Department of Computer Science
University of Copenhagen

Submitted: October 6, 2005
Adviser: David Pisinger
Master thesis for the cand. scient. degree in computer science

Rasmus Resen Amossen

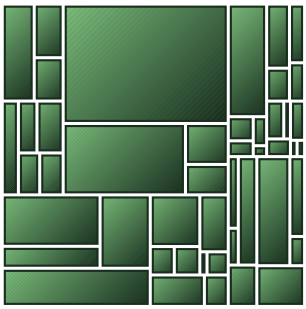
Constructive algorithms and lower bounds

for guillotine cuttable orthogonal bin packing problems

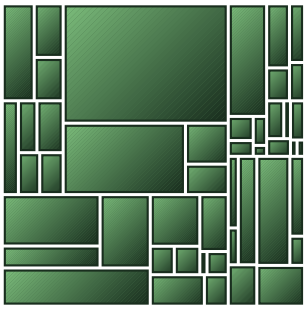
Abstract

The d-dimensional bin packing problem (OBPP-d) is the problem of finding the minimum number of containers needed to contain a set of orthogonally packed d-dimensional rectangular boxes. In OBPP-d solvers two subproblems are crucial: Calculating lower bounds and solving the decision problem (OPP-d) of determining if a set of boxes can be orthogonally packed into a single container. This thesis focuses on these two subproblems with an extra requirement attached: All packings must be guillotine cuttable. That is, the containers must be able to be split into n pieces, each holding a box, by recursively cutting them with face orthogonal cuts.

We present an extension of a framework by Fekete and Schepers for the guillotine cutting requirement, prove that the decision problem is NP-hard and prove a worst-case performance for the corresponding OBPP-2. A new type of packing property, sticky cutting, is presented and an algorithm for sticky cuttings based on the framework by Fekete and Schepers is described. Inspired by the nature of guillotine packings, a new tree representation eliminating redundancy is presented along with a proof-of-concept brute-force algorithm for generating all such trees.



Contents



List of Figures

Chapter 1
Introduction

In the following we consider the problem of packing d-dimensional boxes into d-dimensional containers. Let therefore V be a set of boxes and let $w : V \rightarrow \mathbb{R}_0^{+d}$ be a size function describing the width of each box in all dimensions $x_1, x_2, \ldots, x_d$. All containers are restricted to have the same size $W \in \mathbb{R}_0^{+d}$. Now we can loosely define a feasible *packing* as an arrangement of the boxes V into one or more containers so that no boxes overlap and no box exceed the boundaries of the container in which it is placed. Rotations are not allowed and box edges must be parallel to container edges.

Several problems can be formulated from the task of packing boxes into containers. Some essential ones include:

Orthogonal Packing Problem (OPP-d) Given a set of boxes V, can V be packed into a single container of size W? OPP-d is NP-hard [FS97a].

Orthogonal Bin Packing Problem (OBPP-d) Given the set V, how few containers of size W is required for packing all the boxes in V? Since OPP-d is the corresponding decision problem for OBPP-d and OPP-d is NP-hard also OBPP-d is NP-hard.

Orthogonal Knapsack Problem (OKP-d) Given a single container of size W and a value function $v : V \rightarrow \mathbb{R}_0^+$, choose a subset $V' \subseteq V$ that can be packed into the container so the sum of values for V' are maximized. Again OPP-d is the corresponding decision problem so OKP-d is NP-hard as well.

This thesis focus on the problems OPP-d and OBPP-d attached with an extra so called *guillotine cutting* restriction. In this, a packing is only feasible if there exist a series of face parallel straight cuts that can recursively cut

the container into $|V|$ pieces so that each piece contains a box and no box has been intersected.

The OPP-d has been approached in numerous ways. In [MFNK96] Murata et al. presents a method for representing box placements using *sequence pairs*. This representation is later utilized by Pisinger [Pis03] who builds packings for $d = 2$ by considering *envelopes* that cover all boxes placed so far. In [MPV97] Martello et al. presents an algorithm for $d = 3$ that uses envelopes as well. Pisinger and Sigurd [PS02] presents an algorithm for OPP-2 based on constraint programming (CSP) in which a packing is constructed by assigning appropriate relations to each pair of boxes $u, v \in V$. This algorithm is also described in section 6.3.3. Martello et al. [MPV04] uses the CSP technique for the three-dimensional case. All of the above consider the decision problem for a specific d. A general model for all values of d, based on graph theory, is presented by Fekete and Schepers in [FS97a]. The model is relatively complex and introduces the concept of packing isomorphism. [FS97c] describes a branch-and-bound algorithm based on this model and both the model and this algorithm is fundamental for this thesis. None of the above papers deal with packings required to satisfy the guillotine cutting property. This is however done in work by Wong and Liu [WL86] who model guillotine cuts as a tree structure and represents the tree structure by *normalized polish expressions*. Also Christofides and Whitlock [CW77] uses tree structures to model guillotine cuts. Both [WL86] and [CW77] consider guillotine cuttings from a top-down point of view where different cut configurations are applied to a container until a cut structure that can hold all boxes has been found. Where the previous two articles uses a top-down strategy Wang [Wan83] iteratively assembles the boxes in V in a bottom-up approach.

As seen, several kinds of algorithms have been published for OPP-d but primary two methods are used to solve OBPP-d. One is to distribute boxes into containers with a branch-and-bound algorithm. Such an algorithm is presented by Martello et al. [MPV97] for OBPP-3 but the framework is in fact more general: An outer branch-and-bound algorithm assigns boxes to containers and some inner algorithm solves the OPP-d problem for each such assignment to test that the boxes fit into the respective containers. The other method is to formulate the problem as a mixed integer model. Pisinger and Sigurd [PS02] concern OBPP-2 and use the idea from Dantzig-Wolfe [DW60] to decompose the model into a restricted master problem and a number of subproblems. Each of the subproblems is then split again into a one-dimensional pricing problem and a two-dimensional decision problem. Notice that both methods contain OPP-d as a subproblem.

An efficient way of optimizing algorithms for OPP-d and OBPP-d is to consider lower bounds which can be calculated in polynomial and maybe even linear time: E.g. if an iteration makes a box assignment to k containers and a lower bound gives that the assigned boxes require at least $k+1$ containers there are no need to solve the NP-hard OPP-d problem with an exact solver. Most bounds are based on the same principle of grouping boxes into classes depending on their size and combining various volume estimations for each class. Such bounds are presented in [MT90, DM95, MV98] for $d = 1$, in [BM03] for $d = 2$ and in [MPV97] for $d = 3$. None of these bounds apply to the general d-dimensional problem. Fekete and Schepers describe in [FS97b, FS01] a generalized strategy of scaling box sizes by *conservative scales* and estimating on the transformed sizes. This generalized idea is described for arbitrary d.

No polynomial time bound taking the guillotine cutting requirement into account is, to the best of our knowledge, known. One way of achieving a lower bound, not polynomial though, for packings with this requirement is to settle with the LP-relaxation of the solution found by the algorithm by Pisinger and Sigurd [PS02]. This algorithm is capable of handling both the guillotine cutting restriction and various other requirements.

1.1 Outline

As mentioned above, every currently known algorithm for OBPP-d consist of an outer branch-and-bound or column genaration framework that assigns boxes to containers and an inner sub-algorithm for solving OPP-d for each container. The performance of all algorithms for the bin packing problem therefore heavily depends on the performance of the solver used for the decision problem. This thesis will thus focus on OPP-d and more specifically the graph theory based framework by Fekete and Schepers. Section 1.3 introduces some essential graph theoretic terms and results needed for chapter 2 in which the model and OPP-d algorithm by Fekete and Schepers are described.

The original framework does not take various packing restrictions, such as the guillotine cutting requirement, into account. In chapter 3 we examine how the model behaves if all packings are known to be guillotine cuttable and describe how the algorithm by Fekete and Schepers can be applied to problems with this requirement. The chapter also states a worst-case performance for the guillotine cuttable OBPP-2 and introduces a new *packing tree* representation.

A subset of all guillotine packings behave particularly nice in the graph

theoretic model: In chapter 4 we introduce the concept of *sticky cuttings* and describe how to apply Fekete and Schepers framework to these kind of restrictions. We also give a conjecture for the worst case performance for the sticky cuttable OBPP-2 and describe how the idea from sticky cuttings can be utilized for the special case OPP-2 where the number of *stages* allowed for the cutting recursion is limited to 2.

Some of the presented ideas have been implemented and the computational results obtained are presented in chapter 5.

In chapter 6 we give a survey of various techniques to obtain lower bounds for OBPP-d. We also describe an algorithm by Pisinger and Sigurd [PS02] capable of providing a lower bound for problems required to be guillotine cuttable.

All the previous chapters are coupled in chapter 7 in which a branch-and-bound algorithm for OBPP-d utilizing both exact solvers for OPP-d and lower bounds OBPP-d is described.

Chapter 8 contains a conclusion and ideas for further work.

1.2 Contributions

This thesis provides the following contributions:

- The framework by Fekete and Schepers is extended to handle guillotine cuttings and two versions of the modified algorithm is presented: One that follows the original framework relatively strictly but ensures the guillotine property and one in which the guillotine checking extension is utilized to speed up each iteration.

- A proof for a worst case performance ratio of 2 for the guillotine cuttable OBPP-2 is given. This proof was made in co-operation with David Pisinger.

- A whole new type of graph theoretically nice behaving *sticky cuttings* is presented and a conjecture is given for a worst case performance ratio of 4 for the sticky cuttable OBPP-2.

- It is shown how the theory from sticky cuttings can be utilized in conventional guillotine cuttings for $d = 2$ if the number of stages is limited to 2.

- The sticky cutting algorithm and one of the guillotine algorithms have been implemented. These implementations are the first to solve sticky- and guillotine cuttings for arbitrarily large d.
- A new *packing tree* representation of guillotine cuttings is presented. Packing trees prevent redundant/symmetric representations and is described along with a brute-force algorithm that – as a proof-of-concept – traverse all packing trees.

1.3 Graph theoretic terms and results

The framework presented in the upcoming chapters rely on a set of graph theoretic definitions and results. These will be briefly explained in this section.

Definition 1.1 (Induced subgraph) *Let $G = (V, E)$ be a graph. For $S \subseteq V$ we denote by $G[S] = (S, E[S])$ the* by S induced subgraph of G, *where $(u, v) \in E[S] \Leftrightarrow u, v \in S$ and $(u, v) \in E$.*

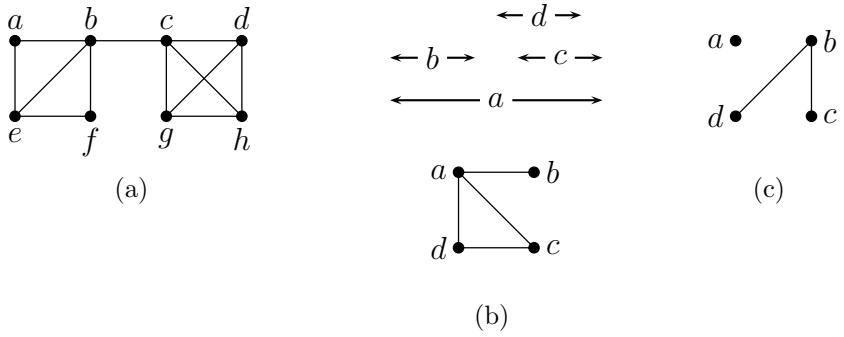


Figure 1.1: 1.1(a) is a graph with 8 vertices refered to from several examples attached to the graph definitions. 1.1(b) shows a set of intervals (above) and the corresponding interval graph (below). The comparability graph for the interval graph is shown in 1.1(c).

Example 1.2 In figure 1.1(a) the subgraph induced by the node set $\{c, d, g, h\}$ is the complete graph k_4. ◇

Definition 1.3 (Stable set) *For a graph $G = (V, E)$, a set $S \subseteq V$ is called a* stable set of G *if the by S induced subgraph of G has no edges.*

Example 1.4 The nodes $\{a, c, f\}$ form a stable set in figure 1.1(a). ◇

Definition 1.5 (k-chord) *Let $C = (v_0, v_1, \ldots, v_n = v_0)$ be a cycle. An edge $(v_i, v_j), i, j \in \{0, \ldots, k-1\}$ is called a $k-$chord if $(|i - j| \mod n) = k$ and $k > 1$. A cycle is called* k-chordless *if it does not contain a k-chord. A chordless cycle may also be referred to as a* hole.

Example 1.6 Consider figure 1.1(a) and the cycle (a, b, f, e). The edge (e, b) is a 2-chord in this cycle. ◇

Definition 1.7 (Clique, maximal-) *For $S \subseteq V$ the induced subgraph $G[S]$ is called a* clique *of order $|S|$ if $G[S]$ is complete. A clique $G[S]$ is* maximal *if there is no other clique $G[S']$ with $|S'| > |S|$.*

Example 1.8 In figure 1.1(a) the subgraph induced by the nodes $\{a, b, e\}$ is a clique but it is not maximal. The clique induced by the set $\{c, d, g, h\}$ is maximal. ◇

Definition 1.9 (Interval graph) *Let I be a set of intervals on $\mathbb{R}$. The undirected graph $G = (V, E)$ is an* interval graph *if there is a bijection between I and V and $(u, v) \in E \Leftrightarrow u \cap v \neq \emptyset$ for all $u, v \in I$.*

Example 1.10 Figure 1.1(b) shows a set of intervals and the corresponding interval graph. ◇

When considering directed graphs, the number of out- and in-going edges for a node v is often relevant:

Definition 1.11 (In-degree) *Let $G = (V, E)$ be a directed graph. Then the number of in-going edges for a node v is denoted $\delta^-(v)$. $\delta^-(v)$ is also referred to as the* in-degree *of v.*

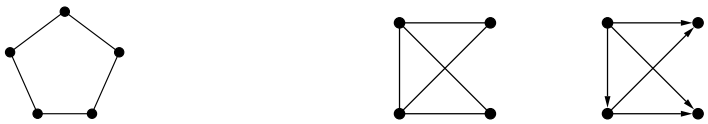
Later, we need to look at directed versions of undirected graphs satisfying the so called *transitive orientation property*:

Definition 1.12 (Transitive orientation) *A graph $G = (V, E)$ is said to satisfy the* transitive orientation property *if there exists an orientation F of E such that the following condition holds:*

$$(u, v) \in F \text{ and } (v, w) \in F \Rightarrow (u, w) \in F. \tag{1.1}$$

An undirected graph satisfying (1.1) *is also referred to as a* comparability graph.

Figure 1.2 shows two examples of graphs and the transitive orientation property.



(a) (b) (c)

Figure 1.2: The graph in 1.2(a) cannot be transitively oriented while 1.2(b) can and may become 1.2(c).

Let $E^{\complement}$ denote the complement of E.

Proposition 1.13 (Ghoulia-Houri) *Let $G = (V, E)$ be an interval graph. Then $G^{\complement} = (V, E^{\complement})$ is a comparability graph. That is, $G^{\complement}$ satisfies the transitive orientation property.*

Proof Let $\{I_v\}_{v \in V}$ be an interval representation of G. We need to show that an orientation of $E^{\complement}$ satisfying (1.1) exists. For every edge (u, v) in $E^{\complement}$, I_u and I_v are disjoint, per definition. If $I_u \leftarrow I_v$ denotes that I_u lies strictly to the left of I_v, we can consider the following orientation F of $E^{\complement}$: For all $(u, v) \in E^{\complement}$ let

$$(u, v) \in F \Leftrightarrow I_u \leftarrow I_v.$$

As $I_u \leftarrow I_v \leftarrow I_w \Rightarrow I_u \leftarrow I_w$, (1.1) clearly holds. □

Definition 1.14 (Cocomparability graph) *A graph $G = (V, E)$ is called a* cocomparability graph *if $G^{\complement} = (V, E^{\complement})$ is a comparability graph.*

Chapter 2

Modelling the packing problem

Most of the theory behind the constructive algorithms in this thesis relies on a graph theoretic model developed by Fekete and Schepers [FS97a, FS04]. In [FS97c] they present a branch-and-bound algorithm for solving OPP-d based on this model and both the model and algorithm will be described in this chapter.

Throughout the rest of this text we denote by ϕ_i the i'th coordinate of ϕ for a map ϕ with a d dimensional image.

2.1 Modelling OPP-d

We aim to formalize the idea of packings by considering the positions of all boxes in a Cartesian coordinate system. For that, define the map $p : V \to \mathbb{R}_0^{+d}$ as the coordinate of the corner closest to the origin of each box. p then uniquely identifies the box position as no rotations are allowed. Also define

$$\begin{aligned} I_i : V &\to \mathbb{R}_0^+ \times \mathbb{R}^+ \\ v &\mapsto [p_i(v), p_i(v) + w_i(v)) \end{aligned} \tag{2.1}$$

as the interval occupied by box v on the x_i-axis. With p and I in hand we can now define a packing formally:

Definition 2.1 (Packing) *A feasible* packing *of the tuple* (V, w, W) *is a function* $p : V \to \mathbb{R}_0^{+d}$ *that satisfies*

$$\forall v \in V : \quad p(v) + w(v) \leq W \tag{2.2}$$

$$\forall u, v \in V, u \neq v, \exists i \in \{1, \ldots, d\} : \quad I_i(u) \cap I_i(v) = \emptyset \tag{2.3}$$

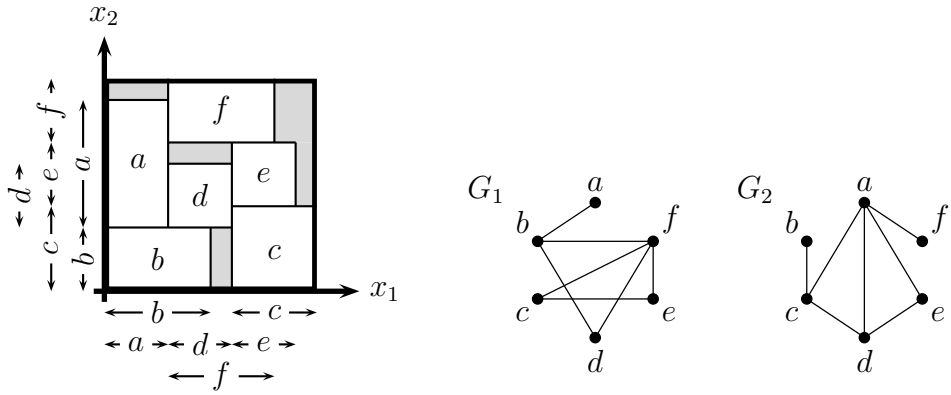
(2.2) ensures that no box exceeds the container boundaries and (2.3) ensures that no two boxes overlap. Figure 2.1(a) shows a feasible packing.

If a packing p arranges the boxes so they are either placed at origin or at the edge of another box we call it gapless. More formally:

Definition 2.2 (Gapless packing) *A packing is said to be* gapless *if for all* $i = 1, \ldots, d$ *and* $v \in V$

$$p_i(v) = 0 \text{ or } \exists u \in V : p_i(v) = p_i(u) + w_i(u)$$

Interval graphs can be used to represent the map I defined in (2.1). This is done by constructing an interval graph $G_i = (V, E_i)$ for each dimension $i = 1, \ldots, d$ for the map I_i: Two vertices u and v are connected in G_i if and only if box u and v is overlapping along the x_i-axis. Figure 2.1 illustrates a packing and the corresponding interval graphs. Notice, that a stable set in G_i corresponds to a subset of boxes that does not overlap along the x_i-axis. Below we will show that interval graphs can be used to represent packings.



(a) A packing and the intervals for each box

(b) The corresponding interval graphs

Figure 2.1: A gapless packing for $d = 2$ and the corresponding interval graphs

We call a subset of boxes $S \subseteq V$ for x_i-*feasible* if S can be lined up along the x_i-axis without overlapping and without exceeding the container width. That is, if $\sum_{v \in S} w_i(v) \leq W_i$.

Now, for the graphs $G_i = (V, E_i)$, $i = 1, \ldots, d$, consider the following three properties. They are crucial for the entire algorithmic part of this thesis and are therefore highlighted:

P1 Each $G_i = (V, E_i)$ is an interval graph

P2 Each stable subset $S \subseteq V$ of G_i is x_i-feasible

P3 $\bigcap_{i=1}^{d} E_i = \emptyset$

We will now show that satisfying the above three properties is both a necessary and sufficient condition for the graphs $G_1, \ldots, G_d$ to represent a feasible packing. For this, we will make use of the term *packing class*:

Definition 2.3 (Packing class) *The tuple $E = (E_1, \ldots, E_d)$ of edge sets for a set V of boxes is called a* packing class *if and only if P1, P2 and P3 are satisfied for all $G_i = (V, E_i)$.*

Theorem 2.4 *Let p be a packing of the tuple (V, w, W). Then there exists a packing class E for p.*

PROOF Our approach is to construct a set of graphs and show that these satisfy P1, P2 and P3. For $i = 1, \ldots, d$ let $G_i = (V, E_i)$ be the interval graphs induced by the intervals I_i as defined in (2.1). P1 is therefore trivially satisfied for all G_i. For an arbitrary i, let $S \subseteq V$ be a stable set. Because of (2.2) and because G_i is an interval graph, the intervals in the stable set S are disjoint with the rightmost element to the left of W_i. Thus $\sum_{s \in S} w_i(s) \leq W_i$ or equivalent: S is x_i-feasible. P2 is therefore satisfied. Last, we can rewrite (2.3) as $\forall u, v \exists i \in \{1, \ldots, d\} : (u, v) \notin E_i$ which implies P3. □

That the converse is also true is shown in theorem 2.5 below. The theorem makes use of a map p_i^F that defines box positions on basis of a transitive orientation. Let $F = (F_1, \ldots, F_d)$ be a transitive orientation and for $i = 1, \ldots, d$ and $v \in V$, define the map $p_i^F : V \to \mathbb{R}_0^+$ as

$$v \mapsto \begin{cases} 0 & \text{if } \nexists u \in V : (u, v) \in F_i \\ \max\{p_i^F(u) + w_i(u) | (u, v) \in F_i\} & \text{else} \end{cases}$$

The map $p^F : V \to \mathbb{R}_0^{+d}$ induced by p_i^F arranges the boxes in a gapless packing:

Theorem 2.5 *Let $E = (E_1, \ldots, E_d)$ be a packing class for (V, w, W). Then there exist a gapless packing p for (V, w, W).*

PROOF We will show that for any transitive orientation F of $E^{\complement}$, the map p^F describes a feasible gapless packing.

First of all, because of P1, $G_i = (V, E_i)$ is an interval graph for all $i = 1, \ldots, d$. So proposition 1.13 on page 7 tells us that $G_i^{\complement} = (V, E_i^{\complement})$ is a comparability graph and a transitive orientation F therefore always will exist for $E_i^{\complement}$. For such an orientation F we need to show that (2.2) and (2.3) holds for p^F.

To show (2.2) choose an arbitrary $v \in V$ and $i \in \{1, \ldots, d\}$. F will then contain a path, $(v^{(0)}, \ldots, v^{(r)} = v)$ with $\delta^-(v^{(0)}) = 0$ where $\delta^-(v)$ denotes the in-degree of v as defined in definition 1.11 on page 6. Because of the construction of p^F we have

$$p_i^F(v) + w_i(v) = \sum_{k=0}^{r} w_i(v^{(k)}).$$

F is a transitive orientation so $S = \{v^{(0)}, \ldots, v^{(r)}\}$ induces a clique in $G_i^F = (V, F)$ and therefore also in $G^{\complement}$. Thus, S must be a stable set in G. For the stable set S P2 now gives

$$\sum_{u \in S} w_i(u) = p_i^F(v) + w_i(v) \leq W_i$$

which proves (2.2) as $v \in V$ was arbitrarily chosen.

To prove (2.3) we need to show that there exist some hyper plane separating every two boxes $u, v \in V$. So let $u, v \in V, u \neq v$ be such two arbitrary chosen boxes. According to P3 there exists an i with $(u, v) \notin E_i$. Per definition we thus have $(u, v) \in E_i^{\complement}$. As F is an orientation of $E_i^{\complement}$ either $(u, v) \in F$ or $(v, u) \in F$. I.e. $p_i^F(v) \geq p_i^F(u) + w(u)$ or $p_i^F(u) \geq p_i^F(v) + w(v)$. In either case $I_i(u) \cap I_i(v) = \emptyset$ so u and v can be separated by the hyper plane orthogonal to x_i. This proves (2.3).

That p^F is gapless is immediately implied by construction of p^F. □

The intuition behind packings represented as graphs can be explained as follows: Given a set of d interval graphs we can tell which boxes that overlap along each axis by looking at an appropriate graph. The complements to the interval graphs are comparability graphs in which two vertices are connected if the corresponding boxes does *not* overlap. Orienting the edges in the comparability graphs is equivalent to placing non-overlapping boxes left/right, above/under or behind/in front of each other. Clearly the orientations must satisfy the transitive orientation property (definition 1.12 on page 7): If a

box v_1 is strictly to the left of another box v_2 and v_2 is strictly to the left of v_3 then also v_1 must be strictly to the left of v_3.

Figure 2.2 shows an example on how a packing can be constructed from a packing class. In 2.2(c) a transitive orientation is chosen which leads to the arrangement in 2.2(d). Another transitive orientation of 2.2(c) would lead to another arrangement and in this way a single packing class represents a whole equivalence class of arrangements. Notice that a transitive orientation can be found in $O(|V|+|E|)$ time [MS97] and thus a packing can be constructed in linear time in the number of edges. So in order to solve OPP we can search for a packing class instead of a packing. In section 2.2 we develop a set of rules to determine whether a set of graphs represents a packing class or not.

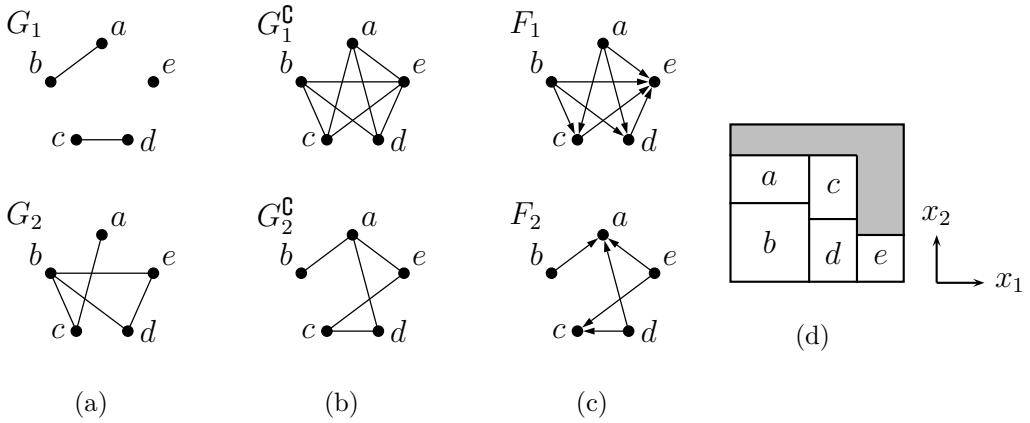


Figure 2.2: The figure shows the construction of a packing for $W = (11, 10)$ and $V = \{a, b, c, d, e\}$ where $w(a) = (5, 3)$, $w(b) = (5, 5)$, $w(c) = (3, 4)$, $w(d) = (3, 4)$ and $w(e) = (3, 3)$. 2.2(a) shows two interval graphs representing a packing class. Their complements which are comparability graphs are shown in 2.2(b). 2.2(c) shows a transitive orientation F of G. The packing induced by F and p^F where F_1 determines the relative horizontal positions and F_2 the relative vertical positions is shown in 2.2(d).

2.1.1 Properties of packing classes

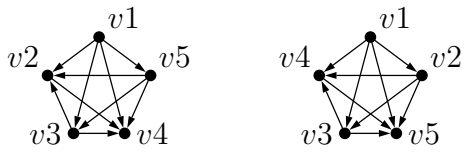
As seen above, packing classes are very useful in the OPP context as each packing class represent a whole set of packings. In this section we will explore the nature of packing classes and present a set of properties that must be satisfied in order for an edge set to be a packing class. These properties are

formulated in theorem 2.8 and 2.9. The theorems rely on a number of results, some of which will be presented in the lemmas below. The first one ensures the existence of a certain ordering of the vertices in a transitive oriented complete graph:

Lemma 2.6 *Let F be a transitive orientation of a complete graph $G = (V, E)$ and let $\delta^-(v_i)$ denote the in-degree of v_i. Then the vertices in V can be ordered such that*

1. $\delta^-(v_i) = i - 1$ *in* F
2. $(v_i, v_j) \in F \Leftrightarrow i < j$

The lemma is a part of theorem 2.4 in [Gol80] which also contains a proof. We will not cover the proof here but figure 2.3 shows an example of the lemma in practice.



(a) (b)

Figure 2.3: 2.3(a) shows a transitive orientation of a complete directed graph. As lemma 2.6 explains, the vertices can be reordered so $\delta^-(v_i) = i - 1$ and $(v_i, v_j) \in F \Leftrightarrow i < j$. Such a relabeling is seen in 2.3(b).

The second lemma tells us about edge connections between certain maximal cliques:

Lemma 2.7 *Let $G = (V, E)$ be a graph without chordless 4-cycles and let K_1 and K_2 be maximal cliques. Let F be a transitive orientation of $E^{\complement}$. Then the following holds:*

1. *F contains an edge connecting K_1 and K_2*
2. *All edges that connects K_1 and K_2 has the same orientation in F*

PROOF (1) Assume that no edge connects K_1 and K_2 in F. Then, all vertices in K_1 will be connected with all vertices in K_2 in E, which means that $K_1 \cup K_2$ is a clique. This contradicts that K_1 and K_2 where maximal.

(2) Assume the opposite: That there exists opposite oriented edges $(a, b) \in F$ and $(d, c) \in F$ where $a, c \in K_1$ and $b, d \in K_2$. If $a = c$ or $b = d$ the transitive property would imply that $(d, b) \in F$ or $(c, a) \in F$ resp. which is not true as $(a, c) \in E$ and $(b, d) \in E$. See figure 2.4(a). So assume

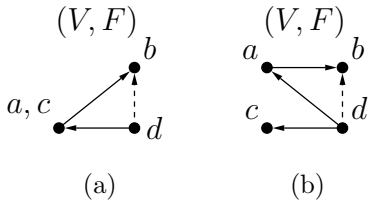


Figure 2.4: 2.4(a) The dashed edge shows that two equal vertices would lead to a contradiction because the edge is in E. 2.4(b) The dashed edge shows the influence of $(d, a) \in F$.

a, b, c, d are 4 different vertices. Because G has no chordless 4-cycle we have $(a, d) \in E^{\complement}$ or $(c, b) \in E^{\complement}$. Otherwise (a, d, b, c, a) would have been a cycle in E. Assume that $(a, d) \in E^{\complement}$. Because of the transitivity in F we have $(a, d) \in F \Rightarrow (a, c) \in F$ and $(d, a) \in F \Rightarrow (d, b) \in F$. See figure 2.4(b). Both statements are false, so all edges connecting K_1 and K_2 must have the same orientation. □

With lemma 2.6 and 2.7 in hand we are ready to prove the following theorem which states a set of properties that packing class are guaranteed to satisfy:

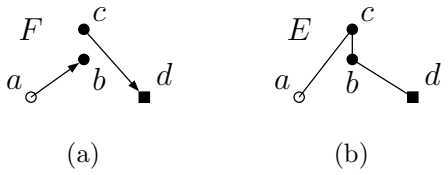
Theorem 2.8 *Let $G = (V, E)$ be an undirected graph. Then the following statements are equivalent:*

1. *G is an interval graph*
2. *$G^{\complement}$ is a comparability graph and G does not contain a chordless C_4 cycle as an induced subgraph*
3. *There exists an ordering of the maximal cliques in G such that for all $u \in V$, the maximal cliques containing v, occur consecutively.*

PROOF $1 \Rightarrow 2$ In proposition 1.13 on page 7 we showed that $G^{\complement}$ is a comparability graph. So let $G = (V, E)$ be an interval graph and assume that G indeed has a chordless cycle $C = (v_0, v_1, \ldots, v_n = v_0)$ with $n \geq 4$. For any $k \in \{0, \ldots, n-2\}$ we have $I(v_k) \cap I(v_{k+1}) \neq \emptyset$ but $I(v_k) \cap I(v_{k+2}) = \emptyset$. This

implies that $\max\{I(v_k)\} < \min\{I(v_{k+2})\}$ or $\min\{I(v_k)\} > \max\{I(v_{k+2})\}$, i.e. the intervals are either strictly increasing or decreasing. Especially we have $I(v_0) \cap I(v_{n-1}) = \emptyset$, which contradicts the undirected edge $(v_{n-1}, v_n) = (v_0, n_{n-1})$. Therefore G cannot have a cordless cycle of length 4 or more.

$2 \Rightarrow 3$ Let $\mathscr{C}$ be the set of all maximal cliques in G and introduce the relation $<$ where $K_1 < K_2$ if and only if $\exists(u, v) \in F : u \in K_1, v \in K_2$. According to lemma 2.7 on page 14 there exists edges in F connecting all cliques and all edges connecting two cliques have the same orientation. Therefore we have a complete orientation of $\mathscr{C}$, a so called *tournament*. If we can show that the orientation of $\mathscr{C}$ is transitive then lemma 2.6 on page 14 tells that an ordering on $\mathscr{C}$ also exists. So, given three cliques K_1, K_2 and K_3 assume that $K_1 < K_2$ and $K_2 < K_3$, i.e. for $a \in K_1$, $b, c \in K_2$ and $d \in K_3$ we have $(a, b) \in F$ and $(c, d) \in F$. See figure 2.5(a). We need to



(a) (b)

Figure 2.5: Illustrations for theorem 2.8.

show that $(a, d) \in F$. If $(a, c) \in F$ or $(b, d) \in F$, also $(a, d) \in F$ because of transitivity of F. So assume this is not the case and that $(a, c) \in E$, $(b, c) \in E$ and $(b, d) \in E$. See figure 2.5(b). E contains no chordless 4-cycle so $(a, d) \in E^{\complement}$. The transitivity of F gives us $(d, a) \in F \Rightarrow (c, a) \in F$, so because $(c, a) \in E \Rightarrow (c, a) \notin E^{\complement}$ we have $(a, d) \in F$. That is, $K_1 < K_3$ and $(\mathscr{C}, <)$ therefore describes an ordering of $\mathscr{C}$.

Next, assume that $K_1, \ldots, K_n$ is ordered so $K_i < K_j \Leftrightarrow i < j$. We need to show that all cliques containing $u \in V$ occur consecutively: Consider $K_i < K_j < K_k$ and assume there exists $u \in V$ where $u \in K_i$, $u \in K_k$ but $u \notin K_j$. Because $u \notin K_j$ there must exist some $v \in K_j$ where $(u, v) \notin E$, otherwise all vertices in K_j would be connected to u in E contradicting that K_j was maximal. So $(u, v) \in E^{\complement}$. But per definition of $< K_1 < K_2 \Rightarrow (u, v) \in F$ and $K_2 < K_3 \Rightarrow (v, u) \in F$ which is a contradiction. Therefore $u \in K_j$. This proves $2 \Rightarrow 3$.

$3 \Rightarrow 1$ For $v \in V$, let $\mathscr{C}(v)$ be the set of cliques containing v. Regarding to the ordering of $\mathscr{C}$, $\mathscr{C}(v)$ describes an interval on $\mathscr{C}$. We need to show that $(u, v) \in E \Leftrightarrow \mathscr{C}(u) \cap \mathscr{C}(v) \neq \emptyset$. $(\Leftarrow)$: There exists at least one maximal clique that both u and v is in. Per definition $(u, v) \in E$. $(\Rightarrow)$: As $(u, v) \in E$, they are both elements the same maximal clique. I.e. $\mathscr{C}(u) \cap \mathscr{C}(v) \neq \emptyset$. $\square$

The above theorem tells that a graph is an interval graph if it does not contain a chordless C_4 and if the complement is a comparability graph. To tell if a graph is a comparability graph the below theorem can be used:

Theorem 2.9 (Ghouilà-Houri 1962, Gilmore and Hoffman 1964) *A graph is a comparability graph if and only if it does not contain a 2-chordless cycle of odd length.*

The proof requires a relatively large toolbox of terms and lemmas and is therefore omitted here. However, it can be found in e.g. as a part of [Gol80], theorem 5.27. Figure 2.6 shows examples of graphs with and without 2-chordless cycles of odd length.

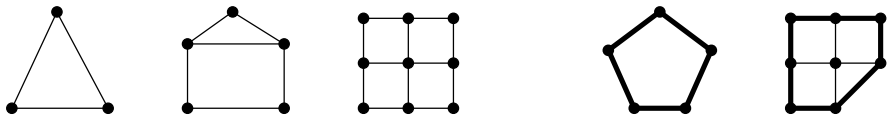


Figure 2.6: 2.6(a)-2.6(c) are examples of graphs without a 2-chordless cycle of odd length. They are therefore all comparability graphs. The 2-chordless cycle of odd length is marked with bold in 2.6(d) and 2.6(e) so neither is a comparability graph.

In order to solve OPP it seems to be a good idea to check for chordless cycles C_4 in G and for 2-chordless cycles of odd length in $G^{\complement}$. Section 2.2 will focus on solving OPP-d based on this idea.

2.2 Solving OPP-d

This section will describe a branch-and-bound algorithm developed by Fekete and Schepers [FS97c] for solving OPP-d based on the properties for packing classes.

The main idea is to determine if we, given the tuple (V, w, W), can construct a packing class. That is, can d edge sets $E_1, \ldots, E_d$ be found so that P1, P2 and P3 are satisfied for all $E_i, i = 1, \ldots, d$? The search for a packing class will be done through a branch-and-bound algorithm where each branch corresponds to the action of fixing the existence of a certain edge e. For each non-leaf node in the search tree there will be two subtrees: The search space below the first subtree only contains packing classes E in which $e \in E$ while

the search space below the second only contains classes E with $e \notin E$. This is done by, in each node N, maintaining two sets $\mathcal{E}_-^N = (\mathcal{E}_{-,1}^N, \ldots, \mathcal{E}_{-,d}^N)$ and $\mathcal{E}_+^N = (\mathcal{E}_{+,1}^N, \ldots, \mathcal{E}_{+,d}^N)$ so that $\mathcal{E}_-^N$ only contains excluded edges and $\mathcal{E}_+^N$ only included edges. Notice that $\mathcal{E}_+^N \cap \mathcal{E}_-^N = \emptyset$ at any node and that

$$\mathcal{E}_{+,i}^N \subseteq E_i \subseteq \mathcal{E}_{-,i}^{N\,\mathtt{C}}$$

for all $i = 1, \ldots, d$. The tuple $\mathcal{E}^N = (\mathcal{E}_+^N, \mathcal{E}_-^N)$ is referred to as the *search information* for N and the whole search space for N is written $\mathcal{L}(\mathcal{E}^N)$. For $\sigma \in \{-, +\}$, adding an edge e to $\mathcal{E}_{\sigma,i}$ is referred to as *augmenting* $\mathcal{E}_{\sigma,i}$. We will also use the notation (e, σ, i) to describe such an augmentation.

The search will stop if $\mathcal{E}_+^N$ is a packing class or if it can be determined that a packing class cannot be obtained without using some of the excluded edges in $\mathcal{E}_-^N$. In the following, we will look at a single node N and will therefore omit N in the notation. I.e. $\mathcal{E}_{+,i}^N$ will be written $\mathcal{E}_{+,i}$.

2.2.1 P1, P2, P3 in practice

Let us have a closer look at each of the properties P1, P2 and P3 and how to build a packing class satisfying them. Consider the search information $\mathcal{E}$ for some node N.

P3 is fairly easy to ensure: If $e \in \mathcal{E}_{+,i}$ for all $i \neq k$ then P3 forces $e \in \mathcal{E}_{-,k}$ in order to achieve $\cap \mathcal{E}_{+,i} = \emptyset$.

Next, P2 requires each stable subset $S \subseteq V$ in $G_i = (V, \mathcal{E}_{+,i})$ to be x_i-feasible. As a stable set S in $\mathcal{E}_{+,i}$ is a clique in $\mathcal{E}_{+,i}^{\mathtt{C}}$ this is equivalent to all cliques in $\mathcal{E}_{+,i}^{\mathtt{C}}$ being x_i-feasible. So if $\mathcal{E}_{-,i}$ has an x_i-*in*feasible clique, no feasible augmentation can be done and the search in the subtree can be terminated. If $\mathcal{E}_{+,i}^{\mathtt{C}}$ contains an x_i-infeasible clique, we must therefore at least add one of its unfixed edges from $\mathcal{E}_{+,i}^{\mathtt{C}} \setminus \mathcal{E}_{-,i}$ to $\mathcal{E}_{+,i}$.

P1 requires each $G_i = (V, \mathcal{E}_{+,i})$ to be an interval graph. From theorem 2.8 on page 15 we know that G_i must be a cocomparability graph and not contain a chordless cycle of length 4. If $\mathcal{E}_{+,i}$ contains a chordless 4-cycle and both chords lies in $\mathcal{E}_{-,i}$, no feasible augmentation can be done. If, however, only one chord is in $\mathcal{E}_{-,i}$ and the other, say e, is in $\mathcal{E}_{+,i}^{\mathtt{C}} \cap \mathcal{E}_{-,i}^{\mathtt{C}}$, we should add e to $\mathcal{E}_{+,i}$. Figure 2.7(a) on the next page illustrates this situation.

$G_i = (V, \mathcal{E}_{+,i})$ was furthermore required to be a cocomparability graph and thus $G_i^{\mathtt{C}} = (V, \mathcal{E}_{+,i}^{\mathtt{C}})$ to be a comparability graph. According to theorem 2.9 on the preceding page $G_i^{\mathtt{C}}$ is required to not contain a 2-chordless cycle of odd length. If $(V, \mathcal{E}_{+,i}^{\mathtt{C}})$ contains an odd 2-chordless cycle, the chords

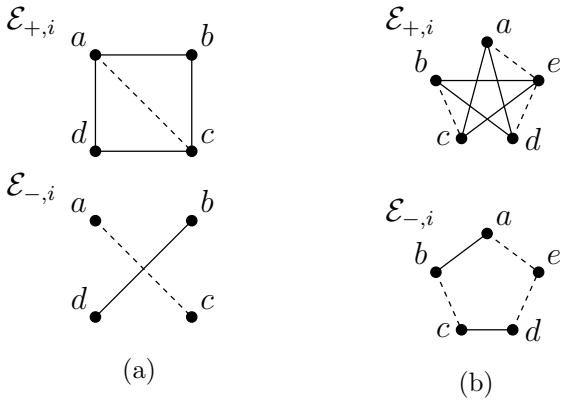


Figure 2.7: 2.7(a) shows a potential chordless C_4 in $\mathcal{E}_{+,i}$. The dashed edge (a, c) is yet to be fixed. To avoid the chordless C_4 we have to fix (a, c) in $\mathcal{E}_{+,i}$. 2.7(b) shows a potential odd chordless cycle in $\mathcal{E}^{\complement}_{+,i}$ with all chords in $\mathcal{E}_{+,i}$ and where the dashed edges are yet to be fixed. The cycle can be avoided in several ways so we branch on every dashed edge.

must be in $\mathcal{E}_{+,i}$ while $\mathcal{E}^{\complement}_{+,i}$ contains the cycle itself. Adding a cycle edge from $\mathcal{E}^{\complement}_{+,i} \setminus \mathcal{E}_{-,i}$ to $\mathcal{E}_{+,i}$ breaks the cycle in $\mathcal{E}^{\complement}_{+,i}$ and an infeasible edge set is thereby avoided. However, the cycle can be broken in several ways (once per unfixed edge) so in this situation we branch on every unfixed edge in the cycle. Figure 2.7(b) illustrates the situation.

To summarize, we have the following properties for a feasible packing:

1. $\cap\mathcal{E}_{+,i} = \emptyset$. If $\cap\mathcal{E}_{+,i} \neq \emptyset$ the search information $\mathcal{E}$ cannot be augmented further in a feasible way and the search in that subtree may stop.

2. $(V, \mathcal{E}_{+,i})$ must not contain a chordless 4-cycle. That is, $\mathcal{E}$ cannot be feasible augmented and the subtree may be excluded if $\mathcal{E}_{+,i}$ contains a chordless 4-cycle with both chords in $(V, \mathcal{E}_{-,i})$.

3. $(V, \mathcal{E}^{\complement}_{+,i})$ must not contain an odd 2-chordless cycle. Therefore the subtree can be skipped if $(V, \mathcal{E}_{-,i})$ contains an odd 2-chordless cycle with all chords in $\mathcal{E}_{+,i}$.

4. $(V, \mathcal{E}^{\complement}_{+,i})$ must not contain an x_i-infeasible clique. If $(V, \mathcal{E}_{-,i})$ contains such a clique the subtree can be excluded.

Using the above deduced rules for augmentation, we have a rough skeleton for the branch-and-bound algorithm. But before diving into further details

with the algorithm we will take a look on box- and edge similarities. By considering the concept of similarity we might be able to improve the algorithm performance if several similar cases can be handled at once.

2.2.2 Box- and edge similarities

Consider an arrangement of boxes in which two of them are of equal size. Then it intuitively makes no difference if these two similar boxes are swaped. This idea of box interchange can be formalized:

Definition 2.10 (Permutation) *A map $\pi : V \to V$ is called a* permutation *if it is bijective and $w(v) = w(\pi(v))$ for all $v \in V$.*

The following proposition shows that a permutation of the vertices keeps feasibility in a packing class:

Proposition 2.11 *Let E be a packing class for (V, w, W) and π a permutation of V. Also, let $E^\pi = (E_1^\pi, \ldots, E_d^\pi)$, where E_i^π is given by*

$$(u, v) \in E_i \Leftrightarrow (\pi(u), \pi(v)) \in E_i^\pi$$

for all $u, v \in V$, $1 \leq d$. Then E^π is a packing class.

Proof We need to show that P1, P2 and P3 are still satisfied. P1 and P3 will hold as the edge structure does not change from E to E^π. Also P2 holds because a permutation per definition keeps vertex sizes. □

Two packing classes whose only difference is a permutation of V in the above sense can be considered similar or *isomorphic*:

Definition 2.12 *Let E and E' be packing classes for (V, w, W). E is said to be* isomorphic *to E' if there exists a permutation π on V such that $E^\pi = E'$.*

In an ideal situation isomorphic packing classes would be considered at most once. However, to the best of our knowledge no algorithm has yet been found that decides in polynomial time if two packing classes are isomorphic. In [Pap94, p. 291] it is even conjectured that no such polynomial algorithm exists. The problematics are increased by the fact that these decisions have to be made repeatedly as the augmentations progress and that packing classes are mostly only known partially. Therefore we cannot fully benefit from the concept of isomorphism. Instead we settle with the special cases where the permutation only affects exactly *two* boxes as these can be

checked relatively easy. Pairs of equally sized boxes with the same search information are called *indistinguishable*. Also edges between indistinguishable boxes are called indistinguishable:

Definition 2.13 *Let $u, v \in V$ be two boxes with $w(u) = w(v)$ in all dimensions and $\mathcal{E}$ be the search information at some node in the search tree. u and v are called* indistinguishable *with respect to $\mathcal{E}$ if*

$$(u, x) \in \mathcal{E}_{\sigma,i} \Leftrightarrow (v, x) \in \mathcal{E}_{\sigma,i}$$

for all $i \in \{1, \ldots, d\}$, $\sigma \in \{+, -\}$ and $x \in V \setminus \{u, v\}$. Also, two edges $e = (u, v)$ and $e' = (u', v')$ are called indistinguishable with respect to $\mathcal{E}$ if u and u' as well as v and v' are indistinguishable with respect to $\mathcal{E}$.

Notice that all boxes are trivially indistinguishable from them selves. Figure 2.8 shows examples of indistinguishable boxes and edges.

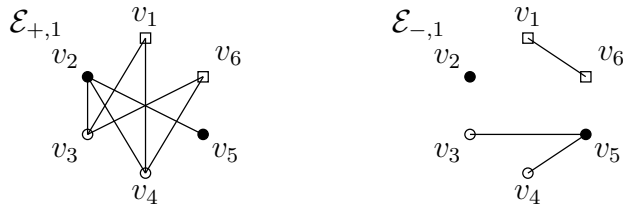


Figure 2.8: Assume $d = 1$, $w(v_1) = w(v_6)$ and $w(v_3) = w(v_4)$ and consider the depicted search information. Then v_1 and v_6 are indistinguishable because they have the same adjacencies for all $\sigma \in \{+, -\}$ and $i \in \{1\}$. Also v_3 and v_4 are indistinguishable. The edges (v_1, v_3), (v_1, v_4), (v_3, v_6) and (v_4, v_6) are indistinguishable. Also the edges (v_2, v_3) and (v_2, v_4) are indistinguishable. In $\mathcal{E}_{-,1}$ the edges (v_3, v_5) and (v_4, v_5) are indistinguishable.

The following lemma ensures the existence of an isomorphic packing class in the same search space in cases where indistinguishable edges exists in a search node.

Lemma 2.14 *Let $\mathcal{E}$ be the search information at a given node and A the set of indistinguishable edges on that node. For any $e \in A$ let A_e denote the subset of edges in A being indistinguishable to e. For any packing class $E \in \mathcal{L}(\mathcal{E})$ having $A_e \cap E_i \neq \emptyset$ there exists an isomorphic packing class $E' \in \mathcal{L}(\mathcal{E})$ with $e \in E'$.*

PROOF For $e' \in A_e \cap E_i$ we have that $e = (u, v)$ and $e' = (u', v')$ are indistinguishable. This means that u, u' and v, v' are pairs of indistinguishable

boxes and there must exist a permutation π of V that swaps u with u' and v with v' such that $E' = E^\pi$. Per definition of indistinguishable boxes we have $(u,x) \in \mathcal{E}_{\sigma,i} \Leftrightarrow (u',x) \in \mathcal{E}_{\sigma,i}$ and $(v,x) \in \mathcal{E}_{\sigma,i} \Leftrightarrow (v',x) \in \mathcal{E}_{\sigma,i}$ for all $i \in \{1,\ldots,d\}$, $\sigma \in \{+,-\}$ and $x \in V \setminus \{u,v\}$. This proves that $E' \in \mathcal{L}(\mathcal{E})$. □

Figure 2.9 illustrates this lemma.

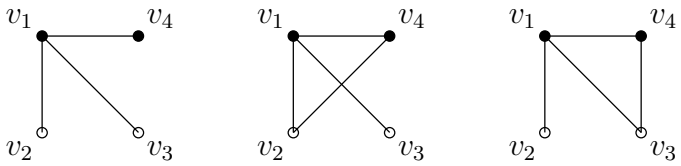


Figure 2.9: Consider this simplified situation where v_2 and v_3 are indistinguishable and 2.9(a) is a graph from the search information $\mathcal{E}$. The edges (v_1,v_2) and (v_1,v_3) are then indistinguishable. Also the yet unfixed edges (v_2,v_4) and (v_3,v_4) are indistinguishable. Lemma 2.14 on the page before tells that both the graph in 2.9(b) and 2.9(c) will exist in $\mathcal{L}(\mathcal{E})$ *and* that these two graphs are isomorphic.

Lemma 2.14 can be used to improve the augmenting of $\mathcal{E}$ at each search node. Two situations emerge:

1. If A is the set of indistinguishable edges and $e \in A$ is the edge that we currently branch at in dimension x_i then for the subtree where $e \notin E_i$ also $e' \notin E_i$ for all $e' \in A_e$: Otherwise $e' \in E_i$ would imply that the search space would indeed contain a packing class with $e \in E$ according to lemma 2.14. Also, there must exist an isomorphic packing class for each $e' \in A$ in the subtree $e \in E_i$.

2. If $A \cap \mathcal{E}_{-,i} \neq \emptyset$ then $(e,-,i)$ is a feasible augmentation for all $e \in A$. That is: We only need to explore the subtree for one $e \in A$ since all other subtrees just represent isomorphic duplicates of the tree.

2.2.3 The algorithm

This section covers the algorithm of Fekete and Schepers presented in [FS97c]. The algorithm consist of three major parts: The main loop, a method for testing for packing classes (`packingclassTest()`) and a method that handles augmentations (`updateSearchInfo()`).

The main loop maintains a list $\mathcal{N}$ holding all nodes to be explored in the search tree. Each $N \in \mathcal{N}$ have the form $N = (\mathcal{E}^N, (e, \sigma, i)^N)$ where $\mathcal{E}^N = (\mathcal{E}^N_+, \mathcal{E}^N_-)$ is the search information for node N and (e, σ, i) is the augmentation to be performed for that node. For the first node the special assignment $(e, \sigma, i)^{N_0} = \text{NULL}$ is made. The main loop does as follows:

1. Initialize the algorithm by setting $\mathcal{N} = \{N_0\}$ and $(e, \sigma, i)^{N_0} = \text{NULL}$.
2. While $\mathcal{N} \neq \emptyset$, select some $N \in \mathcal{N}$ and remove it from $\mathcal{N}$. For N, repeat the following steps until a state different from FIX is obtained in the below loop:
   (a) Use `updateSearchInfo()` to perform the augmentation in N. The method returns either the state OK or EXIT.
   (b) If the state is OK, use `packingclassTest()` to check if N contains a packing class. The method returns an augmentation (e, σ, i) and either EXIT, FIX, BRANCH or SUCCESS. If FIX is returned, `updateSearchInfo()` is called again in order to carry out the augmentation.
3. If the state is SUCCESS, a packing class has been found. Return $\mathcal{E}^N_+$.
4. If the state is BRANCH, we branch on the returned edge: Create two new nodes N' and N'' in $\mathcal{N}$ based on the returned (e, σ, i): $\mathcal{E}^{N'} = \mathcal{E}^{N''} = \mathcal{E}^N$, $(e, \sigma, i)^{N'} = (e, +, i)$ and $(e, \sigma, i)^{N''} = (e, -, i)$.
5. If $\mathcal{N}$ is empty, no packing exist. Else, continue from 2.

The method `updateSearchInfo()` takes as input an augmentation (e, σ, i) and the search information $\mathcal{E}$. It performs the augmentation on $\mathcal{E}$ and returns either EXIT or OK. If some augmentation implies other feasible augmentations (as seen in section 2.2.2) these are carried out as well. A list L contains all augmentations to be carried out in this call of `updateSearchInfo()`. The method works as follows:

1. If $(e, \sigma, i) = \text{NULL}$ we are at N_0 and $\mathcal{E}^{N_0}$ is then initialized in the following way: First, trivially infeasibility for pairwise boxes is avoided. If two boxes are x_i-infeasible they must overlap in a feasible packing. That is, for all $u, v \in V$, if $w_i(u) + w_i(v) > W_i$ then (u, v) is added to $\mathcal{E}^{N_0}_{+,i}$. Next, cliques induced by the individual box types are also fixed in $\mathcal{E}^{N_0}_{+,i}$. This is done by utilizing lemma 18 in [FS97b]. We will not cover the details here. All augmentations done in this initializing step is stored in L.

2. If we are not at N_0 then either $\sigma = +$ or $\sigma = -$. If $\sigma = +$, add e to $\mathcal{E}_{+,i}$ and set $L = L \cup \{(e, +, i)\}$. If instead $\sigma = -$, consider the set A of edges indistinguishable from e. As described in section 2.2.2, lemma 2.14 implies that all $f \in A$ can be fixed in $\mathcal{E}_{-,i}$ as well. So for all $f \in A$, add f to $\mathcal{E}_{-,i}$ and append the augmentations to L.

3. While $L \neq \emptyset$, choose a $(e, \sigma, i) \in L$ and remove it from L. For each e we try to avoid infeasible edge configurations:

   (a) Check if P3 is satisfied for the edge e. This is done as described in section 2.2.1. If P3 cannot be satisfied then return EXIT (drop the branch) else perform the appropriate augmentation and append it to L.

   (b) Check if e is part of a chordless C_4 as described in section 2.2.1. If such a cycle cannot be avoided then return EXIT, else perform the appropriate augmentation and append it to L.

   (c) Check if e is part of an infeasible clique. If such an infeasible clique cannot be avoided then return EXIT, else perform the appropriate augmentation and append it to L.

4. If we have not returned yet (with EXIT) then return OK, indicating that the search may continue.

The algorithmic details for step 3b and 3c can be found in [FS97c].

The method `packingclassTest()` that tests for existence of a packing class takes the search information for the current node as input and returns an augmentation together with either EXIT, FIX, BRANCH or SUCCESS. We need to check feasibility for all $i = 1, \ldots, d$. So for each such i do the following:

1. Let A be an initially empty set containing eventually conflicting edges. If $(V, \mathcal{E}^{\mathtt{C}}_{+,i})$ is not a comparability graph then let A be the set of all edges of the 2-chordless odd cycle. Else, if $(V, \mathcal{E}^{\mathtt{C}}_{+,i})$ contains maximal x_i-infeasible clique then let A be the edge set of this clique. Else, if $(V, \mathcal{E}_{+,i})$ contains an induced C_4 then let A be the set of chords of this C_4. All these infeasible edge configurations have been described in section 2.2.1.

2. A is now a set of edges making the graph infeasible. If $A \neq \emptyset$ then do the following:

(a) If $A \setminus \mathcal{E}_{-,i} = \emptyset$ then all conflicting edges have already been fixed in $\mathcal{E}_{-,i}$ and no augmentation can therefore be done in order to avoid infeasibility. Return EXIT.

(b) Else choose an edge $e \in A \setminus \mathcal{E}_{-,i}$. If $A \setminus \mathcal{E}_{-,i} = \{e\}$ then e must be fixed correctly as a wrong augmentation of this edge alone can make the edge set infeasible. The infeasible edge sets found in step 1 can all be avoided by fixing e in $\mathcal{E}_{+,i}$. Therefore set $(e, \sigma, i)^{\text{return}} = (e, +, i)$ and return FIX. Else, if $|A \setminus \mathcal{E}_{-,i}| > 1$ we branch on e: Return BRANCH. Notice, that only one edge in $A \setminus \mathcal{E}_{-,i}$ is being branched on. The other edges might automatically be irrelevant when the branch augmentation is carried out in `updateSearchInfo()`.

If feasibility has been checked for all $i = 1, \ldots, d$ and the method has not yet returned with either EXIT, FIX or BRANCH, the edge set $\mathcal{E}_+$ is a packing class: Return SUCCESS.

2.2.4 Remarks about performance

Even though the presented algorithm is in the current top league regarding to performance [FS97c], each iteration in the branch-and-bound search may still be very cumbersome: Step 1 in `packingclassTest()` search for a maximal clique. This is also known as the *clique problem* (or CLIQUE) which is NP-hard. [CRLS01] gives a proof for this by reducing CNF-SAT to CLIQUE. Fekete and Schepers get around the NP-hard CLIQUE problem by only considering two cases:

- If a graph is a comparability graph an algorithm from [Gol80] is used to find a maximal weighted clique in linear time.
- If it is not a comparability graph a heuristic is used to find a maximal weighted clique. This clique might be suboptimal though.

Step 1 in `packingclassTest()` also search for chordless cycles or so-called *holes*. [NP04] presents an algorithm for finding holes $O(|V| + |E|^2)$ time. As mentioned, these problems have to be solved repeatedly and are thus contributing significantly to the total computation time.

In the upcoming chapters we will consider packings satisfying the so-called *guillotine property* and introduce a further restricted version of the guillotine property named *sticky cuttings*.

Chapter 3
The guillotine restriction

We will now add a *guillotine cut* condition to feasible packings. A guillotine cut is a side parallel cut through the whole container that does not intersect with any box. This cut splits the container in two smaller cut slices which can each be cut again. We divide the cutting in *stages* and require all cuts of the same cut slice in the same stage to be parallel and the cuts in two consecutive stages to be orthogonal.

The idea of k-stage guillotine cuttings is formalized in the following definitions:

Definition 3.1 (Guillotine packing) *Let p be a d dimensional packing of V. A $d-1$ dimensional axis parallel hyper plane $\mathscr{P}$ is called a* guillotine cut *if it divides V into two disjoint nonempty subsets V_1 and V_2 such that no box $v \in V$ is intersected by $\mathscr{P}$ with respect to p.*

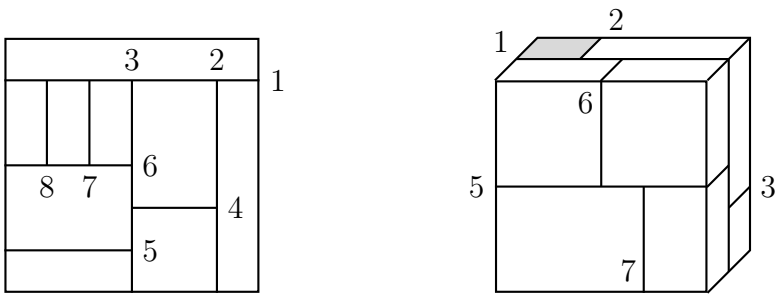
Two subsets $U_1, U_2 \subseteq U \subseteq V$ are called cut slices *if and only if they are a result of a guillotine cut of U.*

The cutting of V is done in recursion stages. In each stage *all cuts splitting a cut slice must be parallel. A cut slice U is in the k'th* stage *if it is made in the k'th recursion step. A packing p of the set V is a k-stage guillotine packing if and only if it can be split into $|V|$ singleton sets in k stages.*

If there are no restrictions on k we just say that p is a guillotine packing.

Notice, that we can uniquely identify a cutting plane $\mathscr{P}$ by the axis parallel unit vector that is orthogonal to $\mathscr{P}$. Figure 3.1 on the following page shows examples of guillotine cuts for $d = 2$ and $d = 3$.

The literature sometimes distinguish between if *trimming* are allowed after the recursive guillotine cutting or not. If trimming is allowed, the recursive cutting must to result in pieces being larger than the box they



(a) 4 stage guillotine cut for $d = 2$

(b) 3 stage guillotine cut for $d = 3$

Figure 3.1: Example of guillotine cutable packings. The numbers indicate one out of many feasible orderings of the cut slices. Cut number 4 in 3.1(b) is hidden in the back. It splits the grey cut slice.

contain. If trimming is not allowed, the recursive cutting must result in pieces with the exact size of the box they contain. In this text we always allow trimming.

The decision problem OPP is NP-hard in the case where no restrictions exist for the cutting. But even though the guillotine restriction reduces the number of feasible packings the following theorem proves that the complexity is not reduced.

Theorem 3.2 *The subproblem of OPP where packings are required to be guillotine cuttable is NP-hard.*

PROOF Let $S = \{w_1, \ldots, w_n\}$ be the set of n weights summarizing to $W = \sum w_i$. The 2-partition problem is the task of finding two disjoint subsets $S', S'' \subset S$ with $S' \cup S'' = S$ where

$$\sum_{w_i \in S'} w_i = \sum_{w_j \in S''} w_j = \frac{W}{2}.$$

This is known to be NP-hard and we will reduce 2-partition to the guillotine restricted OPP-2.

Consider an OPP-2 instance with n boxes of size $(w_1, 1), \ldots, (w_n, 1)$ and a container of size $(\frac{1}{2}W, 2)$ where $W = \sum w_i$. If an OPP-2 solution can be found for this instance it will be a filling of the container by two rows, each being 1 unit high and $\frac{1}{2}W$ units wide. The packing is clearly guillotine cuttable as the two rows can be separated in the first cut stage and each of them sliced up afterwards. Solving the OPP with the guillotine restriction for

this instance also solves the 2-partition problem for the instance mentioned above. Similarly if the 2-partition problem is solved, the solution is equivalent to a guillotine restricted OPP solution similar to the one described above.

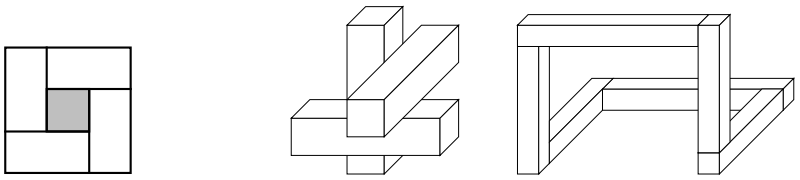
As the reduction is polynomial, OPP-2 is NP-hard. □

3.1 Performance

In the following we the concept of a blocked ring.

Definition 3.3 (Blocked ring) *A set of boxes $B = \{v_1, \ldots, v_n\}$ form a* blocked ring *if for all $u, v \in B$ all planes separating u and v intersect some other $w \in B \setminus \{u, v\}$.*

Figure 3.2 shows examples of blocked rings. Notice that for a blocked ring B we have $|B| \geq 4$ for $d = 2$ and $|B| \geq 3$ for $d \geq 3$. [ST96] introduces the



(a) $d = 2$ (b) $d = 3$

Figure 3.2: Examples of blocked rings for $d = 2$ and $d = 3$.

concept of a *G4-structure* for $d = 2$: A G4-structure is a packing containing at least one blocked ring. The following lemma shows that blocked rings and guillotine packings are strongly connected.

Lemma 3.4 *A set of boxes V can be packed as a guillotine packing if and only if a packing p with no blocked ring exist for V.*

Proof (⇐) First, assume that a packing p with no blocked rings exist for V. As no blocked rings exist there must, per definition, exist a number of parallel cutting planes that split V into a number of cut slices. All these cut slices are subsets of V so if no blocked ring existed for V with the packing p, then also no blocked ring will exist for the cut slices. Applying this argument

recursively, we can cut whole V into $|V|$ singleton cut slices. That is, p is a guillotine packing for V.

($\Rightarrow$) Next, assume that p is a guillotine packing containing some blocked ring B. Then no cutting plane splitting B exist at any cutting stage, contradicting that p is a guillotine packing. Consequently no blocked ring can exist. □

When considering the OBPP-2 problem, it emerge that guillotine packings use at most twice as many containers for the same box set as unrestricted packings:

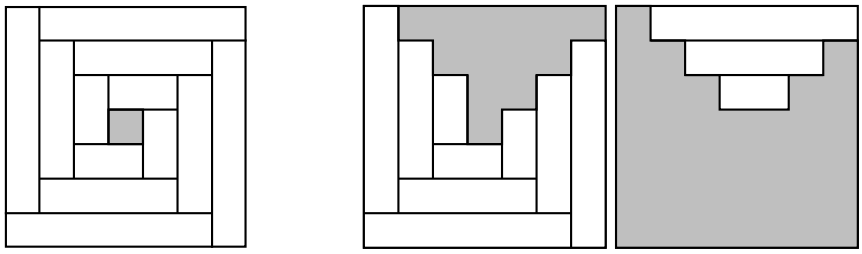
Theorem 3.5 *Let V be a set of boxes that can be packed into OPT containers with the packings $p_1, \ldots, p_{OPT}$ and let OPT_g be the maximal number of containers needed in order to arrange V into a guillotine packing. Then asymptotically $\frac{OPT_g}{OPT} = 2$ for $d = 2$.*

PROOF Given a packing p lemma 3.4 on the preceding page tells that a necessary and sufficient condition for that packing to be a guillotine packing is that no blocked ring exist. Let $C_1, \ldots, C_n$ denote the containers currently holding the packing and $V' \subset V$ denote an initially empty set. For each blocked ring in C_i, pick a box from the blocked ring, remove it from C_i and add it to V'. This makes C_i a guillotine packing.

Next, we show that the content of V' can always be made guillotine packable using no more than n containers. Let $L_0(S)$ denote the *continuous lower bound* defined as the ratio $\frac{\mathcal{V}_S}{\mathcal{V}_C}$ where $\mathcal{V}_S$ is the total volume of the boxes in S and $\mathcal{V}_C$ is the container volume. L_0 will be more formally introduced in chapter 6. In [MV98] Martello and Vigo present an algorithm that produces a guillotine packing using no more than $4L_0(S)$ containers for the case $d = 2$. We utilize this result when moving boxes from C_i to V'. For each of the boxes moved to V', select the box with smallest area from the blocked ring. There are at least 4 boxes in each blocked ring for the case $d = 2$ so the boxes in V' will have a volume of at most $n\frac{\mathcal{V}_C}{4}$ resulting in $L_0(V') \leq \lceil \frac{n}{4} \rceil$. The algorithm can therefore pack V' into at most $4\lceil \frac{n}{4} \rceil$ containers. In total we have $\mathrm{OPT}_g \leq n + 4\lceil \frac{n}{4} \rceil$ and $\lim \frac{\mathrm{OPT}_g}{\mathrm{OPT}} \leq 2$.

To show $\frac{\mathrm{OPT}_g}{\mathrm{OPT}} \geq 2$, consider a G4-structure of $4n$ boxes (see figure 3.3 on the next page). Moving a box from each blocked ring to another container makes each container guillotine packable. □

Open problem 3.6 *What is $\frac{OPT_g}{OPT}$ for $d > 2$?*



(a) Unrestricted (b) Guillotine cutting

Figure 3.3: This G4-structure shows $\mathrm{OPT}_g \geq 2$. It consits of $4n$ boxes with two boxes of size $(1, 2i)$ and two boxes of size $(2i, 1)$ for each $i \in \{1, \ldots, n\}$. The container size is $(2n+1, 2n+1)$. 3.3(a) shows the unrestricted packing and 3.3(b) the split into two containers.

3.2 Characteristics of guillotine cuts

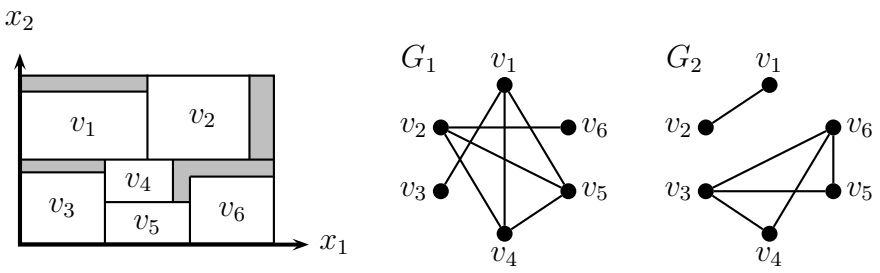
Let us return to the decision problem and have a look on how guillotine packings behave in framework by Fekete and Schepers described in chapter 2.

Consider the packing class $E = (E_1, \ldots, E_d)$ for a guillotine packing and let $\alpha_1, \ldots, \alpha_k$ denote the axes orthogonal to the cutting planes for stage $1, \ldots, k$. This is a minor simplification as we hereby only consider a single path in the cutting-recursion tree, but it clearifies the notation and the results below can without problems be generalized by applying the proofs to every path in the recursion tree.

The cut slices for stage 1 are exactly the connected subgraphs in G_{α_1}. That is, let $V_1, \ldots, V_l$ denote the vertices in the connected subgraphs of G_{α_1}. The induced subgraphs $G_{\alpha_1}[V_1], \ldots, G_{\alpha_1}[V_l]$ are disconnected in G_{α_1} and contain exactly the vertices of the first stage cuts. Figure 3.4 on the following page gives an example of this situation.

Similarly, the second stage cut slices of a first stage cut slice V_j is exactly the connected components in $G_{\alpha_2}[V_j]$. By proceeding with a recursive dissolvement the vertex set for each induced subgraph will end up being singleton. We have then split the packing into individual boxes. The following lemma formalizes this property:

Lemma 3.7 (Graph dissolvement) *For a guillotine packing p and a cut stage $1 \leq l < k$ let U denote the set of vertices in a cut slice for stage l. Also, let $U_1, \ldots, U_n$ denote the vertices of the connected components in $G_{\alpha_{l+1}}[U]$.*



(a) Guillotine packing (b) Corresponding interval graphs

Figure 3.4: The cut slices for the first-stage cut of the packing in 3.4(a) is exactly given by the connected components in G_{α_1} (in this case G_2).

Then $U_1, \ldots, U_n$ is exactly the vertex sets of the cut slices for stage $l+1$.

PROOF As U_i and U_j are unconnected in $G_{\alpha_{l+1}}[U]$ they can be split by a cutting plane orthogonal to α_{l+1}. I.e all components in $G_{\alpha_{l+1}}[U]$ are cut slices for stage $l+1$. Reversely, if U_i and U_j are cut slices of U then U_i and U_j are each connected in $G_{\alpha_{l+1}}[U]$ but U_i is not connected to U_j. □

The order of consecutive cuts in a guillotine packing may be permuted under certain conditions without destroying feasibility:

Lemma 3.8 (Cut directions) *Let α_i and α_j be consecutive cut directions with $i \neq j$ and U be a cut slice. Let $U_{11}, U_{12}, U_{21}, U_{22} \subseteq U$ be disjoint subsets of U with $U_{11} \cup U_{12} \cup U_{21} \cup U_{22} = U$. If $G_{\alpha_i}[U_{11} \cup U_{12}]$ and $G_{\alpha_i}[U_{21} \cup U_{22}]$ are disconnected and also $G_{\alpha_j}[U_{11} \cup U_{21}]$ and $G_{\alpha_j}[U_{12} \cup U_{22}]$ are disconnected then U may be split by a cut order α_i, α_j as well as α_j, α_i.*

PROOF Figure 3.5 on the next page depicts the situation. Consider the generation of the cut slice U_{11}. Cutting U orthogonal to α_i results in the cut slices $U_{11} \cup U_{21}$ and $U_{12} \cup U_{22}$. These sets are disconnected in G_{α_j} so cutting $U_{11} \cup U_{21}$ orthogonal to α_j results in the sets U_{11} and U_{21}.

Next, if we first cut U orthogonal to α_j the subsets $U_{11} \cup U_{12}$ and $U_{21} \cup U_{22}$ are obtained. Both sets are disconnected in G_{α_i} and thus cutting $U_{11} \cup U_{12}$ orthogonal to α_i results in the sets U_{11} and U_{12}.

As seen, both cut orders produce the set U_{11}. A similar argument can be made for U_{12}, U_{21} and U_{22}. □

Lemma 3.7 and lemma 3.8 can be applied recursively in order to dissolve

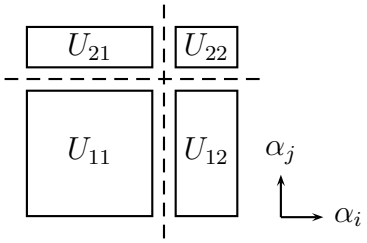


Figure 3.5: Lemma 3.8 deals with permutation of cut stages. For this packing cutting stage i and j may be freely permuted.

a complete packing into individual boxes. Figure 3.6 on the following page illustrates such a recursive dissolvement.

3.2.1 Ensuring the guillotine property

Lemma 3.7 implies a necessary and sufficient condition for a packing class $(\mathcal{E}_{+,1}, \ldots, \mathcal{E}_{+,d})$ *not* being a representation of a guillotine packing: Consider a recursive dissolvement of $\mathcal{E}_+$. If $G_i[U]$ is an induced subgraph produced by a recursive dissolvement, $|U| > 1$ and no graph $G_j[U]$, $j \neq i$ is disconnected, then the dissolvement cannot proceed and the packing can therefore not be a guillotine packing. The following algorithm *ensures* the guillotine property:

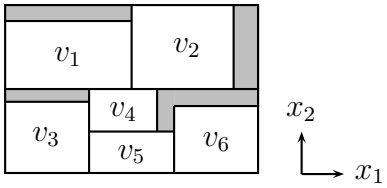
1. Perform a recursive dissolvement as suggested by lemma 3.7. For each iteration let $G_j[U]$ denote the induced subgraph currently being considered.

2. Let $J = \{j \mid (U, \mathcal{E}_{+,j}) \text{ is disconnected}\}$. If $J = \{j\}$ and $G_j[U]$ has exactly two components C_1 and C_2 we have to make sure that these components stay separated: For all edges $e \in \mathcal{E}_+^{\mathtt{C}} \setminus \mathcal{E}_-$ connecting C_1 and C_2, make the augmentation $(e, -, j)$.

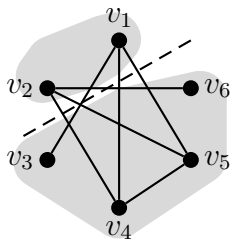
3. If $|J| = 0$ the packing class is not a guillotine packing. Drop the branch.

3.2.2 P2: Handling x_i-infeasibility

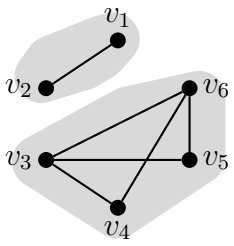
To measure the dimensions of a packing represented by a packing class we can make a transitive orientation and assign coordinates to each box in $O(|V|^2)$ time. Finding a maximal weighted clique in $\mathcal{E}_{+,i}^{\mathtt{C}}$ is, worst case, an NP-hard subproblem as stated in section 2.2.4 but Fekete and Schepers settle with



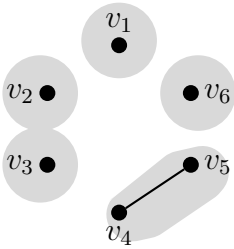
(a) Cut order: x_2, x_1, x_2



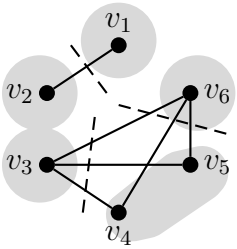
(b) G_1 Initial graph. First cut stage marked.



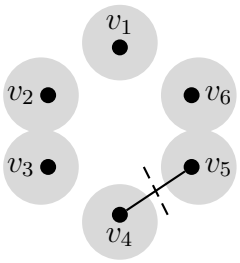
(c) G_2 Initial graph



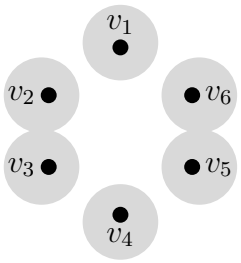
(d) G_1 after first cut



(e) G_2 after first cut. Second stage cuts marked.



(f) G_1 after second cut. Third stage cut marked.



(g) G_2 after second cut.

Figure 3.6: 3.6(a) shows the packing to be dissolved. In 3.6(c) the connected components in G_2 are highlighted and they result in two induced subgraphs of G_1 in 3.6(b) corresponding to the first stage cut. The induced subgraphs of G_1, 3.6(d), suggest where to slice G_2 in the second stage, 3.6(e). Finally the components of G_2 in 3.6(g) indicate the last cut stage, 3.6(f) and the graph dissolvement is then completed.

suboptimal maximal cliques in the case where the graph is not a comparability graph. With the dissolvement tools described above we can construct a polynomial time clique finding algorithm in which a maximal clique is found even if the graph is not a comparability graph. The algorithm utilizes the following dissolvement based width measuring.

1. The x_i-width of an induced subgraph $G_i[U]$ is the sum of the x_i-widths of its connected components $U_1, \ldots, U_n$.

2. Also, the x_i-width of each U_l is the maximum x_i-width of each of the connected components in $G_j[U_l]$ for some appropriate $j \neq i$: If U is a guillotine packing and U_l is not singleton there must, for each $l = 1, \ldots, n$, exist a $j \in \{1, \ldots, d\}$ for which $G_j[U_l]$ is disconnected (lemma 3.7). Lemma 3.8 tells us that we can perform the measuring on an arbitrary graph in which U_l is disconnected.

This measurement is then applied recursively.

It is now interesting to estimate the impact of each measurement. Connected components can be identified by first visiting all edges and enumerating their vertices, and next traversing all vertices to create the subsets. This takes $O(|V|^2 + |V|)$ time. In order to find a graph G_i in which the induced subgraph of a vertex set U is disconnected we have to check at most d graphs and for a k stage packing, these checks must be done at most k times. Thus a worst case complexity of $O(dk|V|^2 + dk|V|) = O(dk|V|^2)$ for measuring of the x_i-width. Notice, that with an appropriate implementation, all d x_i-widths can be measured in the same run.

As shown in lemma 3.8 a guillotine cutting can be obtained by dissolving each cut slice orthogonal to *any* cut direction α_i for which its induced subgraph is disconnected and therefore the dissolvement based measuring can also follow any feasible cut order. But whereas the measuring based on transitive orientations was unique for each packing class, each cut order may lead to different size measurings in the dissolvement based algorithm. Figure 3.7 on the next page shows an example of how different dissolvements leads to different packing topologies. So in order to maintain consistency the dissolvements should always be tried completed in the same order. E.g. starting from G_1 and ending at G_d.

Ensuring the P2 property, i.e. preventing infeasible cliques, is either done by settling with the suboptimal cliques as in the original framework by Fekete and Schepers or by solving the NP-hard CLIQUE subproblem. But with use of the dissolvement based measuring a maximal clique can always be found in $O(dk|V|^2)$ time. When $\mathcal{E}_+$ is augmented with an edge e we avoid infeasible

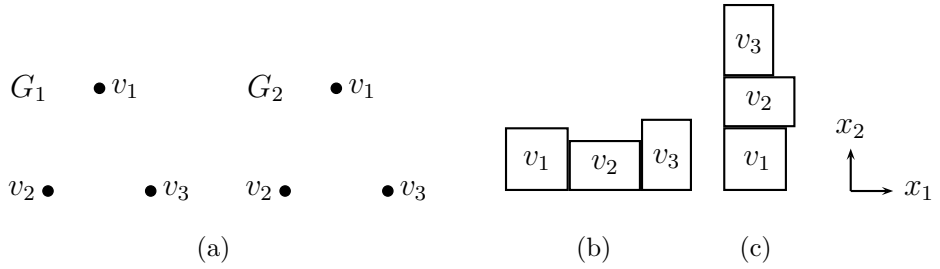


Figure 3.7: Consider the boxes v_1, v_2 and v_3 with dimensions $(15, 15)$, $(17, 12)$ and $(12, 17)$. The graphs in 3.7(a) have no edges and may be dissolved in the order x_1, x_2 as well as x_2, x_1. The former order results in the packing seen in 3.7(b) with dimension $(44, 17)$ whereas the latter results in the packing seen in 3.7(c) with dimension $(17, 44)$. Using the measuring based on transitive orientations as done in the original framework by Fekete and Schepers the graphs would represent a packing where each box was placed at origin or at the corner of another box resulting in a bounding dimension of $(44, 44)$.

stable sets in $\mathcal{E}^{\complement}_{+,i}$ by the following algorithm. For each i let G_i denote the graph $(V, \mathcal{E}_{+,i})$.

1. For each $i = 1, \ldots, d$ calculate the x_i-width as described above.

2. For each recursion step in the x_i measuring do the following: Let $j \in \{1, \ldots, d\}$ be the current cut direction and let $G_j[U]$ denote the induced subgraph currently being measured. If $j \neq i$, mark the component in $G_j[U]$ with maximum x_i-width. Long enough down the recursion this marking will be applied to individual vertices.

3. Mark all edges in $\mathcal{E}^{\complement}_{+,i}$ that connect marked *vertices*.

4. The marked edges in $\mathcal{E}^{\complement}_{+,i}$ form a clique among the x_i-widest subset of boxes. If the packing is measured x_i-infeasible, branch on every marked edge in $\mathcal{E}^{\complement}_{+,i} \setminus \mathcal{E}_{-,i}$.

Example 3.9 Consider figure 3.6(a) on page 34. If the packing had been x_1-infeasible the vertex set $\{v_3, v_5, v_6\}$ would be marked by the above algorithm and induce an x_1-infeasible clique in $\mathcal{E}^{\complement}_{+,1}$. Similarly, if the packing had been x_2-infeasible the vertex set $\{v_2, v_4, v_5\}$ would be marked and induce an x_2-infeasible clique in $\mathcal{E}^{\complement}_{+,2}$. $\diamond$

The algorithm is simply a trivally extended dissolvement and has therefore $O(dk|V|^2)$ time complexity.

The two ways of finding and measuring maximal cliques both have pros and cons. Let M_C denote the original measuring approach based on conventional clique finding and transitive orientations and let M_D denote the approach based on recursive dissolvement. Each measuring method implies a packing topology when used to construct a packing from a set of graphs. For each $i \in \{1, \ldots, d\}$ let $G_i = (V, \mathcal{E}_{+,i})$. We can do the following observations:

O1 The same M_D topology can be represented by numerous graphs. Figure 3.8 gives an example on this.

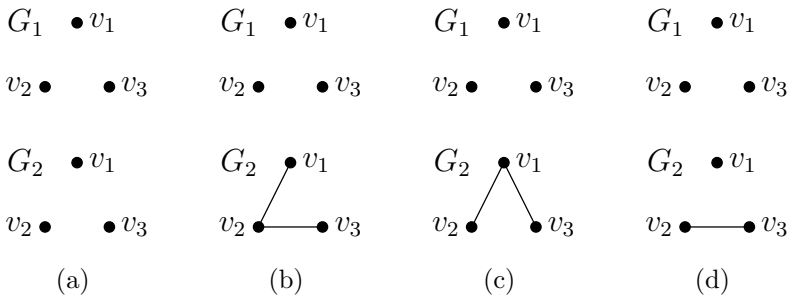
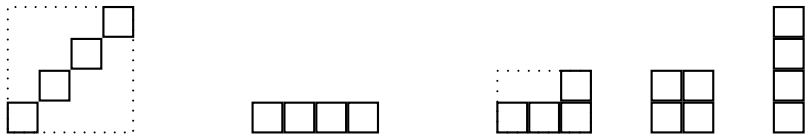


Figure 3.8: All the shown graphs will represent the same topology using M_D and the cut order x_1, x_2. For this vertex set with cardinality 3 and no edges in G_1 a total of $2^3 = 8$ different graphs would represent the same M_D topology.

O2 In contrast there is a one-to-one correspodance between the set of graphs $(G_1, \ldots, G_d)$ and M_C topologies.

O3 Each topology found by using M_C may be "dominated" by several M_D topologies. I.e. the M_D topologies will never have one box at the corner of another box so that no other box aligns to one of its faces. Hence the M_D topologies may be thought of as *flattened* M_C topologies that do not waste container space. Considering a dominated M_C topology may be waste of time. Figure 3.9 on the following page illustrates this thought.

O4 The M_D only results in flattened topologies.



(a) (b) (c)

Figure 3.9: 3.9(a) shows a topology obtained from M_C on a set of stable graphs. The unused space (inside the dotted rectangle) is smaller when using M_D as no two boxes are placed diagonally. 3.9(b) shows an M_D topology with the dissolvement order x_1, x_2. 3.9(c) shows three (out of many) other "flattened" topologies that waste less container space and thus may be thought of as dominating the M_C topology.

So the M_C based topologies have no redundant graph representations but each measuring is either not exact (in case the complement graph is not a comparability graph) or contains an NP-hard subproblem, and also multiple x_i-infeasible packings, not existing in the M_D search space, are created. These pros and cons leave a number of open problems:

Open problem 3.10 *Given the considerations O1-O4, is the search space for M_C smaller than the search space for M_D?*

Open problem 3.11 *Is the* total *time complexity for the algorithm searching M_C topologies smaller than the total time complexity for the algorithm searching M_D topologies?*

The experimental work presented in chapter 5 render probable that the answer to both problems is *yes* but until a formal proof has been given they are still considered open.

3.2.3 P1 and P3

P1 requires all graphs to be interval graphs. According to theorem 2.8 on page 15 we need to ensure that $\mathcal{E}_{+,i}$ does not contain a chordless C_4 and $\mathcal{E}^{\complement}_{+,i}$ is a comparability graph. However, in order for a set of graphs $G_1, \ldots, G_d$ to be recursively dissolveable it is not necessary that these properties are satisfied: As long as connected components can be recursively dissolved into singleton sets by recursive splits of disconnected components, enough information is available to construct a guillotine packing. Thus, we may completely skip

the check for P1 and hereby 1) get rid of a subroutine with a complexity of $O(|V| + |E|^2)$ in each iteration and 2) obtain a larger number of feasible graphs. The latter might also expand the search space with an even larger amount of *infeasible* graphs though and therefore increase the total complexity.

Open problem 3.12 *Does the total complexity decrease by neglecting P1?*

It is always possible to construct "real" P1 satisfying interval graphs from the packings obtained by the recursive dissolvement. These interval graphs could be thought of as the packing class equivalents to the dissolvable graphs. Figure 3.10 illustrates a P1 violation and the corresponding feasible guillotine packing.

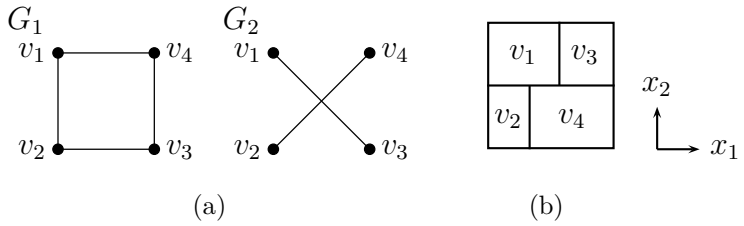


Figure 3.10: G_1 violates P1 and if we construct a packing class from 3.10(b) by creating interval graphs we will not obtain the graphs in 3.10(a). However, the graphs in 3.10(a) still represents the guillotine packing in 3.10(a) by using the recursive dissolvement suggested by lemma 3.7 on page 31.

Lemma 3.7 and lemma 3.8 only rely on the fact that connected components represent cut slices and will therefore still hold when the requirement of P1 is relaxed.

It is possible to completely skip the check for P3 as the dissolvement automatically results in a packing in which P3 is satisfied. However, in order to decrease the search space we ensure P3 as in the original algorithm by, for each augmented edge $e = (u, v)$, considering the intersection $\cap_{i=1}^{d} \mathcal{E}_{+,i}$.

3.3 Tree representation

The recursive cutting used in guillotine packings suggest a tree representation. This section presents a tree representation which only holds topologically strictly different guillotine packings and a brute-force algorithm to traverse all $1, \ldots, k$-stage guillotine packings for a set V.

3.3.1 A short survey

The guillotine constraint has been approached with tree representations before. Below we give a short survey of three different approaches. The articles described all concern differently constrained guillotine cutting problems. As we are only considering unconstrained guillotine cutting problems only the parts relevant for making bounds and representations are described.

Recursive cutting

In [CW77] Christofides and Whitlock present a branch-and-bound algorithm for OKP-2. Starting with the original container the strategy is to recursively perform guillotine cuts. In each iteration a rectangle is chosen for further cutting until a sufficient amount of smaller rectangles has been made which satisfies certain stop criteria. The cutting process can be represented in a binary tree structure where each non-leaf node holds a cut operation and each leaf is a rectangle in the final cut. When a rectangle is chosen for further cutting each possible cut of that rectangle is done in a separate branch. As we shall see below, this cutting will lead to duplicated cut structures in multiple branch nodes but by restricting the set of feasible cuts on each rectangle such redundancy can be avoided:

Consider a rectangle of size (w, h) and a vertical cut at p where $1 \leq p < w$. This cut produces two new rectangles of size (p, h) and $(w-p, h)$. The same two rectangles could have been made by cutting at $w-p$ so the branch node containing the latter cut will be a replica of the branch node holding the former. By restricting the allowed cut positions to $1 \leq p \leq \lfloor w/2 \rfloor$ such symmetry is avoided.

A rectangle can be divided into three slices by using two nodes from the cutting tree. Consider again a rectangle of size (w, h) and two vertical cuts at p and q where $1 \leq p < q < w$. This will produce the rectangles (p, h), $(q-p, h)$ and $(w-q, h)$ which can be obtained by first cutting (w, h) at q and secondly cutting the produced (q, h) rectangle at p. The same three rectangles could be obtained by first cutting (w, h) at p and secondly the produced $(w-p, b)$ rectangle at $q-p$. This kind of symmetry can be avoided by requiring that if a rectangle is cut at p_1 in one tree node then the successors should be cut at at least $p_2 > p_1$.

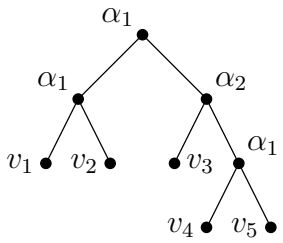
When all feasible cuts have been made on a rectangle it is marked with a special 0-cut, indicating that it may be considered fixed. At each iteration a bound is calculated for the greatest possible value available for the rectangles that fit in the 0-cut. Solving the value maximization subproblem is irrelevant

for this context and will not be discussed further.

The concept of restricting the search space in order to deal with symmetry is also used below.

Normalized polish representation

In [WL86] Wong and Liu introduce a representation of guillotine cut structures by *normalized reverse polish expressions* which can be thought of as a postorder traversal of a binary parse tree where each branch node holds an operator and each leaf holds an operand. A polish expression is normalized if no two consecutive operators are the same. As above, the series of cuts performed in order to construct a guillotine cut structure can be described as a rooted binary cutting tree where each branch node represents a vertical or horizontal cut and each leaf represents a final rectangle. In such a tree representation multiple trees will represent the same cut structure. The concept of *skewed* trees deals with this problem: A *skewed* tree is a binary tree in which no branch node holds an operand that equals the operand of its right child. It can be shown that each guillotine cut structure can be represented by a unique skewed tree. Also it can be shown that a bijective map exist between the set of skewed trees and the set of normalized polish expressions. By only considering skewed trees, symmetry is avoided and by considering polish expressions, cut structures can be modified by manipulating the sequence of operands and operators. Figure 3.11 shows an example of a tree structure and the corresponding representation in reverse polish notation.



(a)

$v_1 v_2 \boldsymbol{\alpha_1} v_3 v_4 v_5 \boldsymbol{\alpha_1 \alpha_2 \alpha_1}$

(b)

Figure 3.11: 3.11(a) shows a skewed tree representing a cut structure for the cut directions α_1 and α_2 and the rectangles $v_1, \ldots, v_5$. A postorder traversal of the tree visits the nodes in the order seen in 3.11(b) and this sequence can be thought of as a representation of the tree based on normalized reverse polish notation where $\boldsymbol{\alpha_1}$ and $\boldsymbol{\alpha_2}$ are cut operators.

An assembling algorithm

The cutting representation in [CW77] and [WL86] described above can be thought of as a top-down approach starting with the container at the top-node and smaller cut slices in successor nodes. In [Wan83] Wang presents a bottom-up approach based on the idea of iteratively assembling the smaller rectangles into larger ones by horizontal or vertical glueing of their bounding rectangles.

The article does not deal with symmetry as in the two examples above but it uses a bound in order to drop search nodes earlier: For each iteration the ratio $\mathcal{V}_{\text{in}}/\mathcal{V}_{out}$ is considered where $\mathcal{V}_{\text{in}}$ is the summarized area of the boxes inside the bounding rectangle and $\mathcal{V}_{out}$ is the area of the bounding box. For some given β each constructed rectangle with $\mathcal{V}_{\text{in}}/\mathcal{V}_{out} < \beta$ is rejected. The difference $\mathcal{V}_{out} - \mathcal{V}_{\text{in}}$ is called *trim waste*. If $\beta = 1$ is chosen only guillotine packings with zero trim waste is accepted while all possible guillotine packings are accepted if $\beta = 0$.

The search space is traversed by stepwise constructing two sets $F^{(k)}$ and $L^{(k)}$. For $k = 0$ we set $F^{(0)} = L^{(0)} = V$. The set $F^{(k)}$ holds all packings that can be build by horizontally or vertically glueing two bounding rectangles of packings from $L^{(k-1)}$ such that 1) the requirement of trim waste is satisfied and 2) container boundaries are not exceeded. Afterwards $L^{(k)} = L^{(k-1)} \cup F^{(k)}$ and equivalent packings are removed from $L^{(k)}$. When $F^{(k)}$ is empty, the packing from $L^{(k)}$ with smallest trim waste is selected.

3.3.2 Packing trees

Below we present a tree representation that has the benefits of symmetry avoidance as seen in [CW77] and [WL86] but also can be used to reject whole subsets of guillotine packings based on volume estimations. The latter is a result of lemma 3.16 on page 45.

The recursive dissolving of guillotine packings described in section 3.2 suggests a tree representation where each leaf node is a vertex $v \in V$ and each cut slice is represented by a branch node, starting with the root node holding the first cutting stage. The only necessary and sufficient condition for a tree to represent a feasible guillotine packing is that the resulting packing is x_i-feasible for all $1 \leq i \leq d$.

We associate each branch node with a cut direction α_i and say that the tree is *normalized* if no branch node has same cut direction as any of its children.

Let U be a cut slice at stage l. We notice that cut slices of U at stage $l+1$ may be permuted arbitrary without destroying the feasibility. This leads to the concept of order equivalent trees:

Definition 3.13 *Two trees T and T' are* order equivalent *if there exist a subset S of the nodes of T so that for each node in S, the children for this node can be reordered and the reordering of the children of S makes $T = T'$.*

Below we introduce the concept of *packing trees* for use in enumeration of all strictly different guillotine packings, taking normalized and order equivalent trees into account. As mentioned, we only consider *unordered* trees. That is, trees for which there is no ordering of the children for branch nodes. Such trees T can be represented by sets of sets together with a map α from the set of all trees to the set $\{1, \ldots, d\}$ that specifies the cut direction for each branch node:

Definition 3.14 (Packing tree) *A set T with a map α is a* packing tree *if and only if the following properties holds:*

T1 $|T| \geq 2$

T2 *For all elements $t \in T$, either $t \in V$ or t is a packing tree itself*

T3 $\mathcal{S}(v, T) \leq 1$ *for all $v \in V$ where*

$$\begin{aligned} \mathcal{S} : V \times \mathcal{T} &\rightarrow \mathbb{N}_0 \\ (v, T) &\mapsto \begin{cases} \sum_{t \in T} \mathcal{S}(v, t) & , t \text{ is a packing tree} \\ 1 & , t = v \\ 0 & , t \neq v \end{cases} \end{aligned}$$

and $\mathcal{T}$ is the set of all packing trees.

T4 *T is normalized with respect to the map α*

The T3 property ensures that each $v \in V$ is contained in at most one subtree. Notice, that the singleton set $\{v\}$ is not a packing tree even though it is a fully legal guillotine packing. The property T1, however, comes in handy as it eliminates a lot of equivalent representations of the same packing. As a packing tree has mathematical sets (without ordering) as branch nodes, order equivalent trees are automatically handled by the same packing tree. Figure 3.12 on the next page shows a set of trees all representing the same packing. The following example illustrates the packing tree definition using sets.

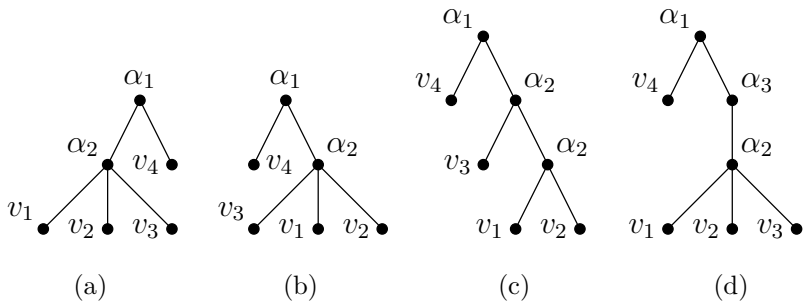


Figure 3.12: The trees in 3.12(a) and 3.12(b) are order equivalent. 3.12(c) is not normalized and is therefore not a packing tree. Joining the two α_2 nodes would result in an equivalent tree which is indeed a feasible packing tree. Also 3.12(d) is not a cutting tree as T1 is not satisfied. However it represents the same packing as the other trees.

Example 3.15 Let $\{\dots\}^i$ denote that the elements are cut by planes orthogonal to the x_i-axis. The sets $\{v_1, \{v_2, v_3\}^2\}^1$ and $\{\{v_1, v_2\}^2, \{v_3, \{v_4, v_5\}^1\}^2\}^1$ are both packing trees, but $\{v_1, \{v_1, v_2\}^2\}^1$ and $\{\{v_1, v_2\}^2, \{v_3, \{v_4\}^1\}^2\}^1$ are not. The former because v_1 occurs in two subsets and the latter because T1 is violated with the singleton subset $\{v_4\}^1$. $\diamond$

Checking feasibility

So how do we perform a feasibility check on a packing tree? x_i-feasibility checks are done equivalent to the recursive measuring described in section 3.2: The x_i-width of a packing tree T is given by the map $w_i : \mathcal{T} \to \mathbb{R}$ where

$$w_i(T) = \begin{cases} \sum_{t \in T} w_i(t) & , \alpha(T) = i \\ \max_{t \in T} w_i(t) & , \alpha(T) \neq i. \end{cases}$$

There are $|V|$ leafs and at most $|V| - 1$ branch nodes in T and thus the complexity for this measuring is $O(|V|)$. In order to make a complete x_i-feasibility check for a given tree topology, we need to examine all combinations of cut-orders, starting with d options for the root node and $d-1$ options for the rest of the branch nodes. This results in a total complexity of $O(|V|d(d-1)^{|V|-1})$ which is polynomial (in fact linear) for $d = 2$.

If we change the image set of w_i from $\mathbb{R}$ to $\mathbb{N}$ a sufficient condition arise for which no sequence $\alpha_1, \dots, \alpha_k$ of cut directions make T feasible:

Lemma 3.16 *Let T be a packing tree and $\mathcal{V} \in \mathbb{N}$ the sum of all volumes of $t \in T$. Let $\mathcal{V}'$ denote the smallest integer having at least d factors and $\mathcal{V}' \geq \mathcal{V}$. Then the volume of T is at least $\mathcal{V}'$.*

PROOF The volume of T is the volume of a d dimensional box. Thus it is a product of at least d integers. The volume of T is trivially at least $\mathcal{V}$ and as $\mathcal{V}'$ is the smallest number above $\mathcal{V}$ with d factors also the volume must be at least $\mathcal{V}'$. □

The number of factors in $\mathcal{V}$ can be found by prime factorization of $\mathcal{V}$.

Example 3.17 Suppose a packing tree T for $d = 3$ consist of 3 cut slices with volumes 3, 3 and 4 respectively. The sum of these volumes is $10 = 2 \cdot 5$ and the nearest larger integer with 3 factors is $12 = 2 \cdot 2 \cdot 3$. Thus the volume of T is at least 12. ◇

The above infeasibility condition can be verified in $O(|V|)$ time.

3.3.3 A brute-force algorithm

We aim to find an algorithm that traverse all possible packing trees. It is not expected to perform well, as it is a brute-force algorithm, but is intended to serve as a proof for the existence of such a traversal algorithm. Define $\delta : \mathcal{T} \to \mathbb{N}_0$ as the *depth* of T:

$$\delta(T) = \begin{cases} 1 + \max\limits_{t \in T} \delta(t) & , T \text{ is a packing tree} \\ 0 & , T \in V \end{cases}$$

For a packing tree T let $\lambda_i(T)$ denote the number of subtrees t of T (including T itself) with $\delta(t) = i$. Then the following properties hold:

- In a k-stage packing, $\lambda_0(T) = |V|$ and $\lambda_k(T) = 1$
- $\lambda_i(T) \leq \lambda_{i-1}(T)$ as each set T with $\delta(T) = i$ must contain at least one set t with $\delta(t) = i - 1$. Also $\lambda_i(T) \leq \lfloor \frac{1}{2} \sum_{j=0}^{i-1} \lambda_j(T) \rfloor$ because of property T1.
- As lower bound for λ_i we only know that $\lambda_i \geq 1$. Figure 3.13 on the following page shows an instance having $\lambda_i = 1$ for all $i > 0$.

Every value assignment of the tuple $(\lambda_0, \ldots, \lambda_k)$ represents a whole class of trees. This is illustrated in figure 3.14 on the next page. All feasible packing trees are traversed as follows:

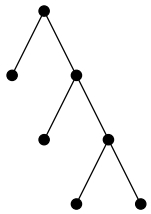


Figure 3.13: Example of tree having $\lambda_i = 1$ for all $i > 0$

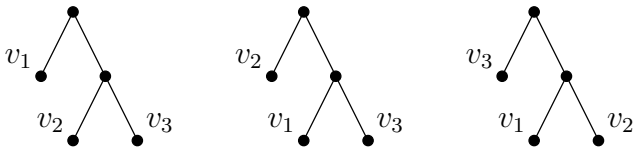


Figure 3.14: For the tuple $(\lambda_0, \lambda_1, \lambda_2) = (3, 1, 1)$ exactly three different tree topologies exist. Notice that permuting the children for a node just results in an order equivalent tree.

1. Construct all feasible assignments of $(\lambda_0, \ldots, \lambda_k)$ systematically following the above conditions.

2. For each $(\lambda_0, \ldots, \lambda_k)$-assignment, construct all trees conforming to these λ-values (see below).

3. For each tree, first do the quick verification of the infeasibility condition described in lemma 3.16 on the preceding page. Next, perform a feasibility check for all $1 \leq x_i \leq d$.

The following paragraph gives an outline of how all trees conforming to a set of λ-values are0 constructed. For a fixed set of depth counts $(\lambda_0, \ldots, \lambda_k)$ the following method traverse all trees conforming to these values. The tree construction is done in $k + 1$ steps: In step 0, $\lambda_0 = |V|$ singleton sets are created, each containing a box $v \in V$. In step 1, λ_1 sets are constructed from at least λ_1 of the sets from step 0. In step 2, λ_2 sets are constructed from at least λ_2 of the sets from step 1 and eventually some of the yet unassigned sets from step 0. Generally, in step j, λ_j sets are constructed from at least λ_j of the sets from step $j - 1$ and eventually some of the yet unassigned sets from earlier steps. In the last step k a set composed by all unassigned sets is constructed. This final superset is the root node. For each level, $0 \leq j \leq k$, we define the sets $T_{j_1}, \ldots, T_{j_{\lambda_j}}$ to hold the λ_j trees of depth j. In level 0, each T_{0_i} is assigned exactly one $v \in V$.

Let $\mathcal{T}$ be the set of trees/root nodes constructed so far and define some arbitrary ordering $R_{\geq}$ of $\mathcal{T}$. That is, in level 0, $\mathcal{T}$ will contain all $T_{0_i}, 1 \leq i \leq \lambda_0 = |V|$. In level $j > 0$, when a subset of root nodes $\mathcal{T}' \subseteq \mathcal{T}$ of depth $\delta(T') < j$ where $T' \in \mathcal{T}'$ is chosen as elements in a new tree/set T we set $\mathcal{T} = \{T'\} \cup \mathcal{T} \setminus \mathcal{T}'$. The set $\mathcal{T}$ will in level k contain exactly one superset – the root node.

Example 3.18 Consider the tuple $(\lambda_0, \lambda_1, \lambda_2) = (5, 2, 1)$. The five trees in level 0 are $T_{0_1} = \{v_1\}$, $T_{0_2} = \{v_2\}$, ..., $T_{0_5} = \{v_5\}$ and $\mathcal{T} = \{T_{0_1}, \ldots, T_{0_5}\}$. $\lambda_1 = 2$ so two trees of depth 1 must be constructed in level 1. One possible construction is $T_{1_1} = \{T_{0_1}, T_{0_3}\}$ and $T_{1_2} = \{T_{0_2}, T_{0_4}\}$ resulting in $\mathcal{T} = \{T_{0_5}, T_{1_1}, T_{1_2}\}$. In the last level we get $T_{2_1} = \{T_{0_5}, T_{1_1}, T_{1_2}\}$ and thus $\mathcal{T} = \{T_{2_1}\}$. ◇

The described stepwise selection of sets from $\mathcal{T}$ is done by enumerating the elements in the following way: In level j each node $T \in \mathcal{T}$ is assigned a value from the set $\{\text{NULL}, 1, \ldots, \lambda_j\}$ indicating which set T_{j_i} that contains T. The special value NULL indicates that T is not assigned any set at this level.

In this assignment, certain rules have to be followed in order to make sure that valid trees are constructed:

1. We have to ensure that exactly λ_j trees of depth j is constructed in level j. This is done by splitting each step in two substeps. In the fist substep we select λ_j elements $T \in \mathcal{T}$ having $\delta(T) = j - 1$. The i'th selected T with respect to $R_{\geq}$ is assigned the set T_{j_i}, i.e. T is assigned the value $i \in \{1, \ldots, \lambda_j\}$. By respecting the ordering $R_{\geq}$ in the assignment symmetry is avoided. There is only one such assignment for $j = 0$ and $j = k$.

   Let S denote the elements selected in this step and observe that all such first-substep assignments can be constructed easily as the task is equivalent to the task of assigning the numbers $1, \ldots, \lambda_j$ to λ_{j-1} horizontally aligned items so that all λ_j numbers are used exactly once and so that the number for item x is smaller than the number for item y if x is to the left of y. In this substep the depth-configuration matching the λ-values is ensured and the tree constructed so far is referred to as the *basis tree*. Example 3.19 on the next page illustrates this substep.

2. In the second substep we extend the basis tree to all possible trees that can be made from the yet unassigned root nodes in $\mathcal{T}$. This is done

in such a way that order equivalence is ensured: Each $T \in \mathcal{T} \setminus S$ is systematically assigned values from $\{\text{NULL}, 1, \ldots, \lambda_j\}$. If $\delta(T) = j-1$ we restrict the possible values of T further: Let $\mathcal{T}_{j-1}$ denote the set $\{T \in \mathcal{T} \setminus S \mid \delta(T) = j-1\}$ of yet unassigned nodes of depth $j-1$. Each $T \in \mathcal{T}_{j-1}$, where T is before (with respect to $R_{\geq}$) the element in T_{j_1} chosen in the first substep, is restricted to the value NULL. Elements in $\mathcal{T}_{j-1}$ before the element in T_{j_2} is restricted to values from the set $\{\text{NULL}, 1\}$. Generally elements in $\mathcal{T}_{j-1}$ before the element in T_{j_i} is restricted to values from the set $\{\text{NULL}, 1, \ldots, i-1 < \lambda_j\}$.

For $T \in \mathcal{T} \setminus \mathcal{T}_{j-1}$ there are no value restrictions.

As mentioned, this step ensures that order equivalent trees are only constructed once.

Example 3.20 illustrates both this substep and the following two steps.

3. The above assignments may result in some set T_{j_i} with $|T_{j_i}| = 1$. If such set exist the iteration is canceled and the next assignment is tried out.

4. When the described enumeration has been made at level j, the sets T_{j_i} induced by this enumeration is created and stored in $\mathcal{T}$. In level $j+1$ the enumeration is done again on this updated $\mathcal{T}$ and so on.

Example 3.19 Assume that we are currently at the *first* substep for construction level 4 and $\mathcal{T} = \{T_{0_7}, T_{1_2}, T_{3_1}, T_{3_2}, T_{3_3}\}$. If $\lambda_4 = 2$ the values in $\{1, 2\}$ must be assigned the three trees of level $4-1=3$ following the explained rules. There are three such assignments:

$$\mathcal{T} = \{T_{0_7}, T_{1_2}, \overset{1}{T}_{3_1}, \overset{2}{T}_{3_2}, T_{3_3}\}$$
$$\mathcal{T} = \{T_{0_7}, T_{1_2}, \overset{1}{T}_{3_1}, T_{3_2}, \overset{2}{T}_{3_3}\}$$
$$\mathcal{T} = \{T_{0_7}, T_{1_2}, T_{3_1}, \overset{1}{T}_{3_2}, \overset{2}{T}_{3_3}\}$$

The three selections result in $S = \{T_{3_1}, T_{3_2}\}$, $S = \{T_{3_1}, T_{3_3}\}$ and $S = \{T_{3_2}, T_{3_3}\}$ respectively. ◇

Example 3.20 Assume that we are currently at the *second* substep for construction level 4 and $\mathcal{T} = \{T_{2_2}, T_{3_1}, T_{3_2}, T_{3_3}, T_{3_4}\}$. Also assume $\lambda_4 = 2$ and

the fist substep selected the set $S = \{T_{3_2}, T_{3_4}\}$. Now the values $\{\text{NULL}, 1, 2\}$ can be assigned to the remaining trees in the following 6 feasible ways (NULL values are represented by "-"):

$$\mathcal{T} = \{\overset{-}{T}_{2_2}, \overset{-}{T}_{3_1}, \overset{1}{T}_{3_2}, \overset{-}{T}_{3_3}, \overset{2}{T}_{3_4}\} \qquad \mathcal{T} = \{\overset{-}{T}_{2_2}, \overset{-}{T}_{3_1}, \overset{1}{T}_{3_2}, \overset{1}{T}_{3_3}, \overset{2}{T}_{3_4}\}$$
$$\mathcal{T} = \{\overset{1}{T}_{2_2}, \overset{-}{T}_{3_1}, \overset{1}{T}_{3_2}, \overset{-}{T}_{3_3}, \overset{2}{T}_{3_4}\} \qquad \mathcal{T} = \{\overset{1}{T}_{2_2}, \overset{-}{T}_{3_1}, \overset{1}{T}_{3_2}, \overset{1}{T}_{3_3}, \overset{2}{T}_{3_4}\}$$
$$\mathcal{T} = \{\overset{2}{T}_{2_2}, \overset{-}{T}_{3_1}, \overset{1}{T}_{3_2}, \overset{-}{T}_{3_3}, \overset{2}{T}_{3_4}\} \qquad \mathcal{T} = \{\overset{2}{T}_{2_2}, \overset{-}{T}_{3_1}, \overset{1}{T}_{3_2}, \overset{1}{T}_{3_3}, \overset{2}{T}_{3_4}\}$$

The last assignment results in the sets $T_{4_1} = \{T_{3_2}, T_{3_3}\}$ and $T_{4_2} = \{T_{2_2}, T_{3_4}\}$. For level 5 we thus get $\mathcal{T} = \{T_{3_1}, T_{4_1}, T_{4_2}\}$. The remaining assignments are all infeasible because they result in one or more singleton sets for level 4. $\diamond$

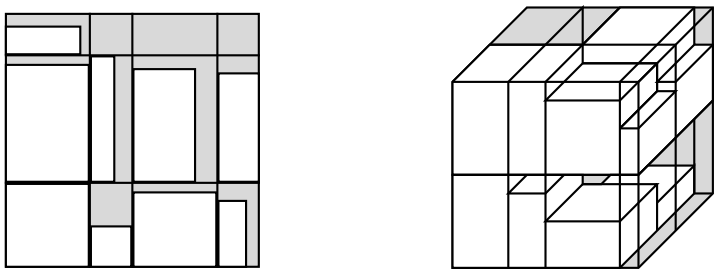
An assignment of all $T \in \mathcal{T}$ at level j corresponds to the construction of all packing trees of depth j (without any assignment of cut-directions to the branch nodes). Each feasible assignment at level k is a packing tree which must be feasibility checked. The set $(\lambda_0, \ldots, \lambda_k)$ has been fully checked when all feasible assignments for all levels $0, \ldots, k$ has been tried out systematically with respect to this set.

Chapter 4
Sticky cutting

The guillotine property ensures that each packing can be split into a number of cut slices $U_1, \ldots, U_n$ by a number of face parallel hyper planes that cut through the whole container. Each cutting slice may then eventually be cut again separately – independent from the surrounding cut slices. Also the cut slices U_i and U_j, $i \neq j$ might have different cut directions.

Consider a guillotine cutting situation in which it is more important to optimize for cutting speed instead of minimizing material waste. Such a situation could be a glass cutting environment where the process of partitioning a glass sheet, splitting each cut slice apart and then remount each cut slice again for further cutting is more expensive than making all guillotine cuts on the same sheet at once and then just re-melt and reuse exceeding material. A similar situation where it might pay off to make all guillotine cuts at once is when fragile material is cut. The separating step between each guillotine cut might damage the material and therefore it may be necessary to minimize the amount of remounts and displacements. Yet another situation requirering such a non-splitting cutting is when mounting thin and light weight partition walls in large halls, e.g. machine halls, fabricating shops or engine rooms. Often such partition walls are spanning through the whole hall and mounted on the solid and strong outer walls and they will therefore form a grid-like division of the hall when seen from a top view.

We will in this section look at a special kind of guillotine packings where cut slices are considered *sticky*. That is, all cuts must be made through the original container and first when all cuts have been made, the individual parts can be split apart. Figure 4.1 on the next page shows an example of such *sticky cutting* for $d = 2$ and $d = 3$. It is relatively obvious that the sticky cutting property implies worse packing than conventional guillotine cuttings. In the following we will try to estimate this and investigate how



(a) $d = 2$ (b) $d = 3$

Figure 4.1: Example of sticky cuttings

Fekete and Schepers algorithm perform when this property is required to be satisfied.

One immediate observation is that $k = d$ for sticky cuttings: If $k < d$ we can instead reduce the problem to a k dimensional packing. Also all packings with $k > d$ are equivalent to $k = d$ packings as all cuts are made by planes through the whole original container. There are at most d such plane orientations. The following lemma is essential for the rest of this chapter:

Lemma 4.1 *For a packing p satisfying the sticky cutting property, let U be a cut slice orthogonal to the x_i-axis. For all $u, v \in U$ we have $(u, v) \in E_i$ or equivalently: $G_i[U]$ is a clique. Also, for all $u \in U$ and $v \in V \setminus U$ we have $(u, v) \notin E_i$.*

PROOF Without loss of generality we may assume that all $p(v)$ are aligned towards origin for all $v \in V$. For all $u, v \in U$ we have $I_i(u) \cap I_i(v) \neq \emptyset$ or equivalent: $(u, v) \in E_i$. If $v \notin U$ we have $I_i(u) \cap I_i(v) = \emptyset$ and thus $(u, v) \notin E_i$. □

Lemma 4.1 tells that all connected components in G_i are cliques and that the cliques in G_i are exactly given by the cut slices along the x_i-axis.

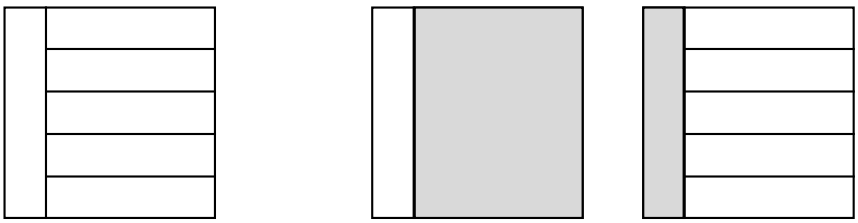
4.1 Performance

An interesting aspect is how good a solution we can find for the OBPP-d problem when using sticky cuttings instead of guillotine cutting or unrestricted packings. It showed up to be difficult to prove an exact performance ratio for bin packings satisfying the sticky cut property. Therefore this sec-

tion only covers a proof of a lower bound for this ratio and gives a conjecture for an upper bound. Let OPT_g and OPT_s denote the number of containers required to make a packing of V satistying the guillotine and sticky cutting property respectively.

Lemma 4.2 *For $d = 2$ we have $\frac{OPT_s}{OPT_g} \geq 2$.*

PROOF Consider an instance with 1 box of size $(1, n)$, $n - 1$ boxes of size $(n - 1, 1)$ and $W = (n, n)$. This can be packed into a single container with a guillotine packing. Clearly, the only way of satisfying the sticky cut property is to pack the different box types in two separate containers. Figure 4.2 shows this packing transformation. □



(a) Guillotine packing

(b) A sticky packing

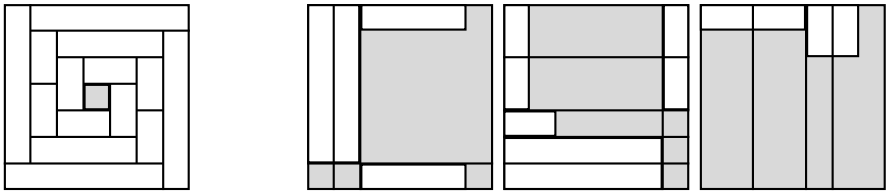
Figure 4.2: Instance transformation used in proof of lemma 4.2

Now, still for $d = 2$, let p be an arbitrary packing and OPT the minimum number of containers needed. The attempt to prove an upper bound for $\frac{\mathrm{OPT}_s}{\mathrm{OPT}}$ or $\frac{\mathrm{OPT}_s}{\mathrm{OPT}_g}$ was unsuccessful. No instance with $\frac{\mathrm{OPT}_s}{\mathrm{OPT}_g} > 2$ was ever found and in fact it was a surprisingly hard job to find an instance having $\frac{\mathrm{OPT}_s}{\mathrm{OPT}} > 2$. An instance which is not guillotine packable *might* be rearranged to satisfy the sticky cut property using only the double amount of containers. E.g. the concentric G4 instance in figure 3.3(a) on page 31 can be made sticky cuttable by separating vertical and horizontal boxes in two containers.

Lemma 4.2 taken into account, one should expect $\frac{\mathrm{OPT}_s}{\mathrm{OPT}} \leq 4$ but an instance with a ratio of 4 was never found. Figure 4.3 on the following page shows an example of an instance with $\frac{\mathrm{OPT}_s}{\mathrm{OPT}} = 3$. Notice, that the third container is only sparingly filled with a few "left over" boxes.

The observations are collected in the following conjecture:

Conjecture 4.3 *For $d = 2$ we have $\frac{OPT_s}{OPT_g} = 2$ and $\frac{OPT_s}{OPT} = 4$.*



(a) (b)

Figure 4.3: The packing in 4.3(a) will use minimum 3 containers when transformed to a sticky cutting. An example of such a transformation is seen in 4.3(b).

4.2 Behavior of sticky cuttings

Consider the decision problem OPP-d extended with the sticky cutting requirement. In theorem 3.2 on page 28 we showed that OPP-d with the conventional guillotine restriction was NP-hard. No such proof was ever found for sticky cuttings but the indications are that these are also NP-hard:

Conjecture 4.4 *OPP-d attached with the sticky cutting requirement is NP-hard.*

With lemma 4.1 in hand, let us take a look at P1, P2 and P3: P1 requires all graphs to be interval graphs. According to theorem 2.8 on page 15 we need to show that G_i does not contain a chordless C_4 and $G_i^{\complement}$ is a comparability graph. As all connected components in G_i are cliques, the graph clearly has no chordless C_4. According to theorem 2.9 on page 17 $G_i^{\complement}$ is a comparability graph if and only if it does not contain any odd 2-chordless cycle. Assume that $G_i^{\complement}$ indeed contains such a cycle $C_n = \{0, \ldots, n-1\}$ of length $n > 4$. $G_i \cup G_i^{\complement}$ is always complete and thus, if the edge (u, v) is a 2-chord for C_n we must have $(u, v) \notin E_i^{\complement}$ and $(u, v) \in E_i$. All these 2-chord edges form a connected subgraph of G_i which is not a clique. This is a contradiction implying that $G_i^{\complement}$ must be a comparability graph after all. In our situation, where all connected components are cliques, we have now shown that P1 is always satisfied removing the need for verifying this property in any branch.

P2 requires every stable set to be x_i-feasible. Consider a cutting stage orthogonal to the x_i-axis, resulting in the cut slices $U_1, \ldots, U_n$. As all cuts are made through the whole container, no successing stage will cut any of the cut slices further by planes orthogonal to the x_i-axis. Thus, the x_i-width of a cutting slice must equal the max width of any of its elements. By

summarizing each cut slice width the x_i-width of the whole packing can be found:

$$\sum_{\mathscr{C} \text{ clique in } G_i} \max_{v \in \mathscr{C}} w_i(v)$$

With an appropriate data structure this can be calculated in $O(V)$ time. This is a speed improvement compared to the $O(dk|V|^2)$ complexity of the dissolvement based measurement described for guillotine cuttings in section 3.2.

An interesting issue is how to handle infeasible stable sets. If a packing is x_i-infeasible let

$$M = \{v \mid w_i(v) = \max_{v \in \mathscr{C}} w_i(v) \text{ and } \mathscr{C} \text{ clique in } G_i\}$$

be a set containing the x_i-widest box from each cut slice of G_i. The edge set M induces a maximal weighted infeasible clique $\mathcal{E}^{\mathtt{C}}_{+,i}$. We branch on each edge from $\mathcal{E}^{\mathtt{C}}_{+,i}[M] \setminus \mathcal{E}_{-,i}$. If this set is empty the graph cannot be made x_i-feasible: Drop the branch.

P3 is ensured as in the conventional case by considering $\cap_{i=1}^{d} \mathcal{E}_{+,i}$.

The sticky cutting property does not only influence on the handling of P1 and P2. Lemma 4.1 also implies an additional rule for argumenting $\mathcal{E}_-$ and $\mathcal{E}_+$: Let $\mathscr{C}$ be a clique in G_i and consider the augmentation (e, σ, i) where $e = (u, v)$. If $v \in \mathscr{C}$ and $u \notin \mathscr{C}$ also the augmentation (f, σ, i) should be made for all $f = (u, v')$ where $v' \in \mathscr{C}$. In other words: For sticky cuttings the concept of augmenting edges is replaced by the concept of joining cliques.

The solution space for sticky cuttings are smaller than for guillotine packings and sticky cuttings will also generally make worse packings as described in section 4.1. However, as widths can be measured in linear time and the clique representation will always result in topologically feasible sticky cuttings more sticky cutting nodes can be processed per second than for guillotine packings. Hence we may hope that we obtain comparable solutions by searching larger parts of the solution space. In open problem 3.11 on page 38 we ask if the dissolvement technique increases the total complexity for guillotine packings even though the complexity for each iteration has been decreased drastically. A similar question can be asked for the clique merging algorithm for sticky cuttings: Each clique configuration in a set of graphs $G_i = \mathcal{E}_{+,i}$ will occur in numerous search nodes and the computational results shown in chapter 5 render probable that the time used in these redundant search nodes surpass the time gained by reducing the complexity of each iteration.

4.2.1 Further reduction of the search space

Consider a first-stage cut slice U orthogonal to the x_i-axis. The cut slice dimensions will then equal the container dimensions except for dimension x_i where it will equal the size of the x_i-widest box in the slice. If another cut slice from the x_i-orthogonal cut is merged to U (the two cliques representing the cut slices are joined) the resulting cut slice size will remain unchanged except at dimension x_i where the width will equal the new x_i-widest box. If $v \in V$ is the x_i-widest box in V then clearly no x_i-orthogonal cut slice will be x_i-wider than $w_i(v)$.

Let $\mathcal{V}$ denote the volume of the largest possible x_i-orthogonal feasible cut slice: $\mathcal{V} = W_1 W_2 \cdots w_i(v) \cdots W_d$. If the summarized volume of the boxes in an arbitrary x_i-orthogonal cut slice exceeds $\mathcal{V}$ then no augmentation can ever make the search node feasible and the node can therefore be dropped.

4.3 The special case $d = 2$ and $k = 2$

As we will see in this section the sticky cutting benefits can be utilized in Fekete and Schepers branch-and-bound algorithm for conventional guillotine packings in the special case $d = 2$ and $k = 2$. The key observation is the following: Assume again that for all $p(v), v \in V$ that $p(v)$ is aligned to origin. Lemma 4.1 ensures that for each first-stage cut slice U the boxes in U form a clique in G_{α_1} and also the cliques containing u and v are disconnected if u and v are placed in different fist-stage cut slices. The first-stage cut slices and second-stage cut slices of each first-stage cut slice may be freely permuted without affecting feasibility. Therefore, for this special case, it is sufficient to know how V is distributed into the the first-stage cut slices in order to determine feasibility. I other words: G_{α_1} holds all information needed making G_{α_2} superfluous. In order to verify feasibility it is sufficient to verify the following:

1. For each first-stage slice, the height[1] of all boxes does not exceed the container height: For each clique $\mathscr{C}$ in G_{α_1},

$$\sum_{v \in \mathscr{C}} w_{\alpha_2}(v) \leq W_{\alpha_2}$$

2. The sum of each first-stage slice widths does not exceed the container width. Each slice is exactly as wide as the widest box it contains so we

[1]The words "height" and "width" may be exchanged depending of the value of α_1

need to check:

$$\sum_{\mathscr{C} \text{ clique in } G_{\alpha_1}} \max_{v \in \mathscr{C}} w_{\alpha_l}(v) \leq W_{\alpha_l}$$

Like in sticky cuttings, the P1 property is automatically satisfied in this case as all connected components are cliques. Furthermore P3 is trivially satisfied as we only consider a single graph, G_{α_1}.

So for the case $d = 2$ and $k = 2$ we can boost each iteration in the branch-and-bound algorithm even further.

Chapter 5
Computational results

Some of the ideas presented for guillotine packings (GP) described in chapter 3 and the sticky cuttings (SC) described in chapter 4 have been implemented and run on a number of instances. This chapter presents and discusses the computational results obtained.

The source code can be found at the following URL:

`http://resen.org/~rra/pub/mastersource.tgz`

5.1 Strategy

The following two OPP-d algorithms have been implemented:

- A GP solver based on the polynomial time dissolvement technique for measuring and clique finding described in chapter 3. Dissolvements are tried out in the order $1, \ldots, d$ as suggested and this implementation does *not* require P1 to be satisfied.
- A SC solver, exactly as described in chapter 4, including the bound described in section 4.2.1 on page 56.

Both solvers was first tested on a small set of handpicked instances with various specific properties. These instances are described in section 5.1.1 on the following page. The solvers were moreover tested on a larger set of 11400 systematically created instances for various values of d, $|V|$ and $\frac{\mathcal{V}_b}{\mathcal{V}_c}$ where $\mathcal{V}_b$ and $\mathcal{V}_c$ are the total box- and container volume, respectively. All the latter instances are guaranteed to be feasible. For each tuple $(T, d, |V|, \frac{\mathcal{V}_b}{\mathcal{V}_c})$ where T is a packing type (e.g. *guillotine* or *sticky*) a set of 20 instances has been randomly created with 20 different random seeds. Notice that the instances

created for GP are different from the SC instances as they have different requirements to satisfy in order to be feasible. The instance generator used is available as part of the source code.

For comparison reasons it was considered to test the solvers against a set of well known instances but no such instances was found for the decision problem for guillotine packings. For obvious reasons, no instances were found for SC packings either. One solution was to use the OPP-d solvers as part of an OBPP-d solver and run the modified OBPP-d solver on some of the many available instances for OBPP-d. But in order to use the results obtained for comparison with other OPP-d solvers, these solvers would also need to be tested by using them as part of the exact same OBPP-d solver. Due to the fact that the instance generator used is public available and provides OPP-d instances for both GP and SC, it was therefore estimated that it was more relevant by using only the systematic tests described above.

The implemented GP solver was furthermore tested against a state-of-the-art solver in order to give a picture of its "absolute" performance. The best GP solver found was an implementation of a CSP based OPP-2 algorithm [PS02] which is also described in section 6.3.3. As no other implementation of SC solvers currently exist, it has not been possible to compare the present SC solver implementation with any similar solvers.

5.1.1 Handpicked instances

The handpicked instances have been specifically constructed in order to test the behavior of the algorithms in different kinds of extreme situations. Below we describe each class of instances in more detailes.

2dunique8, 2dunique16 and 3dunique16 These instances have only one unique solution except for mirror symmetry. They are therefore expected to be rather hard to solve. All of them are guillotine packable but not sticky cuttable.

2dsquare25, 2dsquare100 and 3dsquare8 Another attemt to test the limits of the solvers was to create these three instances with the general form of a square grid in 2 and 3 dimensions with cells (boxes) of unit sizes. That is, for $d = 2$ and $n \in \{5, 10\}$ there are n^2 boxes of size $(1, 1)$ in a container size of (n, n). For $d = 3$ there are $2^3 = 8$ boxes of size $(1, 1, 1)$ in a container size of $(2, 2, 2)$.

2dinf7, 2dinf8, 3dinf8, 3dinf300 Four nontrivially guillotine-infeasible instances. That is, the infeasibility is caused by existence of a nonbreak-

able blocked ring and not by insufficient container volume. The instances consist of a large container, some small boxes and a blocked ring induced by two boxes of size $(W_1 - 1, 1)$ and two boxes of size $(1, W_2 - 1)$ for the case $d = 2$ and three boxes of size $(W_1, 1, 1)$, $(1, W_2, 1)$ and $(1, 1, W_3)$ for the case $d = 3$. The latter blocked ring is depicted as the left part of figure 3.2(b) on page 29. As none of the algorithms will occasionally find any feasible solution, the size of the solution space is expected to play a large role in the execution time.

nx**t**y These instances are created by the test environment from the CSP solver. Their purpose is to give an idea of how the solvers perform on the "home ground" for the CSP solver.

5.2 Implementation

Both solvers were implemented using C++ with extensively use of the Standard Template Library (STL). In order to represent sets of nodes with both fast lookup, insertion and random deletion the `stl::hash_set` container was used. Both the lookup and deletion operations have a complexity of $O(1)$ and if no rehashing is needed also the insertion takes $O(1)$. Where maps from an identifier to a larger data structure was needed the associative container `stl::hash_map` was used in order to obtain the same direct access performance as for sets. For queue-like data structures and in cases where random deletion was not required the container `std::vector` was used. The `std::vector` has a constant time index-operator and provides constant time insertion and deletion of elements at the end.

Section 5.4.2 on page 76 explores the implementation by analyzing a set of runs with the tool gprof. As the section describes, the practical performance of the STL hash structures is surprisingly slow for the small data sets used in our case. It was therefore decided to test an additional implementation using another kind of data structures. The section gives a more detailed motivation for this experiment and also describes the new data structures used. The two implementations will be referred to as *hash* and *static* respectively in the figures showing the results optained.

5.3 Results

The source code was compiled with g++ 4.0.2 and run on a machine with a 2170 MHz AMD Athlon XP 2600+, 512 MB ram and a Linux installation

with kernel 2.6.11.8.

5.3.1 Handpicked instances

Figure 5.1 and 5.2 shows the computational results for the runs performed on the handpicked instances. The timeout was set to 300 seconds. Below

				Guillotine solver			CSP	Sticky cutting solver		
Instance	d	$\|V\|$	$\frac{\mathcal{V}_b}{\mathcal{V}_c}$	Feas.?	Nodes	Sec.	Sec.	Feas.?	Nodes	Sec.
2dunique8	2	8	1.00	Yes	12	0.00	0.00	No	6525	0.7
2dunique16	2	16	1.00	Yes	53	0.05	0.00	No	-	>300
2dsquare25	2	25	1.00	Yes	-	>300	0.00	Yes	139	0.24
2dsquare100	2	100	1.00	Yes	-	>300	63.00	Yes	1079	2.95
2dinf7	2	7	0.04	No	9673	2.93	0.00	No	3301	0.35
2dinf8	2	8	0.04	No	192961	61.8	0.00	No	16547	1.93
n20t21	2	20	0.35	Yes	68	0.04	0.02	?	-	>300
n29t21	2	29	0.40	Yes	79	0.08	0.17	?	-	>300
n18t22	2	18	0.59	Yes	64	0.04	0.01	?	-	>300
n10t23	2	10	0.57	Yes	33	0.01	0.00	Yes	516763	73.32

Figure 5.1: Computational results for the handpicked instances for $d = 2$ and a timeout of 300 seconds. The table colums are divided in four main sections: 1) Common instance data, e.g. number of boxes $|V|$ and volume percentage $\frac{\mathcal{V}_b}{\mathcal{V}_c}$, 2) results for the GP solver, 3) runtimes for the same instances solved by the CSP solver and 4) results for the SC solver. *Feas.?* tells if the instance is feasible for that type of packing, *Nodes* is the number of search nodes visited and *Sec.* is the number of seconds used for solving the instance.

				Guillotine solver			Sticky cutting solver		
Instance	d	$\|V\|$	$\frac{\mathcal{V}_b}{\mathcal{V}_c}$	Feas.?	Nodes	Sec.	Feas.?	Nodes	Sec.
3dunique16	3	16	0.90	Yes	36	0.05	No	-	>300
3dsquare8	3	16	1.00	Yes	-	>300	Yes	11733	1.72
3dinf8	3	8	0.00	No	1	0.00	No	13049	1.76
3dinf300	3	300	0.48	No	1	0.05	No	-	>300

Figure 5.2: Results for handpicked instances for $d = 3$.

we list some observations.

1. The CSP solver outperforms both the GP and SC solver on most instances.

2. For the nxty instances the GP solver almost matches the CSP solver and it even performs better on the n29t21 instance.

3. Declaring an instance infeasible is generally harder than finding a solution: There might be several feasible solutions to an instance, but

in order to declare it infeasible either a good bound must be used or a potentially large search space has to be investigated more or less throughout. The search space for sticky cuttings are smaller than for guillotine packings and as more nodes can be handled per second this may explain why the SC solver outperforms the GP solver in 2dinf7 and 2dinf8. For $d = 2$ the CSP solver is still fastest though. For $d = 3$ the GP solver discovers the unbreakable blocked ring in 3dinf8 and 3dinf300 very fast. The latter instance, having $|V| = 300$ and a box/container volume ratio of 0.48 was declared infeasible in just 0.05 seconds.

4. The SC solver generally performs well on the "square" instances, compared to the GP solver. It even outperforms the CSP solver in the 2dsquare100 instance. The square instances are characterized by having a large amount of symmetry: The packings can be permuted in numerous ways because of the many equally sized boxes.

5. The SC solver seems to meet increased difficulties for most instances at $d = 3$. One exception is the 3dsquare8 instance which was solved in 1.72 seconds on the SC solver but stayed unsolved by the GP solver.

The handpicked instances are all somewhat extreme except for the nxty instances and does not give an objective picture of the solvers (because they were handpicked). So even though the above observations suggest some characteristics of the solvers a much larger set of test instances must be solved in order to provide a trustworthy characterization of their average performance. Section 5.3.2 below describes the results obtained by a more exhaustive investigation.

5.3.2 Systematically created instances

In this section we present the results from a set of 11400 systematically created instances and examine if some of the tendencies seen for the handpicked instances also hold in the general case. Each parameter set $(T, d, |V|, \frac{\mathcal{V}_b}{\mathcal{V}_c})$ where $T \in \{\text{guillotine/hash}, \text{guillotine/static}, \text{sticky/hash}, \text{sticky/static}, \text{CSP}\}$ was tested against 20 randomly created instances matching these parameters and each number or curve-point presented in this section is therefore based on 20 test runs.

A timeout was given for all executions. In order to determine an appropriate threshold a number of initial test runs were made with a relatively high timeout limit. It appeared that the execution times were far from evenly

distributed: For most runs, either the execution timed out or a solution was found in only a few seconds. It also turned out that the threshold for divergence in execution time increased with d. Based on these observations the following timeout thresholds were decided:

d	2	3	4
Seconds	6	20	30

Each figure in this section shows the percentage of runs exceeding the timeout threshold and the average execution time for the runs that did not time out. The tables also show the average number of search nodes visited.

Hash versus static structures

Figure 5.3 on the facing page shows a number of graphs depicting the timeout percentage (dashed curve) and average execution time (solid curve) for the two GP solver implementations for $d = 2$ and $|V|$ running from 4 to 30. Each graph shows data for a specific value of $\frac{\mathcal{V}_b}{\mathcal{V}_c}$. The table in figure 5.4 on page 66 shows a subset of the execution and marks the best execution time in each row with bold. With respect to the timeout ratio, the two implementations are relatively even. The implementation using static data structures, however has a slightly less ratio than the hash based implementation. Still it might happen that the former implementation terminates before the latter on isolated instances: The constructor for the static sized data structure is relatively time consuming so an instance for which disproportionately many creations of the structures are done compared to how much they are actually utilized (updated or indexed into) might perform better with hashing based structures.

The average execution times and number of search nodes visited for the runs that did not time out are generally best for the implementation using static structures. The average number of search nodes visited per second was roughly 400 for the hash based implementation and 1100 for the static based. However, for isolated instances the average execution time is largest for the latter, e.g. for $\frac{\mathcal{V}_b}{\mathcal{V}_c} = 40$ and $|V| = 18$. This can be explained by the timeout ratio for these cases: If an instance is solved by one implementation but is timing out on the other, the (large) execution time will increase the average execution time for the former but not influence on the average for the latter. Thus, this situation might distort the picture of the actual performance.

The graphs in figure 5.5 on page 67 and the table in figure 5.6 on page 68 compares the two implementations for the sticky cutting solver for $d = 2$. The two implementations of the SC solver share the same characteristics as

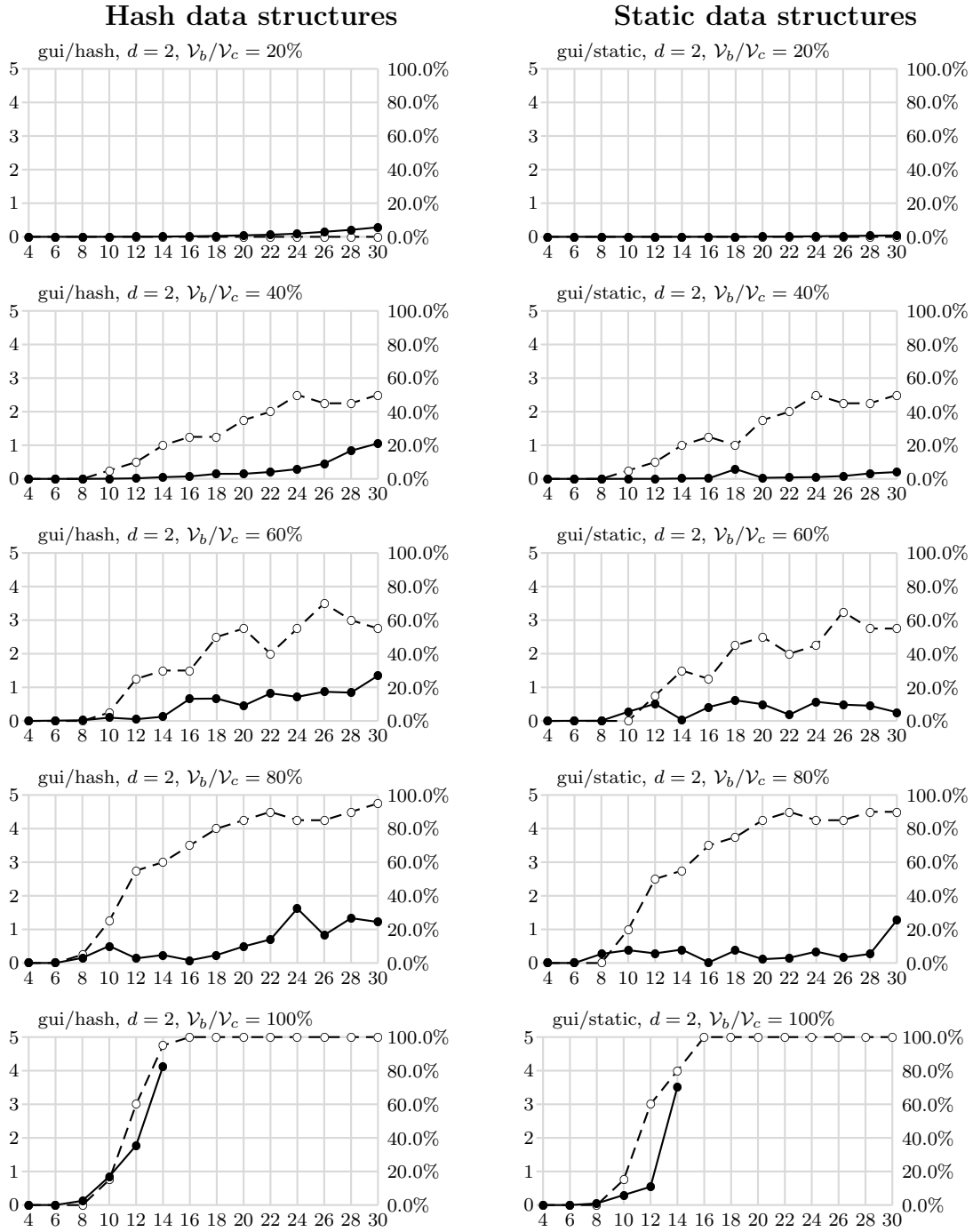


Figure 5.3: Computational results for the two guillotine packing solver implementations for $d = 2$ with a timeout of 6 seconds. The number if boxes $|V|$ is shown horizontaly. The curves show the percentage of runs timing out after 6 seconds (dashed) and the average runtime in seconds for the runs that did not time out (solid). Left (right) column shows results for the algorithm using hashing (static) data structures. $\mathcal{V}_b/\mathcal{V}_c$ is the ratio between the total box- and container volume.

$\frac{\mathcal{V}_b}{\mathcal{V}_c}$	$\lvert V\rvert$	Hash #Nodes	Hash Timeouts	Hash Sec.	Static #Nodes	Static Timeouts	Static Sec.
20	6	2	0.0%	0.000	2	0.0%	0.000
20	12	4	0.0%	0.004	4	0.0%	**0.000**
20	18	10	0.0%	0.026	10	0.0%	**0.002**
20	24	25	0.0%	0.095	25	0.0%	**0.015**
20	30	55	0.0%	0.283	55	0.0%	**0.043**
40	6	2	0.0%	0.000	2	0.0%	0.000
40	12	11	10.0%	0.016	11	10.0%	**0.002**
40	18	72	25.0%	**0.150**	666	20.0%	0.291
40	24	74	50.0%	0.290	74	50.0%	**0.054**
40	30	196	50.0%	1.056	196	50.0%	**0.203**
60	6	19	0.0%	0.009	19	0.0%	**0.004**
60	12	38	25.0%	**0.051**	1284	15.0%	0.510
60	18	399	50.0%	0.666	1458	45.0%	**0.613**
60	24	265	55.0%	0.718	1077	45.0%	**0.570**
60	30	328	55.0%	1.347	328	55.0%	**0.257**
80	6	4	0.0%	0.001	4	0.0%	**0.000**
80	12	112	55.0%	**0.138**	703	50.0%	0.286
80	18	106	80.0%	**0.237**	664	75.0%	0.376
80	24	614	85.0%	1.620	614	85.0%	**0.340**
80	30	312	95.0%	**1.220**	1437	90.0%	1.275
100	6	5	0.0%	0.001	5	0.0%	**0.000**
100	12	1544	60.0%	1.775	1544	60.0%	**0.544**
100	18	–	100.0%	–	–	100.0%	–
100	24	–	100.0%	–	–	100.0%	–
100	30	–	100.0%	–	–	100.0%	–

Figure 5.4: Results for the guillotine packing solvers for $d = 2$ with a timeout of 6 seconds. The column *#Nodes* holds the average number of search nodes visited for the runs that did not time out. The best average execution time is marked with bold.

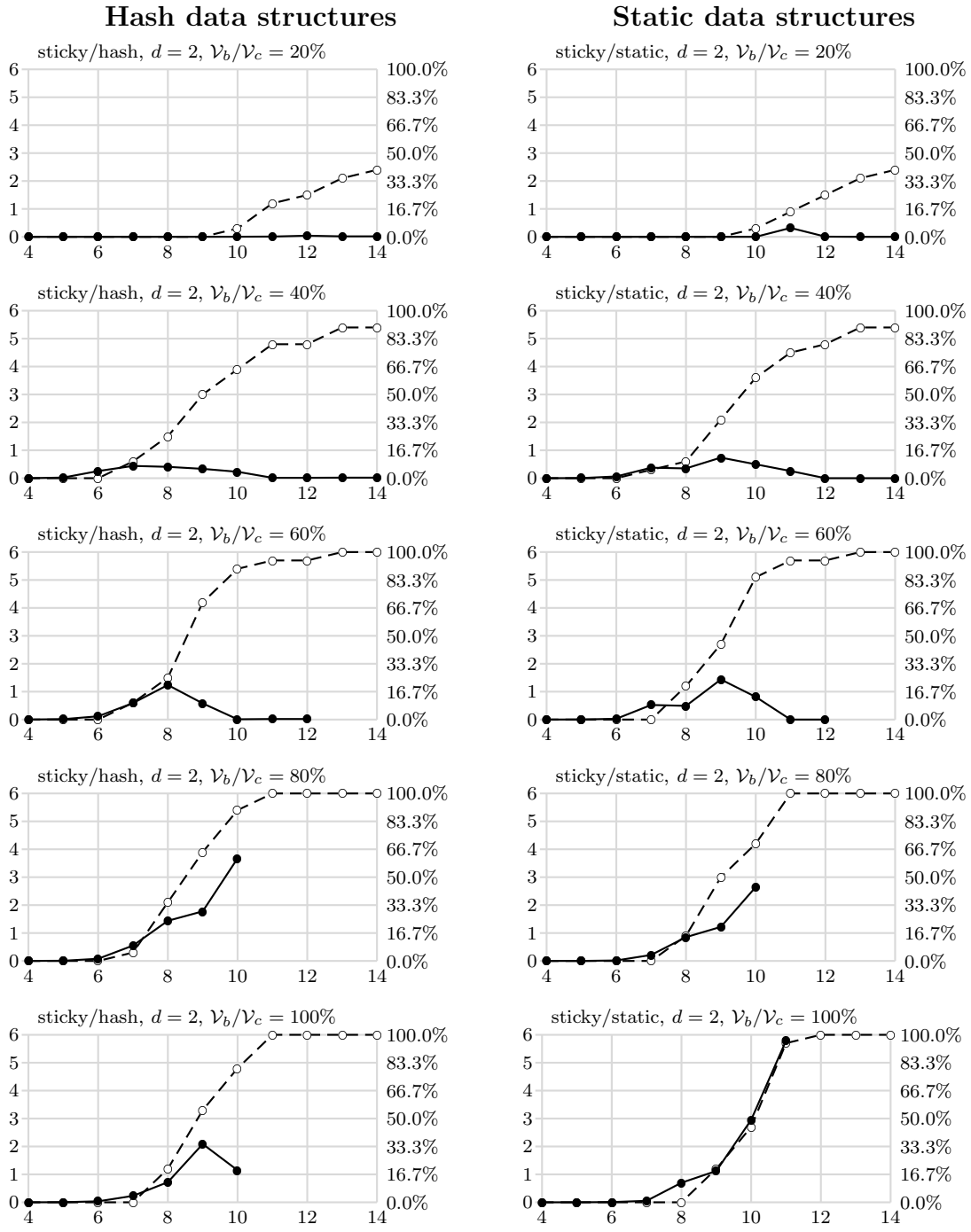


Figure 5.5: Computational results for the two sticky cutting solver implementations for $d = 2$ with a timeout of 6 seconds.

		Hash			Static		
$\frac{\mathcal{V}_b}{\mathcal{V}_c}$	$\lvert V\rvert$	#Nodes	Timeouts	Sec.	#Nodes	Timeouts	Sec.
20	4	2	0.0%	0.000	2	0.0%	0.000
20	6	3	0.0%	0.000	3	0.0%	0.000
20	8	5	0.0%	0.000	5	0.0%	0.000
20	10	7	5.0%	0.004	7	5.0%	**0.000**
20	12	51	25.0%	0.041	51	25.0%	**0.007**
20	14	11	40.0%	0.018	11	40.0%	**0.000**
40	4	5	0.0%	0.000	5	0.0%	0.000
40	6	657	0.0%	0.256	657	0.0%	**0.063**
40	8	786	25.0%	0.407	2996	10.0%	**0.351**
40	10	339	65.0%	**0.230**	3454	60.0%	0.498
40	12	15	80.0%	0.015	15	80.0%	**0.000**
40	14	13	90.0%	0.020	13	90.0%	**0.000**
60	4	8	0.0%	0.001	8	0.0%	**0.000**
60	6	304	0.0%	0.117	304	0.0%	**0.028**
60	8	2436	25.0%	1.239	4177	20.0%	**0.491**
60	10	14	90.0%	**0.010**	5980	85.0%	0.813
60	12	26	95.0%	0.020	26	95.0%	**0.000**
60	14	–	100.0%	–	–	100.0%	–
80	4	9	0.0%	0.001	9	0.0%	**0.000**
80	6	201	0.0%	0.078	201	0.0%	**0.017**
80	8	2842	35.0%	1.438	7297	15.0%	**0.852**
80	10	5458	90.0%	3.675	20807	70.0%	**2.647**
80	12	–	100.0%	–	–	100.0%	–
80	14	–	100.0%	–	–	100.0%	–
100	4	8	0.0%	0.000	8	0.0%	0.000
100	6	98	0.0%	0.035	98	0.0%	**0.008**
100	8	1455	20.0%	0.721	6097	0.0%	**0.701**
100	10	1890	80.0%	**1.147**	22684	45.0%	2.955
100	12	–	100.0%	–	–	100.0%	–
100	14	–	100.0%	–	–	100.0%	–

Figure 5.6: Results for the sticky cutting solvers for $d = 2$ with a timeout of 6 seconds.

the GP implementations: In general, the implementation using static structures performed better with smaller timeout ratio and faster execution times. The average number of search nodes visited per second was roughly 1500 for the hash based implementation and 5200 for the static based. As for the GP solvers, the hash based implementation has smallest average execution time for isolated instances but this can be explained as above.

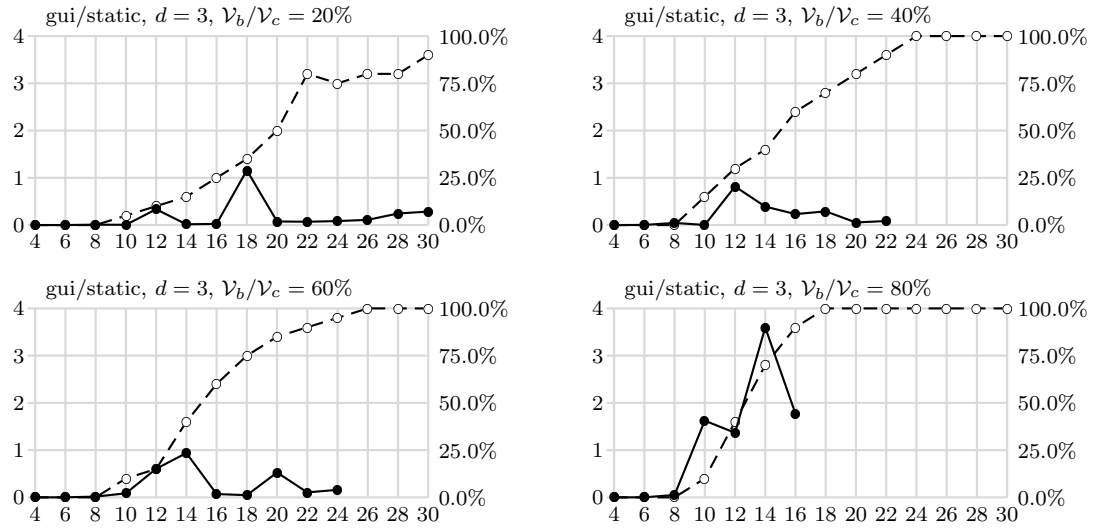
The implementation based on static data structures generally performs best, all tests on higher-dimensional instances are therefore done on this implementation only.

The GP solver

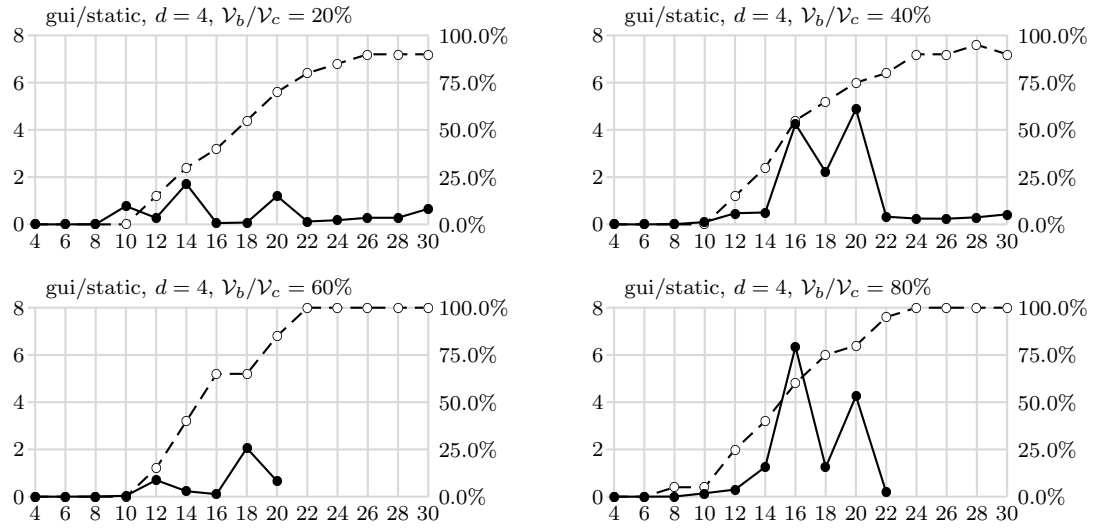
In this section we take a closer look at the results for the GP solver. Consider the graphs in the right column of figure 5.3 on page 65 showing execution data for $d = 2$. For the small volume percentage $\frac{\mathcal{V}_b}{\mathcal{V}_c} = 20\%$ the solver handles all given instance sizes without any difficulties. As $\frac{\mathcal{V}_b}{\mathcal{V}_c}$ increases the problems become more difficult to solve as it leaves less tolerance for the box placements. The difficulties are reflected in increased timeout ratio (the termination time for the instances that did not time out is relatively constant). The volume percent has a big influence on the performance, as seen, and it seems that the influence can be divided in three phases: Increment in the volume percentage from either a small or a high value has a large influence on the timeout ratio. This can be seen in the increment from 20% to 40% and the increment from 80% to 100%. In the middle section the volume percentage does not affect the timeout ratio significantly. This can be seen in the increase from 40% to 60%.

For $\frac{\mathcal{V}_b}{\mathcal{V}_c} = 80\%$ there is a slight drop in the timeout ratio for large $|V|$. This might be explained by a "fortunate" combination of $|\frac{\mathcal{V}_b}{\mathcal{V}_c}|$ and $|V|$: If a small set of large $|V|$-values detects infeasible search nodes a bit earlier for that specific value of $\frac{\mathcal{V}_b}{\mathcal{V}_c}$ then the total execution time will benefit from this and the timeout ratio therefore decrease.

Figure 5.7 on the next page shows computational results for the GP solver for $d = 3$ and $d = 4$. The tests on the higher-dimensional instances have some of the same characteristics as in the two-dimensional case. The problems are clearly harder to solve though: For small values of $\frac{\mathcal{V}_b}{\mathcal{V}_c}$ and even for small $|V|$ the timeout ratio is relatively high. Also the slope of the timeout ratio curve is steeper than for $d = 2$. However, for $d = 3$ the influence of $\frac{\mathcal{V}_b}{\mathcal{V}_c}$ is less significant than for $d = 2$ and for $d = 4$ the timeout ratio is almost unaffected of the volume percentage.



(a) Results for $d = 3$. Timeout threshold: 20 seconds.



(b) Results for $d = 4$. Timeout threshold: 30 seconds.

Figure 5.7: Execution data for the guillotine solver for $d = 3$ and $d = 4$.

Also, it is interesting that for $\frac{\mathcal{V}_b}{\mathcal{V}_c} = 80\%$ the solver is able to solve instance with $|V| = 16$ for $d = 3$ but instances with $|V| = 22$ for $d = 4$.

Comparison to state-of-the-art

As mentioned, the dissolvement based GP solver was compared to a state-of-the-art solver, more specifically an implementation of a CSP based OPP-2 algorithm [PS02]. Figure 5.8 on the following page and 5.9 on page 73 shows the results obtained. A few instances where solved fastest by the dissolvement based solver but in general it was outperformed by the CSP solver. As seen, the timeout ratio for the latter is low compared to the former. In section 3.2.2 on page 33 we discussed pros and cons of purely utilizing the dissolvement technique for determining the packing topology. It seems that the redundancy induced by this technique has a bad influence on the total performance. Section 5.4 on page 75 investigates the performance in more details from a retrospective point-of-view and suggest how to improve it in various ways.

The SC solver

Consider the graphs in the right column of figure 5.5 on page 67 showing execution data for the SC solver for $d = 2$. Despite that each iteration in the SC solver is faster compared to the GP solver[1], the former generally performs worse than the latter: Compared to the GP solver, high timeout ratios are achived for a smaller volume percentage $\frac{\mathcal{V}_b}{\mathcal{V}_c}$ and divergence occur for smaller $|V|$. Section 4.2 suggests that the increased redundancy in the representation of clique configurations might increase the total execution time, despite the low complexity of each iteration. This seems to be the case.

Figure 5.10 on page 74 shows computational results for the SC solver for $d = 3$ and $d = 4$. Except for the worse performance, compared to the GP solver, the two solvers share the same characteristics: The timeout ratio increases with $\frac{\mathcal{V}_b}{\mathcal{V}_c}$ and $|V|$. Also, the influence of the volume percentage decrease for larger d. As for the GP solver the curve showing the the timeout ratio is steeper for the higher-dimensional instances than for $d = 2$.

[1] 1100 nodes/s for the GP solver versus 5200 nodes/s for the SC solver as mentioned earlier

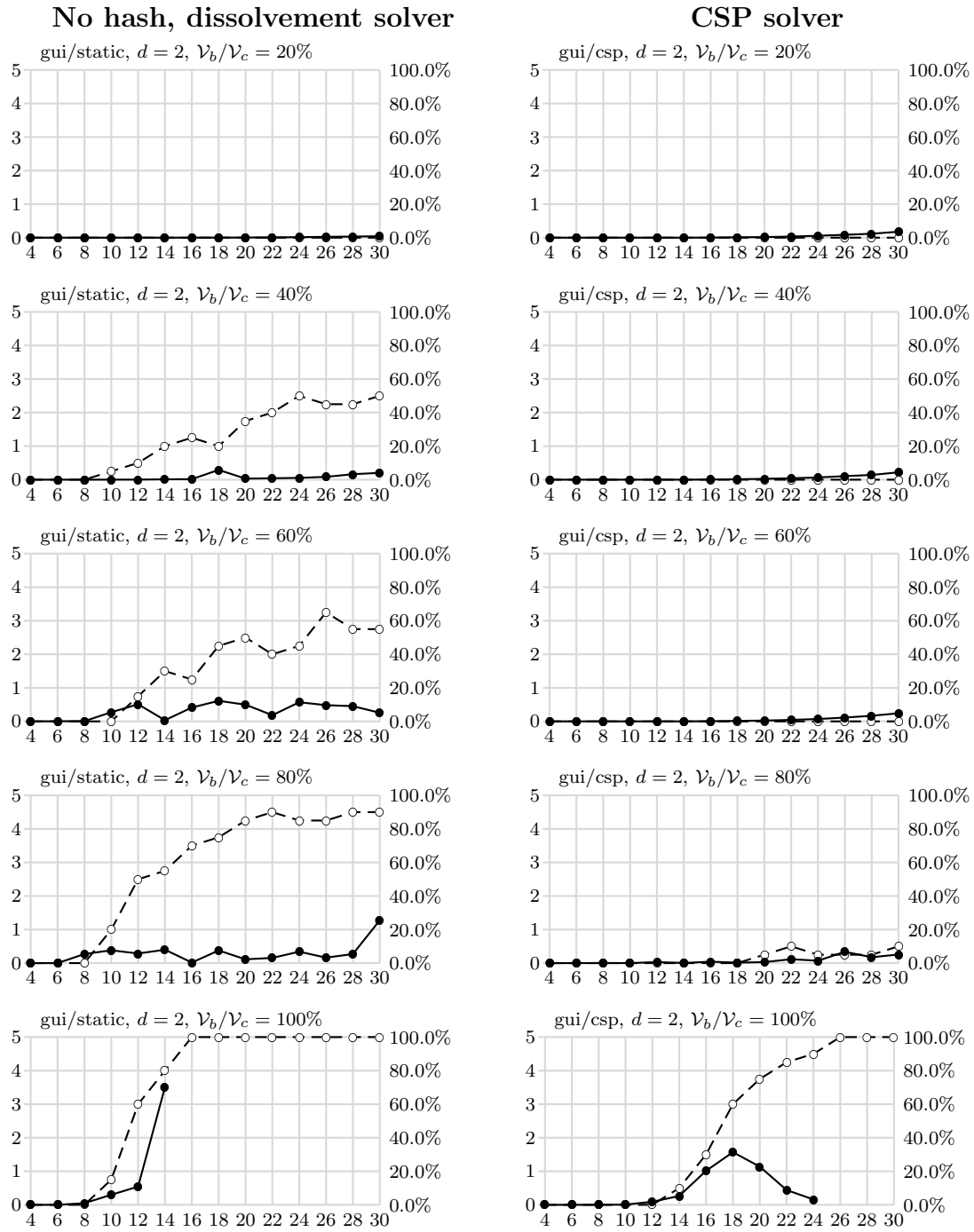
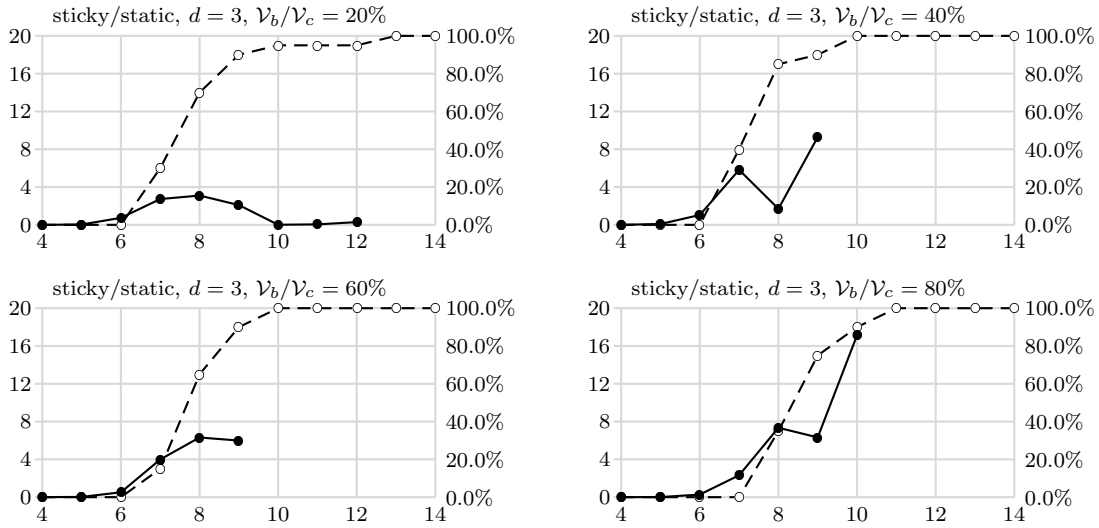


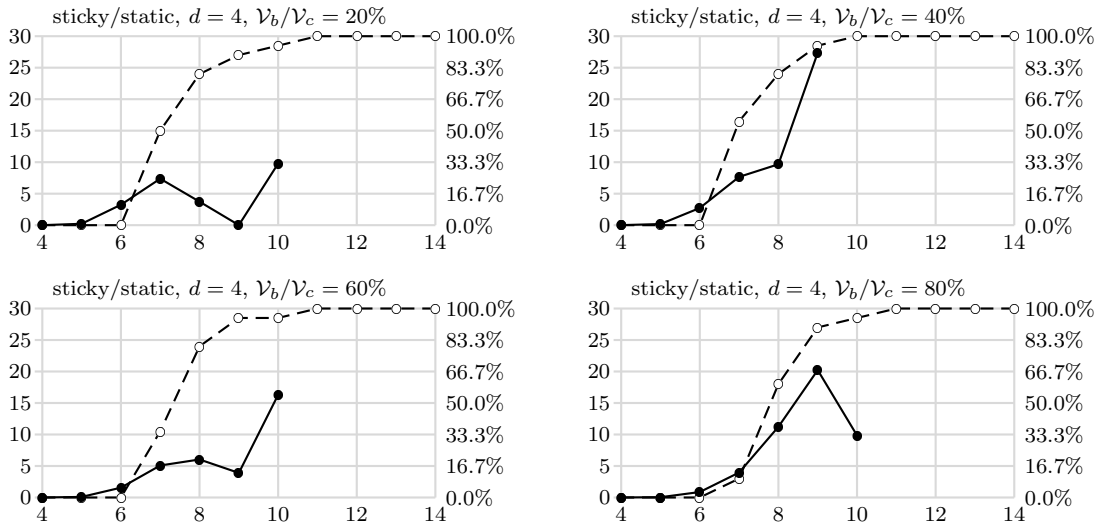
Figure 5.8: A comparison beween the dissolvement based and the CSP based solver for OPP-2.

		Diss. solver		CSP solver	
$\frac{\mathcal{V}_b}{\mathcal{V}_c}$	$\lvert V\rvert$	Timeouts	Sec.	Timeouts	Sec.
20	6	0.0%	0.000	0.0%	0.000
20	12	0.0%	0.000	0.0%	0.000
20	18	0.0%	**0.002**	0.0%	0.012
20	24	0.0%	**0.015**	0.0%	0.057
20	30	0.0%	**0.043**	0.0%	0.178
40	6	0.0%	0.000	0.0%	0.000
40	12	10.0%	0.002	0.0%	**0.000**
40	18	20.0%	0.291	0.0%	**0.013**
40	24	50.0%	**0.054**	0.0%	0.065
40	30	50.0%	**0.203**	0.0%	0.219
60	6	0.0%	0.004	0.0%	**0.000**
60	12	15.0%	0.510	0.0%	**0.000**
60	18	45.0%	0.613	0.0%	**0.015**
60	24	45.0%	0.570	0.0%	**0.071**
60	30	55.0%	0.257	0.0%	**0.242**
80	6	0.0%	0.000	0.0%	0.000
80	12	50.0%	0.286	0.0%	**0.030**
80	18	75.0%	0.376	0.0%	**0.014**
80	24	85.0%	0.340	5.0%	**0.071**
80	30	90.0%	1.275	10.0%	**0.256**
100	6	0.0%	0.000	0.0%	0.000
100	12	60.0%	0.544	0.0%	**0.091**
100	18	100.0%	–	60.0%	**1.569**
100	24	100.0%	–	90.0%	**0.160**
100	30	100.0%	–	100.0%	–

Figure 5.9: A comparison beween the dissolvement based and the CSP based solver for OPP-2.



(a) Results for $d = 3$. Timeout threshold: 20 seconds.



(b) Results for $d = 4$. Timeout threshold: 30 seconds.

Figure 5.10: Execution data for the sticky solver for $d = 3$ and $d = 4$.

5.4 Investigating the performance

In order to investigate the bottlenecks of both solvers, we examine purely algorithmic and purely implementational aspects.

5.4.1 Algorithmic considerations

In this section we describe some ideas on how to improve the performance by making purely algorithmic modifications.

Modifying the model

Slacking the requirement for P1 as described in section 3.2.3 removes an overhead of finding and preventing the existence of certain holes. This speeds up each iteration but also increases the search space. If the time reduction gained from dropping the requirement for P1 is exceeded by the time used to search the extended search space, then the P1 relaxation clearly does not pay off. Depending on the answer to open problem 3.12 on page 39 we might be able to gain some performance by reintroducing P1.

As noticed in section 3.2.2 the framework based on recursive dissolvement might introduce so much redundancy that the time saved on finding maximal infeasible cliques is surpassed by the extra time needed to search the larger search space. Depending on the answer to open problem 3.11 on page 38 it might pay off to reintroduce the original M_C measuring and just do the dissolvement as a feasibility check for guillotine packings. Reintroducing the original measuring model, of course requires P1 to be satisfied.

Using bounds

Currently both algorithms traverse huge parts of the search space. Using bounds to deduce if a node will ever become feasible and drop the node if not, is a way to improve performance. As described in section 4.2.1 on page 56 a simple volume estimation can be done for sticky cuttings: A bounding box smaller than the container is found for each clique and the box volume of the clique was used to deduce if the clique could ever be made feasible. This was possible due to the nice behavior of sticky cutting augmentations. It was attempted to find a similar estimation for guillotine packings but that never succeeded.

Another way of improving the bounding is to scale all boxes with conservative scales before the algorithm starts. This scaling technique will be

described in section 6.2. By performing such a scaling, cliques might render infeasible earlier in the augmentation process and hereby we might be able to drop search nodes earlier.

Changing search strategy

Currently a depth-first search strategy is used. Changing this to a best-first would require some classification method of the quality of each search node. Such classification is not trivial.

5.4.2 Implementational considerations

In order to give a more complete picture of the performance the very implementation was analyzed to uncover if this could have any significant influence on efficiency of the solvers. Both the GP- and SC solver was therefore analyzed with the profiling tool gprof. The tool gprof can be used to create a call graph of the running code with information on execution time and number of calls for each function.

Consider the call graph for the guillotine solver shown in figure 5.11. As

```
index % time    self  children    called     name
[1]     99.7    0.00  756.50                main
[2]     99.7    0.00  756.32           1    Solver::solve(int)
[3]     99.7    0.12  756.19           1    Solver::_solve(int)
[4]     52.3    0.18  396.29     2178285    __gnu_cxx::hash_set<...>::~hash_set()
[5]     52.2    0.64  395.66     2178285    __gnu_cxx::hashtable<...>::~hashtable()
[6]     51.5    0.34  390.69      687392    __gnu_cxx::hash_map<...>::operator[]
[7]     50.6   46.74  337.51     2190069    __gnu_cxx::hashtable<...>::clear()
[8]     34.8    0.13  264.27       11757    Searchnode::update_searchinfo()
...
[14]    28.3    0.18  214.67       11498    Searchnode::packingclass_test()
...
[15]    27.2   64.32  142.42   847243415    std::vector<...>::operator[]
...
```

Figure 5.11: A table showing the gprof output for the guillotine solver. The original gprof output shows also both callers and callees for each function but these informations have been removed for simplicity. The *index* column holds the number n for the n'th most time consuming function. *% time* holds the ratio between the execution time for the current function and the total execution time. This column clearly does not summarize to 100% as the functions may call each other. *self* and *children* shows the cumulative number of seconds spend in the current function body and in its children. *called* is the number of times the current function is called and *name* identifies the function.

seen, the total execution time is 756.50 seconds. Out of these, 52.3% of the time was inside the `stl::hash_set` destructor and 51.5% inside the `stl::hash_map` index operator (clearly, these two calls must share some sub routines as the sum is greater than 100%). The index operator for `stl::hash_map` is costing 0.5683 milliseconds per call in contrast to the 0.0002 milliseconds per call for the `std::vector`. Even though the `stl::hash_map` is known to be more complex than the `std::vector`, this is surprisingly expensive as the index operator for the hash has constant time complexity. As seen, 27.2% of the time is used inside the index operator for `std::vector`. Even though this is a relatively heavy post, the large number of calls makes each fetch relatively cheap.

Profiling data for the sticky cutting solver can be seen in figure 5.12. The structure of the call graph is slightly different from the call graph for

```
index % time    self  children  called  name
[1]     99.7    0.00    186.90          main
[2]     99.7    0.00    186.86       1  Stickysolver::solve(int)
[3]     99.6    0.01    186.83       1  Stickysolver::_solve(int)
[4]     50.5    0.05     94.66  521092  __gnu_cxx::hash_set<>::~hash_set()
[5]     50.5    0.10     94.55  521092  __gnu_cxx::hashtable<>::~hashtable()
[6]     48.8   11.61     79.97  521106  __gnu_cxx::hashtable<>::clear()
[7]     40.7    0.06     76.30  133199  __gnu_cxx::hash_map<...>::operator[]
[8]     31.7    0.04     59.34  369876  __gnu_cxx::hash_set<>::hash_set()
.. .
[11]    31.2    0.07     58.35    3966  Stickynode::packingclass_test()
...
[16]    19.8    0.01     37.20    3966  Stickynode::update_searchinfo()
...
```

Figure 5.12: gprof output for the sticky cutting solver

the guillotine solve but the same symptoms appear. Much time is spend on the `stl::hash_set` destructor and on the indexing operator for the `stl::hash_map`.

Both implementations rely heavily on mappings from a simple integer identifier to a more complex data type. Disregarding the bad performance of the hashing based data structures, these were ideal to the job. It would require a total reimplementation (and thus be a relatively time consuming task) to totally drop the currently used data structure paradigm, so instead only a subset of the data types was replaced for testing purposes. The goal was to verify that some performance could be gained by moving away from the hashing based STL structures. The replacement was a statically sized container using a set of `std::vector`s internally in order to provide constant time lookup, insertion and deletion at random positions and fast traversal of all identifiers contained. The constant size ($|V|$) is of course a drawback when used in subproblems for a small subset of V.

In the previous section we have already seen that the static data types performed better for our usage than the hashing based. That is, an increase in performance by a constant size, $O(1)$. Figure 5.13 shows some of the gprof analysis for the guillotine solver using these new static data structures (simply named `Set` and `Map`). It appears that the load has moved from

```
index % time  self  children  called  name
[1]     99.8  0.00  148.95            main
[2]     99.8  0.00  148.84         1  Solver::solve(int)
[3]     99.8  0.06  148.77         1  Solver::_solve(int)
[4]     48.0  0.53   71.06   4621478  std::_Bit_iterator std::copy<...>(...)
...
[8]     43.2  0.79   63.59   2762067  Set::Set(Set const&)
...
[10]    34.7  0.00   51.83     16728  Searchnode::Searchnode()
...
[25]    24.3  0.10   36.10     11498  Searchnode::packingclass_test()
...
[27]    21.4  0.01   31.86     11757  Searchnode::update_searchinfo()
...
```

Figure 5.13: A table showing the gprof output for the guillotine solver using static data structures.

destructors and index operators to copying of the new data structures. The same tendency is seen in the gprof output for the optimized sticky cutting solver (figure 5.14). Most of the time is now used by creating and copying search nodes.

```
index % time  self  children  called   name
[1]   99.9    0.00  30.46             main
[2]   99.8    0.00  30.44          1  Stickysolver::solve(int)
[3]   99.8    0.01  30.43          1  Stickysolver::_solve(int)
[4]   27.7    0.00   8.45       6616  Stickynode::Stickynode(Stickynode const&)
[5]   23.8    0.05   7.21     312756  Set::Set(Set const&)
[6]   22.7    0.02   6.90       3302  Stickynode::operator=(Stickynode const&)
...
[8]   21.0    0.06   6.33     409154  std::_Bit_iterator std::copy<...>()
...
[13]  19.7    0.06   5.95       3966  Stickynode::packingclass_test()
...
[46]  10.5    0.00   3.20       3966  Stickynode::update_searchinfo()
...
```

Figure 5.14: A table showing the gprof output for the sticky cutting solver using static data structures.

5.4.3 Remarks

As discussed, both algorithms could benefit from a bit refinement. We proved that a change of data structures had a positive influence on the performance.

A more drastic reimplementation with a complete rethinking of the use of data structures would probably speed up the solvers even more. One option to consider would be to use some existing libraries for representing graphs. Both LEDA and BOOST are examples of such libraries and the large amount of effort put into their optimization might come in handy when used in the solvers.

But the biggest performance gain would probably come from purely algorithmic optimizations. As discussed in section 5.4.1 there are still several untried experiments to be done:

- Using bounds and scaling boxes by conservative scales.
- Using a best-first traversal for search nodes.
- For the guillotine solver: Reintroducing P1 and
- Reverting to the original M_C measuring framework and only use the dissolvement technique when checking for and ensuring the guillotine property.

Chapter 6
Lower bounds

Several papers have been published about lower bounds for the unrestricted two- and three-dimensional bin packing problem. In contrast only a very limited amount of work has been published for lower bounds satisfying various restrictions, e.g. for packings satisfying the guillotine cuttable property. Clearly, a lower bound for the unrestricted case is also a feasible lower bound for the restricted one – however, it is likely not as tight. The simplest non-trivial lower bound for OBPP is probably the continuous lower bound:

Definition 6.1 (Continuous lower bound L_0) *Let $\mathcal{V}_v$ denote the volume of a box $v \in V$ and let $\mathcal{V}_c$ denote the container volume. Then*

$$L_0 = \left\lceil \frac{\sum_{v \in V} \mathcal{V}_v}{\mathcal{V}_c} \right\rceil$$

is a valid lower bound. L_0 is also referred to as the continuous lower bound.

This bound can be thought of as the number of containers needed to contain all boxes if they were melted into some liquid substance. Now let $\mathscr{L}$ be a lower bound and let

$$\rho(\mathscr{L}) = \inf_{I \text{ instance}} \left(\frac{\mathscr{L}(I)}{\mathrm{OPT}(I)} \right)$$

denote the *worst case performance* of $\mathscr{L}(I)$ for all instances I.

It is known that $\rho(L_0) = \frac{1}{2}$ for OBPP-1 [MT90]. For OBPP-2 we have $\rho(L_0) = \frac{1}{4}$ [MV98] and for OBPP-3 we have $\rho(L_0) = \frac{1}{8}$ [MPV97]. The general result $\rho(L_0) = \frac{1}{2^d}$ has been proved (asymptotically) by Mauro Dell'Amico in [Del98].

In this chapter we give a short survey of various more tight lower bounds for the unrestricted OBPP. Three types of bounds will be described. Section

6.1 gives some examples of bounds specifically constructed for a specific d by assembling volume considerations of different box groupings. In section 6.2 a more general method for creating bounds for higher-dimensional problems by scaling the boxes with *conservative scales* is described. Section 6.3 covers a more detailed description of an algorithm by Pisinger and Sigurd which can be used to calculate a lower bound for restricted bin packing problems.

6.1 Assembled bounds for $d \in \{1, 2, 3\}$

A typical way of constructing lower bounds is to divide the boxes in classes depending on their size and make various estimations on the volume consumption of each class. The lower bounds obtained hereby may then be assembled into new lower bounds. This section shows some examples of this technique for $d \in \{1, 2, 3\}$.

Example for $d = 1$

Consider the one-dimensional problem with n boxes of size c_i, $i \in B = \{1, \ldots, n\}$ and containers of capacity C. Figure 6.1 depicts the sets S_s, S_m, S_l defined by $S_s = \{i \in B \mid p < c_i \leq \frac{1}{2}C\}$, $S_m = \{i \in B \mid \frac{1}{2}C < c_i \leq C - p\}$ and $S_l = \{i \in B \mid c_i > C - p\}$ where $1 \leq p \leq \frac{1}{2}C$. No two items from $S_m \cup S_l$

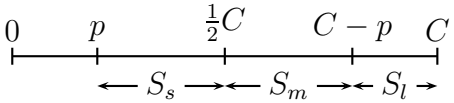


Figure 6.1: Intervals used in L_α and L_β

can be placed in the same container and these items will therefore generally be referred to as *large* items.

L_α defined below is a lower bound for the OBPP-1. It was originally presented in [MT90].

$$L_\alpha = \max_{1 \leq p \leq \frac{1}{2}C} \left\lceil |S_m \cup S_l| + \max\left(0, \left\lceil \frac{\sum_{i \in S_s \cup S_m} c_i}{C} - |S_m| \right\rceil\right) \right\rceil$$

L_α can be thought of as "the number of large items" plus eventually "a volume estimate" where the latter is "the number of containers used to contain the volume of all medium sized items for which there might be two or more

per container" minus "the number of items from S_m already included" – calculated for various definitions of *large*. A similar lower bound by [DM95] is L_β defined as

$$L_\beta = \max_{1 \le p \le \frac{1}{2}C} \left\lceil |S_m \cup S_l| + \max\left(0, \left\lceil \frac{|S_s| - \sum_{i \in S_m} \left\lfloor \frac{C - c_i}{p} \right\rfloor}{\left\lfloor \frac{C}{p} \right\rfloor} \right\rceil\right) \right\rceil$$

In 1998 Martello and Vigo [MV98] propose a, somewhat obvious, lower bound L_1 that combines L_α and L_β:

$$L_1 = \max(L_\alpha, L_\beta)$$

L_1 can be computed in $O(|B|^2)$ time if the boxes are sorted according to decreasing size.

Example for $d = 2$

In [BM03] the method of combining existing bounds is used in order to create a bound for $d = 2$. Let the size of each two-dimensional box be $(w_i, h_i), i \in B$ and let the container size be (W, H). Similar to before we define two sets K_1, K_2 by grouping boxes of certain size:

$$K_1 = \{i \in B \mid p \le w_i \le W - p\}$$
$$K_1 = \{i \in B \mid w_i > W - p\}$$

where $1 \le p \le \frac{1}{2}W$. The intervals are illustrated in figure 6.2. No box

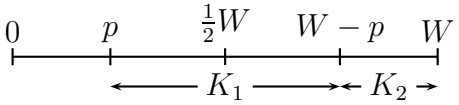


Figure 6.2: Intervals used in L_2

from K_2 can be packed side by side with a box from $K_1 \cup K_2$. Therefore the occupied area of the boxes in K_2 are at least Wh_i. Now, construct a OBPP-1 instance with container capacity WH and n items of size

$$c_i = \begin{cases} w_i h_i & i \in K_1 \\ W h_i & i \in K_2 \end{cases}$$

Let $z_w(p)$ denote the solution for this OBPP-1 instance for some p and let $L_2^W = \max\{z_w(p) \mid 1 \le p \le \frac{1}{2}W\}$. Then L_2^W is a valid lower bound for

OBPP-2. Similarly, by exchanging widths with heights in the definition of L_2^W we can obtain the bound L_2^H. A feasible lower bound dominating L_α and L_β is $L_2 = \max(L_2^W, L_2^H)$.

Example for $d = 3$

Analogous constructions have been made for $d = 3$. Let the container size for the three-dimensional case be (W, H, D) and let the n box sizes be (w_i, h_i, d_i) for $i \in B$. Define by

$$J^{WH} = \{i \in B \mid w_i > \tfrac{1}{2}W \text{ and } h_i > \tfrac{1}{2}H\}$$

the set of all boxes having large width and height and by

$$\begin{aligned} J_l(p) &= \{i \in J^{WH} \mid \tfrac{1}{2}D < d_i \leq D - p\} \\ J_s(p) &= \{i \in J^{WH} \mid p \leq d_i \leq \tfrac{1}{2}D\} \end{aligned}$$

the sets of all boxes in J^{WH} having large and small depth respectively for $1 \leq p \leq \frac{1}{2}D$. Very similar to the idea of L_α and L_β we can now define

$$\begin{aligned} L_3^{WH} = {} & |\{i \in B \mid d_i > \tfrac{1}{2}D\}| \\ & + \max_{1 \leq p \leq \frac{1}{2}D} \left(\left\lceil \frac{\sum_{i \in J_s(p)} d_i - \left(|J_l(p)|D - \sum_{i \in J_l(p)} d_i \right)}{D} \right\rceil , \right. \\ & \left. \left\lceil \frac{|J_s(p)| - \sum_{i \in J_l(p)} \left\lfloor \frac{D - d_i}{p} \right\rfloor}{\left\lfloor \frac{D}{p} \right\rfloor} \right\rceil \right) \end{aligned}$$

In [MPV97] it is shown that L_3^{WH} is a valid lower bound for OBPP-3. By interchanging the depth, width and height variables the lower bounds L_3^{WD} and L_3^{HD} can be obtained. A lower bound dominating all three can be constructed as $L_3 = \max(L_3^{WH}, L_3^{WD}, L_3^{HD})$.

Notice that although $L_3 > L_0$ for some instances it is not dominating L_0: Consider the instance of $n \in 8\mathbb{N}$ boxes of size $(\frac{1}{2}W, \frac{1}{2}H, \frac{1}{2}D)$ where $W, H, D \in 2\mathbb{N}$. As $L_3^{WH} = L_3^{WD} = L_3^{HD} = \emptyset$ we have $L_3 = 0$ while $L_0 = \frac{1}{8}n$. That is, the worst case performance of L_3 is arbitrarily bad.

6.2 Scaling based bounds

The bounds presented in section 6.1 were all constructed to a specific value of d. Consider the d-dimensional bin packing problem and reintroduce the

notation from the earlier chapters with a size function $w_i, i \in \{1, \ldots, d\}$ for each box $v \in V$ and a container of size $(W_1, \ldots, W_d)$. Also let (V, w) be a shorthand notation for the instance (V, w, W) and $\otimes w(v) = \prod_{1 \leq i \leq d} w_i(v)$ denote the volume of box $v \in V$ given the size function w. A trivial necessary condition required in order for a packing to exist for an OPP-d instance is that the total box volume does not exceed the container volume:

Definition 6.2 (Volume criterion) *For an OPP-d instance (V, w) with container size W the volume criterion is satisfied if*

$$\sum_{v \in V} \otimes w(v) \leq \prod_{1 \leq i \leq d} W_i$$

By scaling box sizes appropriately the volume criterion may become an efficient tool to declare an instance infeasible. In the following we will explore the idea of such a scaling.

Lueker [LUE83] introduces the term *dual feasible function*.

Definition 6.3 (Dual feasible function) *A map $u : [0, 1] \to [0, 1]$ is a* dual feasible function *if for all finite non-negative sets S we have*

$$\sum_{v \in S} v \leq 1 \Rightarrow \sum_{v \in S} u(v) \leq 1$$

[FS01] describes how to obtain lower bounds directly from dual feasible functions.

The functions have a practical property: Lower bounds for a transformed instance $u(I)$ is valid for the non-transformed I too:

Theorem 6.4 *Let $\mathscr{L}_{u(I)}$ be a lower bound for the transformed instance $u(I)$ where u is a dual feasible function. Then $\mathscr{L}_{u(I)}$ is also a lower bound for I.*

Proof Assume the opposite, that $\mathscr{L}_{u(I)}$ is not a valid lower bound for I. That is, $\mathrm{OPT}(I) \leq \mathscr{L}_{u(I)} - 1$ or equivalent: There exist some set S that is contained in a single bin in the optimal solution $\mathrm{OPT}(I)$ but will use more than one bin in the transformed solution $\mathrm{OPT}(u(I))$ as $\mathscr{L}_{u(I)} \leq \mathrm{OPT}(u(I))$. This implies the existence of a set, S, for which $\sum_{v \in S} v \leq 1 \not\Rightarrow \sum_{v \in S} u(v) \leq 1$ which contradicts that u is a dual feasible function. □

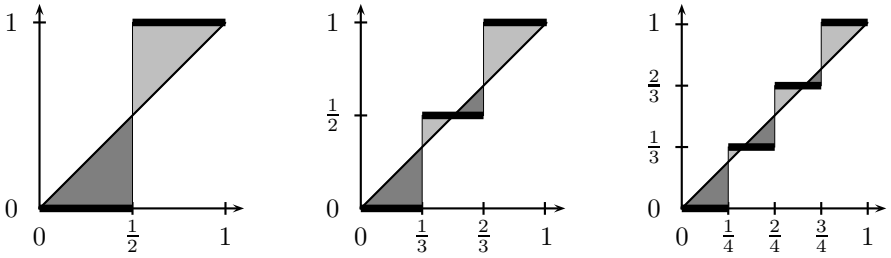
Areas of a dual feasible function that increase (decrease) the argument value is denoted *win zones* (*loose zones*). For a given instance I the goal is to

construct a dual feasible function having as many boxes in the win zones as possible.

Example 6.5 The function

$$u^{(k)} : [0,1] \to [0,1]$$
$$x \mapsto \begin{cases} x & x(k+1) \in \mathbb{Z} \\ \frac{1}{k}\lfloor x(k+1) \rfloor & \text{otherwise} \end{cases}$$

is a dual feasible function (proof is omitted here). Figure 6.3 illustrates $u^{(k)}$. ◇



(a) $k = 1$ (b) $k = 2$ (c) $k = 3$

Figure 6.3: Example of $u^{(k)}$ for $k \in \{1, 2, 3\}$. The win zones are seen above the $y = x$ line (the light grey areas).

Recall the packing class properties for a set of graphs $G_1, \ldots, G_d$. As required by P2 all stable sets in G_i must be i-feasible. We aim to establish a necessary condition, based on box scalings, for P2 to be satisfied. For w_i let $\mathcal{F}(V, w_i)$ denote all subsets of $S \subseteq V$ being i-feasible.

$$\mathcal{F}(V, w_i) = \{S \subseteq V \mid w_i(S) \leq W_i\}$$

where $w_i(S) = \sum_{v \in S} w_i(v)$. Notice that S might not be stable. This requirement will be added later.

The following definition is essential throughout this section:

Definition 6.6 (Conservative scale) *Let (V, w) be an OPP-d instance. A function w' satisfying*

$$\mathcal{F}(V, w_i) \subseteq \mathcal{F}(V, w'_i) \qquad , \forall i \in \{1, \ldots, d\} \tag{6.1}$$

is called a conservative scale *for (V, w).*

A conservative scale can always be constructed by using dual feasible functions:

Lemma 6.7 *Let $u_1, \ldots, u_d$ be dual feasible functions and (V, w) an OPP-d instance. Then $w' = (u_1 \circ w_1, \ldots, u_d \circ w_d)$ is a conservative scale for (V, w).*

With the definition of conservative scales in hand a formal criteria for a transformed instance (V, w') to contain a packing class can be given:

Lemma 6.8 *Let (V, w) be an OPP-d instance and w' a conservative scale. Any packing class for (V, w) is also a packing class for (V, w').*

The lemma is almost trivially true since every stable set in the left hand side of (6.1) is also contained in the right hand side. Lemma 6.8 implies that conservative scales can be used to determine existence of a packing class for a non-transformed instance I:

Theorem 6.9 *Let (V, w) be an OPP-d instance and w' a conservative scale. Then*

$$\sum_{v \in V} \otimes w'(v) \leq \prod_{1 \leq i \leq d} W_i$$

is a necessary condition for the existence of a packing class for (V, w).

The following example brings it all together.

Example 6.10 Let I be an OPP-3 instance with a unit cube container and 9 boxes of size $(\frac{3}{7}, \frac{3}{7}, \frac{3}{7})$. Each box has a volume of $(\frac{3}{7})^3 = \frac{27}{343}$ and since $9 \cdot \frac{27}{343} = \frac{243}{343} < 1$ the volume criterion cannot be used to declare the instance infeasible.

Instead use the dual feasible function $u^{(2)}$ defined in example 6.5 on the facing page and apply lemma 6.7 to obtain a conservative scale w' for the instance. With w' the box sizes are $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ which gives each box a volume of $\frac{1}{8}$. Since $9 \cdot \frac{1}{8} = \frac{9}{8} > 1$ theorem 6.9 tells that no packing class exist for I. $\diamond$

If an augmentation based algorithm like the one described in chapter 2 is used to find a packing class each step provides a tuple of d edge sets $\mathcal{E}_{+,1}, \ldots, \mathcal{E}_{+,d}$ that can be thought of as a partially build packing class. For each packing class $E = (E_1, \ldots, E_d)$ found in a successor search node we have

$$\mathcal{E}_{+,i} \subseteq E_i \qquad , \forall i \in \{1, \ldots, d\} \tag{6.2}$$

The P2 condition concerns only stable sets so it is sufficient to consider those $S \in \mathcal{F}(V, w_i)$ for which S is a stable set in $\mathcal{E}_{+,i}$, i.e. the sets $S \in \mathcal{F}(V, w_i)$ for which $\mathcal{E}_{+,i}[S] = \emptyset$. Let

$$\mathcal{F}(V, w, \mathcal{E}_{+,i}) = \{S \in \mathcal{F}(V, w) \mid S \text{ is stable in } \mathcal{E}_{+,i}\}$$

A conservative scale for $\mathcal{F}(V, w, \mathcal{E}_{+,i})$ can then be defined similar to definition 6.6:

Definition 6.11 (Conservative scale for $(V, w, \mathcal{E}_{+,i})$)
Let (V, w) and (V, w') be OPP-d instances and $\mathcal{E}_+ = (\mathcal{E}_{+,1}, \ldots, \mathcal{E}_{+,d})$ a tuple of edge sets for which (6.2) is satisfied for a packing class E for (V, w). Then w' is a conservative scale *for $(V, w_i, \mathcal{E}_+)$ if*

$$\mathcal{F}(V, w, \mathcal{E}_{+,i}) \subseteq \mathcal{F}(V, w') \qquad , \forall i \in \{1, \ldots, d\}$$

A stable-set-version of lemma 6.8 can now be formulated:

Lemma 6.12 *Let (V, w) be an OPP-d instance, $\mathcal{E}_+$ a tuple of edge sets satisfying (6.2) and for a packing class E for (V, w). If w' is a conservative scale for $(V, w, \mathcal{E}_+)$ then E is also a packing class for (V, w').*

Also a stable-set-version of theorem 6.9 is valid:

Theorem 6.13 *Let w' be a conservative scale for $(V, w, \mathcal{E}_+)$. Then*

$$\sum_{v \in V} \otimes w'(v) \leq \prod_{1 \leq i \leq d} W_i \tag{6.3}$$

is a necessary condition for the existence of a packing class for (V, w).

Theorem 6.13 is exactly what we searched for. By using conservative scales to enlarge the boxes we can perform a cheap check of (6.3) in order to discard a search node earlier.

A lower bound for an OBPP-d instance (V, w) can be achieved by using L_0 on various transformations (V, w). We define L_4 as the maximum of all L_0 bounds for a set of transformations $\mathcal{W}$.

Theorem 6.14 *Let (V, w) be an OBPP-d instance and $\mathcal{W}$ a set of conservative scale for (V, w). Then*

$$L_4(V, w) = \max_{w' \in \mathcal{W}} \left\lceil \frac{\otimes w'(V)}{\prod_{1 \leq i \leq d} W_i} \right\rceil \tag{6.4}$$

is a valid lower bound for (V, w).

If the conservative scales w' are constructed using dual feasible functions the bound in (6.4) can be calculated in $O(|V|d)$ time. Fekete and Schepers [FS97b] show that this method of constructing lower bounds is a generalization of the bounds in [MPV97] and [MV98].

6.3 A bound for guillotine packings

In this section we describe an algorithm presented in [PS02] which is capable to compute a lower bound for the guillotine cuttable OBPP-2. With a slight modification the algorithm can be extended to handle arbitrary d. The main strategy for the algorithm is as follows:

1. The LP-relaxation of a conventional formulation of OBPP-2 as a mixed integer program (MIP) leads to a quite weak lower bound and is therefore hard to solve for ordinary MIP solvers. Instead Dantzig-Wolfe decomposition [DW60, Wol98] is used to give a more tight formulation.

2. In the decomposition the *restricted master problem* (RMP) becomes a setcover problem in which each *set* represents a packing of a single container. This problem is solved through delayed column generation in order to find a bound for the LP-relaxation faster. The *pricing problem* becomes a two-dimensional knapsack problem in which the task is to select a subset of boxes that fits in a knapsack of container size and results in a packing with maximum absolute reduced cost.

   This two-dimensional knapsack problem is then split into a one-dimensional multi-constrained knapsack problem in which a subset of boxes is chosen and a two-dimensional packing decision problem in which it is determined if the chosen subset can be packed in a single container. Additional constraints on the packings, e.g. the guillotine property are handled in the solver for the latter problem.

The RMP is taking care of distributing boxes among the containers so that all boxes are used and it is initially fed with a single feasible packing obtained by a heuristic. Other packings are added through the subproblems (the pricing problems): In each subproblem the task is to find a feasible packing with smallest *reduced cost* and add it to the RMP. The process stops when all subproblems provides packings that are already in RMP.

The lower bound is a side effect of the LP-relaxation of the master problem.

Below we describe each step in further details.

6.3.1 The master problem

As mentioned above, the algorithm uses a set cover formulation of the bin packing problem as RMP. Applying Dantzig-Wolfe decomposition alleviates an inherent problem with the OBPP-2 MIP formulation. As we shall see below, the number of constraints in such a model is very large and the model thus immensely difficult to solve.

Let $B = \{1, \ldots, n\}$ be the set of boxes and let (W, H) denote the container dimensions. Also let the binary variable $l_{ij} = 1$ if and only if box i is located to the left of j. Similarly let $b_{ij} = 1$ if and only if i is below j. Box i is placed at bin $m_i \in \mathbb{N}$ and for $p_{ij} \in \{0, 1\}$ we set $p_{ij} = 1$ if and only if $m_i < m_j$. First of all we must make sure that for every two boxes i and j, either these are beside or above each other or in two different containers. That is,

$$l_{ij} + l_{ji} + b_{ij} + b_{ji} + p_{ij} + p_{ji} \geq 1, \qquad i, j \in B, i < j$$

Let (x_i, y_i) denote the coordinate of the lower left corner of box i with size (w_i, h_i). If two boxes i and j are in the same container, then in order to ensure that they are not overlapping we must have

$$\begin{aligned} l_{ij} = 1 &\Rightarrow x_i + w_i \leq x_j \\ b_{ij} = 1 &\Rightarrow y_i + h_i \leq y_j \end{aligned} \tag{6.5}$$

By reformulating these equations to

$$\begin{aligned} x_i - x_j + W l_{ij} &\leq W - w_i \\ y_i - y_j + H b_{ij} &\leq H - h_i \end{aligned}$$

we obtain two statements that are trivially true for $l_{ij} = 0$ or $b_{ij} = 0$ but are equivalent to the inequalities in (6.5) when $l_{ij} = 1$ or $b_{ij} = 1$.

Also we must ensure that $0 \leq x_i \leq W - w_i$ and $0 \leq y_i \leq H - h_i$.

The requirement $p_{ij} = 1 \Rightarrow m_i < m_j$ is equivalent to

$$p_{ij} = 1 \Rightarrow m_i + 1 \leq m_j$$

which can be reformulated to

$$m_i - m_j + n p_{ij} \leq n - 1$$

in order to obtain a statement that are trivially true for $p_{ij} = 0$ but are equivalent to the above inequality for $p_{ij} = 1$.

If c is the number of containers used the above constraints can be assembled to the following model:

$$
\begin{array}{lll}
\min & c & \\
\text{s.t.} & l_{ij} + l_{ji} + b_{ij} + b_{ji} + p_{ij} + p_{ji} \geq 1 & i,j \in B, i < j \\
& x_i - x_j + W l_{ij} \leq W - w_i & i,j \in B \\
& y_i - y_j + H b_{ij} \leq H - b_i & i,j \in B \\
& m_i - m_j + n p_{ij} \leq n - 1 & i,j \in B \\
& 0 \leq x_i \leq W - w_i & i \in B \\
& 0 \leq y_i \leq H - h_i & i \in B \\
& 1 \leq m_i \leq c & i \in B \\
& l_{ij}, b_{ij}, p_{ij} \in \{0,1\} & i,j \in B \\
& x_i, y_i \in \mathbb{R}_0^+ & i \in B \\
& m_i, c \in \mathbb{N}_0 & i \in B
\end{array}
\tag{6.6}
$$

The model has $3n^2$ binary variables, $2n$ continuous variables and $\frac{1}{2}n^2 + 3n^2 + 6n = \frac{7}{2}n^2 + 6n$ constraints.

Because of the large number of constraints in (6.6) and because of the large amount of symmetric solutions the model is difficult to solve for MIP solvers:

Figure 6.4 shows a typical structure of an LP-model in which there are a few rows of coupling constraints and (larger) number of independant subproblems. This applies to (6.6) too: The independent subproblems is ensuring

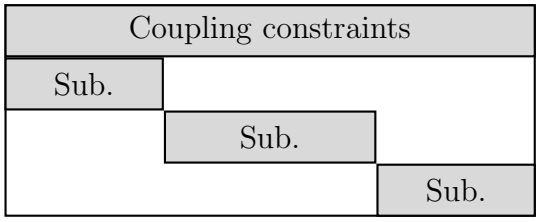


Figure 6.4: A typical structure of a linear program consisting of a lot of zeros (the white area), a number of independent subproblems (marked *Sub.* above) and eventually a set of rows that serves as *coupling constraints*. In Dantzig-Wolfe decomposition a *restricted master problem* (RMP) containing only the coupling constraints is created. The RMP is then gradually extended by solving the subproblems individually.

that a packing of a single container is feasible.

With use of Dantzig-Wolfe decomposition (6.6) is decomposed into a set cover problem as RMP and a number of subproblems as mentioned earlier.

So let $\mathscr{P}$ denote the set of all possible feasible packings of a single bin and let the binary constant $\delta_i^p = 1$ if and only if the packing $p \in \mathscr{P}$ contains the box i. Also let the binary variable $x_p = 1$ if and only if the packing p is used in the solution. In order to ensure that every rectangle in B is packed in at least one bin we must have $\sum_{p\in\mathscr{P}} x_p\delta_i^p \geq 1$ for all $i \in B$. The set cover formulation then looks as follows:

$$\begin{aligned} \min \quad & \sum_{p\in\mathscr{P}} x_p \\ \text{s.t.} \quad & \sum_{p\in\mathscr{P}} x_p\delta_i^p \geq 1 \quad i \in B \\ & x_p \in \{0,1\} \qquad p \in \mathscr{P} \end{aligned} \tag{6.7}$$

A solution for the relaxed (6.7) is also a solution in the relaxed (6.6) since it corresponds to some configuration of variables where $l_{ij}, b_{ij}, p_{ij} \in \{0,1\}$, $m_i \in \mathbb{N}_0$ but $c \in \mathbb{R}_0^+$. On the other hand, a solution for the relaxed (6.6) is not a solution for the relaxed (6.7) since a solution in which $l_{ij}, b_{ij}, p_{ij} \notin \{0,1\}$ cannot be represented in the latter. Hence the bound obtained by the relaxation of (6.7) dominates the bound obtained by the relaxation of (6.6).

In the initial RMP we only consider a small subset $\mathscr{P}' \subset \mathscr{P}$ of packings obtained from a heuristic solution. In subsequent iterations more packings are added. This technique is also known as *delayed column generation*. The RMP then looks as follows:

$$\begin{aligned} \min \quad & \sum_{p\in\mathscr{P}'} x_p \\ \text{s.t.} \quad & \sum_{p\in\mathscr{P}'} x_p\delta_i^p \geq 1 \quad i \in B \\ & x_p \in \mathbb{R}_0^+ \qquad p \in \mathscr{P}' \end{aligned} \tag{6.8}$$

The essential next step is the selection of packings to include into the RMP. Consider the LP-relaxed dual to the original set cover model (6.7):

$$\begin{aligned} \max \quad & \sum_{j\in B} y_j \\ \text{s.t.} \quad & \sum_{j\in B} y_j\delta_j^p \geq 1 \quad p \in \mathscr{P} \\ & y_j \in \mathbb{R}_0^+ \qquad j \in B \end{aligned} \tag{6.9}$$

The week duality theorem tells that if a set of dual variables y_j render the dual set cover (6.9) feasible and also the corresponding primal variables x_i render the relaxation of (6.7) feasible then x_i and y_j are optimal solutions to

the primary and dual program respectively. Otherwise, if the x_i are feasible in (6.7) but y_j render (6.9) infeasible then some set of constraints of the form $\sum_{j \in B} y_j \delta_j^p \geq 1$ must be violated in the latter for a subset of $Q \subseteq \mathscr{P}$. In the initial iterations of the algorithm where the cardinality of $\mathscr{P}'$ is still small each solution to RMP may result in a large number of violations in the corresponding dual (6.9). By adding the packings in Q to RMP (6.8) the solution to RMP in subsequent iterations will correspond to a feasible solution in the dual and hereby the process will terminate in an optimum as $\mathscr{P}$ is finite.

The drawback of adding all packings in Q to RMP is the resulting increased complexity of RMP. Therefore only the single packing inducing the *most violating* constraint in the dual is added in each iteration. Let

$$\gamma_p^y = 1 - \sum_{j \in B} y_j \delta_j^p$$

denote the amount of violation resulting from packing p and the dual variables y_j. γ_p^y is also referred to as the *reduced cost* of p. If $\gamma_p^y < 0$ the constraint has been violated so the most significant reduced cost is

$$\min_{p \in \mathscr{P} \setminus Q} \gamma_p^y.$$

Clearly, if $\gamma_p^y \geq 0$ for all p then no constraints are violated and x_i is thus an optimal solution.

The task of selecting a packing p with smallest reduced cost is referred to as the *pricing problem* which is roughly described in the following section.

6.3.2 The pricing problem

Finding the smallest reduced cost is equivalent to finding the packing with greatest corresponding y_j-sum, which is exactly the formulation of a two-dimensional knapsack problem in which the profit for each box is y_j.

The knapsack problem is NP-hard so a polynomial greedy heuristic is used in every iteration to find a feasible packing with low (not necessarily lowest) reduced cost and only if the heuristic fails the problem is solved with an exact algorithm. The exact algorithm is sketched below:

The MIP model of the knapsack problem has the same drawbacks as the MIP formulation of the original bin packing problem: A large number of variables and a generally bad formulation with a lot of symmetry. Thus the two-dimensional knapsack problem is again split into a one-dimensional

optimization problem selecting a subset of boxes with smallest reduced cost, and a two-dimensional packing decision problem that decides whether the chosen boxes fit into a container or not. Such decision algorithms based on Fekete and Schepers work have already been described in chapter 2, 3 and 4 for arbitrary d. In 6.3.3 yet another approach relying on constraint programming is described.

We will not cover the algorithm for solving the one-dimensional optimization problem in details, but will give a short outline below. Given the profits $\lambda_j = y_j$ the problem is to find a subset of boxes with highest profit so that the area of the chosen boxes does not exceed the container area. In other words, this is a 0-1 knapsack problem:

$$\begin{aligned} \max \quad & \sum_{i \in B} x_i \lambda_i \\ \text{s.t.} \quad & \sum_{i \in B} x_i w_i h_i \leq WH \\ & x_i \in \{0,1\} \qquad\qquad i \in B \end{aligned} \tag{6.10}$$

Let $B' \subseteq B$ denote the set of boxes chosen by (6.10). If an exact solver for the packing decision problem can fit the boxes in B' into a container then the corresponding packing is added to RMP (6.8). Otherwise the cut $\sum_{i \in B'} x_i < |B'|$ is added to (6.10) in order to prevent this subset from being chosen in a subsequent iteration.

[PS02] describes a number of optimizations to the 0-1 knapsack problem, e.g. a branch-and-bound approach that sorts the boxes according to increased profit efficiency and branches on the most efficient. It also makes use of a modified version of the CSP algorithm described in section 6.3.3 but this will not be covered here.

6.3.3 A CSP algorithm for OPP-2

The CSP approach for the OPP problem used in [PS02] is interesting for several reasons: It is a fairly different approach to OPP than the algorithms described earlier in this text and an implementation of it has been used as basis for comparison in chapter 5.

Let B' denote the set of boxes chosen in the one-dimensional optimization problem. The algorithm recursively assigns one of the values from $M = \{\text{above}, \text{below}, \text{left}, \text{right}\}$ to each pair $i, j \in B'$. So let $r_{ij} \in M \cup \{\text{NULL}\}$ denote the relation between each i, j pair and set initially $r_{ij} = \text{NULL}$ for all i, j. Furthermore, in order to avoid mirror symmetric solutions, $r_{12} \in \{\text{left}, \text{below}\}$. In each node in the search tree one r_{ij} is assigned a value from

M and it is checked if the boxes are placed in a gap less packing so that no boxes overlap and no box exceed the container boundaries. This is done by assigning coordinates starting from the bottom left most box towards the top right box similar to the construction of the packing p^F defined on page 11. The coordinate assignment can be done in $O(|B'|^2)$ time. If the check fails the branch is dropped, otherwise the search progresses until all relations have been assigned a value in which case a feasible packing has been found.

As the values left/right and above/below are inverse the assignment $r_{ij} = m$ for $m \in M$ implies that $r_{ji} = \neg m$ where $\neg m$ is the inverse value of m. E.g. $r_{ij} = \text{left} \Rightarrow r_{ji} = \text{right}$. Also the relation assignments should be transitive and thus the following implications holds:

$$r_{ij} = m \wedge r_{jk} = m \quad \Rightarrow r_{ik} = m \quad i, j, k \in B'$$

The guillotine property and other requirements to the packing is also handled in the CSP decision solver. To ensure the guillotine property the following check is made after a successful coordinate assignment: A scan for a vertical cut is done by ordering the boxes after increasing x-coordinate and for each distinct x-coordinate separating B' in two subsets S_1 and S_2 so that $r_{ij} = m \in M$ for all $i \in S_1$ and $j \in S_2$. A similar split is made for horizontal cuts. Such a scan can be done in $O(|B'|^2)$ time. The test is afterwards performed recursively on both S_1 and S_2 until all splits are singleton sets or until no further split can be made. At most $O(|B'|)$ such splits can be made and thus the total check is done in $O(|B'|^3)$ time.

6.3.4 Remarks on the bounds

When the iterations progress the partial LP-solutions to (6.8) will decrease and only when the volumn generation process terminates it is certain that the LP-solution is smaller than OPT (and thereby feasible). This final LP-solution is a lower bound for (6.6). Generalizing the algorithm for the decision subproblem or exchanging it with another appropriate algorithm would make the bound apply to OBPP for arbitrarily large d.

Solving the problem using Lagrange relaxation would not have the drawback with slow convergence. However, this relaxation suffers from the need of finding appropriate Lagrange multipliers.

Chapter 7
Solving OBPP-d

Section 6.3 described an algorithm for OBPP-d based on column generation. With a slight modification the algorithm could be generalized to arbitrary values of d. The described algorithm is one approach to OBPP-d and in this chapter we describe another approach: A branch-and-bound algorithm for OBPP-d that brings all the past chapters together by combining the use of OPP-d solvers and lower bounds for OBPP-d. The algorithm is based on the exact branch-and-bound algorithm presented in [MPV97] but is generalized to utilize arbitrary lower bounds and arbitrary OPP-d solvers. By using appropriate bounds and OPP-d solvers the algorithm is no longer restricted to the three-dimensional case.

The overall concept is an outer branch-and-bound algorithm that assigns boxes to containers without taking their position inside the container into account. Throughout the algorithm execution a global variable z holds the currently best found solution value and z can thus always be considered as an upper bound for the further search. Each node contains a set $M = \{C_1, C_2, \ldots, C_n\}$ of containers where each container C_i is associated with a subset $V_i \subseteq V$ of all boxes V.

A container is called *closed* if it can be proved that no more boxes can be packed into it and *open* otherwise. Also an assignment of a box v to a container C_i is referred to as *fixed* if v is assigned C_i for all successor nodes of a subtree.

The search is initialized by sorting all boxes by non-increasing volume and constructing a heuristic root solution M_0 with some assignment of all boxes into a number of containers. In each search node the (by volume) largest unfixed box v is selected and for each open container C_i a branch node is created in which v is fixed to C_i. If a search node N has $|M_N| < z - 1$ then v is also assigned to a new container (which state is therefore initially

open). This can be done because the extra container will still leave $|M_N| < z$. By this branching strategy each search node will represent a box/container fix. Denote by the *current container* the container involved in the fix at the current search node N. Empty containers can be removed from M_N.

At each search node it is determined if a single-container packing exists for the box set assigned to the current container or if the subtree can be skipped. Let C_i denote the current container and let $V_i \subseteq V$ be the subset of boxes currently assigned to C_i. To check feasibility some lower bound $\mathscr{L}(V_i)$ is computed for V_i. If $\mathscr{L}(V_i) \geq 2$ no feasible packing exists and the subtree can be skipped. Otherwise, a heuristic is computed for V_i. Examples of such heuristics are described for $d = 2$ in [MV98] and for $d = 3$ in [MPV97]. The latter can relatively easily be extended to arbitrary values of d. If a heuristic single-container packing is found the search may continue. Otherwise, the last and most cumbersome test is made: OPP-d for V_i is solved with an exact solver. If no packing can be found, the search node is dropped.

When a node N is accepted in the above step, an attempt to close the current container V_i is made. For each yet unfixed $v \in V \setminus V_i$, let $V_i^v = V_i \cup \{v\}$ and calculate $\mathscr{L}(V_i^v)$. If $\mathscr{L}(V_i^v) > 1$ for all v the current container cannot be extend with any available box and it can therefore be closed. Else, let $V' \subset V \setminus V_i$ denote the set of all v for which $\mathscr{L}(V_i^v) = 1$ and try to construct a heuristic packing of $V_i \cup V'$. The container is closed if such a packing is found.

As a last step, the following test is made in order to minimize the search space: Let c denote the number of closed bins in the current search node and V_f denote the set of "free" boxes not yet assigned to a closed container. If $\mathscr{L}(V_f) + c \geq z$ the search node can be dropped. In that case, the subtree will not contain a better solution than the currently best found.

Chapter 8
Conclusion

A number of lower bounds for OBPP and constructive algorithms for OPP-d have been described. Some of the ideas for extending Fekete and Schepers framework have been implemented and in order to test their performance, a total of 11400 test runs were carried out. As seen in chapter 5 the implemented solvers for $d = 2$ could not compare to the CSP solver. Even though some performance can be gained by changing data structures, the most significant improvements are likely to be obtained by purely algorithmic optimizations reducing the large amount of redundancy in the dissolvement based guillotine representation and in the branch-and-bound algorithm for sticky cuttings.

Open problem 3.11 on page 38 ask if the total time complexity for the algorithm searching M_C topologies is smaller than the total time complexity for the algorithm searching M_D topologies. The original solver by Fekete and Schepers is known to perform well [FS97c] and as the performance of the dissolvement based solver did not compare to the CSP based solver in the computational results presented, chances are that the relaxation of P1 and the redundancy introduced by M_D actually worsened the total time complexity. So even though no formal proof has been made, the above observations makes it likely that the answer to open problem 3.11 is *yes*.

The advantage of Fekete and Schepers model is that one single graph holds a whole class of packings, namely the packings represented by all transitive orientations of that graph. The CSP based solver does not handle such classes of equivalence so if some redundancy minimizing changes could be made to the Fekete and Schepers based solver, chances are that it would then indeed be able to compare to the CSP solver.

The newly introduced sticky cutting property may become interesting for several reasons: It is useful in practise, it behaves extreamly nice in a graph

theoretic model and for the special case $k = d = 2$ the results can be utilized in guillotine cuttings. However, further research needs to be made in order to answer the conjecture and open problem given in chapter 4.

The following items summarize the contributions of this thesis:

- The framework by Fekete and Schepers was extended to handle guillotine cuttings and two versions of the modified algorithm was presented: One that follows the original framework relatively strictly but ensures the guillotine property and one in which the guillotine checking extension is utilized to speed up each iteration.
- A proof for a worst case performance ratio of 2 for the guillotine cuttable OBPP-2 was given in co-operation with David Pisinger.
- A whole new type of graph theoretically nice behaving *sticky cuttings* was introduced and a conjecture given for a worst case performance ratio of 4 for the sticky cuttable OBPP-2.
- It was shown how the theory from sticky cuttings could be utilized in conventional guillotine cuttings for $k = d = 2$.
- Both the sticky cutting algorithm and one of the guillotine algorithms was implemented. These are hereby the first known solvers to solve sticky- and guillotine cuttings for arbitrarily large d. Their performance could not compare to the CSP solver for $d = 2$ but as stated, a significant gain is likely to be obtained by applying the algorithmic optimizations suggested in section 5.4.1.
- A new redundancy minimizing *packing tree* representation of guillotine cuttings was presented along with a brute-force algorithm to traverse all packing trees.

8.1 Further work

This text left a number of problems open to further research. Below some ideas for further work are summarized:

1. For guillotine packings some problems are:
   (a) Make a formal proof for the worst case performance $\frac{\mathrm{OPT}_g}{\mathrm{OPT}}$ for $d > 2$ (open problem 3.6).

(b) Various relaxations of Fekete and Schepers model for guillotine cuttings results in different complexity and open problem 3.10, 3.11 and 3.12 try to clarify the impact of various relaxations. Make a formal proof for the answers to one or more of these problems.

(c) Section 3.3.3 described a brute-force algorithm to traverse all packing trees. It could be interesting to construct a branch-and-bound algorithm for this representation.

2. The sticky cutting property is currently quite unexplored, so a general research in this and a classification of its behavior is still to be done. Also:

   (a) Conjecture 4.3 claims that $\frac{\mathrm{OPT}_s}{\mathrm{OPT}} = 4$. Prove it or give a counter evidence.

   (b) Prove or disprove conjecture 4.4 claiming that OPP-d attached with the sticky cutting requirement is NP-hard.

3. Find a way to measure the *quality* of both guillotine- and sticky cutting search nodes in order to be able to test a best-first search strategy.

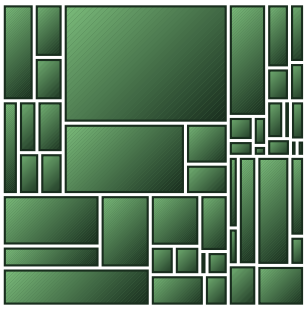
4. It could be interesting to make some experimental work testing the suggestions on performance optimization. That is, make a reimplementation in which

   (a) the use of data structures are optimized for small data sets.

   (b) boxes are enlarged by conservative scales

   (c) P1 is reintroduced.

   These improvements are almost certain to improve the performance. As an experiment two versions of this reimplentation could be made: One searching M_D topologies and one searching the less redundant M_C topologies.



Bibliography

[BM03] Marco A. Boschetti and Aristide Mingozzi. The two-dimensional finite bin packing problem. part I: New lower bounds for the oriented case. *4OR*, 1(1):27–42, 2003.

[CRLS01] Thomas H. Cormen, Ronald L. Rivest, Charles E. Leiserson, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[CW77] N. Christofides and C. Whitlock. An algorithm for two dimensional cutting problems. 25:30–44, 1977.

[Del98] M. Dell'Amico. On the continuous relaxation of packing problems. Technical report, Dipartimento di Economia Politica, Università di Modena, 1998. Materiali di discussione 182.

[DM95] M. Dell'Amico and S. Martello. Optimal scheduling of tasks on identical parallel processors. *ORSA Journal on Computing*, 7(2):191–200, 1995.

[DW60] G.B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8:101–111, 1960.

[FS97a] S. P. Fekete and J. Schepers. On higher-dimensional packing I: Modeling. Technical Report 97–288, 1997.

[FS97b] S. P. Fekete and J. Schepers. On more-dimensional packing II: Bounds. *submitted to: Discrete Applied Mathematics*, 1997.

[FS97c] S. P. Fekete and J. Schepers. On more-dimensional packing III: Exact algorithms. *submitted to: Discrete Applied Mathematics*, 1997.

[FS01] S. P. Fekete and J. Schepers. New classes of lower bounds for the bin packing problem. *Mathematical Programming*, 91:11–31, 2001.

[FS04] Sándor P. Fekete and Jörg Schepers. A combinatorial characterization of higher-dimensional orthogonal packing. *Math. Oper. Res.*, 29(2):353–368, 2004.

[Gol80] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, San Diego, 1980.

[LUE83] G. S. LUEKER. Bin packing with items uniformly distributed over intervals [a,b]. *Proceedings 24th Ann. Symp. on Foundations of Computer Science*, pages 289–297, 1983.

[MFNK96] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. VLSI module placement based on rectangle-packing by the sequence pair. *IEEE Trans. on CAD*, 15(12):1518–1524, 1996.

[MPV97] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. Technical report, 1997. Technical Report DEIS-OR-97-6, 1997. Available at http://www.deis.unibo.it.

[MPV04] Silvano Martello, David Pisinger, and Daniele Vigo. Algoriths for general and robot-packable variants of the three-dimensional bin packing problem. *ACM Transactions on Mathematical Software*, 2004. Accepted for publication.

[MS97] Ross M. McConnell and Jeremy P. Spinrad. Linear-time transitive orientation. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 19–25, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[MT90] Silvano Martello and Paolo Toth. *Knapsack problems: Algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[MV98] Silvano Martello and Daniele Vigo. Exact solution of the two-dimensional finite bin packing problem. *Manage. Sci.*, 44(3):388–399, 1998.

[NP04] Stavros D. Nikolopoulos and Leonidas Palios. Hole and anti-hole detection in graphs. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 850–859, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[Pis03] David Pisinger. Denser placements in vlsi design obtained in $O(n \log \log n)$ time. *INFORMS Journal on Computing*, 2003. Accepted for publication.

[PS02] David Pisinger and Mikkel Sigurd. Using decomposition techniques and constraint programming for solving the two-dimensional bin packing problem. Technical report, Dept. of Computer Science, University of Copenhagen, 2002. To appear in *INFORMS Journal on Computing* (2005).

[ST96] Guntram Scheithauer and Johannes Terno. A new heuristic for the pallet loading problem. *JORS*, 47:511–522, 1996.

[Wan83] P.Y. Wang. Two algorithms for constrained two-dimensional cutting stock problems. *Oper. Res.*, 31:573–586, 1983.

[WL86] D. F. Wong and C. L. Liu. A new algorithm for floorplan design. In *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*, pages 101–107, Piscataway, NJ, USA, 1986. IEEE Press.

[Wol98] L. A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.