Update: We have released an open-source version of the algorithm at
https://github.com/JeroenGar/gdrr-2bp

A Goal-Driven Ruin and Recreate Heuristic for the 2D Variable-Sized Bin Packing Problem with Guillotine Constraints

Jeroen Gardeyn[a,*], Tony Wauters[a]

[a]*KU Leuven, Department of Computer Science, NUMA, Belgium*

Abstract

This paper addresses the two-dimensional bin packing problem with guillotine constraints. The problem requires a set of rectangular items to be cut from larger rectangles, known as bins, while only making use of edge-to-edge (guillotine) cuts. The goal is to minimize the total bin area needed to cut all required items. This paper also addresses variants of the problem which permit 90° rotation of items and/or a heterogeneous set of bins. A novel heuristic is introduced which is based on the ruin and recreate paradigm combined with a goal-driven approach. When applying the proposed heuristic to benchmark instances from the literature, it outperforms the current state-of-the-art algorithms in terms of solution quality for all variants of the problem considered.

Keywords: Packing, 2D Bin Packing, Guillotine, Heuristic, Variable-sized bins

1. Introduction

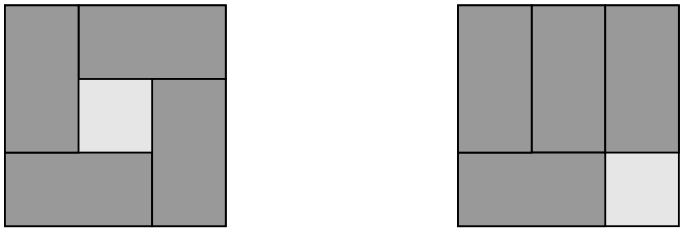
The Two-Dimensional Bin Packing Problem (2BP) consists of packing a heterogeneous set of small rectangular items into larger rectangular bins. In general, a 2BP solution is considered feasible if (1) the items are fully inside the bin, (2) items are not overlapping and (3) the edges of the items are parallel to the edges of the bin. The goal is to minimize the amount of unused bin area incurred by packing all the items.

Lodi et al. [21] defined four variants of the 2BP problem using a three-field notation. This notation defines whether the items are allowed to rotate 90 degrees and/or whether guillotine cuts are required. This paper focuses on variants of the 2BP which require guillotine cuts (2BP|*|G). Both fixed orientation (2BP|O|G) and the variant that allows 90° rotation of items (2BP|R|G) are addressed.

The guillotine constraint defines that all cuts must be edge-to-edge cuts. Figure 1(a) shows a simple visual example of a pattern which does not comply with the guillotine constraint. It is not possible to cut this pattern by exclusively using edge-to-edge cuts. The pattern in Figure 1(b), on the other hand, satisfies the guillotine constraint by first making a horizontal cut followed by several vertical cuts. Guillotine cuts have many real-world applications as they are often required in glass- and wood-cutting industries.

*Corresponding author

Email addresses: jeroen.gardeyn@kuleuven.be (Jeroen Gardeyn), tony.wauters@kuleuven.be (Tony Wauters)



(a) Non-guillotine pattern (b) Guillotine pattern

Figure 1: Visual representation of non-guillotine and guillotine patterns

This work introduces a novel heuristic based on the ruin and recreate paradigm combined with a goal-driven approach (GDRR). The heuristic attempts to improve the solution at each iteration by removing and reinserting items into the bins in a greedy fashion. GDRR is goal-driven in the sense that it iteratively lowers the available bin area. Each time the algorithm finds a feasible solution with a certain total bin area, it is henceforth forced to find new solutions which use less bin area. As a result, most of the time, the heuristic will be unable to fit all items into the available bins and will therefore be working with infeasible solutions. After each run, items which could not be placed are automatically reconsidered for reinsertion during the next run. In order to reach complete solutions, the objective function penalizes unassigned items in order to steer the heuristic towards feasibility.

GDRR is also capable of addressing the variant with a heterogeneous set of bins. This generalization is commonly referred to as the 2BP with variable-sized bins. Since the classification made by Lodi et al. [21] does not cover this variant, it will henceforth be referred to as the 2VSBP.

In summary, the proposed heuristic addresses four variants of the 2BP:

- **2BP|O|G**: identical bins, no rotation
- **2BP|R|G**: identical bins, 90° rotation
- **2VSBP|O|G**: heterogeneous set of bins (variable-sized), no rotation
- **2VSBP|R|G**: heterogeneous set of bins (variable-sized), 90° rotation

The remainder of the paper is organized as follows. In Section 2 a compact literature review is provided which includes papers covering both the traditional 2BP and the variant with a heterogeneous set of bins. Section 3 introduces the data structure for solution representation. Next, a detailed explanation of the complete ruin and recreate heuristic is given in Section 4. A comparison of the heuristic against the current state of the art is conducted in Section 5. Finally, Section 6 summarizes the paper and highlights some future research directions.

2. Literature review

General typologies for cutting and packing problems are published in Dyckhoff [12] and Wäscher et al. [30]. Surveys for the 2BP have been provided by Lodi et al. [22] and

Lodi et al. [20]. These surveys cover heuristics, exact methods and bounds for the 2BP. A more recent survey focusing on exact methods was conducted by Iori et al. [18].

Gilmore and Gomory [15] were the first to tackle the 2BP and proposed a column generation approach for the multidimensional bin packing problem. Many heuristics for 2BP|*|G have since been published in the literature and this review will focus on the most competitive among these. Polyakovsky and M'Hallah [29] developed a 'guillotine bottom left' constructive heuristic to pack one bin at a time in combination with a pseudo-parallel agent-based system wherein each agent has its own characteristics. Later, Charalambous and Fleszar [5] introduced a constructive heuristic which supports item rotation. At each iteration, their method packs one bin by creating multiple simple cutting patterns and selects the one with the highest quality by way of a sufficiency criterion. To improve solution quality, they proposed a post-optimization routine which prioritizes items that most often do not fit during the constructive phase. Later, Fleszar [13] proposed three new insertion heuristics and a justification improvement heuristic. Lodi et al. [23] introduced a heuristic based on an enumeration tree for the 2BP|O|G. At each level of the tree, a new bin is packed. When filling a new bin, a set of selection and guillotine split rules are considered. These different strategies create new nodes in the tree. In an enhanced version of the heuristic, they attempt to reduce the search space to improve performance. More recently, Cui et al. [10] present a sequential value correction heuristic which repeatedly creates cutting patterns while trying to maximize the total item value. The values of the items are updated after each iteration based on information from current and past solutions. Finally, a heuristic which uses a pattern generation procedure that creates triple-block patterns was introduced by Cui et al. [11]. The procedure is followed by an improvement phase which uses an Integer Linear Programming model to select the best patterns.

The concept of heterogeneous sets of bins (variable-sized) was first introduced by Friesen and Langston [14]. They proposed three heuristics for one-dimensional bin packing problems. Literature on the 2VSBP|*|G is more scarce compared to the variant with identical bins. To the best of our knowledge, only the variant without rotation of items has been studied. Ortmann et al. [27] proposed a two-stage heuristic, which begins with a strip packing procedure. Next, attempts are made to repack these strips into smaller and smaller bins. Later, Hong et al. [16] designed a hybrid heuristic for the 2VSBP|O|G. They proposed a mixed bin packing algorithm which is used in combination with iterative simulated annealing runs and a binary search. Finally, Wei et al. [31] introduced a heuristic with a goal-driven approach for the 2VSBP|O|F (no guillotine constraint). Their algorithm packs bins with a sequential packing heuristic, followed by a local search to improve the solution. It also performs a binary search on bin combinations and increases the search effort each time.

It should be clear that many of these solution methods are either based on exact methods or focus heavily on the constructive aspect. Exact methods, however, often lack the flexibility required to adapt to real-world scenarios and generally do not scale well for larger problems. On the other hand, although constructive heuristics are usually more flexible and very fast, their solutions are often lacking in terms of quality. The approach proposed in this paper attempts to find a balance by focusing on heuristic improvement

strategies.

3. Solution representation

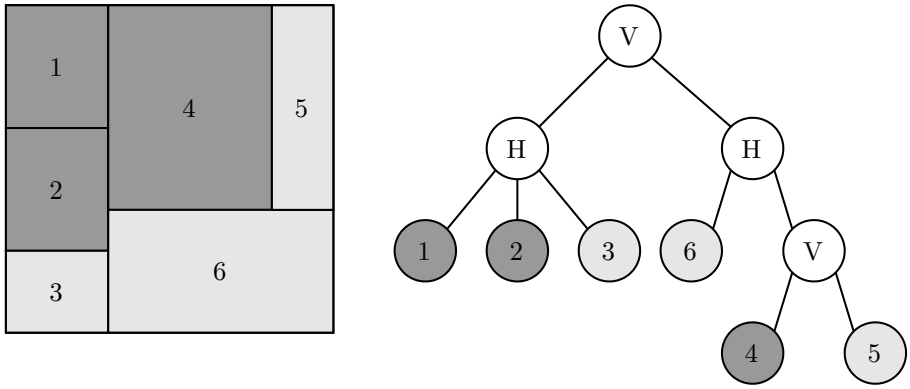


Figure 2: Visual representation of the data structure

Each cutting pattern is represented as a rooted tree, as shown in Figure 2. The tree consists of three different types of node. There are item (dark gray), leftover (light gray) and structure nodes (white). Leaves are always either an item or leftover node. All non-leaf nodes define the structure of a cutting pattern in a hierarchical way and are denoted as either vertical (V) or horizontal (H). The orientation of the node corresponds to the direction in which its children are cut. This orientation is identical for every node of a level and each new level switches the orientation compared to the previous one.

A similar, but not identical, data structure was used before by Clautiaux et al. [7], Fleszar [13] and Kröger [19]. As noted by Fleszar [13], this representation always satisfies the guillotine constraint and never allows for overlapping items.

Note that the data structure does not define any coordinates. It only describes relative positions of items and leftovers. This allows for simpler insertion or removal of items (or groups of items) compared to more rigid structures where a cutting pattern is defined by a set of items and their locations.

As was first described by Gilmore and Gomory [15], in real-world scenarios there is often a limit on the number of possible stages. A stage includes one or more horizontal or vertical cuts, but never a mixture of the two. Progressing from one stage to another involves rotating the cut orientation. The number of stages therefore represents the number of blade rotations needed to cut the pattern. In this representation, since the cutting orientations are switched at every level, each level represents a stage. While not the focus of this paper, limiting the number of stages in this data structure involves simply defining a tree's maximum depth.

Due to the nature of the heuristic, there is a need to be able to represent incomplete solutions. Therefore, a solution $S = \{C, E\}$ not only contains a set of cutting patterns C, it also consists of a set of excluded items E. A solution is deemed feasible if all items are included ($E = \varnothing$).

4. Ruin and recreate heuristic

Figure 3 provides a high-level overview of the heuristic. At each iteration of the local search, the solution is partially destroyed and rebuilt in an attempt to find improvements. Every time a new feasible solution is found, the bin area limit is reduced. The general idea behind this is discussed in Section 4.1. Detailed explanations of the ruin and recreate procedures are given in Sections 4.2 and 4.3. Afterwards, the modified solution is evaluated and has the possibility of being accepted. This is covered in Section 4.4. The entire process is repeated until a given time limit is reached.

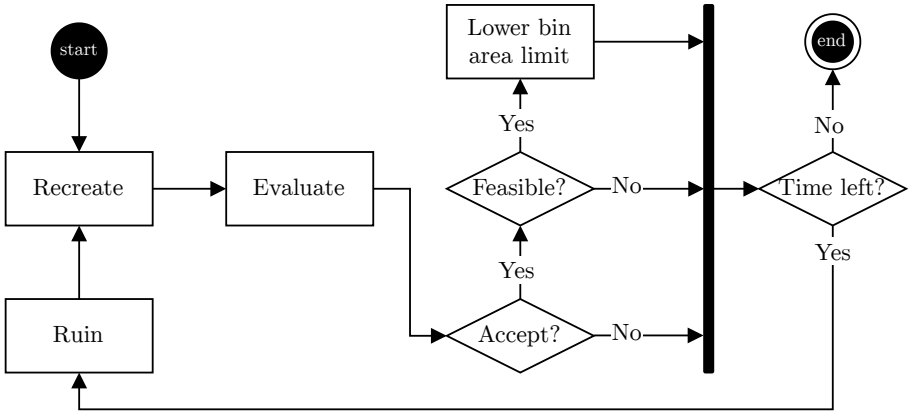


Figure 3: High-level overview of GDRR

4.1. Goal-driven approach

In the 2BP, the quality of a solution is based on the sum of areas of the bins required to pack all items. Due to the nature of the problem, all feasible solutions which result in the same total bin area are deemed to be equal in quality. This means that, once a complete solution is found with a certain total bin area, solutions with equal (or greater) total bin area are no longer of interest.

Wei et al. [31] proposed a goal-driven approach (GDA) heuristic for the 2BP with variable-sized bins. It generates all combinations of bins that have a total area between a lower and upper bound, sorts them by area and attempts to find the best combination that produces a feasible solution. The sorted list is traversed using a binary search. At each iteration, the GDA heuristic must use that specific combination of bins. Although this particular algorithm does not incorporate the guillotine constraint, the general idea is applicable.

Similar to GDA, the proposed approach could also be viewed as being goal-driven. The recreate phase, responsible for rebuilding the ruined solution, must adhere to a certain bin area limit. A new bin can only be opened if the total bin area of the solution remains below this threshold. Therefore, most of the time, the recreate phase will be unable to fit all items into the available bins and will produce infeasible solutions. A cost function guides the heuristic towards a solution capable of fitting all items.

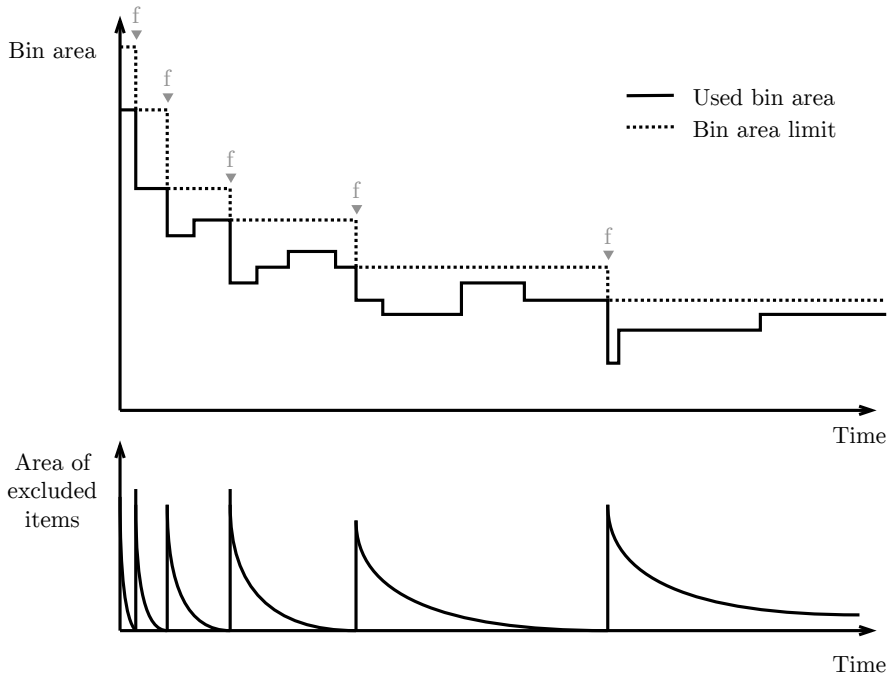


Figure 4: Progression of GDRR over time

Figure 4 represents one possible course of the heuristic over time. At all times, the goal is to find a set of cutting patterns capable of packing all items with a total bin area below the current limit. Whenever a new feasible solution (denoted by 'f') is found, the limit is lowered to the total bin area of this new solution. A formal description of the complete GDRR heuristic and how the bin area limit tightens is presented in Algorithm 3 at the end of Section 4.

Unlike Wei et al. [31], the search does not begin from scratch each time, but rather the solution is destroyed until the bin area constraint is once again satisfied. Therefore, high-quality features from a previous solution can be carried over when the limit is lowered. This can result in a significant amount of saved computational time. However, the heuristic may be more likely to get stuck in local optima when solutions are already tightly packed.

Another difference lies in the combination of available bins. GDA tries to make a feasible solution with a fixed combination of bins. By contrast, GDRR is free to use whichever combination of bins yields good results according to the cost function. GDRR only needs to ensure that the total bin area is below the current limit. Furthermore, it can sometimes happen that there is a feasible solution for a combination of bins A, but not for a combination of bins B despite the total bin area of A being smaller than B. This situation most often occurs with instances that contain small bins (few items per bin) or items with extreme dimensions. Due to the binary search nature of GDA, these combinations might not be reachable.

It should become clear that, by design, the heuristic will be working with infeasible solutions most of the time. This might seem counterproductive at first, but such a strategy

has two major advantages compared to incorporating bin area into a cost function. First of all, items which are not included in the solution have a new chance at being inserted every iteration. This drastically increases the likelihood of finding better solutions. If this were not the case, reaching complete solutions with less bin area would be unlikely, especially when the cutting patterns are complex. Second, as shown in Figure 4, the total area of excluded items can be used as a very gradual indicator of a solution's quality. This is in stark contrast to the abrupt drops when using bin area as a cost factor.

In conclusion, although the proposed heuristic is also goal-driven, it significantly differs from the approach introduced by Wei et al. [31]. They merely share the same philosophy of constraining the use of bins in some way.

4.2. Ruin

The ruin procedure removes a number of nodes from the solution. All item and structure nodes are candidates for removal. Leftover nodes are ineligible since removing them would not have any effect. They can only be deleted as a result of removing a parent node.

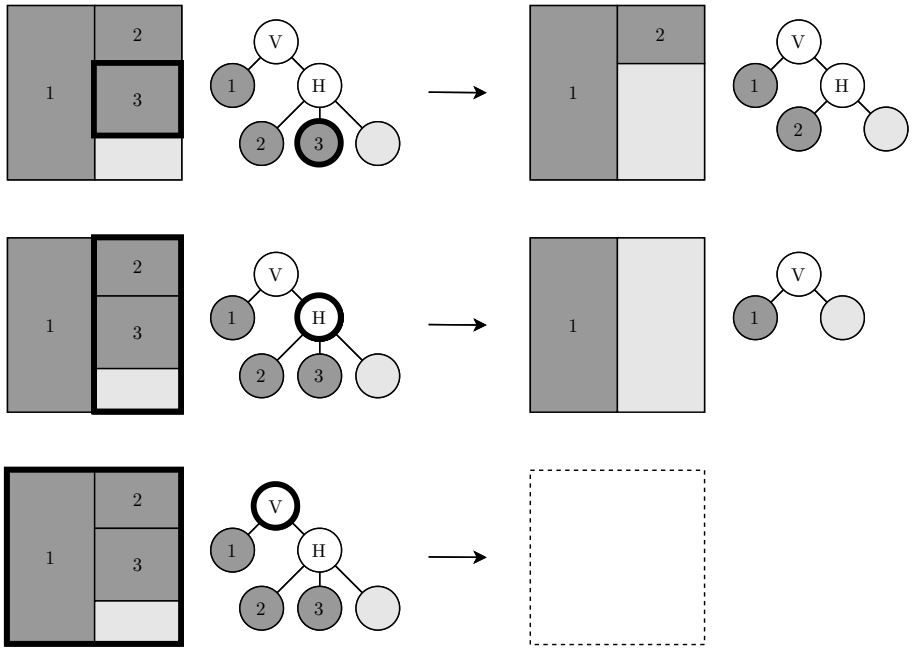


Figure 5: Three examples of node removal

Three cases of node removal are shown in Figure 5. These figures illustrate the situation before (left) and after (right) the removal of a node. In each example, the node with a bold edge is the one selected for removal. In the first example, the node containing item 3 is removed. As a result, the leftover node is expanded to account for the freed-up space. In the second example, a first-level structure node is deleted. This results in items 2 and 3 being removed since they are children. The final example shows the root node being selected for removal. This results in the entire cutting pattern being removed. It should

be clear that impact of a single removal can range from individual items to entire cutting patterns. The closer a structure node is to the root of the tree, the more significant an impact its removal will have.

There are two main motivations for allowing structure nodes to be removed. First, removing these nodes produces large regions of contiguous leftover space. This, in turn, provides the recreation phase more flexibility, resulting in diverse neighboring solutions. The second main advantage lies in the fact that it is probable that some of the removed items will form good patterns again at another location. In the second example, for instance, items 2 and 3 are likely to form a compact pattern again because they share the same width. This is particularly apparent in the later stages of the search, when the solution is already quite good in terms of quality. It could be viewed as a form of related removal.

However, dramatically destroying the solution can hinder the chances of finding better solutions. Radically ruining a solution too often can result in a lot of wasted time. Because the most impactful nodes are closest to the root of the tree, they are less numerous compared to the less impactful nodes at the bottom of the tree. Due to the fact that the ruin method selects nodes at random, removals with a significant impact have a lower chance of occurring. This could be viewed as a sort of built-in balancing mechanism.

Algorithm 1 Ruin phase

Input: A solution $S = \{C, E\}$ where C is a set of cutting patterns and E is a set of excluded items, bin area limit $\mathcal{A}_{lim}$, and the average number of nodes to be removed μ

Output: A modified (ruined) solution S'

1: **function** RUIN($S, \mathcal{A}_{lim}, \mu$)
2: $S' = \{C', E'\} \leftarrow S$
3: $i \in_R \{1, ..., 2\mu - 1\}$
4: **while** $i > 0$ **or** $\sum_{c \in C'} \mathcal{A}_c \geq \mathcal{A}_{lim}$ **do**
5: $c \in_R C'$
6: $N_c \leftarrow$ item and structure nodes of c
7: $n \in_R N_c$
8: $c' \leftarrow$ REMOVE(c, n)
9: $C' \leftarrow C' \setminus \{c\}$
10: **if** c' is not empty **then**
11: $C' \leftarrow C' \cup \{c'\}$
12: $E_n \leftarrow$ items in n
13: $E' \leftarrow E' \cup E_n$
14: $i \leftarrow i - 1$
15: **return** S'

Algorithm 1 provides the pseudocode for the ruin phase. The number of nodes to be removed, i, is uniformly randomized ($\in_R$) at each iteration and averages out at μ (line 3). The procedure continues ruining until i nodes have been removed or while the total bin

area of the modified solution exceeds the bin area limit $\mathcal{A}_{lim}$ (line 4). For each removal, a cutting pattern c is selected uniformly at random (line 5). A random item or structure node n is selected from c (lines 6-7). This node is removed from c, resulting in c' (lines 8-9). Finally, both the set of cutting patterns and the set of excluded items associated with the modified solution S' are updated (lines 10-13).

4.3. Recreate

During the recreate phase, the (ruined) solution needs to be rebuilt. This is done by inserting items into either leftover nodes or new bins. Of course, due to the bin area limit, it is highly unlikely that it will be possible to insert all items.

Algorithm 2 Recreate phase

Input: A solution $S = \{C, E\}$ where C is a set of cutting patterns and E is a set of excluded items, bin area limit $\mathcal{A}_{lim}$
Output: A modified (recreated) solution S'

1: **function** RECREATE($S, \mathcal{A}_{lim}$)
2: $S' = \{C', E'\} \leftarrow S$
3: $\tilde{E} \leftarrow E'$
4: **while** $\tilde{E} \neq \emptyset$ **do**
5: $e_{mr} \leftarrow$ most restricted item $\in \tilde{E}$
6: $O \leftarrow$ insertion options for e_{mr} for all cutting patterns $\in C'$
7: **if** $O = \emptyset$ **then** //attempt to open new bin
8: $\Delta \leftarrow \mathcal{A}_{lim} - \sum_{c \in C'} \mathcal{A}_c$
9: **if** there is a bin with $\mathcal{A} < \Delta$ **then**
10: $c \leftarrow$ empty cutting pattern using random bin with $\mathcal{A}_c < \Delta$
11: $C' \leftarrow C' \cup \{c\}$
12: $O \leftarrow O \cup \{$insertion options for e_{mr} in $c\}$
13: **if** $O \neq \emptyset$ **then** //insert the item
14: $o \leftarrow$ select best (with blinks) $\in O$
15: $c \leftarrow$ cutting pattern of o
16: $c' \leftarrow$ INSERT(c, o)
17: $C' \leftarrow C' \setminus \{c\} \cup \{c'\}$
18: $E' \leftarrow E' \setminus \{e_{mr}\}$
19: $\tilde{E} \leftarrow \tilde{E} \setminus \{e_{mr}\}$
20: **return** S'

Algorithm 2 provides the pseudocode for the recreate phase. During this phase, all currently excluded items have the opportunity to be inserted into the solution (lines 3-4). Set $\tilde{E}$ corresponds to all items which have yet to receive a chance at insertion. At each iteration, the most restricted item is selected as the next insertion candidate (line 5). The restrictiveness of an item is determined by its number of possible insertion places: the more

possibilities, the less restricted[1]. When multiple items are deemed equally restricted, a random item is selected. The definition of insertion options is provided in Section 4.3.1 in addition to how their costs are calculated. For the selected item, e_{mr}, all insertion options are generated (line 6). If there are no available options (line 7), an attempt is made to open a new bin (lines 8-12). Only bins which will not result in exceeding the bin area limit $\mathcal{A}_{lim}$ are eligible (lines 8-9). If possible, a new and empty cutting pattern is created with a random eligible bin (lines 10-11) and the set of insertion options is updated (line 12). If the item can be inserted (line 13), all options are evaluated (line 14) and eventually the item is inserted into the solution (lines 15-18). Most of the time, the cheapest option is selected in a greedy fashion, but sometimes the best option are blinked over. The concept of blinks is explained in Section 4.3.2. Regardless of whether a successful insertion takes place, item e_{mr} is removed from $\tilde{E}$ as it has now received a chance at insertion (line 19). Once every item has been given a chance to be inserted, a recreated solution S' is returned (line 20).

4.3.1. Insertion Options

An insertion option defines how exactly an item is inserted into a leftover node. If 90° rotation is allowed, there are two possible orientations for each item. Due to the guillotine constraint, each feasible orientation of an item in a leftover node, in turn, generates two insertion options depending on the direction of the first cut (vertical or horizontal). As a result, there are at most four different ways of inserting an item into a leftover node. An illustrative example of these four possibilities is shown in Figure 6. For each excluded item, all leftover nodes capable of fitting the item are considered.

All insertion options have an associated cost. This cost is used as a measure of an option's desirability and is calculated using Equation 1. Inserting items not only consumes existing leftover nodes, but also generates new ones. The cost of an insertion option o corresponds to the difference in total leftover value before and after the insertion. Each leftover node thus has a corresponding value (Equation 2) and the aim is to minimize the loss in value resulting from insertion.

n_o^u : leftover node used by insertion option o
N_o^c : set of leftover nodes created by insertion option o
α : leftover area value exponent

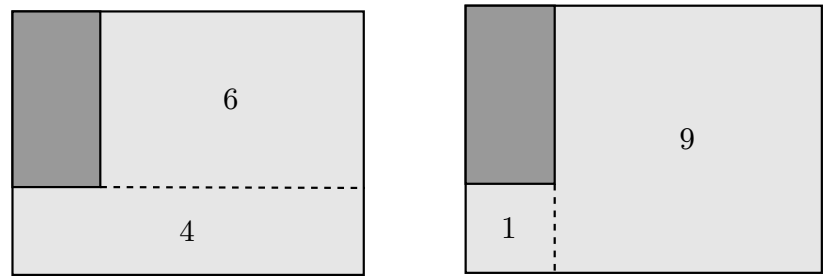
$$cost(o) = value(n_o^u) - \sum_{n \in N_o^c} value(n) \tag{1}$$

$$value(n) = (width_n \cdot height_n)^\alpha \tag{2}$$

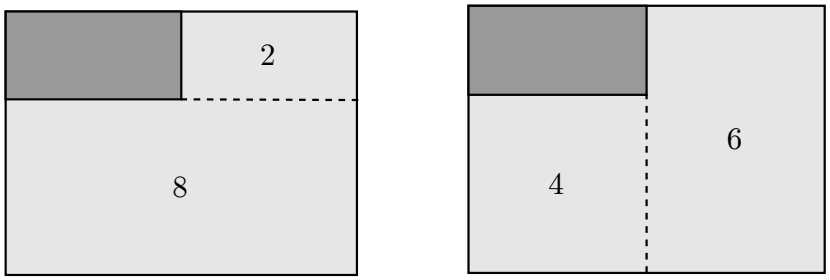
As shown in Equation 2, the value of a leftover node increases polynomially with respect to area. This is to encourage the heuristic to preserve large contiguous leftover areas instead of many smaller ones. Generally speaking, it will be easier to pack items into few, but larger, leftover nodes rather than many small ones. In Figure 6, option (b)

[1]Determining the most restricted item can be performed in linear time ($O(n)$) by caching the insertion options for all excluded items in a table and performing updates throughout the recreate phase.

has the lowest cost since it preserves the largest leftover value.



(a) $cost = 12^2 - (6^2 + 4^2) = 92$ (b) $cost = 12^2 - (9^2 + 1^2) = 62$



(c) $cost = 12^2 - (8^2 + 2^2) = 74$ (d) $cost = 12^2 - (6^2 + 4^2) = 92$

Figure 6: Four options of inserting an item into a leftover node ($width_{n_o^u} = 4$, $height_{n_o^u} = 3$, $\alpha = 2$, 2BP|R|G).

The value of exponent α has an impact on the behavior of the algorithm because it defines the ordering of insertion options. Making this value too large can lead to the creation of very few well-sized leftover areas. It will ignore many medium-sized leftover nodes and only care about a few large ones. On the other hand, setting the value of α too small can be detrimental for preserving large leftover areas.

4.3.2. Blinks

Always selecting the best option would result in a deterministic recreate phase and may lead to making the same bad decisions over and over again. To combat this, it is sometimes necessary to turn a blind eye to an insertion option. The concept of blinks was first introduced by Christiaens and Vanden Berghe [6] in a state-of-the-art heuristic for solving Vehicle Routing Problems. The general idea, however, is also applicable here.

In a pure best-fit, the best option is always selected. But with blinks, there is a small chance for an option to be ignored. Blinking rate β defines the likeliness of skipping over an option. Equation 3 shows the chance of an option being selected based on its rank r. A value for β of 0.05, for example, means each option has a 5% chance of being ignored.

$$p(r) = (1 - \beta)\beta^{(r-1)} \qquad r \in \{1, ..., \infty\} \tag{3}$$

Blinking exhibits the same behavior as Heuristic-Biased Stochastic Sampling, introduced by Bresina [3], with an exponential bias function. However, as Christiaens and Vanden Berghe [6] note, blinking is more efficient since there is no need to sort options based

on their fitness.

4.4. Evaluation

After each ruin and recreate iteration, the solution needs to be evaluated. This is accomplished by determining the quality of the solution in question, as explained in Section 4.4.1. Based on this quality, the solution is either accepted or rejected using the Late-Acceptance Hill-Climbing metaheuristic, which is described in Section 4.4.2.

4.4.1. Solution Quality

Due to bin area limit constraints, the recreate phase will most likely be unable to incorporate all items into the solution. The goal, therefore, is to guide the heuristic towards a feasible solution. Equation 4 describes how two solutions are compared against each other.

$$
\begin{aligned}
&E_S && : \text{set of excluded items in solution } S \\
&N_S^l && : \text{set of leftover nodes from cutting patterns in } C \text{ in solution } S \\
&a_e(S) && : \sum_{e \in E_S} width_e \cdot height_e \\
&v_l(S) && : \sum_{n \in N_S^l} value(n)
\end{aligned}
$$

$$
compare(S_1, S_2) = \begin{cases}
\text{-1} & \text{if } a_e(S_1) < a_e(S_2) & \texttt{//S1 superior} \\
1 & \text{if } a_e(S_1) > a_e(S_2) & \texttt{//S2 superior} \\
\text{-1} & \text{if } a_e(S_1) = a_e(S_2) \text{ and } v_l(S_1) > v_l(S_2) & \texttt{//S1 superior} \\
1 & \text{if } a_e(S_1) = a_e(S_2) \text{ and } v_l(S_1) < v_l(S_2) & \texttt{//S2 superior} \\
0 & \text{otherwise} & \texttt{//equal}
\end{cases} \tag{4}
$$

A solution is always superior if it excludes less item area. If two solutions include the same item area, the one with the highest total leftover value is superior. Therefore, leftover value can be viewed as a tiebreaker in the evaluation of solutions. Note that, despite the fact that the objective is to minimize the total bin area, this factor is not explicitly included in the cost function. Total bin area is irrelevant so long as it remains below the limit. As mentioned in Section 4.1, the limit is lowered every time a new feasible solution is reached.

4.4.2. Late-Acceptance Hill-Climbing

GDRR employs the Late Acceptance Hill-Climbing (LAHC) metaheuristic, introduced by Burke and Bykov [4]. This local search algorithm accepts non-improving moves when a candidate solution is better than the best solution a certain number of iterations ago. The pseudocode in Algorithm 3 shows how LAHC is implemented.

The main argument for using LAHC is its scale independence. Given that LAHC is insensitive to the scale of the objective function, the metaheuristic does not need to be tuned on an instance-specific basis. This is a major advantage over metaheuristics that make use of cooling schemes, such as Simulated Annealing.

Furthermore, since solutions are accepted based on their superiority with respect to previous solutions, no absolute difference in cost is needed. The only requirement is being

able to compare two costs. No delta value is necessary. This allows for the easy use of tiebreakers, such as the one outlined in Section 4.4.1, where the amount of included item area always dominates leftover value.

The final argument for using LAHC is that there is only a single parameter to tune: the history length ($\mathcal{L}_h$). This parameter defines how far back to look when evaluating a solution.

Algorithm 3 GDRR

Input: A starting solution $S = \{C, E\}$ where C is a set of cutting patterns and E is a set of excluded items, bin area limit $\mathcal{A}_{lim}$, history length $\mathcal{L}_h$, the average number of nodes to remove μ, and the current best solution S_{best}
Output: The best (complete) solution S_{best}

1: **function** GDRR($S, \mathcal{A}_{lim}, \mathcal{L}_h, \mu, S_{best}$)
2: **for all** $k \in \{0, ..., \mathcal{L}_h - 1\}$ **do** $S_k \leftarrow S$
3: $i \leftarrow 0$
4: $S^* = \{C^*, E^*\} \leftarrow S$ //local optimum (incomplete)
5: **while** no stopping criteria met **do**
6: $S' \leftarrow$ RUIN($S^*, \mathcal{A}_{lim}, \mu$)
7: $S' \leftarrow$ RECREATE($S', \mathcal{A}_{lim}$)
8: $v \leftarrow i \bmod \mathcal{L}_h$
9: **if** $compare(S', S_v) \leq 0$ **or** $compare(S', S^*) \leq 0$ **then** //solution accepted
10: $S^* \leftarrow S'$
11: **if** $compare(S^*, S_v) < 0$ **then** //fitness array updated
12: $S_v \leftarrow S^*$
13: $i \leftarrow i + 1$
14: **if** $E^* = \emptyset$ **then** //complete solution reached
15: $S_{best} \leftarrow S^*$
16: $\mathcal{A}'_{lim} \leftarrow \sum_{c \in C^*} \mathcal{A}_c$
17: $S_{next} \leftarrow$ RUIN($S_{best}, \mathcal{A}'_{lim}, 0$)
18: $S_{best} \leftarrow$ GDRR($S_{next}, \mathcal{A}'_{lim}, \mathcal{L}_h, \mu, S_{best}$)
19: **return** S_{best}

Algorithm 3 provides a high-level overview of GDRR. As mentioned in Section 4.1, the heuristic will continuously attempt to fit all items inside the bins while respecting the bin area limit constraint. This limit is lowered each time the heuristic is able to construct a feasible solution.

The LAHC fitness array is initially filled entirely with the starting solution (line 2). This array keeps track of the last $\mathcal{L}_h$ best solutions. During each local search iteration, the solution is partially destroyed and rebuilt (lines 6-7) according to the procedures described in Sections 4.2 and 4.3. Index v corresponds to the index of the first element in the fitness array. Instead of shifting all elements of the fitness array every time a new solution is

accepted, the head of the array is moved (line 8). This is similar to how a circular buffer works. In order for a modified solution (S') to be accepted, it must either be superior or equal to the best solution $\mathcal{L}_h$ iterations ago (S_v) or improve upon the local optimum (S^*) (lines 9-10). The fitness array is only updated with improving values (line 11-12). The implementation of LAHC is analogous to that outlined in Burke and Bykov [4], but with one major exception. Counter i is only incremented when a solution is accepted (line 13). This is in contrast to Burke and Bykov's original LAHC, where the counter is always incremented. The main motivation behind this decision is to avoid the heuristic from becoming a pure hill-climbing algorithm in later stages and thus easily getting stuck in local optima.

If a solution is obtained that has no excluded items (line 14), GDRR has reached its current goal. When this occurs, the best complete solution S_{best} is updated (line 15), a new bin area limit $\mathcal{A}'_{lim}$ is calculated based on the total bin area of S_{best} (line 16) and a starting solution for the next goal S_{next} is created which complies with this new limit (line 17). Finally, GDRR is 'restarted' for the next goal (line 18). Once a stopping criterion is met, the best complete solution is returned (line 19).

In essence, the heuristic continuously strives towards feasibility with an iteratively lowering bin area limit. To initiate the optimization, Algorithm 3 can be called with an empty S (no cutting patterns and all items excluded) and with $\mathcal{A}_{lim}$ set to ∞. There is no need for a separate algorithm to create an initial solution.

4.5. Multithreading

Multithreading can help to improve the performance of the heuristic and makes use of the capabilities of modern processors. Each thread runs its own separate instance of GDRR, but a single global bin area limit is shared across all threads. This means that once one thread reaches a feasible solution, the limit is lowered across all threads. Solutions themselves are, by contrast, not shared between threads. This is mainly to preserve diversity.

The multithreaded implementation is largely analogous to Algorithm 3. The only major difference is that the bin area limit is not only lowered when the heuristic reaches a feasible solution (lines 12-14), but rather whenever any thread reaches a feasible solution at the current limit. It goes without saying that the best feasible solution among all threads is saved.

Threads are almost entirely independent and only influence each other indirectly. There are, for example, no explicit mechanisms in place to avoid duplicate searched areas. Due to the vast search space and the heuristic containing a number of stochastic elements (e.g. blinks), there is no real need for such mechanisms.

Multithreading is primarily used to maintain diversity, not necessarily to accelerate execution speed. The use of multithreading is particularly beneficial for instances with many different bins. At later stages of the optimization, solutions are often too fixed to significantly change the combination of used bins. Multithreading helps overcome this issue since different threads can have solutions that are close in terms of quality, but consist of very different combinations of bins. This level of diversity would be difficult to achieve with only a single thread, as it would require very extreme ruin procedures.

5. Computational Results

To evaluate the performance of GDRR, the algorithm was tested on a number of different benchmarks from the literature. These benchmarks contain datasets with both identical and heterogeneous sets of bins. Oliveira et al. [26] provides a central repository for a large number of cutting and packing datasets from the literature. All datasets included in the repository are formatted in the same structure, which saves a lot of time when testing across a number of different benchmarks.

Section 5.1 defines how the parameters of GDRR were configured for the computational experiments. The performance of the algorithm is then evaluated in Sections 5.2-5.5. Finally, Sections 5.6 and 5.7 analyze the impact of multithreading and calculation time on the performance of GDRR. All experiments were conducted on an Intel Xeon Gold 6240 CPU (2.6GHz) with 8GB of RAM. For the comparisons in Sections 5.2-5.5, GDRR was configured to use 8 threads and 600 seconds of runtime. Solutions for all datasets tested in the comparison are included as an online supplement of this paper.

5.1. Parameters

GDRR has a number of parameters which must be configured. The tuning of these parameters was conducted using the dataset from Ortmann et al. [27]. This dataset was chosen because it contains a large variety of instances and has variable-sized bins. Heterogeneous bins enable gradual improvements in solution quality to be visible. All parameters are tuned to achieve best results when optimizing for 600 seconds and 8 threads. Parameters are fixed and varied one by one to examine their individual effect on the behavior of the algorithm. However, the history length ($\mathcal{L}_h$) and the average number of removed nodes (μ) were tuned as a pair since they are so closely related. The parameters listed below are used throughout the experiments:

- **Leftover valuation power:** α
  In Section 4.3.1, parameter α was defined as the power with which to multiply the area of a leftover node in order to derive its value. The leftover value is not only used to evaluate insertion options, but also as a tiebreaker when determining a solution's quality (Section 4.4.1). A greater α will favor few, but large, leftover areas. When α is smaller, many medium-sized leftover pieces are not penalized as much. Experiments show that the value of α has a minimal impact so long as it is greater than 1. A value of 1.2 resulted in the best overall performance.
  - $\alpha = 1.2$
- **History Length:** $\mathcal{L}_h$
  $\mathcal{L}_h$ determines the length of the history queue in LAHC. The best value for $\mathcal{L}_h$ depends largely on the size of the instance. The larger an instance, the shorter the optimal history length. This is because larger instances result in less ruin and recreate iterations per second and generally have more diverse neighborhoods. Preliminary experiments resulted in the following rules of thumb:
  - $0 < \#items \leq 100$: $\mathcal{L}_h = 2000$

- $100 < \#items \leq 300$: $\mathcal{L}_h = 1000$
- $300 < \#items \leq 500$: $\mathcal{L}_h = 500$

- **Average number of removed nodes:** μ
  As with $\mathcal{L}_h$, the optimal value of μ depends on the instance size. Ruining the solution less significantly results in more iterations per second, which is beneficial for larger instances. Preliminary experiments showed that the following values for μ are a suitable choice:

  - $0 < \#items \leq 100$: $\mu = 8$
  - $100 < \#items \leq 300$: $\mu = 6$
  - $300 < \#items \leq 500$: $\mu = 4$

- **Blink rate :** β
  Across the dataset, a blinking chance of 5% was deemed to be a good balance between greediness and randomness.

  - $\beta = 0.05$

5.2. Results for 2BP|O|G

To compare the performance of GDRR for the 2BP with guillotine cuts and no rotation of items, the well-known instances from Berkey and Wang [1] (classes 1-6) and Lodi et al. [21] (classes 7-10) are used. Each class has ten instances of 20, 40, 60, 80 and 100 items. As a result, the dataset contains a total of 500 instances.

A comparison between GDRR and other heuristics in the literature is presented in Table 1. The sum of bins for each instance category is displayed. As Cui et al. [11] note, the majority of instances from this dataset have already been solved to optimality. Therefore, the differences between algorithms might seem small. The comparison includes a partial enumeration heuristic (ENH) by Lodi et al. [23], a constructive heuristic (CHBP) by Charalambous and Fleszar [5], a hybrid heuristic algorithm (HHA) by Hong et al. [16], the critical fit insertion heuristic (CFIH) by Fleszar [13] and the hybrid heuristic with triple block patterns (HHTB) by Cui et al. [11]. For all of these heuristics, the best results reported in each paper are presented in the comparison.

The proposed heuristic achieves the best scores in each category. However, it should be noted that, with the exception of ENH[23], GDRR has longer runtimes than most of the other algorithms. Since many of these algorithms are based heavily on constructive approaches their runtimes are around or under the 1-minute mark.

Class	Number of items	GDRR	ENH [23]	CHBP [5]	HHA [16]	CFIH [13]	HHTB [11]
1	20	**71**	**71**	**71**	**71**	**71**	**71**
	40	**134**	**134**	**134**	**134**	135	**134**
	60	**200**	**200**	201	201	201	**200**
	80	**275**	**275**	**275**	**275**	**275**	**275**
	100	**317**	**317**	321	320	322	**317**
	All	**997**	**997**	1002	1001	1004	**997**
2	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**19**	20	20	**19**	20	20
	60	**25**	**25**	26	**25**	27	**25**
	80	**31**	32	33	**31**	32	**31**
	100	**39**	**39**	**39**	**39**	40	**39**
	All	**124**	126	128	**124**	129	125
3	20	**52**	54	**52**	**52**	53	**52**
	40	**94**	96	97	**94**	96	**94**
	60	**139**	140	140	140	141	**139**
	80	**189**	190	196	192	195	**189**
	100	**223**	225	230	228	226	**223**
	All	**697**	705	715	706	711	**697**
4	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**19**	**19**	**19**	**19**	**19**	**19**
	60	**24**	25	25	25	25	25
	80	**31**	33	33	32	33	**31**
	100	**37**	39	39	38	39	**37**
	All	**121**	126	126	124	126	122
5	20	**65**	66	**65**	**65**	66	**65**
	40	**119**	**119**	121	**119**	120	**119**
	60	**180**	181	183	181	182	**180**
	80	**247**	**247**	**247**	**247**	248	**247**
	100	**282**	286	288	287	290	**282**
	All	**893**	899	904	899	906	**893**
6	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**17**	19	19	18	19	**17**
	60	**21**	22	22	22	22	**21**
	80	**30**	**30**	**30**	**30**	**30**	**30**
	100	**32**	35	34	35	34	**32**
	All	**110**	116	115	115	115	**110**
7	20	**55**	**55**	**55**	**55**	56	**55**
	40	**111**	113	112	**111**	115	**111**
	60	**158**	159	160	159	161	**158**
	80	**230**	232	233	232	232	231
	100	**271**	275	275	273	274	**271**
	All	**825**	834	835	830	838	826
8	20	**58**	**58**	**58**	**58**	60	**58**
	40	**113**	**113**	114	**113**	116	**113**
	60	**161**	162	163	162	165	**161**
	80	**224**	226	226	225	227	**224**
	100	**277**	280	279	279	281	**277**
	All	**833**	839	840	837	849	**833**
9	20	**143**	**143**	**143**	**143**	**143**	**143**
	40	**278**	**278**	279	**278**	**278**	**278**
	60	**437**	**437**	438	**437**	**437**	**437**
	80	**577**	**577**	**577**	**577**	**577**	**577**
	100	**695**	**695**	**695**	**695**	**695**	**695**
	All	**2130**	**2130**	2132	**2130**	**2130**	**2130**
10	20	**42**	44	44	43	43	43
	40	**74**	**74**	**74**	**74**	75	**74**
	60	**101**	102	103	102	104	**101**
	80	**128**	130	130	130	132	**128**
	100	**158**	159	163	160	163	159
	All	**503**	509	514	509	517	505
Total bins:		**7233**	7281	7311	7275	7325	7238

Table 1: Results for the 2DBP|O|G benchmark from Berkey and Wang [1] and Lodi et al. [21]

Class	Number of items	GDRR	CHBP [5]	A-B [29]	SVCRG [10]	CFIH [13]	HHTB [11]
1	20	**66**	**66**	**66**	**66**	**66**	**66**
	40	**128**	129	129	**128**	129	**128**
	60	**195**	**195**	**195**	**195**	**195**	**195**
	80	**270**	271	**270**	**270**	271	**270**
	100	**313**	314	314	314	314	**313**
	All	**972**	975	974	973	975	**972**
2	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**19**	**19**	**19**	20	**19**	20
	60	**25**	**25**	**25**	29	**25**	**25**
	80	**31**	**31**	**31**	33	32	**31**
	100	**39**	**39**	**39**	40	**39**	**39**
	All	**124**	**124**	**124**	132	125	125
3	20	**47**	48	49	**47**	48	**47**
	40	**92**	94	94	**92**	94	**92**
	60	**134**	136	137	135	136	**134**
	80	**183**	186	188	184	185	**183**
	100	**219**	223	223	223	223	**219**
	All	**675**	687	691	681	686	**675**
4	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**19**	**19**	**19**	**19**	**19**	**19**
	60	**23**	25	25	25	25	25
	80	**30**	33	31	31	33	31
	100	**37**	38	**37**	38	38	**37**
	All	**119**	125	122	123	125	122
5	20	**59**	**59**	60	**59**	**59**	**59**
	40	**114**	115	115	**114**	117	**114**
	60	**173**	176	176	174	175	**173**
	80	239	240	243	240	241	**238**
	100	**277**	282	284	281	281	**277**
	All	862	872	878	868	873	**861**
6	20	**10**	**10**	**10**	**10**	**10**	**10**
	40	**16**	18	18	19	18	18
	60	**21**	**21**	**21**	**21**	**21**	**21**
	80	**30**	**30**	**30**	**30**	**30**	**30**
	100	**32**	34	34	34	34	**32**
	All	**109**	113	113	114	113	111
7	20	**52**	**52**	**52**	**52**	**52**	**52**
	40	102	104	103	103	106	**101**
	60	**145**	148	148	**145**	151	**145**
	80	208	211	211	210	214	**207**
	100	249	255	252	251	258	**248**
	All	756	770	766	761	781	**753**
8	20	**53**	**53**	**53**	**53**	**53**	**53**
	40	104	105	104	104	105	**103**
	60	**146**	150	150	147	152	**146**
	80	204	210	209	207	211	**203**
	100	**252**	258	256	254	258	**252**
	All	759	776	772	765	779	**757**
9	20	**143**	**143**	**143**	**143**	**143**	**143**
	40	**275**	**275**	**275**	**275**	**275**	**275**
	60	**435**	**435**	**435**	**435**	**435**	**435**
	80	**573**	**573**	**573**	**573**	**573**	**573**
	100	**693**	**693**	**693**	**693**	**693**	**693**
	All	**2119**	**2119**	**2119**	**2119**	**2119**	**2119**
10	20	**41**	**41**	**41**	**41**	42	**41**
	40	**72**	73	73	73	73	73
	60	**99**	100	100	100	101	**99**
	80	**124**	130	129	126	129	126
	100	**155**	159	161	159	159	**155**
	All	**491**	503	504	499	504	494
Total bins:		**6986**	7064	7063	7035	7080	6989

Table 2: Results for the 2DBP|R|G benchmark from Berkey and Wang [1] and Lodi et al. [21]

5.3. Results for 2BP|R|G

To compare the performance of the heuristic for the 2BP with guillotine cuts and 90° rotation of items, once again the dataset by Berkey and Wang [1] and Lodi et al. [21] is used. In addition to some of the heuristics from Section 5.2, this comparison also includes an agent-based approach (A-B) by Polyakovsky and M'Hallah [29] and the sequential value correcting heuristic (SVCRG) by Cui et al. [10].

Unlike in Section 5.2, GDRR does not always come out on top here. On average the proposed heuristic achieves the best quality solutions, but there are some classes where HHTB is superior.

5.4. Results for 2VSBP|O|G

The performance on instances with a heterogeneous set of bins is tested using three well-known benchmarks. In this comparison, the average utilization is compared across different algorithms. Utilization γ corresponds to the percentage of area of the bins that are packed with items. A utilization of 95%, for example, corresponds to 5% wasted bin area.

$$\gamma = \frac{\text{total item area}}{\text{total bin area}} \cdot 100\%$$

All the results from other heuristics in the tables are taken from the comparison performed by Hong et al. [16]. Table 3 provides a comparison for the instances created by Hopper and Turton [17] which consist of three problem categories, each containing five instances. The second dataset was introduced by Pisinger and Sigurd [28] and is divided into ten classes, the results for which are summarized in Table 4. Note that this dataset does not define limits on the quantity of bins of each type. The last dataset, introduced by Ortmann et al. [27], contains instances based on nice (relatively normal) and pathologically (unusual, more extreme) sized items. These instances range from 25-500 items and 2-6 different bin sizes. Table 5 provides the results compared to other heuristics. For all of these datasets and classes, GDRR always achieves the best solutions in terms of quality.

Class	GDRR	FFDH[8]	BFDH[9]	JOIN[24]	FCOG[21]	BFDH*[2]	SAS[25]	SASm[27]	BFS[27]	HHA[16]
M1	**98.4**	93.5	93.5	86.8	94.9	94.9	83.5	91.6	95.5	**98.4**
M2	**97.2**	87.5	88.8	82.8	89.6	89.6	81.7	88.0	90.0	95.6
M3	**98.0**	92.0	92.6	85.5	93.6	93.6	86.8	91.8	94.9	97.4
Average utilization [%]:	**97.85**	91.00	91.63	85.03	92.70	92.70	84.00	90.47	93.47	97.13

Table 3: Results for the 2VSBP|O|G benchmark from Hopper and Turton [17]

5.5. Results for 2VSBP|R|G

To the best of our knowledge, there are currently no algorithms available in the literature with which results for the 2BP variant with 90° rotation and heterogeneous sets of bins can be compared. Since GDRR supports this variant, experiments were run on the same three datasets in Section 5.4. The solutions are included in the online supplement to facilitate comparisons in future research.

Class	GDRR	FFDH[8]	BFDH[9]	JOIN[24]	FCOG[21]	BFDH*[2]	SAS[25]	SASm[27]	BFS[27]	HHA[16]
I	**94.1**	86.3	86.6	83.0	87.3	87.4	79.4	86.3	88.6	91.5
II	**96.5**	83.7	83.7	80.9	85.2	85.0	80.1	82.2	85.1	96.3
III	**91.5**	81.1	81.7	76.7	81.9	81.9	69.6	75.7	82.2	86.6
IV	**93.5**	80.0	80.0	79.1	82.7	82.1	76.0	78.0	82.0	91.7
V	**89.3**	80.4	81.1	77.6	81.3	81.2	72.6	78.3	81.4	84.6
VI	**92.7**	79.1	79.3	78.3	80.7	80.5	76.1	77.1	79.5	90.2
VII	**90.1**	79.9	80.2	79.6	80.8	80.6	74.0	80.4	80.9	86.9
VIII	**89.4**	80.7	81.1	74.2	81.3	81.2	76.4	79.5	81.6	85.9
IX	**75.6**	72.8	72.6	72.1	72.6	72.8	71.6	72.9	73.0	74.3
X	**93.0**	83.3	83.8	79.3	85.4	84.6	73.2	79.4	85.5	90.3
Average utilization [%]:	**90.66**	80.73	81.01	78.08	81.92	81.73	74.90	78.98	81.98	87.83

Table 4: Results for the 2VSBP|O|G benchmark from Pisinger and Sigurd [28]

Class	GDRR	FFDH[8]	BFDH[9]	JOIN[24]	FCOG[21]	BFDH*[2]	SAS[25]	SASm[27]	BFS[27]	HHA[16]
Nice25i	**99.8**	73.9	73.6	70.6	73.9	73.6	68.3	71.8	73.6	94.3
Nice50i	**93.5**	76.7	76.7	73.3	77.9	77.7	70.8	73.1	77.8	89.2
Nice100i	**91.8**	79.4	79.4	77.5	79.9	79.4	75.7	76.3	79.4	88.3
Nice200i	**92.6**	82.0	82.0	81.5	84.5	84.5	78.5	79.4	84.5	91.0
Nice300i	**93.3**	85.8	85.8	83.3	86.0	86.5	80.2	81.7	86.8	92.0
Nice400i	**94.3**	85.1	85.1	82.7	86.6	85.7	80.2	81.0	85.7	92.8
Nice500i	**94.1**	87.2	87.2	84.8	87.7	87.7	81.8	83.8	87.7	93.1
Nice	**94.04**	81.44	81.40	79.10	82.35	82.15	76.50	78.15	82.21	91.52
Path25i	**100**	76.3	76.3	73.9	77.9	77.6	72.2	74.4	78.3	97.2
Path50i	**98.1**	76.4	78.6	74.2	81.6	79.4	72.4	75.6	81.9	94.9
Path100i	**95.5**	79.7	79.7	77.8	83.2	81.3	72.4	75.8	83.5	93.5
Path200i	**94.6**	84.1	84.0	81.8	88.0	85.9	77.7	79.0	87.5	93.3
Path300i	**95.5**	82.9	82.9	82.7	87.0	86.0	81.4	81.7	87.3	94.9
Path400i	**95.4**	82.7	82.7	82.3	89.6	87.0	79.9	80.1	88.5	94.8
Path500i	**96.6**	82.9	82.9	81.2	88.7	86.4	81.3	82.4	86.7	94.0
Path	**96.43**	80.71	81.01	79.13	85.14	83.37	76.76	78.43	84.81	94.66
Average utilization [%]:	**95.24**	81.22	81.35	79.26	83.89	82.91	76.78	78.43	83.66	93.10

Table 5: Results for the 2VSBP|O|G benchmark from Ortmann et al. [27]

5.6. Impact of multiple threads

To test the impact of multithreading on the solution quality, experiments were run with a subset of instances from Ortmann et al. [27]. From each class, the first instance with 6 different bins was selected. This decision was made because a greater variety of bins allows for more gradual differences in solution quality to be visible. The results are shown in Figure 7. As expected, the first few extra threads result in a significant performance improvement. Increasing the number of threads from one to four, for example, results in an improvement of 0.5% in average utilization. However, adding additional threads quickly leads to diminishing returns.

5.7. Impact of calculation time

The impact of available calculation time on solution quality was tested using the same subset of instances detailed in Section 5.6. Figure 8 shows how the heuristic scales with respect to calculation time. Despite improvements slowing down, there are still gains to be had, even after 30 minutes. This is most likely because the history length ($\mathcal{L}_h$) was scaled with respect to the calculation time. When running GDRR for 5 minutes, for example, $\mathcal{L}_h$ was set to half of the value described in Section 5.1.

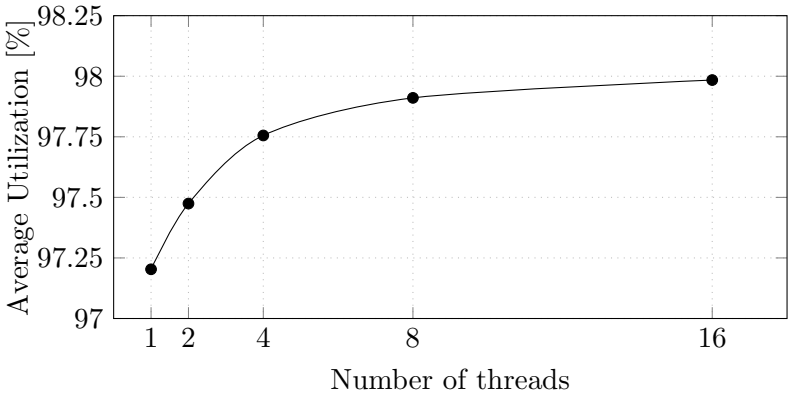


Figure 7: Impact of multithreading on solution quality (600s of runtime)

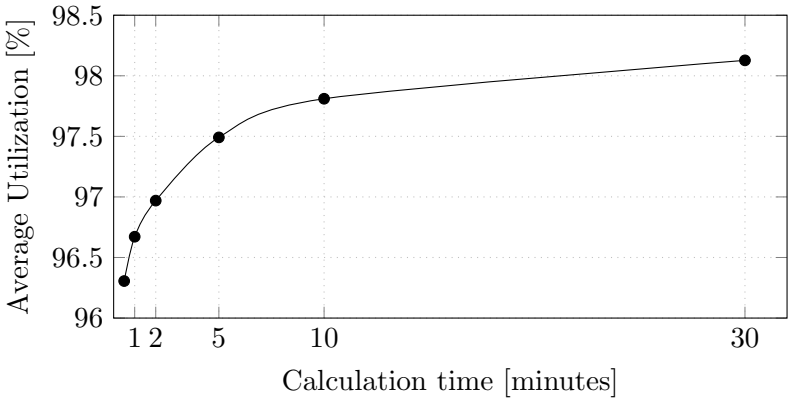


Figure 8: Impact of calculation time on solution quality (8 threads)

6. Conclusion

This paper introduced GDRR, a heuristic for solving the 2D bin packing problem with guillotine constraints. Variants of the problem with a heterogeneous set of bins (variable-sized) and 90° rotation of items are also supported. The heuristic contains a ruin and recreate procedure which iteratively attempts to improve the solution. The search can be described as being goal-driven since it continuously strives to create feasible solutions with an ever decreasing limit of available bin area. To escape local optima, GDRR employs the Late Acceptance Hill-Climbing metaheuristic. Unlike most other heuristics for this problem, the focus lies primarily on the improvement phase rather than the constructive aspect. It is capable of consistently producing the best results in terms of solution quality across several benchmarks in the literature.

Future research may include exploring the 2BP variant without guillotine constraint and other cutting and packing problems with real-world constraints. Finally, it would also be interesting to investigate the applicability of a goal-driven approach in other types of problems.

Acknowledgment

Editorial consultation provided by Luke Connolly (KU Leuven). The authors wish to thank the anonymous reviewers and the editor whose comments significantly improved the quality of the paper.

References

[1] Berkey, J. O. and Wang, P. Y. (1987). Two-dimensional finite bin-packing algorithms. *Journal of the operational research society*, 38(5):423–429.

[2] Bortfeldt, A. (2006). A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. *European Journal of Operational Research*, 172(3):814–837.

[3] Bresina, J. L. (1996). Heuristic-Biased Stochastic Sampling. *AAAI/IAAI*, 1:271–278.

[4] Burke, E. K. and Bykov, Y. (2017). The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78.

[5] Charalambous, C. and Fleszar, K. (2011). A constructive bin-oriented heuristic for the two-dimensional bin packing problem with guillotine cuts. *Computers and Operations Research*, 38(10):1443–1451.

[6] Christiaens, J. and Vanden Berghe, G. (2020). Slack induction by string removals for vehicle routing problems. *Transportation Science*, 54(2):417–433.

[7] Clautiaux, F., Jouglet, A., and Moukrim, A. (2013). A new graph-theoretical model for the guillotine-cutting problem. *INFORMS Journal on Computing*, 25(1):72–86.

[8] Coffman, Jr, E. G., Garey, M. R., Johnson, D. S., and Tarjan, R. E. (1980). Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826.

[9] Coffman Jr, E. and Shor, P. (1990). Average-case analysis of cutting and packing in two dimensions. *European Journal of Operational Research*, 44(2):134–144.

[10] Cui, Y.-P., Cui, Y., and Tang, T. (2015). Sequential heuristic for the two-dimensional bin-packing problem. *European Journal of Operational Research*, 240(1):43–53.

[11] Cui, Y.-P., Yao, Y., and Zhang, D. (2018). Applying triple-block patterns in solving the two-dimensional bin packing problem. *Journal of the operational research society*, 69(3):402–415.

[12] Dyckhoff, H. (1990). A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145–159.

[13] Fleszar, K. (2013). Three insertion heuristics and a justification improvement heuristic for two-dimensional bin packing with guillotine cuts. *Computers & Operations Research*, 40(1):463–474.

[14] Friesen, D. K. and Langston, M. A. (1986). Variable sized bin packing. *SIAM journal on computing*, 15(1):222–230.

[15] Gilmore, P. and Gomory, R. E. (1965). Multistage cutting stock problems of two and more dimensions. *Operations research*, 13(1):94–120.

[16] Hong, S., Zhang, D., Lau, H. C., Zeng, X., and Si, Y. W. (2014). A hybrid heuristic algorithm for the 2D variable-sized bin packing problem. *European Journal of Operational Research*, 238(1):95–103.

[17] Hopper, E. and Turton, B. (2002). Problem generators for rectangular packing problems. *Stud. Inform. Univ.*, 2(1):123–136.

[18] Iori, M., de Lima, V. L., Martello, S., Miyazawa, F. K., and Monaci, M. (2020). Exact solution techniques for two-dimensional cutting and packing. *arXiv preprint arXiv:2004.12619*.

[19] Kröger, B. (1995). Guillotineable bin packing: A genetic approach. *European Journal of Operational Research*, 84(3):645–661.

[20] Lodi, A., Martello, S., and Monaci, M. (2002a). Two-dimensional packing problems: A survey. *European journal of operational research*, 141(2):241–252.

[21] Lodi, A., Martello, S., and Vigo, D. (1999). Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS Journal on Computing*, 11(4):345–357.

[22] Lodi, A., Martello, S., and Vigo, D. (2002b). Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1-3):379–396.

[23] Lodi, A., Monaci, M., and Pietrobuoni, E. (2017). Partial enumeration algorithms for Two-Dimensional Bin Packing Problem with guillotine constraints. *Discrete Applied Mathematics*, 217:40–47.

[24] Martello, S., Monaci, M., and Vigo, D. (2003). An exact approach to the strip-packing problem. *INFORMS journal on Computing*, 15(3):310–319.

[25] Ntene, N. and van Vuuren, J. H. (2009). A survey and comparison of guillotine heuristics for the 2d oriented offline strip packing problem. *Discrete Optimization*, 6(2):174–188.

[26] Oliveira, O., Gamboa, D., and Silva, E. (2019). Resources for two-dimensional (and three-dimensional) cutting and packing solution methods research. *In Proceedings of the 16th International Conference on Applied Computing*, 53:131–138. `https://github.com/Oscar-Oliveira/OR-Datasets`.

[27] Ortmann, F. G., Ntene, N., and Van Vuuren, J. H. (2010). New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems. *European Journal of Operational Research*, 203(2):306–315.

[28] Pisinger, D. and Sigurd, M. (2005). The two-dimensional bin packing problem with variable bin sizes and costs. *Discrete Optimization*, 2(2):154–167.

[29] Polyakovsky, S. and M'Hallah, R. (2009). An agent-based approach to the two-dimensional guillotine bin packing problem. *European Journal of Operational Research*, 192(3):767–781.

[30] Wäscher, G., Haußner, H., and Schumann, H. (2007). An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130.

[31] Wei, L., Oon, W.-C., Zhu, W., and Lim, A. (2013). A goal-driven approach to the 2d bin packing and variable-sized bin packing problems. *European Journal of Operational Research*, 224(1):110–121.