Solving the non-oriented three-dimensional bin packing problem with stability and load bearing constraints

Jesper Hansen

June 16, 2003

Abstract

The three-dimensional bin packing problem is concerned with packing a given set of rectangular items into rectangular bins. We are interested in solving real-life problems where rotations of items are allowed and the packings must be packable and stable. Load bearing of items is taken into account as well. An on-line heuristic and an exact method have been developed and compared on real-life instances and as well on some benchmark instances. The on-line algorithm consistently reaches good solutions within a few seconds. The exact method is able to improve the solutions, but a significant amount of computation time is required.

1 Introduction

The three-dimensional bin packing problem (3D-BPP) is concerned with orthogonally packing a given set of rectangular items in rectangular bins. The bins can be identical or of different sizes. If they are identical we want to minimize the number of bins used; in case we can choose from different sizes, we want to minimize the cost of the used set of bins. We assume that it is possible to fit each item in at least one of the bin types.

We are interested in solving real-life problems, and hence additional constraints must be taken into account. It is generally allowed to rotate the items in the 6 different possible orientations, but restrictions on some items can occur, allowing only a subset of the orientations. The packing of items must be stable and possible to pack by a person or a packing robot. In order to guarantee this we will restrict ourselves to so-called robot-packable packings a concept introduced in den Boef et.al. [5] and Martello et. al. [11]. Basically items are placed starting from the bottom-left-back corner of the bin and successively placing items in front, on top or right of already placed items. The robot arm will in that way avoid collision with already placed items.

Dyckhoff [6] introduced a typology of cutting and packing problems denoting the 3D-BPP as 3/V/D/M. V and D denote problems where all items must be packed in possibly different bins and M that we consider instances of many items of different sizes. In principles of optimization there is no difference between cutting, nesting and packing problems. Note however that usually all items are considered different in Bin Packing, but we also consider the case of many items of a few different types denoted 3/V/D/R.

Lodi et. al. introduce a more detailed classification scheme focused on bin packing problems. They classify problems in the following dimensions:

- Dimension of the problem: 1,2,3,..
- Orientation of items: fixed orientation (O) or rotation allowed (R).
- Cutting restrictions on packings: guillotine cutting (G) or free cutting (F)

In our version of the problem we only allow robot-packable packings, which we denote (R) and the problem class is then 3BP|R|R.

The first attempt on solving multi-dimensional bin packing or cutting stock problems dates back to Gilmore and Gomory [8]. They were using column generation, but they restricted themselves to guillotine cuts. Chen et.al. [3] formulates an Integer Programming Model, but it is only solvable for very small instances. The first exact algorithm solving the 3D-BPP was proposed by Martello et.al. [11], but they were not considering rotation, stability and load bearing constraints. They used what could be considered a direct Branch & Bound approach branching on items in bins and afterwards position of the items in the bins.

A large number of papers have been published on heuristic approaches for the 3D-BPP. Ivancic et.al. [10] propose an integer programming based heuristic approach, which basically is a column generation approach without branching. Feasible solutions are merely achieved by solving the integer programming problem over the generated packings. The packings are generated with a greedy construction heuristic taking the dual prices into consideration. More recent is the use of Guided Local Search by Faroe et.al. [7].

We are interested in solving another variant of the problem introduced earlier – an on-line version where the order of the items arriving at the packing site is unknown. The items are in a queue, where we can observe and pick an item to pack from the first Q items. At the packing site S bins are available to pack at a time. When no more items can be packed in the bins one or more of the bins must be shipped off and replaced by one or more empty bin(s). To solve the on-line version we use the greedy heuristic described in section 6.

We use another approach for the problem where all data is known a priori. We also adopt the column generation approach. The pricing problem is a 3D Knapsack Problem, which we decompose as suggested by Pisinger and Sigurd [12] for solving the 2D Bin Packing Problem. The 3D Knapsack Problem is decomposed into a 1D Knapsack Optimization Problem and a 3D Knapsack Feasibility Problem. The 1D Knapsack Problem is a relaxation of the 3D Knapsack Problem. Pisinger and Sigurd use Constraint Programming to solve their 2D feasibility problem and at failures they add cuts to the 1D Knapsack Problem in a Branch & Cut fashion. We use an algorithm similar to the ONEBIN packing algorithm of Martello et.al [11] to solve the feasibility problem.

Figure 1 on the following page gives an overview of the entire column generation scheme. Duals from the Restricted Master Problem is sent to the 1D

Knapsack relaxation. The optimal solutions are checked for 3D Knapsack feasibility. Either a cut is sent to the 1D Knapsack or a feasible packing is sent to the Restricted Master Problem. This procedure continues until no feasible packing exists, which will improve the LP solution of the Restricted Master Problem.

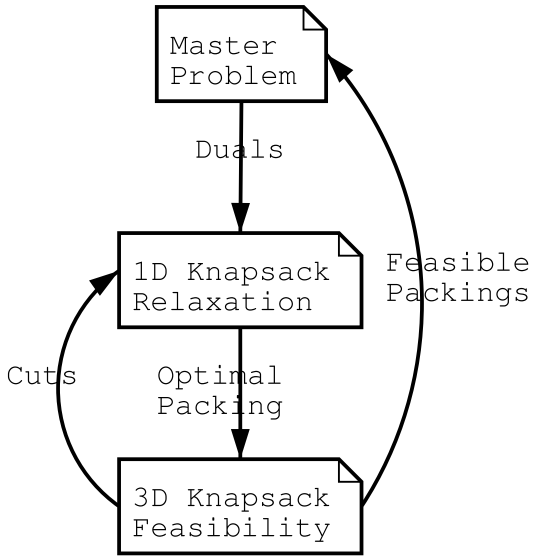

Figure 1: Overview of the solution approach.

The rest of the paper is organized as follows: In section 2 the column generation approach is described. The pricing routine is the topic of section 3. In section 4 we discuss branching schemes for column generation and in section 5 we give some lower bounds useful for pruning the Branch & Bound tree. In sections 7 and 8 we consider stability issues and load bearing constraints on the packed items. Finally in section 9 we compare the on-line algorithm described in section 6 with the exact approach on both real-life instances and different benchmark instances of varying difficulty.

2 Column Generation

Assume that we have generated all feasible packings of a single bin. The problem of selecting the best subset of packings covering all items can then be formulated as an integer programming problem in the following form where $\mathcal{P}$ is the set of generated packings:

$$\min \sum_{p \in \mathcal{P}} c_p q_p$$

$$\sum_{p \in \mathcal{P}} a_p^t q_p = d_t, \quad t \in \mathcal{T} \tag{1}$$

$$q_p \geq 0 \text{ and integer}, \quad p \in \mathcal{P}$$

c_p is the cost of packing p depending on the cost of the used bin type, a_p^t is the number of items of type t in packing p and d_t is the number of items of type t to be packed. The problem can also be formulated with $a_p^t \in \{0, 1\}$ and $d_t = 1$ by regarding identical items as different items. The size of $\mathcal{P}$ is exponential in the number of items, which is why we solve the problem with column generation. The integrality constraint is relaxed in (1) and we solve over a smaller set $\mathcal{P}' \subset \mathcal{P}$ of packings. The result is a lower bound for (1) and if an integral solution is found it is optimal. This problem is called the Restricted Master Problem. Initially we use a small set of packings for instance generated from a greedy construction heuristic ensuring a feasible solution. In following iterations we add columns improving the current solution. Let π_t be the dual variables from (1). An improving packing will now have a reduced cost, $\bar{c}_p = c_p - \sum_{t \in \mathcal{T}} a_p^t \pi_t < 0$. If for all $p \in \mathcal{P}$ $\bar{c}_p \geq 0$, then no improving packing exist and the current solution is optimal. After adding columns the problem is resolved giving new dual variables etc. The problem of finding columns with negative reduced costs is called the subproblem or pricing problem.

3 Pricing Problem

The pricing problem is a 3D Knapsack Problem. As Pisinger and Sigurd [12], we have chosen to decompose the problem into a 1D Knapsack Optimization Problem and a 3D Knapsack Feasibility Problem. The 1D problem is a relaxation of the 3D problem, which means that the solution value will be a lower bound on the optimal value.

There is a pricing problem for each bin type. Bin types are characterized by a length L_b, width W_b, height H_b, volume $V_b = L_b W_b H_b$, weight limit E_b and cost c_b for each b of the types in $\mathcal{B}$. A set of different item types $t \in \mathcal{T}$ are to be packed in bins. Each type is defined by the length l_t, width w_t, height h_t, volume $v_t = l_t w_t h_t$, possible rotations $R_t \subseteq \{1, 2, 3, 4, 5, 6\}$, weight e_t and the number of items of this type to be packed d_t.

We can now formulate the problem as a Knapsack Problem for a bin type $b \in \mathcal{B}$, where the variable u_t denote the number of items of type t in the knapsack:

$$\begin{aligned}
&\eta_b = \min \ c_b - \sum_{t \in \mathcal{T}} \pi_t u_t \\
&\sum_{t \in \mathcal{T}} v_t u_t \leq V_b, \\
&\sum_{t \in \mathcal{T}} e_t u_t \leq E_b, \\
&0 \leq u_t \leq d_t \text{ and integer}, \quad t \in \mathcal{T}
\end{aligned} \tag{2}$$

If $\eta_b < 0$ then the packing will improve the solution to the restricted master problem. Note that the problem has two knapsack constraints while the Knapsack Problem only have one. The two constraints do however not increase the dimension number of the problem.

3.1 3D Knapsack Feasibility Problem

After achieving the solution $u^\star$ to (2), it must be checked if the items can actually be packed in the bin. $u_t^\star$ defines how many items of type t must be packed in the bin; let $\mathcal{J}_t$ us denote the set of items. Each item $j \in \mathcal{J}_t$ is then of a specific type $t \in \mathcal{T}$ and the number of items is $|\mathcal{J}_t| = u_t^\star$.

The variables x_{jt}, y_{jt}, z_{jt} specify the lower-left-back position of the item and $r_{jt} \in \mathcal{R}_t$ specify the orientation of the item given the possible rotations of the type. Given the orientation r_{jt} of the item we can determine the length l_{jt}, width w_{jt} and height h_{jt} by using the element constraint:

$$\begin{array}{ll} \text{element}(r_{jt}, [l_t, l_t, w_t, w_t, h_t, h_t], l_{jt}), & j \in \mathcal{J}_t, t \in \mathcal{T} \\ \text{element}(r_{jt}, [w_t, h_t, l_t, h_t, l_t, w_t], w_{jt}), & j \in \mathcal{J}_t, t \in \mathcal{T} \\ \text{element}(r_{jt}, [h_t, w_t, h_t, l_t, w_t, l_t], h_{jt}), & j \in \mathcal{J}_t, t \in \mathcal{T} \end{array} \tag{3}$$

Each item must be inside the considered bin of type $b \in \mathcal{B}$:

$$\begin{array}{ll} 0 \le x_{jt} \le L_b - l_{jt}, & j \in \mathcal{J}, t \in \mathcal{T} \\ 0 \le y_{jt} \le W_b - w_{jt}, & j \in \mathcal{J}, t \in \mathcal{T} \\ 0 \le z_{jt} \le H_b - h_{jt}, & j \in \mathcal{J}, t \in \mathcal{T} \end{array} \tag{4}$$

No pair of items $j, k \in \mathcal{J}$ may overlap:

$$\begin{array}{ll} x_{jt} + l_{jt} \le x_{kt'} \vee x_{kt'} + l_{kt'} \le x_{jt} \quad \vee & \\ y_{jt} + w_{jt} \le y_{kt'} \vee y_{kt'} + w_{kt'} \le y_{jt} \quad \vee & \\ z_{jt} + h_{jt} \le z_{kt'} \vee z_{kt'} + h_{kt'} \le z_{jt}, & jt \ne kt',\ j \in \mathcal{J}_t,\ k \in \mathcal{J}_{t'}, \\ & \qquad t, t' \in \mathcal{T} \end{array} \tag{5}$$

The disjunctive non-overlap constraints are exactly the constraints making the problem extremely difficult to solve with MIP solvers. There is no guarantee that the solution will actually be packable. Instead we use a variant of the ONEBIN procedure by Martello et. al. [11]. The items are packed in so-called corner points. Initially the only corner point is the lower-left-back corner of the bin. At any following stage, items can only be placed to the right, above or in front of already placed items. Figure 2 on the next page is an illustration of available corner points for a given set of packed items.

The procedure is a depth first search where we at a given node consider placing all item types in all possible corner points and with each feasible item rotation. Each time an item is placed, the corner points are updated. The search is stopped when a feasible solution is achieved. As in Martello et. al. [11] we prune a node in the search tree if the volume of the remaining items is larger than the remaining volume of the bin.

Any packing found during the search phase with negative reduced cost is added to the master problem. If it was not possible to find such a packing, we want to resolve the 1D Knapsack Problem, but with the infeasible solution cut off the solution space. The procedure is then to add a cut removing the infeasible solution, resolve the 1D Knapsack Problem, check 3D Knapsack feasibility etc. until we achieve a feasible 3D packing. This approach was first suggested by Pisinger and Sigurd [12] for solving the 2DBPP.

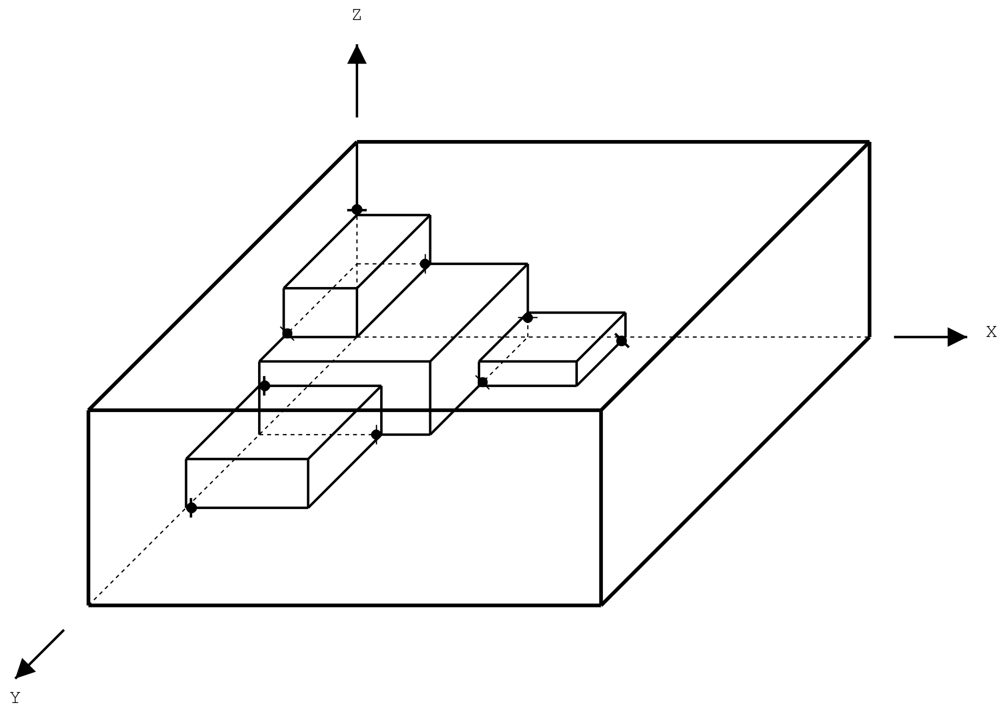

Figure 2: Illustration of corner points for a given packing.

3.2 Adding Infeasibility Cuts

Let U be the set of item types picked by the 1D Knapsack Problem. Now consider the following inequality:

$$\sum_{i \in U} u_i \leq \sum_{i \in U} u_i^\star - 1 \tag{6}$$

At first sight this may seem like a valid inequality to cut off the solution $u^\star$ with item types U. Assume that $U = \{1, 2\}$, $u_1^\star = 2$ and $u_2^\star = 1$ resulting in the cut $u_1 + u_2 \leq 2$. This cut will however remove for instance the solution $u_1 = 0$ and $u_2 = 3$ as well, which is not what we want. Instead we reformulate the 1D Knapsack in the following equivalent way where the variables u_{jt} is 1 if the j'th item of type t is placed in the knapsack:

$$\eta_b = \min \; c_b - \sum_{t \in \mathcal{T}} \sum_{j \in \mathcal{J}_t} \pi_t u_{jt}$$

$$\sum_{t \in \mathcal{T}} \sum_{j \in \mathcal{J}_t} v_t u_{jt} \leq V_b,$$

$$\sum_{t \in \mathcal{T}} \sum_{j \in \mathcal{J}_t} e_t u_{jt} \leq E_b,$$

$$u_{jt} \geq u_{it}, \qquad j < i, \; i, j \in \mathcal{J}_t, \; t \in \mathcal{T} \tag{7}$$

$$u_{jt} \in \{0,1\}, \quad j \in \mathcal{J}_t,\ t \in \mathcal{T}$$

The constraints (7) are included to avoid symmetries and are actually necessary for the introduced 3D infeasibility cuts: Consider again our example $U = \{1,2\}$, $u_1^\star = 2$ and $u_2^\star = 1$. In the new variables, we will have $u_{11}^\star = 1, u_{21}^\star = 1$, $u_{12}^\star = 1$ and all other variables equal to zero. The cut will then be $u_{11} + u_{21} + u_{12} \leq 2$ allowing the solution $u_{12}^\star = u_{22}^\star = u_{32}^\star = 1$. Without constraint (7) the solution $u_{11} = 1, u_{21} = 1, u_{12} = 0$ and $u_{22} = 1$ would be feasible, which is obviously an error.

Let U' be the set of pairs consisting of item type and item number in the optimal solution for the new formulation. The cut inequality then becomes:

$$\sum_{(j,t)\in U'} u_{jt} \leq |U'| - 1 \tag{8}$$

When we add a cut, we check if one or more of the items can be exchanged by other dominating items and hereby add more cuts. An item of type t is dominated by item t', if $l_{t'} \geq l_t$, $w_{t'} \geq w_t$, $h_{t'} \geq h_t$ and $R_{t'} \subseteq R_t$. All combinations of dominating items are generated. If an item t' is dominating all the items in the constraint, we instead add only one stronger constraint:

$$\sum_{(j,t)\in U'} u_{jt} + u_{1t'} \leq |U'| - 1 \tag{9}$$

Thousands of cuts can be generated even on relatively small instances. This will make it quite time consuming to solve the 1D Knapsack Problem. Many of the cuts will fortunately dominate or be dominated by other cuts and can hence be removed speeding up the time to solve the problem. Another trick from Pisinger and Sigurd [12] is to identify unpackable subsets of unpackable sets and hereby strengthening and reducing the number of cuts. Basically we remove the smallest item from the set until the subset is packable and then the last unpackable subset defines the cut.

3.3 Heuristic Packing Generation

The 1D/3D Knapsack phase can be very time consuming. To reduce that time we apply a similar search, as described earlier, before solving the 1D Knapsack Problem. The heuristic packing generation procedure is also a depth first search, but the size of the search tree is reduced. At a given node in the tree we consider placing only a subset of the item types with positive duals and only in a subset of all the possible corner points. For each level from the root of the tree the size of the subsets are halved, although always one possibility is tried. Any packing found during the search with a negative reduced cost is added to the master problem.

For the heuristic packing generation we apply another pruning strategy. During the search we save the volume of the packed items in a given node, if it is the largest volume packed so far. We then prune the tree at a node, if it is not possible to improve the best volume found so far, given the volume of the packed items so far and the remaining volume of the bin. The search is heuristic and if no interesting packings are found, we switch to the 1D/3D Knapsack phase.

4 Branch & Price

There is no guarantee that the optimal solution to the restricted master problem is integral. To achieve that we must embed the column generation in a Branch & Bound scheme resulting in a Branch & Price algorithm. The obvious branching strategy would be to do normal branching directly on the variables in the master problem – basically removing columns from the solution or forcing columns into the solution. Forcing columns into the solution works fine, but when removing a column there is a significant probability that exactly that column will be generated in the next pricing iteration. The branching scheme must work in the pricing problem as well. This is relatively easy to do for Set Partitioning and Covering type of problems, which we will see later, but we are solving general IP problems, which make things more complicated. Vanderbeck and Wolsey [15] developed a branching scheme also discussed in Barnhart et.al. [1], which we will use.

4.1 The Master Problem

Assume that the optimal restricted master solution, $q^{\star}$, of (1), is fractional. We may find a row t and an integer α_t, such that δ in

$$\sum_{p \in \left\{a_p^t \geq \alpha_t\right\}} q_p^{\star} = \delta$$

is fractional. Basically, we sum the $q_p^{\star}$ of columns p covering row t resulting in a fractional sum, δ. We can then branch on the following constraints:

$$\sum_{p \in \left\{a_p^t \geq \alpha_t\right\}} q_p \leq \lfloor \delta \rfloor \quad \text{and} \quad \sum_{p \in \left\{a_p^t \geq \alpha_t\right\}} q_p \geq \lceil \delta \rceil$$

These constraints are upper and lower bounds on the number of bins in the solution with $a_p^t \geq \alpha_t$. Note, that we are not able to remove or force columns into the solution by adding these constraints – the constraints must be added to the restricted master problem explicitly. This leads to an extra dual value in the reduced cost of any new column with $a_p^t \geq \alpha_t$.

In some cases it is not possible to achieve a fractional δ when considering one row only and we will have to consider multiple rows. For instance for the Set Partitioning Problem α_t is always one, since $a_p^t \in \{0,1\}$ and $\sum_{p \in \left\{a_p^t \geq 1\right\}} q_p^{\star} = 1$ for every row. To overcome this in the general case we search for two rows t and t' with α_t and $\alpha_{t'}$ such that δ is fractional in

$$\sum_{p \in \left\{a_p^t \geq \alpha_t \wedge a_p^{t'} \geq \alpha_{t'}\right\}} q_p^{\star} = \delta$$

We continue increasing the number of rows until δ is fractional. Let α be a vector with an entry for each constraint row and $P(\alpha) = \{p \in \mathcal{P}' | a_p^t \geq \alpha_t, t \in \mathcal{T}\}$. Then generally we search for a vector α where δ is fractional in

$$\sum_{p \in P(\alpha)} q_p^{\star} = \delta. \tag{10}$$

The restricted master problem at node n now becomes

$$\min \sum_{p\in\mathcal{P}'} c_p q_p$$

$$\sum_{p\in\mathcal{P}'} a_p^t q_p = d_t, \quad t\in\mathcal{T}$$

$$\sum_{p\in P(\alpha^j)} q_p \leq \lfloor \delta^j \rfloor, \quad j\in F^n \tag{11}$$

$$\sum_{p\in P(\alpha^j)} q_p \geq \lceil \delta^j \rceil, \quad j\in G^n \tag{12}$$

$$q_p \geq 0 \text{ and integer}, \quad p\in\mathcal{P}'$$

F^n and G^n in (11) and (12) are index sets over branching constraints defined by the pairs (α^j, δ^j) where δ^j is fractional in (10). The constraints are added from the root of the tree and down to the present node n. At a node a constraint of type (11) is added in one branch and of type (12) in the other branch.

4.2 The Pricing Problem

Now let us consider the changes to the pricing problem, (7). Let $R(\alpha)$ denote the chosen set of branching rows for a given α: $R(\alpha) = \{t\in\mathcal{T}|\alpha_t > 0\}$. Further, let λ be the dual variables corresponding to the added upper bound branching constraints and κ for the lower. The values of the dual variables will be $\lambda \leq 0$ and $\kappa \geq 0$. Finally, let s be a binary variable, where $s=1$, if $\sum_{j\in\mathcal{J}_t} u_{jt} \geq \alpha_t$, $\forall t\in R(\alpha)$ and $s=0$, otherwise. The pricing problem at node n of the tree now becomes

$$\eta_b = \min\ c_b - \sum_{t\in\mathcal{T}}\sum_{j\in\mathcal{J}_t} \pi_t u_{jt} - \sum_{j\in F^n} \lambda_j s_j - \sum_{j\in G^n} \kappa_j s_j \tag{13}$$

$$\sum_{t\in\mathcal{T}}\sum_{j\in\mathcal{J}_t} v_t u_{jt} \leq V_b,$$

$$\sum_{t\in\mathcal{T}}\sum_{j\in\mathcal{J}_t} e_t u_{jt} \leq E_b,$$

$$u_{jt} \geq u_{it}, \qquad j<i,\ i,j\in\mathcal{J}_t,\ t\in\mathcal{T}$$

$$s_j \leq u_{\alpha_t^j t}, \qquad j\in G^n,\ t\in R(\alpha^j) \tag{14}$$

$$s_j \geq 1 - \sum_{t\in R(\alpha^j)} (1-u_{\alpha_t^j t}), \qquad j\in F^n, \tag{15}$$

$$u_{jt}\in\{0,1\}, \quad j\in\mathcal{J}_t,\ t\in\mathcal{T}$$

$$s_j\in\{0,1\}, \quad j\in F^n\cup G^n$$

For each node n in the search tree extra constraints are added. Type (14) is added in the case that an upper bound has been added to the master problem and (15) in case of a lower bound.

There is a constraint of type (14) for each row $t \in R(\alpha^j)$ for $j \in G^n$. For $\kappa_j \geq 0$ implies that $s_j, j \in G^n$ should be 1 to minimize the objective. $s_j, j \in G^n$ can only be 1, if for all types $t \in R(\alpha^j)$ the α^j_t'th item is chosen, which basically is $u_{\alpha^j_t t} = 1$.

In case of a type (15) only one constraint is added in the given node. For $\lambda_j \leq 0$ implies that $s_j, j \in G^n$ should be 0 to minimize the objective. Here the variable s_j is forced to 1, if for at least one of the item types $t \in R(\alpha^j)$, the α^j_t'th item is chosen.

4.3 Searching for Branching Constraints

Clearly we want $|R(\alpha)|$ and the number of Branch & Bound nodes n to be as small as possible. Different strategies can be applied to find $R(\alpha)$ or more specifically α, such that $|R(\alpha)|$ is small or minimal. Other possibilities is searching for $\delta - \lfloor \delta \rfloor$ fractional, but close to one in order to quickly find an integral solution or $\delta - \lfloor \delta \rfloor$ close to 1/2 in order for the duals in the pricing problem to be as large as possible. We decided to consider only the last two. We show on this page the algorithm implemented in the procedures `findAlphaAndDelta` and `searchForOtherAlphas`. We assume that we have a compact representation of each column, $p \in \mathcal{P}'$, consisting of two arrays, A^p and T^p with entries for each $a^p_t > 0$ in A^p and corresponding $t \in \mathcal{T}$ in T^p. The arrays have length m^p. The results are stored in a global storage object, *Save*, which is accessibly by both `findAlphaAndDelta` and `searchForOtherAlphas`.

First the algorithm iterates through all fractional variables. For the corresponding column, f, we iterate through the positive entries and save the row number from T^f in the set R. Then `searchForOtherAlphas` is called.

Procedure `findAlphaAndDelta`

```
Data   : Given $q^\star$ and $A^p, T^p, m^p,\ p \in \mathcal{P}'$, global object $Save$
Result : Return the best $\alpha$ and $\delta$ from $Save$
for all fractional variables, $q^\star_f$ do
    for $i = 1, ..., m^f$ do
        $R = \{T^f[i]\}$
        searchForOtherAlphas($A^f, T^f, m^f, i, R, Save$)
```

In `searchForOtherAlphas` iteration is done over the remaining positive entries in column f. The procedure is called recursively to consider all combinations of entries. For each combination we, in the procedure `saveAlphaDelta`, save the corresponding α and δ, if δ is fractional in the corresponding sum $\sum_{p \in \{a_p \geq \alpha\}} q^\star_p = \delta$. The values are stored in *Save*.

Procedure `searchForOtherAlphas`($A^f, T^p, m^f, i, R, Save$)

```
Data   : Given $q^\star$ and global object $Save$
Result : The sets of potential $R$'s are stored in $Save$
for $j = i + 1, ..., m^f$ do
    $R = R \cup \{T[j]\}$
    if fractionalSum($q^\star, R$) then
        saveAlphaDelta($A^f, R, Save$)
    searchForOtherAlphas($A^f, m^f, j, R, Save$)
    $R = R \setminus \{T[j]\}$
```

If several rows have $d_t = 1$, then we can in some cases simplify the branching. For fractional $q^\star$, if there exists a pair of rows t, t' and $\alpha_t = \alpha_{t'} = 1$, such that $0 < \sum_{p \in \{a_p^t \geq 1\}} q_p^\star < 1$, then we avoid explicitly adding branching constraints. In one branch we simply remove all columns with $a_p^t = a_p^{t'} = 1$ and add the constraint $u_{1t} + u_{1t'} \leq 1$ to the pricing problem, i.e. items t and t' cannot be packed together. In the other branch we remove all columns with $a_p^t + a_p^{t'} \leq 1$ and add the constraint $u_{1t} = u_{1t'}$ to the pricing problem, i.e. items t and t' must be packed together. This is the branching strategy of Ryan and Foster [14].

5 Lower Bounds

Good lower bounds are always important for pruning the search tree when applying Branch & Bound. When using heuristics lower bounds are of interest for measuring the quality of the heuristic. For the 3D-BPP we can derive lower bounds directly from the problem data, but when applying column generation we can use the LP-solution of the restricted master problem to derive bounds as well.

The most obvious bound to calculate is the continuous lower bound:

$$L_0 = \min_{b \in \mathcal{B}} \max \left\{ \left\lceil \frac{\sum_{t \in \mathcal{T}} d_t v_t}{V_b} \right\rceil, \left\lceil \frac{\sum_{t \in \mathcal{T}} d_t e_t}{E_b} \right\rceil \right\}$$

Martello et. al. [11] show that the asymptotic worst-case performance ratio of L_0 is $\frac{1}{8}$. It is quite simple to realize that it must be close to that value, but harder to prove that it is tight. Here we are satisfied with the first. Consider the case of an instance with T items all of size $l = \frac{L}{2} + \sigma, w = \frac{W}{2} + \sigma$ and $h = \frac{H}{2} + \sigma$. For $\sigma = 0$ exactly $\frac{T}{8}$ bins are needed to pack the items. For $\sigma > 0$, T bins are necessary, but L_0 is still be close to $\frac{T}{8}$ for σ sufficiently close to 0.

The continuous bound is reasonably good when the item sizes are small compared to the bin size, but as we have seen can be very poor for relatively large items. Martello et.al. [11] introduce a new stronger bound, L_2, taking all three dimensions into account. When rotation of items is allowed, we have to use the smallest possible size of the items in all three dimensions, which deteriorates the bound. An alternative is to generalize the bounds of Dell'Amico et.al. [4] for the non-oriented 2D-BPP. All items are cut into smaller cubes, which allows a simplification of the bound calculation. The number of items created on the other hand can be quite large. If rotation is restricted to a subset of items and/or orientations, then the L_2 is definitely preferable. We still calculate the bound using the smallest possible size in each dimension, but these sizes can now be different for each side of the item, since the possible rotations are restricted. This makes the bound stronger than the Dell'Amico bound where the items are cut in cubes.

Note that the bound is not only used to get an initial lower bound on the number of bins. It is also used for detecting infeasible 1D Knapsack solutions in the pricing procedure.

The LP-solution to the restricted master problem is only a valid lower bound when no more columns can improve the solution. Let us formulate the Lagrangian Relaxation of the restricted master problem at node n (shown on page 9):

$$L(\pi,\lambda,\kappa) = \min \sum_{p\in\mathcal{P}} c_p q_p + \sum_{t\in\mathcal{T}} \pi_t \left(d_t - \sum_{p\in\mathcal{P}} a_p^t q_p \right) \tag{16}$$

$$+ \sum_{j\in F^n} \lambda_j \left(\lfloor \delta_j \rfloor - \sum_{p\in P(\alpha_j)} q_p \right) + \sum_{j\in G^n} \kappa_j \left(\lceil \delta_j \rceil - \sum_{p\in P(\alpha_j)} q_p \right)$$

$$q_p \in \mathbf{R}_+,\ p\in\mathcal{P}, \quad \pi_t\in\mathbf{R},\ t\in\mathcal{T}, \quad \lambda_j \le 0,\ j\in F^n, \quad \kappa_j \ge 0,\ j\in G^n$$

$L(\pi,\lambda,\kappa)$ is a lower bound to the LP-relaxation of the master problem – specifically π,λ,κ taking the values of the dual variables from the restricted master problem. By rearranging the terms in (16) we reach the following expression:

$$L(\pi,\lambda,\kappa) = \min \sum_{p\in\mathcal{P}} \left(c_p - \sum_{t\in\mathcal{T}} a_p^t \pi_t \right) q_p - \sum_{p\in P(\alpha_j)} \left(\sum_{j\in F^n} \lambda_j + \sum_{j\in G^n} \kappa_j \right) q_p$$

$$+ \sum_{t\in\mathcal{T}} d_t \pi_t + \sum_{j\in F^n} \lambda_j \lfloor \delta_j \rfloor + \sum_{j\in G^n} \kappa_j \lceil \delta_j \rceil \tag{17}$$

$$q_p \in \mathbf{Z}_+,\ p\in\mathcal{P}, \quad \pi_t\in\mathbf{R},\ t\in\mathcal{T}, \quad \lambda_j \le 0,\ j\in F^n, \quad \kappa_j \ge 0,\ j\in G^n$$

Note that the first line of the expression is a sum over the reduced costs, which is basically the pricing problem. Assume that we have a found a feasible solution for each bin type and let U_b be the number of bins of type b for each solution. Assume further that we have the optimal reduced costs for each pricing problem, η_b, from the last iteration. A valid lower bound is then

$$LB^n = \sum_{b\in\mathcal{B}} U_b \eta_b + \sum_{t\in\mathcal{T}} d_t \pi_t + \sum_{j\in F^n} \lambda_j \lfloor \delta_j \rfloor + \sum_{j\in G^n} \kappa_j \lceil \delta_j \rceil$$

For instances with more than one bin type this bound is rather weak. In conclusion we have a valid lower bound for each iteration of the column generation using the reduced costs of the pricing problems. We can at any time stop the column generation, if $\lceil LB^n \rceil \ge LP^n$, where LP^n is the solution value to LP-relaxation of the restricted master problem, since no better bound can be achieved in that node. If $\lceil LB^n \rceil \ge UB$ where UB is any upper bound to the 3D-BPP, then we can prune the node altogether.

6 The On-Line Algorithm

We have developed an on-line greedy algorithm to solve the on-line packing problem. The order of the items arriving at the packing site is unknown. The items are in a queue, where we can observe and pick an item to pack from the first Q items. At the packing site S bins are available to pack at a time. When no more items can be packed in the bins, then one or more of the bins must be shipped off and replaced by one or more empty bin(s).

When considering which item to place, we evaluate the placement of each available item in each available bins, in all available corner points and all possible rotations. We primarily choose that combination of item, bin, corner and rotation where the increase in wasted space is minimal. Wasted space is basically space, which is not used by an item, but is unavailable for packing after the particular item is placed. Before evaluating all the items we sort them in non-increasing order of the volume of the items. Hence in cases of equal waste, we place the larger item first. When none of the available items can be placed, we replace all available bins with new empty bins. The procedure continues until all items are placed.

The algorithm is not only used for solving the on-line problem, but is also used off-line to generate an initial upper bound and first columns in the Branch & Price approach.

7 Stability Issues

In real-life there is requirements on how items may be packed. Items should be placed on the floor of the bin or be somewhat supported underneath by other items. Support of items from the side of the bin or from the side of other items is not considered. An example of no support from the sides is the case where the bins are pallets.

A common measure of support is to require a certain percentage of an item to be supported – for instance at least 90%. Items supported by more than one item, may result in more stable packings. Generally the presence of guillotine cuts in packings make the packing less stable, but often these cuts are unavoidable, or deeming them infeasible is too restrictive causing less efficient packings.

The above measure of support is however not a sufficient condition for a packing to be stable. Subsets of items in the packing must also be stable. Consider for instance items placed as a stair-case. At some stage, increasing the number of items constructing the stair-case will make the packing unstable, caused by the combination of weight of the items and gravity. Stair-case types of packings are however not likely to be produced when requiring a significant amount of support. Things get even more complicated if the weight of an item is not evenly distributed (which we generally assume is the case). An additional measure of stability is the location of the center of gravity of a packing, which should preferably be located near the center of the bottom face of the bin.

We have in spite of the above considerations only implemented a simple constraint on the amount of support and number of supporting items.

8 Load Bearing Constraints

In practice the load bearing ability of the packed items must be taken into account. We assume that we are given a maximum allowed load per square unit (e.g. mm^2), o_t, for each item type t. This value might depend on which face of the item is up, but we ignore this fact and assume that it is constant or be the lowest value over the different faces. Note that it is straight forward to generalize the method to cope with different values depending on the orientation of the item. We assume that the load bearing ability is the same at any point on the top face, even though it might be stronger close to the edges of the item, also depending on the density of the item. Also we assume that the weight of an item, e_t, is evenly distributed over the contact area of the items below.

The above assumptions are similar to the assumptions of Ratcliff and Bischoff [13]. They, however, allow different load bearing depending on the orientation of the item. In their packing heuristic, they collect a block of identical items and place it on a surface, which completely supports the block of items.

The algorithm `checkLoadBearing` on the current page shows the procedure to check if an item j can be placed at a specific location in the bin. Line 1 and the loop starting in line 2 identify items below touching j. The algorithm `overlap` returns the size of overlap between to line segments. Line 3 of `checkLoadBearing` calculates the area of overlap between the two considered items. In this way the total area of support of item j is determined. The weight per square unit is then calculated in line 4 and the load bearing of item j is updated depending on its original load bearing and the maximum extra load items below can bear. If in the end the load bearing of item j is negative, it indicates that some item below is overloaded.

9 Experiments and Results

We have tested the approach described on a number of benchmark instances due to Martello et.al. [11], Ivancic et. al. [10] and as well on some instances that we have generated from real-life data made available by Bang & Olufsen.

The code was implemented in ECL^{i}PS^e [9], which is a Constraint Logic Programming framework based on Prolog. We have used Xpress-MP 13.26 for solving LP and IP problems. The experiments were executed on a SUN Fire 3800, 750 MHz computer.

Early experiments showed that the feasibility problem is extremely time consuming to solve. To limit the time spent in the feasibility problem, two phases of the packing feasibility problem is applied. We first set a time limit of up to 60 seconds depending on the number of items to be packed. During the search, if no more packings and branches can be applied the algorithm switches to a state with no time limit on the feasibility packing. It is checked at certain time intervals during the search, whether a packing has been added to the master problem. If an item has been added, the search is halted and

Procedure `checkLoadBearing`

```
Data   : Item $j$ to be placed. Set of already placed items, $PL$.
Result : The boolean variable $status$, which is false if constraints are
         violated.
if $z_j = 0$ then
    $status = true$
else
1   $PL^{below} = \{i \in PL | z_j = z_i + h_i\}$
    $PL^{overlap} = \emptyset$
    $total = 0$
2   foreach $i \in PL^{below}$ do
3       $area = \text{overlap}(x_j, l_j, x_i, l_i)\ \text{overlap}(y_j, w_j, y_i, w_i)$
        if $area > 0$ then
            $PL^{overlap} = PL^{overlap} \cup \{i\}$
            $total = total + area_i$
    if $total > 0$ then
        for $i \in PL^{overlap} \wedge o_j \geq 0$ do
4           $o_j = \min\{o_i - \frac{e_j}{total}, o_j\}$
        $status = (o_j < 0)$
    else
        $status = false$
```

Procedure `overlap(`x_1, l_1, x_2, l_2`)`

```
Data   : Location, $x$, and size, $l$, of two line segments: $(x_1, l_1, x_2, l_2)$
Result : The size of the overlap between the segments: $l_3$.
$x_3 = \max(x_1, x_2)$
$l_3 = \max(0, \min(x_1 + l_1 - x_3, x_2 + l_2 - x_3))$
```

the master problem resolved. Every 10th iteration we try to improve the upper bound by solving the master problem including the integrality constraints.

When using ECL^{i}PS^e it was found that a "lazy" depth first search Branch & Bound was the easiest approach. This basically means that when branching, we do not find the bound in each child node before choosing the next node to branch on. Furthermore the descendants in one of the branches are fully investigated before considering the second branch. Better lower bounds would be achievable if both children of a node was considered before branching further on one of them. In fact the lower bound of the entire problem can only be improved if the second child in the root node is considered at some stage.

9.1 Bang & Olufsen instances

The Bang & Olufsen instances are generated from real-life data on item sizes, weight, load bearing ability and allowed orientations. Stability in form of 90% support is also required. Data on 21 different item types were used for creating 9 instances. The data and instances are described and listed in appendix A on page 21.

In the following experiment the on-line heuristic could consider 20 items at a time and only place them on one pallet. The packings in the on-line solution was the starting point of the Branch & Price algorithm. Note that the Branch & Price is allowed to consider all packings, i.e. there are no restrictions on the packing order of the items. In case of an on-line problem, we can only use the Branch & Price for comparing the performance of the on-line algorithm. The lower bound of the Branch & Price algorithm is still valid, but the upper bound might not be achievable for the on-line problem.

A significant saving in packing costs by using Branch & Price instead of the on-line heuristic might however allow for a change in the packing procedure of the company, such that the items can be packed in any arbitrary order. For these instances a limit of 3600 seconds of computation time was available.

Table 1 on the next page gives the results of the on-line heuristic followed by the Branch & Price. "LB" and "UB" are lower and upper bounds. "Cols" and "Cuts" are generated columns and infeasibility cuts, "Nodes" are Branch & Bound nodes and finally "Time" is the computation time in seconds for finding the best upper bound. The initial gap between the lower and upper bounds is 35%, which is reduced to 28%. It is still a quite large gap, but the instances seem to be quite difficult to solve. In one case, over 16.000 cuts are added. This indicates that certain combinations of items are attractive to put together because of volume and dimensions, but the exact shapes of the items make them impossible to pack. These combinations of items takes a very long time to prove infeasible. In fact, 99% of the time is spend in the one bin packing algorithm. The saving by using Branch & Price for these test cases was only 10%. With more computation time it could possibly be increased.

As mentioned earlier, we use a lazy Branch & Bound approach. All the nodes considered for the results in table 1 are in the "left" side of the tree, which means that no improvement of the lower bound was possible.

The branching strategy in the runs reported in table 1 was to branch on

	On-Line		Branch & Price				
Items	LB	UB	UB	Cols	Cuts	Nodes	Time
19	3	5	4	665	2842	11	241
29	4	6	5	1164	5310	13	71
32	5	7	6	591	16056	11	48
33	4	7	6	782	8548	1	49
46	5	8	7	938	5877	11	96
45	7	10	9	728	9972	8	49
46	6	9	9	976	9935	13	0
54	6	9	9	814	8280	7	0
58	7	11	10	454	9209	13	226
Total	47	72	65	7112	76029	88	780

Table 1: B&O instances with “half” branching.

fractional sums close 0.5 as discussed in section 4 on page 10 in order for the duals in the pricing problem to be as large as possible. The results in table 2 are on the other hand with the strategy of branching on fractions close to 1 to quickly reach integral solutions. The quality of the solutions are exactly the same, but slightly more nodes and time are used in table 2.

	On-Line		Branch & Price				
Items	LB	UB	UB	Cols	Cuts	Nodes	Time
19	3	5	4	610	3598	15	342
29	4	6	5	925	8455	12	51
32	5	7	6	675	14114	12	1137
33	4	7	6	822	4517	11	49
46	5	8	7	710	4808	13	332
45	7	10	9	848	7595	12	73
46	6	9	9	687	8130	14	0
54	6	9	9	873	4950	12	0
58	7	11	10	938	6046	13	130
Total	47	72	65	7088	62213	114	2114

Table 2: B&O instances with “one” branching.

Now we turn our attention to the on-line algorithm. We have made several experiments investigating the impact of stability and load bearing requirements as well as how many items are available for packing and finally how many open pallets that can be packed.

The results are compiled in table 3 on the following page. The columns are divided into three groups: All the items are visible or packable, the first item is visible and the first 20 items are visible. Each of the groups are again divided into packing on 1 or 2 open pallets at a time. The results are sums over the 9 instances.

In the first row we have no stability constraints (0% support) and no load bearing constraints. In the next row we introduce load bearing constraints.

Comparing to the 90% support case, the load bearing constraints accounts for almost half the increase in number of necessary pallets. Going from 90 to 100% support does not introduce a significant increase in pallets.

Two open pallets seem to have a slightly negative effect, which is a bit surprising. More short-sighted decisions must be made in the two pallet case or perhaps the strategy of replacing both pallets leads to lost packing opportunities. The two cases of all or 20 visible items seem to have an insignificant impact. Clearly, the case of only one visible item is much worse.

	All items visible		1 item visible		20 items visible	
Support	1 Pallet	2 Pallets	1 Pallet	2 Pallets	1 Pallet	2 Pallets
no load, 0%	60	61	88	89	60	61
0%	67	69	102	97	68	71
90%	74	74	131	129	72	75
100%	72	73	138	129	73	77

Table 3: On-Line results for B&O instances.

To test the robustness of the on-line algorithm we, for the 58 item case with two pallets, did several runs shuffling the order of item arrivals. The results are shown in table 4. Not that surprisingly the arrival order gets more important when fewer items are visible. The difference between 10 and 20 visible items is insignificant.

Visible items	(no. of runs, bins)	Av. bins
20	(11, 12)	12.0
10	(1, 11), (7, 12), (3, 13)	12.2
1	(1, 19), (6, 21), (2, 22), (2,23)	21.4

Table 4: Results when repeating runs for B&O instances.

By re-running the algorithm shuffling the input order, we have created a randomized algorithm. For instance with 10 visible items, repeating the algorithm the result improves from 13 to 11 bins.

9.2 Martello et. al. [11] instances

In the instances constructed by Martello et. al. [11], we have only one item of each type and the items cannot be rotated. 9 different classes with 10 instances each are given. Class 4 for example has a significant number of very large items making the instances relatively easy to solve. Class 5 on the other hand has mainly small items. Finally we will mention class 9. Here the items are cut out of 3 boxes, which means that the optimal solution is always 3 bins with no waste.

In table 5 on the following page is given the number of instances solved to proven optimality within 1200 seconds for the different classes and instance sizes. A total of 344 instances was solved to proven optimum out of 810 possible.

Clearly, the algorithm does not scale very well for these problem instances. Class 9 seems to be particularly difficult to solve. Class 4 is the exception, since

	Class									
Items	1	2	3	4	5	6	7	8	9	Total
10	10	10	10	10	10	10	10	10	10	90
15	9	10	10	10	10	9	9	10	10	90
20	6	8	8	10	9	10	5	9		65
25	3	2	4	10	8	4	1			32
30	4	1	1	10	2		1	1		20
35	1	1		10	4			1		17
40		1		10						11
45		1		10			1			12
50				10						10
Total	33	34	33	90	43	33	33	31	20	344

Table 5: Instances solved to proven optimality.

all instances are solved to optimality. Looking at the computational effort, instances with 10 items are solved in 0.8 seconds on average, while for 15 items 109.5 seconds are needed.

This is not particularly impressive compared to the results of Martello et. al. [11]. They solve 698 to proven optimality only leaving 112. There are several reasons for that. They use tricks and heuristics for the single bin packing that we do not. Their algorithm is implemented in C, which is faster than ECL^{i}PS^e – their hardware on the other hand is slower. Finally there is the issue of optimization approach: They use a direct Branch & Bound while we use a Branch & Price approach. The exact reasons are of course difficult to establish.

9.3 Ivancic et. al. [10] instances

Ivancic et. al. [10] constructed 17 instances with multiple container types and later Bischoff and Ratcliff [2] made 47 modified versions available electronically. The 47 instances are basically the 17 instances with only one container type per instance.

The instances have relatively few item types, i.e. 2 to 5, but many items are to be packed; in one instance a total of 180 items of 4 different types. The items can be packed in all 6 different orientations. 90% of the bottom face of the items are to be supported, but no load bearing is considered. 1800 seconds was available for each of the 47 instances.

Table 6 on the next page shows the results achieved on the 47 instances. The first column refers to the numbering scheme in [2] while the second column refers to the one in [10]. The following column gives the number of different item types per instance. The following two columns are the results reported in [10] and [2]. Note that the results of the latter actually is the best over two different methods. The next two columns are lower and upper bounds for our approach without stability constraint and the last two columns are the results with 90% support required. In [2] the items are 100% supported due to the design of their algorithm and in [10] no support is required though a packing routine similar to ours is applied.

Ref.	Ref.	Box	Method	Method			Stability	
[2]	[10]	types	of [10]	of [2]	LB	UB	LB	UB
1	1a	2	26	27	25	25	25	25
2	1b	2	11	11	9	9	10	10
3	2a	4	20	21	19	19	19	19
4	2b	4	27	27	26	26	26	26
5	2c	4	65	61	51	51	51	51
6	3a	3	10	10	10	10	10	10
7	3b	3	16	16	16	16	16	16
8	3c	3	5	4	4	4	4	5
9	4a	2	19	19	19	19	19	19
10	4b	2	55	55	55	55	55	55
11	4c	2	18	19	16	16	16	16
12	5a	3	55	55	53	53	53	53
13	5b	3	27	25	25	25	25	25
14	5c	3	28	27	27	27	27	27
15	6a	3	11	11	11	11	11	11
16	6b	3	34	28	26	26	26	26
17	6c	3	8	8	8	8	10	10
18	7a	3	3	2	2	2	3	3
19	7b	3	3	3	3	3	4	4
20	7c	3	5	5	5	5	6	6
21	8a	5	24	24	20	20	20	20
22	8b	5	10	11	8	9	8	12
23	8c	5	21	22	19	19	20	20
24	9a	4	6	6	5	6	5	6
25	9b	4	6	5	4	6	4	6
26	9c	4	3	3	3	4	3	5
27	10a	3	5	5	4	5	4	7
28	10b	3	10	11	10	10	10	10
29	11a	4	18	17	17	17	17	17
30	11b	4	24	24	22	22	22	22
31	11c	4	13	13	11	13	13	13
32	12a	3	5	4	4	5	4	5
33	12b	3	5	5	4	5	4	5
34	12c	3	9	9	9	9	9	9
35	13a	2	3	3	3	3	3	3
36	13b	2	18	14	14	14	14	14
37	14a	3	26	23	23	23	23	23
38	14b	3	50	45	45	45	45	45
39	14c	3	16	16	15	15	17	17
40	15a	4	9	10	7	9	7	11
41	15b	4	16	16	15	15	16	16
42	16a	3	4	5	4	5	5	5
43	16b	3	3	3	3	4	3	4
44	16c	3	4	4	3	5	3	6
45	17a	4	3	3	2	4	2	4
46	17b	4	2	2	2	3	2	5
47	17c	4	4	3	3	5	3	4
Total			763	740	689	713	730	759

Table 6: Results for the Ivancic et. al. [10] instances.

The gap between the total lower and upper bounds without stability considerations is only 3.4% and the solutions are 6.6% better than [10] and 3.6% better than [2]. The instances 25, 26, 32 and 42 to 47 indicate that our approach has difficulties on instances resulting in solutions with relatively few bins. Our method, on the hand, is superior on instances resulting in solutions with more bins: Instances 1 to 5, 11, 12, 16, 21 to 23 and 30.

The picture is the same with stability constraints. The gap is 3.8%, but now comparing to [2] the difference in quality is 2.5% in favour of [2]. Their results are however the minimum over two different methods with totals 763 and 777. Compared to those numbers our method is still competitive.

These instances are clearly much easier to solve than the B&O and Martello et. al. instances. Fewer packing patterns and cuts are necessary to reach the optimum and prove it. Even though the instances are quite large when considering the total number of items, the algorithm takes advantage of the few types of items when packing. In each corner point only one item for each type is tried, but there is still many corner points to consider.

10 Conclusion

In this paper we have described both the off- and on-line versions of the non-oriented 3DBPP with additional constraints concerning the stability of the packing and load bearing of the items.

We have proposed a Branch & Price approach for solving the off-line problem combined with a Branch & Cut procedure for solving the 3D Knapsack Problem. Design and implementation issues have been discussed specially regarding stability and load bearing constraints. For the on-line problem a greedy on-line heuristic was developed and described. In conclusion, our method can solve general problem classes and at the same time take real-life considerations into account.

Experiments were conducted on instances generated from real-life item data. The instances were found to be quite difficult to solve with the proposed approach, but relatively good results were achieved with the on-line heuristic. More work is required for optimally solving the instances and hence exactly evaluate the performance of the on-line heuristic. Our implementation of the Branch & Price approach were not competitive compared to the approach in Martello et. al. designed specifically for the 3DBPP. On the Ivancic et. al. instances however the results were quite promising with a gap between lower and upper bounds on less than 4%.

A Construction of Bang & Olufsen instances

In the following are the item data and constructed instances listed. Note that the dimensions of the items are measured in mm, weight in kg and load bearing in g/mm^2. The size of the bin is $(L, W, H) = (1200, 800, 2700)$ with unlimited weight capacity (In reality the bins are pallets). The 21 item types are listed in table 7 on the following page. l_t, w_t and h_t are the length, width and height of item t, R_t is the set of possible orientations, e_t is the weight of item t and o_t is the load bearing capacity per square mm of item t. 9 random instances were

t	l_t	w_t	h_t	R_t	e_t	o_t
1	1200	800	1370	$\{1,3\}$	80	0.16
2	1100	800	1470	$\{1,3\}$	80	0.17
3	800	800	1260	$\{1,3\}$	80	0.23
4	800	600	1020	$\{1,3\}$	80	0.31
5	1100	800	1090	$\{1,3\}$	80	0.17
6	965	590	820	$\{1,3\}$	80	0.26
7	765	595	795	$\{1,3\}$	40	0.00
8	620	495	680	$\{1,3\}$	35	0.00
9	800	600	1175	$\{1,2,3,4,5,6\}$	7	2.08
10	2020	290	320	$\{1,2,3,4,5,6\}$	20	1.71
11	490	490	120	$\{1,2,3,4,5,6\}$	25	4.16
12	455	396	440	$\{1,2,3,4,5,6\}$	15	5.55
13	380	360	450	$\{1,2,3,4,5,6\}$	8	7.31
14	380	360	450	$\{1,2,3,4,5,6\}$	7	7.31
15	1216	170	205	$\{1,2,3,4,5,6\}$	12	4.84
16	1425	360	220	$\{1,2,3,4,5,6\}$	18	1.95
17	1190	280	190	$\{1,2,3,4,5,6\}$	16	3.00
18	650	460	265	$\{1,2,3,4,5,6\}$	12	3.34
19	1060	410	220	$\{1,2,3,4,5,6\}$	22	2.30
20	690	510	270	$\{1,2,3,4,5,6\}$	13	2.84
21	850	435	210	$\{1,2,3,4,5,6\}$	11	2.70

Table 7: Data for the 21 different item types.

generated from these item types as shown in table 8 on the next page. The best lower and upper bounds found so far are found in the last 2 rows.

References

[1] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, *Branch-and-price: column generation for solving huge integer programs*, Operations Research, 46 (1998), pp. 316–329.

[2] E. E. Bischoff and M. S. W. Ratcliff, *Issues in development of approaches to container loading*, Omega, International Journal of Management Science, 23 (1995), pp. 377–390.

[3] C. S. Chen, S. Lee, and Q. Shen, *An analytical model for the container loading problem*, European Journal of Operational Research, 80 (1995), pp. 68–76.

[4] M. Dell'Amico, S. Martello, and D. Vigo, *A lower bound for the non-oriented two-dimensional bin packing problem*, Discrete Applied Mathematics, 118 (2002), pp. 13–24.

Item	Number of items								
type	19	29	32	33	46	45	46	54	58
1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	4	4	3	3
3	1	1	1	1	2	1	1	1	1
4	1	2	2	2	1	2	2	2	4
5	1	1	2	1	2	3	2	1	5
6	1	1	2	2	3	1	1	1	3
7	1	1	2	2	3	1	1	2	4
8	1	2	2	3	3	3	4	4	2
9	1	2	1	1	2	2	1	3	2
10	1	1	2	2	2	1	2	6	5
11	1	2	2	1	2	1	1	1	1
12	1	1	1	1	2	3	4	2	2
13	1	1	1	2	3	3	3	2	5
14	1	2	1	2	2	2	3	4	1
15	1	1	2	2	2	3	2	2	1
16	1	1	1	2	3	2	4	1	4
17	1	2	1	2	2	3	1	4	3
18	1	2	2	1	3	3	3	3	3
19	1	2	1	1	2	3	1	2	4
20		2	2	3	3	3	2	5	4
21			2		3		3	4	
LB	3	4	5	4	5	7	6	6	7
UB	4	5	6	6	8	9	9	8	11

Table 8: Constructed instances and lower bounds.

[5] E. den Boef, J. Korst, S. Martello, D. Pisinger, and D. Vigo, *A note on robot-packable and orthogonal variants of the three-dimensional bin packing problem*, DIKU Technical Report 03/02, (2003).

[6] H. Dyckhoff, *A typology of cutting and packing problems*, European Journal of Operational Research, 44 (1990), pp. 145–159.

[7] O. Faroe, D. Pisinger, and M. Zachariasen, *Guided local search for the three-dimensional bin packing problem*, to appear in INFORMS Journal on Computing, (2002). Also available as DIKU Technical Report 99/13.

[8] P. C. Gilmore and R. E. Gomory, *Multistage cutting stock problems of two and more dimensions*, Operations Research, 13 (1965), pp. 94–120.

[9] IC-Parc, ECL^{i}PS^e, Imperial College, London, 5.5 ed., 2002. http://www.icparc.ic.ac.uk/eclipse/.

[10] N. Ivancic, K. Mathur, and B. B. Mohanty, *An integer programming based heuristic approach to the three-dimensional packing problem*, Journal of Manufacturing and Operations Management, 2 (1989), pp. 268–298.

[11] S. Martello, D. Pisinger, and D. Vigo, *The three-dimensional bin packing problem*, Operations Research, 48 (2000), pp. 256–267.

[12] D. Pisinger and M. Sigurd, *On using decomposition techniques and constraint programming for solving the two-dimensional bin packing problem*, DIKU Technical Report 03/01, (2003).

[13] M. S. W. Ratcliff and E. E. Bischoff, *Allowing for weight considerations in container loading*, OR Spektrum, 20 (1998), pp. 65–71.

[14] D. M. Ryan and B. A. Foster, *An integer programming approach to scheduling*, in Computer Scheduling of Public Transport, A. Wren, ed., North-Holland Publishing Company, 1981.

[15] F. Vanderbeck and L. A. Wolsey, *An exact algorithm for ip column generation*, Operations Research Letters, 19 (1996), pp. 151–159.