Incremental Move for Strip-Packing

Bertrand Neveu, Gilles Trombettoni
INRIA Sophia-Antipolis, University of Nice-Sophia, ENPC, Projet COPRIN, France
{neveu, trombe}@sophia.inria.fr

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

Abstract

When handling 2D packing problems, numerous incomplete and complete algorithms maintain a so-called bottom-left (BL) property: every rectangle placed in a container is propped up bottom and left. While it is easy to make a rectangle BL when it is 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 only 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 generic metaheuristic using this move and standard greedy heuristics. Experimental results show that the approach is competitive with the best known incomplete algorithms, especially the other metaheuristics (able to escape from local minima).

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 to extract the objects; 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.

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 all the rectangles are propped up left and bottom. They also maintain the BL property when a rectangle R is removed from the container (or placed in another location), which often implies that rectangles above R or to the right side of R must be moved as well. First, the BL property lowers the number of possible locations for rectangles. Second, it can be proven that a simple operation can transform any solution of a 2D packing problem into a solution respecting the BL property. However, this operation is not local to the removed rectangle and to its neighbors, but modifies the whole layout in the worst case (e.g., when the rectangle placed on the bottom-left corner of the container is removed).

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 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 maximal holes during the addition and removal of rectangles. The experiments presented in Section 5 show the interest of the metaheuristic based on this 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 [13]. Although not competitive with incomplete approaches, the branch and bound algorithm proposed by Martello et al. is interesting [15] 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 [10] 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) [9] is a generalization of the first greedy heuristic proposed by Baker et al. [2] 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. That is why the built layout always maintains the BL property. Contrarily to the algorithm presented in this paper, many metaheuristics consider a move that exchange two rectangles in the order followed by BLF. This is in particular the case of the hybrid tabu search / genetic algorithm designed by Iori et al. [12].

Hopper [11] presented 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 [14] 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 [7] 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[1]. In [8], 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 [6] proposes 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 [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.

[1]Three variants in fact exist where the considered hole is the most bottom-left, the most bottom-right, or randomly chosen between both. The whole heuristic returns the best solution obtained by these three variants.

3 Maintaining "maximal holes"

The key idea behind our approach is to incrementally maintain a set of *maximal holes* when rectangles are added and 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 a 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 is 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 $H \subset H'$)*[2].

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.

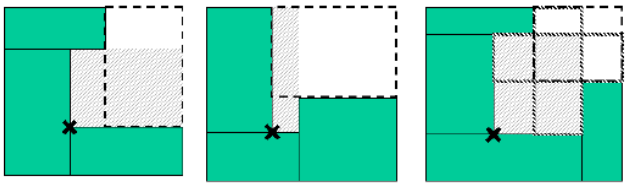

Figure 1. Examples of maximal holes

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 (in 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 the container will be propped up bottom and left (if it exactly covers a maximal hole)[3]. 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.

[2]This property implies that H is propped up bottom and left w.r.t. the container or rectangles in S.

[3]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.

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

`AddRectangle`(in R, in/out C, in/out S) updates the set S of maximal holes when the rectangle R is added to 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` and preserve 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. Details are given below.

```
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.
```

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`, this 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.

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.

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 objects placed in the container are very dependent 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 really incremental. Most of the approaches are based on a classical greedy heuristic, e.g., BLF or BF, and 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.

In this paper, we propose a generic metaheuristic based on a more incremental move. This incomplete algorithm uses any metaheuristic M (such as tabu search or simulated annealing) and any greedy algorithm G. It also uses a move based on the "geometry" of the rectangles on the strip. Informally, any move (or any evaluated candidate neighbor) is performed by taking a rectangle R on the top of the strip and putting it in the middle of the strip. 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. With `RemoveRectangle`, remove the set S of rectangles that would intersect R in its new position, including R' if it is a rectangle (the rectangles in S must be placed elsewhere).
4. Place R in the new position selected at step 2.
5. Place again the rectangles in S with the greedy heuristic G.

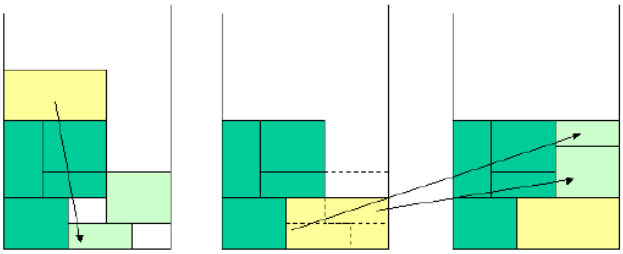

Figure 2. One complete move

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

The evaluation of the objective function (to be minimized) could be the (one-dimensional) ordinate of the highest side of a rectangle on the strip. However, we have selected a finer two-dimensional objective function equal to $w \times h + n$, where w is the width of the strip, h is the ordinate of the top side of a highest rectangle on the strip *minus one*, and n is the number of units filled by rectangles on the highest line of the strip.

A move in the 2D strip-packing variant where rectangles can rotate with an angle of 90 degrees is modified as follows. The step 1 above must also select an orientation for the rectangle R, with a probability 0.5 for both possibilities.

We report in Table 1 some statistics from the experiments with the Hopper and Turton instances (5) that show that the number of possible locations for a rectangle (maximal holes + placed rectangles) grows in a linear way w.r.t the number of rectangles, and that the number of displaced rectangles in one move remains always very small in average.

Class	N	maxpl	rectmax	rectav
C1	16–17	33	8	1.9
C2	25	44	10	1.6
C3	28–29	55	11	2.0
C4	49	94	16	2.5
C5	73	128	18	2.4
C6	97	175	19	2.6
C7	196–197	342	25	2.8

Table 1. Statistics from H-T instances. N: number of rectangles, maxpl: maximum number of possible locations, rectmax (rectav): maximum (average) number of displaced rectangles during one move.

Selected metaheuristic and greedy heuristics

We wanted to propose a generic metaheuristic able to be specialized with number of metaheuristics and greedy heuristics. We have tried the main metaheuristics available in the `INCOP C++` library [17] developed by the first author: tabu search, simulated annealing (and a Metropolis variant), and `ID(best)` [18] which is a simple variant of `ID Walk` with only one parameter[4]. The simulated annealing has been discarded because it yields the worst performance. Tabu search, `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` [18] 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): with tabu search (when only a portion of the neighbors is visited), all the `MaxNeighbors` candidates are necessarily visited and the best among them (and not the first one) is returned.

[4] `ID Walk` with two parameters has an additional parameter `SpareNeighbor` that can take any value between 1 (fixed in `ID(any)`) and `MaxNeighbors` (fixed in `ID(best)`).

Our algorithm works with any of the standard greedy heuristics: BF or BLF, considering one among the three possible orders among rectangles: largest Width first (w), largest Height first (h), largest Surface first (s), providing six possible combinations: `BFw`, `BFh`, `BFs`, `BLFw`, `BLFh`, `BLFs`. Overall, the proposed metaheuristic works as follows:

1. the greedy heuristic first places all the rectangles on the strip,
2. `ID(best)`, driven by the the automatic tuning procedure, repairs the first solution.

A promising variant of this metaheuristic has been designed for which the choice of heuristic is directly incorporated into the neighborhood: in addition to the choice of rectangle R to be placed elsewhere (picked on the top of the layout) and to the location in which the rectangle R will be moved, one adds a choice among the greedy heuristics. In other words, every trial starts with a randomly chosen heuristic for providing the first layout; then, for every visited neighbor, one uses a randomly chosen greedy heuristic to place again on the strip the removed rectangles.

This variant presents two advantages. First, it avoids the user to choose among the (six) available greedy heuristics. More precisely, used with `ID(best)` and its automatically tuned parameter, this paper proposes a metaheuristic with *no* parameter. Second, the variant seems rather efficient because, 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 (bottom-left) BF greedy heuristic is to place a lot of rectangles "vertically" on the right side of the strip...

A description of the automatic tuning procedure can be found in [18]. 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 arbitrarily set to the half of the number of rectangles to be positioned. For the strip-packing variant allowing the rotation of rectangles, the value is equal to the number of rectangles. Although simple, this procedure is robust (i.e., the tuned value converges and produces a good configuration) in a majority of trials.

Class	Width	N	Opt.	IDW-1000	IDW-2*1000	IDW-100	Iori	Lesh	Burke	Bortfeldt	GRASP
C1	20	16–17	20	0.00	0.00	0.00	1.59	–	0.00	1.59	0.00
C2	40	25	15	0.00	0.00	0.00	2.08	–	6.25	2.08	0.00
C3	60	28–29	30	1.08	1.08	2.15	2.15	–	3.23	3.23	1.08
C4	60	49	60	1.64	1.64	1.64	4.75	–	1.64	2.70	1.64
C5	60	73	90	1.10	1.46	1.81	3.92	2.17	1.46	1.46	1.10
C6	80	97	120	1.37	1.37	1.37	4.00	1.64	1.37	1.64	0.83
C7	160	196.3	240	1.64	1.90	1.77	–	–	1.77	1.23	1.23
# optimal solutions				8/21	8/21	7/21	5/18	0/6	3/21	4/21	8/21

Table 2. Comparison on Hopper and Turton instances.

5 Experiments

We have performed experiments on five series enclosing 547 benchmarks. The 21 zero-waste instances by Hopper and Turton [11] are classified into 7 classes of increasing strip width. The corresponding results are reported in Table 2. Table 3 shows the results obtained on the 13 *gcut* instances by Beasley [3], the 3 *cgcut* instances, and the 10 *beng* instances by Bengtsson [4]. The results obtained on the 12 *ngcut* instances also proposed by Beasley are not reported because they are all optimaly 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.

Table 4 includes the results obtained on the 500 instances proposed by Martello and Vigo [16], Berkey and Wang [5]. 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 5 reports the results obtained on the Hopper and Turton instances [11] 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 [12]. The BLD* algorithm is run by Lesh et al on a Pentium at 2 Ghz [14]. 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 [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 a given category of benchmarks, our metaheuristic follows one or both of the following policies. A first approach runs `ID(best)` with the two most promising greedy heuristic, 10 runs for both (the two selected heuristics are specified in tables): an instance is thus solved 20 times, with an average time per run of 100 (sometimes 1000) seconds on a Pentium IV at 2.66 Ghz. A second approach runs the variant of our metaheuristic with no parameter: an instance is solved only 10 times in the same conditions. Thus, we allow a total time of 2000 or 1000 (sometimes 20000 or 10000) seconds depending on the selected protocol.

For all the heuristics, we report the best bound obtained. The average bounds are not reported because they are not always available in the literature and are indeed sometimes meaningless for certain algorithms (including our metaheuristic with two different greedy heuristics). However, the comparison based on the average time leads to similar conclusions.

Table 2

Every class 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. The columns `IDW-1000` and `IDW-2*1000` report the results of our metaheuristic in resp. 1000 seconds per run (10 runs using the variant with variable greedy heuristic) and 2000 seconds per run (10 runs with BFs and 10 runs with BLFw). The third column includes the results obtained by the automatic ID Walk variant in only 100 seconds per run. Note that a manual tuning of `ID Walk` in 1000 seconds per run allows us to reach an average percentage deviation from optimum of 1.37 for the class 7.

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 3

The column LB yields Lower Bound computations of the optimums (which are not necessarily reached). The cells include the bound of the best solution computed by the cor-

Instance	Width	N	LB	IDWalk	Variant	Iori	Lesh 60	Lesh 3600	GRASP
beng01	25	20	30	30	30	31	–	–	30
beng02	25	40	57	57	57	58	–	–	57
beng03	25	60	84	84	84	86	–	–	84
beng04	25	80	107	108	108	110	–	–	107
beng05	25	100	134	134	134	136	–	–	134
beng06	40	40	36	36	36	37	–	–	36
beng07	40	80	67	68	68	69	–	–	67
beng08	40	120	101	101	101	–	–	–	101
beng09	40	160	126	126	126	–	–	–	126
beng10	40	200	156	156	156	–	–	–	156
cgcut01	10	16	23	23	23	23	–	–	23
cgcut02	70	23	63	65	65	65	–	–	65
cgcut03	70	62	636	669	669	676	–	–	661
gcut01	250	10	1016	1016	1016	1016	1016	1016	1016
gcut02	250	20	1133	1204	1205	1207	1211	1195	1191
gcut03	250	30	1803	1803	1803	1803	1803	1803	1803
gcut04	250	50	2934	3066	3022	3130	3072	3054	3002
gcut05	500	10	1172	1273	1273	1273	1273	1273	1273
gcut06	500	20	2514	2656	2665	2675	2682	2656	2627
gcut07	500	30	4641	4694	4694	4758	4795	4754	4693
gcut08	500	50	5703	6192	6191	6240	6181	6081	5908
gcut09	1000	10	2022	2317	2317	–	–	–	2256
gcut10	1000	20	5356	**5973**	**5979**	–	–	–	6393
gcut11	1000	30	6537	**7037**	**6997**	–	–	–	7736
gcut12	1000	50	12522	14690	14690	–	–	–	13172
gcut13	3000	32	4772	**4962**	**4995**	–	–	–	5009

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

responding algorithm. We report the bound of the best solution obtained by Lesh et al's algorithm in resp. 60 and 3600 seconds. Two columns correspond to the results of IDWalk. The first one (IDWalk) corresponds to 10 runs of 100 s each with BLFw plus 10 runs of 100 s each with BFw (i.e., a total amount of 2000 seconds). The second column (Variant) corresponds to the variant with variable greedy heuristic in 1000 seconds (10 runs of 100 seconds each). On the three presented categories, we can also note that GRASP is better than ID Walk which is itself better than Iori's algorithm. Note however that ID Walk outperforms GRASP on three *gcut* instances with a wide strip (in bold). ID Walk (even in 1000 seconds) gives similar results as Lesh's approach in 3600 seconds. ID Walk finds the optimal solution of *beng* instances, except for *beng04* and *beng07*.

Table 4

The cells include the average percentage deviation from the (not necessarily reached) lower bound. ID Walk is run 20 times per instance, with both greedy heuristics (10+10) specified in the column *Greedy*. This means that one percentage in a cell comes from 1000 trials (20 trials on 50 instances in the class). One trial has a time limit of 100 seconds. From best to worst, the order between competitors is GRASP, ID Walk, Bortfeldt's algorithm, Lesh's algorithm, Iori's algorithm.

Table 5

IDW-1000 and IDW-100 correspond to the automatic variant of ID Walk in resp. 1000 seconds and 100 seconds per run. IDW-2*1000 is launched with BLFw and BFs (1000 seconds per run). Note that a ID Walk can find the 3 optimums 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. The first results were reported by Hopper and Turton themselves [10, 11] in 2000.

On the strip-packing variant with non-fixed orientation of rectangles, Bortfeldt's algorithm behaves very well, except on small instances[5]. The approach by Hopper and Turton is not competitive with our metaheuristic.

Synthesis

Lesh's algorithm behaves well but does not improve its solution a lot when spending more time (e.g., from 60 s to 3600 s). This highlights the interest of a metaheuristic, based on the geometry of the layout, for escaping from local minima.

The automatic variant of ID Walk behaves well. Indeed, it is generally better than the basic algorithm (requiring the user to select one or two given greedy heuristics) in the half laps of time. (The comparison is not easy on *gcut* instances.)

Overall, on strip-packing with rectangles of fixed orientation, ID Walk is worse than GRASP, but generally outperforms the others (including Bortfeldt's algorithm). On the variant with non-fixed orientation of rectangles, ID Walk behaves well while outperformed by Bortfeldt's algorithm on large instances.

[5] In our understanding, it may be due to its last postprocessing performed when handling non guillotine instances...

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

Table 4. Comparison on the 500 instances proposed by Martello, Vigo, Berkey, Wang [16, 5].

Class	Wid.	N	Opt.	IDW			H	Bortf.
				1000	2000	100		
C1	20	16–17	20	0.00	0.00	0.00	4	1.70
C2	40	25	15	0.00	0.00	0.00	6	0.00
C3	60	28–29	30	2.22	2.22	3.33	5	2.22
C4	60	49	60	1.67	1.67	1.67	3	0.00
C5	60	73	90	1.11	1.11	1.11	3	0.00
C6	80	97	120	1.11	1.11	1.67	3	0.33
C7	160	196.3	240	1.25	1.53	1.67	4	0.33
# optimal solutions				7(10)/21	7/21	6/21	0/21	15/21

Table 5. Comparison on Hopper and Turton instances with non-fixed orientation of rectangles [11].

6 Discussion

The contribution described in this paper is twofold. First, we have proposed incremental operators to maintain a set of maximal holes during the addition and removal of rectangles on 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, we have proposed a variant with no parameter to be (manually) tuned and with no greedy heuristic to be specified. This variant behaves well on the tested benchmarks.

The good performance obtained by GRASP, by Bortfeldt's algorithm and by our metaheuristic yield an experimental evidence that the best methods for handling strip-packing exploit the geometry of the layout.

However, we have not yet demonstrated that it is really interesting in practice to propose incremental moves allowing the violation of the BL property. These doubts are based on the rather good results obtained by Lesh [14], and on the excellent results obtained by the GRASP approach (for the moment limited in its implementation to rectangles with fixed orientation). These approaches are not really incremental but redo all the job (or at least a large part of it) when they perform rectangle removals.

Thus, could our metaheuristic be improved by using a more sophisticated greedy heuristic, such as the BF-like one used by the GRASP heuristic?

References

[1] R. Alvarez-Valdes, F. Parreño, and J. Tamarit. Reactive grasp for the strip packing problem. In *Proceedings Metaheuristic Conference MIC*, 2005.

[2] B. Baker, E. Coffman, and R. Rivest. Orthogonal packings in 2D. *SIAM Journal on Computing*, 9:846–855, 1980.

[3] J. Beasley. Algorithms for unconstrained two-dimensional guillotine cutting. *J. of the operational research society*, 33:49–64, 1985.

[4] B. Bengtsson. Packing rectangular pieces – a heuristic approach. *The computer journal*, 25:353–357, 1982.

[5] J. Berkey and P. Wang. Two-dimensional finite bin packing algorithms. *J. of the oper. resear. society*, 38:423–429, 1987.

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

[7] E. Burke, G. Kendall, and G. Whitwell. A new placement heuristic for the orthogonal stock cutting problem. *Operations Research*, 52:697–707, 2004.

[8] E. Burke, G. Kendall, and G. Whitwell. Metaheuristic enhancements of the best-fit heuristic for the orthogonal stock cutting problem. *submitted in INFORMS*, : , 2006.

[9] B. Chazelle. The bottom left bin packing heuristic: an efficient implementation. *IEEE Transactions on Computers*, 32:697–707, 1983.

[10] E. Hopper. *Two-Dimensional Packing Utilising Evolutionary Algorithms and other Meta-Heuristic Methods*. PhD. Thesis Cardiff University, 2000.

[11] E. Hopper and B. Turton. An empirical investigation on metaheuristic and heuristic algorithms for a 2d packing problem. *European J. of Operational Research*, 128:34–57, 2001.

[12] M. Iori, S. Martello, and M. Monaci. *Metaheuristic algorithms for the strip packing problem*, pages 159–179. Kluwer Academic Publishers, 2003.

[13] N. Lesh, J. Marks, A. M. Mahon, and M. Mitzenmacher. Exhaustive approaches to 2D rectangular perfect packings. *Information Processing Letters*, 90:7–14, 2004.

[14] N. Lesh, J. Marks, A. M. Mahon, and M. Mitzenmacher. New heuristic and interactive approaches to 2D strip packing. *ACM J. of Experimental Algorithmics*, 10:1–18, 2005.

[15] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *INFORMS Journal of Computing*, 15:310–319, 2003.

[16] S. Martello and D. Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management science*, 15:310–319, 1998.

[17] B. Neveu and G. Trombettoni. INCOP: An Open Library for INcomplete Combinatorial OPtimization. In *Proc. Constraint Programming, LNCS 2833*, pages 909–913, 2003.

[18] B. Neveu, G. Trombettoni, and F. Glover. ID Walk: A Candidate List Strategy with a Simple Diversification Device. In *Proc. Constraint Prog., LNCS 3258*, pages 423–437, 2004.