Title

Pedro Henrique Centenaro*
Supervisor: Luiz-Rafael Santos

Abstract. TBD

*Undergraduate Control, Automation and Computation Engineering student at UFSC/Campus Blumenau.

1 Introduction

TBD

2 Exact models for packing problems

2.1 The binary knapsack problem

Informally, knapsack problems (KPs) may be described as finding the optimal combination of items from a set. Usually, this decision involves a certain value intrinsic to each item, and the objective is to maximize the total value of the selected items. To complicate this decision, items may have certain limiting values associated with them, which means not all of them may be selected.

The simplest example of a KP is the binary knapsack problem (BKP), also known as 0-1 knapsack problem [1]. It consists of a list of n items, each with a value v_i and a weight w_i associated with them, for $i \in \{1, \dots, n\}$. A subset of items must be selected in order to maximize the total value, subject to a maximum weight capacity[1] C. One may thus formulate the BKP as Integer Program (1),

$$\begin{aligned} &\max \sum_{i=1}^{n} x_i v_i \\ &\text{s. t.} \sum_{i=1}^{n} x_i w_i \leq C, \\ &\qquad x_i \in \{0,1\} \quad \forall i \in \{1, \dots, n\}, \end{aligned} \tag{1}$$

where x_i are decision variables that specify whether item $i \in \{1, \dots, n\}$ has been selected or not.

The selection of subsets implies that an upper bound on the number of solutions to the BKP is 2^n. Therefore, a natural first solution to the problem is a recursive binary tree algorithm [11]. Let $I = \{1, \dots, n\}$ be a set enumerating the items that may be put in the knapsack. Then, by representing the values and weights of items as entries in vectors $v, w \in \mathbb{R}^n$, respectively, and the remaining knapsack weight capacity as $c \in \mathbb{R}$, we arrive at Algorithm 1 for an instance $\phi(n, c, X, s)$ of the problem, where $X \subseteq I$ is the set of selected items and s is the sum of said items' values at the node.

The algorithm consists of a top-bottom phase, in which the nodes are generated, and a bottom-top phase, in which the optimal solution is found. The top-bottom phase begins at Line 3, where a new instance of ϕ is called with item n discarded from the solution. Subsequently, if there is enough capacity left in the knapsack, a second node is

[1] This limiting factor, which can be interpreted as a bag or a container, is what gives the KP its name.

Algorithm 1 Recursive binary tree algorithm for the KP: $\phi(n, c, X, s)$

1: **if** $n = 0$ **then**
2: return (X, s)
3: $X_1, s_1 \coloneqq \phi(n-1, c, X, s)$
4: $X_2, s_2 \leftarrow (\emptyset, \emptyset)$
5: **if** $w_n \leq c$ **then**
6: $X_2, s_2 \coloneqq \phi(n-1, c-w_n, X \cup \{n\}, s+v_n)$
7: $s \coloneqq max(s_1, s_2)$
8: $X_3 \leftarrow 0$
9: **if** $v = v_1$ **then**
10: $X_3 \coloneqq X_1$
11: **else**
12: $X_3 \coloneqq X_2$
13: return (X_3, s)

generated, with n included in the solution.

When $n = 0$ (Line 1), it is impossible to generate a new node. Thus, the only remaining action is to return the solution set at the node and its value sum. Once the whole tree has been generated, Line 7 is reached. From this point on, the bottom-top phase begins, with each node selecting the best solution its children have to offer. Hence, the optimal solution is returned by the root node.

Due to the combinatorial nature of the KP, Algorithm 1 is only efficient at solving small problems [11].

Example 2.1. Suppose we have a knapsack with maximum capacity $C = 5$ and three items. The weight vector is $w = [1, 2, 3]$ and the value vector is $v = [2, 4, 3]$. By applying Algorithm 1 to this problem, we obtain the tree in Figure 1.

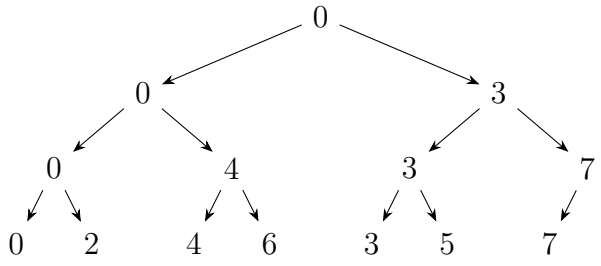


Figure 1: BKP binary tree algorithm example.

Each node in the tree represents the value sum of a feasible combination of items. It immediately becomes clear that the optimal solution in this case is 7, with the combination

of items 2 and 3. However, this is not obvious to the computer, which must solve every recursion and return to the root node first.

Dynamic programming [4, 21] approaches to the KP exist [7, 11, 13]. The idea behind these approaches is to generate a table with solutions to subproblems of the KP, using previous calculations to shorten the path to the optimal solution.

Let $T \in \mathbb{R}^{(n+1)\times(c+1)}$ be the dynamic programming table for the KP. Algorithm 2 exemplifies how T can be constructed and used to arrive at the optimal KP solution.

Algorithm 2 Dynamic programming algorithm for the KP: $\Phi(n, c)$

```
1:  for i from 1 to n + 1 do
2:      T_{i,1} := 0
3:  for j from 2 to c + 1 do
4:      T_{1,j} := 0
5:      for i from 2 to n + 1 do
6:          T_{i,j} ← T_{i−1,j}
7:          if w_i ≤ j then
8:              T_{i,j} ← max(T_{i,j}, v_{i−1} + T_{i−1,j−w_{i−1}})
9:  n_i, c_i, X ← (n, c, ∅)
10: while c_i > 0 do
11:     if T_{n_i+1,c_i+1} ≠ T_{n_i,c_i+1} then
12:         X ← X ∪ n_i
13:         c_i ← c_i − w_n
14:     n_i ← n_i − 1
15: return X
```

$T_{i,j}$ represents the optimal value for a subproblem of the KP with $i-1$ items and knapsack capacity $j-1$. Consequently, no set of items can satisfy $j = 1$, and so $T_{i,1} = 0\,\forall i \in \{1, \ldots, n+1\}$. Furthermore, since there are no items to select when $i = 1$, it follows that $T_{1,j} = 0\,\forall j \in \{1, \ldots, c+1\}$. This explains lines 1 through 4.

Lines 5 through 8 fill the remaining entries in the table. If $w_{i-1} > j-1$, we simply keep the best solution found so far for capacity $j-1$. Otherwise, we compare said solution with what would be obtained if $i-1$ was selected. The larger value becomes entry $T_{i,j}$.

Once table T is finished, starting at Line 10, the process of determining what items constitute an optimal solution occurs. Following the logic of lines 5 through 8, if $T_{i,j} = T_{i-1,j}$, then the current item has not been selected, and we may move on to the next item. Otherwise, the item is included in the final solution, and we move on to the next item with less capacity, $T_{i-1,j-w_i}$.

Though better than the binary tree approach, the dynamic programming algorithm's

complexity is exponential as well [11]. In fact, this approach is notably inefficient for large capacity values [13].

Example 2.2. Suppose the following problem suggested by Feofiloff [11]: We have 4 items and a capacity 5. The weights and values of items are, respectively, $w = [4, 2, 1, 3]$ and $v = [500, 400, 300, 450]$. To find the optimal solution to this BKP, we start by applying the part of Algorithm 2 to generate Table 1.

Table 1: BKP dynamic programming table example.

0	0	0	0	0	0
0	0	0	0	500	500
0	0	400	400	500	500
0	300	400	700	700	800
0	300	400	700	750	850

Next, we start looking for the solution from the table's bottom right entry. We note that $T_{5,6} \neq T_{4,6}$. This means that item $4 \in X$ (the solution set). Next, we investigate item 3. Since $4 \in X$, that means the capacity is now at $5 - w_4 = 2$. So we look at $T_{4,3}$, which is equal to $T_{3,3}$. We conclude that $3 \notin X$. Moving up a row, $T_{3,3} \neq T_{2,3}$, which means $2 \in X$. By repeating this one more time, we conclude that $X = \{2, 4\}$.

Along with these algorithms, an integer programming model of the BKP was implemented in `JuMP`.

2.2 The two-dimensional knapsack problem

Since the BKP contains a single capacity constraint (see Integer Program (1)), we can think of each item as a line segment whose length is w_i. It follows that the sum of lengths of the items must not exceed capacity C.

We wish to extrapolate this geometric interpretation of the BKP in a way that allows us to solve problems related to areas and volumes. In other words, we want to formulate two- and three-dimensional knapsack problems. To arrive at such formulations, first consider the multidimensional knapsack problem (MKP) [15], formulated in Integer Program (2).

$$
\begin{aligned}
&\max \sum_{i=1}^{n} v_i x_i \\
&\text{s. t.} \sum_{i=1}^{n} w_{ji} x_i \leq C_j, \quad \forall j \in \{1, \ldots, d\} \\
&\qquad x_i \in \{0, 1\}, \quad \forall i \in \{1, \ldots, n\}.
\end{aligned} \tag{2}
$$

A number of differences can be spotted in this model, compared to the BKP's. The first is the introduction of parameter $d \in \mathbb{R}$, the number of dimensions that are being taken into account. Capacity $C \in \mathbb{R}$ becomes $C \in \mathbb{R}^d$, and each item $i \in \{1, \ldots, n\}$ now has an associated vector $w_{ji} \in \mathbb{R}^d$ describing its weight across the problem's dimensions.

Specifically in the case of two dimensions, we may rewrite this formulation as Integer Program (3).

$$
\begin{aligned}
&\max \sum_{i=1}^{n} v_i x_i \\
&\text{s. t.} \sum_{i=1}^{n} w_i x_i \leq W, \\
&\qquad \sum_{i=1}^{n} h_i x_i \leq H, \\
&\qquad x_i \in \{0, 1\}, \quad \forall i \in \{1, \ldots, n\}.
\end{aligned} \tag{3}
$$

Here, w_i and h_i represent the width and height of item i, while W and H represent the width and height of the container where we wish to place the items. We assume both the items and the container to be rectangular.

Suppose we want to place three items within a 30×20 container. The width, height and value vectors for this problem are $w = [10, 10, 5]$, $h = [5, 10, 15]$ and $v = [10, 15, 5]$. Through the plot on the left in Figure 2, we can see that this is a trivial task. All items are selected, and the optimal value is 30.

Our MKP formulation begs to differ: Only items 1 and 2 are selected, for an optimal value of 25. Why? The plot on the right illustrates the issue: The MKP is applying the line segment logic to multiple dimensions, as if each chosen subset of items ended up constituting a larger rectangle of dimensions $\sum_{i=1}^{n} w_i \times \sum_{i=1}^{n} h_i$.

We need to sophisticate our model's constraints in order to solve 2D knapsack

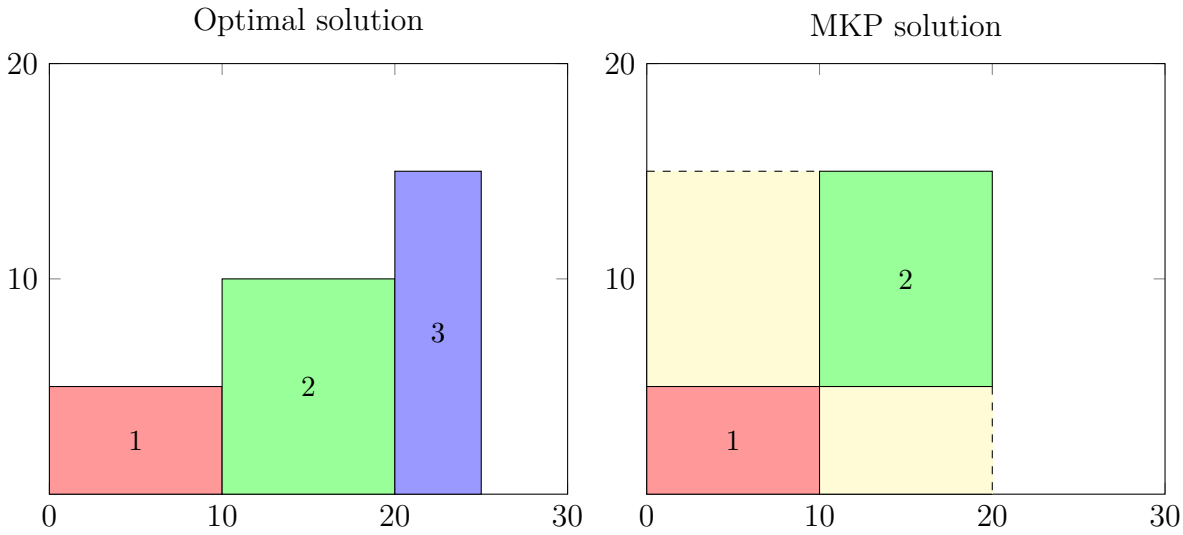


Figure 2: Two-dimensional items within a 30×20 container.

problems. We begin by introducing variables $x_i, y_i \in \mathbb{R}$, which describe the position of each item's bottom-left corner. It immediately becomes clear that every selected item $i \in \{1, \ldots, n\}$ must obey Constraint Set (4),

$$\begin{aligned} 0 \leq x_i \leq W, \\ 0 \leq x_i + w_i \leq W, \\ 0 \leq y_i \leq H, \\ 0 \leq y_i + h_i \leq H, \end{aligned} \tag{4}$$

where W and H are the width and height of the container, respectively.

Our task is complicated by the need to avoid item overlap. To accomplish this, we adopt the approach described by Kalvelagen [14]. For every selected item $i, j \in \{1, \ldots, n\}$ with $i \neq j$, at least one inequality from Constraint Set (5) must hold true.

$$x_i + w_i \leq x_j, \tag{5a}$$
$$x_j + w_j \leq x_i, \tag{5b}$$
$$y_i + h_i \leq y_j, \tag{5c}$$
$$y_j + h_j \leq y_i. \tag{5d}$$

If we force all of these constraints upon the model, that might result in slower solve

times. Thus, we need a way to enforce one constraint *or* another. In other words, we have *alternative constraints* [4].

To implement this behavior in a 2DKP model, Kalvelagen [14] resorts to *conditional constraints*, defined by Bradley, Hax, and Magnanti [4] as constraints of the form

$$f_1(x_1, x_2, \ldots, x_n) > b_1 \Rightarrow f_2(x_1, x_2, \ldots, x_n) \leq b_2. \tag{6}$$

In our case, the desired relations are given by variables $\delta_{i,j,k} \in \{0, 1\}$, $k \in \{1, \ldots, 4\}$:

$$\delta_{i,j,1} = 1 \Rightarrow x_i + w_i \leq x_j, \tag{7}$$
$$\delta_{i,j,2} = 1 \Rightarrow x_j + w_j \leq x_i, \tag{8}$$
$$\delta_{i,j,3} = 1 \Rightarrow y_i + h_i \leq y_j, \tag{9}$$
$$\delta_{i,j,4} = 1 \Rightarrow y_j + h_j \leq y_i. \tag{10}$$

These constraints can be mathematically implemented as *big M constraints*[2]. The idea is to introduce a constant $M \to +\infty$ and rewrite the constraints as

$$x_i + w_i - M(1 - \delta_{i,j,1}) \leq x_j, \tag{11}$$
$$x_j + w_j - M(1 - \delta_{i,j,2}) \leq x_i, \tag{12}$$
$$y_i + h_i - M(1 - \delta_{i,j,3}) \leq y_j, \tag{13}$$
$$y_j + h_j - M(1 - \delta_{i,j,4}) \leq y_i. \tag{14}$$

Finally, we add the following constraint:

$$\sum_{k=1}^{n} \delta_{i,j,k} \geq 1. \tag{15}$$

For $M \to +\infty$, if a certain constraint is not selected, it's obvious that the inequations hold. This remains true for any item smaller than the container when $M = \max\{W, H\}$.

It is possible to introduce rotations to this model. For this purpose, we create new variables σ_i. When item i is rotated, $\sigma_i = 1$. Otherwise, $\sigma_i = 0$. Then, the following constraints must be introduced:

[2] the concept is similar to the big M method for solving linear programming problems [2, 3]

$$x'_i = x_i + (1 - \sigma_i)w_i + \sigma_i h_i, \tag{16}$$
$$y'_i = y_i + (1 - \sigma_i)h_i + \sigma_i w_i. \tag{17}$$

Having made all of these considerations, we can finally produce our continuous 2DKP model. Here, s_i are the decision variables. For brevity, consider $N = \{1, \dots, n\}$.

$$\max \sum_{i \in N} v_i s_i \tag{18a}$$
$$\text{s. t.} 0 \leq x_i \leq W, \quad \forall i \in N, \tag{18b}$$
$$0 \leq x'_i \leq W, \quad \forall i \in N, \tag{18c}$$
$$0 \leq y_i \leq H, \quad \forall i \in N, \tag{18d}$$
$$0 \leq y'_i \leq H, \quad \forall i \in N, \tag{18e}$$
$$x'_i = x_i + (1 - \sigma_i)w_i + \sigma_i h_i, \quad \forall i \in N, \tag{18f}$$
$$y'_i = y_i + \sigma_i w_i + (1 - \sigma_i)h_i, \quad \forall i \in N, \tag{18g}$$
$$x'_i - M(3 - \delta_{i,j,1} - s_i - s_j) \leq x_j, \quad \forall i \in N, \forall j \in N,\ i < j, \tag{18h}$$
$$x'_j - M(3 - \delta_{i,j,2} - s_i - s_j) \leq x_i, \quad \forall i \in N, \forall j \in N,\ i < j, \tag{18i}$$
$$y'_i - M(3 - \delta_{i,j,3} - s_i - s_j) \leq y_j, \quad \forall i \in N, \forall j \in N,\ i < j, \tag{18j}$$
$$y'_j - M(3 - \delta_{i,j,4} - s_i - s_j) \leq y_i, \quad \forall i \in N, \forall j \in N,\ i < j, \tag{18k}$$
$$\sum_{k=1}^{4} \delta_{i,j,k} \geq 1, \quad \forall i \in N,\ j \in N, \tag{18l}$$
$$\delta_{i,j,k} \in \{0, 1\}, \quad \forall i \in N,\ j \in N,\ k \in \{1, 2, 3, 4\} \tag{18m}$$
$$\sigma_i \in \{0, 1\}, \quad \forall i \in N, \tag{18n}$$
$$s_i \in \{0, 1\}, \quad \forall i \in N. \tag{18o}$$

Note that we have made a few changes to Constraints (18h) to (18k). To make them even stronger, we require both items to have been selected for the constraints to take effect. Moreover, since these constraints go both ways, it is only necessary to evaluate them for $i < j$.

We have written an implementation of the model proposed by Kalvelagen [14], which is an equivalent to Mixed-Integer Program (18) that is more geared towards computational work, making heavy use of matrices for conciseness.

Kalvelagen [14] also made a discrete 2DKP model, referred to as *covering model*. Say each item's width and height is represented by continuous variables $\tilde{w}_i$ and $\tilde{h}_i$. We begin by discretizing these quantities as $w_i = \lceil \tilde{w}_i \rceil$ and $h_i = \lceil \tilde{h}_i \rceil$. The container is discretized with $W = \lfloor \tilde{W} \rfloor$ and $H = \lfloor \tilde{H} \rfloor$.

This initial process allows for the interpretation of items as collections of cells. As shown in Figure 3, a 7.23×5.55 item (darker blue) takes 8×6 cells (lighter blue).

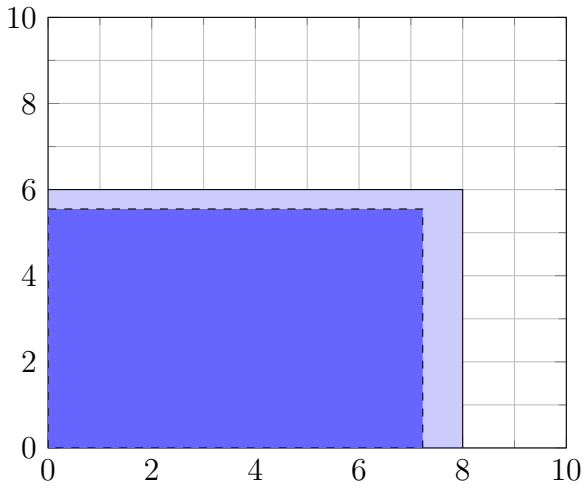


Figure 3: Discretization of an item for the 2DKP covering model.

In the covering model, we introduce binary matrix $A \in \mathbb{R}^{n \times 2 \times W \times H}$. If item k can be placed at cell (i, j) with rotation r without leaving the boundaries of the container, then $A_{k,i,j,r} = 1$. Otherwise, $A_{k,r,i,j} = 0$. By convention,

$$r = 1 \Rightarrow \text{item is not rotated},$$
$$r = 2 \Rightarrow \text{item is rotated}.$$

Next, we introduce binary matrix $C \in \mathbb{R}^{n \times 2 \times W \times H \times W \times H}$. If cells (i', j') are covered as a result of item k being placed at cell (i, j) with rotation r, then $C_{k,r,i,j,i',j'} = 1$. Otherwise, $C_{k,r,i,j,i',j'} = 0$.

Lastly, we make the decision variables $s_{k,r,i,j}$, since item placement is based on the top-left cell of each item. This is enough to formulate Integer Program (19).

$$\max \sum_{n,r,i,j | A_{n,r,i,j}=1} v_i s_{k,r,i,j} \tag{19a}$$

$$\text{s. t.} \sum_{k,r,i,j|C_{k,r,i,j,i',j'}=1} s_{k,r,i,j} \leq 1, \quad \forall i' \in \{1,\ldots,W\},\ j' \in \{1,\ldots,H\}, \tag{19b}$$

$$\sum_{r,i,j|A_{k,r,i,j}=1} s_{k,r,i,j} \leq 1, \quad \forall k \in \{1,\ldots,n\} \tag{19c}$$

$$s_{k,r,i,j} \in \{0,1\}. \tag{19d}$$

We have also written an implementation of this model. The model we implemented is the one presented by Kalvelagen [14], which is more memory-efficient than Integer Program (19), since it groups items of the same type together in the matrices.

Example 2.3. Say we wish to place as many items as possible in a 30×20 container, from the following list:

type	**width**	**height**	**quantity**
1	9	6	4
2	7	7	3
3	5	10	4
4	14	14	2
5	6	5	6
6	3	8	5

Since the idea is to maximize the number of items in the container, we consider all the items to be of equal value. By applying these data to Integer Program (19), we obtain the packing in Figure 4.

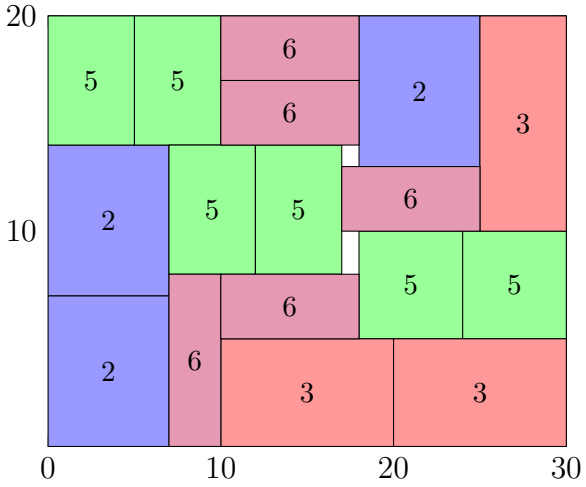


Figure 4: Covering solution to a 2D knapsack example.

2.3 The three-dimensional knapsack problem

With similar ideas to those behind Mixed-Integer Program (18) and Integer Program (19), a three-dimensional model is possible. Mixed-Integer Program (20) is one such model, presented by Egeblad and Pisinger [9].

The container in this problem has volume $W \times H \times D$ (width, height and depth). Similarly, each item has associated dimensions (w_i, h_i, d_i) and is positioned at coordinates (x_i, y_i, z_i).

To handle no-overlap, this model has six binary variables that state where item i is in regards to item j: ℓ_{ij} (left), r_{ij} (right), u_{ij} (under), o_{ij} (over), b_{ij} (behind), f_{ij} (in front of).

Positioning constraints in this model are similar to Mixed-Integer Program (18). For example, if $\ell_{ij} = 1$, then Constraints (20c) may be rewritten as $x_i + w_i \leq x_j$.

It should be noted that Constraint (20b) does *not* work in an actual implementation. This constraint only makes sense if at least i or j is selected, and will result in an unfeasible model if $s_i + s_j = 0$. This is why we rewrite this constraint as $\ell_{ij} + r_{ij} + u_{ij} + o_{ij} + b_{ij} + f_{ij} = s_i s_j$ in our implementation of this model.

One notable limitation of this model is that it does not support item rotations.

$$\max \sum_{i=1}^{n} v_i s_i \tag{20a}$$

$$\text{s. t.} \ell_{ij} + r_{ij} + u_{ij} + o_{ij} + b_{ij} + f_{ij} = s_i + s_j - 1,\ i, j \in \{1, \ldots, n\}, \tag{20b}$$

$$x_i - x_j + W\ell_{ij} \leq W - w_i,\ i, j \in \{1, \ldots, n\}, \tag{20c}$$

$$x_j - x_i + Wr_{ij} \leq W - w_i,\ i, j \in \{1, \ldots, n\}, \tag{20d}$$

$$y_i - y_j + Hu_{ij} \leq H - h_i,\ i, j \in \{1, \ldots, n\}, \tag{20e}$$

$$y_j - y_i + Ho_{ij} \leq H - h_i,\ i, j \in \{1, \ldots, n\}, \tag{20f}$$

$$z_i - z_j + Db_{ij} \leq D - d_i,\ i, j \in \{1, \ldots, n\}, \tag{20g}$$

$$z_j - z_i + Df_{ij} \leq D - d_i,\ i, j \in \{1, \ldots, n\}, \tag{20h}$$

$$0 \leq x_i \leq W - w_i,\ i \in \{1, \ldots, n\}, \tag{20i}$$

$$0 \leq y_i \leq H - h_i,\ i \in \{1, \ldots, n\}, \tag{20j}$$

$$0 \leq z_i \leq Z - d_i,\ i \in \{1, \ldots, n\}, \tag{20k}$$

$$\ell_{ij}, r_{ij}, u_{ij}, o_{ij}, b_{ij}, f_{ij} \in \{0, 1\},\ i \in \{1, \ldots, n\}, \tag{20l}$$

$$s_i \in \{0, 1\},\ i \in \{1, \ldots, n\}, \tag{20m}$$

$$x_i, y_i, z_i \geq 0,\ i \in \{1, \ldots, n\}. \tag{20n}$$

Testing this model with a method developed to visualize solutions to this formulation of the 3D knapsack, we obtain results such as the one in Figure 5 for a $20 \times 20 \times 20$ container and the items in Table 2. The best solution includes everything, except for the black item.

Table 2: Set of items for the 3DKP.

color	length	width	height	quantity	value
red	2	4	5	6	10
green	6	8	4	2	12
blue	5	10	7	2	15
gray	2	2	2	4	5
white	8	9	10	6	13
black	12	12	12	1	20
olive	3	7	10	2	15
teal	4	8	6	3	11

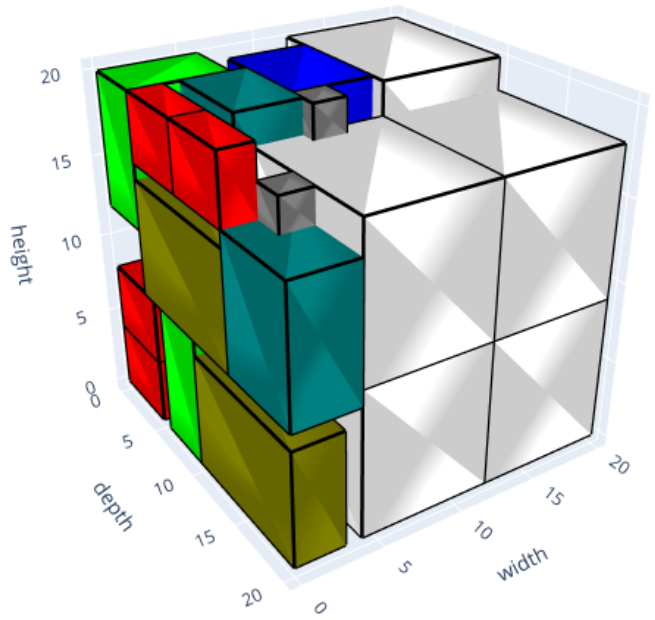


Figure 5: Solution example to a 3D knapsack problem.

2.4 Container loading problems

When the KP is generalized to multiple containers of varying volumes, it becomes a multi-container loading problem[3] (CLP). A model for solving such problems has been proposed by Chen, Lee, and Shen [6]. The model concerns m containers and N items. To each item are assigned positioning variables (x_i, y_i, z_i) and dimensions (p_i, q_i, r_i) representing length, width and height, respectively. Much like Mixed-Integer Program (20), this model uses binary variables to represent the placement of item i relative to item k: a_{ik} (left), b_{ik} (right), c_{ik} (behind), d_{ik} (front), e_{ik} (below), f_{ik} (above).

If item i is assigned to container j, then $s_{ij} = 1$; otherwise, $s_{ij} = 0$. Rotations are introduced to the model through nine binary variables: $l_{xi}, l_{yi}, l_{zi}, w_{xi}, w_{yi}, w_{zi}, h_{xi}, h_{yi}, h_{zi}$. These variables determine which axis an item's length, width or height is parallel to. For example, if $l_{xi} = 1$, then item i has its length parallel to the X axis.Evidently, the rotation variables must obey Constraint Set (21).

$$l_{xi} + l_{yi} + l_{zi} = 1, \tag{21a}$$
$$w_{xi} + w_{yi} + w_{zi} = 1, \tag{21b}$$
$$h_{xi} + h_{yi} + h_{zi} = 1, \tag{21c}$$
$$l_{xi} + w_{xi} + h_{xi} = 1, \tag{21d}$$
$$l_{yi} + w_{yi} + h_{yi} = 1, \tag{21e}$$
$$l_{zi} + w_{zi} + h_{zi} = 1. \tag{21f}$$

From this set of equations, Chen, Lee, and Shen [6] obtain the following equalities:

$$l_{yi} = 1 - l_{xi} - l_{zi}, \tag{22a}$$
$$w_{xi} = l_{zi} - w_{yi} + h_{zi}, \tag{22b}$$
$$w_{zi} = 1 - l_{zi} - h_{zi}, \tag{22c}$$
$$h_{xi} = 1 - l_{xi} - l_{zi} + w_{yi} - h_{zi}, \tag{22d}$$
$$h_{yi} = l_{xi} + l_{zi} - w_{yi}. \tag{22e}$$

The model that Chen, Lee, and Shen [6] propose is Mixed-Integer Program (23). By replacing the rotation variables with the equations in Equation Set (22), this model

[3] When these dimensions are the same for all containers, the KP becomes a bin-packing problem.

becomes simpler to solve.

$$\min \sum_{j=1}^{m} L_j W_j H_j n_j - \sum_{i=1}^{N} p_i q_i r_i \tag{23a}$$

$$\text{s. t.} x_i + p_i l_{xi} + q_i w_{xi} + r_i h_{xi} \leq x_k + (1 - a_{ik})M,\ \forall i, k,\ i < k, \tag{23b}$$

$$x_k + p_k l_{xk} + q_k w_{xk} + r_k h_{xk} \leq x_i + (1 - b_{ik})M,\ \forall i, k,\ i < k, \tag{23c}$$

$$y_i + q_i w_{yi} + p_i l_{yi} + r_i h_{yi} \leq y_k + (1 - c_{ik})M,\ \forall i, k,\ i < k, \tag{23d}$$

$$y_k + q_k w_{yk} + p_k l_{yk} + r_k h_{yk} \leq y_i + (1 - d_{ik})M,\ \forall i, k,\ i < k, \tag{23e}$$

$$z_i + r_i h_{zi} + q_i w_{zi} + p_i l_{zi} \leq z_k + (1 - e_{ik})M,\ \forall i, k,\ i < k, \tag{23f}$$

$$z_k + r_k h_{zk} + q_k w_{zk} + p_k l_{zk} \leq z_i + (1 - f_{ik})M,\ \forall i, k,\ i < k, \tag{23g}$$

$$a_{ik} + b_{ik} + c_{ik} + d_{ik} + e_{ik} + f_{ik} \geq s_{ij} + s_{kj} - 1,\ \forall i, k, j,\ i < k \tag{23h}$$

$$\sum_{j=1}^{m} s_{ij} = 1,\ \forall i, \tag{23i}$$

$$\sum_{i=1}^{N} s_{ij} \leq M n_j,\ \forall j, \tag{23j}$$

$$x_i + p_i l_{xi} + q_i w_{xi} + r_i h_{xi} \leq L_j + (1 - s_{ij})M,\ \forall i, j, \tag{23k}$$

$$y_i + q_i w_{yi} + p_i l_{yi} + r_i h_{yi} \leq W_j + (1 - s_{ij})M,\ \forall i, j, \tag{23l}$$

$$z_i + r_i h_{zi} + q_i w_{zi} + p_i l_{zi} \leq H_j + (1 - s_{ij})M,\ \forall i, j, \tag{23m}$$

$$l_{xi}, l_{yi}, l_{zi}, w_{xi}, w_{yi}, w_{zi}, h_{xi}, h_{yi}, h_{zi} \in \{0, 1\},\ \forall i \tag{23n}$$

$$a_{ik}, b_{ik}, c_{ik}, d_{ik}, e_{ik}, f_{ik}, s_{ij}, n_j \in \{0, 1\},\ \forall i \tag{23o}$$

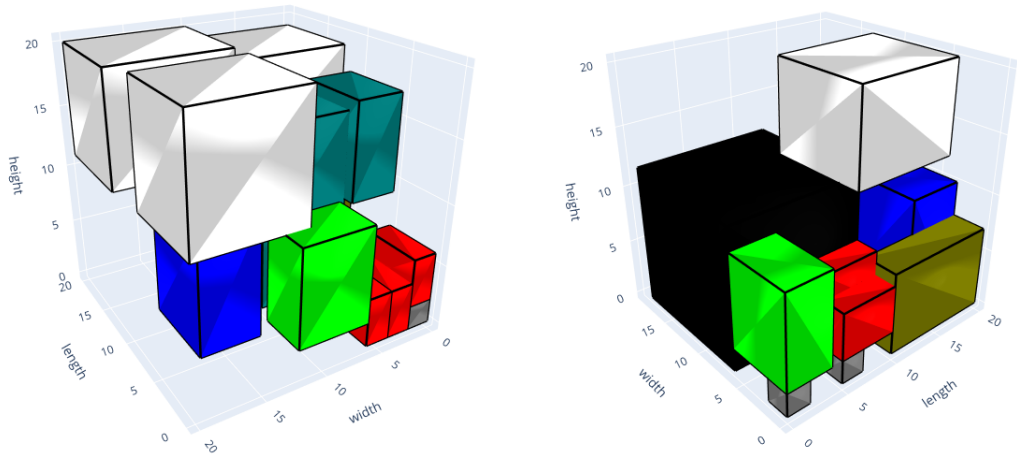
$$x_i, y_i, z_i \geq 0. \tag{23p}$$

Expression (23a) defines the CLP's objective to minimize the container volume that goes unused. Decision variables n_j indicate whether a container has been selected or not. Constraints (23b) to (23g) establish the relative placements of items in container j, in the same vein as Mixed-Integer Program (20). To guarantee that these alternative constraints work, Constraints (23h) force at least one of the positional variables to equal one, whenever two items share the same container.

Constraints (23i) make it so every item is assigned to exactly one container. Constraints (23j) ensure that items are only assigned to selected containers. Finally, Constraints (23k) to (23m) make it impossible for items to breach their container's boundaries.

Our implementation of this model and a visualization method allows us to verify

that this CLP model works when tested with the item set in Table 2 and two $20 \times 20 \times 20$ containers. Figure 6 shows the solution to this problem.



(a) Container 1 (b) Container 2

Figure 6: Containers in an example solution to the CLP.

2.5 Complexity of exact solutions

The BKP is an NP-hard problem [17], which means that no algorithm is known that can solve it in polynomial time, and that if such an algorithm can be found, then all other problems in the NP category can also be solved in polynomial time (P = NP) [10].

The dynamic programming solution to the BKP (Algorithm 2) might elicit doubt about this affirmation. Recall that this algorithm creates a table that grows according to nC, and then updates each entry once. Then, the algorithm performs even fewer steps of backtracking to determine which items constitute the optimal solution. Hence, the algorithm's time complexity is $\mathcal{O}(nC)$.

However, for an algorithm to be *truly* polynomial, it must also be polynomial when the *length in bits* of the input is used as a metric. Otherwise, we have only a *pseudo-polynomial* algorithm [22], bounded above by a unary representation of the problem's values [10, 20]. As this is the case for dynamic programming approaches to the BKP, no true polynomial time algorithm is known for it [8, 17].

Example 2.4. We will show that the dynamic programming approach is pseudo-polynomial. Consider a BKP with $n > 0$ (the specific value is unimportant), and let us vary the value of C in powers of 2, as shown in Table 3.

By counting, we see that there is a one-to-one relation between runtime and the

Table 3: Comparison between unary and binary input lengths regarding runtime.

C			Runtime
Decimal	Unary	Binary	
2	11	10	2
4	1111	100	4
8	11111111	1000	8
16	1111111111111111	10000	16

number of ones in a unary representation of C. Hence, the dynamic programming solution is polynomial in this sense. For b binary digits, though, runtime is 2^{b-1}, which is exponential behavior. Figure 7 illustrates both behaviors for larger values of C.

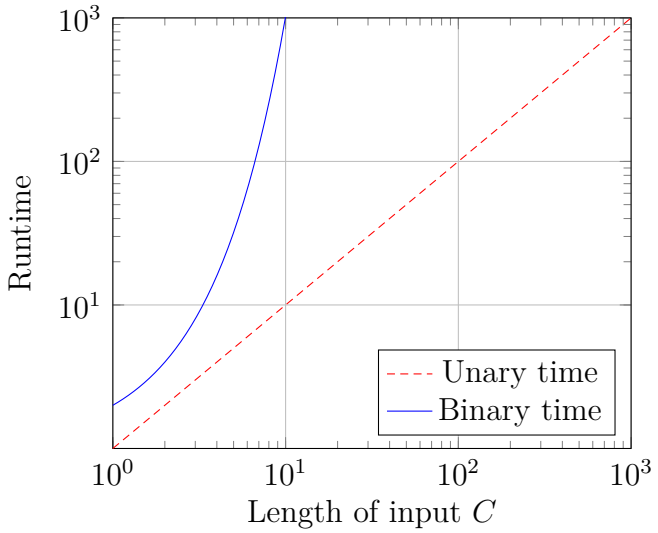


Figure 7: Unary vs binary time for the dynamic programming solution to the BKP

The 2D, 3D and any multidimensional KP are *strongly* NP-hard [5, 16], which means that not even a pseudo-polynomial algorithm is known for them.

To further complicate this matter, exact methods are notably inefficient at solving large-scale packing problems [17–19]. This is due to a multitude of reasons, the first being that KPs and CLPs make heavy use of binary (integer) variables to decide which items to select, which rotations to apply, where an item stands compared to another, etc. Bradley, Hax, and Magnanti [4] show the issue with this through the following example. Consider the problem

$$\max z = 5x_1 + 8x_2, \tag{24a}$$

$$\text{s. t. } x_1 + x_2 \leq 6, \tag{24b}$$

$$5x_1 + 9x_2 \leq 45, \tag{24c}$$

$$x_1, x_2 \in \mathbb{N}_0. \tag{24d}$$

Figure 8 shows part of the steps towards an exact solution to this problem, using a branch-and-bound approach. First, the problem is relaxed, which means Constraints (24d) become $x_1, x_2 \in \mathbb{R}_0^+$. The problem is then solved with a linear programming method (e.g. simplex), yielding solution $(2.25, 3.75)$. This is not a solution to the original problem, and no reliable rounding procedure exists in this circumstance, since the optimal integer solution is at $(0, 5)$. So we must keep searching. Since both coordinates of the optimal solution to the relaxation are fractional, four subproblems may be derived from the original problem, by introducing constraints $x_1 \leq 2$, $x_1 \geq 3$, $x_2 \leq 3$ and $x_2 \geq 4$ to the problem. From this, we arrive at the four new polyhedra in Figure 9.

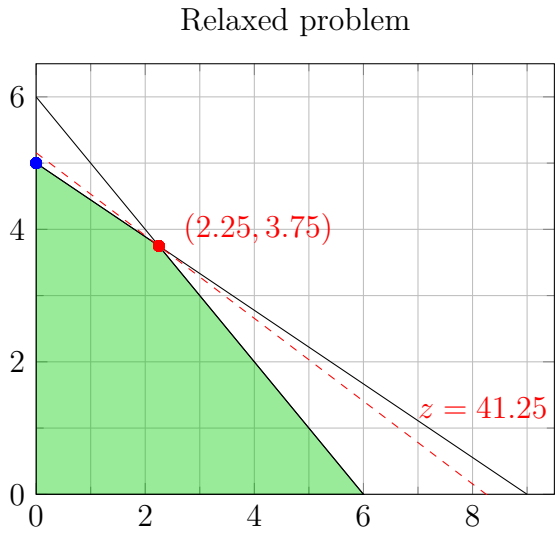


Figure 8: Linear programming relaxation

Effectively, a tree is generated with subproblems for nodes. Since the subproblems themselves can be cut into other subproblems, a real problem's tree may quickly grow thousands of nodes. Though there are procedures to determine if the integer solution to a subproblem is an optimal solution to the original problem, this does not guarantee that exact solutions can be found in a reasonable amount of time.

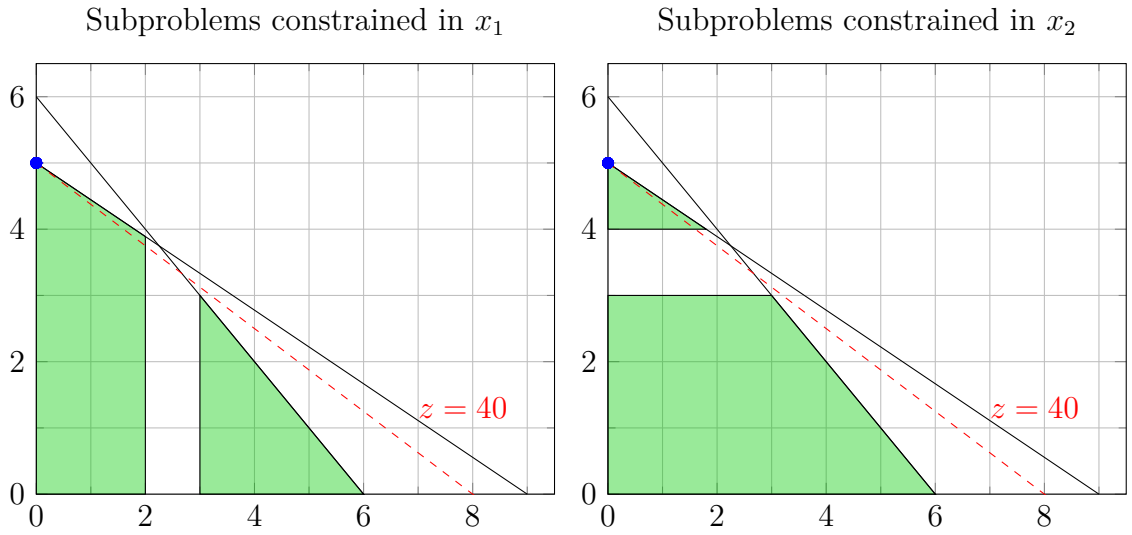


Figure 9: Branch-and-bound subproblems

One particularly bad characteristic of the continuous 2DKP, the 3DKP and the CLP formulations presented in this section (Mixed-Integer Programs (18), (20) and (23)) is that their alternative constraints rely on big M constants that reduce the models' *tightness*. As Egeblad and Pisinger [9] remark, "These [constraints] will loose their effect when solving the LP-relaxation, and thus bounds from LP-relaxation are in general far from the MIP-optimal solution value."

An important consequence of these issues is that contemporary approaches to real KP and CLP problems involve the use of heuristics to determine near-optimal solutions in an adequate amount of time. Nonetheless, exact methods can still aid heuristics by providing important solution bounds that can be found quickly [9, 18].

3 Heuristics for packing problems

We now look at heuristics for the CLP.

3.1 Wall-building

This constructive heuristic was proposed by George and Robinson [12], aiming to solve the CLP for a single container. In each iteration of the wall-building heuristic (WBH), what remains of the container is sliced by a plane parallel to its width and height, up to a certain depth. The resulting space is called a layer of the container.

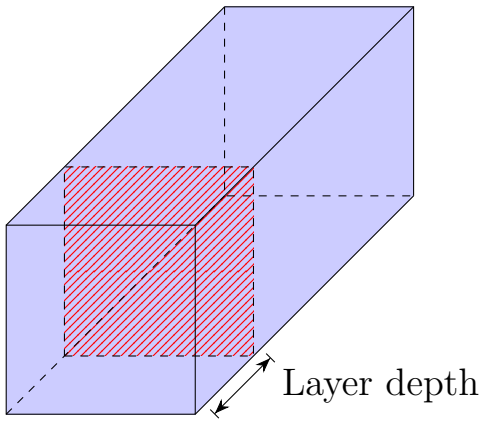
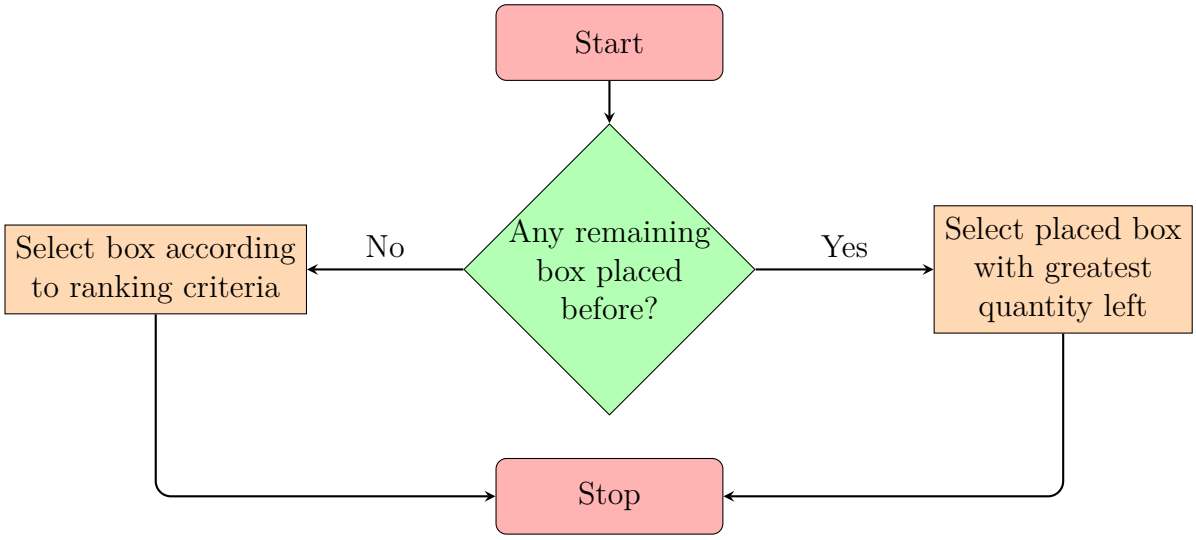


Figure 10: Creation of a layer inside a container

To choose the depth of a new layer, we must first select a box, which we will call the layer's primary box. The primary box is selected as detailed in Flowchart 11. Notice that we use the term "box" to refer to box types – that is, the set of dimensions that define a box's volume. Hence, a box's "quantity" is the amount of boxes of said type that has yet to be packed.

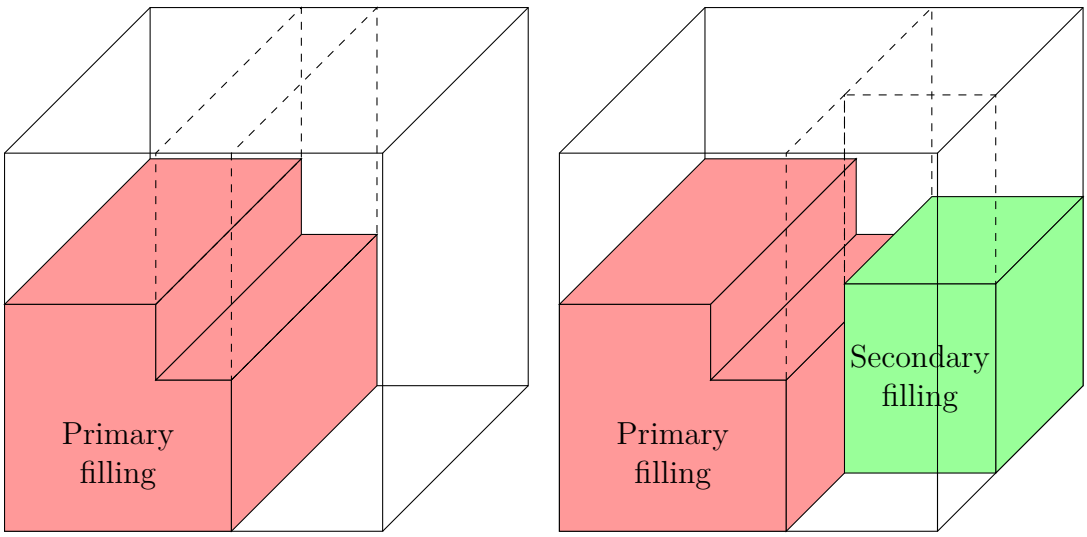
The "ranking criteria" in Flowchart 11 are as follows: First, we filter the boxes with the greatest smallest dimension. In case of ties, we filter the selected box with the greatest remaining quantity to pack. If ties persist, we choose one of the boxes with the greatest dimension. The intuition behind these choices is that it is better to pack larger items first, to avoid complications trying to fit them with smaller items later on. It is also preferable to pack numerous boxes first, in order to reduce their numbers quickly.

Layers provide an initial space into which boxes can be placed. For this initial space, the chosen box to start the filling process is the primary box, since it can be rotated to perfectly fit the space's depth. Once this space is filled with as many primary boxes as possible, certain unfilled spaces might remain. Figure 12a illustrates this situation. The



Flowchart 11: Primary box selection procedure

two spaces on top of the boxes are called height-wise spaces, while the space to the right is called width-wise space. If, for example, we fill any of the remaining spaces, new unfilled spaces might appear. Figure 12b shows this situation. Notice that an unfilled space in front of the filling appears – this is called a depth-wise space.



(a) Unfilled spaces around primary filling (b) Unfilled spaces around secondary filling

Figure 12: Remaining spaces in the first filling of a layer

To figure out whether a space should be stored in memory for filling, we only need to check if its height or its width (in the case of a height- or width-wise space, respectively) is greater than the smallest dimension of any remaining box. Depth-wise spaces are created

regardless of any such check, because they may be *amalgamated* later on. For example, consider the top view of two layers in Figure 13. The previous layer was packed with two depth-wise spaces left unfilled. We select the depth-wise space with minimal depth to amalgamate with the space in the current layer. The original space is reduced to space A, with the amalgam $B \cup C$ to its right. Space B is defined by a *flexible width* parameter, which allows the algorithm to fill space A past its boundaries, as long as its last column of boxes is both in A and B, but not in C. In such circumstances, the width of A is increased and the width of the amalgam is decreased according to the length of flexible width consumed by this last column of boxes.

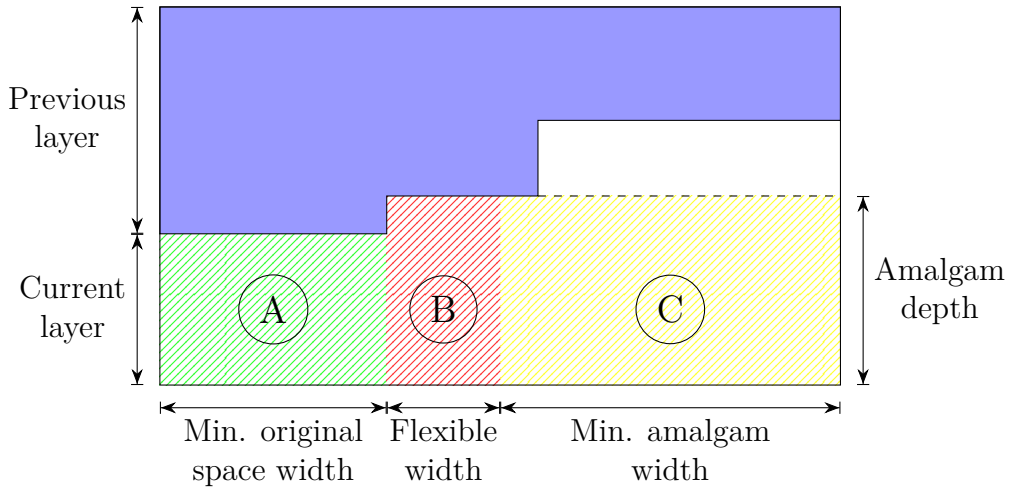


Figure 13: Amalgamation procedure

It should be noted that in this example we assume the depth-wise space chosen is at the same height or lower than the current layer's space. Otherwise, it does not make sense to attempt to amalgamate them.

Finally, we discuss this heuristic's packing method. First, we determine if a space can be filled by filtering the box rotations that fit in it. If there is any, then there are two scenarios:

1. There are box rotations whose width is at most half of the space's width: Choose one such box with maximum depth.

2. Otherwise: Choose a box rotation with maximum width-by-depth area.

Both selections try to minimize the unused base area of the space. In particular, the logic behind scenario 1 is that such boxes can be stacked in columns side by side, so we count the width-by-depth area of all the columns of boxes. After making the selection, we fill the space according to Flowchart 14.

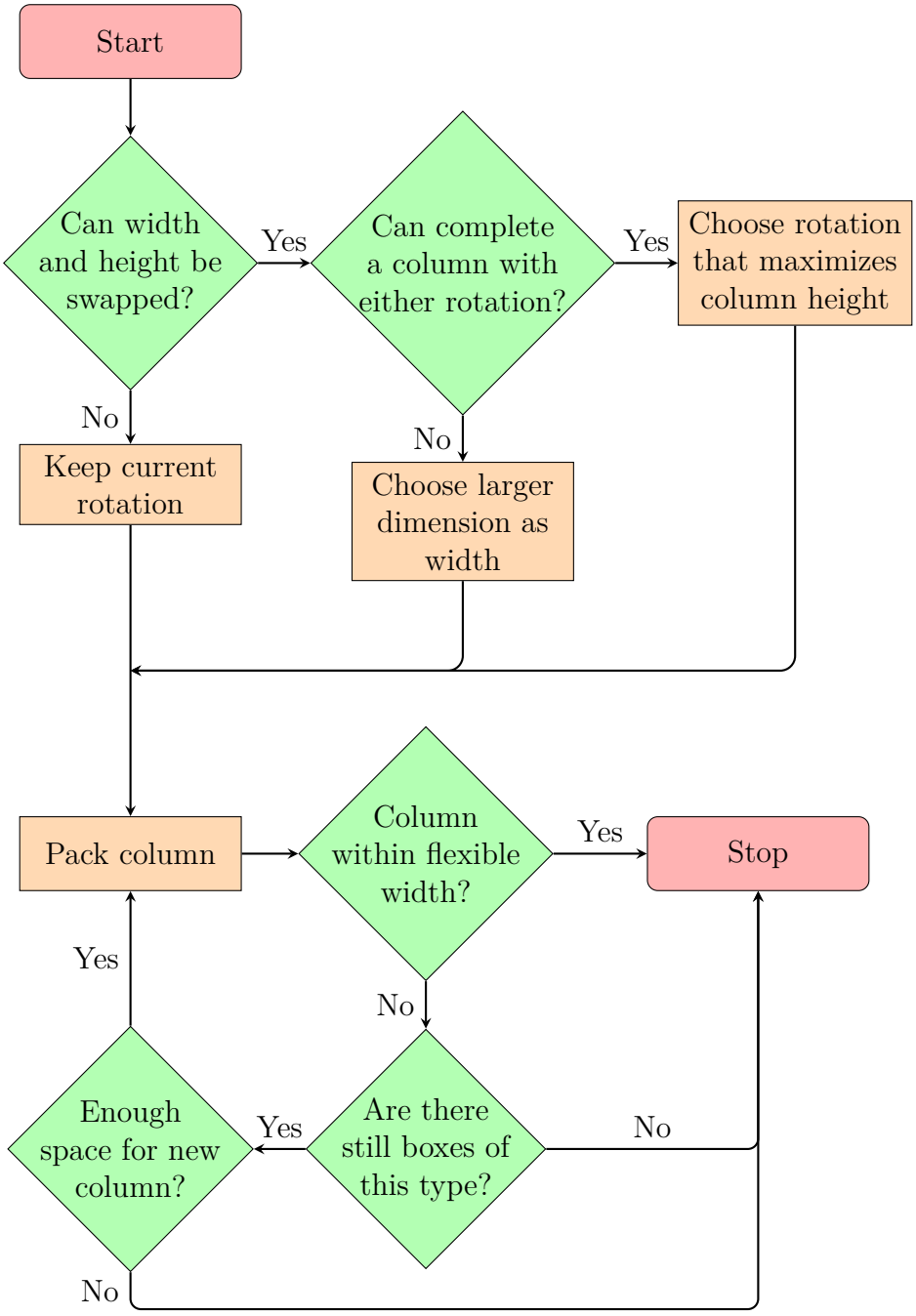


Figure 14: Space filling procedure

References

[1] Maram Assi and Ramzi A Haraty. "A survey of the knapsack problem." In: *2018 International Arab Conference on Information Technology (ACIT)*. IEEE. 2018, pp. 1–6.

[2] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows*. 4th ed. Hoboken, New Jersey: Wiley, 2010.

[3] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. 4th ed. Belmont, Massachusetts: Athena Scientific and Dynamic Ideas, 1997.

[4] Stephen P Bradley, Arnoldo C Hax, and Thomas L Magnanti. "Applied mathematical programming." In: *(No Title)* (1977).

[5] Alberto Caprara and Michele Monaci. "On the two-dimensional Knapsack Problem." In: *Operations Research Letters* 32.1 (2004), pp. 5–14. DOI: https://doi.org/10.1016/S0167-6377(03)00057-9. URL: https://www.sciencedirect.com/science/article/pii/S0167637703000579.

[6] Chin-Sheng Chen, Shen-Ming Lee, and QS Shen. "An analytical model for the container loading problem." In: *European Journal of operational research* 80.1 (1995), pp. 68–76.

[7] Erik Demaine and Srini Devadas. *Polynomial Time vs Pseudo-Polynomial Time*. 2011. URL: https://courses.csail.mit.edu/6.006/fall11/rec/rec21_knapsack.pdf (visited on 02/09/2024).

[8] Andrew Dudley. *Why is the knapsack problem pseudo-polynomial?* 2016. URL: https://www.youtube.com/watch?v=9oI7fg-MIpE (visited on 02/17/2024).

[9] Jens Egeblad and David Pisinger. "Heuristic approaches for the two-and three-dimensional knapsack packing problem." In: *Computers & Operations Research* 36.4 (2009), pp. 1026–1049.

[10] Jeff Erickson. *Algorithms*. 1st ed. 2019. URL: http://jeffe.cs.illinois.edu/teaching/algorithms/.

[11] Paulo Feofiloff. *Mochila booleana*. 2020. URL: https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-bool (visited on 02/09/2024).

[12] John A George and David F Robinson. "A heuristic for packing boxes into a container." In: *Computers & Operations Research* 7.3 (1980), pp. 147–156.

[13] Maya Hristakeva and Dipti Shrestha. "Different approaches to solve the 0/1 knapsack problem." In: *The Midwest Instruction and Computing Symposium*. 2005.

[14] Erwin Kalvelagen. *2d knapsack problem*. 2021. URL: http://yetanothermathprogrammingconsult blogspot.com/2021/10/2d-knapsack-problem.html (visited on 02/10/2024).

[15] Soukaina Laabadi, Mohamed Naimi, Hassan El Amri, Boujemâa Achchab, et al. "The 0/1 multidimensional knapsack problem and its variants: A survey of practical models and heuristic approaches." In: *American Journal of Operations Research* 8.05 (2018), p. 395.

[16] Hanan Mostaghimi Ghomi. "Three-Dimensional Knapsack Problem with Pre-Placed Boxes and Vertical Stability." In: (2013). URL: https://scholar.uwindsor.ca/cgi/viewcontent.cgi?article=5986&context=etd.

[17] David Pisinger. "Where are the hard knapsack problems?" In: *Computers & Operations Research* 32.9 (2005), pp. 2271–2284.

[18] Luiz FO Moura Santos et al. "A variable neighborhood search algorithm for the bin packing problem with compatible categories." In: *Expert Systems with Applications* 124 (2019), pp. 209–225.

[19] Yu-Chung Tsao, Jo-Ying Tai, Thuy-Linh Vu, and Tsung-Hui Chen. "Multiple bin-size bin packing problem considering incompatible product categories." In: *Expert Systems with Applications* 247 (2024), p. 123340. DOI: https://doi.org/10.1016/j.eswa.2024.123340. URL: https://www.sciencedirect.com/science/article/pii/S0957417424002057.

[20] Vijay V Vazirani. *Approximation algorithms*. Vol. 1. Springer, 2001.

[21] David B Wagner. "Dynamic programming." In: *The Mathematica Journal* 5.4 (1995), pp. 42–51.

[22] Alex Wendland. *Pseudo-polynomial time*. URL: https://obsidian.awendland.co.uk/general/Pseudo-polynomial+time (visited on 02/17/2024).