OPERATIONS RESEARCH
Vol. 55, No. 3, May–June 2007, pp. 569–587
ISSN 0030-364X | EISSN 1526-5463 | 07 | 5503 | 0569

DOI 10.1287/opre.1060.0369

An Exact Algorithm for Higher-Dimensional Orthogonal Packing

Sándor P. Fekete
Department of Mathematical Optimization, Braunschweig University of Technology,
D–38106 Braunschweig, Germany, s.fekete@tu-bs.de

Jörg Schepers
IBM Germany, Gustav-Heinemann-Ufer 120/122, D–50968 Köln, Germany, schepers@de.ibm.com

Jan C. van der Veen
Department of Mathematical Optimization, Braunschweig University of Technology,
D–38106 Braunschweig, Germany, j.van-der-veen@tu-bs.de

Higher-dimensional orthogonal packing problems have a wide range of practical applications, including packing, cutting, and scheduling. Combining the use of our data structure for characterizing feasible packings with our new classes of lower bounds, and other heuristics, we develop a two-level tree search algorithm for solving higher-dimensional packing problems to optimality. Computational results are reported, including optimal solutions for all two-dimensional test problems from recent literature.

This is the third in a series of articles describing new approaches to higher-dimensional packing.

Subject classifications: production, scheduling: cutting stock, trim.
Area of review: Optimization.
History: Received May 2000; revisions received May 2004, November 2005; accepted April 2006.

1. Introduction

Most combinatorial optimization problems that need to be solved in practical applications turn out to be members of the class of NP-hard problems. Algorithmic research for several decades has provided strong evidence that for all of these problems, it is highly unlikely that there is a *polynomial algorithm*. Such an algorithm is guaranteed to find an optimal solution in time that, even in the worst case, can be bounded by a polynomial in the size of the input. If no such bound can be guaranteed, the necessary time for solving instances tends to grow very quickly as the instance size increases. That is why NP-hard problems have also been dubbed intractable. See the classical monograph by Garey and Johnson (1979) for an overview.

When confronted with an NP-hard problem, there are several ways to deal with its computational difficulty:

We can look for a different problem. While this way out may be quite reasonable in a theoretical context, it tends to work less well when a problem arises in practical applications that have to be solved somehow.

We can look for special properties of a problem instance or relax unimportant constraints to get a polynomial algorithm. Unfortunately, practical instances and their additional constraints tend to be more difficult at a second glance, rather than simpler.

We can look for a good solution instead of an optimal one. This approach has received an increasing amount of attention in recent years. In particular, there has been a tremendous amount of research dealing with polynomial time approximation algorithms that are guaranteed to find a solution within a fixed multiplicative constant of the optimum. See Hochbaum (1996) for an overview.

We can look for an optimal solution without a bound on the run time. While the time for finding an optimal solution may be quite long in the worst case, a good understanding of the underlying mathematical structure may allow it to find an optimal solution (and prove it) in reasonable time for a large number of instances. A good example of this type can be found in Grötschel (1980), where the exact solution of a 120-city instance of the traveling salesman problem is described. In the meantime, benchmark instances of size up to 13,509 and 15,112 cities have been solved to optimality (Applegate et al. 1998), showing that the right mathematical tools and sufficient computing power may combine to explore search spaces of tremendous size. In this sense, intractable problems may turn out to be quite tractable.

In this paper, we consider a class of problems that is not only NP-hard, but also difficult in several other ways. Packing rectangles into a container arises in many industries, where steel, glass, wood, or textile materials are cut, but it also occurs in less obvious contexts, such as machine scheduling or optimizing the layout of advertisements in newspapers. The three-dimensional problem is important for practical applications such as container loading or scheduling with partitionable resources. Other applications arise from allocating jobs to reconfigurable computer chips—see Teich et al. (2001). For many of these problems, objects must be positioned with a fixed orientation; this is a requirement that we will assume throughout the paper. The *d-dimensional orthogonal knapsack problem* (OKP-*d*) requires selection of a most valuable subset S from a given set of boxes, such that S can be packed into the container. Being a generalization of the one-dimensional bin-packing problem, the OKP-*d* is NP-complete in the strict sense. Other NP-hard types of packing problems include the *strip-packing problem* (SPP), where we need to minimize the height of a container of given width, such that a given set of boxes can be packed, and the *orthogonal bin-packing problem* (OBPP), where we have a supply of containers of a given size and need to minimize the number of containers that are needed for packing a set of boxes. The decision version of these problems is called the *orthogonal packing problem* (OPP), where we have to decide whether a given set of boxes fits into a container.

Relatively few authors have dealt with the exact solution of orthogonal knapsack problems. All of them focus on the problem in two dimensions. One of the reasons is the difficulty of giving a simple mathematical description of the set of feasible packings: As soon as one box is packed into the container, the remaining feasible space is no longer convex, excluding the straightforward application of integer programming methods. Biró and Boros (1984) give a characterization of nonguillotine patterns using network flows, but derived no algorithm. Dowsland (1987) proposes an exact algorithm for the case that all boxes have equal size. Arenales and Morabito (1995) extend an approach for the guillotine problem to cover a certain type of nonguillotine patterns. So far, only three exact algorithms have been proposed and tested for the general case. Beasley (1985) and Hadjiconstantinou and Christofides (1995) give different 0-1 integer programming formulations of this problem. Even for small-problem instances, they have to consider very large 0-1 programs, because the number of variables depends on the size of the container that is to be packed. The largest instance that is solved in either article has nine out of 22 boxes packed into a 30×30 container. After an initial reduction phase, Beasley gets a 0-1 program with more than 8,000 variables and more than 800 constraints; the program by Hadjiconstantinou and Christofides still contains more than 1,400 0-1 variables and over 5,000 constraints. From Lagrangean relaxations, they derive upper bounds for a branch-and-bound algorithm, which are improved using subgradient optimization. The process of traversing the search tree corresponds to the iterative generation of an optimal packing. More recent work by Caprara and Monaci (2004) on the two-dimensional knapsack problem uses our previous results (cited as Fekete and Schepers 1997a, 1997d) as the most relevant reference for comparison; we compare our results and approaches later in this paper and discuss how a combination of our methods may lead to even better results.

Other research on the related problem of two- and three-dimensional bin packing has been presented: Martello and Vigo (1998) consider the two-dimensional case, while Martello et al. (2000) deal with three-dimensional bin packing. We discuss aspects of those papers in Fekete and Schepers (2004b), when considering bounds for higher-dimensional packing problems. Padberg (2000) gives a mixed-integer programming formulation for three-dimensional packing problems, similar to the one anticipated by the second author in his thesis (Schepers 1997). Padberg expresses the hope that using a number of techniques from branch and cut will be useful; however, he does not provide any practical results to support this hope.

In our papers (Fekete and Schepers 2004a, b), we describe a different approach to characterizing feasible packings and constructing optimal solutions. We use a graph-theoretic characterization of the relative position of the boxes in a feasible packing (Fekete and Schepers 2004a). Combined with good heuristics for dismissing infeasible subsets of boxes, which are described in Fekete and Schepers (2004b), this characterization can be used to develop a two-level tree search. In this third paper of the series, we describe how this exact algorithm can be implemented. Our computational results show that our code outperforms previous methods by a wide margin. It should be noted that our approach has been used and extended in the practical context of reconfigurable computing (Teich et al. 2001), which can be interpreted as packing in three-dimensional space, with two coordinates describing chip area and one coordinate describing time. Order constraints for the temporal order are of vital importance in this context; as it turns out, our characterization of feasible packings is particularly suited for taking these into account. See our follow-up paper (Fekete et al. 2001) for a description of how to deal with higher-dimensional packing with order constraints.

The rest of this paper is organized as follows: After recalling some basics from our papers (Fekete and Schepers 1997b, c; 2004a, b) in §2, we give detailed account of our approach for handling OPP instances in §3. This analysis includes a description of how to apply graph-theoretic characterizations of interval graphs to searching for optimal packings. Section 4 provides details of our branch-and-bound framework and the most important subroutines. In §5, we discuss our computational results. Section 6 gives a brief description of how our approach can be applied to other types of packing problems.

2. Preliminaries

We are given a finite set V of d-dimensional rectangular boxes with "sizes" $w(v) \in \mathbb{R}_0^{+d}$ and "values" $c(v) \in \mathbb{R}_0^+$ for $v \in V$. As we are considering fixed orientations, $w_i(v)$ describes the size of box v in the x_i-direction. The objective of the *d-dimensional orthogonal knapsack problem* (*OKP-d*) is to maximize the total value of a subset

$V' \subseteq V$ fitting into the container C and to find a complying packing. Closely related is the *d-dimensional orthogonal packing problem* (*OPP-d*), which is to decide whether a given set of boxes B fits into a unit-size container, and to find a complying packing whenever possible.

For a d-dimensional packing, we consider the projections of the boxes onto the d coordinate axes x_i. Each of these projections induces a graph G_i: Two boxes are adjacent in G_i if and only if their x_i projections overlap. A set of boxes $S \subseteq V$ is called x_i*-feasible* if the boxes in S can be lined up along the x_i-axis without exceeding the x_i-width of the container.

As we show in Fekete and Schepers (1997b, 2004a), we have the following characterization of feasible packings:

Theorem 1. *A set of boxes V can be packed into a container if and only if there is a set of d graphs* $G_i = (V, E_i)$, $i = 1, \dots, d$, *with the following properties*:
$P1$: *the graphs* $G_i := (V, E_i)$ *are interval graphs.*
$P2$: *each stable set S of* G_i *is* x_i*-feasible.*
$P3$: $\bigcap_{i=1}^{d} E_i = \varnothing$.

A set $E = (E_1, \dots, E_d)$ of edges is called a *packing class* for (V, w) if and only if it satisfies the conditions $P1$, $P2$, $P3$. Note that when constructing a packing from a packing class, some edges may be added in case of a degenerate packing; see Ferreira and Oliveira (2005) for such an example. This does not impede the correctness of the theorem or its applicability.

3. Solving Orthogonal Packing Problems

For showing feasibility of any solution to a packing problem, we have to prove that a particular set of boxes fits into the container. This subproblem is called the *orthogonal packing problem* (*OPP*).

To get a fast positive answer, we can try to find a packing by means of a heuristic. A fast way to get a negative answer has been described in our paper (Fekete and Schepers 2004b): Using a selection of bounds (conservative scales), we can try to apply the volume criterion to show that there cannot be a feasible packing.

In this section, we discuss the case in which both of these easy approaches fail. Because the OPP is NP-hard in the strong sense, it is reasonable to use enumerative methods. As we showed in our paper (Fekete and Schepers 2004a) the existence of a packing is equivalent to the existence of a packing class. Furthermore, we have shown that a feasible packing can be constructed from a packing class in time that is linear in the number of edges. This allows us to search for a packing class, instead of a packing. As we will see in the following, the advantage of this approach lies not only in exploiting the symmetries discussed in Fekete and Schepers (1997b, 2004a), but also in the fact that the structural properties of packing classes give rise to very efficient rules for identifying irrelevant portions of the search tree.

3.1. Basic Idea of the Enumeration Scheme

The enumeration of packings described by Beasley (1985) emulates the intuitive idea of packing objects into a box: Each branching corresponds to placing a box at a particular position in the container, or disallowing this placement. Thus, each search node corresponds to a partial packing that is to be augmented to a complete packing. Our enumeration of packing classes is more abstract than that: At each branching, we decide whether two boxes b and c overlap in their projection onto the i-axis, so that the edge $e := bc$ is contained in the ith component graph of the desired packing class E. Accordingly, in the first resulting subtree, we only search for packing classes E with $e \in E_i$; in the second, we only search for E with $e \notin E_i$. Hence, the resulting "incomplete packing classes" do not correspond to packing classes of subsets of boxes; instead, they are (almost) arbitrary tuples of edges.

More precisely, we will store the necessary and excluded edges for each node N of the search tree and each coordinate direction i in two data structures $\mathscr{E}^N_{+,i}$ and $\mathscr{E}^N_{-,i}$. Therefore, the search space for N contains precisely the packing classes that satisfy the condition

$$\mathscr{E}^N_{+,i} \subseteq E_i \subseteq \bar{\mathscr{E}}^N_{-,i}, \quad i \in \{1, \dots, d\}, \tag{1}$$

where $\bar{\mathscr{E}}^N_{-,i}$ is the complement of $\mathscr{E}^N_{-,i}$. Summarizing, we write

$$\mathscr{E}^N_{+} := (\mathscr{E}^N_{+,1}, \dots, \mathscr{E}^N_{+,d}), \qquad \mathscr{E}^N_{-} := (\mathscr{E}^N_{-,1}, \dots, \mathscr{E}^N_{-,d}),$$
$$\mathscr{E}^N := (\mathscr{E}^N_{+}, \mathscr{E}^N_{-}),$$

and denote by $\mathscr{L}(\mathscr{E}^N)$ the search space for N; by virtue of (1), this search space is only determined by $\mathscr{E}^N$. $\mathscr{E}^N$ is called the *search information* of node N, because this tuple of data structures represents the information that is currently known about the desired packing class.

An important part of the procedure consists of using the characteristic properties $P1$, $P2$, $P3$ for increasing the information on the desired packing class that is contained in $\mathscr{E}^N$. For example, let the edge e be contained in $\mathscr{E}_{+,i}$ for all $i \neq k$. Furthermore, let $E \in \mathscr{L}(\mathscr{E}^N)$. For $i \neq k$, we have $e \in E_i$ because of $\mathscr{E}^N_{+,i} \subseteq E_i$. Because of $P3$, the intersection of all E_i must be empty, implying $e \notin E_k$. Therefore, $\mathscr{E}^N_{-,k}$ can be augmented by e without changing the search space. Similar augmentation rules can be described for $P1$ and $P2$.

Depending on whether the edge e is added to $\mathscr{E}^N_{+,i}$ or to $\mathscr{E}^N_{-,i}$, we describe augmentations of $\mathscr{E}^N$ by the triples $(e, +, i)$ or $(e, -, i)$.

Because it suffices to find a single packing class, these augmentations may reduce the search space, as long as it is guaranteed that not all packing classes are removed from it. This fact allows us to exploit certain symmetries. Thus, we use *feasible augmentations* of $\mathscr{E}^N$ in the sense that a nonempty search space $\mathscr{L}(\mathscr{E}^N)$ stays nonempty after the augmentation.

When augmenting $\mathscr{E}^N$ we follow two objectives:

(1) obtain a packing class in $\mathscr{E}^N_+$,

(2) prove that every augmentation of $\mathscr{E}^N_+$ to a packing class has to use excluded edges from $\mathscr{E}^N_-$.

In the first case, our tree search has been successful. In the second case, the search on the current subtree may be terminated because the search space is empty. Otherwise, we have to continue branching until one of the two objectives is reached.

3.2. Excluded Induced Subgraphs

For our algorithm, we need three components: a test "Is $\mathscr{E}^N_+$ a packing class?," a sufficient criterion that $\mathscr{E}^N_+$ has no feasible augmentation, and a construction method for feasible augmentations. As we describe in our paper (Fekete and Schepers 1997b, 2004a), all three of these components can be reduced to identifying or avoiding particular induced subgraphs in the portions of E that are fixed by $\mathscr{E}^N$.

As we have already seen, it is easy to determine all edges that are excluded by condition $P3$. By performing these augmentations of $\mathscr{E}^N_-$ immediately, we can guarantee that $P3$ is satisfied. Thus, we will assume in the following that $P3$ is satisfied. Furthermore, we will implicitly refer to the current search node N and abbreviate $\mathscr{E}^N$ by $\mathscr{E}$.

$P2$ explicitly excludes certain induced subgraphs: i-infeasible stable sets, i.e., i-infeasible cliques in the complement of each component graph.

To formulate $P1$ in terms of excluded induced subgraphs, we recall the following Theorems 3 and 4—see the book by Golumbic (1980), as well as a resulting linear-time algorithm by Korte and Möhring (1989). The following terminology is used:

Definition 2. For a graph $G := (V, E)$, a set $F \subseteq V^2$ of directed edges is an *orientation of* G, iff

$$\forall b, c \in V\colon\ \ bc \in E \Longleftrightarrow (\vec{bc} \in F \wedge \vec{cb} \notin F) \vee (\vec{cb} \in F \wedge \vec{bc} \notin F)$$

holds. An orientation F of a graph (V, E) is called *transitive*, if in addition,

$$\forall b, c, z \in V\colon\ \ \vec{bc} \in F \wedge \vec{cz} \in F \Rightarrow \vec{bz} \in F$$

holds.

A graph is called a *comparability graph*, iff it has a transitive orientation.

For a cycle $C := [b_0, \dots, b_{k-1}, b_k = b_0]$ of length k, the edges b_ib_j, $i, j \in \{0, \dots, k-1\}$ with $(|i-j| \bmod k) > 1$ are called *chords*; the chords b_ib_j, $i, j \in \{0, \dots, k-1\}$ with $(|i-j| \bmod k) = 2$ are called 2-*chords* of C. A cycle is (2-) *chordless*, iff it does not have any (2-) chords.

A graph $G = (V, E)$ is a *cocomparability graph*, if the complement graph $G = (V, \bar{E})$ is a comparability graph.

Theorem 3 (Gilmore and Hoffman 1964). *A cocomparability graph is an interval graph, iff it does not contain the chordless cycle C_4 of length 4 as an induced subgraph.*

Theorem 4 (Ghouilà-Houri 1962, Gilmore and Hoffman 1964). *A graph is a comparability graph, iff it does not contain a two-chordless cycle of odd length.*

Thus, $\mathscr{E}_+$ is a packing class, if for all $i \in \{1, \dots, d\}$ the following holds (recall that $P3$ is assumed to be satisfied):

(1) $(V, \mathscr{E}_{+,i})$ does not contain a C_4 as an induced subgraph.

(2) $(V, \overline{\mathscr{E}}_{+,i})$ does not contain an odd two-chordless cycle.

(3) $(V, \overline{\mathscr{E}}_{+,i})$ does not contain an i-infeasible clique.

With the help of this characterization, we get a stop criterion for subtrees. Because only those edges can be added to $\mathscr{E}_{+,i}$ that are not in $\mathscr{E}_{-,i}$, $\mathscr{E}_+$ cannot be augmented to a packing class, if for $i \in \{1, \dots, d\}$ one of the following conditions holds:

(1) $(V, \mathscr{E}_{+,i})$ contains a C_4 as an induced subgraph, with both chords lying in $\mathscr{E}_{-,i}$.

(2) $(V, \mathscr{E}_{-,i})$ contains an odd two-chordless cycle, with all its two-chords lying in $\mathscr{E}_{+,i}$.

(3) $(V, \mathscr{E}_{-,i})$ contains a i-infeasible clique.

Suppose that except for one edge e, one of these excluded configurations is contained in $\mathscr{E}$. Because of condition (1), the corresponding incomplete induced subgraph is contained in the ith component graph of each packing class $E \in \mathscr{L}(\mathscr{E})$. Because completing the excluded subgraph would contradict condition $P1$ or $P2$, the membership of e in E_i is determined. The resulting forced edges can be added to $\mathscr{E}_{+,i}$ or to $\mathscr{E}_{-,i}$ without decreasing the search space.

Example. If $(V, \mathscr{E}_{+,i})$ contains an induced C_4, for which one chord is contained in $\mathscr{E}_{-,i}$, for any packing class of the search space, the other chord e must be contained in the ith component graph. Thus, the augmentation $(e, +, i)$ is feasible, but not the augmentation $(e, -, i)$.

In this way, we can reduce the search for a feasible augmentation to the search for incomplete excluded configurations. In the next section, we will relax the condition that only one edge is missing from a configuration, and only require that the missing edges are equivalent in a particular sense.

3.3. Isomorphic Packing Classes

When exchanging the position of two boxes with identical sizes in a feasible packing, we obtain another feasible packing. Similarly, we can permute equal boxes in a packing class:

Theorem 5. *Let E be a packing class for (V, w) and π: $V \to V$ be a permutation with*

$$\forall b \in V\colon\ \ w(b) = w(\pi(b)). \tag{2}$$

Then, the d-tuple of edges in E^π that is given by

$$\forall b, c \in V\ \forall i \in \{1, \dots, d\} \quad bc \in E_i \Leftrightarrow \pi(b)\pi(c) \in E_i^\pi$$

is a packing class.

Proof. Because the structure of the component graphs does not change, conditions $P1$ and $P3$ remain valid. Because of (2), the weight of stable sets remains unchanged, so that $P2$ remains valid as well. □

We get a notion of isomorphism that is similar to the isomorphism of graphs:

Definition 6. Two packing classes E and E' are called isomorphic, iff there is a permutation π: $V \to V$ satisfying (2), such that $E' = E^\pi$.

Keeping only one packing class from each isomorphism class in search space avoids unnecessary work. For this purpose, we may assume that the ordering of equal boxes in a packing class follows the lexicographic order of their position vectors. As a result, in the two-dimensional case, the left-most and bottom-most box of a box type will have the lowest index. This corresponds to generating packings according to "left-most downward placement" in Hadjiconstantinou and Christofides (1995). This approach cannot be used for packing classes because there are no longer any orientations (left/right, up/down, etc.).

Until now, no algorithm has been found that can decide in polynomial time whether two graphs are isomorphic, and it has been conjectured that no such algorithm exists (see Papadimitriou 1994, p. 291). When deciding whether two packing classes are isomorphic, this decision has to be made repeatedly. In addition, packing classes may only be known partially. This makes it unlikely that there is an efficient method for achieving optimal reduction of isomorphism. Therefore, we are content with exploiting certain cases that occur frequently.

In the initializing phase, we may conclude by Theorem 16 from our paper (Fekete and Schepers 2004b) (corresponding to Theorem 11 in Fekete and Schepers 1997c) that for a box type T, there is a component graph $(V, E_i)[T]$ for which there is a clique of size $k \geqslant 2$. Then, we can choose the numbering of T, such that the first k boxes from T belong to the clique. Thus, the corresponding $\binom{k}{2}$ edges can be fixed in $\mathscr{E}_{+,i}$. This restriction of numbering T corresponds to excluding isomorphic packing classes.

In the following, we only consider isomorphic packing classes for which the permutation in Definition 6 exchanges precisely two boxes, while leaving all other boxes unchanged. This restricted isomorphism can be checked easily. We have to search for pairs of boxes that can be exchanged in the following way:

Definition 7. Let (V, w) be an OPP instance with search information $\mathscr{E}$. Two boxes $b, c \in V$ with $w(b) = w(c)$ are called *indistinguishable* (*with respect to* $\mathscr{E}$), iff all adjacencies of b and c have identical search information, i.e.,

$$\forall i \in \{1, \dots, d\}\ \forall \sigma \in \{+, -\}\ \forall z \in V \backslash \{b, c\}: \quad bz \in \mathscr{E}_{\sigma,i} \Leftrightarrow cz \in \mathscr{E}_{\sigma,i}. \quad (3)$$

Two edges $e, e' \in E_V$ are called *indistinguishable* (*with respect to* $\mathscr{E}$), if there are representations $e = bc$ and $e' = b'c'$, such that the boxes b and b', as well as c and c', are indistinguishable (with respect to $\mathscr{E}$).

The property of being indistinguishable is an equivalence relation for boxes as well as for edges.

The following lemma allows it to exploit the connection between indistinguishable edges and isomorphic packing classes:

Lemma 8. *Let (V, w) be an OPP instance with search information $\mathscr{E}$. Let A be set of indistinguishable edges on V. Let e be an arbitrary $e \in A$. Then, for any packing class $E \in \mathscr{L}(\mathscr{E})$ that satisfies $A \cap E_i \neq \varnothing$, there is an isomorphic packing class $E' \in \mathscr{L}(\mathscr{E})$ with $e \in E'_i$.*

Proof. Let e' be an edge from the set $A \cap E_i$. Then, e and e' are indistinguishable. Hence, there is a representation $e = bc$ and $e' = b'c'$, such that b, b' and c, c' are pairs of indistinguishable boxes. Let π be the permutation of V that swaps b and b', and c and c', and let $E' := E^\pi$. Applying (3) twice, it follows from $E \in \mathscr{L}(\mathscr{E})$ that $E' \in \mathscr{L}(\mathscr{E})$. □

Lemma 8 can be useful in two situations:

(1) If we branch with $e \in A$ with respect to the i-direction, then we may assume for all $e' \in A$ in the subtree given by $e \notin E_i$ that $e' \notin E_i$: For each packing class excluded in this way, there is an isomorphic packing class that is contained in the search space of the subtree $e \in E_i$.

(2) If during the course of our computations we get $A \cap \mathscr{E}_{+,i} \neq \varnothing$, then for an arbitrary $e \in A$, the augmentation $(e, +, i)$ is feasible because only isomorphic duplicates are lost.

3.4. Pruning by Conservative Scales

By Lemma 20 from our paper (Fekete and Schepers 2004b, Lemma 15 in Fekete and Schepers 1997c), we can use the information given by $\mathscr{E}$ to modify a given conservative scale, such that the resulting total volume of V is increased. (See our related paper (Fekete and Schepers 2001) for a general technique based on dual-feasible functions.) Before branching, we try to apply the lemma repeatedly, such that the transformed volume exceeds the volume of the container. In this case, the search can be stopped.

Because this reduction heuristic requires the computation of several one-dimensional knapsack problems, it only pays off to use it on nodes where it may be possible to cut off large subtrees. Therefore, we have only used it on nodes of depth of at most five.

4. Detailed Description of the OPP Algorithm

In this section, we give a detailed description of our implementation of the OPP algorithm. We will omit the description of standard techniques like efficient storage of sets, lists, graphs, or the implementation of graph algorithms. The interested reader can find these in Mehlhorn (1984) and Golumbic (1980).

Figure 1. OPP tree search.

```
Call:    Solve_OPP(P)
Input:   An OPP-n instance P := (V, w).
Output:  A packing class for (V, w), if there is one,
         and SUCCESS, otherwise NULL.

 1. 𝒩 := {N_0}.
 2. initialize ℰ^{N_0}.
 3. (e, σ, i)^{N_0} := NULL.
 4.
 5. while (𝒩 ≠ ∅) do
 6.
 7.   choose N ∈ 𝒩.
 8.   𝒩 := 𝒩\{N}.
 9.   (e, σ, i) := (e, σ, i)^N.
10.
11.   repeat
12.     if (Update_searchinfo(P, (e, σ, i), ℰ^N) = EXIT) then
13.       result := EXIT.
14.     else
15.       result := Packingclass_test(P, ℰ^N, (e, σ, i)).
16.     end if
17.   until (result ≠ FIX)
18.
19.   if (result = SUCCESS) then return ℰ_+^N.
20.
21.   if (result = BRANCH) then
22.     Create two new nodes N', N''.
23.     ℰ^{N'} := ℰ^N,   (e, σ, i)^{N'} := (e, +, i).
24.     ℰ^{N''} := ℰ^N,  (e, σ, i)^{N''} := (e, −, i).
25.     𝒩 := 𝒩 ∪ {N', N''}.
26.   end if
27.
28. end while
29.
30. return NULL.
```

4.1. Controlling the Tree Search

The nodes of the search tree are maintained in a list $\mathcal{N}$. For each node $N \in \mathcal{N}$, there is the search information $\mathscr{E}^N$ (see above) and a triple $(e, \sigma, i)^N$ with $e \in \binom{V}{2}$, $\sigma \in \{+, -\}$, and $i \in \{1, \ldots, d\}$. This triple represents the new information when branching at N, i.e., $e \in E_i$ for $\sigma = +$, or $e \notin E_i$ for $\sigma = -$.

Figure 1 shows the course of the tree search. Lines 1 through 3 initialize $\mathcal{N}$ with the root node N_0. Initially, the components of $\mathscr{E}^{N_0}$ do not contain any edges. $(e, \sigma, i)^{N_0}$ is assigned a special value of NULL.

In the while loop of lines 5 through 28, individual nodes are processed; if necessary, their children are added to $\mathcal{N}$. The particular branching strategy (breadth first or depth first) can be specified by a selection mechanism in line 7. If line 30 is reached, then the whole search tree was checked without finding a packing pattern.

Each node N of the search tree is processed as follows:

In routine **Update_searchinfo**, the augmentation of $\mathscr{E}^N$ described by (e, σ, i) is carried out, as long as there are feasible augmentations. If it is detected that the search on N can be stopped, **Update_searchinfo** outputs the value EXIT. Otherwise, the routine terminates with OK.

If **Update_searchinfo** was terminated with OK, then the routine **Packingclass_test** checks whether $\mathscr{E}_+^N$ already is a packing class. In case of a positive answer, SUCCESS is output, and the algorithm terminates in line 19. Otherwise, there are three possibilities:

(1) FIX: The triple (e, σ, i) was updated in **Packingclass_test** to a new feasible augmentation that was returned to **Update_searchinfo**.

(2) EXIT: $\mathscr{E}_+^N$ cannot be augmented to a packing class without using edges from $\mathscr{E}_-^N$. The search on this subtree is stopped.

(3) BRANCH: In lines 22 through 25, two children of N are added to $\mathcal{N}$. The triple (e, σ, i) that was set in **Packingclass_test** contains the branching edge e and the branching direction i.

4.2. Testing for Packing Classes

Figure 2 shows routine **Packingclass_test**. As we have seen, $\mathscr{E}_+$ is a packing class, iff no excluded configuration occurs in any coordinate direction. In this case, in each iteration of the i loop, we keep $A = \varnothing$, and the routine terminates

Figure 2. Routine Packingclass_test.

```
Call:    Packingclass_test(P, ℰ, (e, σ, i)^out)
Input:   An OPP-n instance P := (V, w), search information ℰ.
Output:  (e, σ, i)^out, and EXIT, FIX, BRANCH, or SUCCESS

for i ∈ {1, ..., d} do

  A := ∅.
  if (V, ℰ̄_{+,i}) is not a comparability graph then
    A := set of edges of the 2-chordless odd cycle.
  else
    if a maximal weighted clique in (V, ℰ̄_{+,i})
        is i-infeasible then
      A := edge set of this clique.
    else
      if (V, ℰ_{+,i}) contains an induced C_4 then
        A := set of chords of this C_4.
      end if
    end if
  end if


  if (A ≠ ∅) then
    if (A\ℰ_{−,i} = ∅) then
      return EXIT.
    else
      Choose an edge e from A\ℰ_{−,i}.
      (e, σ, i)^out := (e, +, i).
      if (A\ℰ_{−,i} = {e}) then
        return FIX.
      else
        return BRANCH.
      end if
    end if
  end if

end for i

return SUCCESS.
```

in line 33 with value SUCCESS. Otherwise, A contains a set of edges, out of which at least one has to be added to $\mathscr{E}_{+,i}$ to remove the excluded configuration. This edge must not be from $\mathscr{E}_{-,i}$. Thus, the search on the subtree can be stopped for $A\backslash\mathscr{E}_{-,i} = \varnothing$, and for $|A\backslash\mathscr{E}_{-,i}| = 1$, the only edge must be added to $\mathscr{E}_{+,i}$.

Otherwise, an arbitrary edge from $A\backslash\mathscr{E}_{-,i}$, together with a coordinate direction i, is returned in the triple $(e, \sigma, i)^{out}$ and used for branching.

In line 4 it is tested with the help of the decomposition algorithm from Golumbic (1980, p. 129f) to determine whether we have a comparability graph. The run time is $O(\delta|E|)$, where δ is the maximal degree of a vertex, and E is the edge set of the examined graph. It is simple to modify the algorithm, such that it returns a two-chordless cycle in case of a negative result.

With the help of the algorithm from Golumbic (1980, p. 133f), we can determine a maximal weighted clique in a comparability graph in time that is linear in the number of edges. This algorithm is called in line 7, because graphs at this stage have already passed the test for comparability graphs.

The search for a C_4 in line 10 can be realized by two nested loops that enumerate possible pairs of opposite edges in a potential C_4.

4.3. Updating the Search Information

Figure 3 gives an overview of Routine **Update_searchinfo**. In the following, we will always refer to the current search node N and denote the search information $\mathscr{E}^N$ by $\mathscr{E}$.

The input triple $(e, \sigma, i)^{in}$ either describes an augmentation of the search information (e is fixed in $\mathscr{E}_{\sigma,i}$), or it contains the value NULL on the root node.

On the root node, the search information is initialized as follows: First, the edges are fixed for which the vertices form an infeasible stable set with two elements. For $i \in \{1, \ldots, d\}$, this means that all edges $e = bc$ with $w_i(b) + w_i(c) > 1$ are added to $\mathscr{E}_{+,i}$. Furthermore, we use Theorem 16 from our paper Fekete and Schepers (2004b) (corresponding to Theorem 11 in Fekete and Schepers 1997c) to fix cliques within the subgraphs induced by the individual box types.

The augmentation $(e, \sigma, i)^{in}$ has either been fixed in the last branching step, or it was returned by the routine **Packingclass_test** together with the value FIX. In the latter case, $\sigma = +$ holds, so we know in case of $\sigma = -$ that the augmentation results from a branching step. In §3.3, we concluded from Lemma 8 that in this case, all edges in $\mathscr{E}_{-,i}$ that are indistinguishable from e can be fixed. This is done in lines 10 through 13.

In the main loop (lines 17 through 26), for each augmentation of $\mathscr{E}$ it is checked whether it arises from a configuration that allows it to fix further edges, or the search is stopped. This recursive process is controlled by the list L.

The crucial work is done by the subroutines **Check_P3**, **Avoid_C4**, and **Avoid_cliques**.

Figure 3. Routine Update_searchinfo.

Call: **Update_searchinfo**$(P, (e, \sigma, i)^{in}, \mathscr{E})$
Input: An OPP-n instance $P := (V, w)$, an augmentation $(e, \sigma, i)^{in}$, the search information $\mathscr{E}$.
Output: The updated search information $\mathscr{E}$, and EXIT or OK.

```
1.  (e, σ, i) := (e, σ, i)^in.
2.
3.  if ((e, σ, i) = NULL) then
4.     initialize 𝓔 and mark this augmentation in L
5.  else
6.     if (σ = +) then
7.        𝓔_{+,i} := 𝓔_{+,i} ∪ {e}.
8.        L := {(e, σ, i)}.
9.     else
10.       for f ∈ E_V cannot be distinguished from e do
11.          𝓔_{−,i} := 𝓔_{−,i} ∪ {f}.
12.          L := {(f, −, i)}.
13.       end for f
14.    end if
15. end if
16.
17. while (L ≠ ∅) do
18.
19.    choose (e, σ, i) ∈ L.
20.    L := L\{(e, σ, i)}.
21.
22.    if (Check_P3(P(e, σ, i)𝓔L) ≠ OK) then return EXIT.
23.    if (Avoid_C4(P(e, σ, i)𝓔L) ≠ OK) then return EXIT.
24.    if (Avoid_cliques(P(e, σ, i)𝓔L) ≠ OK) then return EXIT.
25.
26. end while
27.
28. return OK.
```

Checking Condition $P3$. In subroutine **Check_P3**, for an augmentation $(e, +, i)$ the set of free coordinate directions

$$F := \{j \in \{1, \ldots, d\} \mid e \notin \mathscr{E}_{+,j}\}$$

is computed. If this set only has one element k, then for all $E \in \mathscr{L}(\mathscr{E})$ the condition $e \notin E_k$ must hold because of $\bigcap_{i=1}^{d} E_i = \varnothing$. In this case, e can be fixed in $\mathscr{E}_{-,k}$, and **Check_P3** terminates with the value OK. If there is no free coordinate direction left, then the search space is empty, and the routine terminates with the value EXIT.

Avoiding Induced C_4s. Routine **Avoid_C4** tries to detect edges that can be used for completing an induced C_4 in $(V, \mathscr{E}_{+,i})$, with chords lying in $\mathscr{E}_{-,i}$. Such an edge f is then added to $\mathscr{E}_{+,i}$ or to $\mathscr{E}_{-,i}$, such that this excluded induced subgraph is avoided.

Because this configuration must have been caused by the augmentation (e, σ, i) that was given to **Avoid_C4**, e must either be an edge of the cycle, or a chord. Because f can occur as an edge of the cycle as well as a chord, we have to check a total of four cases. Figure 4 shows the routine in detail.

Avoiding Infeasible Cliques. The subroutine **Avoid_cliques** (see Figure 5) checks whether an edge $e = bc$ that has been added to $\mathscr{E}_{-,i}$ completes one of the following

Figure 4. Routine Avoid_C4.

Call: **Avoid_C4**$(P, (e,\sigma,i)^{in}, \mathscr{E}, L)$
Input: An OPP-n instance $P := (V, w)$, an augmentation $(e,\sigma,i)^{in}$, the search information $\mathscr{E}$, the augmentation list L.
Output: The updated search information $\mathscr{E}$, the updated augmentation list L, and the value EXIT or OK.

$(e,\sigma,i) := (e,\sigma,i)^{in}$.

if $(\sigma = +)$ **then**

  for $f \notin \mathscr{E}_{-,i}$ completes a C_4 in $\mathscr{E}_{+,i}$
    that contains e and has chords in $\mathscr{E}_{-,i}$. **do**
   if $(f \in \mathscr{E}_{+,i})$ **then** **return** EXIT.
   $\mathscr{E}_{-,i} := \mathscr{E}_{-,i} \cup \{f\}$.
   $L := L \cup \{(f,-,i)\}$.
  end for f

  for $f \notin \mathscr{E}_{+,i}$ is chord of a C_4 in $\mathscr{E}_{+,i}$
    that contains e and has its other chord in $\mathscr{E}_{-,i}$. **do**
   if $(f \in \mathscr{E}_{-,i})$ **then** **return** EXIT.
   $\mathscr{E}_{+,i} := \mathscr{E}_{+,i} \cup \{f\}$.
   $L := L \cup \{(f,+,i)\}$.
  end for f

else $(\sigma = -)$

  for $f \notin \mathscr{E}_{-,i}$ completes a C_4 in $\mathscr{E}_{+,i}$ that has e as a chord
    and that has its other chord also in $\mathscr{E}_{-,i}$. **do**
   if $(f \in \mathscr{E}_{+,i})$ **then** **return** EXIT.
   $\mathscr{E}_{-,i} := \mathscr{E}_{-,i} \cup \{f\}$.
   $L := L \cup \{(f,-,i)\}$.
  end for f

  for $f \notin \mathscr{E}_{+,i}$ is chord of a C_4 in $\mathscr{E}_{+,i}$ that has e as
    its other chord. **do**
   if $(f \in \mathscr{E}_{-,i})$ **then** **return** EXIT.
   $\mathscr{E}_{+,i} := \mathscr{E}_{+,i} \cup \{f\}$.
   $L := L \cup \{(f,+,i)\}$.
  end for f

end if

return OK.

Figure 5. Routine Avoid_cliques.

Call: **Avoid_cliques**$(P, (e,\sigma,i)^{in}, \mathscr{E}, L)$
Input: An OPP-n instance (V, w), an augmentation $(e,\sigma,i)^{in}$, the search information $\mathscr{E}$, the augmentation list L.
Output: The updated search information $\mathscr{E}$, the updated augmentation L, and EXIT or OK.

$(bc,\sigma,i) := (e,\sigma,i)^{in}$.

if $(\sigma = +)$ **then** **return** OK.

compute S'_0 as described.
if $(w_i(S'_0 \cup \{b,c\}) > 1)$ **then** **return** EXIT.

if (b and c are indistinguishable) **then**
  if $\exists b' \in V$ with $bb' \in \overline{\mathscr{E}}_{+,i} \cap \overline{\mathscr{E}}_{-,i}$ **then**
   compute $B := \{b,c\} \cup S' \cup X$ as described.
   if $(w_i(B) > 1)$ **then** $\mathscr{E}_{+,i} := \mathscr{E}_{+,i} \cup \{bb'\}$,
     $L := L \cup \{(bb',+,i)\}$.
  end if
end if

for $b' \in V$ with $bb' \in \overline{\mathscr{E}}_{+,i} \cap \overline{\mathscr{E}}_{-,i}$ and $cb' \in \mathscr{E}_{-,i}$ **do**
  compute $B := \{b,c\} \cup S' \cup X$ as described.
  if $(w_i(B) > 1)$ **then** $\mathscr{E}_{+,i} := \mathscr{E}_{+,i} \cup \{bb'\}$,
    $L := L \cup \{(bb',+,i)\}$.
end for b'

for $b' \in V$ with $bb' \in \mathscr{E}_{-,i}$ and $cb' \in \overline{\mathscr{E}}_{+,i} \cap \overline{\mathscr{E}}_{-,i}$ **do**
  compute $B := \{b,c\} \cup S' \cup X$ as described.
  if $(w_i(B) > 1)$ **then** $\mathscr{E}_{+,i} := \mathscr{E}_{+,i} \cup \{cb'\}$, $L := L \cup \{(cb',+,i)\}$.
end for b'

for $b' \in V$ with $bb' \in \mathscr{E}_{-,i}$ and $cb' \in \mathscr{E}_{-,i}$ **do**
  for $c' \in V$ with $bc' \in \mathscr{E}_{-,i}$, $cc' \in \mathscr{E}_{-,i}$ and
    $b'c' \in \overline{\mathscr{E}}_{+,i} \cap \overline{\mathscr{E}}_{-,i}$ **do**
   compute $B := \{b,c\} \cup S' \cup X$ as described.
   if $(w_i(B) > 1)$ **then** $\mathscr{E}_{+,i} := \mathscr{E}_{+,i} \cup \{b'c'\}$,
     $L := L \cup \{(b'c',+,i)\}$.
  end for c'
end for b'

return OK.

configurations:
(1) an i-infeasible clique in $(V, \mathscr{E}_{-,i})$,
(2) an i-infeasible clique in $(V, \overline{\mathscr{E}}_{+,i})$, with edges not in $\mathscr{E}_{-,i}$ being indistinguishable.

As we have seen in §3.2, in the first case the search space is empty. The routine terminates with value EXIT. In the second case, we can find a feasible augmentation by virtue of Lemma 8.

Computing S'_0. We search for a clique in $(V, \mathscr{E}_{-,i})$ that contains $e = bc$ and has large weight. Trivially, the box set of such a clique can only contain b, c, and boxes from

$$S_0 := \{z \in V \mid bz \in \mathscr{E}_{-,i} \wedge cz \in \mathscr{E}_{-,i}\}.$$

Now our approach depends on whether $(V, \mathscr{E}_{-,i})[S_0]$ is a comparability graph. In the positive case, we can use the linear time algorithm from Golumbic (1980) (just like for the test of packing classes) to determine a set of boxes S'_0 that induces a maximal weighted clique in $(V, \mathscr{E}_{-,i})[S_0]$. Then, $\{b,c\} \cup S'_0$ induces a clique in $(V, \mathscr{E}_{-,i})$ that has maximal weight among all cliques containing e.

If, on the other hand, $(V, \mathscr{E}_{-,i})[S_0]$ is not a comparability graph, then we skip the computation of a maximal weighted clique. As a generalization of the CLIQUE problem (problem [GT19] in Garey and Johnson 1979), this problem is NP-hard. Instead, we compute S'_0 by using a greedy strategy. Starting with $S'_0 = \varnothing$, we keep augmenting S'_0 by the box with the largest weight w_i, as long as the property $E_{S'_0} \subseteq \mathscr{E}_{-,i}$ remains valid. The clique induced by $\{b,c\} \cup S'_0$ in $(V, \mathscr{E}_{-,i})$ may be suboptimal.

In both cases, the routine terminates in case of an i-infeasible $S'_0 \cup \{b,c\}$ with the value EXIT.

To determine whether $(V, \mathscr{E}_{-,i})[S_0]$ is a comparability graph, we use the decomposition algorithm from Garey and Johnson (1979) in **Packingclass_test**. In the implementation, it is worthwhile taking into account that in the case $|S_0| \leqslant 4$, the testing for a comparability graph can be omitted. The corresponding induced subgraphs must be comparability graphs because a two-chordless cycle must contain at least five different vertices. In our numerical experiments, this turned out to be a common situation.

Finding an Augmenting Edge by Computing B. We test whether an edge $e' \in \overline{\mathscr{E}}_{+,i} \cap \overline{\mathscr{E}}_{-,i}$ can be fixed. A sufficient condition is the existence of a set $B \subseteq V$ that satisfies the following conditions:

(1) B contains all vertices of e and e'.
(2) All edges in $E_B \backslash \mathscr{E}_{-,i}$ are indistinguishable.
(3) B is i-infeasible.

Because of $P2$, an edge in E_B must be in the ith component graph of the desired packing class. Because this edge must not be in $\mathscr{E}_{-,i}$, it must be indistinguishable from $e' \in E_B \backslash \mathscr{E}_{-,i}$ by virtue of two. In other words, Lemma 8 means that augmentation with indistinguishable edges leads to isomorphic packing classes. This implies the feasibility of augmentation $(e', +, i)$.

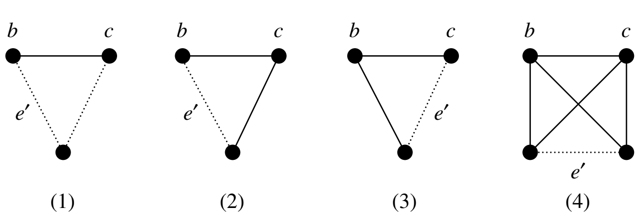
The requirement that the vertices of e lie in B results from the fact that we only search for incomplete excluded configurations that arise from adding e to $\mathscr{E}_{-,i}$.

When identifying edges that are candidates for e', we get four cases for the position of e' relative to $e = bc$ in E_B, as shown in Figure 6. Dotted lines represent the (unfixed) edges in $\overline{\mathscr{E}}_{+,i} \cap \overline{\mathscr{E}}_{-,i}$, while solid lines represent edges in $\mathscr{E}_{-,i}$. The second requirement for B implies that b and c are indistinguishable. Note that after the first resulting augmentation, b and c are indistinguishable with respect to the current search information. Thus, cases (1), (2), (3), and (4) in the figure correspond to lines (8–13), (15–18), (20–23), and (25–30).

Therefore, constructing the set B for a given edge e' is done as follows. Let S be the set of boxes that are adjacent in $(V, \mathscr{E}_{-,i})$ to all vertices of e and e'. Similar to the above construction of S'_0 from S_0, we construct a set of boxes S' from S that induces a clique in $(V, \mathscr{E}_{-,i})$. The comparability graph test is skipped if $(V, \mathscr{E}_{-,i})[S_0]$ has been recognized as a comparability graph: If $S \subseteq S_0$, then $(V, \mathscr{E}_{-,i})[S]$ is an induced subgraph and inherits its property of being a comparability graph.

By adding the vertices of e and e' to S', we get a set that satisfies the first two conditions that B needs to satisfy. e' is the only set in the complete graph on this set that does not belong to $\mathscr{E}_{-,i}$. Now we add boxes that provide edges indistinguishable from e'.

Figure 6. Relative position of e' and $e = bc$.



Let the set X contain the vertices of e'. The indistinguishable boxes for each vertex form a stable set in $(V, \mathscr{E}_{-,i})$, or they induce a clique in this graph. Only in the latter case do we add these boxes to X. With the help of this construction, any edge in the complete graph on $B := \{b, c\} \cup S' \cup X$ is either in the set $\mathscr{E}_{-,i}$, or it is indistinguishable from e'. If this set is i-infeasible, then we fix e' in $\mathscr{E}_{+,i}$ by virtue of Lemma 8.

5. A Tree Search Algorithm for Orthogonal Knapsack Problems

In this section, we elaborate on how the data structure introduced in Fekete and Schepers (1997b, 2004a), the lower bounds described in Fekete and Schepers (1997c, 2004b), and the exact algorithm for the OPP from §3 can be used as building blocks for new exact methods for orthogonal packing problems.

We concentrate on the most difficult problem, the OKP. After a detailed description of the new branch-and-bound approach, the following §6 gives evidence that our algorithm allows it to solve considerably larger instances than previous methods. In particular, we present the first results for three-dimensional instances.

Similar exact algorithms for the SPP and the OBPP are sketched in §7.

5.1. The Framework

For solving the OKP, we have to determine a subset $S \subseteq V$ of boxes that has maximum value among all subsets of boxes fitting into the container. Like Beasley (1985) and Hadjiconstantinou and Christofides (1995), we will prove feasibility of a particular set S by displaying a feasible packing for (V, w, W). For most practical applications, this is of key importance.

In the branch-and-bound algorithms (Beasley 1985, Hadjiconstantinou and Christofides 1995), the iterative choice of a subset and the corresponding packing are treated simultaneously: With each branching step, it is decided whether a particular position in the container is occupied by a particular box type.

In contrast to this, our approach works on two levels. Only after the first level has determined the subset $S \subseteq V$ will the OPP algorithm from §3 try to find a feasible packing. This allows us to use the lower bounds described in our paper (Fekete and Schepers 1997c, 2004b) for excluding most of the first-level search tree without having to consider the particular structure of a packing. Our numerical results show that the second-level search has to be used only in a small fraction of search nodes. Note that the main innovation of our approach lies in this second level; it is to be expected that tuning the outer level (as was done by Caprara and Monaci 2004) yields even better results.

5.2. Branch-and-Bound Methods

We assume that the reader is familiar with the general structure of a branch-and-bound algorithm. (A good description can be found in Nemhauser and Wolsey 1988.) We start by introducing some notation.

We remind the reader of the partitioning of the set V of boxes into classes of boxes with identical size and value, called *box types*:

$$V = \bigcup_{t=1}^{m} T_t.$$

For box type t, we set $n_t := |T_t|$ and denote the elements by

$$T_t =: \{b_{t,1}, \ldots, b_{t,n_t}\}.$$

For ease of notation, we write $w^{(t)}$ instead of $w(b_{t,1})$, and $v^{(t)}$ instead of $v(b_{t,1})$.

In the test instances that we will be dealing with, all sizes of boxes and containers are integers. We denote measures of the container by $W \in \mathbb{N}^d$; when discussing mathematical arguments, we will assume without loss of generality that the container is a unit cube.

5.3. Search Nodes at Level One

The first-level search tree enumerates the subsets $S \subseteq V$ that are candidates for a solution subset for the OKP. Each node N of the search tree corresponds to an OKP instance with the additional constraint that each box type T_t has upper and lower bounds for the number of boxes that are used. These bounds are denoted by $\bar{n}_t^N$ and $\underline{n}_t^N$.

For a search node N, we set

$$\underline{S}^N := \bigcup_{t=1}^{m} \{b_{t,1}, \ldots, b_{t,\underline{n}_t^N}\},$$

and, similarly,

$$\bar{S}^N := \bigcup_{t=1}^{m} \{b_{t,1}, \ldots, b_{t,\bar{n}_t^N}\}.$$

For a partial search tree with root N, we will consider only subsets S that satisfy

$$\mathcal{S}(N) := \{S \subseteq V \mid \underline{S}^N \subseteq S \subseteq \bar{S}^N\}.$$

Thus, for a search node N, the corresponding restricted OKP is given by

$$\begin{aligned} &\text{Maximize} && v(S), \\ &\text{such that} && \text{there is a feasible packing for } (S, w), \\ &&& \underline{S}^N \subseteq S \subseteq \bar{S}^N. \end{aligned} \tag{4}$$

On the root node N_0, we start with the original problem, i.e., $\underline{n}_t^{N_0} = 0$ and $\bar{n}_t^{N_0} = n_t$ for $t \in \{1, \ldots, m\}$. Then, $\underline{S}^{N_0} = \varnothing$ and $\bar{S}^{N_0} = V$.

Enumerating the first-level search tree is done by *best first search*: Each node N is assigned a preliminary local upper bound, given by the minimum of $v(\bar{S}^N)$ and the local upper bound of its parent node. (A better upper bound will be determined while evaluating the partial tree at N.) At each stage, we choose a new node where this local upper bound is maximal.

5.4. Branching

When a subset S has been uniquely determined by the condition $S \in \mathcal{S}(N)$, we have reached a leaf of the first-level search tree. In this case, we have

$$\underline{S}^N = S = \bar{S}^N$$

and

$$\forall\, t \in \{1, \ldots, m\}: \ \underline{n}_t^N = \bar{n}_t^N.$$

Then, problem (4) is an OPP that is solved by the second-level tree search.

Otherwise, we have box types T_t, with $\underline{n}_t^N < \bar{n}_t^N$. We choose the one with largest size $\max_{1 \leqslant i \leqslant d} w_i^{(t)}$ for an arbitrary coordinate direction. By our experience, boxes that are "bulky" in this sense have the biggest influence on the overall solution of the problem.

Now let T_{t^*} be the box type chosen in this way. We branch by splitting $\mathcal{S}(N)$ into subspaces, where the number of boxes in T_{t^*} is constant. For each $\nu \in \{\underline{n}_{t^*}^N, \ldots, \bar{n}_{t^*}^N\}$, we determine a child node N_ν. For this node, we set

$$\underline{n}_t^{N_\nu} := \begin{cases} \nu, & t = t^*, \\ \underline{n}_t^N, & t \neq t^*, \end{cases}$$

and

$$\bar{n}_t^{N_\nu} := \begin{cases} \nu, & t = t^*, \\ \bar{n}_t^N, & t \neq t^*. \end{cases}$$

A different branching strategy builds a binary search tree, where the two children of N each get one half of $\{\underline{n}_{t^*}^N, \ldots, \bar{n}_{t^*}^N\}$ as a range for the number of boxes in T_{t^*}. For technical reasons, we have used the first variant.

5.5. Lower Bounds

On each node N, the container is filled with boxes from $\bar{S}^N$ by using the following greedy heuristic. The best objective value of the OKP for any of these feasible solutions is stored in v_{lb}. Trivially, v_{lb} is a lower bound for the optimal value of the OKP.

In our heuristic, we build a sequence of packings, where each lower coordinate of a box equals zero (i.e., the boundary of the container), or the upper coordinate of a preceding box. These positions are called *placement points*. Placement points are maintained in a list that is initialized by the container origin. At each step, a placement point is removed from the list as we try to use it for placing another box. Following a given ordering, we use the first box type that fits at the chosen placement point without overlapping any of the boxes that are already packed into the container. In case of success, we compute the new placement points and add them to the list. This step is repeated until the list is empty, or all boxes have been packed.

This construction of a packing is repeated for several orderings of box types. In the first round, we use the order of decreasing value. Following rounds use a random weighting of values before sorting; weights are chosen from a uniform distribution on $[0, 1]$.

In our implementation, 50 iterations of this heuristic are performed at the root, and 10 iterations at all other nodes.

5.6. Upper Bounds

The upper bound v_{ub}^N refers to the set of boxes from $\mathcal{S}(N)$. As we showed in Fekete and Schepers (1997c, 2004b), for any conservative scale w' for $(\bar{S}^N, w)$, a relaxation of (4) is given by

$$\begin{aligned} \text{Maximize}\quad & v(S), \\ \text{such that}\quad & \sum_{b\in S} \otimes w'(b) \leqslant 1, \\ & \underline{S}^N \subseteq S \subseteq \bar{S}^N, \end{aligned} \tag{5}$$

where $\otimes w'(b) := \prod_{i=1}^d w'_i(b)$ denotes the volume of the modified box $w'(b)$. To avoid technical difficulties, we only consider conservative scales that are constant for each box type. For the benefit of later generalizations, we formulate problem (5) explicitly as a restricted one-dimensional knapsack problem:

$$\begin{aligned} \text{Maximize}\quad & \sum_{t=1}^m v(b_{t,1})\xi_t, \\ \text{such that}\quad & \sum_{t=1}^m \otimes w'(b_{t,1})\xi_t \leqslant 1, \\ & \underline{n} \leqslant \xi \leqslant \bar{n}, \\ & \xi \text{ integer}. \end{aligned} \tag{6}$$

A problem of this type can be solved by the routine MTB2 from Martello and Toth (1990, Appendix A.3.1). This transforms the restricted knapsack problem into a 0-1 knapsack problem to which the algorithm of Martello and Toth (1990, pp. 61ff) is applied.

In our implementation, we use as an upper bound the minimum of the optimal values of the relaxations (5) for the conservative scales

$$w' \in \{(w_1, \dots, u^{(k)} \circ w_i, \dots, w_d) \mid i = 1, \dots, d,\ k = 1, 2, 3, 4\}$$

from our paper (Fekete and Schepers 1997c, 2004b).

5.7. Removal of Partial Search Trees

We can stop the search on the current search tree if one of the following conditions is satisfied:

(1) $v_{ub}^N \leqslant v_{lb}$.
(2) $\bar{S}^N$ fits into the container.
(3) $\underline{S}^N$ does not fit into the container.

The first stop criterion is used in any branch-and-bound procedure. In this case, the currently best solution cannot be improved on the current search tree.

In the second case, $\bar{S}^N \in \mathcal{S}(N)$ is a best feasible solution in $\mathcal{S}(N)$. Because we are always trying to pack all of $\bar{S}^N$ when updating the lower bound v_{lb}, this condition is checked when performing the update.

In the third case, $\underline{S}^N \subseteq S$ implies that no set $S \in \mathcal{S}(N)$ can be packed into the container, so $\mathcal{S}(N)$ cannot contain a feasible solution. This means that we have to solve another OPP.

5.8. Solving Orthogonal Packing Problems

To solve the OPPs that occur on the leaves of the search tree and when checking the stop criterion "$\underline{S}^N$ does not fit into the container," we use the following strategy:

First, we try to use the volume criterion for a selection of conservative scales. Other than the original weight function w, we use the conservative scales

$$w' \in \{(w_1, \dots, u^{(k)} \circ w_i, \dots, w_d) \mid i = 1, \dots, d,\ k = 1, \dots, W_i/2\}.$$

If this does not produce a (negative) result, we try to find a packing pattern by 10 iterations of our search heuristic. If this fails as well, we use our algorithm from §3 to decide the OPP.

5.9. Problem Reduction

There are several ways to decrease the gap between the bounds $\underline{n}_t^N$ and $\bar{n}_t^N$ on a search node. These rules are based on corresponding reduction tests of Beasley (1985). If the areas of boxes and container are used, we generalize the tests from two to d dimensions. By using conservative scales, we get a generalization of these tests, with markedly increased efficiency.

We start with the rule *Free Value*, which remains unchanged. An optimal solution S can have at most value v_{ub}^N. Because $\underline{S} \subseteq S$, further boxes from T_t in S can contribute at most a value of $v_{ub}^N - v(\underline{S})$. Because each of these boxes has value $v^{(t)}$, we set

$$\bar{n}_t := \min\left\{\bar{n}_t, \underline{n}_t + \left\lfloor \frac{v_{ub}^N - v(\underline{S})}{v^{(t)}} \right\rfloor\right\}. \tag{7}$$

A similar argument, used on volumes, is the basis for Beasley's reduction test *Free Area*. The volume used by $\underline{S}$ is at least as big as the sum of the volumes of the individual boxes in $\underline{S}$. Further boxes from T_t can use at most the volume of the container, reduced by this amount. Because each of these boxes uses a volume of $\otimes w(t)$, we can use the following update for $t \in \{1, \dots, m\}$:

$$\bar{n}_t := \min\left\{\bar{n}_t, \underline{n}_t + \left\lfloor \frac{1 - \otimes w(\underline{S})}{\otimes w(t)} \right\rfloor\right\}. \tag{8}$$

With the help of Corollary 8 from our paper (Fekete and Schepers 2004b), we can generalize *Free Area* by replacing w in (8) by an arbitrary conservative scale w' for (V, w)

that is constant on T_t. In our implementation, we use w and the conservative scales

$$w' \in \{(w_1, \ldots, u^{(k)} \circ w_i, \ldots, w_d) \mid i = 1, \ldots, d, k = 1, \ldots, W_i/2\}.$$

To allow for further possible improvement of a bound $\underline{n}_{t^*}^N$, $t^* \in \{1, \ldots, m\}$, we expand the relaxation (6) by the additional constraint $\xi_{t^*} = \underline{n}_{t^*}$. The optimal values of the resulting knapsack problems are upper bounds for the value of those solutions $S \in \mathcal{S}(N)$ that contain precisely $\underline{n}_{t^*}$ boxes from T_{t^*}. If the minimum of these bounds does not exceed v_{lb}, a solution $S \in \mathcal{S}(N)$ with a better value than the current best must contain more than $\underline{n}_{t^*}^N$ boxes from T_{t^*}. In this case, we can increment $\underline{n}_{t^*}^N$. This test is repeated for each box type t until no bound can be improved. If in this process we get $\underline{n}_t^N > \bar{n}_t^N$, then the search on the partial search tree with root N can be stopped. Thus, we have derived a generalization of the reduction test *Area Program* with the help of conservative scales.

6. Computational Results

The above OKP procedure has been implemented in C++ and was tested on a Sun workstation with Ultra SPARC processors (175 MHz), using the compiler gcc. To allow for a wider range of comparisons with other two-dimensional efforts, we also tested an implementation on a PC with a Pentium IV processor (2.8 GHz) with 1 GB memory using g++3.2.

6.1. Results for Benchmark Instances from the Literature

The only benchmark instances for the OKP that have been documented in the literature can be found in the articles by Beasley (1985) and Hadjiconstantinou and Christofides (1995). These are restricted to the two-dimensional case. We ran our algorithm on all of these instances that were available. Like Caprara and Monaci (2004), we also use a number of other instances that were originally designed for guillotine-cut instances.

The 12 instances *beasley1* through *beasley12* are taken from Beasley's OR library (see Beasley 1990). They can be found on the Internet at http://mscmga.ms.ic.ac.uk/jeb/orlib/ngcutinfo.html. The data for *hadchr3* and *hadchr11* are given in Hadjiconstantinou and Christofides (1995).

Tables 1 and 2 show our results for these OKP-2 instances. For all instances, we found an optimal solution in at most 0.02 seconds. The small number of OKP search nodes (at most 65) as well as OPP search nodes (at most 294) shows the high efficiency of the rules for reducing the search tree. It is also remarkable that on less than a quarter of the search nodes, the enumeration procedure for the OPP had to be used. The majority of reductions resulted from transformed volumes.

Instances *wang20* and *chrwhi62* are considerably larger. They were taken from Wang (1983) and Christofides and Whitlock (1977) and were originally designed for testing the efficiency of guillotine-cut algorithms, as were the next three sets: Instances 3 through CHL5 are taken from the benchmark sets by Hifi, which can be found at ftp://panoramix.univ-paris1.fr/pub/CERMSEM/hifi/2Dcutting/.

The sets *cgcut* and *gcut* are also guillotine-cut type instances and can be found at the OR library. Finally, we created five new OKP instances *okp* that are listed in detail in Table 3. They are random instances generated in the same way as *beasley 1–12* after applying initial reduction.

A detailed listing of our results for these OKP-2 instances can be found in Table 1. The first column lists the names of the instances; the second shows the size of the container, followed by the number of different box types and the total number of boxes. The fifth column shows the number of nodes in the outer search tree, followed by the total number of calls to the inner search tree, i.e., the times an OPP had to be re-solved. The last two columns show the number of boxes in an optimal solution, and the optimal value. (Instance *gcut13* is still unsolved; our lower bound corresponds to the best solution found so far.) Results are shown in Table 2, where the first column lists the instance names and the second column shows the run time on a PC with a Pentium IV processor. Columns 3 and 4 give the run times as reported by Beasley (1985) on a CYBER 855 and by Hadjiconstantinou and Christofides (1995) on a CDC 7600. In 2004, Caprara and Monaci give run times for four different algorithms. None of these algorithms dominates all the others; the best of them (called A_3) uses a clever hybrid strategy for checking feasibility during branching.

The comparison in Table 2 should be considered with some care because different computers with different compilers were used for the tests. Some indication for the relative performance of the different machines can be found at http://www.netlib.org/benchmark/performance.ps, where the results of the Linpack100 benchmark are presented (see Dongarra 2004). According to these results, a CDC 7600 manages 120 Mflop/s, a CYBER 875 (Cyber 855 does not appear on the list) gets 480 Mflop/s, a Sun Ultra SPARC achieves 7,000 Mflop/s, an Intel Pentium III (750 MHz) 13,800 Mflop/s, while an Intel Pentium IV (2.8 GHz) manages 131,700 Mflop/s. Note that these speeds may not be the same for other applications. Furthermore, there is always a certain amount of chance involved when comparing branch-and-bound procedures on individual instances.

Despite these difficulties in comparison, it is clear that our new method constitutes significant progress. One indication is the fact that the ratio of running times between large and small instances is smaller by several orders of magnitude: As opposed to our two-level algorithm, the search trees in the procedures by Beasley (1985) and by Hadjiconstantinou and Christofides (1995) appear to be reaching the threshold of exponential growth for some of the bigger instances. After 800 seconds, the procedure by Hadjiconstantinou and Christofides timed out on instances

Table 1. Two-dimensional benchmark instances from previous literature.

Problem	Container size	Box types	No. of boxes	OKP nodes	OPP calls	OPP nodes	Opt. boxes	Opt. solution
beasley1	(10, 10)	5	10	19	1	1	5	164
beasley2	(10, 10)	7	17	5	0	0	5	230
beasley3	(10, 10)	10	21	25	6	36	7	247
beasley4	(15, 10)	5	7	1	0	0	6	268
beasley5	(15, 10)	7	14	1	0	0	6	358
beasley6	(15, 10)	10	15	15	5	5	7	289
beasley7	(20, 20)	5	8	0	0	0	8	430
beasley8	(20, 20)	7	13	53	23	301	8	834
beasley9	(20, 20)	10	18	3	0	0	11	924
beasley10	(30, 30)	5	13	1	0	0	6	1,452
beasley11	(30, 30)	7	15	36	10	16	9	1,688
beasley12	(30, 30)	10	22	48	14	105	9	1,865
hadchr3	(30, 30)	7	7	1	0	0	5	1,178
hadchr7	(30, 30)	10	22	48	14	105	9	1,865
hadchr8	(40, 40)	10	10	7	0	0	6	2,517
hadchr11	(30, 30)	15	15	30	1	1	5	1,270
hadchr12	(40, 40)	15	15	5	0	0	7	2,949
wang20	(70, 40)	20	42	794	176	1,003	8	2,726
chrwhi62	(40, 70)	20	62	356	102	7,991	10	1,860
3	(40, 70)	20	62	356	102	7,991	10	1,860
3s	(40, 70)	20	62	757	166	3,050	8	2,726
A1	(50, 60)	20	62	935	254	19,283	11	2,020
A1s	(50, 60)	20	62	4,291	504	8,156	7	2,956
A2	(60, 60)	20	53	267	70	35,747	11	2,615
A2s	(60, 60)	20	53	8,598	2,365	143,002	8	3,535
CHL2	(62, 55)	10	19	688	317	225,011	9	2,326
CHL2s	(62, 55)	10	19	1,419	557	158,450	10	3,336
CHL3	(157, 121)	15	35	0	0	0	35	5,283
CHL3s	(157, 121)	15	35	0	0	0	35	7,402
CHL4	(207, 231)	15	27	0	0	0	27	8,998
CHL4s	(207, 231)	15	27	0	0	0	27	13,932
CHL5	(30, 20)	10	18	363	194	57,115	11	589
cgcut1	(15, 10)	7	16	14	1	1	8	244
cgcut2	(40, 70)	10	23				12	2,892
cgcut3	(40, 70)	20	62	356	102	7,991	10	1,860
gcut01	(250, 250)	10	10	33	0	0	3	48,368
gcut02	(250, 250)	20	20	519	51	78	6	59,798
gcut03	(250, 250)	30	30	2,234	235	742	6	61,275
gcut04	(250, 250)	50	50	72,159	18,316	145,057	4	61,380
gcut05	(500, 500)	10	10	52	13	13	5	195,582
gcut06	(500, 500)	20	20	278	22	22	4	236,305
gcut07	(500, 500)	30	30	852	124	152	4	240,143
gcut08	(500, 500)	50	50	55,485	9,037	15,970	4	245,758
gcut09	(1,000, 1,000)	10	10	12	2	8	5	939,600
gcut10	(1,000, 1,000)	20	20	335	31	40	5	937,349
gcut11	(1,000, 1,000)	30	30	1,616	212	463	6	969,709
gcut12	(1,000, 1,000)	50	50	8,178	593	1,236	5	979,521
gcut13	(3,000, 3,000)	32	32					$\geqslant$8,622,498 $\leqslant$9,000,000
okp1	(100, 100)	15	50	3,244	661	35,523	11	27,718
okp2	(100, 100)	30	30	23,626	7,310	8,721	11	22,502
okp3	(100, 100)	30	30	8,233	816	921	11	24,019
okp4	(100, 100)	33	61	1,458	15	50	10	32,893
okp5	(100, 100)	29	97	5,733	643	13,600	8	27,923

beasley12, *hadchr8*, and *hadchr11* without finding a solution. The comparison with Caprara and Monaci (2004) is less conclusive: Both implementations fare pretty well on medium-sized instances, with different behavior for large instances. (Comparing a previous implementation of our algorithm with A_3, Caprara and Monaci (2004, p. 7) concluded that "…the algorithm of [Fekete and Schepers] appears to be more stable… .") This behavior may also be

Table 2. Run times of our implementation, compared to other methods.

Problem	Time/s	B85	HC95	CM04 A_0	CM04 A_1	CM04 A_2	CM04 A_3
beasley1	<0.01	0.9					
beasley2	<0.01	4.0					
beasley3	<0.01	10.5					
beasley4	<0.01	0.1	0.04				
beasley5	<0.01	0.4					
beasley6	<0.01	55.2	45.20				
beasley7	<0.01	0.5	0.04				
beasley8	0.02	218.6					
beasley9	<0.01	18.3	5.20				
beasley10	<0.01	0.9					
beasley11	<0.01	79.1					
beasley12	0.02	229.0	>800				
hadchr3	<0.01		532				
hadchr7	0.01		>800				
hadchr8	<0.01		>800				
hadchr11	<0.01		>800				
hadchr12	<0.01		65.2				
wang20	0.67			6.75	6.31	17.84	2.72
chrwhi62	0.54						
3	0.54						
3s	0.46						
A1	1.12						
A1s	1.51						
A2	1.62						
A2s	8.35						
CHL2	10.36						
CHL2s	6.84						
CHL3	<0.01						
CHL3s	<0.01						
CHL4	<0.01						
CHL4s	<0.01						
CHL5	4.66						
cgcut1	<0.01			0.30	1.47	1.46	1.46
cgcut2	>1,800			>1,800	>1,800	533.45	531.93
cgcut3	0.54			23.76	23.68	4.59	4.58
gcut1	0.01			0.00	0.00	0.01	0.01
gcut2	0.47			0.52	0.19	25.75	0.22
gcut3	4.34			>1,800	2.16	276.37	3.24
gcut4	195.62			>1,800	346.99	>1,800	376.52
gcut5	0.02			0.00	0.50	0.03	0.50
gcut6	0.38			0.06	0.09	9.71	0.12
gcut7	2.24			1.31	0.63	354.50	1.07
gcut8	253.54			1,202.09	136.71	>1,800	168.50
gcut9	0.01			0.01	0.09	0.05	0.08
gcut10	0.67			0.01	0.13	6.49	0.14
gcut11	8.82			16.72	14.76	>1,800	16.30
gcut12	109.81			63.45	16.85	>1,800	25.39
gcut13	>1,800			>1,800	>1,800	>1,800	>1,800
okp1	10.82			24.06	25.46	72.20	35.84
okp2	20.25			>1,800	>1,800	1,535.95	1,559.00
okp3	5.98			21.36	1.91	465.57	10.63
okp4	2.87			40.40	2.13	0.85	4.05
okp5	11.78			40.14	>1,800	513.06	488.27

Note. The columns B85, HC95, and CM04 show the run times as reported in Beasley (1985), Hadjiconstantinou and Christofides (1995), and Caprara and Monaci (2004).

Table 3. The new problem instances okp1–okp5.

Problem okp1: container = (100, 100), 15 box types (50 boxes)
size = [(4, 90), (22, 21), (22, 80), (1, 88), (6, 40), (100, 9), (46, 14), (10, 96), (70, 27), (57, 18), (10, 84), (100, 1), (2, 41), (36, 63), (51, 24)]
value = [838, 521, 4,735, 181, 706, 2,538, 1,349, 1,685, 5,336, 1,775, 1,131, 129, 179, 6,668, 3,551]
$n = [5, 2, 3, 5, 5, 5, 3, 1, 3, 1, 1, 5, 5, 2, 4]$

Problem okp2: container = (100,100), 30 box types (30 boxes)
size = [(8, 81), (5, 76), (42, 19), (6, 80), (41, 48), (6, 86), (58, 20), (99, 3), (9, 52), (100, 14), (7, 53), (24, 54), (23, 77), (42, 32), (17, 30), (11, 90), (26, 65), (11, 84), (100, 11), (29, 81), (10, 64), (25, 48), (17, 93), (77, 31), (3, 71), (89, 9), (1, 6), (12, 99), (33, 72), (21, 26)]
value = [953, 389, 1,668, 676, 3,580, 1,416, 3,166, 537, 1,176, 3,434, 676, 1,408, 2,362, 4,031, 1,152, 2,255, 3,570, 1,913, 1,552, 4,559, 713, 1,279, 3,989, 4,850, 299, 1,577, 12, 2,116, 2,932, 1,214]
$n_j = 1, \quad j \in \{1, \ldots, 30\}$

Problem okp3: container = (100, 100), 30 box types (30 boxes)
size = [(3, 98), (34, 36), (100, 6), (49, 26), (14, 56), (100, 3), (10, 90), (23, 95), (10, 97), (50, 47), (41, 45), (13, 12), (19, 68), (50, 46), (23, 70), (28, 82), (12, 65), (9, 86), (21, 96), (19, 64), (21, 75), (45, 26), (19, 77), (5, 84), (16, 21), (23, 69), (5, 89), (22, 63), (41, 6), (76, 30)]
value = [756, 2,712, 1,633, 2,332, 2,187, 470, 1,569, 4,947, 2,757, 4,274, 4,347, 396, 3,866, 5,447, 2,904, 6,032, 1,799, 929, 5,186, 2,120, 1,629, 2,059, 2,583, 953, 1,000, 2,900, 1,102, 2,234, 458, 5,458]
$n_j = 1, \quad j \in \{1, \ldots, 30\}$

Problem okp4: container = (100, 100), 33 box types (61 boxes)
size = [(48, 48), (6, 85), (100, 14), (17, 85), (69, 20), (12, 72), (5, 48), (1, 97), (66, 36), (15, 53), (29, 80), (19, 77), (97, 7), (7, 57), (63, 37), (71, 14), (3, 76), (34, 54), (5, 91), (14, 87), (62, 28), (6, 7), (20, 71), (92, 7), (10, 77), (99, 4), (14, 44), (100, 2), (56, 40), (86, 14), (22, 93), (13, 99), (7, 76)]
value = [5,145, 874, 2,924, 3,182, 2,862, 1,224, 531, 249, 6,601, 1,005, 6,228, 3,362, 907, 473, 6,137, 1,556, 313, 4,123, 581, 1,999, 5,004, 2,040, 3,143, 795, 1,460, 841, 1,107, 280, 5,898, 2,096, 4,411, 3,456, 1,406]
$n = [1, 2, 1, 1, 1, 1, 3, 3, 2, 1, 3, 1, 1, 2, 2, 1, 3, 1, 2, 1, 3, 3, 1, 1, 2, 3, 2, 3, 2, 1, 1, 3, 3]$

Problem okp5: container = (100, 100), 29 box types (97 boxes)
size = [(8, 81), (5, 76), (42, 19), (6, 80), (41, 48), (6, 86), (58, 20), (99, 3), (9, 52), (100, 14), (7, 53), (24, 54), (23, 77), (42, 32), (17, 30), (11, 90), (26, 65), (11, 84), (100, 11), (29, 81), (10, 64), (25, 48), (17, 93), (77, 31), (3, 71), (89, 9), (1, 6), (12, 99), (21, 26)]
value = [953, 389, 1,668, 676, 3,580, 1,416, 3,166, 537, 1,176, 3,434, 676, 1,408, 2,362, 4,031, 1,152, 2,255, 3,570, 1,913, 1,552, 4,559, 713, 1,279, 3,989, 4,850, 299, 1,577, 12, 2,116, 1,214]
$n = [3, 4, 4, 4, 1, 5, 5, 5, 5, 4, 5, 1, 1, 5, 5, 4, 2, 3, 1, 1, 2, 1, 4, 1, 5, 4, 5, 2, 5]$

the result of some differences in branching strategies, which can always turn out differently on individual instances. It should be noted that the main basis for the success of our method is the underlying mathematical characterization, and tuning of branching strategies and bounds can be expected to provide further progress. Promising may also be a combination of the first-level strategy of Caprara and Monaci (2004) with our second-level strategy.

Table 4. Random generation of OKP-2 test instances.

Class of box types	$w_1(x)$ evenly distributed on	$w_2(x)$ evenly distributed on
1 (bulky in 2)	[1, 50]	[75, 100]
2 (bulky in 1)	[75, 100]	[1, 50]
3 (large)	[50, 100]	[50, 100]
4 (small)	[1, 50]	[1, 50]

	Classes of box types (%)			
Instance type	1	2	3	4
I	20	20	20	40
II	15	15	15	55
III	10	10	10	70

At this stage, an instance like *cgcut13* is still out of reach, even though we were able to improve the best-known solution to 8,622,498, from 8,408,316 in Caprara and Monaci (2004) with an upper bound of 9,000,000, leaving a gap of about 4%. It should be interesting to develop long-running, special-purpose exact algorithms, just like Applegate et al. (1998) did for the traveling salesman problem.

6.2. Generating New Test Instances for 2D and 3D

To get a broader test basis, and also include the three-dimensional case, we generated 300 new test instances. We followed the method described in Martello and Vigo (1998) and Martello et al. (2000).

Our test instances are characterized by three parameters:
(1) type of the instance (I, II, III) (see Tables 4 and 5),
(2) number m of box types, and
(3) number ν of boxes for each box type.

Each of the instances consists of a container of size 100 in each coordinate direction, and m box types, which are obtained as follows:

There are four (OKP-2) or five (OKP-3) classes of box types. The type of the instance determines the probability

Table 5. Random generation of OKP-3 test instances.

Class of box types	$w_1(x)$ evenly distributed on	$w_2(x)$ evenly distributed on	$w_3(x)$ evenly distributed on
1 (bulky in 2, 3)	[1, 50]	[75, 100]	[75, 100]
2 (bulky in 1, 3)	[75, 100]	[1, 50]	[75, 100]
3 (bulky in 1, 2)	[75, 100]	[75, 100]	[1, 50]
4 (large)	[50, 100]	[50, 100]	[50, 100]
5 (small)	[1, 50]	[1, 50]	[1, 50]

	Class of box types (%)				
Instance type	1	2	3	4	5
I	20	20	20	20	20
II	15	15	15	15	40
III	10	10	10	10	60

Table 6. Results for randomly generated OKP-2 instances.

				No. of OKP nodes			No. of OPP nodes		
Class	m	$\lvert V\rvert$	Solved (out of 10)	Min	Avg	Max	Min	Avg	Max
I	20	20	10	5	57	174	0	8	23
	30	30	10	19	307	914	0	158	969
	40	40	9	40	933	3,826	0	431	2,411
	20	60	9	31	231	677	1	287	1,018
	20	80	8	82	336	943	234	147,864	727,719
II	20	20	10	6	139	1,038	0	305	2,699
	30	30	10	28	548	1,568	0	2,873	26,866
	40	40	8	32	5,062	28,754	2	14,910	84,975
	20	60	7	36	297	571	5	237,144	1,633,573
	20	80	6	62	536	1,110	83	168,530	280,688
III	20	20	10	3	117	516	0	299	1,169
	30	30	10	82	737	1,860	3	10,588	53,510
	40	40	9	342	3,865	10,655	745	62,065	416,200
	20	60	8	31	1,006	4,064	3	345,130	1,174,938
	20	80	2	96	196	296	241	85,729	171,218

of each new box type T_t to belong to one of these classes. We use the distributions shown in Tables 4 and 5.

Depending on its class, the sizes of a box type are generated randomly, according to the distributions in Tables 4 and 5. We round up to integer values. To get the value of a box type, the volume is multiplied with a random number from $\{1, 2, 3\}$. The number of boxes in a new box type is determined by the parameter ν, independent of t.

In this manner, we generated (for two as well as for three dimensions) 10 OKP instances for each of the three instance types and each of the five parameter combinations:

$$(m, \nu) \in \{(20, 1), (30, 1), (40, 1), (20, 3), (20, 4)\}.$$

6.3. Results for New Test Instances

Tables 6, 7, 8, and 9 show the results for test runs on two- and three-dimensional instances. For 10 test instances of any combination of parameters, we show how many of these instances we could solve within a time limit of 1,000 seconds on a Sun Ultra SPARC with 175 MHz. From the solved instances, we show the minimum (Min), the average (Avg), and the maximum (Max) of the number of OKP and OPP nodes, as well as the resulting run times.

It is evident that the difficulty grows with the percentage of "small" boxes. This is not very surprising because these boxes do not restrict the possibilities for the rest of a selected subset as much as large or bulky boxes do.

The large difference in difficulty for instances with identical parameterization does not arise from our method of generation instances, but is characteristic for instances of hard combinatorial optimization problems. This effect has been known even for one-dimensional packing problems, which have a much simpler structure. Because of this spread, the number of nodes and run times are only significant for combinations of parameters where most instances could be solved.

For the OKP-2 with $m \leqslant 40$ and $|V| \leqslant 40$, we could find an optimal solution in tolerable run time for almost all instances. For 60 and more boxes, Classes II and III started to have higher numbers of instances that could not be solved within the time limit. Only for instances with 80 boxes and about 70% of small boxes, our algorithm seemed to reach its limits for the current implementation.

Even when taking into account that classes of three-dimensional instances vary more with respect to the percentage of small boxes than those for two dimensions, it is remarkable that this percentage makes a huge difference with respect to the difficulty of the resulting instances. For an average of 20% of small boxes (Class I), all instances (with the exception of a single one with $|V| = 80$) could be solved. For an average of 40% of small boxes (Class II),

Table 7. Run times for randomly generated OKP-2 instances using a Sun Ultra SPARC with 175 MHz.

				Run time/s		
Class	m	$\lvert V\rvert$	Solved (out of 10)	Min	Avg	Max
I	20	20	10	0.06	0.56	1.26
	30	30	10	0.29	4.48	13.59
	40	40	9	1.31	22.02	76.30
	20	60	9	0.43	2.35	5.95
	20	80	8	0.99	62.46	243.41
II	20	20	10	0.06	2.18	17.69
	30	30	10	0.36	10.64	39.59
	40	40	8	0.55	51.12	152.46
	20	60	7	0.45	95.44	640.47
	20	80	6	1.86	112.89	267.90
III	20	20	10	0.08	1.48	5.77
	30	30	10	1.07	17.67	53.00
	40	40	9	6.66	103.10	313.91
	20	60	8	0.36	191.98	719.67
	20	80	2	2.18	34.52	66.86

Table 8. Results for randomly generated OKP-3 instances.

Class	m	$\|V\|$	Solved (out of 10)	No. of OKP nodes Min	Avg	Max	No. of OPP nodes Min	Avg	Max
I	20	20	10	1	73	352	0	22	82
	30	30	10	11	276	1,190	1	59	291
	40	40	10	73	953	2,848	5	2,684	20,975
	20	60	10	20	541	2,961	3	19,896	198,091
	20	80	9	42	414	1,511	14	145	399
II	20	20	10	11	75	328	1	35	166
	30	30	10	5	327	972	0	6,579	62,827
	40	40	8	59	2,197	13,064	20	85,465	671,934
	20	60	5	1	292	719	0	232	912
	20	80	3	142	149	161	23	46	65
III	20	20	10	5	57	138	0	4,433	36,747
	30	30	6	1	859	2,250	1	3,794	10,063
	40	40	3	17	652	1,715	7	1,326	3,885
	20	60	3	51	3,728	10,842	27	55,164	165,276
	20	80	1	73	73	73	38	38	38

our method works well, at least for instances with $m \leqslant 40$, $|V| \leqslant 40$. If the percentage of small boxes rises to 60% in Class III, then even for $m = 30$, $|V| = 30$, our program does not find an optimal solution for a large number of instances.

Summarizing, we can say that our new method has greatly increased the size of instances that are practically solvable. In particular, the size of the container is no longer a limiting factor. It should be noted that even for three-dimensional instances with $m = 20$, the 0-1 programs following the approach by Beasley (1985) and Hadjiconstantinou and Christofides (1995) contain several 100,000 variables, even making the generous assumption of a grid reduction to 10%.

Table 9. Run times for randomly generated OKP-3 instances on a Sun Ultra SPARC with 175 MHz, timeout after 1,000s.

Class	m	$\|V\|$	Solved (out of 10)	Run time/s Min	Avg	Max
I	20	20	10	0.06	1.63	7.76
	30	30	10	0.36	9.15	43.58
	40	40	10	2.66	44.99	121.96
	20	60	10	0.50	18.33	125.76
	20	80	9	0.67	10.76	37.04
II	20	20	10	0.26	1.76	6.92
	30	30	10	0.37	18.94	81.69
	40	40	8	2.26	133.48	845.70
	20	60	5	0.28	12.00	38.94
	20	80	3	4.13	5.43	6.95
III	20	20	10	0.29	4.73	21.69
	30	30	6	0.26	35.69	101.61
	40	40	3	2.01	29.66	78.53
	20	60	3	1.21	211.63	607.83
	20	80	1	2.06	2.06	2.06

6.4. A New Library of Benchmark Instances

We are in the process of setting up a new library for multidimensional packing problems, called PackLib[2] (Fekete and van der Veen 2007). The idea is to have one place where benchmark instances, results, and solution history can be found. For this purpose, we are using a universal XML-format that allows inclusion of all this information. We provide parsers for conversion directly into C formats, and converters for all standard data formats. Results include visualization of solutions by drawings of the feasible packings. Finally, we hope to provide a number of algorithms at the website. Interested researchers are encouraged to contact the first author.

7. Solving Other Types of Packing Problems

7.1. Strip-Packing Problems

In an exact SPP procedure, we start with a heuristic for generating a packing; its "height" is used as an upper bound. A lower bound $\underline{h}$ can be obtained with the help of the methods described in our paper (Fekete and Schepers 1997c, 2004b). If there is a gap between these bounds, we have to use enumeration. Because the OPP is the decision version of the SPP for a fixed objective value, an obvious approach would be binary search in combination with the OPP algorithm from §3.

A more efficient method can be obtained by observing that any OPP node that did not find a solution for height h cannot possibly find a solution for height $h' < h$. Thus, we can solve the SPP with the help of a modified version of our OPP routine.

For finitely many boxes, there are only finitely many possibilities for the minimal height of a packing. The set H of these values can be determined by using the method from Christofides and Whitlock (1977) for computing normalized coordinates.

The height of the packing obtained by the heuristic is stored under h. The variable h' is initialized with the largest value from H below h. We start the OPP tree search for the container with height h'. If the algorithm finds a feasible packing, then h is updated to the value h', and h' is replaced by the next smaller value of H. Now the OPP search is done for container height h'. As noted above, no search node that was dismissed before has to be considered again. The search is performed until all search nodes have been checked, or h reaches the value $\underline{h}$ of the lower bound.

7.2. Orthogonal Bin-Packing Problems

The basic scheme of our exact method follows the outline by Martello and Vigo (1998) and Martello et al. (2000):

Within a branch-and-bound framework, a packing (for a number of containers) is produced iteratively. A list L maintains all containers that are used. In the beginning, L is empty. At each branching step, a box b is either assigned to a container C in L, or a new container is generated for b and added to L. The crucial step is to check whether a container C can hold all boxes that are assigned to it.

We get upper bounds by packing the unassigned boxes heuristically. Our new suggestions concern the other steps of the approach, which cause the largest computational effort:

(1) computing lower bounds, and
(2) solving the resulting OPPs.

The improvement of the lower bounds from Martello and Vigo (1998) and Martello et al. (2000) have already been discussed in our paper (Fekete and Schepers 1997c).

In Martello and Vigo (1998), the resulting OPPs are enumerated by using the method of Hadjiconstantinou and Christofides (1995). For solving the three-dimensional OPPs in Martello et al. (2000), there is a special enumeration scheme using the principle of placement points described in §5.1. As discussed in our paper Fekete and Schepers (1997b, 2004a), we get a drastic improvement by using our method from §3, which is based on packing classes.

8. Conclusion

In this paper, we have shown that higher-dimensional packing problems of considerable size can be solved to optimality in reasonable time by making use of a structural characterization of feasible packings. Further progress may be achieved by refined lower bounds and by using a more sophisticated outer tree search, as in the recent paper by Caprara and Monaci (2004). Currently, we are working on a more advanced implementation, motivated by ongoing research on reconfigurable computing. We expect this work to lead to progress for other problem variants.

Acknowledgments

The authors thank Eleni Hadjiconstantinou, who provided the data for instances *hadchr8* and *hadchr12*. They also thank Alberto Caprara for some interesting discussions. Two anonymous referees helped with a number of useful comments that improved the overall presentation of this paper. A previous extended abstract version of this paper appears in *Algorithms—ESA'97* (Fekete and Schepers 1997a). Jörg Schepers was supported by the German Federal Ministry of Education, Science, Research, and Technology (BMBF, Förderkennzeichen 01 IR 411 C7).

References

Applegate, D., R. Bixby, V. Chvátal, W. Cook. 1998. On the solution of traveling salesman problems. *Documenta Mathematica J. Deutschen Mathematiker-Vereinigung* **ICM III** 645–656.

Arenales, M., R. Morabito. 1995. An AND/OR-graph approach to the solution of two-dimensional nonguillotine cutting problems. *Eur. J. Oper. Res.* **84** 599–617.

Beasley, J. E. 1985. An exact two-dimensional non-guillotine cutting stock tree search procedure. *Oper. Res.* **33** 49–64.

Beasley, J. E. 1990. OR-Library: Distributing test problems by electronic mail. *J. Oper. Res. Soc.* **41** 1069–1072.

Biró, M., E. Boros. 1984. Network flows and nonguillotine cutting patterns. *Eur. J. Oper. Res.* **16** 215–221.

Caprara, A., M. Monaci. 2004. On the 2-dimensional knapsack problem. *Oper. Res. Lett.* **32** 5–14.

Christofides, N., C. Whitlock. 1977. An algorithm for two-dimensional cutting problems. *Oper. Res.* **25** 31–44.

Dongarra, J. J. 2004. Performance of various computers using standard linear equations software. Working paper, University of Tennessee, Knoxville, TN. Continuous updates available at http://www.netlib.org/benchmarks/performance.ps.

Dowsland, K. A. 1987. An exact algorithm for the pallet loading problem. *Eur. J. Oper. Res.* **31** 78–84.

Fekete, S. P., J. Schepers. 1997a. A new exact algorithm for general orthogonal d-dimensional knapsack problems. *Algorithms—ESA '97. Springer Lecture Notes in Computer Science*, Vol. 1284. Springer, 144–156.

Fekete, S. P., J. Schepers. 1997b. On more-dimensional packing I: Modeling. ZPR Technical Report 97-288. http://www.zpr.uni-koeln.de/~paper.

Fekete, S. P., J. Schepers. 1997c. On more-dimensional packing II: Bounds. ZPR Technical Report 97-289. http://www.zpr.uni-koeln.de/~paper.

Fekete, S. P., J. Schepers. 1997d. On more-dimensional packing III: Exact algorithms. ZPR Technical Report 97–290. http://www.zpr.uni-koeln.de/~paper.

Fekete, S. P., J. Schepers. 2001. New classes of lower bounds for bin packing problems. *Math. Programming* **91** 11–31.

Fekete, S. P., J. Schepers. 2004a. A combinatorial characterization of higher-dimensional orthogonal packing. *Math. Oper. Res.* **29** 353–368. [Journal version of S. P. Fekete, J. Schepers (1997b), http://www.zpr.uni-koeln.de/ ~paper.]

Fekete, S. P., J. Schepers. 2004b. A general framework for bounds for higher-dimensional orthogonal packing problems. *Math. Methods Oper. Res.* **60** 81–94. [Journal version of S. P. Fekete, J. Schepers (1997c), http://www. zpr.uni-koeln.de/ ~paper.]

Fekete, S. P., J. van der Veen. 2007. Packlib^2: An integrated benchmark library of multi-dimensional packing probelms. *Eur. J. Oper. Res.* Forthcoming. Benchmark library available at http://www.math.tu-bs.de/packlib2.

Fekete, S. P., E. Köhler, J. Teich. 2001. Higher-dimensional packing with order constraints. *Algorithms and Data Structures* (*WADS 2001*). *Springer Lecture Notes in Computer Science*, Vol. 2125, 192–204. Full version to appear in *SIAM J. Discrete Math.*

Ferreira, E. P., J. F. Oliveira. 2005. A note on Fekete and Schepers' algorithm for the non-guillotinable two-dimensional packing problem. Technical report. http://paginas.fe.up.pt/˜jfo/techreports/Fekete%20and%20Schepers%20OPP%20degeneracy.pdf.

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA.

Ghouilà-Houri, A. 1962. Caractérization des graphes non orientés dont on peut orienter les arrêtes de manière à obtenir le graphe d'une relation d'ordre. *C.R. Acad. Sci. Paris* **254** 1370–1371.

Gilmore, P. C., A. J. Hoffmann. 1964. A characterization of comparability graphs and of interval graphs. *Canadian J. Math.* **16** 539–548.

Golumbic, M. C. 1980. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York.

Grötschel, M. 1980. On the symmetric travelling salesman problem: Solution of a 120-city problem. *Math. Programming Stud.* **12** 61–77.

Hadjiconstantinou, E., N. Christofides. 1995. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *Eur. J. Oper. Res.* **83** 39–56.

Hochbaum, D. 1996. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing, Boston, MA.

Korte, N., R. H. Möhring. 1989. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comput.* **18** 68–81.

Martello, S., P. Toth. 1990. *Knapsack Problems—Algorithms and Computer Implementations*. Wiley, Chichester, UK.

Martello, S., D. Vigo. 1998. Exact solution of the two-dimensional finite bin packing problem. *Management Sci.* **44** 388–399.

Martello, S., D. Pisinger, D. Vigo. 2000. The three-dimensional bin packing problem. *Oper. Res.* **48** 256–267.

Mehlhorn, K. 1984. *Data Structures and Efficient Algorithms*, Vol. 1: *Sorting and Searching*. Springer Verlag, Berlin, Germany.

Nemhauser, G., L. Wolsey. 1988. *Integer and Combinatorial Optimization*. Wiley, Chichester, UK.

Padberg, M. 2000. Packing small boxes into a big box. *Math. Methods Oper. Res.* **52** 1–21.

Papadimitriou, C. H. 1994. *Computational Complexity.* Addison Wesley, New York.

Schepers, J. 1997. *Exakte Algorithmen für orthogonale Packungsprobleme*. Dissertation, Universität zu Köln, Köln, Germany. http://www.zpr.uni-koeln.de/˜paper.

Teich, J., S. P. Fekete, J. Schepers. 2001. Optimal hardware reconfigurations techniques. *J. Supercomputing* **19** 57–75.

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

Wottawa, M. 1996. *Struktur und algorithmische Behandlung von praxisorientierten dreidimensionalen Packungsproblemen*. Dissertation, Mathematisches Institut, Universität zu Köln, Köln, Germany. http://www.zpr.uni-koeln.de/˜paper.