNKNOWN cells that are neither adjacent to OUTSIDE nor INSIDE (or INSIDE*) cells (see Figure 4.7(a)). We cannot decide whether these components belong to the interior or exterior. The same holds if there is only one such component and some cells elsewhere are already labeled as INSIDE.

- There is a large hole in the boundary of the trunk such that a component of UNKNOWN cells is adjacent to OUTSIDE as well as INSIDE (or INSIDE*) cells (see Figure 4.7(b)). We cannot decide which of these cells belong to the interior and exterior, respectively.

- There is a large hole in the boundary of the trunk such that a component of UNKNOWN cells is adjacent to OUTSIDE cells, but not to INSIDE (or INSIDE*) cells (see Figure 4.7(c)). The whole component is wrongly classified as OUTSIDE.

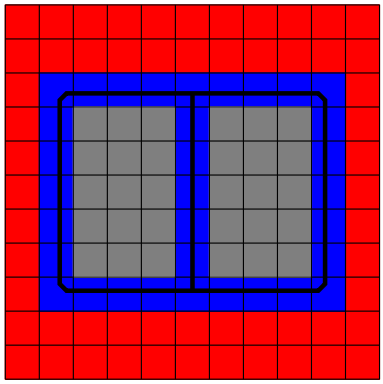
In the first case, we ask the user to classify the components as INSIDE or OUTSIDE. The problems in the second and third case originate from holes in the boundary description of the trunk. There are holes large enough such that a box can pass them. As we mentioned in Section 3.2, we only allow small holes in the input data. The user must close such large holes manually.

4.4 Transformation of Grids

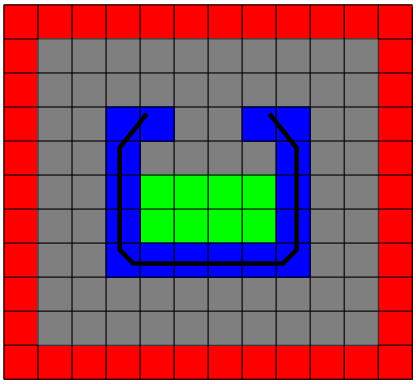
In this section we discuss efficient ways to update the cell labels after a transformation of the grid. Translations for instance are heavily used to find a good placement of the grid (see Section 4.5). To obtain a reasonable runtime, an efficient implementation of the translation process is necessary. A natural way to do this is not to recompute the cell lables from scratch, but rather to reuse the information that was valid just before the transformation.

We are given a grid $\mathcal{G}$ with origin $\boldsymbol{o}$, orientation $\mathbf{R}$ and spacing l. All cells of $\mathcal{G}$ are already classified. Suppose we want to classify the cells of a grid $\mathcal{G}'$ with origin $\boldsymbol{o}'$, orientation $\mathbf{R}'$ and spacing l'.

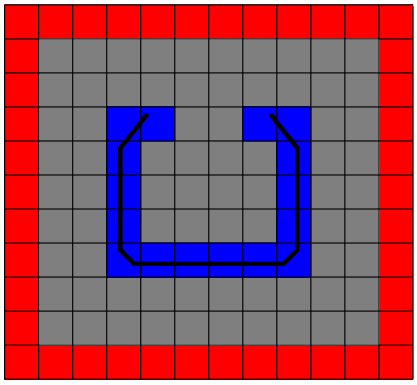
A change in exactly one of the three grid parameters can be associated with three classic kinds of transformation: scaling (l), translation ($\boldsymbol{o}$) and rotation ($\mathbf{R}$). In the remainder of this section we discuss the general problem of arbitrary transformations. Special cases of scaling and translation are



(a) Two or more connected components of UNKNOWN cells are neither adjacent to OUTSIDE or INSIDE (or INSIDE*) cells.



(b) Due to a large hole in the boundary a connected component of UNKNOWN cells is adjacent to OUTSIDE as well as INSIDE (or INSIDE*) cells.



(c) Due to a large hole in the boundary a connected component of UNKNOWN cells is adjacent to OUTSIDE cells, but not to INSIDE (or INSIDE*) cells.

Figure 4.7: Situations in which UNKNOWN cells cannot be labeled correctly.

discussed in more detail in Section 4.4.1 and 4.4.2, respectively. Rotations are hardly needed in our application. Typically, the orientation of the grid is fixed in the beginning, whereas origin and spacing are changed from time to time.

The algorithm for the general case is outlined in Algorithm 4.8. For each cell c' of the transformed grid $\mathcal{G}'$, we compute the set S of cells of the original grid $\mathcal{G}$ that intersect c'. The set L denotes the labels of all cells in S. There are two cases in which we can directly determine the correct label of the cell c'. In all other cases, c' is labeled as UNKNOWN.

Note that there is no rule that labels c' as BOUNDARY if $S = \{\text{BOUNDARY}\}$. (One might expect such a rule, given similar rules for INSIDE and OUT-

Algorithm 4.8 Computation of cell labels after transformation of $\mathcal{G}$ into $\mathcal{G}'$

TRANSFORMGRID($\mathcal{G}, \mathcal{G}'$)
1: **for all** cells $c' \in \mathcal{G}'$ **do**
2: $S \leftarrow \{c \in \mathcal{G} \mid c \cap c' \neq \emptyset\}$
3: $L \leftarrow$ set of all labels of cells in S
4: **if** $L \subseteq \{\text{INSIDE}, \text{INSIDE}^*\}$ **then**
5: label c' as INSIDE
6: **else if** $L = \{\text{OUTSIDE}\}$ **then**
7: label c' as OUTSIDE
8: **else**
9: label c' as UNKNOWN
10: **end if**
11: **end for**

SIDE cells.) Consider a BOUNDARY cell $c \in \mathcal{G}$ and a scaling operation that halves the spacing l of the original grid. The cell c is decomposed into eight smaller cells, but not necessarily all of them intersect the boundary of the trunk (or given boxes). We cannot classify the smaller cells without further computational effort. Therefore, for the time being such cells are labeled as UNKNOWN.

A similar effect can be observed for translation operations (see Figure 4.8). The left figure shows four adjacent cells that are intersected by the trunk boundary, and consequently, are labeled as BOUNDARY. The transformed grid $\mathcal{G}'$ in the right figure is obtained by a translation in horizontal direction by $\frac{l}{2}$. The upper cell in the center is completely covered by BOUNDARY cells of $\mathcal{G}$, but does not intersect the boundary.

Algorithm 4.8 returns a grid with cells labeled as INSIDE, OUTSIDE and UNKNOWN. The sets of INSIDE and OUTSIDE cells of $\mathcal{G}'$ approximate the corresponding sets of $\mathcal{G}$. In order to get rid of UNKNOWN cells, we rerun the algorithms presented in Section 4.3. Their runtime improves significantly compared to the case where all cells are labeled as UNKNOWN.

In particular, we are interested in transformations that keep the number of UNKNOWN cells as low as possible. The number of UNKNOWN cells is in direct relation to the size of the sets S: The larger the set S, the higher is the number of UNKNOWN cells.

4.4.1 Scaling

A scaling of a grid relates to a change of the spacing parameter l; the orientation $\mathbf{R}$ and the origin $\boldsymbol{o}$ remain unchanged. We denote the ratio of the old value l and the new value l' by $k := l/l' > 0$. We restrict ourselves in the following to $k \in \mathbb{N}$, i.e., all cells of the given grid are subdivided uniformly into k^3 cells. In practice, we only use the value $k = 2$.

(a) Original grid $\mathcal{G}$ (b) Translated grid $\mathcal{G}'$

Figure 4.8: State of cells covered by BOUNDARY cells. Cells that are entirely covered by BOUNDARY cells of a different grid do not necessarily intersect the boundary of the trunk.

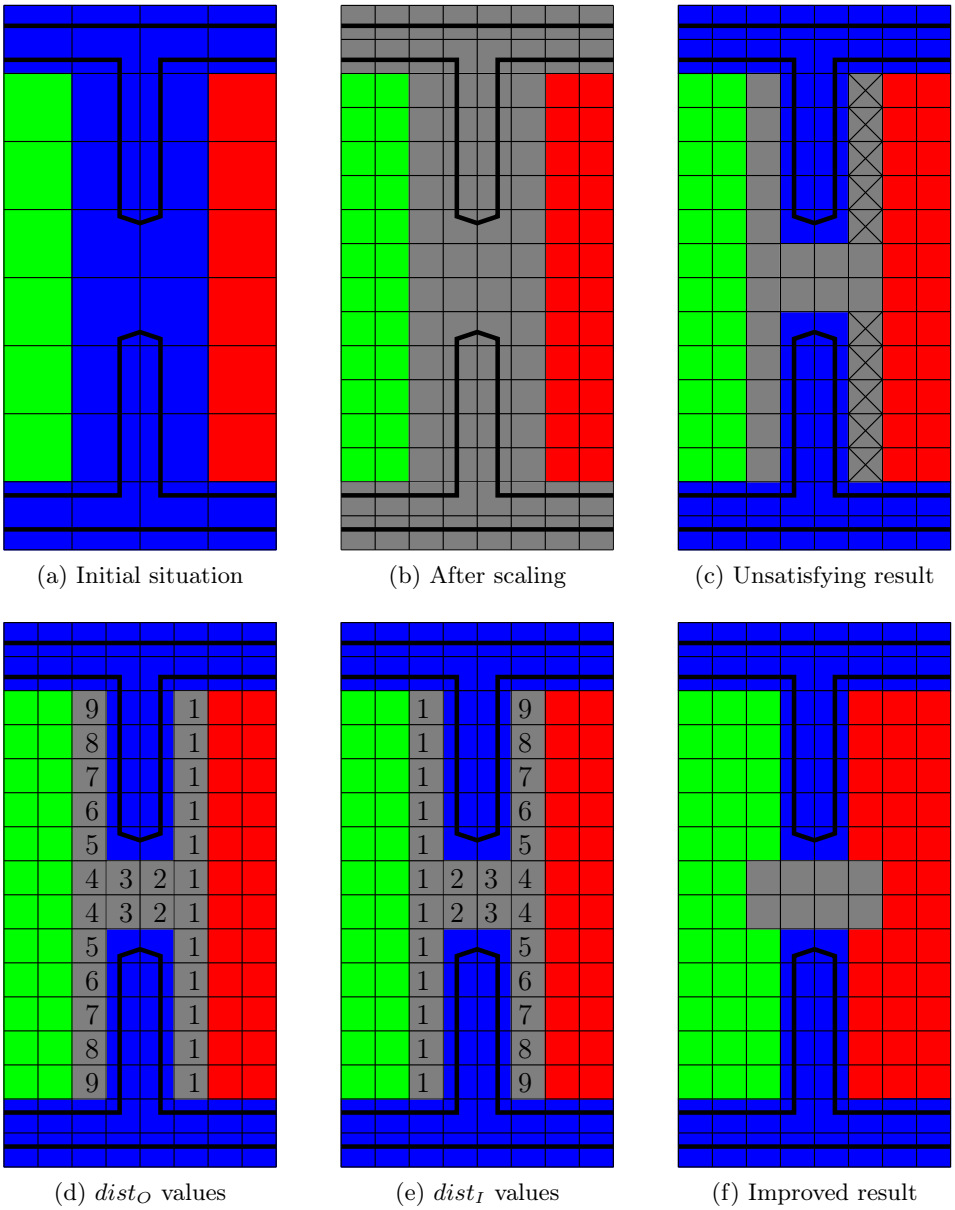
Consider the algorithm presented in Algorithm 4.8. Each cell c' of the transformed grid $\mathcal{G}'$ is covered by exactly one cell c of the original grid $\mathcal{G}$. In other words, the sets S and L in Algorithm 4.8 contain exactly one element. Thus the entire space previously covered by INSIDE and OUTSIDE cells is still covered by appropriate labeled cells (INSIDE* cells are replaced as INSIDE cells).

We want to attract your attention to a phenomenon that occurs in conjunction with scaling. Consider the example in Figure 4.9. A fragment of a grid $\mathcal{G}$ with $l = 50\text{mm}$ is shown in Figure 4.9(a). Assume we want to pack rectangles of size $100\text{mm} \times 50\text{mm}$. The hole in the boundary is large enough such that a rectangle can pass it. In general we consider such input data as ill-formed and require the user to fix the errors. However, in this special case, there is no problem, because the BOUNDARY cells still separate INSIDE and OUTSIDE cells.

Now we scale the grid $\mathcal{G}$ with $k = 2$ and obtain a new grid $\mathcal{G}'$. The result of Algorithm 4.8 is displayed in Figure 4.9(b). The information abound INSIDE and OUTSIDE cells has been carried over to the new grid. All cells that constitute the former BOUNDARY cells are now labeled as UNKNOWN. Next we rerun the algorithms of Section 4.3 and classify BOUNDARY and unusable cells (see Figure 4.9(c)). Because INSIDE and OUTSIDE cells are no longer separated by BOUNDARY (or UNKNOWN*) cells, we fail to further reduce the number of UNKNOWN and UNKNOWN* cells.

Consider both branches of UNKNOWN* cells. On one hand, these cells are adjacent to OUTSIDE cells. On the other hand, they are far away from INSIDE cells (distance is measured as the length of a shortest path of UNKNOWN or UNKNOWN* cells). Intuitively, one would classify these cells as OUTSIDE. In a similar way, one would classify the lower and upper branch of UNKNOWN cells as INSIDE.

We present an algorithm based on this idea (see Algorithm 4.9). First, we use breadth-first search to compute the values $dist_O$ and $dist_I$ for each UNKNOWN and UNKNOWN* cell. The values $dist_O[c]$ and $dist_I[c]$ denote the distance of cell c to the next OUTSIDE and INSIDE (or INSIDE*) cell,



(a) Initial situation (b) After scaling (c) Unsatisfying result

(d) $dist_O$ values (e) $dist_I$ values (f) Improved result

Figure 4.9: A problem caused by large holes in the boundary. The boundary of the trunk contains a hole larger than 100mm×50mm (a). After scaling (b), the sets of INSIDE and OUTSIDE cells are no longer separated by BOUNDARY cells, which leads to unsatisfying results (c). The results can be improved (f) by taking into account the shortest distance to the sets of OUTSIDE (d) and INSIDE (e) cells.

respectively. The values of $dist_O$ and $dist_I$ in our example are shown in Figure 4.9(d) and 4.9(e). We compare the ratio of both values with two thresholds t_I and t_O with $t_I < t_O$. If a threshold is exceeded, the cell is labeled accordingly. If the distance ratio is contained in the interval $[t_I, t_O]$, the cell label remains unchanged.

The result for $t_I = \frac{1}{4}$ and $t_O = 4$ is depicted in Figure 4.9(f). Apart from a few cells close to the hole, all cells are labeled as desired.

Algorithm 4.9 Labeling of cells based on distances to other cells

CompareDistances($\mathcal{G}$)
1: compute $dist_I$ and $dist_O$ for all Unknown and Unknown* cells of $\mathcal{G}$
2:
3: **for all** Unknown and Unknown* cells $c \in \mathcal{G}$ **do**
4: **if** $dist_I[c]/dist_O[c] > t_O$ **then**
5: label c as Outside
6: **else if** $dist_I[c]/dist_O[c] < t_I$ **then**
7: label c as Inside (or Inside*)
8: **end if**
9: **end for**

4.4.2 Translation

A translation of a grid is related to a change of the origin $\boldsymbol{o}$; the parameters for orientation $\mathbf{R}$ and spacing l remain the same. We denote the distance between the old and the new origin, expressed in the local coordinate system of the grid, by $\boldsymbol{d} := \mathbf{R}^{-1}(\boldsymbol{o}' - \boldsymbol{o})$.

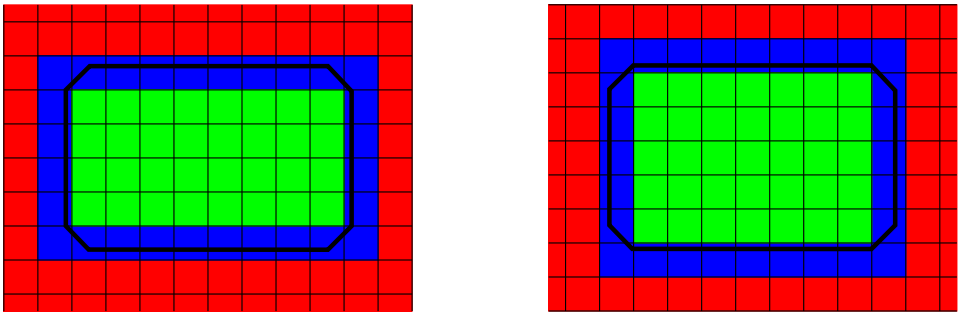
A translation by any element of $l \cdot \mathbf{R} \cdot \mathbb{Z}^3$ does not change the subdivision of the space into cells — it just changes the indices associated with each cell. Therefore we can restrict ourselves to $\boldsymbol{d} \in [0, l)^3$.

Consider the algorithm presented in Algorithm 4.8. It holds $|S| \leq 8$, since each cell c' of the transformed grid $\mathcal{G}'$ is covered by the union of at most eight cells of the original grid. More exactly, it holds $|S| = 2^i$, where i denotes the number of components of d that are not equal to zero.

The case $i = 1$ corresponds to translations along one axis of the grid. Each cell of the transformed grid is covered by the union of two adjacent cells of the original grid. Such translations will be extensively used in the following section.

4.5 Optimization

The previous sections explained how to classify the cells of a grid, provided the parameters that describe its geometry are given. We did not yet explain how to choose these parameters.



(a) 32 INSIDE cells, 4 rectangles (b) 35 INSIDE cells, 3 rectangles

Figure 4.10: Drawbacks of a simple optimization criterion (I). Both sets of INSIDE cells have similar shape, but differ in cardinality. Although the set in the right figure is larger, the size of an optimal packing is smaller.

Our ultimate goal is to compute a packing of as much boxes as possible. Since the computation of packings is computationally very expensive, we cannot run our algorithms on several grid instances. We have to compute a single grid on which all following computations are based.

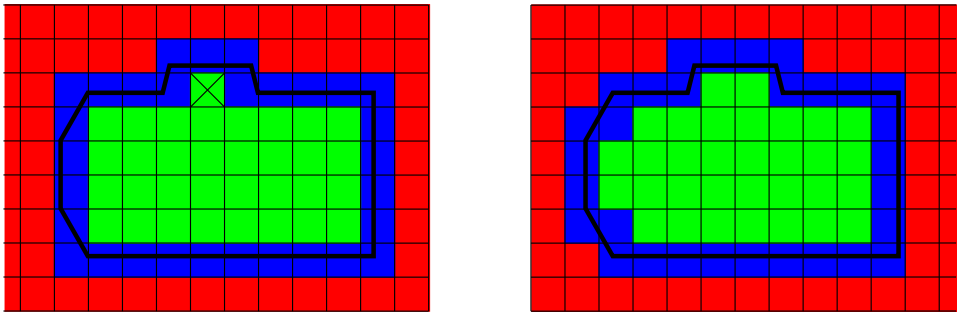
Clearly, the number of boxes that can be packed is bounded by the available volume. In the discrete setting, the available volume corresponds to the number of INSIDE cells. Therefore, a natural choice for a quality criterion is the number of INSIDE cells.

However, a higher number of INSIDE cells does not necessarily imply a packing of higher cardinality. Such a simple criterion does not take into account the relative positions of the INSIDE cells (which relate to the structure of the corresponding conflict graph). We present two examples that demonstrate the drawbacks of this criterion.

Figure 4.10 shows two grids that have a similar shape, but a different number of INSIDE cells. The left grid consists of 32 INSIDE cells and can be packed with four rectangles of size 4×2 cells. The right grid consists of even 35 INSIDE cells, but cannot be packed with more than three rectangles. Figure 4.11 shows two grids that have the same number of INSIDE cells, but a different shape. The rectangular shape of the left grid allows a packing with four rectangles, whereas a maximum packing for the irregular-shaped grid on the right consists of three boxes.

We have seen that the number of INSIDE cells has its drawbacks as a quality criterion. We use it nevertheless because it can be evaluated very fast. An improved criterion is proposed as an item for further work at the end of this thesis (see Section 7.2).

We briefly discuss the influence of spacing, orientation and position of the grid on the number of INSIDE cells. The choice of the spacing and the orientation is left to the user because it is quite easy to come up with



(a) Rectangular shape, 4 rectangles (b) Irregular shape, 3 rectangles

Figure 4.11: Drawbacks of a simple optimization criterion (II). Both sets of INSIDE cells have the same cardinality, but differ in shape. The irregular shape in the right figure leads to a smaller packing.

a meaningful choice, and thus to strongly reduce the number of potential grids. On the other hand, it is quite difficult for the user to select a good origin for the grid. Therefore, we apply an optimization scheme to improve the position of the grid.

We leave the choice of the spacing of the grid to the user. The spacing has a strong influence on the number of INSIDE cells and on the overall complexity of the discrete problem. The requirement that $n \cdot l = 50\text{mm}$ holds for some integer n limits the number of choices for l to 50mm, 25mm and 12.5mm. A spacing of 50mm is quite coarse and often leads to unsatisfying results. On the other hand, the problem complexity is quite small and results can be obtained very fast. Most of the time, a spacing of 25mm provides a good compromise between problem complexity and quality of the obtained solution. We also use fine grids with a spacing of 12.5mm for smaller volumes. Grids with a spacing of 6.25mm (or even less) lead to packing problems that are too large for the presented algorithms.

Similarly, we leave the choice of the orientation to the user. Usually, the shape of the trunk gives a strong suggestion for a promising orientation. In most instances, there is a large, planar face, e.g., the floor of the trunk. In order not to waste too much space, it is reasonable to align the grid with this large face. The faces bounding the large face, e.g., the walls of the trunk, indicate good choices for the last degree of rotational freedom.

The orientation of the grid defaults to the orientation of the global coordinate system and can be overridden by the user in the following way. The user may specify up to three planes (the planes orthogonal the axes of the global coordinate system are the default). A plane can be specified either by selecting three vertices or by picking a face. We apply an orthogonalization scheme to the specified plane normals. The resulting vectors determine the axes of the grid (column vectors of $\mathbf{R}$). Two planes are sufficient to define

the orientation. The third plane is used to specify the position of the grid: The intersection point of all three planes is used as an initial value for the origin.

We use a simple brute-force approach to find a good candidate for the origin. Suppose the grid is aligned with a face with normal $(0, 0, 1)^{\mathrm{T}}$. We are left with two degrees of translational freedom, e.g., the origin of the grid can be translated in the plane spanned by $(1, 0, 0)^{\mathrm{T}}$ and $(0, 1, 0)^{\mathrm{T}}$. We choose a parameter $k \in \mathbb{N}$ and translate the initial origin by $(\frac{i}{k}l, \frac{j}{k}l, 0)$ for all integral pairs $(i, j) \in [0, k)^2$. We return the instance that has the highest number of INSIDE cells. Our brute-force approach can be easily replaced by local search methods (see Section 7.2).

Chapter 5

Algorithms

In this chapter we present different algorithms to solve the optimization variant of the Discrete-Box-Packing problem (see Definition 2.3). The input of the problem is given as follows: We are given a cubic grid $\mathcal{G}$, a parameter $n \in \mathbb{N}$ and a subset $I \subseteq \mathbb{Z}^3$ of the set of grid cells. The parameter n is a small constant that relates the spacing l of the grid to the extensions of the boxes. More precisely, the boxes consist of $4n \times 2n \times n$ grid cells, and it holds $n \cdot l = 50\text{mm}$. The set I defines the cells that are available for packing. The union of the cells in I is an inner approximation of the trunk interior.

Given the set I and the parameter n, the geometric parameters of the grid $\mathcal{G}$ are irrelevant for the Discrete-Box-Packing problem. The spacing l of the grid is encoded by the parameter n; the origin $\boldsymbol{o}$ and the orientation $\mathbf{R}$ do not matter. The latter two parameters are solely needed to transform a solution back into three-dimensional space. For simplicity, we can assume that the grid is axis-aligned; hence the boxes are the cartesian product of right-open intervals.

The task, as formulated in the definition of the Discrete-Box-Packing problem, is to compute a maximum packing of cell-aligned boxes with extensions $4n$, $2n$ and n, i.e., a maximum set of boxes such that only cells in I are covered and each cell in I is covered by at most one box. Given the problem difficulty, we relax the problem as follows: We are not only interested in *maximum* packings, but also in packings with a cardinality close to the optimum. First we present a problem formulation based on integer linear programming (ILP), which can be used to compute a maximum packing. Further presented algorithms are heuristics.

The Discrete-Box-Packing problem can also be formulated in a different way. Given the input $(\mathcal{G}, n, I)$, it is straightforward to compute the corresponding conflict graph $G := G(\mathcal{G})$ as defined in Section 4.1.3. Then the Discrete-Box-Packing problem $(\mathcal{G}, n, I)$ corresponds to a maximum stable set problem for the conflict graph G.

Both representations have their advantages and disadvantages. The grid representation is compact and encodes the geometric structure of the problem. The representation as conflict graph explicitly represents all possible boxes (nodes) and conflict pairs (edges). It also offers other useful information, e.g., node degrees. On the other hand, the conflict graph requires more space and hides the geometric structure of the grid. For example, given the grid representation, the maximal cliques of the conflict graph can be easily computed. Depending on the nature of the algorithms, we will use either one or the other representation.

We remark that there are two upper bounds for our packing problem. The trivial bound $\lfloor \frac{|I|}{8n^3} \rfloor$ relates the volume of a box to the total volume of the cells in I. This bound does not take into account the shape of the trunk. A second, better bound can be obtained from a linear programming formulation of the problem (see Section 5.1.2 for details).

For practical reasons, all algorithms have to adhere to a given time bound. After this time has elapsed, the algorithm has to return a non-trivial, probably suboptimal packing. This requirement rules out algorithms that have a fixed runtime (depending on the instance, of course) and do not produce any intermediate solution until the very end. For example, an algorithm that works on subsets of boxes that do not denote a valid packing is impractical, unless its runtime can be controlled.

We distinguish two classes of algorithms. One class called *direct approaches* contains all algorithms that can solve the given problem on its own. The algorithms in the second class called *divide-and-conquer approaches* generate a set of smaller subproblems, which require an algorithm from the first class as a subroutine. We want to remark that the divide-and-conquer approaches are not recursive as it is the case for typical divide-and-conquer algorithms. The subdivision step occurs only once rather than several times.

5.1 Direct Approaches

In this section we discuss five direct approaches for the Discrete-Box-Packing problem, i.e., algorithms that do not require another algorithm to solve subproblems. We begin with a fast and simple greedy heuristic. Afterwards, we turn to integer linear programming (ILP) techniques and present an algorithm that is capable of computing an optimal solution. We continue with a heuristic based on the linear programming (LP) relaxation. It follows an algorithm called *Reactive Local Search* which is based on tabu search. Finally, we present a simple, but promising heuristic called *Simplefill*.

For reasons of simplicity, code related to the compliance of the time bound is omitted from the pseudocode descriptions. Suitable statements can be easily added to the main loop of the algorithms.

5.1.1 Greedy

The most obvious idea for a heuristic for the maximum stable set problem in a graph is to use a greedy approach. The Greedy algorithm selects a node

with minimum degree from the conflict graph G and adds it to the stable set determined so far, then removes this node and all its neighbors from the graph and repeats. Priority queues can be used to efficiently select a node with minimum degree. The pseudocode for an implementation using such a priority queue is shown in Algorithm 5.1. The set $N_i(v)$ denotes all nodes $u \in V$ for which the length of a shortest from u to v equals i (all edges have unit length).

Algorithm 5.1 Greedy algorithm

```
GREEDY (G(V, E))
1: S ← ∅
2: Q ← priority queue containing nodes V, sorted by non-decreasing degree
3: while Q ≠ ∅ do
4:     v ← GETMIN(Q)
5:     S ← S ∪ {v}
6:     remove v and N_1(v) from G and Q
7:     update degrees of N_2(v) in Q
8: end while
9: return S
```

The Greedy algorithm tends to place boxes first close to the boundary of the trunk. It keeps adding boxes close to the boundary of the remaining space, and hence fills the trunk from the boundary to the center. This behavior is due to the fact that placements close to the boundary of the available space are in conflict with fewer other placements. Correspondingly, nodes representing the former placements have lower degree than nodes representing the latter.

Usually there are several nodes with minimum degree; thus the implementation of the priority queue determines which node is actually chosen. Instead of selecting a fixed node, we can choose a node with minimum degree uniformly at random. This modification can be accomplished by replacing the priority queue with an array holding the node degrees. The time to select a node increases, but is still dominated by the update costs and does therefore not drastically influence the runtime. By repeating the randomized version several times, we can improve the size of the computed stable set at the cost of runtime. We compare the deterministic and randomized version in Section 6.1.1.

5.1.2 Integer Linear Programming

Integer linear programming (ILP) techniques for packing problems have been extensively studied (see for example [Bea85] and [HC95]). Theoretically, an algorithm based on ILP techniques guarantees —in contrast to all other presented algorithms— an optimal solution of the maximum stable set problem.

However, the required runtime increases exponentially with the problem size. Our instances are often too large to obtain an optimal solution in reasonable time. Nevertheless, an algorithm based on ILP is an important tool for smaller subproblems.

In our ILP formulation of the stable set problem, we use a decision variable x_v for each node $v \in V$. The variable x_v indicates whether v is contained in the stable set ($x_v = 1$) or not ($x_v = 0$). Recalling the definition of stable sets (see Definition 2.5), we directly obtain the so-called *edge formulation*

$$\begin{aligned} \max \sum_{u \in V} x_v & \\ x_u + x_v \leq 1 & \qquad \text{for all } \{u, v\} \in E \\ x_v \in \{0, 1\} & \qquad \text{for all } v \in V\,. \end{aligned} \tag{5.1}$$

It is easy to see that the vectors $\boldsymbol{x}$ that satisfy the constraints of (5.1) are exactly the characteristic vectors of stable sets of G.

Let P denote the polytope that is defined by the convex hull of all vectors $\boldsymbol{x}$ that satisfy the constraints of (5.1). Let P_E denote the polytope of the LP relaxation of (5.1), i.e., the set of all vectors $\boldsymbol{x}$ that satisfy the constraints $x_u + x_v \leq 1$ for all $\{u, v\} \in E$ and $x_v \geq 0$ for all $v \in V$. It holds $P \subseteq P_E$, and hence $\max\{\sum_{v \in V} x_v \mid \boldsymbol{x} \in P\} \leq \max\{\sum_{v \in V} x_v \mid \boldsymbol{x} \in P_E\}$. The optimal value of the LP relaxation is an upper bound for the optimal value of (5.1).

The goal is to use a strong problem formulation, i.e., a set of constraints such that the polytope of the corresponding LP relaxation is as small as possible. In particular, we are interested in constraints that define facets of the polytope P. A trivial class of such facet-defining constraints is $x_v \geq 0$ for all $v \in V$. We present two other classes, *maximal clique inequalities* and *lifted odd hole inequalities*.

Maximal Clique Inequalities

Definition 5.1 (Clique). *A subset C of the nodes V of a graph $G = (V, E)$ is called* clique *if each pair of nodes from C is connected by an edge in E.*

The following observation follows directly from the definitions of cliques and stable sets (see Definition 2.5).

Proposition 5.2. *Let S be a stable set and C a clique, then $|S \cap C| \leq 1$.*

It follows that for each clique C the inequality $\sum_{v \in C} x_v \leq 1$ is a valid inequality for P. Thus we obtain the so-called *clique formulation*

$$\begin{aligned} \max \sum_{u \in V} x_v & \\ \sum_{v \in C} x_v \leq 1 & \qquad \text{for all } C \in \mathcal{C} \\ x_v \in \{0, 1\} & \qquad \text{for all } v \in V\,, \end{aligned} \tag{5.2}$$

where $\mathcal{C}$ denotes the set of all cliques of G.

Let P_C denote the polytope of the LP relaxation of (5.2). Note that the clique inequalities imply the edge inequalities of (5.1). Hence $P \subseteq P_C \subseteq P_E$ holds. With the exception of some pathological cases $P_C \subsetneq P_E$ holds, i.e., the clique formulation is stronger than the edge formulation. In particular, the optimal value of the LP relaxation based on the clique formulation is not larger than the corresponding value based on the edge formulation. Hence the former value is a better (or equal) upper bound for the packing problem.

It is not necessary to consider all cliques $C \in \mathcal{C}$. We can restrict ourselves to the subset $\mathcal{C}_{max} \subseteq \mathcal{C}$ of maximal cliques, i.e., cliques that are maximal with respect to set inclusion. Let $C \in \mathcal{C}_{max}$ be a maximal clique and $M \subseteq C$. Then the inequality $\sum_{v \in C} x_v \leq 1$ implies the inequality $\sum_{v \in M} x_v \leq 1$. Thus we can replace $\mathcal{C}$ by $\mathcal{C}_{max}$ in (5.2) and obtain the so-called *complete clique formulation* (see [SVA00]):

$$\begin{aligned} \max & \sum_{u \in V} x_v & & \\ & \sum_{v \in C} x_v \leq 1 & & \text{for all } C \in \mathcal{C}_{max} \\ & x_v \in \{0,1\} & & \text{for all } v \in V\,, \end{aligned} \tag{5.3}$$

This modification significantly reduces the number of inequalities. The polytope defined by the LP relaxation of (5.3) is equal to P_C.

The importance of maximal cliques is established in the following result by Padberg (see [Pad73]):

Proposition 5.3. *If $C \in \mathcal{C}_{max}$ is a maximal clique of G, then the corresponding clique inequality $\sum_{v \in C} x_v \leq 1$ defines a facet of P.*

Erdös [Erd62], and later Moon and Moser [MM65] proved that the number of maximal cliques in a general graph is exponential in the number of nodes. But in our case, the class of conflict graphs G has a special structure, such that the number of maximal cliques is polynomial in $|V|$. Moreover, the set of all maximal cliques of G can be computed in polynomial time.

In order to proof this result, we start with an observation that establishes a relation between cliques and sets of boxes:

Proposition 5.4. *A clique of G corresponds to a set of boxes with non-empty intersection.*

We need the following argument to proof the statement. A generalization thereof for closed convex sets in any dimension was proved by Helly in [Hel23].

Proposition 5.5. *Let $I = \{I_1, \ldots, I_m\}$ be a set of right-open intervals such that $I_i \cap I_j \neq \emptyset$ for all $1 \leq i < j \leq m$. Then $\bigcap_{i=1}^{m} I_i \neq \emptyset$.*

Proof. Define $\underline{I_i} := \min I_i$ and $\overline{I_i} := \sup I_i$. Since $I_i \cap I_j \neq \emptyset$, it holds $\underline{I_i} < \overline{I_j}$ for all $1 \leq i, j \leq m$. It follows that $\underline{I} := \max_{1 \leq i \leq m} \underline{I_i} < \min_{1 \leq i \leq m} \overline{I_i} =: \overline{I}$ and $\bigcap_{i=1}^{m} I_i = \left[\underline{I}, \overline{I}\right) \neq \emptyset$. □

We are now able to present the proof of Proposition 5.4.

Proof of Proposition 5.4. We are given a clique $C = \{v_i \in V | 1 \leq i \leq m\}$ in G. Let B_i denote the box corresponding to node v_i. Let I_i^k denote the projection of box B_i onto the k-th axis of the coordinate system. Any pair $\{I_i^k, I_j^k\}$ of such intervals has non-empty intersection since the corresponding nodes v_i and v_j are contained in the clique C and hence are adjacent. By Proposition 5.5, we have $I^k := \bigcap_{i=1}^{m} I_i^k \neq \emptyset$. Note that each box B_i is the cartesian product of its projection intervals $I_i^1 \times I_i^2 \times I_i^3$. Hence the cartesian product $I^1 \times I^2 \times I^3 \neq \emptyset$ is contained in any box B_i. It follows that the intersection of all boxes is not empty.

Now suppose we are given a set $B = (B_1, \ldots, B_m)$ of boxes with non-empty intersection. Thus any pair of boxes has non-empty intersection and the corresponding nodes are adjacent. Hence the set of nodes that corresponds to the set of boxes B is a clique of G. □

The next result follows directly from Proposition 5.4:

Corollary 5.6. *A maximal clique of G corresponds to a maximal set of boxes with non-empty intersection.*

Recall that the boxes consist of a set of $4n \times 2n \times n$ cells. Hence "non-empty intersection" is equivalent to "the intersection covers at least one cell".

Consider a cell $c \in \mathcal{G}$. The maximal set of boxes that cover the cell c is called *the (set of) boxes generated by the cell c*. Similarly, the clique that corresponds to this set of boxes is called *the clique generated by the cell c*.

Note that there are cells such that the generated clique is not maximal. This is due to the fact that there are cells $c \in I$ such that the set of boxes that cover the cell c is restricted by the geometry of the trunk. As a consequence, there might be cells in close proximity to c such that the generated set of boxes is strictly larger than the set of boxes generated by c.

Note also that there are maximal cliques such that the intersection of the corresponding boxes covers more than one cell. In other words, distinct grid cells do not necessarily generate distinct cliques. For example, consider a tunnel of cells with a cross section of size $2n \times n$ cells and sufficient length. The shape of the tunnel enforces the same orientation for all boxes. There are maximal cliques of $4n - 1$ nodes such that the intersection of the corresponding boxes covers $2n \times n$ cells. All these cells generate the same clique.

We take both observations into account when generating the set of maximal cliques.

Proposition 5.7. *The number $|\mathcal{C}_{max}|$ of maximal cliques is bounded by $|I|$. The size of any maximal clique is bounded by $48n^3$. The set $\mathcal{C}_{max}$ of maximal cliques can be enumerated in polynomial time.*

Proof. Corollary 5.6 and the subsequent observation yield that the number $|\mathcal{C}_{max}|$ of maximal cliques is bounded by the number of the cells $|I|$.

Boxes consist of $4n \times 2n \times n$ cells and there are six orientations per box. Hence the set of boxes generated by a cell contains at most $48n^3$ items. This number bounds the size of any clique of G.

We compute the set $\mathcal{C}_{max}$ of maximal cliques as follows: Fix a cell $c \in I$ and compute the set of boxes and the clique that is generated by this cell. We assume that we can check in constant time whether a particular box covers only cells marked as INSIDE (this information was already computed during construction of the conflict graph). Hence the clique generated by the cell c can be computed in $O(n^3)$ time. We check whether the clique is maximal which can be done in $O(|V|\, n^3)$ time. We also check whether the clique has already been generated by a previously considered cell. This check requires $O(|I|\, n^3)$ time provided that the node sets of all cliques computed so far are sorted by a common criterion (which can be easily achieved by construction). If both checks succeed, add the clique to the set of maximal cliques. Repeat for all cells $c \in I$. The total running time of this algorithm is $O(|I|\, n^3(|I| + |V|))$. □

We remark that the algorithm described in the above proof is not very efficient. Actually, it is not necessary to implement a filter that removes generated cliques that are not maximal. State-of-the-art ILP solver contain a so-called *presolve phase* that efficiently eliminates such redundant constraints.

Lifted Odd Hole Inequalities

CHVÁTAL proved in [Chv75] that the polytope P consists of facets that cannot be described by non-negativity or maximal clique constraints, unless the underlying graph is perfect. PADBERG described in [Pad73] a third class of facet-defining inequalities, the so-called *lifted odd hole inequalities.*

Definition 5.8 (ODD HOLE). *A subset H of the nodes V of a graph $G = (V, E)$ is called* odd hole *(or* chordless cycle of odd length*) if H is a cycle of odd length such that there is no pair of non-consecutive adjacent nodes.*

The following observation follows directly from the definition of odd holes and stable sets (see Definition 2.5).

Proposition 5.9. *Let S be a stable set and H an odd hole, then $|S \cap H| \leq \lfloor |H|/2 \rfloor$.*

It follows that for each odd hole H the constraint

$$\sum_{v \in H} x_v \leq \frac{|H| - 1}{2} \tag{5.4}$$

is a valid inequality for the polytope P. In contrast to the maximal clique inequalities, the odd hole inequalities do not define facets of P. However, there exists a sequential lifting process that strengthens an odd hole inequality. The result is called *lifted odd hole inequality* and it defines a facet of P.

This lifting process works as follows: Suppose we have a valid inequality

$$\sum_{v \in U} \alpha_v x_v \leq \beta, \tag{5.5}$$

where U is a subset of the nodes V. The goal is to strengthen this inequality by adding the decision variable x_u of a new node $u \in N_1(U) \backslash U$ with non-zero coefficient α_u to the left side. Since we want to strengthen the inequality as much as possible, we are looking for the maximal α_u such that the inequality

$$\alpha_u x_u + \sum_{v \in U} \alpha_v x_v \leq \beta \tag{5.6}$$

is valid for P. If the decision variable x_u is zero, then there is no restriction on α_u. If $x_u = 1$, then the maximal α_u is equal to $\beta - \gamma$, where γ is the maximum weight $\sum_{v \in U \backslash N_1(u)} \alpha_v x_v$ of a stable set of the subgraph induced by U without the nodes adjacent to u. We set α_u to $\beta - \gamma$ and add the term $\alpha_u x_u$ to the left side of (5.5). The process is repeated with the remaining nodes in $N_1(U) \backslash U$.

Padberg has shown in [Pad73] that the lifted odd hole inequalities obtained by the previously described procedure define facets of P. Note that this property holds independently of the order in which the nodes of $N_1(U) \setminus U$ are considered.

In contrast to the maximal clique inequalities, we do not add odd holes inequalities a priori to the set of constraints. Instead, we dynamically compute violated odd hole inequalities during the branch-and-cut phase and add the corresponding constraint to the system of inequalities. Next we describe how to identify violated odd holes.

Let $\boldsymbol{x} \in [0, 1]^V$ denote a fractional solution of the LP relaxation of (5.3). We describe how to identify odd holes H such that inequality (5.4) is violated. We are also interested in nearly violated inequalities, i.e., odd holes H such that

$$\sum_{v \in H} x_v > \frac{|H| - 1}{2} - \varepsilon \tag{5.7}$$

holds for some fixed $\varepsilon > 0$. The motivation for this relaxation is that the following sequential lifting process strengthens the inequality and might produce a violated inequality. Define for each edge $e = \{u, v\} \in E$ the edge

weight $w_e := 1 - x_u - x_v \geq 0$. Let H be an odd hole and E_H the edges of H. Then

$$\sum_{e \in E_H} w_e = |H| - 2 \sum_{v \in H} x_v \tag{5.8}$$

holds and inequality (5.7) is equivalent to

$$\sum_{e \in E_H} w_e < 1 + 2\varepsilon. \tag{5.9}$$

Gerards and Schrijver described in [GS86] a construction that reduces the problem of finding odd holes H satisfying (5.9) to a shortest path problem in an auxiliary graph G'. The graph $G' = (V', E')$ is constructed as follows: For each node $v \in V$ we add two nodes v^+ and v^- to V'. Two nodes $u^+ \in V'$ and $v^- \in V'$ are adjacent in G' iff the nodes $u \in V$ and $v \in V$ are adjacent in G. The edge weights in G' are defined according to the edge weights in G. Observe that a path from u^+ to u^- in G' corresponds to an odd cycle in G that contains the node u.

Since all edge weights are non-negative we can use Dijkstra's algorithm to compute shortest paths. Find a node $u \in V$ such that there is a shortest path from u^+ to u^- in G' with total length at most $1+2\varepsilon$. The corresponding walk in G may contain some node multiple times and may have chords, i.e., a pair of non-consecutive adjacent nodes. Strijk et al. proved in [SVA00] that this walk contains a subsequence of nodes that forms an odd hole H satisfying (5.7). Note that an odd hole consisting of three nodes forms a clique and an inequality corresponding to a maximal clique that contains this clique is already part of the problem formulation. Hence the odd hole that was computed by the previously described procedure consists of at least five nodes.

Once we have identified a violated odd hole, we strengthen the corresponding inequality using the previously described lifting procedure. We have not yet discussed the *lifting sequence*, i.e., the order in which the nodes of $N_1(H) \setminus H$ are considered. Nemhauser and Sigismondi [NS92] present an algorithm that computes all facet-defining inequalities that can be obtained by lifting a given odd hole inequality. We propose a simpler approach using several heuristics.

Note that the coefficient α_u, $u \in H$, is larger if the variable is considered earlier in the lifting sequence. Thus the lifted inequality is in some sense stronger with respect to the earlier lifted variables and weaker with respect to the later lifted ones. The proposed heuristics are:

- Consider the variables $x_u, u \in H$ ordered by non-decreasing fractional value, i.e., the value $\left|\frac{1}{2} - x_u\right|$.
- Lift each variable $x_u, u \in H$ independently and let α'_u denote the obtained lifting coefficient. Consider the variables $x_u, u \in H$ ordered by non-increasing value α'_u.

- Let H_u denote the subgraph of G induced by $H \setminus N_1(u)$. Note that smaller subgraphs H_u tend to produce smaller values γ_u and larger values α_u. Hence consider the variables $x_u, u \in H$ ordered by non-decreasing size of the subgraph H_u.
- Consider the variables in a fixed order and its reverse order.

All unique inequalities obtained by these heuristics are added to the problem formulation of the considered subtree in the branch-and-cut tree.

5.1.3 LP Rounding

The computation of an optimal solution of the ILP formulation presented in the previous chapter is pretty difficult. This is not surprising, since the Discrete-Box-Packing problem is *NP*-complete. On the other hand, it is much simpler to compute an optimal solution of its LP relaxation. For our instances, the LP relaxation can still be solved in reasonable time.

A well-known family of heuristics to exploit the information contained in the LP relaxation is called *LP rounding*. These approaches round the fractional values of the LP relaxation to integral values. Special care is needed to avoid violated constraints. Such heuristics are part of many ILP solvers and can be used to obtain a lower bound for the cardinality of the maximum stable set. Unfortunately, the quality of the solutions obtained by these heuristics is quite bad.

Therefore we designed our own heuristic based on the values of the LP solution. In contrast to general LP rounding heuristics, our heuristic is specifically tailored to the maximum stable set problem. The main motivation of our algorithm is the assumption that high fractional values are good indicators for variables that should be fixed to 1.

The LP Rounding algorithm is described in Algorithm 5.2. It takes the graph G and an additional parameter $0 \leq p \leq 1$ as input. The parameter p controls the fraction of variables that are fixed in each iteration.

The algorithm starts with the LP relaxation of the complete clique formulation for G (see (5.3)). Let $\boldsymbol{x}$ denote the optimal solution of this linear program. The variable m denotes the number of variables in the linear program that have already been fixed (initially 0).

In each iteration, the algorithm considers the highest fractional variables of the linear program in non-increasing order. The number k of variables that are considered is proportional to the remaining problem complexity (line 6). If such an LP variable x_{i_j} can be set to 1 without violating any constraints, an inequality that fixes x_{i_j} to 1 is added to the problem formulation. Note that the set of k largest fractional variables might contain variables corresponding to adjacent nodes. Thus the test in line 8 is indeed necessary. After all k variables have been considered, the modified LP problem is resolved to optimality. This process is repeated as long as there are fractional variables in the LP solution.

Algorithm 5.2 LP Rounding (LPR) algorithm

LPR $(G(V,E),p)$
1: let LP denote the LP relaxation of the complete clique formulation for G
2: $\boldsymbol{x} \leftarrow$ optimal solution of LP
3: $m \leftarrow 0$
4: **while** a fractional variable x_i exists **do**
5: let the sequence $(x_{i_1}, x_{i_2}, \ldots)$ denote the fractional variables of $\boldsymbol{x}$, sorted in non-increasing order
6: $k \leftarrow \max\left\{1, \left\lfloor p \cdot \left(\sum_{i=1}^{|V|} x_i - m\right)\right\rfloor\right\}$
7: **for** $j := 1$ **to** k **do**
8: **if** $x_{i_j} = 1$ does not violate any constraints **then**
9: add constraint $x_{i_j} \geq 1$ to LP
10: $m \leftarrow m + 1$
11: **end if**
12: **end for**
13: $\boldsymbol{x} \leftarrow$ optimal solution of LP
14: **end while**
15: **return** $\{v \in V | x_v = 1\}$

The parameter p controls the fraction of variables that are fixed in each iteration (line 6). Note that $\sum_{i=1}^{|V|} x_i$ is the objective function of the linear program and m denotes the number of fixed variables. Thus the difference $\sum_{i=1}^{|V|} x_i - m$ is an upper bound on the number of variables that still can be fixed to 1. In the case of the extremal value $p = 1$, a single iteration is performed and exactly one linear program is solved. The variables are set to 1 based on the order imposed by the values of the initial LP solution, additionally taking into account the constraints. This leads to a low runtime and also low solution quality. In the other extremal case $p = 0$, exactly one single fractional variable is fixed in each iteration (note the maximum operation in line 6). The number of LP problems equals the size of the computed stable set. This choice leads to a good solution, but also results in a high runtime. The choice of the parameter $p \in (0, 1)$ allows to weight the controversial goals of low runtime and high quality of the solution. We present experimental results on the choice of p in Section 6.1.3.

The following modification of Algorithm 5.2 helps to improve the quality of the solution. We leave the loop in line 4 as soon as the number of fractional variables in the LP solution drops below a fixed threshold. Afterwards, we use the ILP algorithm to compute an optimal solution of the remaining subproblem. Higher thresholds lead to better overall solutions, but impose a higher runtime at the same time.

5.1.4 Reactive Local Search

The Reactive Local Search (RLS) algorithm is based on an algorithm with the same name presented by Battiti and Protasi in [BP01]. While the original algorithm is stated in terms of the maximum clique problem, we present it here directly applied to the maximum stable set problem. We give a high-level description of the algorithm and refer the interested reader to the given reference for the details, including a discussion of data structures for an efficient implementation.

Software systems for optimization problems often contain frameworks for local search algorithms, see for example the tabu search framework OpenTS in the software package COIN-OR [COI].

The RLS algorithm is based on a local search strategy complemented by a history-sensitive feedback scheme to control the amount of diversification. In each iteration a local search algorithm replaces the current configuration by a neighboring, better configuration. In our setting, a configuration corresponds to a stable set, and a configuration is called better than another one if the corresponding stable set has a higher cardinality. Two configurations are neighbors if they can be transformed into each other by adding or dropping a single node.

In order to escape from local optima, the algorithm also accepts configurations that decrease the objective value. To avoid cycles, the inverse move is prohibited for a certain number of iterations. This time span called *prohibition period* T is adapted during the run of the algorithm. The algorithm keeps track of the last time step in which a given configuration was reached. The parameter T is increased if frequent configuration cycles occur. It is decreased if the cycles occur seldom.

Moreover, the algorithm is restarted from a random node if the objective value of the best configuration found so far has not improved after a certain number of iterations.

The main function of the RLS algorithm is described in Algorithm 5.3. It starts with an initialization of the needed variables, i.e., the iteration counter t, the time of the last restart t_R, the currently considered stable set S, the best stable set S^* found so far, the time step T^* at which S^* was found, the prohibition period T, and the last time step t_T at which T was changed.

In the main loop, the function MemoryReaction(S, T) adapts the prohibition period T based on the current stable set S and the history. Then the algorithm performs a local search step by replacing S by its best neighbor. If a new best solution is found, it is stored in S^* and the corresponding time step t^* is updated. The local search is restarted if a sufficiently large number of iterations have passed since the last restart or improvement.

The main loop can be left at any time, e.g., if the best found stable set S^* is acceptable or a given runtime has been exceeded.

Algorithm 5.3 Reactive Local Search (RLS) algorithm

RLS $(G(V, E))$
1: $t \leftarrow 0$ ▷ iteration counter
2: $t_R \leftarrow 0$ ▷ time of last restart
3: $S \leftarrow \emptyset$ ▷ current stable set
4: $S^* \leftarrow \emptyset$; $t^* \leftarrow t$ ▷ best stable set so far
5: $T \leftarrow 1$; $t_T \leftarrow t$ ▷ prohibition period
6: // initialize additional data structures
7:
8: **repeat**
9: $T \leftarrow$ MEMORYREACTION(S, T)
10: $S \leftarrow$ BESTNEIGHBOR(S)
11: $t \leftarrow t + 1$
12: **if** $|S| > |S^*|$ **then**
13: $S^* \leftarrow S$; $t^* \leftarrow t$
14: **end if**
15: **if** $t - \max\{t^*, t_R\} > \Delta_1$ **then**
16: $t_R \leftarrow t$; RESTART()
17: **end if**
18: **until** $|S^*|$ is acceptable **or** runtime exceeded
19: **return** S^*

Before we turn to the implementation of BESTNEIGHBOR(S), we need to introduce some notation. Given a stable set $S \subseteq V$, we define

$$C := \{v \in V \setminus S \mid \forall u \in S : (u, v) \notin E\}, \tag{5.10}$$

i.e., the set C is the set of nodes (*candidates*) from $V \setminus S$ that can be added to S such that S still is a stable set. Furthermore, for each node $v \in C$ we define

$$\Delta C[v] := |\{u \in V \setminus C \mid (u, v) \in E,\ \forall w \in S \setminus \{v\} : (u, w) \notin E\}|\,, \tag{5.11}$$

i.e., the quantity $\Delta C[v]$ denotes the number of nodes that can be added to C if the node v is dropped from S. In other terms, it holds

$$S^{(t+1)} = S^{(t)} \setminus \{v\} \Rightarrow \Delta C^{(t)}[v] = |C^{(t+1)}| - |C^{(t)}|, \tag{5.12}$$

where the superscript indices denote the corresponding iteration.

Note the parallelism between lines 1–5 (adding a node to S) and lines 7–11 (dropping a node from S) in the implementation of BESTNEIGHBOR(S) (see Algorithm 5.4). First we compute the subset A of candidates that have not been moved (i.e., added or dropped) in the last T iterations. If $A \neq \emptyset$, we select a node v that will be later added to S. The node v is chosen

Algorithm 5.4 Function BestNeighbor of the RLS algorithm

```
BestNeighbor (S)
 1: A ← {v ∈ C | t > last_moved[v] + T}
 2: if A ≠ ∅ then                                   ▷ if possible add node
 3:     type ← AddVertex
 4:     m ← min{deg_C[v] | v ∈ A}
 5:     v ← random element of {v ∈ A | deg_C[v] = m}
 6: else                                            ▷ else drop node
 7:     type ← DropVertex
 8:     D ← {v ∈ S | t > last_moved[v] + T}
 9:     if D ≠ ∅ then
10:         M ← max{ΔC[v] | v ∈ D}
11:         v ← random element of {v ∈ D | ΔC[v] = M}
12:     else
13:         v ← random element of S                ▷ S ≠ ∅
14:     end if
15: end if
16: S ← IncrementalUpdate(v, type)
17: return S
```

uniformly at random among those nodes of A that have minimum degree in the subgraph induced by C. If $A = \emptyset$, we compute the subset D of nodes from S that have not been moved in the last T iterations. If $D \neq \emptyset$, we select a node v that will later be removed from S. The node v is chosen uniformly at random among those nodes of D for which $\Delta C[v]$ is maximal. If $D = \emptyset$, the node v is chosen uniformly at random from S.

Finally, the function IncrementalUpdate$(v, type)$ is called. This function updates the sets S and C, as well as the quantities $last\_moved[v]$, $\deg_C[v]$, and $\Delta C[v]$ (note that the latter two quantities need also to be updated for nodes different from the selected node v).

We remark that $S \neq \emptyset$ in line 13 holds. The assumption $S = \emptyset$ implies $C = V$. The implementation of the function MemoryReaction(S, T) enforces $T \leq 2(|V| - 1)$, hence $A \neq \emptyset$. Thus a node for addition is selected and line 13 not reached.

The function MemoryReaction(S, T) controls the prohibition period T based on the frequency of configuration cycles during the iteration. For each stable set we store the iteration counter of its last occurrence in a hash table. Actually, for efficiency reasons, we do not use the stable sets S as keys in the hash table but the values $h :=$ Hash(S) of a suitable hash function Hash(). Experiments have shown that collisions due to this simplification are very rare and do not cause significant changes of the parameter T.

The implementation of the function MemoryReaction(S, T) is shown in Algorithm 5.5. The hash key h for the current stable set S is looked up

Algorithm 5.5 Function MEMORYREACTION of the RLS algorithm

```
MEMORYREACTION (S, T)
 1: h ← HASH(S)
 2: if hash table HT contains h then
 3:     Δt ← t − HT[h]
 4:     if Δt < 2(|V| − 1) then                    ▷ frequent repetitions
 5:         INCREASE(T); t_T ← t                   ▷ increase T
 6:     end if
 7: end if
 8: HT[h] ← t                                      ▷ update hash table
 9: if t − t_T > Δ_2 then                          ▷ few repetitions
10:     DECREASE(T); t_T ← t                       ▷ decrease T
11: end if
12: return T
```

in the hash table. If it is found and the repetition interval Δt is sufficiently short, the prohibition period T is increased (lines 2–6). In all cases, the current iteration counter t is stored with key h in the hash table (line 8). If no cycle occurred in the last Δ_2 iterations, the prohibition period is decreased (lines 9–11). The functions INCREASE(T) and DECREASE(T) are realized as

$$\begin{aligned} \text{INCREASE}(T) &= \min\{\max\{\lfloor 1.1 \cdot T \rfloor, T+1\}, |V|-2\} \quad \text{and} \\ \text{DECREASE}(T) &= \min\{\max\{\lfloor 0.9 \cdot T \rfloor, T-1\}, 1\}. \end{aligned}$$

These functions enlarge respectively reduce T by 10 percent (with a minimum change of one unit), provided the lower and upper bounds of 1 and $|V| - 2$ are not exceeded.

The values of the parameters Δ_1 and Δ_2 depend on the given instance. BATTITI and PROTASI propose to choose $\Delta_1 = 100\,|S^*|$ and $\Delta_2 = 10\,|S^*|$. Experiments show that the results of the algorithm are not affected by small changes in the involved constants.

We remark that the RLS algorithm has a worst-case complexity of $O(|V|+|E|)$ per iteration (provided the graph G is stored as adjacency matrix). Actually, this bound is very pessimistic and the average-case complexity is much better. The algorithm requires $O(|V|^2 + t_{max})$ space. All data structures excluding the hash table need at most $O(|V|^2)$ space. The size of the hash table is linear in the number of iterations. The space requirement of the hash table can be reduced by periodically removing all entries that are no longer needed, i.e., entries older than $2(|V| - 1)$ iterations.

Given the special structure of the graph in our case, the space complexity can be further reduced to $O(|E| + t_{max})$ by representing the graph G by adjacency lists. If the geometric parameters of a box are attached to the corresponding node, an adjacency query for two nodes can still be answered in constant time.

Finally, we want to present a variant of the RLS algorithm. In the BestNeighbor(S) function the node degrees $\deg_C[v]$ in the subgraph induced by C are used to break ties among the nodes in A. The question arises whether these subgraph degrees $\deg_C[v]$ can be replaced by the degrees $\deg_G(v)$ in the original graph G. This simplification alleviates the necessity to maintain a data structure for $\deg_C[v]$ (or to recompute the information when needed). Thereby, the worst-case complexity of an iteration reduces to $O(|V|)$. The experiments in [BP01] suggest that the benefit of this variant depends on the problem instance. We investigate the performance of this variant in our context in Section 6.1.4.

5.1.5 Simplefill

All previously presented algorithms use the graph based representation of the packing problem and solve a stable set problem on the conflict graph. These algorithms are based on well-known techniques, e.g., linear programming or local search. The computed solutions are irregular, i.e., boxes of all orientations are mixed. These solutions have no recognizable structure and are quite different from solutions produced by humans.

The Simplefill algorithm is based on the grid representation of the packing problem and tries to mimic human packing strategies. Often humans fix a particular box orientation and pack the trunk in layers, starting from the trunk floor. Within a layer, they start at the boundary and add boxes one after another while trying to locally minimize wasted space. Boxes in a different orientation are used for regions that cannot be packed reasonably with the previously fixed orientation.

The pseudocode for the Simplefill algorithm is depicted in Algorithm 5.6. Actually, the presented pseudocode is a simplification of the Simplefill algorithm. First we discuss the simplified version. Afterwards we explain the full construction of the Simplefill algorithm.

The outermost loop considers all six box orientations in turn. Let (w, h, d) denote the box extensions that correspond to the orientation of the current iteration. Let further x_{min}, x_{max}, and so on denote the coordinate ranges of the cells in the set I. We use three nested loops to consider each cell in I as a box anchor. If the box with anchor (x, y, z) and orientation (w, h, d) can be packed, it is added to the set S. To evaluate the predicate "box can be packed", it is necessary to check whether all cells covered by this box are in I and whether is does not intersect any boxes in S. If the box (x, y, z, w, h, d) is packed, the next $d - 1$ iterations of the innermost loop variable z can be skipped, since the corresponding boxes would intersect the just packed box. At the end, the set S of all packed boxes is returned.

Experiments show that in our instances a very high fraction of about 90% of the boxes of a solution are packed in the first iteration of the outermost

Algorithm 5.6 Simplefill algorithm (simplified)

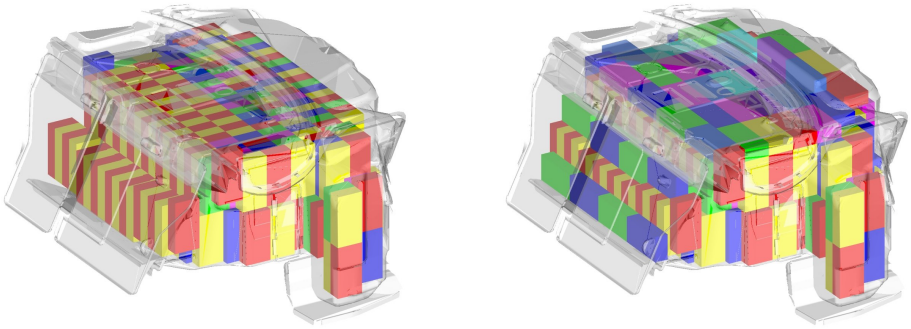
```
SIMPLEFILL (G, n, I)
 1: S ← ∅
 2: for all six orientations o do
 3:     let (w, h, d) denote the box extensions corresponding to o
 4:     for x := x_min to x_max do
 5:         for y := y_min to y_max do
 6:             for z := z_min to z_max do
 7:                 if box (x, y, z, w, h, d) can be packed then
 8:                     add box (x, y, z, w, h, d) to S
 9:                     z ← z + (d − 1)                    ▷ speed-up
10:                 end if
11:             end for
12:         end for
13:     end for
14: end for
15: return S
```

loop. In other words, the orientation considered first dominates the solution. Subsequent iterations contribute, if at all, only very few boxes. A packing computed by the Simplefill algorithm is shown in Figure 5.1.

Now we want to discuss the construction of the Simplefill algorithm, based on the simplified version shown in Algorithm 5.6. This pseudocode does not specify the order in which the box orientations are considered. As stated above, the orientation of the first iteration has a strong influence on the structure of the solution. Furthermore, the order of the three inner loops has been chosen arbitrarily. In the end, the direction of the loop variables x, y and z can be —independently of each other— reversed. Thus there are $6! \cdot 3! \cdot 2^3 = 34560$ equal variations of the shown algorithm. Experiments show that there is a large deviation in the solution cardinality among all these variations. Since the depicted algorithm has a very low runtime, the Simplefill algorithm actually repeats the given pseudocode 34560 times, taking into account all permutations of the orientation order, all permutations of inner loop variables and all combinations of inner loop directions. The optimization in line 9 needs to be adjusted accordingly. Thereby, the cardinality of the solution is significantly increased, while the overall runtime is still low compared to other algorithms presented.

5.2 Divide-and-Conquer Approaches

We investigate some divide-and-conquer approaches as a response to drawbacks of the direct approaches. Due to their runtime and space requirements,



(a) after the first orientation (b) after the sixth orientation

Figure 5.1: Packing computed by the Simplefill algorithm. The majority of the boxes has the orientation that is considered in the first iteration.

the ILP, LPR and RLS algorithms are impracticable for larger instances. On the other hand, the ILP algorithm is well suited for smaller instances, which can be optimally solved.

With exception of the Simplefill algorithm, all direct approaches produce solutions that do not exhibit any structure. Boxes in all six orientations are mixed, apparently without any plan. Moreover, the set of uncovered cells is scattered over the whole trunk. If uncovered cells are near to the boundary, a slight modification of the trunk geometry and a local rearrangement of boxes might make use of these regions. In contrast, the space of uncovered cells in the center of the packing is definitively lost.

These observations are the motivation for two divide-and-conquer approaches called *Matching* and *Easyfill*. Both algorithms use a simple geometry-guided heuristic to produce a tight partial packing with a clear structure. A direct approach is used to solve the remaining, smaller subproblems.

A third heuristic called *Partition* splits the set of available grid cells into disjoint subsets. We take care of the waste that arises from breaking the original problem into subproblems. Although the technique is general and can be used together with other direct approaches, we designed it particularly with the ILP algorithm in mind.

Due to the nature of the divide-and-conquer approaches, it is —in contrast to the direct approaches— not obvious how to enforce a bounded runtime. Therefore, we explicitly discuss this issue for each heuristic. However, for clarity, the respective statements are omitted from the pseudocode descriptions.

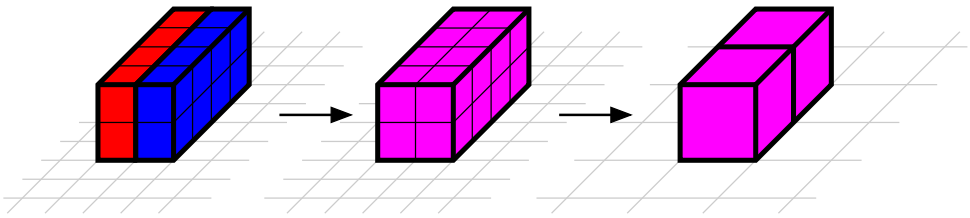


Figure 5.2: Construction of boxes used by the Matching algorithm. Two boxes consisting of $4n \times 2n \times n$ cells are glued together to a single box of $4n \times 2n \times 2n$ cells. This new object can be interpreted as a box of $2 \times 1 \times 1$ cells on a grid with $l = 100$mm spacing.

5.2.1 Matching

This algorithm computes maximum matchings to solve subproblems, hence the name Matching algorithm. The main idea originates from the 2×1-Rectangle-Packing problem, which can be solved in polynomial time using matching techniques (see Section 2.2). It is straightforward to extend this approach to three-dimensional space and to apply it to the Discrete-Box-Packing problem for boxes of size $2 \times 1 \times 1$.

The original packing problem for boxes of size $4n \times 2n \times n$ can be transformed into a packing problem for boxes of size $2 \times 1 \times 1$ as follows (see also Figure 5.2). A pair of boxes consisting of $4n \times 2n \times n$ cells is glued together such that it forms a box consisting of $4n \times 2n \times 2n$ cells. This larger box can be interpreted as a box consisting of $2 \times 1 \times 1$ cells on a grid $\mathcal{G}'$ with spacing $l' = 100$mm.

Note that each cell of the new grid $\mathcal{G}'$ consists of $8n^3$ cells of $\mathcal{G}$. Consequently, there are $8n^3$ different ways to align the grid $\mathcal{G}'$ with the cells of $\mathcal{G}$. This freedom results in not only one, but in $8n^3$ matching problems.

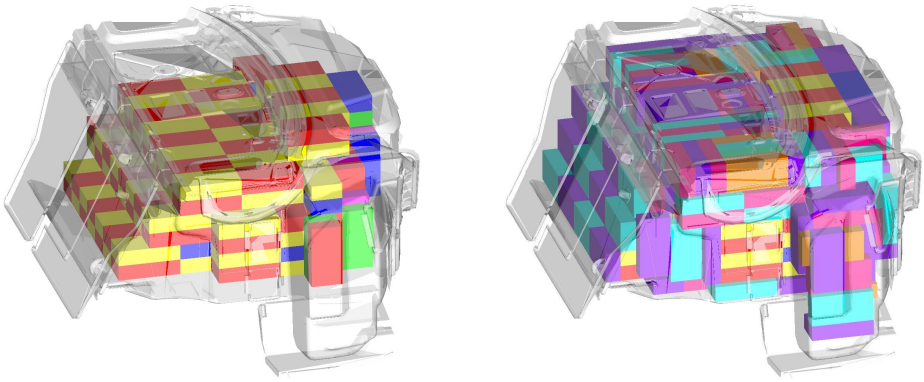
The pseudocode of the Matching algorithm is described in Algorithm 5.7. It consists of two phases. The first phase (lines 1–13) computes partial packings of boxes based on the previously outlined idea of maximum matchings. In the second phase (line 14), the resulting subproblems are solved using a direct algorithm, the results are combined and the best packing is returned. A packing computed by the Matching algorithm is shown in Figure 5.3.

The first phase starts with the initialization of the list L. This list is used to store the partial packings computed in the first phase.

Three nested loops are used to iterate over all different ways to align the coarse grid $\mathcal{G}'$ with the original grid $\mathcal{G}$ (line 5). In each iteration we compute the set I', which denotes the inner approximation of the trunk with respect to the coarse grid $\mathcal{G}'$ (line 6). The set I' contains all cells of $\mathcal{G}'$ that are completely covered by the cells in I. Next we compute the graph $G' = (V', E')$ as follows (line 7). For each cell $c' \in I'$ there is a node $v_{c'} \in V'$. Two nodes are adjacent iff the corresponding cells are adjacent. Thus an edge

Algorithm 5.7 Matching algorithm

MATCHING $(\mathcal{G}, n, I)$
1: $L \leftarrow$ empty list ▷ phase 1
2: **for** $x := 0$ **to** $2n - 1$ **do**
3: **for** $y := 0$ **to** $2n - 1$ **do**
4: **for** $z := 0$ **to** $2n - 1$ **do**
5: $\mathcal{G}' \leftarrow$ grid with spacing 100mm aligned with cell (x, y, z) of $\mathcal{G}$
6: $I' \leftarrow \{c' \in \mathcal{G}' \mid$ cell c' is covered by cells in $I\}$
7: $G' \leftarrow$ graph induced by I'
8: $M' \leftarrow$ maximum matching of G'
9: $S' \leftarrow \{(b_1, b_2) \mid$ boxes b_1 and b_2 correspond to an edge in $M'\}$
10: L.append (S')
11: **end for**
12: **end for**
13: **end for**
14: $S \leftarrow$ SOLVEANDCOMBINE$(\mathcal{G}, n, I, L)$ ▷ phase 2
15: **return** S



(a) after the first phase (b) after the second phase

Figure 5.3: Packing computed by the Matching algorithm. Note that the boxes packed in the first phase always come in pairs. The Greedy algorithm was used for the second phase.

of G' corresponds to a box of size $2 \times 1 \times 1$. Observe that the graph G' is different from the conflict graph (see Section 4.1.3) used in other algorithms. Actually, in the special case of boxes of size $2 \times 1 \times 1$, the conflict graph is the line graph of G'.

Afterwards, we compute a maximum matching of G' and transform it back to the original problem (lines 8–9). Each edge in the matching corresponds to a single box of size $2 \times 1 \times 1$ on the coarse grid $\mathcal{G}'$, and hence to a pair of boxes on the original grid $\mathcal{G}$. Note that there is an ambiguity with respect to the latter. There are always two such pairs of boxes that correspond to an edge in G'. For example, the pairs $((x, y, z, 4n, 2n, n), (x, y, z+n, 4n, 2n, n))$ and $((x, y, z, 4n, n, 2n), (x, y+n, z, 4n, n, 2n))$ correspond to the same edge in G'. This ambiguity can be resolved by restricting the set of feasible box orientations, e.g., to the subset of cyclic shifts of $(4n, 2n, n)$. Finally, the partial packing S' is stored in L (line 10).

The second phase of the algorithm is implemented in the function SolveAndCombine$(\mathcal{G}, n, I, L)$ (see Algorithm 5.8). This function realizes a generic framework for handling a set of partial packings. It iterates over a list L of stable sets S' which represent a partial packing of $(\mathcal{G}, n, I)$. For each stable set S', we first compute the set I'' of still uncovered cells of I (line 3). Then the framework calls the function DirectApproach to solve the remaining subproblem $(\mathcal{G}, n, I'')$ (line 4). The name DirectApproach is used as a placeholder for one of the algorithms described in Section 5.1. Finally, the solution S'' of the subproblem is combined with the previously computed partial packing S' (line 5). The union of S' and S'' represents a solution of the original packing problem. The framework returns a solution with highest cardinality.

Algorithm 5.8 Generic framework for a set of partial packings

SolveAndCombine $(\mathcal{G}, n, I, L)$

```
 1: S* ← ∅
 2: for all partial packings S' in L do
 3:     I'' ← I \ {cells covered by boxes in S'}        ▷ compute subproblem
 4:     S'' ← DirectApproach(G, n, I'')                 ▷ solve subproblem
 5:     S ← S' ∪ S''                                    ▷ combine partial packings
 6:     if |S| > |S*| then
 7:         S* ← S
 8:     end if
 9: end for
10: return S*
```

We remark that the graph G' is a bipartite graph. Therefore, one can use an algorithm specialized for bipartite matchings to solve the matching problem more efficiently.

A Modification

We want to describe a modification which significantly increases the cardinality of the computed packings. The final packings S that are computed in the function SolveAndCombine$(\mathcal{G}, n, I, L)$ often are inferior. Even if a subproblem of the second phase is optimally solved, the combined packing S can be easily improved. Often it is possible to locally rearrange a few boxes (including some of S') and to add an additional box. The reason for this phenomenon lies in the small size, but rather difficult shape of the subproblems of the second phase. There is simply not enough flexibility to compute a good packing.

To overcome this problem we propose the following modification: At the beginning of the first phase, we remove some of the outmost layers of the cells in the set I. At the end of the first phase, the original value of I is restored. Thus the first phase of the algorithm is restricted to the core of the set I and the boundary is reserved for the second phase. This modification increases the flexibility in the second phase and leads to better overall packings. The benefit of this modification is experimentally studied in Section 6.1.5.

Alternatively, one could also remove the outmost layers of the boxes that are packed in the first phase. However, the previously presented method has the advantage that it allows a finer control.

Bounded Runtime

Finally, we want to discuss how to modify the algorithm in case of a bounded overall runtime. The Matching algorithm consists —in contrast to previously discussed algorithms— of two phases and several subproblems. The question arises how to distribute the available runtime among the phases and subproblems.

An immediate observation is that only a very small amount of time is spent in the first phase. Instances of typical size can be handled in about one minute. The vast majority of the overall runtime is used for the second phase of the algorithm. Thus it is not reasonable to further limit the time spent in the first phase, either by restricting the set of subproblems that are considered or by limiting the time for each subproblem.

This leaves us with the question how to allocate the remaining time to the subproblems of the second phase. Experiments show that in general there is no correlation between the size $|S'|$ of the partial packing of the first phase and the size $|S|$ of the final packing. In particular, the maximum matching among all iterations does in general not lead to the best packing after the second phase. Thus it is difficult to restrict the second phase to a subset of promising partial packings.

On the other hand, the size $|S'|$ of a partial packing of the first phase correlates with the problem complexity of the subproblem remaining for the

second phase. The smaller $|S'|$, the larger the number $|I''|$ of uncovered cells, and hence the larger the problem complexity and required runtime. Moreover, experiments show that there is a large variance in the cardinality of S', and hence also in the runtime of the subproblems of the second phase.

Therefore we propose the following heuristic: The function SolveAndCombine$(\mathcal{G}, n, I, L)$ is modified such that it first sorts the partial packings in the list L by their cardinality in non-increasing order. When using algorithms with a high runtime, e.g., the ILP or RLS algorithm, we also need to impose time limits on the individual subproblems. Whenever a subproblem is started, we set its time limit as the total remaining runtime divided by the number of remaining subproblems.

This strategy ensures that cheap subproblems, i.e., those with a runtime below the fair share of the total runtime, are not omitted due to time constraints. The difference between the time that was allocated to and the time that was actually spent by cheap subproblems can be used for more expensive subproblems later. Moreover, single subproblems with an unexpected high runtime do not disrupt the processing of the following subproblems.

5.2.2 Easyfill

The Easyfill algorithm is motivated by the following simple strategy. Suppose there is an initial box with anchor (x, y, z) and orientation (w, h, d). If possible, pack boxes with the same orientation and anchors $(x + w, y, z)$, $(x, y + h, z)$ and $(x, y, z + d)$. The repetition of this pattern leads to a tight packing with a regular structure. All boxes in such a packing have the same orientation.

The pseudocode of the Easyfill algorithm is shown in Algorithm 5.9. Its structure is similar to the structure of the Matching algorithm. Both algorithms consist of two phases. The first phase of the Easyfill algorithm generates a set of partial packings which are passed to the second phase. The second phase completes the partial packings and returns the best packing. This phase is identical to the Matching algorithm.

The first phase starts with an initialization of the list L, which stores the partial packings. We consider each of the six possible box orientations in turn (line 2). Let (w, h, d) denote the box extensions that correspond to the current orientation. Next we consider all anchor cells (x, y, z) that result in a unique packing (lines 4–6). This set of anchor cells equals the set of integer triples in $[0, w) \times [0, h) \times [0, d)$. The partial packing S' consist of all boxes anchored at (i, j, k) with orientation (w, h, d) such that only cells in I are covered and $i \in x + w\mathbb{Z}$, $j \in y + h\mathbb{Z}$ and $k \in z + d\mathbb{Z}$. Finally, the packing S' is stored in L (line 8).

The second phase is identical to the Matching algorithm and is implemented by the function SolveAndCombine$(\mathcal{G}, n, I, L)$ (see Algorithm 5.8). Similarly, the remarks about a modification and the bounded runtime of the

Algorithm 5.9 Easyfill algorithm

```
EASYFILL (G, n, I)
 1: L ← empty list                                              ▷ phase 1
 2: for all six orientations o do
 3:     let (w, h, d) denote the box extensions corresponding to o
 4:     for x := 0 to w − 1 do
 5:         for y := 0 to h − 1 do
 6:             for z := 0 to d − 1 do
 7:                 S' ← {(i, j, k, w, h, d) ∈ ℤ^6 | the box (i, j, k, w, h, d) covers
                           only cells in I and i ∈ x+wℤ, j ∈ y+hℤ, k ∈ z+dℤ}
 8:                 L.append (S')
 9:             end for
10:         end for
11:     end for
12: end for
13: S ← SOLVEANDCOMBINE(G, n, I, L)                             ▷ phase 2
14: return S
```

Matching algorithm also apply to the Easyfill algorithm. Figure 5.4 shows a packing computed by the Easyfill algorithm.

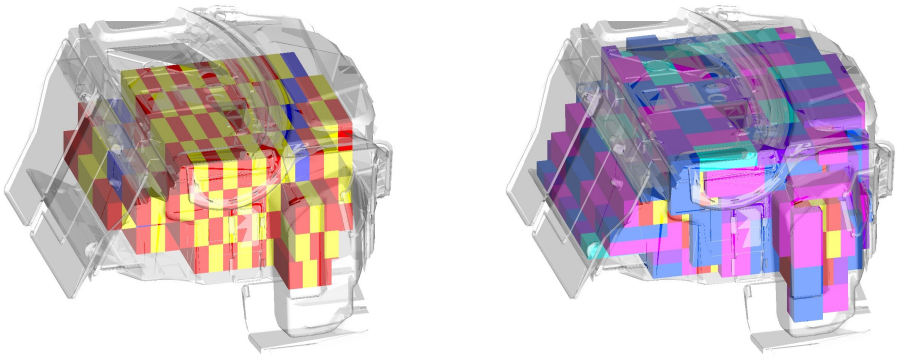
Although the structure of the Matching and Easyfill algorithm is similar, there are some differences worth mentioning. The first phase of the Easyfill algorithm generates $6 \cdot 4n \cdot 2n \cdot n = 48n^3$ subproblems, whereas the Matching algorithm generates only $8n^3$ subproblems. This difference results from the fact that the Easyfill algorithm handles each of the six possible box orientations in a separate subproblem. In contrast, the Matching algorithm handles different box orientations simultaneously in a single subproblem. The larger number of subproblems increases the runtime of the Easyfill algorithm compared to the Matching algorithm. Conversely, if the overall runtime is bounded, the runtime that is available for each subproblem is reduced.

5.2.3 Partition

The idea of this algorithm is to partition the interior of the trunk into smaller regions. For each region the resulting packing problem can be solved independently. Finally, the packings of all regions are combined into a solution for the original problem.

This idea is motivated by the ILP algorithm, which computes —in contrast to all other presented algorithms— optimal solutions for the DISCRETE-BOX-PACKING problem. Because its runtime increases sharply with the problem size, the algorithm is practicable only for small problem instances.

We use planes that are perpendicular to the axes of the given grid to partition the set I into subsets. The position of these cutting planes is



(a) after the first phase (b) after the second phase

Figure 5.4: Packing computed by the Easyfill algorithm. Note that all boxes packed in the first phase have the same orientation. The Greedy algorithm was used for the second phase.

chosen such that the emerging subproblems have roughly the same size. The number of cutting planes depends on the problem instance and the algorithm that is used to solve the subproblems. For example, the ILP algorithm works quite well for subproblems of 50 to 100 liters (with a grid of 25mm spacing).

The pseudocode of the Partition algorithm is depicted in Algorithm 5.10. In a preliminary step we calculate the extension of I, i.e., the coordinate ranges in all three dimensions (lines 1–6). Afterwards, we compute the position of the cutting planes (lines 8–16). The parameters k_x, k_y and k_z denote the number of cutting planes perpendicular to the x-, y- and z-axis, respectively. The cutting planes are approximately equally spaced within the corresponding coordinate range.

The algorithm initializes the sets S and I_r as empty sets. The set S stores the partial packings computed so far. The set I_r holds cells that have been considered in previous iterations, but are not covered by any boxes in S.

Three nested loops are used to iterate over all subproblems, which are identified by the triple (i, j, k) of the corresponding loop variables. In each iteration, we compute the set $I_{i,j,k}$ of cells to be considered. The set $I_{i,j,k}$ is the union of the previously considered, but uncovered cells I_r and the subset of I that falls in the currently considered region $[x_i, x_{i+1}) \times [y_j, y_{j+1}) \times [z_k, z_{k+1})$. We use one of the direct approaches described in Section 5.1 to solve the packing problem $(\mathcal{G}, n, I_{i,j,k})$. Finally, the solution $S_{i,j,k}$ of this subproblem is added to the set S and the set I_r of previously considered, but uncovered cells is updated.

Algorithm 5.10 Partition algorithm

PARTITION $(\mathcal{G}, n, I)$

1: $x_{min} \leftarrow \min\{x \mid \exists\, y, z \in \mathbb{Z} : (x, y, z) \in I\}$ ▷ compute extension of I
2: $x_{max} \leftarrow \max\{x \mid \exists\, y, z \in \mathbb{Z} : (x, y, z) \in I\}$
3: $y_{min} \leftarrow \min\{y \mid \exists\, x, z \in \mathbb{Z} : (x, y, z) \in I\}$
4: $y_{max} \leftarrow \max\{y \mid \exists\, x, z \in \mathbb{Z} : (x, y, z) \in I\}$
5: $z_{min} \leftarrow \min\{z \mid \exists\, x, y \in \mathbb{Z} : (x, y, z) \in I\}$
6: $z_{max} \leftarrow \max\{z \mid \exists\, x, y \in \mathbb{Z} : (x, y, z) \in I\}$
7:
8: **for** $i := 0$ **to** $k_x + 1$ **do** ▷ compute position of cutting planes
9: $\quad x_i \leftarrow x_{min} + \left\lfloor \frac{i}{k_x+1}(x_{max} - x_{min} + 1) \right\rfloor$
10: **end for**
11: **for** $j := 0$ **to** $k_y + 1$ **do**
12: $\quad y_j \leftarrow y_{min} + \left\lfloor \frac{j}{k_y+1}(y_{max} - y_{min} + 1) \right\rfloor$
13: **end for**
14: **for** $k := 0$ **to** $k_x + 1$ **do**
15: $\quad z_k \leftarrow z_{min} + \left\lfloor \frac{k}{k_z+1}(z_{max} - z_{min} + 1) \right\rfloor$
16: **end for**
17:
18: $S \leftarrow \emptyset$
19: $I_r \leftarrow \emptyset$
20: **for** $i := 0$ **to** k_x **do**
21: $\quad$ **for** $j := 0$ **to** k_y **do**
22: $\quad\quad$ **for** $k := 0$ **to** k_z **do**
23: $\quad\quad\quad I_{i,j,k} \leftarrow I_r \cup \big(I \cap [x_i, x_{i+1}) \times [y_j, y_{j+1}) \times [z_k, z_{k+1})\big)$
24: $\quad\quad\quad S_{i,j,k} \leftarrow$ DIRECTAPPROACH$(\mathcal{G}, n, I_{i,j,k})$
25: $\quad\quad\quad S \leftarrow S \cup S_{i,j,k}$
26: $\quad\quad\quad I_r \leftarrow I_{i,j,k} \setminus \{\text{cells covered by boxes in } S_{i,j,k}\}$
27: $\quad\quad$ **end for**
28: $\quad$ **end for**
29: **end for**
30: **return** S

No restriction on the time bound for subproblems seems unfavorable, because this might lead to a situation in which some subproblems are never considered, and thus the solution is far from optimal. If the subproblems are solved independently, an approach similar to the case of the Matching and Easyfill algorithm is possible (see Section 5.2.1), i.e., reordering the subproblems by non-decreasing problem complexity and distributing the remaining runtime equally among the remaining subproblems. But solving subproblems independently has a serious drawback (which is discussed in the next paragraph). The proposed solution does not work well with an arbitrary order of subproblems. Therefore, we abandon the idea of reordering subproblems. In each iteration, we simply distribute the remaining runtime among the remaining subproblems.

Note that the subdivision of the original problem into subproblems has a serious drawback with respect to the quality of the final solution. Even if all subproblems are solved to optimality, the combination of these solutions is in general not optimal. Often it is possible to improve the cardinality of the obtained solution by local optimization across region boundaries. Consider the grid depicted in Figure 5.5(a). A vertical cutting plane cuts this grid into two regions. Any optimal solution of the given grid consists of nine boxes. Assume that the subproblems on the left and on the right are solved independently, i.e., I_r is fixed to the empty set. Any such solution is suboptimal since it consists of at most eight boxes.

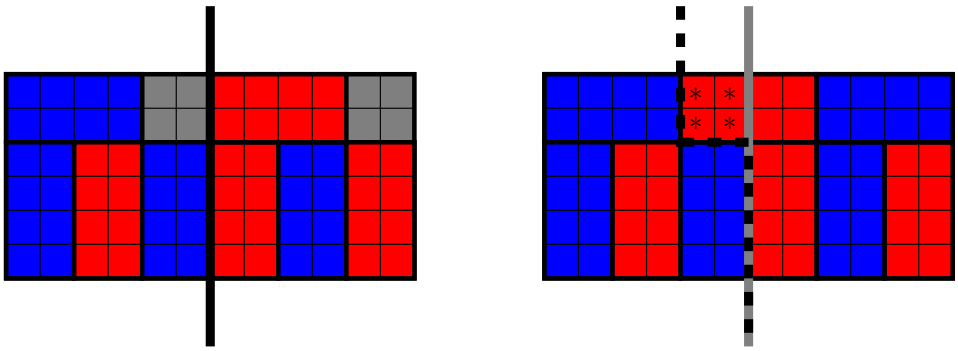
The introduction of the set I_r helps to avoid suboptimal situations. The subproblems are no longer solved independently. Instead, later handled subproblems are aware of uncovered cells of previous iterations. Figure 5.5(b) depicts the same situation as before. Suppose that the left subproblem is solved first and the cells marked with an asterisk remain uncovered. The set I_r adds these cells to the right subproblem, which has a packing of five boxes. Hence the overall solution consists of nine boxes.

To make this work, it is crucial that uncovered cells are located next to unprocessed regions. For example, the uncovered cells of the left subproblem in Figure 5.5(c) cannot be used by the right subproblem. Next we discuss a method to influence the position of uncovered cells.

Weighted Stable Set Problems

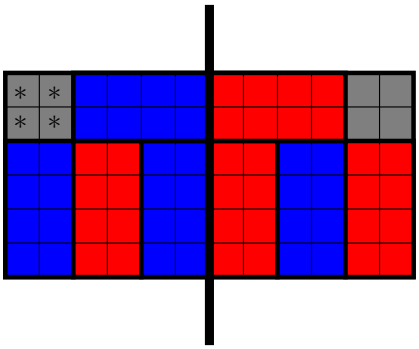
Up to now, we were facing a stable set problem. The goal was to compute a stable set with a cardinality as high as possible. All boxes were treated equally, and hence we did not have any preferences among stable sets of the same cardinality.

Now the objective changes. We still want to compute stable sets with high cardinality, but stable sets of the same cardinality are no longer treated equally. Certain box positions are preferred over other positions. This modified problem can be represented as a *weighted stable set* problem.



(a) Suboptimal solution

(b) Improved solution



(c) No improvement possible

Figure 5.5: Impact of the position of uncovered cells. Any optimal solution for the depicted grid consists of nine boxes. Independent processing of the two regions leads to suboptimal packings of at most eight boxes (a). The cardinality of a packing can be increased if uncovered cells (marked with (*)) of previously processed regions are are taken into account (b). However, this is not always possible due to the position of the uncovered cells (c).

Definition 5.10 (Maximum Weighted Stable Set–optimization variant). *Given a graph $G = (V, E)$ and a weight function $c : V \to \mathbb{R}$, compute a stable set $S \subseteq V$ that maximizes $\sum_{v \in S} c(v)$.*

We describe how to adapt the ILP algorithm to the maximum weighted stable set problem. The LPR algorithm can be adapted in a similar way. Other algorithms, e.g., the Greedy, RLS, or Simplefill algorithm can be modified for the weighted case by incorporating the node weights as a tie-breaking rule.

We replace the objective function $\sum_{v \in V} x_v$ in the complete clique formulation (5.3) by $\sum_{v \in V} c_v x_v$. The weights $c_v \in \mathbb{R}$ are computes as follows: Let (x_v, y_v, z_v) denote the anchor of the box that corresponds to the node $v \in V$. Let further x_{min}, x_{max}, and so on denote the coordinate ranges of the cells in the current subproblem $I_{i,j,k}$. Then, the weight c_v is computed as

$$c_v = 1 + \frac{1}{u_{i,j,k}} \cdot \frac{(x_{max} - x_v) + (y_{max} - y_v) + (z_{max} - z_v)}{(x_{max} - x_{min}) + (y_{max} - y_{min}) + (z_{max} - z_{min})}, \quad (5.13)$$

where $u_{i,j,k}$ denotes an upper bound for the maximum number of boxes for this subproblem. For example, the trivial bound $\lfloor \frac{|I_{i,j,k}|}{8n^3} \rfloor$ can be used. Note that the right factor is bounded by 0 and 1 from below and above, respectively. The scaling factor $\frac{1}{u_{i,j,k}}$ ensures that solutions with a higher cardinality always have a higher objective value. Among solutions with the same cardinality, the objective function prefers those where the boxes are anchored as near as possible to the cell $(x_{min}, y_{min}, z_{min})$. In other words, it prefers solutions where the uncovered cells are located next to the planes $x = x_{max}$, $y = y_{max}$ and $z = z_{max}$. These planes are the boundaries to adjacent subproblems that are handled in later iterations of the Partition algorithm. Thus the algorithm can make use of the uncovered cells that are lost otherwise. This is also the reason why we do not reorder the subproblems. The order in which the subproblems are considered has to correspond to the weights c_v.

Observe that the introduction of the weights c_v strongly increases the running time of the ILP algorithm. In the unweighted case, the coefficients c_v are implicitly fixed to 1 and the objective vector $\boldsymbol{c} := (c_v)_{v \in V} \in \mathbb{R}^{|V|}$ is the all-ones vector. In the weighted stable set problem the objective vector $\boldsymbol{c}$ is slightly perturbed as depicted in (5.13). This modification increases the bit-complexity of the problem description. Moreover, objective values are no longer integral, but fractional. This imposes an additional burden during the branch-and-cut phase. Previously, in the unweighted case, subtrees with an LP value less than the cardinality of the best solution found so far plus 1.0 can be pruned in the branch-and-cut phase. Such an optimization is no longer possible in the weighted case.

Chapter 6

Experimental Results

In this chapter we give an experimental evaluation of the presented algorithms. All instances are based on real-world data sets provided by our industrial partner.

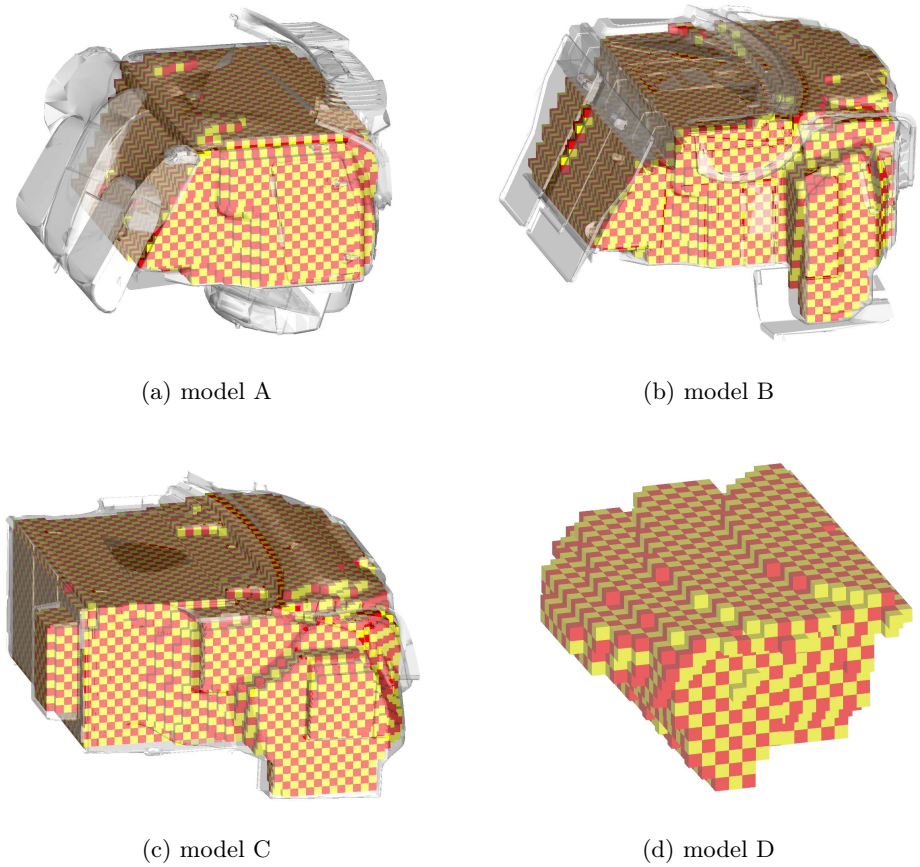
Our software package is implemented in C++. It is platform-independent and has been tested under Linux, Solaris and Windows. We use several third-party libraries, namely LEDA 4.5 [MN99, Alg], CPLEX 9.0 [ILO03b, ILO03a, ILO], Qt 3.3.5 [BS04, Tro] and OpenGL [WNDS01, Ope]. We use several data structures offered by LEDA throughout our algorithms, in particular, we use graphs, hash tables, priority queues, lists, and tuples. We also use its shortest path and matching algorithms. CPLEX is used as an (I)LP solver by our ILP and LPR algorithm. The graphical user interface is realized with the Qt toolkit and uses OpenGL for visualization.

All experiments were run on a SunFire 15k machine with 40 UltraSparc III processors at 900 MHz clock speed. Our implementation is single-threaded and does not make use of multiple CPUs. Note that the given runtimes can be reduced by a factor of two to three on modern x86-based hardware. We used the GNU g++ 3.3.5 compiler with the options -O2 and -NDEBUG.

In the first section, we discuss algorithm-specific details, e.g., proposed variants. We continue with a general technique that is helpful to overcome some weaknesses of the discretization process and to improve the results. Finally, we compare all algorithms on a large set of instances.

6.1 Algorithm-Specific Experiments

We report on the results of algorithm-specific experiments. We investigate the influence of parameters and compare proposed variants. The goal is to fine-tune the presented algorithms. We do not yet compare different algorithms. Such a comparison can be found in Section 6.3.



(a) model A (b) model B

(c) model C (d) model D

Figure 6.1: Small set of models. The figures show the shape of the trunk and an inner approximation by a grid with 25mm spacing.[3]

For this purpose, we compiled a small set of typical problem instances. The set consists of four models, which from now on we shall identify as models A, B, C and D. The four models represent instances of typical size. Model A is a convertible with a trunk volume of about 270 liters, models B and C are (middle and upper class) sedans with roughly 400 liters and 500 liters, respectively. Model D is a sports car with a relatively small trunk (about 60 liters). For larger instances of up to 2000 liters we refer the reader to Section 6.3. Screenshots of these four models including a grid with a spacing of 25mm can be seen in Figure 6.1. For comparison, the corresponding expert solutions are shown in Figure 4.1.

Some characteristic values of these models are depicted in Table 6.1. For each model the table lists the value of the expert solution, bounds on the continuous volume, and two upper bounds for the discrete volume with

[3]For legal reasons we may not publish the geometry of model D.

model	expert solution	continuous volume lower bd.	continuous volume upper bd.	spacing [mm]	discrete volume trivial bd.	discrete volume LP bd.
A	272	319.2	353.0	50	276	268
				25	307	281
B	404	450.3	499.2	50	394	389
				25	427	398
C	513	564.3	622.3	50	492	486
				25	536	506
D	62	92.0	101.6	50	62	61
				25	75	64

Table 6.1: Characteristics of selected models

model	spacing [mm]	#nodes	#edges	density	avg. degree	#maximal cliques	avg. size max. cliq.
A	50	8787	649 k	0.0084	147	2195	32.0
	25	68548	62.7 M	0.0134	1830	15733	265.1
B	50	12857	974 k	0.0059	151	3105	33.1
	25	95380	88.7 M	0.0097	1859	21530	269.3
C	50	16358	1257 k	0.0047	153	3925	33.3
	25	126014	120.2 M	0.0076	1907	27832	276.2
D	50	1383	73 k	0.0382	105	495	22.3
	25	10019	6.2 M	0.0612	1225	3227	182.4

Table 6.2: Characteristics of the conflict graphs for selected models

grids of different spacing. The expert solution was constructed by an experienced engineer with a CAD system. The bounds on the continuous volume were obtained using a discretization with a spacing of 12.5mm (models A, B and C) and 6.25mm (model D). The lower bound is given by all cells entirely contained in the trunk, i.e., the set of INSIDE and INSIDE* cells. The upper bound is given by all cells that have non-empty intersection with the trunk, i.e., the set of INSIDE and INSIDE* cells augmented by one additional layer of cells. We also list two upper bounds for the discrete volume, the trivial bound based on the number of INSIDE cells and the LP bound obtained from the LP relaxation. Note that both upper bounds relate to the chosen discretization, in particular they depend on the position and orientation of the chosen grid. We use the chosen grids throughout this section. Therefore, we refer to those grids using the definite article, e.g., "*the* grid with 25mm spacing of model A".

More information about the corresponding conflict graphs can be found in Table 6.2. This table lists the number of nodes and edges of the conflict graph, the graph density, the average node degree, and the number and the average size of the maximal cliques. Note that the graph density decreases with increasing problem size. This is due to the fact the the maximum

model	spacing [mm]	determ. Greedy [l]	randomized Greedy [l]			
			min.	avg.	max.	std. dev.
A	50	260	250	256.0	261	3.61
	25	261	253	257.5	261	2.74
B	50	373	366	372.6	378	3.89
	25	373	365	373.5	382	5.34
C	50	464	457	465.6	473	5.05
	25	475	464	473.5	481	4.34

Table 6.3: Comparison of the deterministic and the randomized Greedy algorithm

degree of a node is bounded from above independently of the problem size (see Section 4.1.3).

6.1.1 Greedy

In this section we want to investigate the behavior of the randomized variant of the Greedy algorithm (see Section 5.1.1). In this variant, all ties are broken at random. We compare this variant with the deterministic Greedy algorithm, in which ties are broken by fixed rules in the implementation of the priority queue datatype `p_queue` in LEDA.

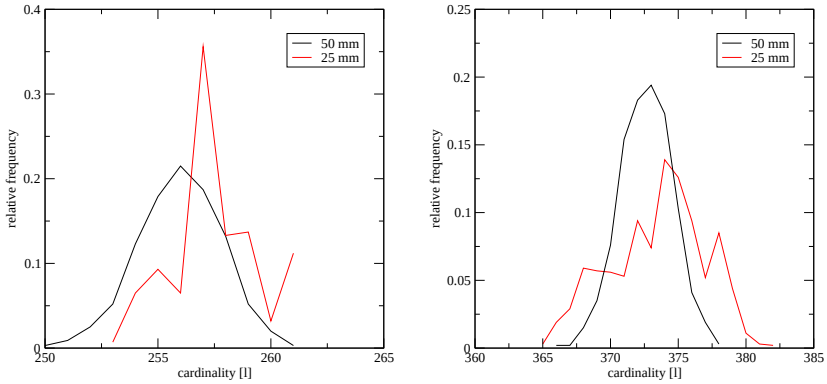
We use the three models A, B, and C, and grids of spacing 50mm and 25mm as examples. Table 6.3 summarizes the results for the deterministic and randomized variant. The third column of the table contains the cardinality of the packing that was obtained by the deterministic Greedy algorithm. The subsequent columns show the minimum, average and maximum cardinality of 1000 runs of the randomized variant. The standard deviation is shown in the last column. The cardinality distribution of the randomized variant is displayed in Figure 6.2.

Surprisingly, the cardinality of the deterministic variant for model A is very close to the maximum cardinality of the randomized variant. In general, the maximum of the randomized variant is significantly higher than the result of the deterministic algorithm. Therefore, we prefer the randomized variant and use several runs to boost the result.

For the remainder of this chapter, the term Greedy algorithm denotes 10 runs of the randomized variant. We report the maximum cardinality found in those runs, as well as the total runtime of all runs. This convention does not only apply to the Greedy algorithm when used stand-alone, but also in conjunction with the Matching or Easyfill algorithm.

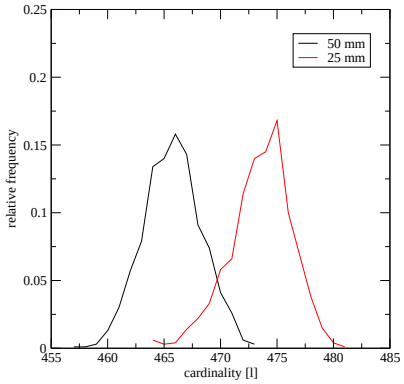
6.1.2 Integer Linear Programming

In this section we discuss the influence of several options offered by CPLEX for mixed integer problems. For example, there are different algorithms



(a) model A

(b) model B



(c) model C

Figure 6.2: Distribution of results of the randomized Greedy algorithm

to solve linear programs. CPLEX is also capable of generating additional constraints for a given problem. The details for all the options mentioned in this section can be found in the CPLEX manuals [ILO03a, ILO03b]. We also discuss the influence of nearly violated odd hole constraints.

We refer to Table 6.4 to get an impression of the size of the integer linear programs. The number of binary variables equals the number of nodes in the conflict graph. The number of constraints equals the number of maximal cliques. The average number of non-zeros in a constraint equals the average size of the maximal cliques.

The LP Algorithm

CPLEX offers several algorithms to solve linear programs. Of particular interest for our instances are the primal simplex, dual simplex and barrier algorithm. It is possible to select the LP algorithm independently for the root relaxation phase and the branch-and-cut phase. We will use the terms

model	spacing [mm]	#variables	#constraints	avg. number of non-zeros
A	50	8787	2195	32.0
	25	68548	15733	265.1
B	50	12857	3105	33.1
	25	95380	21530	269.3
C	50	16358	3925	33.3
	25	126014	27832	276.2
D	50	1383	495	22.3
	25	10019	3227	182.4

Table 6.4: Characteristics of the ILP formulation for selected models

root relaxation algorithm and *branch-and-cut algorithm* to denote the LP algorithm used in the corresponding phase.

We compare the performance of different root relaxation algorithms in Figure 6.3. The diagram shows the time needed to solve the LP relaxation for different models and root relaxation algorithms. The experiments demonstrate that the barrier algorithm is significantly faster than the primal and dual simplex algorithm, in some cases even more than one order of magnitude.

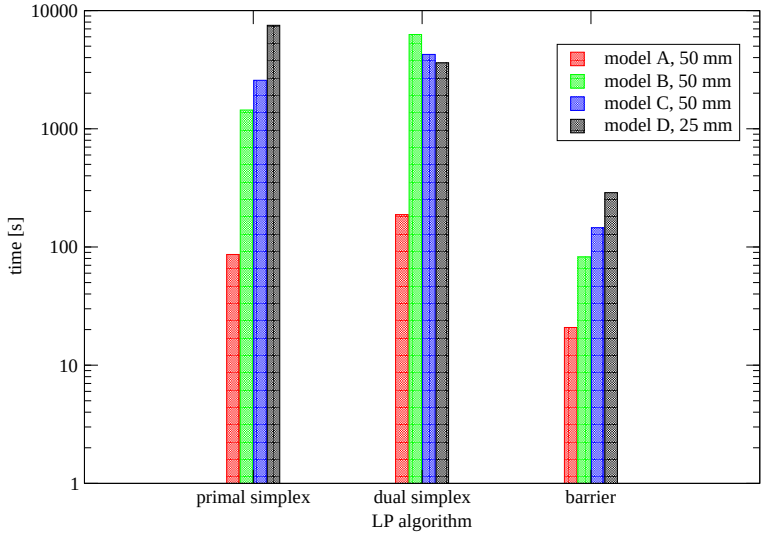


Figure 6.3: Comparison of different root relaxation algorithms. The barrier algorithm is significantly faster than the primal and dual simplex algorithm.

In order to measure the performance of the branch-and-cut algorithm, we consider the size of the largest stable set after a fixed amount of runtime. Table 6.5 shows the results for this experiment after 24 hours of runtime.

model	spacing [mm]	primal simplex	dual simplex	barrier
A	50	266	266	268
B	50	381	379	382
C	50	467	465	476
D	25	61	62	62

Table 6.5: Comparison of different branch-and-cut algorithms. The results of the barrier algorithm are better than or equal to those of the primal and dual simplex algorithm.

In summary, the solutions of the barrier algorithm are better than or equal to those of the primal and dual simplex algorithm. In this experiment, the barrier algorithm has also been used as root relaxation algorithm. If the primal or dual simplex algorithm is used instead, the given numbers decrease slightly or remain equal. The predominance of the barrier algorithm as branch-and-cut algorithms remains unaffected.

Given the above results, from now on we use the barrier algorithm as root relaxation as well as branch-and-cut algorithm.

We would like to remark that the packing problem for model A can be optimally solved with the ILP algorithm in 61 minutes when using the barrier algorithm in both phases. Using a different algorithm (or any combination of two algorithms), CPLEX is not able to find an optimal solution for this model within 24 hours. Likewise, CPLEX does not succeed in finding optimal solutions for models B, C, and D in the same amount of time, no matter which of the three algorithms is used.

Other CPLEX Options

CPLEX is able to generate different kinds of additional constraints that cut away non-integral solutions from the linear relaxation of the given problem. For example, CPLEX knows the following type of constraints (so-called *cuts*): clique cuts, cover cuts, disjunctive cuts, flow cover cuts, flow path cuts, gomory fractional cuts, generalized upper bound (GUB) cover cuts, implied bound cuts and mixed integer rounding (MIR) cuts.

CPLEX contains a heuristic that generates such cuts if they are helpful. Since the generation of cuts might be computational expensive, it is possible to provide hints to this heuristic. The CPLEX interface allows to forbid or to encourage the generation of the various kinds of cuts. All maximal clique constraints are already part of the problem formulation. Thus clique cuts can be safely turned off. It has turned out that the gomory fractional cuts are helpful and should be encouraged. For all other kinds of cuts, we obtained best results with the default settings.

It is possible to control the way in which CPLEX explores the branch-and-cut tree. For example, there are different orders in which the nodes of the

model	spacing [mm]	#maximal cliques	#violated odd hole ineq. (w/o lifting)	#violated lifted odd hole inequal.	
				$\varepsilon = 0.1$	$\varepsilon = 0.2$
A	50	2195	127	302	355
B	50	3105	224	600	1008
C	50	3925	57	91	100
D	25	3227	349	1033	1239

Table 6.6: Numbers of generated odd hole inequalities

tree are considered. Once a node has been selected, different strategies exist to choose the variable for branching at that node. We performed various tests with different settings for these parameters, but could not observe significant improvements over the default settings.

Furthermore, it is possible to provide an integral solution of the problem as starting solution for the branch-and-cut phase. For example, a solution obtained from a heuristic like the Greedy or Simplefill algorithm can be used. This step reduces the time that the heuristic of CPLEX spends searching for an initial integral solution. Note that in general the vast majority of the runtime is spent in the branch-and-cut phase, whereas the root relaxation phase and search for an initial integral solution need only a small fraction of time. Thus there is no large benefit in providing an integral solution. But if the available runtime is sharply limited, e.g., for a subproblem, providing a starting solution can save precious runtime.

Odd Hole Inequalities

We proposed odd hole inequalities to strengthen the ILP formulation. We compare four different versions of the ILP algorithm: (1) default (no additional inequalities), (2) including odd holes inequalities (without lifting), and (3/4) including lifted odd hole inequalities, generated from nearly violated odd holes for two values of the parameter ε, namely $\varepsilon = 0.1$ and $\varepsilon = 0.2$.

The runtime for all experiments was fixed to 24 hours. Experiments have shown that we hardly find any violated (lifted) odd hole in the later phase of the branch-and-cut process. In order to save the runtime for the exploration of the branch-and-cut tree, we restrict the generation of additional inequalities to the first 100 nodes of the branch-and-cut phase.

Table 6.6 relates the number of generated odd hole inequalities to the number of given clique inequalities in the original problem formulation. As expected, the number of violated lifted odd hole inequalities increases with ε and is higher than the number of violated odd hole inequalities. The ratio of the number of additionally generated inequalities to the number of clique inequalities strongly depends on the instance.

model	spacing [mm]	default	incl. odd hole inequalities (w/o lifting)	incl. lifted odd hole inequalities $\varepsilon = 0.1$	$\varepsilon = 0.2$
A	50	268	266	265	267
B	50	382	383	383	382
C	50	476	475	475	466
D	25	62	62	61	61

Table 6.7: Results for odd hole inequalities

The cardinalities of the obtained solutions are shown in Table 6.7. The heuristic contained in CPLEX, which has a large influence on the good (and fast) solution for model A, seems to be negatively influenced by the additional generated inequalities. While adding odd hole inequalities slightly improves the result for model B, the results for models C and D get worse.

Originally, we expected a clear benefit from the odd hole inequalities. At least, we did not expect a worsening of the results. Given the rather dissappointing results, we opted to disable the generation of odd hole inequalities in the default configuration.

6.1.3 LP Rounding

In the following we discuss two important factors of the LPR algorithm, namely the choice of the algorithm used to solve the linear programs and the choice of the parameter p.

Note that the LPR algorithm switches to the ILP algorithm when the remaining problem complexity becomes small enough. In order to study the influence of the LP algorithm and the parameter p, we disable the switch-over for the experiments in this section.

The LP Algorithm

In the preceding section we have shown that —in the context of the ILP algorithm— the barrier algorithm of CPLEX is significantly faster than the primal and dual simplex algorithm.

However, the setting of the LPR algorithm is slightly different. We do not only have to solve a single linear program, but a series of similar linear programs that emerge by repeatedly fixing variables, i.e., adding new constraints. Therefore, it seems sensible to use the dual simplex algorithm in subsequent iterations, since it can be restarted from a feasible basic solution.

We study the behavior of the following four versions of the LPR algorithm.

- **barrier/primal** The barrier algorithm is used to solve the initial linear program. The primal simplex algorithm is used for the modified problems in subsequent iterations.

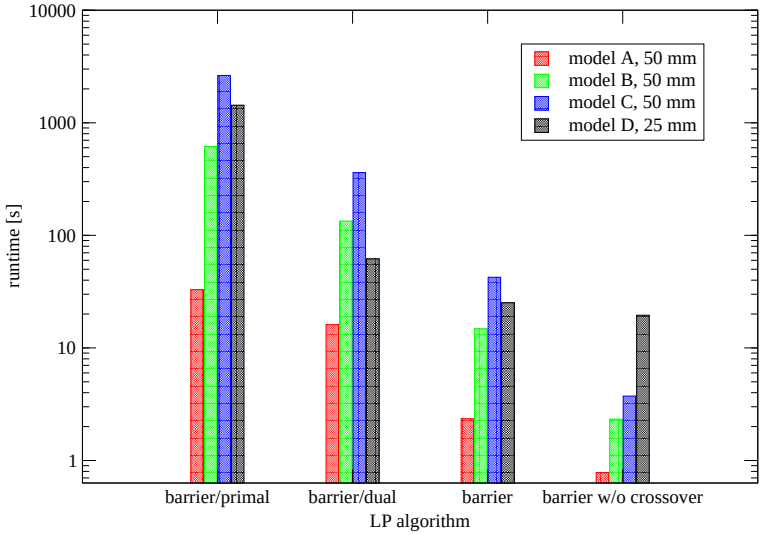


Figure 6.4: Runtime of the LPR algorithm. The runtime of the LPR algorithm strongly depends on the chosen LP algorithm. The barrier algorithm without subsequent crossover phase is about two orders of magnitude faster than the barrier/primal version.

- **barrier/dual** This version is identical to the previous one, except that the dual simplex algorithm is used for the modified problems.
- **barrier** The barrier algorithm is used to solve the initial as well as all modified linear programs. In each iteration the barrier algorithm is restarted from scratch. Note that the barrier algorithm does not produce a basic solution as it is the case for the primal and dual simplex algorithm. Therefore, by default, a so-called *crossover phase* at the end of each barrier invocation transforms the obtained solution into a basic solution.
- **barrier w/o crossover** This version is identical to the previous one, except that at the end of the barrier algorithm no crossover to a basic solution is performed.

We compare the runtime of these four LP algorithms in Figure 6.4 for several models. We use a grid with 50mm spacing for the models A, B and C and a grid with 25mm spacing for the smaller model D. We note that the cardinalities of the solutions for different LP algorithms do not significantly differ. As a general observation, the runtime of the LPR algorithm decreases with respect to the above given order of different LP algorithms.

The barrier/dual version is much faster than barrier/primal. This result goes along with the general observation that the dual simplex algorithm of CPLEX is often faster than the primal simplex algorithm. Moreover, the

dual simplex algorithm benefits from the fact that the optimal solution of the previous iteration is a feasible basic solution for the modified problem.

Surprisingly, the barrier version is faster than barrier/dual. In particular, restarting the barrier algorithm from scratch in each iteration is faster than starting the dual simplex algorithm from a feasible basic solution.

The variant barrier w/o crossover was motivated by the good performance of the barrier variant. Since the barrier algorithm cannot use a basic solution as a starting point for a modified problem, one might wonder whether the crossover to a basic solution is necessary at all. Clearly, omitting the crossover process results in a significant smaller runtime. On the other hand, the structure of the solution changes. Barrier solutions tend to be midface solutions, whereas simplex algorithms tend to set variable values to their lower or upper bound. The question is, whether this change in the solution structure has a negative effect on the quality of the overall packing. It turns out that this is not the case. There is no significant difference between the solution cardinalities of the barrier and barrier w/o crossover version.

Thus we use the barrier w/o crossover version as default for the LPR algorithm.

The Parameter p

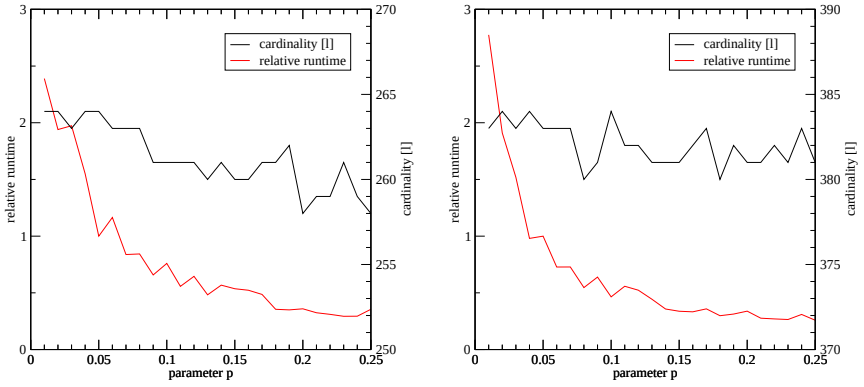
We want to discuss the influence of the parameter p, which controls the fraction of the variables that are fixed in each iteration. In particular, we are interested in the runtime of the algorithm and the cardinality of the computed solutions. The higher the parameter p, the lower the number of iterations and LP problems. On the other hand, the more variables are fixed in an iteration, the less accurately the LP solution reflects the optimal solution of the packing problem. Therefore, we expect that the cardinality of a solution as well as the runtime decreases with increasing values of p.

Figure 6.5 shows the runtime of the LPR algorithm and the cardinality of the solution as a function of the parameter p. The experiments show that the cardinality of the solutions slightly decreases with increasing value of p. The runtime is roughly proportional to p^{-1}.

We set the parameter p to 0.05 for all further tests. In general, smaller values do not produce better solutions, but impose a strongly higher runtime. Larger values of p lead to a slightly lower runtime, but often produce inferior solutions.

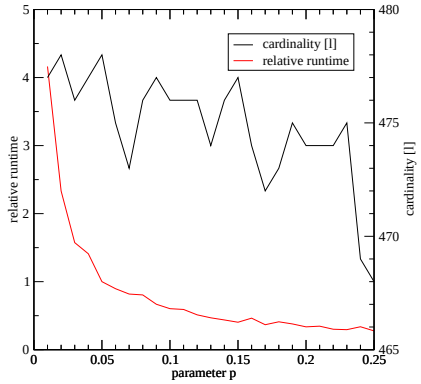
6.1.4 Reactive Local Search

We implemented the RLS algorithm and associated data structures from scratch based on the description in [BP01]. We chose the LEDA datatype `leda_h_array` as implementation for the hash table. Our implementation differs in one detail from the algorithm in [BP01]: Since our instances are



(a) model A

(b) model B



(c) model C

Figure 6.5: Results and runtime of the LPR algorithm. The graphs display the cardinality of the solution and the needed runtime as a function of the parameter p. The runtimes are relative to the runtime for $p = 0.05$ of the respective model.

much larger than the instances presented in this paper, we cannot afford to maintain the data structure MissingList. This data structure is used in the function IncrementalUpdate$(v, type)$ to speed up the update of all other data structures. The data structure requires $3|V|^2$ `int`'s, which equals 846 MB of RAM for the 50mm grid of the small model A with 8787 nodes. Due to the high memory requirements, we drop the data structure and recompute the information on the fly when needed.

We compared our implementation with the implementation of the authors of [BP01]. We enabled also in their implementation the option not to maintain the MissingList data structure, but to recompute the information. The speed (in iterations per second) and the quality (cardinality of the computed solutions) of both implementations are comparable.

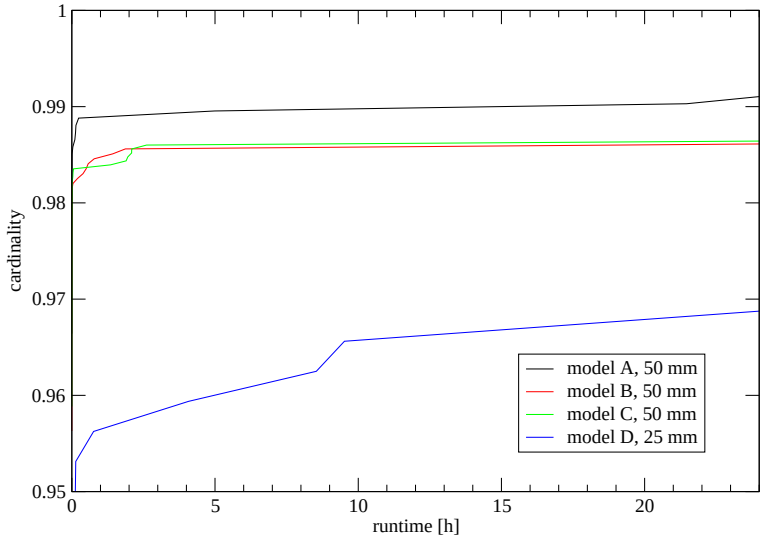


Figure 6.6: Results of the RLS algorithms. The graph displays the cardinality of the solution as a function of the runtime. The solution cardinality is scaled with respect to the LP bound of the respective model.

The size of the best stable set as a function of the runtime is depicted in Figure 6.6. The graphs represent the average of five runs for each model. The values are scaled such that the LP bound of the respective model equals 1.0. In an initial phase, the RLS algorithm is often able to improve the best solution found so far. Later, further improvements are very rare.

We would like to remark that, unlike the Greedy algorithm, the variance in the cardinality of the obtained solutions is very small. In all tests so far, the cardinalities of several runs for a particular model differ by at most one, provided the runtime is sufficiently long.

Substitution of $deg_C[v]$ by $deg_G(v)$

The authors of [BP01] propose a variant of the RLS algorithm, in which the degrees in the graph G are used to break ties, instead of the degrees in the subgraph induced by the set C. This modification is worth considering, since it reduces the worst-case complexity of an iteration from $O(|V| + |E|)$ to $O(|V|)$.

We compare the performance of this variant with the initial version. In Table 6.8 we summarize the averages of five runs for 24 hours each. The number of iterations per second increases by a factor of about 2.5 if we use grids with a spacing of 50mm. The increase is even higher for grids with a spacing of 25mm. On the other hand, the solution cardinality slightly decreases. Therefore, we reject this variant.

model	spacing [mm]	$deg_C[v]$ card.	$deg_C[v]$ iter./s	$deg_G(v)$ card.	$deg_G(v)$ iter./s
A	50	265.5	11319	265.4	28462
B	50	383.6	7110	382.8	19135
C	50	479.6	5347	478.2	13597
D	25	62.0	2586	60.0	16998

Table 6.8: Performance of a variant of the RLS algorithm

6.1.5 Matching and Easyfill

In this section we discuss the performance gain of a modification for the Matching and Easyfill algorithm. This modification restricts the set I in the first phase of both algorithms by removing some of the outmost layers of cells of I, and hence increases the flexibility for the second phase.

We consider the three models A, B, and C — the model D is too small to give meaningful results. In contrast to other tests in this section we use grids with a spacing of 25mm. The results for the Easyfill algorithm in combination with the Greedy algorithm are shown in Table 6.9. The number k of layers removed from the set I ranges from zero to five. As expected, the number of boxes packed in the second phase of the algorithm increases with k, while the number of boxes packed in the first phase decreases (the number ranges are due to the fact that the Easyfill algorithm generates a series of subproblems). Since the majority of the runtime is spent in the second phase, the runtime also increases with the parameter k.

Most interesting is the maximum number of boxes packed in both phases, i.e., the size of the computed packing. If the subproblems of the second phase were packed optimally, then the size of the overall packing would increase with k. Unfortunately, the size of these subproblems is too large for the ILP algorithm. Hence we have to use a different algorithm, and there is no guarantee that an optimal solution is computed.

In our experiments the cardinalities of the packings for $k = 1$ are significantly higher than those for $k = 0$. Higher values for k do not seem to produce better solutions, but need a significantly higher runtime. Similar results can be observed for the Matching algorithm, as well as for other algorithms in the second phase. Based on these results we decided to set the parameter k to 1 for all further tests.

6.2 Subdivision into Several Regions

During application of the presented algorithms to a large number of instances we found several models for which our algorithms yield quite bad results. For example, consider the grid with 50mm spacing for model C. The relaxation

model	#layers (k)	#boxes 1st phase	#boxes 2nd phase	max. #boxes (both phases)	relative runtime
A	0	93 – 229	32 – 167	266	1.00
	1	84 – 184	75 – 178	271	1.58
	2	64 – 142	117 – 202	269	2.41
	3	42 – 101	153 – 221	269	3.42
	4	20 – 75	176 – 243	268	4.53
	5	0 – 59	194 – 266	268	5.63
B	0	124 – 311	57 – 225	375	1.00
	1	106 – 256	91 – 263	377	1.66
	2	82 – 208	161 – 279	377	2.57
	3	72 – 150	204 – 296	375	3.66
	4	0 – 119	240 – 366	375	4.87
	5	0 – 90	263 – 366	376	5.96
C	0	191 – 407	56 – 268	472	1.00
	1	161 – 347	98 – 311	481	1.60
	2	142 – 271	191 – 329	480	2.48
	3	93 – 219	232 – 377	480	3.55
	4	15 – 171	293 – 455	481	4.70
	5	0 – 136	315 – 479	481	5.79

Table 6.9: Removal of outmost layers in the first phase of the Matching or Easyfill algorithm. The removal of some layers significantly increases the cardinality of the solution. Good results in terms of solution cardinality and required runtime are achieved for $k = 1$.

of the linear program yields an upper bound of 486 liters, and our best packing consists of 480 boxes. For the grid of 25mm spacing, we obtain an upper bound of 506 liters, whereas our best packing contains only 488 boxes. In contrast to these numbers, there is a manual packing of an experienced engineer that comprises 513 boxes.

These numbers demonstrate the drawback of the discretization process. Not only the cardinality of the solutions, but even the upper bounds are clearly below the size of the manual packing. The reason for this behavior is the fact that there are regions in the trunk where the set of INSIDE cells poorly reflects the local geometry. In these regions, it is often possible to choose a different orientation for the grid, such that it adapts much better to the local geometry. We identify three typical examples for such regions in the following subsections.

To overcome this drawback, we propose the following solution: Instead of using a single grid, we subdivide the interior of the trunk into different regions. In each region, a grid that is suited for the local geometry is used. This idea is similar to the approach of the Partition algorithm. However, the

motivation here is not to reduce the problem complexity, but to improve the discretization.

The shape of the regions can often be defined by the intersection of the trunk interior and a halfspace. Therefore we use oriented planes (called *separators*) to cut off a region. In general, a single separator per region suffices. In rare cases two or three separators are needed.

We need to take care that the subdivision process does not impair the situation. For example, if the regions are processed independently, boxes that cross region boundaries are ruled out. We propose the following approach. Typically, there is a quite large main problem and a very low number of small regions that have been cut off from the main problem. First we process the small cut-off regions. Then the solutions of these subproblems are imported into the original problem as a fixed (partial) packing. Before importing such a solution, we consider the relative position of the subproblem and the main problem and use a simple heuristic to move the boxes of the subproblem as far away from the main problem as possible. This procedure increases the space that is available to the main problem.

6.2.1 Storage Spaces next to Wheel Houses

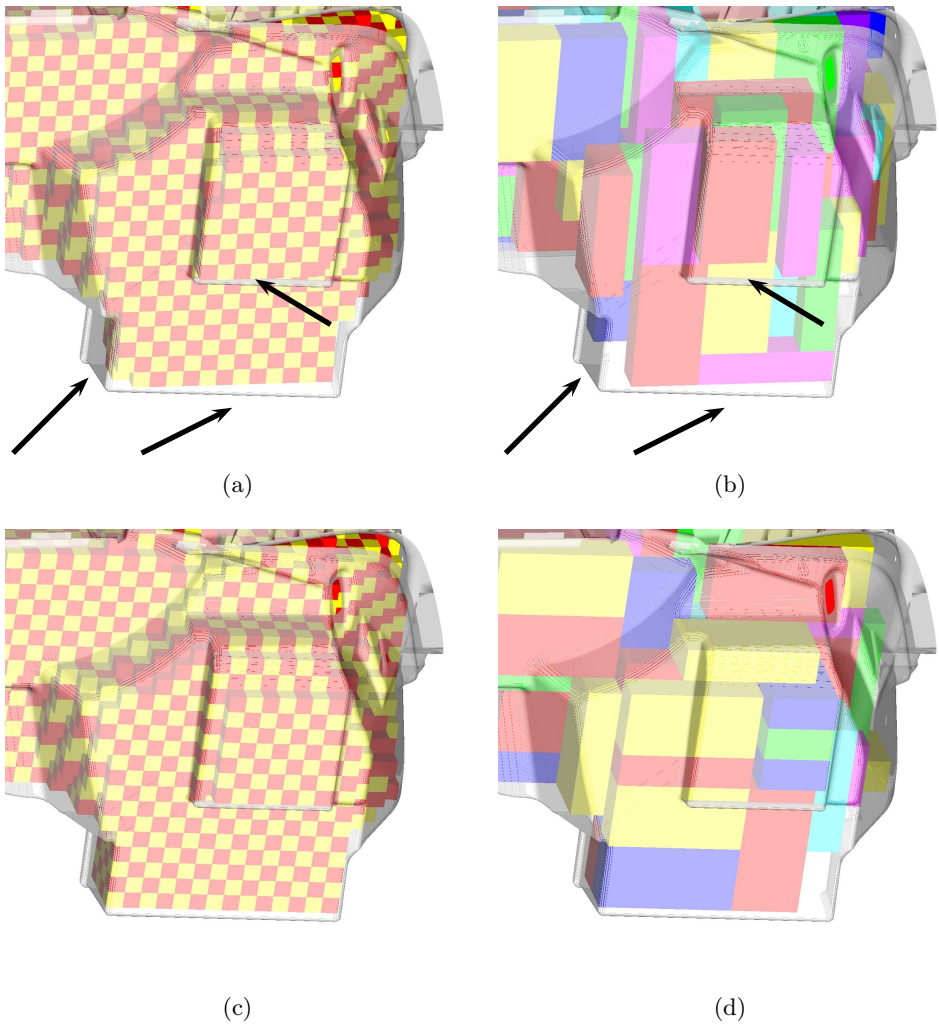
Often, the width of a trunk increases behind the rear wheels. Furthermore, the shape of the trunk also extends beneath the floor. This feature is typical and can be found in many models. An example for this feature can be seen in Figure 6.7, which shows model C from the left. The grid in Figure 6.7(a) is aligned with the floor of the trunk. The orientation of the grid does not fit the local geometry of the trunk. Consequently, the corresponding packing in Figure 6.7(b) wastes much space. In contrast, a grid that aligns with the local geometry is shown in Figure 6.7(c). An optimal packing for this grid is shown in Figure 6.7(d).

Such regions can be defined with two parallel planes that are orthogonal to the wheel axes. The distance of the planes is chosen such that it is a multiple of 50mm. This reduces the set of reasonable distances to a few values. In most cases, the distances 950mm, 1000mm or 1050mm are used.

A similar improvement can be achieved on the right side of the trunk. Both measures together allow to add eight more boxes to our solution, which consists now of 496 boxes.

6.2.2 Breaks in the Floor of the Trunk

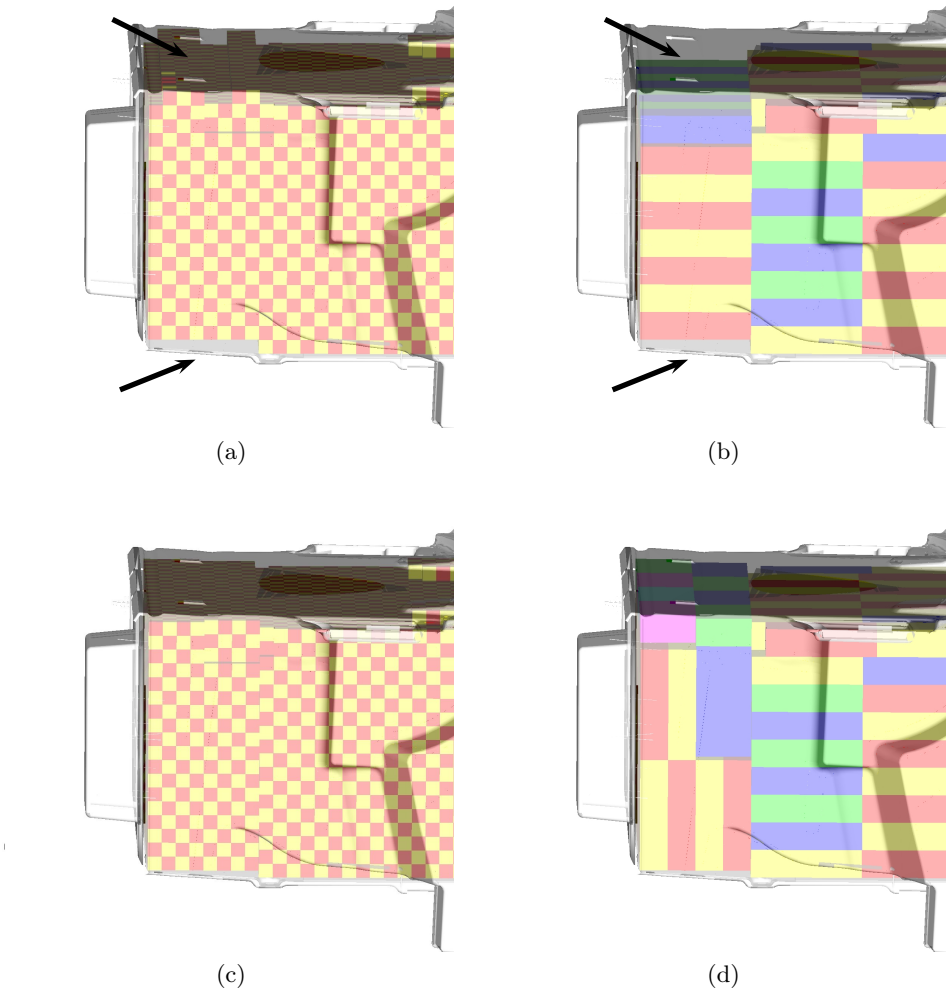
In many car models, the floor of the trunk is a large planar face. However, in a few cases there is a break in the floor, often located near the rear seatback. A small part of the trunk does not have a horizontal, but slightly sloping floor (see Figure 6.8(a)). There is a large wedge-shaped region that is not taken into account at all. Additionally, at the top of the trunk, the grid



(a) (b)

(c) (d)

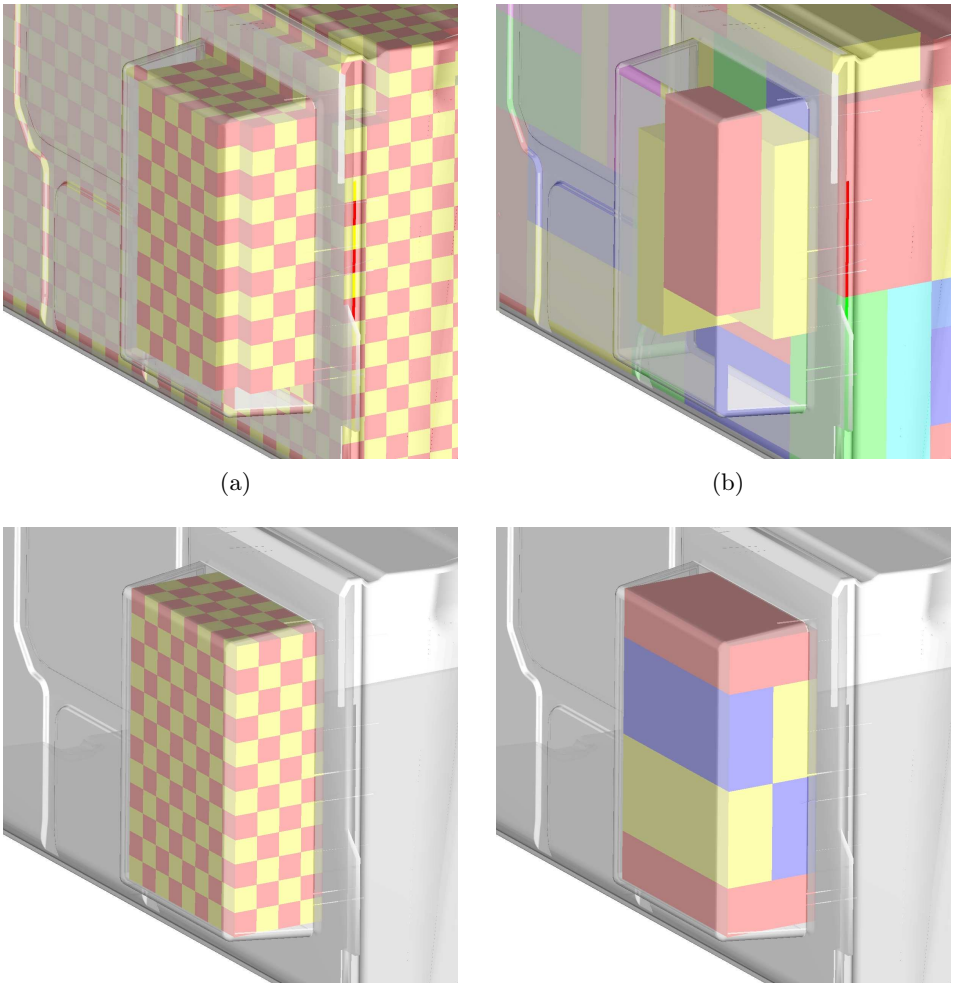
Figure 6.7: Storage spaces next to wheel houses. The global grid (a) and the corresponding packing (b) are aligned with the floor of the trunk and do not adapt to the local geometry. A local grid (c) and its packing (d) exploit the available space much better.



(a) (b)

(c) (d)

Figure 6.8: Breaks in the floor of the trunk. The global grid (a) and the corresponding packing (b) are aligned with the floor of the trunk and do not adapt to the local geometry. A local grid (c) and its packing (d) exploit the available space much better.



(a) (b)

(c) (d)

Figure 6.9: Additional storage spaces. The global grid (a) and the corresponding packing (b) are aligned with the floor of the trunk and do not adapt to the local geometry. A local grid (c) and its packing (d) exploit the available space much better.

has an irregular structure, which is highly disadvantageous for packing (see Figure 6.8(b)). We cut off the region with sloping floor and shift the grid in this region downwards until it touches the floor (see Figure 6.8(c)). The uncovered space has been significantly reduced, both at the bottom and at the top of the trunk. Thus eight more boxes can be packed into the cut-off region (see Figure 6.8(d)). In summary, we can pack 504 boxes into the trunk.

6.2.3 Additional Storage Spaces

A very rare example is shown in Figure 6.9. The extension near the rear seatback is planned to hold control devices for additional equipment, e.g., air-conditioning. In model variants without this equipment, the space is added to the trunk. The shape of this extension is explicitly designed such that exactly six boxes can be packed. An optimal placed grid and a corresponding packing can be seen in Figure 6.9(c) and Figure 6.9(d), respectively.

The local grid for the additional storage space makes it possible to add three more boxes. In total, the subdivision of the trunk into regions has increased the packing from 488 to 507 boxes. This value lies within an acceptable range of the expert solution of 513 boxes. We remark that our combinatorial solution can be further improved to 513 boxes using the SGCSA approach [Rie05].

6.3 Comparison of Algorithms

For our tests we use a set of 21 models including the already presented models A, B, C, and D. Table 6.10 gives an overview of these models, which are roughly ordered by increasing volume. The models H and I denote convertibles and come in two variants representing an open and a closed roof. In this set we also included our largest model S, despite the fact that no expert solution is available.

For each model the table lists the size of the expert solution as well as the best combinatorial solution. The expert solutions were constructed by an experienced engineer with the help of a CAD system. Note that these expert solutions are not bound to our discretization. To measure the quality of our solutions, we consider the relative deviation from the expert's one. We also list the number of regions that were used to compute the combinatorial solution (see Section 6.2).

The quality of the best combinatorial solution varies. In half of the cases we meet the quality requirements of at most 1% deviation. In a few cases we are significantly better than the expert solution. Only in 4 of 20 cases (models B, H1, H2 and M) we fall short of the expert solution by more than 2%. Although a deviation within 1-2% does not meet the initially prescribed quality bound, such solutions are still acceptable in practice. The

model	type of vehicle	expert solution	combinat. solution	relative deviation	#regions
A	sedan	272	272	0.00%	1
B	sedan	404	390	−3.47%	3
C	sedan	513	507	−1.17%	4
D	sports car	62	62	0.00%	1
E	—[a]	6	7	+16.67%	1
F	—[a]	46	50	+8.70%	1
G	—[b]	80	80	0.00%	1
H1	convertible[c]	185	171	−7.57%	2
H2	convertible[d]	277	256	−7.58%	1
I1	convertible[c]	212	209	−1.14%	1
I2	convertible[d]	330	324	−1.82%	1
J	sedan	315	326	+3.49%	1
K	sedan	384	391	+1.82%	1
L	station wagon	396	394	−0.51%	2
M	sedan	434	425	−2.07%	4
N	sedan	480	475	−1.04%	4
O	sedan	499	492	−1.40%	3
P	minivan	722	720	−0.28%	1
Q	minivan	1203	1191	−1.00%	2
R	minivan	1775	1751	−1.35%	3
S	minivan	n/a	2068	n/a	3

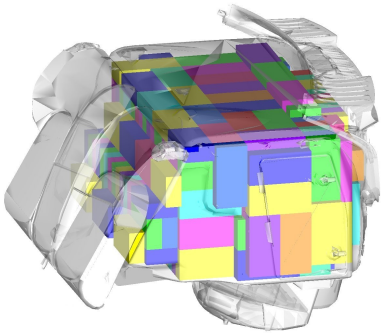
[a] additional storage compartment
[b] spare wheel compartment
[c] open roof
[d] closed roof

Table 6.10: Overview of test results. For each model the table lists its vehicle type, the size of the expert solution, the size of the best combinatorial solution, the relative deviation of the latter from the former, and the number of regions that were used.

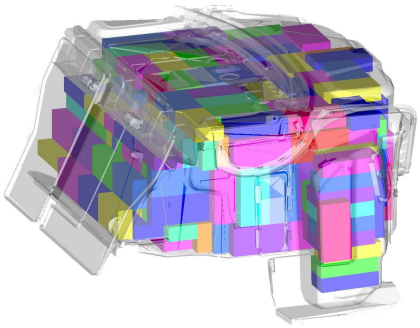
second quality requirement demands an absolute deviation of not more than 10 liters. We meet this requirement for all models with exception of the models B, H1, H2 and R.

The results for both variants of model H are extremely bad. Model H is a convertible, therefore, its trunk is rather small and has a very complicated shape due to the roof-folding mechanism. For both instances the discretization of the solution space is a too severe restriction. For example, the LP bound for model H1 is 178, which is already significantly smaller than the expert solution of 185 boxes.

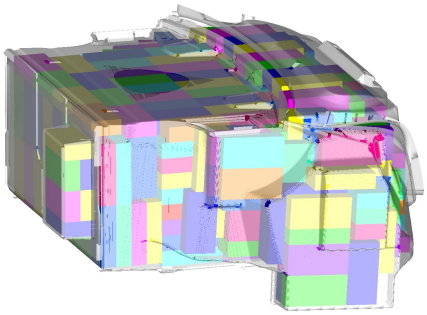
The best combinatorial solutions for models A to D are depicted in Figure 6.10.



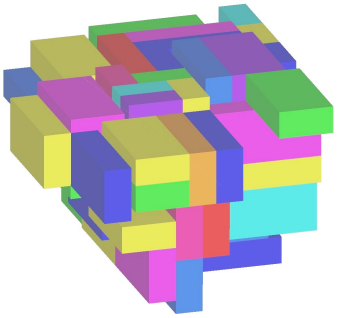
(a) model A



(b) model B



(c) model C



(d) model D

Figure 6.10: Best combinatorial solutions. The corresponding expert solutions are depicted in Figure 4.1. The grids for the models A and D are shown in Figure 6.1; the models B and C were subdivided in several regions, hence the grids depicted in Figure 6.1 do not match the solutions shown here.[4]

We remark that the combinatorial solution often can be improved with the help of the continuous SGCSA approach (see Section 1.2). In particular, this holds for trunks with a complicated shape. If an instance is too large to be efficiently handled in its entirety by the SGCSA approach, if is often useful to apply the SGCSA approach only to regions with a complicated geometry, e.g., storage spaces next to wheel houses (see Section 6.2). For example, the results for the models H1 and H2 can be improved to 181 and 285 boxes, respectively [Rie05].

Detailed results for each combination of a particular model and algorithm can be found in the next two tables. The cardinality of a solution and the actually needed runtime are shown in Tables 6.11 and 6.12, respectively.

[4]For legal reasons we may not publish the geometry of model D.

The list of algorithms comprises the Greedy, ILP, LPR and ILP algorithm, stand-alone and in combination with the Matching and Easyfill algorithm. Furthermore, the results for the Simplefill and Partition algorithm (the latter in combination with the ILP algorithm) are given.

The second column gives the time limit for each instance. Usually, we set a time limit of 24 hours as described in the requirements for the software system (see Chapter 1). For smaller models we reduce the limit to 2 or 12 hours, whereas we set the limit to 48 hours for the very large instances Q, R, and S.

The following remarks apply to models for which the trunk was subdivided into several regions (see Table 6.10). With exception of the models Q, R, and S, the size of the regions apart from the main region is very small. Therefore, the ILP algorithm was used to compute a solution within a few minutes. The columns in Tables 6.11 and 6.12 indicate the algorithm that was used to solve the main problem, i.e., the largest region. The size of a solution in Table 6.11 is the sum of all regions, whereas the runtimes in Table 6.12 hold for the main problem. For the models Q, R, and S, half of the total runtime of 48 hours was spent on the main region, the other half was equally distributed among the remaining regions.

A grid with 25mm spacing was used for all models except the very small model E, for which we constructed a grid with 12.5mm spacing. Grids with 12.5mm spacing were also used for very small regions (typically storage spaces next to wheel houses).

Let us first consider non-(I)LP-based algorithms, i.e., the Greedy, RLS, and Simplefill algorithm. Comparing the results of these algorithms it turns out that in all cases the Easyfill+RLS algorithm is better than or equal to the Matching+RLS algorithm. Moreover, Easyfill+RLS is better than or equal to the RLS algorithm (2 exceptions), the maximum of all Greedy algorithms (1 exception), and the Simplefill algorithm (4 exceptions). Therefore, the Easyfill+RLS algorithm is clearly the best choice among the non-ILP-based algorithms. If a first estimate is needed and the available time is very short, we propose to use the Simplefill algorithm instead.

Now let us consider (I)LP-based algorithms, i.e., the ILP, LPR, and Partition+ILP algorithm. The results of the Partition+ILP algorithm are clearly worse than those of the other (I)LP-based algorithms. The results get even worse when the number of cutting planes, and hence the number of subproblems is increased. As expected, the ILP and LPR algorithm produce very good results on small models (see models D to H1). Larger instances (above ca. 200 liters) are too complex to be handled by these algorithms within the time limit of 24 hours. The results of the Matching or Easyfill algorithm combined with the ILP or LPR algorithm are very close; the results for the combinations involving the LPR algorithm are slightly better than those using the ILP algorithm. Apart from the set of small models D to H1, the results of Easyfill+RLS are often better than or equal to those of an (I)LP-based algorithm.

model	time limit [h]	Greedy	Match. +Gr.	Easyf. +Gr.	ILP	Match. +ILP	Easyf. +ILP	LPR	Match. +LPR	Easyf. +LPR	RLS	Match. +RLS	Easyf. +RLS	Simple-fill	Part. +ILP
A	24	265	268	271	—[a]	270	269	—[a]	269	269	269	270	272	270	260
B	24	382	386	387	—[a]	390	387	—[a]	390	390	385	387	389	384	379
C	24	502	506	507	—[a]	507	506	—[a]	506	506	507	507	507	507	497
D	12	56	59	59	62	60	61	61	59	61	62	60	61	60	60
E	2	5	7	6	7	7	7	7	7	7	7	7	7	6	7
F	2	47	48	48	50	49	49	50	50	50	49	49	49	47	48
G	12	74	76	77	80	78	79	80	78	80	78	78	79	77	79
H1	24	167	169	168	170	170	170	171	171	171	170	170	170	165	170
H2	24	250	250	255	—[a]	255	252	—[a]	256	255	255	253	254	248	246
I1	24	204	203	203	—[a]	207	206	—[a]	209	206	204	205	206	203	206
I2	24	307	316	319	—[a]	321	313	—[a]	321	321	315	319	321	312	307
J	24	318	321	324	—[a]	326	324	—[a]	325	324	320	323	325	322	322
K	24	376	384	389	—[a]	388	388	—[a]	390	391	385	387	390	385	381
L	24	394	394	394	—[a]	394	394	—[a]	394	394	394	394	394	394	391
M	24	420	424	424	—[a]	425	425	—[a]	425	425	424	425	425	425	422
N	24	467	473	474	—[a]	475	474	—[a]	474	474	474	475	475	473	475
O	24	483	485	488	—[a]	489	491	—[a]	489	487	487	486	491	492	440
P	24	694	712	717	—[a]	717	712	—[a]	716	695	—[a]	715	720	708	702
Q	48	1188	1189	1189	—[a]	1191	1191	—[a]	1191	1191	1190	1191	1190	1189	1190
R	48	1716	1747	1747	—[a]	1743	1740	—[a]	1743	1746	—[a]	1747	1750	1751	1750
S	48	2031	2051	2058	—[a]	2021	2031	—[a]	2060	2066	—[a]	2054	2059	2068	2003

[a] The instance is too large to be handled by this algorithm.

Table 6.11: Comparison of packing sizes. For each algorithm the table lists the size of the packing that was achieved within the given runtime limit. The ILP algorithm is the preferred choice for smaller instances, whereas the Matching+RLS is a good candidate for larger problems. The Simplefill algorithm is a recommended alternative if a fast solution is needed.

model	time limit [h]	Greedy	Match. +Gr.	Easyf. +Gr.	ILP	Match. +ILP	Easyf. +ILP	LPR	Match. +LPR	Easyf. +LPR	RLS	Match. +RLS	Easyf. +RLS	Simple-fill	Part. +ILP
A	24	6	42	413	—[a]	376	1440[b]	—[a]	150	1440[b]	1440[b]	1440[b]	1440[b]	61	1440[b]
B	24	6	40	382	—[a]	485	1440[b]	—[a]	286	1440[b]	1440[b]	1440[b]	1440[b]	54	1440[b]
C	24	6	40	250	—[a]	1440[b]	1440[b]	—[a]	1440[b]	1440[b]	1440[b]	1440[b]	1440[b]	56	1440[b]
D	12	1	8	80	720[b]	58	720[b]	18	85	689	720[b]	720[b]	720[b]	9	720[b]
E	2	1	120[b]	120[b]	2	120[b]	120[b]	2	120[b]	120[b]	120[b]	120[b]	120[b]	10	1
F	2	1	7	61	30	55	120[b]	30	34	120[b]	120[b]	120[b]	120[b]	10	120[b]
G	12	1	10	114	720[b]	59	720[b]	720[b]	41	720[b]	720[b]	720[b]	720[b]	14	720[b]
H1	24	1	22	166	1440[b]	1065	1440[b]	1440[b]	263	1440[b]	1440[b]	1440[b]	1440[b]	20	1440[b]
H2	24	4	31	269	—[a]	888	1440[b]	—[a]	167	1440[b]	1440[b]	1440[b]	1440[b]	40	1440[b]
I1	24	4	43	438	—[a]	615	1440[b]	—[a]	218	1440[b]	1440[b]	1440[b]	1440[b]	39	1440[b]
I2	24	5	45	421	—[a]	1419	1440[b]	—[a]	255	1440[b]	1440[b]	1440[b]	1440[b]	48	1440[b]
J	24	5	31	294	—[a]	762	1440[b]	—[a]	467	1440[b]	1440[b]	1440[b]	1440[b]	47	1440[b]
K	24	6	36	361	—[a]	644	1440[b]	—[a]	257	1440[b]	1440[b]	1440[b]	1440[b]	50	1440[b]
L	24	1	26	157	—[a]	1440[b]	1440[b]	—[a]	1440[b]	1440[b]	1440[b]	1440[b]	1440[b]	16	585
M	24	4	26	234	—[a]	583	1440[b]	—[a]	550	1440[b]	1440[b]	1440[b]	1440[b]	39	1440[b]
N	24	5	21	177	—[a]	192	1440[b]	—[a]	249	1440[b]	1440[b]	1440[b]	1440[b]	31	311
O	24	7	43	419	—[a]	391	1440[b]	—[a]	1440[b]	1440[b]	1440[b]	1440[b]	1440[b]	54	1440[b]
P	24	13	57	547	—[a]	1244	1440[b]	—[a]	859	1440[b]	—[a]	1440[b]	1440[b]	89	1440[b]
Q	48	3	19	250	—[a]	190	1440[b]	—[a]	199	1440[b]	1440[b]	1440[b]	1440[b]	32	39
R	48	29	132	1030	—[a]	1440[b]	1440[b]	—[a]	1440[b]	1440[b]	—[a]	1440[b]	1440[b]	89	1440[b]
S	48	34	173	1440[b]	—[a]	1440[b]	1440[b]	—[a]	1440[b]	1440[b]	—[a]	1440[b]	1440[b]	212	651

[a] The instance is too large to be handled by this algorithm.

[b] The algorithm was stopped after the time limit was reached.

Table 6.12: Comparison of runtimes. For each algorithm the table lists the actually needed runtime (in minutes). The Greedy and Simplefill algorithms are the fastest algorithms, whereas the RLS algorithm always utilizes the available runtime by design. Combinations involving the Matching algorithm are faster than those using the Easyfill algorithm due to the smaller number of subproblems.

The corresponding runtimes are shown in Table 6.12. As a general observation, the runtime increases with the problem complexity, which is in close relation to the discrete trunk volume. This relation can be directly observed for algorithms that finish within the given time limit, e.g., the Greedy and Simplefill algorithm. A second factor that influence the problem complexity and runtime is the shape of the trunk.

The Greedy and Simplefill algorithms are also the fastest algorithms. In contrast, RLS based algorithms always utilize the available runtime by design. Combinations involving the Matching algorithm are faster than those using the Easyfill algorithm. This is due to the fact that the number of subproblems generated by the Easyfill algorithm is six times as high as in the case of the Matching algorithm. Therefore, the Easyfill algorithm often reaches the time limit, whereas the Matching algorithm usually finishes much earlier. As pointed out at the beginning of this chapter, all runtimes can be reduced by a factor of two or three on modern x86-based hardware.

We can summarize these findings in the following guidelines:

- For small models (up to 100 liters), use the ILP algorithm.
- For larger models, use the Easyfill+RLS algorithm.
- If a fast estimate is desired, use the Simplefill algorithm.

We mentioned in Chapter 1 that, apart from the major objective of maximizing the number of boxes, there are also minor objectives regarding the structure of the packings. Tight packings and packings where the majority of the boxes have the same orientation are preferred.

Due to the discrete model, all boxes are aligned with the axes of the coordinate system of the underlying grid. If the trunk was subdivided in several regions, there is a small, limited number of such alignments.

Moreover, the Easyfill algorithm ensures that the boxes in the core, i.e., the boxes packed in the first phase, have the same orientation (see Figure 5.4). This property is not guaranteed for the Matching algorithm, but in general holds for the majority of the boxes (see Figure 5.3). The Simplefill algorithm packs the majority of the boxes in the first iteration, resulting in the same orientation (see Figure 5.1). In contrast, solutions produced by the Greedy, ILP, LPR, RLS, and Partition+ILP algorithm do not exhibit any visible structure. Therefore, in reality, the reproduction of a packing with a physical mockup is much easier if the packing results from the Matching, Easyfill or Simplefill algorithm.

These algorithms are also favorable with respect to the second minor objective. The Easyfill algorithm always produces a tight core packing. Uncovered space can still occur in the outer parts of the trunk, which are packed in the second phase of the algorithm. The same holds in most cases for the Matching and Simplefill algorithm. If the Greedy, ILP, LPR, or RLS algorithm is used directly, uncovered grid cells are scattered throughout the whole trunk.

All in all, the presented algorithms do no only pursue the goal of packing a high number of boxes into the trunk, but also address the minor objectives.

Chapter 7

Summary

7.1 Conclusion

In this thesis we presented a combinatorial approach for the trunk packing problem according to DIN 70020. This problem differs from many packing problems studied in the literature insofar as the container has an irregular shape and the number of items to be packed is very high. Apart from the continuous SGCSA approach, our approach is the first fully automated solution for this problem.

We showed that this packing problem is *NP*-complete and gave a polynomial-time approximation scheme. Unfortunately, this approximation scheme is only useful for huge instances, but it inspired the Partition algorithm.

We discretized the problem in a two-fold way: discretization of the space and restriction of the feasible box placements. Our classification algorithm for the cells of the discretized space is able to handle the deficiencies in the input data, in particular, the holes in the boundary description of the trunk. We also presented a heuristic for the reconstruction of the face normals, which are crucial to exploit specified tolerances for the trunk geometry.

We formulated the discrete packing problem as an integer linear program based on the complete clique formulation. To strengthen this formulation, we dynamically added violated lifted odd hole inequalities during the branch-and-bound phase. It turned out that, even with the help of state-of-the-art ILP solvers, this approach is practicable only for instances up to about 100 liters.

Therefore we presented different heuristics capable of handling larger instances, e.g., the LP Rounding, the Reactive Local Search, and the Simplefill algorithm. A second class of heuristics reduces the packing problem to a set of smaller subproblems. For example, the Matching and the Easyfill algorithm produce solutions that exhibit a regular structure, similar to manually constructed packings. Combinations involving one of both algorithms generate very promising solutions.

We tested our algorithms on a large set of real-world instances and compared the results to solutions produced by human experts. It turned out that the best results are obtained by a combination of the Easyfill and the Reactive Local Search algorithm, whereas the ILP formulation is the method of choice for smaller instances. We propose to use the Simplefill algorithm if a fast estimate of the luggage capacity is needed. We have seen that the size of a packing often can be significantly improved by subdividing the trunk into several regions, which are discretized independently based on the local geometry of the trunk. In most cases, we meet the prescribed quality bounds. For some instances, we even significantly outperform the expert solutions.

The algorithms presented in this work as well as the continuous SGCSA approach are implemented in an industrial-strength software system, which is used by our project partner in the design process of new cars.

7.2 Further Work

In this section we present some directions for further research.

The SAE Packing Problem

A natural extension of this work is to adapt and extend the presented algorithms to the SAE packing problem (see Section 1.2). Similar to our packing problem, the SAE packing problem is a three-dimensional packing problem involving a single container with irregular shape. The main differences are:

- **Item size** The SAE boxes are much larger than the boxes in our problem, e.g., an SAE box of type A has a volume of about 67 liters. Thus the number of items that can be packed into a trunk is much smaller than in our case, e.g., the trunk of a mid-size car usually contains about 15 SAE boxes.
- **Item shape** SAE boxes of different types are not congruent. The side lengths of the boxes (with exception of the golf-bags and H-boxes) are multiples of 0.5 inch, but almost all side length ratios are not integral.
- **Weights** The items contribute in different quantities to the objective value. These quantities (*weights*) correspond to the volume of the respective item.
- **Subset selection** The standard defines which subsets of a single (or multiple) standard luggage set(s) are feasible. For example, it is infeasible to pack the trunk using exclusively H-boxes of multiple standard luggage sets.
- **Golf bag** The standard luggage set contains an item that does not have cuboid shape.

The adaption of our algorithms and datastructures to the SAE packing problem requires modifications at several levels.

The side lengths of the SAE boxes (with exception of the golf-bags and the H-boxes) are multiples of 0.5 inch (more precisely, only two side lengths are odd multiples of 0.5 inch). A grid spacing of 0.5 inch is most probably too complex for mid-size instances, but a spacing of 1.0 inch seems to be reasonable. The algorithms used in the discretization process can easily be adopted to the new setting. Since the SAE boxes are significantly larger than the DIN boxes, the ratio of the numbers of INSIDE* cells to the number of INSIDE cells increases. Additionally, we can handle holes in the boundary description up to size of $152\text{mm} \times 114\text{mm}$, which is the smallest cross section of an SAE box.

Since the items are not congruent, the Matching and Easyfill algorithms can no longer be used. Other algorithms, like the Greedy, ILP or RLS algorithm, can be adapted to the new setting. Note that the structure of the conflict graph drastically changes. Due to larger item sizes, the number of anchor cells in the grid decreases while the density of the conflict graph increases. Since we are dealing with non-congruent items, we do not only need a single node for each feasible pair of anchor cell and orientation, but also one node per item type.

The SAE packing problem can be reduced to a *weighted* stable set problem. Additionally, it is necessary to model various other constraints, e.g., the limited number of items per type in a standard luggage set, or the order in which the different item types are to be considered.

Stronger ILP Formulation

In Section 5.1.2 we presented an ILP formulation based on maximal clique constraints. We also proposed lifted odd hole inequalities to strengthen the problem formulation. Unfortunately, the experimental results in Section 6.1.2 do not show a clear benefit of these inequalities. More detailed studies are necessary to understand why the lifted odd hole inequalities do not help in practice.

Several other classes of facet-defining inequalities are known in the literature. STRIJK et al. successfully use *mod-k cuts* for a stable set problem arising from map labeling [SVA00] (see also [CFL00]). TROTTER gives a generalization of odd holes known as *webs* [Tro75]. CHENG and CUNNINGHAM introduced *wheel* inequalities [CC97].

It would be interesting to study the influence of these inequality classes on our problem instances. An efficient algorithm for the separation problem (specifically for our case) is an important element for an overall performance improvement.

Optimal Grid Placement

In Section 4.5 we discussed how to obtain a good placement for the grid. The presented approach can improved with respect to the objective function as well as the optimization method.

We used the number of INSIDE cells as objective function to find a good grid position. We also mentioned that a criterion like this has its drawbacks (see also Figure 4.10 and 4.11). We propose to consider not only the number, but also the arrangement of INSIDE cells. One idea is to consider the number of grid cells in each row of the grid. Let k denote the number of adjacent INSIDE cells in such a row. At least $k \bmod n$ out of the k cells are never covered by any boxes. Depending on the orientation of the boxes, this number increases to $k \bmod 2n$ or $k \bmod 4n$. We propose to incorporate these quantities into the objective function. Another idea is to look at the boundary faces of the union of all INSIDE cells. Large planar faces indicate a regular structure that is profitable for packing purposes.

We used a brute-force approach to compute an optimal placement of the grid. This approach can easily be replaced by local search methods, which are likely to produce equal (or better) grid placements in shorter time. Note that the objective function is discrete; thus there is no notion of gradients. We propose to use local optimization methods confined to function evaluations only. Suitable algorithms are for example the downhill simplex algorithm by NELDER and MEAD [NM65], POWELL's method [Pow65] or pattern search [HJ61]. Implementations of the first two algorithms can be found in [PTVF99]. Note that these algorithms compute local optima, and not necessarily the global optimum. Several restarts with a randomly chosen starting point should produce a grid placement close to the global optimum. Such an approximation of the global optimum is sufficient in our case.

Different Box Geometries

Although the size of the boxes is defined in DIN 70020 [Deu93] as exactly 200mm × 100mm × 50mm, other interpretations of this standard exploit tolerances specified in ISO 3832 [Int02] to reduce the effective size of the boxes. The latter standard permits tolerances of up to 1mm for the side lengths of the boxes. Hence the boxes can be as small as 199mm × 99mm × 49mm. Moreover, the edges of the boxes can be rounded to some extent. To be more precise, edges and vertices may be replaced by surface patches from cylinders and spheres with a radius of up to 10mm. These modifications allow to pack more boxes into the trunk. Since the volume of such a box is still counted as one liter, the luggage capacity of the trunk increases.

Unfortunately, the new boxes (without rounded edges) no longer have side length ratios of 4 : 2 : 1 and therefore do not fit on a cubic grid. The best we can do is packing boxes of size 199mm × 99.5mm × 49.75mm by using grids with spacing 49.75mm (24.875mm, 12.4375mm) instead of 50mm (25mm, 12.5mm).

Boxes with rounded edges are advantageous near the boundary of the trunk. The use of such modified boxes is most likely to increase the number of boxes that can be packed. The discretization process can be extended to

handle the new item shape as follows: Conceptually, all cells of the grid are replaced by rounded cubes with the same extension. This modification leaves tiny regions of the space uncovered. Hence it is possible that a box aligned with $4n \times 2n \times n$ INSIDE cells still intersects the trunk boundary or given boxes. It is necessary to maintain a blacklist of such box placements. The integration of rounded boxes requires an extension of the existing intersection predicates. Besides the cases *triangle–box* and *box–box*, we also need to handle the cases *triangle–rounded box* and *rounded box–rounded box*.

Bibliography

[Alg] Algorithmic Solutions. *LEDA*. `http://www.algorithmic-solutions.com/`.

[ANM96] Andrea L. Ames, David R. Nadeau, and John L. Moreland. *The VRML sourcebook*. John Wiley & Sons, 1996.

[BBPP99] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. The maximum clique problem. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization (Supplement Volume A)*, pages 1–74. Kluwer Academic, 1999.

[Bea85] J. E. Beasley. An exact two-dimensional non-guilllotine cutting tree search procedure. *Operations Research*, 33(1):49–64, 1985.

[BF01] Christoph Baur and Sándor P. Fekete. Approximation of geometric dispersion problems. *Algorithmica*, 30:450–470, 2001.

[BJL+90] Fran Berman, David Johnson, Tom Leighton, Peter W. Shor, and Larry Snyder. Generalized planar matching. *Journal of Algorithms*, 11:153–184, 1990.

[BKOS00] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer, 2nd, revised edition, 2000.

[BP01] Roberto Battiti and Marco Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4):610–637, 2001.

[BPP96] Garvin Bell, Anthony Parisi, and Mark Pesce. *The Virtual Reality Modeling Language*. May 26, 1996. Version 1.0 specification, `http://www.web3d.org/x3d/specifications/vrml/VRML1.0/index.html`.

[BS04] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 3*. Prentice Hall, 2004.

[CC97] Eddie Cheng and William H. Cunningham. Wheel inequalities for stable set polytopes. *Mathematical Programming*, 77:389–421, 1997.

[CFL00] Alberto Caprara, Matteo Fischetti, and Adam N. Letchford. On the separation of maximally violated mod-k cuts. *Mathematical Programming*, 87:37–56, 2000.

[Chv75] Vašek Chvátal. On certain polytopes associated with graphs. *Journal of Combinatorial Theory, Series B*, 18:139–154, 1975.

[COI] *Computational Infrastructure for Operations Research (COIN-OR).* `http://www.coin-or.org/`.

[CSY02] Jonathan Cagan, K. Shimada, and Su Yin. A survey of computational approaches to three-dimensional layout problems. *Computer Aided Design*, 34(8):597–611, 2002.

[DC03] Quan Ding and Jonathan Cagan. Automated trunk packing with extended pattern search. *Virtual Engineering, Simulation and Optimization*, pages 33–41, 2003.

[DD92] Kathryn A. Dowsland and William B. Dowsland. Packing problems. *European Journal of Operational Research*, 56:2–14, 1992.

[Deu93] Deutsches Institut für Normung e.V., editor. *DIN 70020, Teil 1, Straßenfahrzeuge; Kraftfahrzeugbau; Begriffe von Abmessungen.* 1993. February 1993 revision.

[DMO] Erik D. Demaine, Joseph S. B. Mitchell, and Joseph O'Rourke. *The Open Problems Project.* `http://maven.smith.edu/~orourke/TOPP/`.

[DST97] Harald Dyckhoff, Guntram Scheithauer, and Johannes Terno. Cutting and packing. In Martin Dell'Amico, F. Maffioli, and Silvano Martello, editors, *Annotated Bibliographies in Combinatorial Optimization.* John Wiley & Sons, 1997.

[Dyc90] Harald Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44:145–159, 1990.

[Ebe01] David H. Eberly. *3D game engine design: a practical approach to real-time computer graphics.* Morgan Kaufmann, 2001.

[EFK+05] Friedrich Eisenbrand, Stefan Funke, Andreas Karrenbauer, Joachim Reichel, and Elmar Schömer. Packing a trunk - now with a twist. In *Proc. of the 2005 ACM Symposium on Solid and Physical Modeling (SPM'05)*, pages 197–206. ACM, June 2005.

[Erd62] Paul Erdős. On the number of complete subgraphs contained in certain graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 7:459–464, 1962.

[FKT01] Sándor P. Fekete, Ekkehard Köhler, and Jürgen Teich. Optimal FPGA module placement with temporal precedence constraints. In *Design, Automation and Test in Europe*. ACM, 2001.

[FPT81] Robert F. Fowler, Michael S. Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are NP-complete. *Information Processing Letters*, 12:133–137, 1981.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[GLM96a] Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. OBB-Tree: a hierarchical structure for rapid interference detection. Technical Report TR96-024, Department of Computer Science, University of Chapel Hill, 1996.

[GLM96b] Stefan Gottschalk, Ming C. Lin, and Dinesh Manocha. OBB-Tree: a hierarchical structure for rapid interference detection. In *Proc. of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. ACM, 1996.

[GS86] A. M. H. Gerards and Alexander Schrijver. Matrices with the Edmonds-Johnson property. *Combinatorica*, 6:365–379, 1986.

[HC95] Eleni Hadjiconstantinou and Nicos Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European Journal of Operational Research*, 83:39–56, 1995.

[Hel23] Eduard Helly. Über Mengen konvexer Körper mit gemeinschaftlichen Punkten. *Jahresbericht der Deutschen Mathematikerververeinigung*, 1923.

[HJ61] Robert Hooke and T. A. Jeeves. "Direct search" solutions of numerical and statistical problems. *Journal of the ACM*, 8:212–229, 1961.

[HM85] Dorit S. Hochbaum and Wolfgang Maass. Approximation schemes for covering and packing problems in image processing and VLSI. *Journal of the ACM*, 32:130–136, 1985.

[ILO] ILOG. *CPLEX*. `http://www.ilog.com/`.

[ILO03a] ILOG. *ILOG CPLEX 9.0 Callable Library: Reference Manual*. ILOG, October 2003.

[ILO03b] ILOG. *ILOG CPLEX 9.0: User's Manual*. ILOG, October 2003.

[Int02] International Organization for Standardization, editor. *ISO 3832, Passenger cars – Luggage compartments – Method of measuring reference volume*. 2002. Third edition.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[MM65] J. W. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3:23–28, 1965.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculation by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[Nel93] Josef Nelissen. New approaches to the pallet loading problem. Technical Report Nr. 155, RWTH Aachen, July 1993.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[NS92] George L. Nemhauser and G. Sigismondi. A strong cutting plane/branch-and-bound algorithm for node packing. *Journal of the Operational Research Society*, 43:443–457, 1992.

[Ope] *OpenGL*. `http://www.opengl.org/`.

[Pad73] Manfred W. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.

[Pow65] M. J. D. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *Computer Journal*, 7:155–162, 1965.

[PTVF99] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1999.

[Rie05] Jens H. Rieskamp. Automation and optimization of Monte Carlo based trunk packing. Diploma thesis, Universität des Saarlandes, Saarbrücken, 2005.

[Rob86] J. M. Robson. Algorithms for maximum independent set. *Journal of Algorithms*, 7:425–440, 1986.

[Sch97] Jörg Schepers. *Exakte Algorithmen für orthogonale Packungsprobleme*. PhD thesis, Universität Köln, 1997.

[Soc01] Society of Automotive Engineers, editor. *SAE J1100, Motor Vehicle Dimensions*. 2001. February 2001 revision.

[SP92] Paul E. Sweeney and Elizabeth Ridenour Paternoster. Cutting and packing problems: A categorized, application-oriented research bibliography. *Journal of the Operational Research Society*, 43(7):691–706, 1992.

[SVA00] Tycho Strijk, Bram Verweij, and Karen Aardal. Algorithms for maximum independent set applied to map labelling. Technical Report UU-CS-2000-22, Universiteit Utrecht, 2000.

[Tro] Trolltech. *Qt*. `http://www.trolltech.com/`.

[Tro75] L. E. Trotter. A class of facet producing graphs for vertex packing polyhedra. *Discrete Mathematics*, 12:373–388, 1975.

[TT77] R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6:537–546, 1977.

[TT97] Virginia Torczon and Michael W. Trosset. From evolutionary operation to parallel direct search: Pattern search algorithms for numerical optimization. *Computer Science and Statistics*, 29:396–401, 1997.

[TTS94] Alexander Tarnowski, Johannes Terno, and Guntram Scheithauer. A polynomial time algorithm for the guillotine pallet loading problem. *Information Systems and Operations Research*, 32:275–287, 1994.

[WNDS01] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison Wesley, 3rd edition, 2001. The Official Guide to Learning OpenGL, Version 1.2.

[YC00] Su Yin and Jonathan Cagan. An extended pattern-search algorithm for three-dimensional component layout. *ASME Journal of Mechanical Design*, 122:102–108, March 2000.

[ZE00] Afra Zomorodian and Herbert Edelsbrunner. Fast software for box intersections. In *Proc. of the 16th Annual Symposium on Computational Geometry (SCG'00)*, pages 129–138, 2000.