International Journal on Artificial Intelligence Tools

A Strip Packing Solving Method Using an Incremental Move Based on Maximal Holes*

Bertrand Neveu†

INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis cedex, France
neveu@sophia.inria.fr

Gilles Trombettoni‡

INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis cedex, France
trombe@sophia.inria.fr

Ignacio Araya§

Department of Computer Science, Universitad Técnica Federico Santa Maria, Valparaiso, Chile
Ignacio.Araya@sophia.inria.fr

Maria-Cristina Riff¶

Department of Computer Science, Universitad Técnica Federico Santa Maria, Valparaiso, Chile
riff@utfm.cl

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

When handling 2D packing problems, numerous incomplete and complete algorithms maintain a so-called bottom-left (BL) property: no rectangle placed in a container can be moved more left or bottom. While it is easy to make a rectangle BL when it is added in a container, it is more expensive to maintain all the placed pieces BL when a rectangle is removed. This prevents researchers from designing incremental moves for metaheuristics or efficient complete optimization algorithms. This paper investigates the possibility of violating the BL property. Instead, we propose to maintain the set of *maximal holes*, which allows incremental additions and removals of rectangles.

To validate our alternative approach, we have designed an incremental move, maintaining maximal holes, for the strip packing problem, a variant of the famous 2D bin-packing. We have also implemented a metaheuristic, *with no user-defined parameter*, using this move and standard greedy heuristics. We have finally designed two variants of this incomplete method. In the first variant, a better first layout is provided by a hyperheuristic proposed by some of the authors. In the second variant, a fast repacking procedure recovering the BL property is occasionally called during the local search.

Experimental results show that the approach is competitive with the best known incomplete algorithms.

Keywords: local search, 2D packing, bin packing, strip packing

*With the financial support of CONICYT and INRIA.
†COPRIN team, INRIA Sophia-Antipolis, ENPC, France
‡COPRIN team, University of Nice-Sophia, INRIA, France
§Department of Computer Science, Universitad Técnica Federico Santa Maria, Valparaiso, Chile
¶Department of Computer Science, Universitad Técnica Federico Santa Maria, Valparaiso, Chile

1. Introduction

Packing problems consist in placing pieces in containers, such that the pieces do not intersect. Specific variants differ in the considered dimension (1D, 2D or 3D), in the type of pieces, or in additional constraints: for cutting applications, whether the (2D) container is guillotinable or not; whether the objects can rotate, and so on. The 2D strip packing problem studied in this paper finds the best way for placing rectangles of given heights and widths, without overlapping, into a strip of given width and infinite height. The goal is to minimize the required height. We also study the variant that allows the rotation of rectangles with an angle of 90 degrees.

Packing problems have numerous practical applications. Strip packing occurs for instance in the cutting of rolls of paper or metal. In 3D, solving packing problems helps transporting a volume of goods in containers. The most interesting packing problems are all NP-hard, leading to the design of complete combinatorial algorithms, incomplete greedy heuristics, metaheuristics or genetic algorithms. To limit the combinatorial explosion, most algorithms maintain the *Bottom-Left (BL) property, that is, a layout where the bottom and left segments of every rectangle touch the container or another rectangle*. First, the BL property lowers the number of possible locations for rectangles. Second, it can be proven that any solution of a 2D packing problem can be transformed into a solution respecting the BL property with a simple repacking procedure. However, when a rectangle is removed from the container, this repacking procedure is not local to the removed rectangle and to its neighbors, but modifies the whole layout in the worst case (e.g., when the removed rectangle is placed on the bottom-left corner of the container).

After a brief survey of existing algorithms in Section 2, we present in Section 3 how to add/remove one rectangle in/from a container. These operations are original in that they incrementally maintain a set of so-called *maximal holes* without necessarily recovering the BL property. These operations are generic and can be applied to any 2D packing problem. The second part of this paper experimentally shows that it is possible to design algorithms that, although they do not always respect the BL property, do not "fragment" the container, i.e., do not provide a bad layout with a lot of small holes between rectangles. Section 4 introduces a new and incremental move for 2D strip packing that maintains the set of maximal holes during the addition and removal of rectangles. This move leads to a new incomplete algorithm for strip packing with **no** user-defined parameter. Section 5 presents variants of this method that is improved in two ways. First, the metaheuristic starts with a better initial configuration obtained by a hyper-heuristic (proposed by some of the authors). Second, when the walk cannot improve the current solution, the metaheuristic launches a repacking procedure that recovers the BL property. The experiments presented in Section 6 show the interest of these new methods based on our move.

2. Existing algorithms for strip packing

A lot of researchers have proposed different algorithms to handle bin packing so that we focus on strip packing in this section. In the last few years, the interest in strip packing has increased, hence the proposal of new approaches and the improvement of existing strategies.

Exact approaches are in general limited to small instances (Ref. [16]). Although not competitive with incomplete approaches, the branch and bound algorithm proposed by Martello et al. (Ref. [18]) is interesting and can solve some instances of up to 200 rectangles. Their algorithm computes good bounds obtained by geometrical considerations and a relaxation of the problem.

E. Hopper's thesis (Ref. [13]) exhaustively describes existing incomplete algorithms for strip packing. We just provide an overview of these heuristics ranging from simple greedy (constructive) algorithms to complex metaheuristics or genetic algorithms.

Bottom Left Fill (BLF) (see Ref. [10]) is a generalization of the first greedy heuristic proposed by Baker et al. (Ref. [3]) in 1980. BLF handles the rectangles in a predefined order, e.g., by decreasing width, height or surface. A rectangle R is placed in the first location that can contain R. The locations (i.e., corners or holes) are sorted according to their ordinate in the strip as first criterion and according to their abcissa as second criterion, so that an added rectangle is positioned on the strip as far down and to the left as possible. Therefore, the built layout always respects the BL property. Contrarily to the algorithm presented in this paper, many metaheuristics consider a move that exchanges two rectangles in the order followed by BLF. This is the case of the hybrid tabu search / genetic algorithm designed by Iori et al. (Ref. [15]).

Hopper presented in Ref. [14] an improved strategy of BLF called BLD, where the objects are ordered using various criteria (e.g., height, width, perimeter) and the algorithm selects the best result obtained. Lesh et al. in Ref.[17] have improved the BLD heuristic. Their BLD^* strategy repeats greedy placements with a specific randomized ordering until a time limit is reached.

The Best-Fit (BF) greedy heuristic proposed by Burke et al. in Ref. [8] adopts in a sense a dual strategy while also respecting the Bottom[-Left] property. At each step, a most bottom location in the partial solution is considered, and the rectangle fitting best into it is selected, if any. In Ref. [9], Burke et al. improve their approach by using a metaheuristic phase (implemented by a tabu search, a simulated annealing or a genetic algorithm) for repairing the last part of the solution obtained by BF.

Finally, two last approaches must be mentioned and will constitute our main competitors. Bortfeldt proposes in Ref. [7] a very sophisticated genetic algorithm directly working with the geometry of the layout. The best algorithm for handling strip packing with rectangles of fixed orientation is a reactive GRASP algorithm (Ref. [1]) working as follows: all the rectangles are first placed on the strip with a randomized (and improved) BF greedy heuristic. Some rectangles on the top of the

strip are then removed and placed again with the greedy algorithm (in a different order). Several such steps are performed with an increasing portion of rectangles, adopting a variable neighborhood search strategy.

3. Maintaining "maximal holes"

The key idea behind our approach is to incrementally maintain a set of *maximal holes* when rectangles are added or removed.

Definition 1. (Maximal hole) Let us consider a container C partially filled with a set S of rectangles. A maximal hole H (w.r.t. C and S) is an empty rectangular surface in C such that:

- H does not intersect any rectangle in S (i.e., H is a "hole" in the container),
- H is maximal, i.e., there exists no hole H' such that H is included inside H' (notation: the inclusion of a rectangle H inside a rectangle H' will be denoted by $H \subset H'$)[a].

Fig. 1 shows three examples with resp. 2, 2 and 4 maximal holes (from left to right). The maximal hole in grey corresponds to the most bottom-left one.

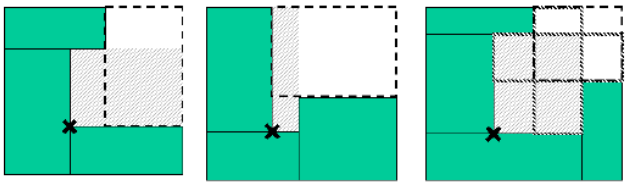

Fig. 1. Examples of maximal holes

Most of existing algorithms can use such a set of maximal holes for implementing their atomic operations. In particular, the BLF and BF greedy heuristics introduced above can implement the possible locations (into which the rectangles are added) as the set of maximal holes.

However, the interest is even greater. We claim that it is possible to design algorithms whose number of maximal holes remains small in practice during the search, even though the rectangles are removed, violating the BL property. The idea is the following. Thanks to the set of maximal holes, when a rectangle is removed, we do not modify the partial solution to make the rectangles BL again. Instead, we just update the set of maximal holes. Thus, a rectangle R placed in the future in

[a]This property implies that the bottom and left segments of H touch the container or rectangles in S.

the container will be BL (if it exactly covers a maximal hole)[b]. The main interest is to still limit the set of possible locations for the rectangles (to the maximal holes) while preserving the incrementality after a rectangle removal. In a sense, the good results obtained on strip packing problems by the metaheuristic proposed in this article experimentally validate this claim.

Atomic operations between two holes

Although other modelings are possible, a rectangle or a rectangular hole R is represented by four coordinates: $R.x_L$, $R.y_B$, $R.x_R$, $R.y_T$: $(R.x_L, R.y_B)$ represents the bottom-left corner of R while $(R.x_R, R.y_T)$ is the top-right corner.

The incremental additions and removals of rectangles into/from a container are based on two operations between rectangles and rectangular holes. The **Minus**(H, R) operation between a hole H and a rectangle R is used when a rectangle is added in a container. It returns the set of maximal holes that remain when (the newly added) R intersects H. A simple computation of the newly created maximal holes (`Holes`) is performed as follows:

(1) `Holes` $\leftarrow \emptyset$
(2) `If` $R.x_R < H.x_R$ `then Holes` $\leftarrow$ `Holes` $\cup \{(R.x_R, H.y_B, H.x_R, H.y_T)\}$ `EndIf`
(3) `If` $R.y_T < H.y_T$ `then Holes` $\leftarrow$ `Holes` $\cup \{(H.x_L, R.y_T, H.x_R, H.y_T)\}$ `EndIf`
(4) `If` $H.x_L < R.x_L$ `then Holes` $\leftarrow$ `Holes` $\cup \{(H.x_L, H.y_B, R.x_L, H.y_T)\}$ `EndIf`
(5) `If` $H.y_B < R.y_B$ `then Holes` $\leftarrow$ `Holes` $\cup \{(H.x_L, H.y_B, H.x_R, R.y_B)\}$ `EndIf`

`Minus`(H, R) may create less than four holes because the tested conditions are generally not fulfilled simultaneously.

The second operation **Plus**(H_1, H_2) holds between two rectangular maximal holes. If H_1 and H_2 intersect or are contiguous, `Plus` returns at most the following two new maximal holes:

- ($\max(H_1.x_L, H_2.x_L)$, $\min(H_1.y_B, H_2.y_B)$, $\min(H_1.x_R, H_2.x_R)$, $\max(H_1.y_T, H_2.y_T)$)
- ($\min(H_1.x_L, H_2.x_L)$, $\max(H_1.y_B, H_2.y_B)$, $\max(H_1.x_R, H_2.x_R)$, $\min(H_1.y_T, H_2.y_T)$)

Once again, a returned "degenerate" hole reduced to a single segment will not be considered.

Addition and removal of rectangles

Based on these operations, we present the two procedures used in most algorithms handling any 2D packing problem: `AddRectangle` and `RemoveRectangle`.

[b]We have done simply no effort in the metaheuristic presented hereafter to locally improve the layout when a (small) rectangle is added in a (large) hole so that the rectangle R is not BL. We let the evaluation of the objective function do the selection between neighbor candidates.

`AddRectangle`(in R, in/out C, in/out S) updates the set S of maximal holes when the rectangle R is added into the container (C is the set of rectangles placed in the container. At the beginning, S is reduced to the initial empty rectangular container.) `AddRectangle` mainly applies the `Minus` operator to R and to the holes in S that intersect R, as follows:

(1) Add R in C.
(2) For every hole in S intersecting R, add in a set `setH` the holes returned by `Minus`(H, R).
(3) Filter `setH` by preserving only the *maximal* holes.

Remarks:

- The case may occur that two holes H_1 and H_2, each created by two different calls to `Minus`, verify $H_1 \subset H_2$. This justifies the third step.
- (Correction) A proof by contradiction helps us to understand that a newly created hole needs not be "merged" with a contiguous hole H' which does not intersect R to (recursively) build a larger hole. Otherwise indeed, it would mean that H' was not maximal. This point has a significant and positive impact on the efficiency of the procedure that visits only holes intersecting R (and not their "neighbors").

The procedure `RemoveRectangle` is a bit more complex. `RemoveRectangle` (in R, in/out C, in/out S) updates the set S of maximal holes when the rectangle R is removed from the container. It replaces R by a hole H and applies the `Plus` operation on H and its contiguous holes (if any). A fixed-point process is applied to ensure completeness. The detailed pseudo-code is described hereafter.

Proposition 1. *(Termination and correction of `RemoveRectangle`) Let R be a rectangle to be removed from a container, let S be the corresponding set of maximal holes.*

A call to `RemoveRectangle` terminates and updates S with the set of all the maximal holes of the container.

Proof. (sketch; the full proof requires an induction)

The termination is based on several points:

- Like for `AddRectangle`, it is not necessary to visit holes that are not pushed initially in `HolePairs` (i.e., the procedure visits only the neighbors of R).
- Because the `Plus` operation does not return more than two holes, the number of holes in `Holes` never increases during the execution of `RemoveRectangle`.
- The items above and the definition of `Plus` imply that the union of all the holes created during the execution of `RemoveRectangle` does not change. In other words, the "surface" covered by all the considered holes (R and neighbor holes) is constant during the execution of `RemoveRectangle`.
- The `Plus`(H_1, H_2) operation generates at most two holes H_1', H_2' that are larger than or equal to the input holes.

```
Algorithm RemoveRectangle (in R, in/out C, in/out S)
    Remove R from C; Add R in S
    Initialize a list Holes with holes in S that are contiguous to R
    Initialize a list HolePairs with pairs (R, H) such that H is in Holes
    while HolePairs is not empty do
        Select (hole1, hole2) in Holepairs
        Remove (hole1, hole2) from HolePairs
        newHoles ← Plus (hole1, hole2) /* newHoles contains at most 2 holes */
        for every newHole in newHoles do
            for every hole in Holes do
                /* Ensure maximality */
                if newHole ⊂ hole then
                    delete newHole from newHoles; break
                else
                    if hole ⊂ newHole then
                        delete hole from Holes
                    end
                end
            end
            if newHole ∈ newHoles then
                Add newHole to Holes
            else
                Add to HolePairs all the pairs (newHole, H) such that H is in Holes
            end
        end
    end
end.
```

These points explain why a fixed-point is reached. The correction is ensured by the exhaustive application of the `Plus` operation to every possible pair. □

The `RemoveRectangle` procedure can be used by a classical 2D packing algorithm satisfying the BL property: when a rectangle is removed (or placed elsewhere in the container), the rectangles already placed in the container must be moved to recover the BL property, trying to limit the "fragmentation" of the container. However, this article explores the possibility of doing nothing special after a rectangle removal. Such an approach is described hereafter for solving strip packing.

4. An incremental move for strip packing

The strip packing is a variant of the 2D bin packing problem. A set of rectangles must be positioned in *one* container, called *strip*, which is a rectangular area. The strip has a fixed width dimension and a variable height. The goal is to place all the rectangles on the strip with no overlapping, using a minimum height of the container.

As said in the introduction, once we work in more than one dimension, the ob-

jects placed in the container are very dependent on each other and it is very difficult to incrementally repair the current solution. This explains why existing metaheuristics or genetic algorithms, allowing the search to escape from local minima, are not endowed with a low-cost "move". Most of the approaches use a classical greedy heuristic, e.g., BLF or BF. A widespread move consists, for instance, in exchanging two rectangles in the order in which the greedy heuristic will handle the rectangles. We understand that exchanging two rectangles i and j in the order implies, in the worst case, to position again all the rectangles after i in the order.

To handle strip packing, our metaheuristic uses a move based on the "geometry" of the rectangles on the strip. This move makes an intensive use of the incremental `AddRectangle` and `RemoveRectangle` procedures. It removes one rectangle R on the top of the layout and places it inside the strip. More precisely, the new location for R is a maximal hole or a placed rectangle. The rectangles of the layout that intersect R in its new location are placed again on the strip with a greedy heuristic such as Best-Fit Decreasing (`BF`). More precisely, a move is implemented as follows:

(1) Take one rectangle R the top side of which is the highest on the strip (the case may occur that several rectangles are candidates).
(2) Select R', which is one rectangle on the strip or one maximal hole, such that when R is placed in the bottom-left corner of R' then:
    - R remains inside the strip,
    - the new position of R is strictly lower than its previous position.
(3) Consider the set S of rectangles that would intersect R in its new position. The rectangles in S must be placed elsewhere. First remove them from the strip with calls to `RemoveRectangle`.
(4) Place R in the new position selected at step 2.
(5) Place again the rectangles in S with the greedy heuristic G.

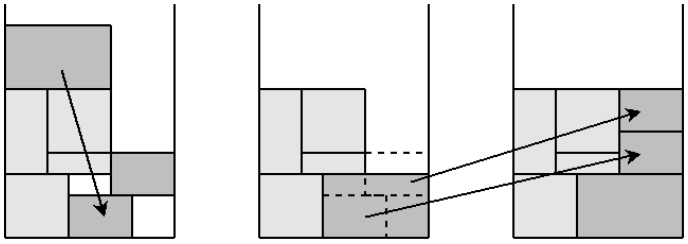

Fig. 2. One complete move

The steps 1 and 2 above pursue better solutions in an aggressive way (intensification). A similar move has been mentioned in Ref.[1] while it has not been used in their final heuristic.

The evaluation of the objective function (to be minimized) could be the (one-dimensional) height h of the layout. However, to break ties, we have also considered

the number u of units filled by rectangles on the highest line of the layout. The finer two-dimensional objective function is thus equal to $w \times (h-1) + u$, where w is the width of the strip.

In the strip packing variant where rectangles can rotate with an angle of 90 degrees, a move is modified as follows. The step 2 (resp. 4) above must also select an orientation for the rectangle R, with a probability 0.5 for both possibilities.

Time complexity

It appears that the notion of maximal holes, designed by *maximal areas*, has been introduced independently by El Hayek et al. in Ref. [12] for 2D bin packing problems. They have not detailed operators like `Minus`, `Plus`, `AddRectangle`, `RemoveRectangle`. However, they have proven that the maximum number a of maximal areas managed during a packing procedure is $O(n^2)$, where n is the number of rectangles to be placed.

Our operators `Minus`(H, R) and `Plus`(H_1, H_2) are $O(1)$ because they return at most respectively four and two new maximal holes. Therefore, our procedure `AddRectangle` is $O(a_1)$, where a_1 is the number of holes intersecting R ($a_1 \leq a$). Our procedure `RemoveRectangle` is $O(a_2^2)$, where a_2 is the number of holes contiguous to R, i.e., a_2 is the size of the list `Holes` in the pseudo-code ($a_2 \leq a$).

We report in Table 1 some statistics from experiments on Hopper and Turton's instances (every class contains three instances; see Section 6). They show that our procedures using maximal holes have a low time complexity in practice. This justifies why our move is claimed to be *incremental*.

The last five columns report statistics performed during the whole sequence of moves run to handle the different instances. The statistics first show (column 3) that the maximum number a of maximal holes grows linearly and is close to the number n of rectangles. Second, the number of displaced rectangles in one move (column 5) remains always very small on average. Finally, the last two columns concern `RemoveRectangle`. They report the maximum and average number a_2 of contiguous holes managed by `RemoveRectangle`. Note that a_2 remains small on average.

Table 1. Statistics on Hopper and Turton's instances.

Class	#rectangles	maximal holes	max rect.	aver. rect.	max cont. holes	aver. cont. holes
C1	17	16	08	1.9	10.00	2.95
C2	25	19	10	1.6	12.66	3.01
C3	29	26	11	2.0	18.00	3.76
C4	49	45	16	2.5	17.33	3.85
C5	73	55	18	2.4	21.00	3.73
C6	97	78	19	2.6	22.66	3.90
C7	197	145	25	2.8	30.33	4.21

Note: #rectangles: number of rectangles of the instance; maximal holes: maximum number of maximal holes during the search; max rect. (resp. aver. rect.): maximum (resp. average) number of displaced rectangles during one move; max (resp. aver.) cont. holes: maximum (resp. average) number of contiguous holes considered by `RemoveRectangle`.

Selected metaheuristic and greedy heuristics

We have designed an incomplete method able to be specialized with number of greedy heuristics and metaheuristics (performing a sequence of the moves described above). The method adopts the following scheme:

(1) a greedy heuristic first computes an initial layout,
(2) a metaheuristic, driven by an automatic tuning procedure, then repairs the first solution.

We have tried the main metaheuristics available in the `INCOP C++` library (Ref. [20]) developed by the first author: tabu search, simulated annealing (and a Metropolis variant), and `ID(best)` (Ref. [21]) which is a simple variant of `ID Walk` with only one parameter (`MaxNeighbors`). The simulated annealing has been discarded because it yields the worst performance on strip packing. Tabu search (with two parameters), `ID Walk` with two parameters and `ID(best)` gave a good performance, and we have chosen `ID(best)` for its simplicity. In particular, the automatic tuning procedure provided by `INCOP` is more robust when it tunes only one parameter.

`ID(best)` is a *candidate list strategy* that uses one parameter `MaxNeighbors` to perform one move from a configuration x to a configuration x', as follows:

(1) `ID(best)` picks randomly neighbor candidates one by one and evaluates them. The *first* neighbor x' with a cost better than or equal to the cost of x is accepted.
(2) If `MaxNeighbors` neighbors have been rejected, then the *best* neighbor among them (with a cost strictly worse than the cost of x) is selected.

`ID(best)` has a common behavior with a variant of tabu search once all the candidates have been rejected (item 2 above). However, the differences are the absence of tabu list and another policy for selecting a neighbor x' (step 1 above).

A description of the automatic tuning procedure can be found in Ref. [21]. Every trial is independent from the others and is interrupted when a maximal amount of CPU time is exceeded. A trial is a succession of one automatic *tuning step*, where the parameter `MaxNeighbors` is tuned in a dichotomic way on short walks, and one *exploring step* where the parameter is kept fixed during a long walk. After one tuning step and one exploring step, the trial is continued with a larger number of moves.

The same policy has been followed for all the tested strip packing benchmarks. In a same trial, the first tuning step runs 24 walks (with different values for the parameter) of 200 moves each; the first exploring step is a walk made of 10000 moves; the second tuning step runs 24 walks of 800 moves each; the second exploring step is a walk made of 40000 moves, and so on. It turns out that the tuning time represents about 30% of the global time.

The initial value of `MaxNeighbors` has been empirically set to the number of rectangles to be positioned. For the strip packing variant allowing the rotation of

rectangles, the initial value is the number of rectangles multiplied by 2.

Although simple, the automatic tuning procedure is robust (i.e., the tuned value converges and produces a good configuration) in a majority of trials.

Our algorithm works with any of the standard greedy heuristics: BF or BLF, considering one among the four possible orders among rectangles: largest Width first (w), largest Height first (h), largest Surface first (s), largest Perimeter (p), providing eight possible combinations: `BFw`, `BFh`, `BFs`, `BFp`, `BLFw`, `BLFh`, `BLFs`, `BLFp`[c]. The initial layout is provided by a greedy heuristic randomly picked among the eight ones.

During the moves performed by `ID(best)`, the choice of greedy heuristic has been directly incorporated into the neighborhood: in addition to the choice of rectangle R picked on the top of the layout and to the location in which the rectangle R will be moved, for placing again on the strip the displaced rectangles, one of the eight greedy heuristics is chosen at random.

This randomization presents two advantages. First, in our understanding, some biases are avoided. For instance, it is well-known that, when handling 2D packing with allowed rotation of rectangles, a bias introduced by the BF greedy heuristic is to place a lot of rectangles "vertically" on the right side of the strip. Second, it avoids the user to choose among the (eight) available greedy heuristics.

Thus, used with `ID(best)` and its automatically tuned parameter, the method proposed in this paper has simply *no* user-defined parameter.

5. Variants of our incomplete algorithm

We have added two features to our metaheuristic (denoted by `IDW` below). The first one consists in replacing the greedy heuristic used for the first layout with a hyper-heuristic (`HH`) recently proposed by some of the authors in Ref. [2] (the approach is denoted by `HH+IDW` below). The second feature consists in "rePacking" the layout at times for recovering the BL property (the corresponding approaches are denoted by `IDW+P` and `HH+IDW+P` below).

5.1. *First layout obtained by a hyperheuristic*

The hyperheuristic framework manages a set of low-level heuristics and tries to find a way to apply them. In Ref. [2], the authors have designed a hyperheuristic for handling strip packing instances. The hyperheuristic builds a sequence of greedy heuristics. Each element of the sequence places a given number of rectangles on the strip with the corresponding greedy heuristic. The hyperheuristic performs a hill-climbing on the sequence by applying moves that add, remove or replace elements (i.e., greedy heuristics) in the sequence.

[c]On the tested instances, among the eight combinations, some greedy heuristics are better than some others *on average*, but none of them is always strictly dominated by one of the others.

Four greedy heuristics are used inside the hyperheuristic: the standard BLF and BF heuristics mentioned above; the recursive heuristic HR which is also used for problems respecting the guillotine cut constraint (Ref. [22]); the BFDH* heuristic used by Bortfeldt for generating the initial configurations in its genetic algorithm (Ref. [7]).

Our metaheuristic has been extended by using this hyperheuristic to generate the first layout, yielding the `HH+IDW` variant.

5.2. *Repacking procedure*

The idea behind our metaheuristic was to not recover the BL property every time a rectangle is removed during the moves performed by local search. Instead, the set of maximal holes is incrementally recomputed. However, this is not contradictory with calls to a *repacking procedure* (recovering the BL procedure), provided that the time spent by the repacking procedure is dominated by that of the local search.

The repacking procedure is a simple loop on all the rectangles of the current layout handled in a bottom-left order. At the i^{th} iteration:

- the rectangles from 1 to $i-1$ in the order verify the BL property,
- the i^{th} rectangle is removed from the layout and placed again by a BLF-like greedy heuristic (i.e., in the most bottom-left maximal hole of the current layout), making BL the rectangles from 1 to i in the order.

The main difference with the BLF greedy heuristic is that all the rectangles are considered to be on the layout when the i^{th} rectangle is handled: the rectangles from 1 to $i-1$ in the order, but also the other ones. At the end of this loop, all the rectangles thus verify the BL property.

It is straightforward to prove by induction that the layout obtained by such a repacking procedure is better than or equal to the initial layout. Indeed, every rectangle does not move or is displaced in a most bottom-left position.

We have decided to call the repacking procedure so occasionally that the time spent is dominated by the local search. We also wanted the local search to run the repacking procedure in an adaptive way. That is why the repacking procedure is launched when the lattest move led the current solution to a significantly worse cost, i.e., an increase of 1 of the upper line of the layout. In our experiments below, we have evaluated the part of the repacking procedure in the overall CPU time. This ratio is comprised between 3% and 20% according to the considered instance. A ratio of 20% occurs when the selected neighborhood is very small (i.e., `MaxNeighbors` is small) and a lot of accepted moves in the walk increase the cost.

Interleaving this rePacking procedure with our metaheuristic leads to the `IDW+P` variant. Adding as a preprocessing a call to `HH` leads to the `HH+IDW+P` variant.

6. Experiments

We have performed experiments on five series enclosing 547 benchmarks. The 21 zero-waste instances by Hopper and Turton [14] are classified into 7 classes of increasing strip width. The corresponding results are reported in Tables 2 and 5. Tables 3 and 6 show the results obtained on the 13 *gcut* instances by Beasley [4], the 3 *cgcut* instances, and the 10 *beng* instances by Bengtsson [5]. The results obtained on the 12 *ngcut* instances also proposed by Beasley are not reported because they are all optimally solved by our metaheuristic (in less than 3 seconds) and by competitors. Note that the `gcut` instances have a reasonable number of rectangles but have a wide strip ranging from 250 to 3000 units.

Tables 4 and 7 include the results obtained on the 500 instances proposed by Martello and Vigo [19], Berkey and Wang [6]. This huge number of instances are classified into 10 classes, themselves subdivided into 5 series of 10 instances each. The classes define different strip widths ranging from 10 to 300. The 5 series define instances with resp. 20, 40, 60, 80 or 100 rectangles. Finally, Table 8 reports the results obtained on the Hopper and Turton instances (Ref. [14]) for a variant of strip packing where rectangles can rotate with an angle of 90 degrees.

Competitors

Not all the presented competitors have tested the five presented benchmark series. Also, they have adopted slightly different experimental conditions.

The hybrid tabu/genetic algorithm is run by Iori during 300 seconds on a Pentium III at 800 Mhz (Ref. [15]). The BLD* algorithm is run by Lesh et al on a Pentium at 2 Ghz (Ref. [17]). The two presented results correspond to time limits of respectively 60 seconds and 3600 seconds.

The results presented for Burke et al. correspond to their BF heuristic enhanced with tabu search, simulated annealing or a genetic algorithm. They run their heuristic 10 times with a time limit of 60 seconds per run on a Pentium IV at 2 Ghz. Bortfeldt's genetic algorithm is run 10 times on every instance with an average time per run of 160 seconds on a Pentium at 2 Ghz. The GRASP algorithm (Ref. [1]) is also run 10 times on every instance with a time limit of 60 seconds on Pentium IV Mobile at 2 Ghz.

Experimental conditions

For every instance, our metaheuristic (`IDW`, `HH+IDW`, `IDW+P` or `HH+IDW+P`) spends a total time of 1000 seconds on a Pentium IV at 2.66 Ghz. This time corresponds to 10 runs of 100 seconds each. The same amount of CPU time is allocated to `HH`. When `HH` computes the first layout used by `IDW` (resp. `IDW+P`), 10 seconds are allocated to `HH` while 90 seconds are used for `IDW` (resp. `IDW+P`).

For all the heuristics, we report the best bound obtained. The average bounds are not reported in the tests reported in Section 6.1 because they are not always

available in the literature and are indeed sometimes meaningless for certain algorithms. However, the average costs appear in Section 6.2 that recalls the results of the best competitors.

Although all the algorithms have been performed neither on the same computers nor in exactly the same amount of CPU time, we think that the comparison between competitors is rather fair. Indeed, if the machines and the allowed times were normalized, the (ratio) differences between competitors would not exceed a factor 2. It is not sufficient to bring a significant gap in terms of computed bound on the tested instances of this NP-hard strip packing problem.

6.1. *Comparison of our metaheuristic with competitors*

Table 2

Every class of the Hopper and Turton's instances contains 3 zero-waste instances with a given width (column Width), a given number of rectangles (N), and a given optimum - the ordinate of the top side of a highest rectangle in the strip - obtained by construction (*Opt.*). The cells report the average percentage deviation from optimum. The reported results for Burke's algorithm is their best tested metaheuristic: BF + simulated annealing. The reported results for Lesh et al's algorithm were obtained in 3600 seconds.

GRASP outperforms the other algorithms, especially on the largest classes 6 and 7. `ID Walk` is generally better than other competitors (except GRASP). Bortfeldt's approach behaves well on class 7.

Table 2. Comparison on Hopper and Turton instances.

Class	Width	N	Opt.	`IDW`	Iori	Lesh	Burke	Bortfeldt	GRASP
C1	20	16–17	20	0.00	1.59	–	0.00	1.59	0.00
C2	40	25	15	0.00	2.08	–	6.25	2.08	0.00
C3	60	28–29	30	2.15	2.15	–	3.23	3.23	1.08
C4	60	49	60	1.64	4.75	–	1.64	2.70	1.64
C5	60	73	90	1.81	3.92	2.17	1.46	1.46	1.10
C6	80	97	120	1.37	4.00	1.64	1.37	1.64	0.83
C7	160	196.3	240	1.77	–	–	1.77	1.23	1.23
# optimal solutions				7/21	5/18	0/6	3/21	4/21	8/21

Table 3

The column LB yields Lower Bound computations of the optima (which are not necessarily reached). The following columns report the bound of the best solution computed by the corresponding algorithm. We report the bound of the best solution obtained by Lesh et al.'s algorithm in resp. 60 and 3600 seconds. The benchmarks `gcut09'...gcut13'` (Ref. [1]) are variants of the benchmarks `gcut09...gcut13` in which the rectangles have been rotated with an angle of 90 degrees (i.e., the width and the height of every rectangle have been exchanged).

On the three presented categories, note that GRASP is generally better than or similar to `ID Walk` (except on *gcut09'*) which is itself better than Iori's algorithm and Lesh's approach in 3600 seconds.

Table 3. Comparison on *beng*, *cgcut* and *gcut* instances.

Instance	Width	N	LB	`IDW`	Iori	Lesh 60	Lesh 3600	GRASP
beng01	25	20	30	30	31	–	–	30
beng02	25	40	57	58	58	–	–	57
beng03	25	60	84	85	86	–	–	84
beng04	25	80	107	108	110	–	–	107
beng05	25	100	134	135	136	–	–	134
beng06	40	40	36	36	37	–	–	36
beng07	40	80	67	68	69	–	–	67
beng08	40	120	101	102	–	–	–	101
beng09	40	160	126	126	–	–	–	126
beng10	40	200	156	156	–	–	–	156
cgcut01	10	16	23	23	23	–	–	23
cgcut02	70	23	63	65	65	–	–	65
cgcut03	70	62	636	675	676	–	–	661
gcut01	250	10	1016	1016	1016	1016	1016	1016
gcut02	250	20	1133	1194	1207	1211	1195	1191
gcut03	250	30	1803	1803	1803	1803	1803	1803
gcut04	250	50	2934	3030	3130	3072	3054	3002
gcut05	500	10	1172	1273	1273	1273	1273	1273
gcut06	500	20	2514	2686	2675	2682	2656	2627
gcut07	500	30	4641	4697	4758	4795	4754	4693
gcut08	500	50	5703	5960	6240	6181	6081	5908
gcut09'	1000	10	2022	2241	–	–	–	2256
gcut10'	1000	20	5356	6399	–	–	–	6393
gcut11'	1000	30	6537	7736	–	–	–	7736
gcut12'	1000	50	12522	13172	–	–	–	13172
gcut13'	3000	32	4772	5055	–	–	–	5009

Table 4

The cells in Table 4 include the percentage deviation between the (not necessarily reached) lower bound and a value v: v is an average value over the 50 best solution costs obtained for the 50 instances in a given class.

From best to worst, the order between competitors is GRASP, `ID Walk`[d], Bortfeldt's algorithm, Lesh's algorithm, Iori's algorithm.

[d]The tests reported in this table correspond to an old version of our metaheuristic, where `ID Walk` is run 20 times per instance, with two specified greedy heuristics: 10 trials with `BFw`, and 10 trials with `BLFw` or `BFh`, depending of the considered instance. However, on the same instances, the results of the variants presented in Table 7 have been obtained with the described version, where the selected greedy heuristics are no more defined by the user.

Table 4. Comparison on the 500 instances proposed by Martello, Vigo, Berkey, Wang.

Class	Width	IDW	Iori	Lesh 60	Lesh 3600	Bortfeldt	GRASP
01	10	0.67	0.64	0.81	0.68	0.75	0.63
02	30	0.58	1.78	1.12	0.42	0.88	0.10
03	40	2.16	3.05	2.71	2.23	2.52	1.73
04	100	3.47	5.08	4.41	3.54	3.19	2.02
05	100	2.20	3.15	2.85	2.43	2.59	2.05
06	300	4.86	5.99	6.45	5.13	4.96	3.08
07	100	1.12	1.16	1.17	1.12	1.19	1.10
08	100	4.19	6.16	5.99	4.93	3.85	3.57
09	100	0.07	0.07	0.07	0.07	0.07	0.07
10	100	3.12	4.67	4.11	3.48	3.05	2.93
Overall		2.24%	3.17%	2.97%	2.40%	2.31%	1.73%

6.2. *Results obtained by variants of our metaheuristic*

Tables 5, 6 and 7 report a comparison between different variants of our metaheuristic (IDW): HH+IDW, IDW+P and HH+IDW+P. The tables also show the best and average times of HH and of our best competitor: GRASP.

These tables mainly underline the good performance obtained by a hybridization between HH and IDW which nearly reaches (and sometimes exceeds) the performance of GRASP.

Table 5. Comparison on Hopper and Turton's instances.

Class	N	Opt.	IDW		HH+IDW		IDW+P		HH+IDW+P		HH		GRASP	
			best	aver.	best	aver.	best	aver.	best	aver.	best	aver.	best	aver.
C1	16–17	20	0.0	0.79	0.0	0.48	0.0	0.16	0.0	0.95	1.59	2.38	0.0	0.0
C2	25	15	0.0	3.12	0.0	1.87	0.0	1.90	0.0	1.87	0.0	1.46	0.0	0.0
C3	28–29	30	2.15	3.10	1.08	2.58	2.15	2.90	2.15	2.26	2.15	2.80	1.08	1.08
C4	49	60	1.64	2.81	1.64	2.43	2.17	2.70	1.64	2.49	2.70	3.22	1.64	1.64
C5	73	90	1.81	2.73	1.10	1.78	1.81	2.52	1.10	1.67	1.10	2.24	1.10	1.10
C6	97	120	1.37	2.49	1.10	1.75	1.37	2.25	1.10	1.48	1.64	1.93	0.83	1.56
C7	196.3	240	1.77	2.74	1.10	1.42	1.67	2.53	1.10	1.34	1.10	1.40	1.23	1.36

6.3. *Results on Hopper and Turton's instances with rotation*

Table 8 reports the result of the variant of strip packing where rectangles can rotate with an angle of 90 degrees.

The first results were reported by Hopper and Turton (column HT) themselves in 2000 (Refs. [13,14]). Note that IDW can find the 3 optima of the class 3 (i.e., an average deviation of 0.00) and one optimum of the class 5 (0.74) in 1000 seconds per run when it is manually tuned with the BFs greedy heuristic.

6.4. *Synthesis*

Iori's algorithm is generally the worst one on the tested instances. It is better than IDW only on gcut06 and on the class 1 by Martello/Vigo. None of these results hold

Table 6. Comparison on *beng*, *cgcut* and *gcut* instances.

Instance	Width	N	LB	IDW		HH+IDW		IDW+P		HH+IDW+P		HH		GRASP	
				best	aver.	best	aver.	best	aver.	best	aver.	best	aver.	best	aver.
beng01	25	20	30	30	30.7	30	30.8	30	30.5	30	30.4	30	30.5	30	30.0
beng02	25	40	57	58	58.0	58	58.0	58	58.1	58	58.0	57	57.0	57	57.0
beng03	25	60	84	85	85.3	84	84.6	84	85.2	84	84.5	85	85.0	84	84.0
beng04	25	80	107	108	108.9	108	108.1	108	108.9	107	107.9	108	108.6	107	107.0
beng05	25	100	134	135	135.8	134	134.0	134	135.2	134	134.0	134	134.0	134	134.0
beng06	40	40	36	36	36.0	36	36.0	36	36.0	36	36.0	36	36.0	36	36.0
beng07	40	80	67	68	68.0	67	67.8	68	68.0	67	67.3	67	67.9	67	67.0
beng08	40	120	101	102	102.4	101	101.0	102	102.4	101	101.0	101	101.0	101	101.0
beng09	40	160	126	126	126.5	126	126.0	126	126.6	126	126.0	126	126.0	126	126.0
beng10	40	200	156	156	157.2	156	156.0	156	157.2	156	156.0	156	156.0	156	156.0
cgcut01	10	16	23	23	23.0	23	23.0	23	23.0	23	23.0	23	23.0	23	23.0
cgcut02	70	23	63	65	65.5	65	65.0	65	65.5	65	65.0	65	65.0	65	65.0
cgcut03	70	62	636	675	680.2	663	667.8	671	677.3	662	667.9	662	665.5	661	661.0
gcut01	250	10	1016	1016	1016.0	1016	1016.0	1016	1016.0	1016	1016.0	1016	1016.0	1016	1016.0
gcut02	250	20	1133	1194	1214.4	1196	1205.8	1204	1209.0	1195	1203.9	1196	1206.1	1191	1191.0
gcut03	250	30	1803	1803	1808.8	1803	1803.0	1803	1803.0	1803	1803.0	1803	1803.0	1803	1803.0
gcut04	250	50	2934	3030	3100.5	3011	3028.9	3031	3075.6	3020	3032.0	3019	3025.8	3002	3002.0
gcut05	500	10	1172	1273	1273.0	1273	1273.0	1273	1273.0	1273	1273.0	1284	1287.0	1273	1273.0
gcut06	500	20	2514	2686	2706.9	2644	2659.6	2646	2668.2	2639	2651.1	2644	2654.2	2627	2627.0
gcut07	500	30	4641	4697	4769.4	4702	4703.5	4694	4737.1	4704	4710.1	4694	4705.0	4693	4693.0
gcut08	500	50	5703	5960	6061.3	5915	5957.5	5891	5977.6	5895	5947.7	5922	5951.5	5908	5912.2
gcut09	1000	10	2022	2317	2317.0	2317	2317.0	2317	2317.0	2317	2317.0	2317	2317.0	–	–
gcut10	1000	20	5356	5973	6049.4	5965	5983.1	5970	5990.9	5969	5972.0	5965	5995.7	–	–
gcut11	1000	30	6537	7066	7139.5	6980	7043.6	7043	7111.7	6966	6994.1	6973	7029.7	–	–
gcut12	1000	50	12522	14690	14762	14690	14690	14690	14690	14690	14690	14690	14690	–	–
gcut13	3000	32	4772	4998	5063.5	4944	4986.8	4994	5012.7	4914	4975.7	4945	5019.0	–	–
gcut09'	1000	10	2022	2241	2248.5	2254	2257.2	2241	2248.6	2241	2251.3	2290	2291.3	2256	2256.0
gcut10'	1000	20	5356	6399	6470.7	6399	6408.0	6399	6427.0	6422	6427.3	6402	6426.3	6393	6393.0
gcut11'	1000	30	6537	7736	7749.2	7736	7736.0	7736	7736.0	7736	7736.0	7736	7736.0	7736	7736.0
gcut12'	1000	50	12522	13172	13646	13172	13217	13172	13523	13172	13213	13172	13183	13172	13172
gcut13'	3000	32	4772	5055	5104.5	5037	5078.7	5061	5084.0	5028	5075.4	5028	5070.2	5009	5009.5

Table 7. Comparison on the 500 instances proposed by Martello, Vigo, Berkey, Wang.

Class	Width	IDW	HH+IDW	HH+IDW+P	HH	GRASP
		best	best	best	best	best
01	10	0.67	0.64	0.65	0.72	**0.63**
02	30	0.58	0.25	0.16	0.34	**0.10**
03	40	2.16	**1.71**	1.72	1.72	1.73
04	100	3.47	2.44	2.40	2.60	**2.02**
05	100	2.20	2.08	2.09	**2.05**	**2.05**
06	300	4.86	3.71	3.72	3.80	**3.08**
07	100	1.12	1.13	1.13	1.13	**1.10**
08	100	4.19	3.81	3.68	3.74	**3.57**
09	100	0.07	0.07	0.07	0.07	0.07
10	100	3.12	2.80	**2.78**	2.80	2.93
Overall		2.24%	1.86%	1.84%	1.90%	**1.73%**

against `HH+IDW[+P]`. This provides an experimental evidence (by contradiction) that a good approach for handling strip packing should be based on the geometry of the layout.

Lesh's algorithm behaves sometimes well but does not improve its solution a lot when spending more time (e.g., from 60 s to 3600 s). It is better than `IDW` only on `gcut06` and on the class 2 by Martello/Vigo. This highlights the interest of a metaheuristic able to escape from local minima.

Table 8. Comparison on Hopper and Turton's instances with non-fixed orientation of rectangles.

Class	N	Opt.	IDW		HH+IDW		IDW+P		HH+IDW+P		HH		HT	Bortfeldt
			best	aver.	best	aver.	best	aver.	best	aver.	best	aver.	best	best
C1	16–17	20	0.00	1.17	0.00	1.50	1.67	1.67	0.00	0.17	1.67	2.50	4	1.70
C2	25	15	0.00	4.44	0.00	0.22	0.00	3.11	0.00	0.44	0.00	0.00	6	0.00
C3	28–29	30	3.33	3.33	2.22	3.00	2.22	3.00	2.22	2.44	2.22	3.00	5	2.22
C4	49	60	1.67	2.28	1.67	1.72	1.67	1.94	1.67	1.72	1.67	2.50	3	0.00
C5	73	90	1.48	2.15	1.11	1.19	1.48	2.07	1.11	1.15	1.11	1.22	3	0.00
C6	97	120	1.67	2.14	0.83	1.11	1.67	1.83	0.83	1.14	0.83	1.56	3	0.33
C7	196.3	240	2.08	2.54	0.83	1.19	1.80	2.28	0.83	1.19	0.69	1.33	4	0.33
Overall			1.46		0.95		1.50		0.95		1.17		4	0.654

Burke's algorithm applied to Hopper and Turton's instances seems competitive with `IDW` only on large instances, while it is not competitive with `HH+IDW[+P]`. The same conclusion can be drawn when comparing Bortfeldt's approach, `IDW` and `HH+IDW[+P]` (on the problem with no allowed rotation of rectangles).

Thus, on strip packing with rectangles of fixed orientation, `IDW` generally outperforms the other competitors, but it is generally worse than (or equal to) GRASP (except for `gcut09'`), which highlights the interest of a sophisticated and randomized greedy heuristic (based on BF). If we compare GRASP and `HH+IDW+P`, both algorithms obtain similar best solutions. The difference between both is small. Both are close to each other on Hopper and Turton's instances, `beng` instances and `cgcut` instances. GRASP remains slightly better on `gcut` instances. Note that `HH+IDW+P` outperforms GRASP on the Hopper and Turton's class 7 (thanks to `HH`), on `gcut09'` (thanks to `IDW`), on `gcut08` (thanks to `IDW+P`), and on classes 3 and 10 by Martello et al. (thanks to `HH`).

We must observe the very good results obtained by `HH`. However, the approach fails on certain instances that are generally easy for other algorithms, e.g., on Hopper and Turton's class 1 or on `gcut05`. This could come from a current lack of the approach that may be unable to reach any point in the search space. The hybridization between `IDW` and `HH` is particularly beneficial since the hybrid version is always competitive and outperforms sometimes `IDW` and `HH` individually (see for instance the class 6 by Hopper and Turton, `beng03`, `beng04`, `gcut11`, `gcut13`).

On the variant with non-fixed orientation of rectangles, it is difficult to evaluate our approach due to the lack of competitors. `IDW` seems to behave well. It is far above Hopper and Turton's algorithm but it is below Bortfeldt's algorithm on large instances. However, the difference between Bortfeldt's algorithm and `HH+IDW` is small. The good behavior of Bortfeldt's algorithm might be explained by its postprocessing phase performed on non-guillotine instances.

Interest of using maximal holes

Two points highlight the interest of the maximal holes: the good performance obtained by our approach and the slight advantage given to the repacking procedure.

First, it is worthwhile to underline that the good behavior of our method is mainly due to our incremental move. Our move makes an intensive use of `AddRectangle` and `RemoveRectangle` whose time complexities are closely related

to the maximal holes.

Second, the interest of the repacking procedure is not significant. `IDW+P` almost always outperforms `IDW` if we compare the average bounds, but the comparison on the best bounds is not so clear. Also, the difference is even less significant when `HH` is used to compute the first layout. In other terms, `HH+IDW` and `HH+IDW+P` obtain very similar results. This provides an experimental evidence that maintaining the bottom-left property during the search is not crucial.

7. Conclusion

The contribution described in this paper is twofold. First, we have proposed incremental operators to maintain a set of maximal holes during the addition/removal of rectangles into/from a container for any 2D packing problem. We have suggested to relax the BL property which is respected by most of complete and incomplete algorithms. Second, we have designed a metaheuristic for handling 2D strip packing, endowed with an incremental move based on the geometry of the layout, and maintaining the set of maximal holes. In particular, this metaheuristic has no user-defined parameter and no greedy heuristic to be specified. This metaheuristic behaves well on the tested benchmarks.

We have designed more efficient variants of this metaheuristic that start with a better first layout provided by a hyperheuristic (`HH`). These variants are really competitive with state-of-the-art algorithms. This hybridization is particularly beneficial since, although `HH` often shows a good performance, `HH` cannot efficiently handle certain instances on which `IDW` and `HH+IDW` behave well.

The good performance obtained by GRASP, Bortfeldt's algorithm, `HH` or our metaheuristic yields an experimental evidence that the best methods for handling strip packing:

- exploit the geometry of the layout,
- make use of several well-known greedy heuristics or of a sophisticated one.

It turns out that all the efficient approaches (except ours) implement improved greedy heuristics: Bortfeldt's algorithm uses the BFDH* heuristic while GRASP uses a very sophisticated BF-like heuristic. Also, the hyperheuristic approach tries to better exploit the best greedy heuristics known for strip packing. This suggests that our method could be improved by using more sophisticated greedy heuristics.

The greedy heuristic proposed by Chen and Huan in Ref. [11] will be studied in a future work. Although their heuristic does not handle strip packing but 2D rectangle packing (i.e., 2D bin packing with a unique bin), they obtain impressive results on Hopper and Turton's instances. One reason is that they determine in advance the height of the container, using the fact that these instances are zero-waste. However, the good performance could also be due to a great attention paid by their heuristic when selecting the next rectangle to be placed in the container. Such greedy algorithms can also benefit from maintaining the set of maximal holes.

References

1. R. Alvarez-Valdes, F. Parreño, and J.M. Tamarit. Reactive GRASP for the Strip Packing Problem. *Computers and Operations Research*, 35:1065–1083, 2008.
2. I. Araya, B. Neveu, and M-C. Riff. An Efficient Hyperheuristic for Strip Packing Problems. In C. Cotta, M. Sevaux, and K. Sarensen, editors, *Adaptive and Multilevel Metaheuristics*, volume 136 of *Studies on Computational Intelligence*. Springer, 2008.
3. B.S. Baker, E.G. Coffman, and R.L. Rivest. Orthogonal Packings in 2D. *SIAM Journal on Computing*, 9:846–855, 1980.
4. J.E. Beasley. Algorithms for Unconstrained Two-Dimensional Guillotine Cutting. *J. of the operational research society*, 33:49–64, 1985.
5. B.E. Bengtsson. Packing Rectangular Pieces – A Heuristic Approach. *The computer journal*, 25:353–357, 1982.
6. J.O. Berkey and P.Y. Wang. Two-Dimensional Finite Bin Packing Algorithms. *Journal of the operational research society*, 38:423–429, 1987.
7. A. Bortfeldt. A Genetic Algorithm for the Two-Dimensional Strip Packing Problem with Rectangular Pieces. *European J. of Operational Research*, 172:814–837, 2006.
8. E. Burke, G. Kendall, and G. Whitwell. A New Placement Heuristic for the Orthogonal Stock Cutting Problem. *Operations Research*, 52:697–707, 2004.
9. E. Burke, G. Kendall, and G. Whitwell. Metaheuristic Enhancements of the Best-Fit Heuristic for the Orthogonal Stock Cutting Problem. *Submitted in INFORMS*, 2006.
10. B. Chazelle. The Bottom Left Bin Packing Heuristic: An Efficient Implementation. *IEEE Transactions on Computers*, 32:697–707, 1983.
11. D. Chen and W. Huang. A Novel Quasi-Human Heuristic Algorithm for Two-Dimensional Rectangle Packing Problem. *International Journal of Computer Science and Network Security*, 6(12):115–120, 2006.
12. J. El Hayek, A. Moukrim, and S. Negre. New Resolution Algorithm and Pretreatments for the Two-Dimensional Bin-packing Problem. *Computers & Operations Research*, 35(10):3184–3201, 2008.
13. E. Hopper. *Two-Dimensional Packing Utilising Evolutionary Algorithms and Other Meta-Heuristic Methods*. PhD. Thesis Cardiff University, 2000.
14. E. Hopper and B.C.H. Turton. An Empirical Investigation on Metaheuristic and Heuristic Algorithms for a 2D Packing Problem. *European Journal of Operational Research*, 128:34–57, 2001.
15. M. Iori, S. Martello, and M. Monaci. *Metaheuristic Algorithms for the Strip Packing Problem*, pages 159–179. Kluwer Academic Publishers, 2003.
16. N. Lesh, J. Marks, A. Mc. Mahon, and M. Mitzenmacher. Exhaustive Approaches to 2D Rectangular Perfect Packings. *Information Processing Letters*, 90:7–14, 2004.
17. N. Lesh, J. Marks, A. Mc. Mahon, and M. Mitzenmacher. New Heuristic and Interactive Approaches to 2D Strip Packing. *ACM J. of Exp. Algorithmics*, 10:1–18, 2005.
18. S. Martello, M. Monaci, and D. Vigo. An Exact Approach to the Strip Packing Problem. *INFORMS Journal of Computing*, 15:310–319, 2003.
19. S. Martello and D. Vigo. Exact Solution of the Two-Dimensional Finite Bin Packing Problem. *Management science*, 15:310–319, 1998.
20. B. Neveu and G. Trombettoni. INCOP: An Open Library for INcomplete Combinatorial OPtim. In *Proc. Constraint Programming, LNCS 2833*, pages 909–913, 2003.
21. B. Neveu, G. Trombettoni, and F. Glover. ID Walk: A Candidate List Strategy with a Simple Diversification Device. In *Proc. Constraint Programming CP'04, LNCS 3258*, pages 423–437, 2004.
22. D. Zhang, Y. Kang, and A. Deng. A New Heuristic Recursive Algorithm for the Strip Packing Problem. *Computers and Operations Research*, 33:2209–2217, 2006.