A constructive bin-oriented heuristic for the two-dimensional bin packing problem with guillotine cuts

Christoforos Charalambous[a,*], Krzysztof Fleszar[b]

[a]*Department of Computer Science and Engineering, Frederick University, P.O. Box 24729, 1303 Nicosia, Cyprus*
[b]*Olayan School of Business, American University of Beirut (AUB), P.O. Box 11–0236, Riad El Solh, Beirut 1107 2020, Lebanon*

Abstract

A new heuristic algorithm for solving the two-dimensional bin packing problem with guillotine cuts (2DBP|*|G) is presented. The heuristic constructs a solution by packing a bin at a time. Central to the adopted solution scheme is the principle of average-area sufficiency proposed by the authors for guiding selection of items to fill a bin. The algorithm is tested on a set of standard benchmark problem instances and compared with existing heuristics producing the best-known results. The results presented attest to the efficacy of the proposed scheme.

Keywords: two-dimensional bin packing, constructive heuristic, guillotine cut

1. Introduction

In several industrial applications, two-dimensional objects (items) must be arranged without overlapping into larger objects (bins), with the overall objective of minimizing total waste. Examples include the metal, glass, and wood industries, where a set of items demanded by customer orders must be cut from larger sheets or boards. Similarly, a common application in the transportation industry is the packing of rectangular items of fixed height into predefined spaces.

Assuming both items and bins to be rectangular, these problems can be modeled as two-dimensional bin packing problems. Additional constraints lead to various variants. The most common [1] arise out of the four combinations of orientation and type of cuts: items may have to be packed in a given orientation or may be rotated 90 degrees, and items may be required to be obtained through a sequence of horizontal or vertical edge-to-edge cuts (guillotine cuts) or may not be. In industrial settings, further constraints may be imposed. For example, in

*Corresponding author

the glass-cutting industry, which forms the background against which the work described in this paper was carried out, a minimum distance between parallel cuts may be additionally imposed or the available bins (glass sheets) may have different sizes and each size may have a different cost and a different limit on availability.

This paper addresses two variants of the problem: both require guillotine cuts, but in one variant item orientation is fixed (2DBP|O|G), while in the other items can be rotated (2DBP|R|G). Section 2 provides a review of the relevant literature. Section 3 presents a new constructive bin-oriented heuristic for addressing the two variants of the problem under consideration. Section 4 presents the results of extensive computational experiments on a large number of standard benchmark problem instances, in which various components of the heuristic are evaluated. The heuristic's performance is compared with the best performing heuristics and metaheuristics on the given benchmark problem instances available in the literature. Finally, section 5 offers concluding remarks.

2. Literature Review

The two-dimensional bin packing (2DBP) problem belongs to the general class of cutting and packing problems, a typology for which was proposed by Dyckhoff [6] and, more recently, by Schumann, Wäscher, and Haussner [12]. Cutting and packing problems, as a whole, have received considerable attention from the academic community in the past decades. A significant part of this literature has been devoted to the 2DBP problem. Surveys on 2DBP are provided by Lodi, Martello, and Vigo [7, 2], who describe lower bounds and solution methods, both exact and heuristic, available up to 2002. In the following, we briefly review the works published since 2002 on 2DBP with guillotine cuts (2DBP|*|G), which is the subject of this paper. We also review those earlier works that describe heuristics with which we compare our proposed methods.

A common two-dimensional packing strategy is to obtain a feasible solution by packing the items in rows forming shelves, with the first shelf of a bin placed on the bottom of the bin, and each following shelf placed on top of the previous shelf. It is also common to adopt a two-phase approach. In the first phase, a two-dimensional strip packing problem is solved, in which shelves are constructed and stacked in a strip of unlimited height, such that the total used height of the strip is minimized. In the second phase, shelves are repacked into bins by solving a one-dimensional bin-packing problem.

Following this strategy, Berkey and Wang [4] developed two effective heuristics, the Finite First Fit and the Finite Best Strip. However, these were later dominated by two algorithms developed by Lodi, Martello and Vigo [1]. The first of these is the Floor-Ceiling (FC) algorithm, which also packs items into shelves, but instead of placing items solely on the floor (from left to right), items are also placed on the ceiling (from right to left). The second is the KP algorithm, which creates shelves by solving a series of instances of the 0–1 knapsack problem. At each iteration, the new shelf is initialized with the tallest unpacked item, and is then completed by solving an instance of the 0–1 knapsack problem, with each item having a profit equal to its area and a cost equal to its width, and the knapsack capacity equal to the width of the strip minus the width of the initial item. Lodi, Martello, and Vigo presented versions of both FC and KP that are capable of solving the problem with and without rotation as well as with and without the guillotine-cut restriction. They also proposed an effective tabu-search scheme based on dynamic neighborhood size and structure.

Recently, Polyakovsky and M'Hallah proposed the Guillotine Bottom Left (GBL) constructive heuristic [10]. GBL packs one bin at a time. The largest unpacked item is positioned at the bottom left position of the bin. Subsequently, either a vertical or a horizontal guillotine cut is imposed on the packed item: the sizes of the two free rectangles that a cut would generate are computed, and the cut that yields the rectangle with the largest area is chosen. The same procedure is then applied recursively on the free rectangles that have been defined by it. The process continues until no item can be further packed in the bin.

In the same work, a complex Agent-Based (A-B) algorithm was also proposed. Based on distributed artificial intelligence, A-B mimics agent-based economic systems. It dynamically creates agents that communicate among themselves in a (pseudo) parallel fashion. Each agent assimilates the items to be packed and maintains its own characteristics (fitness, decision process) and employs the GBL algorithm to construct pieces of the solution. Through communication, agents can improve the feasible solutions obtained or can diversify the search through different item orderings. The results of the activity of the agents are disseminated to the system and the process is reiterated until the system reaches a steady state, from which a local optimum solution can be generated.

A recent work by Puchinger and Raidl [11] addressed a problem similar to the one being tackled here; that of three-stage two-dimensional bin packing. In that problem, it is required to have at most three levels of cuts in each bin: the first horizontal (when the bin is cut by one or more horizontal cuts), the second vertical (when each resulting piece can additionally be cut by any number of vertical cuts), and the third again horizontal (when each resulting piece can additionally be cut by any number of horizontal cuts). The authors presented integer linear programming models for a restricted version of the problem and solved them using CPLEX. They also offered a branch-and-price algorithm for a set covering formulation of the unrestricted problem, with columns generated by a variety of means. The same rescricted version of the problem was recently addressed by Alvelos et al. [3] who propose a greedy best-fit heuristic, using various criteria for evaluating fitting and ordering items. The initial solution was followed by local search heuristics and, subsequently, by a Variable Neighborhood Decent method.

3. The Constructive Heuristic

Our new constructive heuristic, shown in Algorithm 1, constructs solutions of the 2DBP|*|G in a bin-oriented fashion: while unpacked items exist, open a new bin, construct a pattern using a subset of the unpacked items, and place the pattern in the bin. A *pattern*, P, is a feasible arrangement of items that fits in a bin. In our algorithm, a pattern is represented by the set of items and their coordinates. Each pattern is constructed using procedure genPattern, which is described later.

```
Input: $(I, H, W)$; $I$ — set of items with dimensions $(h_i, w_i)$ for $i \in I$,
       $(H, W)$ — bin dimensions
Output: $X$ — a solution represented by a list of patterns
$X \leftarrow$ empty list of patterns;
while $I \neq \emptyset$ do
    $P \leftarrow$ empty pattern;
    $(P, I) \leftarrow$ genPattern$(P, I, [(0, 0, H, W)])$;
    Add $P$ to $X$;
end
return $X$;
```

Algorithm 1: The constructive heuristic

Throughout the algorithm, unpacked items are kept in a set, I, which is updated every time items are added to a pattern. To allow for item rotation, two copies of each item are stored in I, one for each orientation, and an availability vector is maintained to disallow double packing.

Since our constructive heuristic depends on the orientation of the problem, it is run twice, once with original bins and items, and once with all bins and all items rotated 90 degrees, and the better of the two solutions is returned. The second run is omitted if the problem considered is with rotation and the bins are square.

3.1. Comparing Patterns

During pattern construction, several partial patterns are considered and the best of them is committed. A key component of the new heuristic is the criterion for comparing patterns. A natural criterion is one that chooses the pattern with maximum total area of packed items. Unfortunately, such a criterion may favor selecting patterns that use many small (easy to pack) items. As a result, many small items can be used in the early stages of the construction of the solution, leaving larger, more difficult to pack, items for later stages.

To guard against excessive use of small items in early stages of solution construction, this work proposes to use an average-area sufficiency criterion in addition to the objective of maximizing the total area of packed items. Let $\bar{a}(I)$ be the average area of items in I. A pattern, P, having an average area $\bar{a}(P)$ over its items is said to meet the sufficiency criterion if $\bar{a}(P) \geq \bar{a}(I)$. In other words, the average-area sufficiency criterion specifies a desired lower limit on the average area of items included in a pattern.

When two patterns are compared, the choice is performed as follows:

- If both patterns satisfy the sufficiency criterion, then the one with the larger total area of items is chosen.

- If only one pattern satisfies the sufficiency criterion, then this pattern is chosen (total item area is ignored).

- If neither pattern satisfies the sufficiency criterion, then the one violating the sufficiency criterion least is chosen (total item area is ignored again).

Application of the sufficiency criterion in combination with the total item area maximization seeks to have tight packings, and, therefore, high area-utilization, while preferring larger items over smaller items, providing a look-ahead capability. This strategy is dynamic: when large items are packed, the average area of the remaining items decreases; thus increasing the possibility of using smaller items in subsequent bins.

In the following sections, the proposed comparison method is represented by the function fitsbetter(P_1, P_2, I) that returns true if pattern P_1 is preferred over pattern P_2 in the presence of unpacked items, I, and false otherwise.

3.2. Generating Simple Patterns

Our heuristic constructs patterns for bins incrementally, by inserting a *simple pattern* into a free (unoccupied) rectangle within the current pattern, P, until no insertion is possible. A *simple pattern* is a subset of unpacked items arranged side-by-side such that the bottom edges of all items form a single segment.

Simple patterns are generated using procedure genSimplePattern, shown in Algorithm 2. The procedure accepts as input: the current partial pattern, P, to which the new simple pattern will be added; the set of unpacked items, I; and the dimensions, (H, W), of the free rectangle within P, for which a simple pattern is to be constructed. The procedure returns the constructed simple pattern, S^*.

Procedure genSimplePattern begins by creating a reduced set of unpacked items, I', which includes only items that are small enough to fit in the free rectangle, (H, W). Then, several candidate simple patterns are considered. Each of them is generated by sorting I' and adding items in a first-fit manner until no more items fit. The sorting is based on a combination of two natural preferences, one for larger-area items, and the other for taller items. To this end, items are sorted by the nonincreasing values of $\lambda h'_i + (1 - \lambda)a'_i$, where h'_i and a'_i are, respectively, the height and area of item i, each normalized over the respective maximum from the unpacked items, I', and λ is a weighting parameter ranging from 0 to 1. When $\lambda = 0$, the items are sorted by nonincreasing area; and when $\lambda = 1$, by nonincreasing height. By varying λ, several orders of unpacked items are generated and, hence, several candidate simple patterns are constructed in the first-fit manner. The best among these is chosen using the fitsBetter function, described in Section 3.1. In order to ensure appropriate behavior of the sufficiency criterion (see Section 3.1), comparison is performed on simple patterns combined with the currently committed pattern, P, considering the union of I and P as the set of unpacked items.

```
Input: $(P, I, H, W)$; $P$ — pattern to which the simple pattern will be added,
       $I$ — set of unpacked items (excluding $P$), $(H, W)$ — dimensions of the
       free rectangle
Output: $S^*$ — the best simple pattern represented by a sequence of items
$I' \leftarrow \{i \in I : h_i \leq H \wedge w_i \leq W\}$;
$S^* \leftarrow$ empty sequence;
foreach $\lambda \in \Lambda$ do
    $L \leftarrow$ sort $I'$ depending on $\lambda$ (see the description);
    $W_{\text{left}} \leftarrow W$;
    $S \leftarrow$ empty sequence;
    foreach $i \in L$ do
        if $w_i \leq W_{left}$ then
            Add $i$ at the end of $S$;
            $W_{\text{left}} \leftarrow W_{\text{left}} - w_i$;
        end
    end
    if $fitsBetter(P \cup S, P \cup S^*, I \cup P)$ then $S^* \leftarrow S$;
end
return $S^*$;
```

Algorithm 2: Procedure genSimplePattern

3.3. Generating Patterns

Algorithm 3 shows procedure genPattern, which is used to generate patterns for all bins. It accepts as input: the current partial pattern, P; the set of unpacked items, I; and the list of free overlapping rectangles, R, in which items can be packed. Each rectangle in R is represented by (x, y, h, w), where x and y are the coordinates of the left-bottom corner of the rectangle, and h and w are the dimensions of the rectangle. The rectangles are always sorted by increasing x. The procedure returns the updated pattern, P, and the reduced set of unpacked items, I.

The procedure starts by removing from R all free rectangles that are too small to accommodate any of the unpacked items. If R is not empty, the following steps are performed. For each rectangle in R, a simple pattern is generated using genSimplePattern (see Algorithm 2). All simple patterns are compared using function fitsBetter (see Section 3.1) and the best is saved as S^* with the index of the corresponding free rectangle saved as j^*. Then, items of S^* are sorted by nonincreasing height and added to the pattern, P, by placing them left-to-right on the bottom edge of rectangle $R[j^*]$. Items of S^* are also removed from the unpacked items, I. Subsequently, three lists of rectangles are generated: R_{left} with all rectangles in R before index j^*, R_{below} with all rectangles in R after index j^*, and R_{sub} with all largest free overlapping rectangles remaining within $R[j^*]$ after packing S^* in it. Rectangles in R_{left} and R_{below} are trimmed from right and above, respectively, to those parts that are not overlapping with $R[j^*]$. Finally, procedure genPattern is invoked recursively three times, first for R_{sub}, then for R_{below} (with increased heights, described later), and last for R_{left} (with increased widths, also described later).

```
Input: $(P, I, R)$; $P$ — pattern, $I$ — set of unpacked items (excluding $P$),
          $R$ — list of free overlapping rectangles
Output: $(P, I)$; $P$ — updated pattern, $I$ — reduced set of unpacked items
Remove from $R$ all rectangles too small for any of the items in $I$;
if $R = \emptyset$ then return $(P, I)$;
$(S^*, j^*) \leftarrow (\emptyset, 0)$;
for $j \leftarrow 1$ to $|R|$ do
    $S \leftarrow$ genSimplePattern$(P, I, R[j].h, R[j].w)$;
    if $S^* = \emptyset$ or fitsBetter$(P \cup S, P \cup S^*, I \cup P)$ then $(S^*, j^*) \leftarrow (S, j)$;
end
Sort items in $S^*$ by height nonincreasing;
Insert simple pattern $S^*$ into pattern $P$ in rectangle $R[j^*]$;
$I \leftarrow I \setminus S^*$;
$R_{\text{left}} \leftarrow [R[1], \ldots, R[j^* - 1]]$ trimmed from right by $R[j^*]$;
$R_{\text{below}} \leftarrow [R[j^* + 1], \ldots, R[|R|]]$ trimmed from above by $R[j^*]$;
$R_{\text{sub}} \leftarrow$ free overlapping rectangles above items of $S^*$ just inserted into $P$;
$(P, I) \leftarrow$ genPattern$(P, I, R_{\text{sub}})$;
Increase heights of rectangles in $R_{\text{below}}$ (see the description);
$(P, I) \leftarrow$ genPattern$(P, I, R_{\text{below}})$;
Increase widths of rectangles in $R_{\text{left}}$ (see the description);
$(P, I) \leftarrow$ genPattern$(P, I, R_{\text{left}})$;
return $(P, I)$;
```
Algorithm 3: Procedure genPattern

Recall that our constructive heuristic invokes the genPattern procedure repeatedly (see Algorithm 1 on page 4), once for each new bin, each time with $P =$ empty pattern, $I =$ currently unpacked items, and $R = [(0, 0, H, W)]$; the latter being a list containing a single free rectangle with dimensions equal to those of the bin. Since in this call of genPattern there is only one free rectangle, a simple pattern is always committed on the bottom edge of this rectangle and R_{left} and R_{below} are always empty. If R_{sub} is not empty, recursive calls of genPattern occur, in which all three lists of rectangles may be non-empty.

Figure 1a shows an example partial solution with the first four-item simple pattern committed inside the initial call of genPattern. At this point, four free overlapping rectangles are generated and saved in R_{sub}, each defined by the corresponding item of the simple pattern, as shown in Figure 1a (R_{left} and R_{below} are empty). Procedure genPattern is now called recursively for $R = R_{\text{sub}}$. In this recursive call, four simple patterns, one for each of the four free overlapping rectangles, is generated. Due to overlapping, only one of those simple patterns can be committed. Assume that a three-item simple pattern was committed in the third free rectangle, as shown in Figure 1b. R_{left} will now contain the first two rectangles of R trimmed from the right, and R_{below} will contain the fourth rectangle of R trimmed from above, and R_{sub} will contain two rectangles above items 6 and 7, as shown in Figure 1b. For each of these three lists, genPattern will be called recursively.

Before genPattern invokes itself recursively for R_{below} and R_{left}, the dimen-

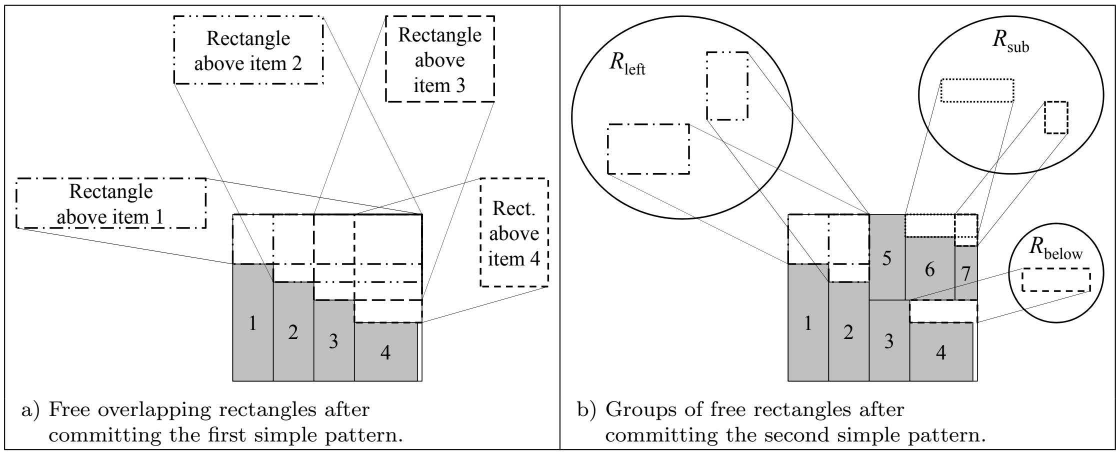
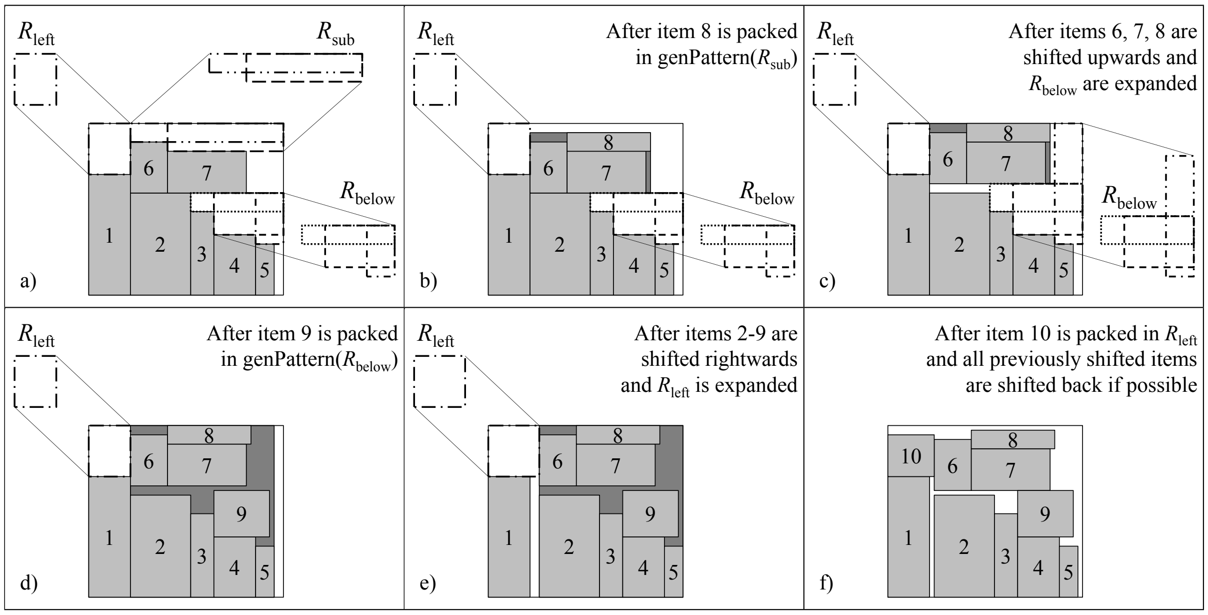

Rectangle above item 2
Rectangle above item 3
Rectangle above item 1
Rect. above item 4
1
2
3
4
a) Free overlapping rectangles after committing the first simple pattern.
R_{left}
R_{sub}
R_{below}
5
6
7
b) Groups of free rectangles after committing the second simple pattern.

Figure 1: Example 1.

sions of free rectangles in these lists are increased, if possible, in order to increase the likelihood of packing larger items. The procedure performs the following steps. Since increasing sizes of rectangles of R_{sub} is never possible, genPattern is invoked first for unmodified set R_{sub}. Once processing of R_{sub} is completed, all items packed above R_{below} (all items in $R[j^*]$) are shifted upwards as much as possible (all by the same amount), and the height of each rectangle in R_{below} is increased as much as possible, without violating guillotine cuts. Then, genPattern is invoked recursively for R_{below}. When processing R_{below} is completed, all items packed to the right of R_{left} (all items in $R[j^*]$ and below it) are shifted rightwards as much as possible (all by the same amount), and the width of each rectangle in R_{left} is increased accordingly, without violating guillotine cuts. Subsequently, genPattern is invoked recursively for R_{left}. Finally, when processing of R_{left} is completed, all items that were previously shifted upwards or rightwards are shifted back as much as possible, maintaining guillotine cuts, in order to allow for shifting of items after backtracking.

To illustrate how sizes of free rectangles are increased, consider the example presented in Figure 2. Figure 2a shows a stage in which a five-item simple pattern is committed in the initial call of genPattern, and a two-item simple pattern is committed in the recursive call. At this point, genPattern is invoked recursively three times. First, it is invoked for R_{sub}, which manages to commit only item 8, as shown in Figure 2b. (In general, the heuristic could pack many more items in R_{sub}.) Once processing R_{sub} is completed, sizes of rectangles in R_{below} are increased. Items 6, 7, and 8 (enveloped by a dark-gray rectangle in Figure 2b-c) are together shifted upwards and the height of each rectangle in R_{below} is increased, as shown in Figure 2c. Note that for the first two rectangles in R_{below}, the height is increased as much as 6, 7, and 8 were shifted upwards. However, for the third rectangle, the height is increased even more, as shown in Figure 2c. At this point, genPattern is invoked recursively for R_{below}, which



R_{left}
R_{sub}
R_{below}
After item 8 is packed in genPattern(R_{sub})
After items 6, 7, 8 are shifted upwards and R_{below} are expanded
After item 9 is packed in genPattern(R_{below})
After items 2-9 are shifted rightwards and R_{left} is expanded
After item 10 is packed in R_{left} and all previously shifted items are shifted back if possible
a)
b)
c)
d)
e)
f)
1
2
3
4
5
6
7
8
9
10

Figure 2: Example 2.

commits only item 9, as shown in Figure 2d. (Again, many more items could be packed in R_{below}, in general.) When processing R_{below} is completed, sizes of rectangles in R_{left} are increased. All items right of both R_{left} and item 1, i.e. items 2–9 (enveloped by a dark-gray rectangle in Figure 2d-e), are shifted together rightwards and the width of each rectangle in R_{left} is increased accordingly, as shown in Figure 2e. At this point, genPattern is invoked recursively for R_{left}, which commits item 10. Finally, after processing R_{left} is completed, all items previously shifted rightwards and upwards are shifted back as much as possible, as show in Figure 2f.

3.4. Time Complexity

Since items committed in a pattern are never reconsidered by the heuristic, the time complexity of the heuristic is modest. In each call of genPattern, at most n free rectangles are considered, where n is the number of items, and for each rectangle the complexity of generating a simple pattern is bounded by the sorting with $O(n \log n)$ complexity. Clearly, the repetition of the process several times for several values of λ imposes only a constant coefficient on the execution, leaving the complexity unaffected. Since in each call of genPattern at least one item is committed, the number of recursive calls over all bins is limited to n. Thus, the worst case complexity of our constructive heuristic is $O(n^3 \log n)$. Note that the average case should be much better, as the number of free rectangles is often much less than n, the sorted set is continually reduced, and simple patterns that are committed often contain more than one item. Experimental results attest to this, as the algorithm requires only a few milliseconds even for the largest problem instances.

3.5. Sufficiency Bias

Experimentation has shown that application of the sufficiency criterion along with covered area maximization (as described in Section 3.1) yields considerable improvements of solutions, compared with relying solely on the latter. However, it also became evident that relaxing or restricting the adopted sufficiency limit may be advantageous for certain problem instances. This may be the case when there is a relatively large number of large items that need to be packed together early on. To obtain a good solution in such instances, a pattern comprising large items may be preferred to another that uses smaller items, even if both respect the sufficiency criterion. Similarly, some instances require very tight patterns to reach better solutions, even if the sufficiency criterion is moderately violated.

To cater for such instances, a bias strengthening or weakening the sufficiency criterion may be introduced. The algorithm is first executed without bias. If the optimal solution is not obtained (lower bound is not reached), a bias is applied to the sufficiency limit and the algorithm is rerun. When a bias of α is applied, the lower limit on average area is multiplied by $(1 + \alpha)$. Reruns are performed for several values of α.

3.6. Postoptimisation

The stage in the constructive process in which an item is committed to a bin depends to a great extend on its dimensions. Items of large area are often addressed early on as they contribute towards meeting the sufficiency criterion. The same holds for elongated items (items with one large and one small dimension) as they can easily be added to develop efficient simple patterns or utilize short vertical spaces without having considerable impact on the sufficiency of the pattern. Also, small items (both dimensions relatively small) are packed at various stages in the process as the algorithm always enforces maximal packs. However, it is often the case that medium-sized square-like items (height and width close to each other) have a tendency to be left towards the latter stages of the constructive process. Through experimentation, it was observed that forcing the packing of these items earlier in the constructive process may yield improved results.

From this observation stems the proposed postoptimization algorithm. The algorithm commences by running the constructive heuristic on the problem instance without bias and then several times with varying bias values, as described in Section 3.5. Throughout this process, the best obtained solution is stored. A solution is considered better if it uses fewer bins, breaking ties in favor of larger slack in the last bin. For each generated solution, bins are numbered (bin 1 being the first bin packed and so on), and for each item, the sum of the bin numbers in which it was packed, over all the solutions, is calculated. If the sum of bin numbers for an item is relatively high, then it can be concluded that this item consistently appears in the latter bins and, therefore, is avoided by the heuristic. Hence, developing solutions with such items packed early may improve results.

The postoptimization improves the incumbent solution as follows. Our constructive heuristic is used to build a new pattern for one bin, assuming all items are available, and forcing the most avoided item (the item with the largest sum of bin numbers) in the first simple pattern. The new bin with the new pattern is added to the solution. The new pattern will obviously use items that were already allocated to other bins. To fix the solution, all affected bins are unpacked. To increase the possibility of generating efficient bins, a number of bins filled last in the solution process is also unpacked, even if unaffected. Any unpacked items are then packed by our constructive heuristic. If the new solution generated in this manner improves the incumbent, then it is adopted. The process is repeated, initializing the first bin with the second most avoided item, and so on. Clearly, applying the postoptimization routine with items selected early in the constructive algorithm would lead to solutions already developed as similar patterns would be constructed. Therefore, postoptimization is repeated for only a number of the most avoided items.

4. Computational Experiments

The proposed algorithms were tested on the 500 benchmark problem instances proposed by Berkey and Wang [4] (classes 1–6) and Lodi et al. [1] (classes 7–10). Each class includes 10 instances of size 20, 40, 60, 80, and 100 items, providing a total of 50 instances per class. In all problem classes square bins are used. Item dimensions are random integers from uniform distributions in various ranges. Classes differ in the bin size (ranging from 10x10 to 300x300) and the distribution ranges used for obtaining the item sizes.

All our heuristics were coded in C# and compiled using Microsoft Visual Studio 2008. All tests were performed on a PC with an Intel Core i3 M330 2.13GHz processor. Each heuristic was executed as a single process with a single thread and real computation times were recorded.

Lower bounds were used for early termination of heuristics as well as for calculating the deviations. For the oriented case, lower bounds were calculated as in the work of Martello and Vigo [9]. For the case with rotation, the tighter lower bounds of Clautiaux et al. [5] were used.

Inside the genSimplePattern procedure (see Section 3.2), six values of λ were used, namely $\Lambda = \{0.001, 0.2, 0.4, 0.6, 0.8, 0.999\}$. A value of 0.001 (and 0.999) was used instead of 0 (and 1) so that when the items are sorted by nonincreasing area (height), ties are broken in favor of larger height (area). For biasing (see Section 3.5), α varied from -0.4 to $+0.6$ in 0.2 steps. For postoptimization (see Section 3.6), the processing was repeated for the 20 most avoided items; each time unpacking the latter one fourth of the bins (rounded up to the nearest integer) and any bin affected by packing the first bin in the new solution.

Table 1 shows the results of all versions of the proposed heuristic for the two versions of the problem; one without rotation (2BP|O|G) and the other with

Table 1: Results of the various versions of the proposed heuristic.

Heuristic	2BP|O|G				2BP|R|G			
	Avg. Dev.	Total Bins	Avg. Time	Max. Time	Avg. Dev.	Total Bins	Avg. Time	Max. Time
CH without sufficiency	1.048	7403	0.005	0.022	1.071	7272	0.006	0.027
CH	1.045	7375	0.005	0.023	1.059	7191	0.006	0.028
CHB	1.041	7345	0.018	0.130	1.050	7117	0.018	0.129
CHBP	1.036	7311	0.033	0.347	1.037	7064	0.066	0.951

rotation (2BP|R|G). The quality of solutions is reported in terms of average deviations from lower bounds and the total number of bins used. Deviation for each instance is computed as the number of bins divided by the lower bound on the number of bins. Average and maximum processing times are reported in seconds.

The four heuristic versions reported in Table 1 are: the constructive heuristic (CH) without and then with the sufficiency criterion, CH followed by biasing the sufficiency limit (CHB) (see Section 3.5), and CHB followed by the postoptimization (CHBP) (see Section 3.6). The impact of using a sufficiency criterion on the construction of solutions is significant. For both versions of the problem, with and without rotation, adding the sufficiency criterion improves the results without increasing computation time. Both biasing sufficiency and postoptimization further improve the results of CH. The processing time increases but is still modest.

Table 2 compares the CH heuristic (with the sufficiency criterion), CHB and CHBP with the best methods available in the literature. CH is set against other fast heuristics: the Floor-Ceiling (FC) and Knapsack-Problem-based (KP) heuristics of Lodi et al. [1], and the Guillotine Bottom Left (GBL) heuristic of Polyakovsky [10]. The heuristics proposed by Berkey and Wang are not included as they are dominated in all cases by either the FC or KP. CHB and CHBP are set against metaheuristics: the Tabu Search (TS) of Lodi et al. [1], the Variable Neighborhood Decent (VND) method of Alvelos et al. [3] and, the Agent Based (A-B) approach of Polyakovsky [10]. It is worth recalling that VND is designed to solve the problem with constraints on the number of cutting stages. For all algorithms, the same performance measures as in Table 1 are reported, whenever available. Additionally, the processor used for testing each method is reported.

The published deviations of FC, KP, and TS for the problem with rotation were computed with respect to weaker lower bounds than those used in this paper; namely, those of Dell'Amico [8]. For comparability, the deviations for these algorithms were scaled by multiplying them by the ratio of the two lower bounds. Although this may cause slight rounding errors, the results should be quite accurate. For the case without rotation, the published deviations of FC, KP, and TS were used. For GBL and A-B, the reported results are based on the detailed results obtained from the authors. The results for VND were computed based on those published in [3].

Examining the results in Table 2, both the efficiency and effectiveness of the

Table 2: Comparison of CH and CHBP with the best-known heuristics and metaheuristics.

	2BP|O|G				2BP|R|G				
Heuristic	Avg. Dev.	Total Bins	Avg. Time	Max. Time	Avg. Dev.	Total Bins	Avg. Time	Max. Time	Processor
FC	1.087			<0.5	1.089			<0.5	Silicon Graphics*
KP	1.089	7480		<0.5	1.085	7297		<0.5	Silicon Graphics*
GBL	1.109	7636	∼0.03	0.07	1.094	7367	∼0.03	0.07	Athlon XP 2800
CH	1.045	7375	0.005	0.023	1.059	7191	0.006	0.028	Intel i3 2.13GHz
TS	1.079	7433	32.55		1.055	7101	29.28		Silicon Graphics*
VND	1.075		0.246						Intel Core 2 2.13GHz
A-B	1.043	7374		3.5	1.045	7130		3.5	Athlon XP 2800
CHB	1.041	7345	0.018	0.130	1.050	7117	0.018	0.129	Intel i3 2.13GHz
CHBP	1.036	7311	0.033	0.347	1.037	7064	0.066	0.951	Intel i3 2.13GHz

*Silicon Graphics INDY R10000sc 195MHz

proposed heuristics are evident, both in the oriented case and the non-oriented case. CH overwhelmingly outperforms FC, KP, and GBL in terms of average deviations and total bins used, while requiring similar computation times. In fact, for the oriented case, CH outperforms even the complex TS and VND metaheuristics and performs comparably with A-B, while requiring a fraction of their computation time.

When compared to TS and A-B, CHB performs comparably in the non-oriented case and better in the oriented case, in both requiring significantly less execution time, taking into account the different processors. When CHBP is compared with TS and A-B it is evident that it achieves significantly better results in terms of both average deviation and total bins used, again requiring less execution time. VND is significantly outperformed by both CHB and CHBP in terms of both solution quality and execution time.

Detailed results of the four fast heuristics, FC, KP, GBL, and our CH are provided in the Appendix in Table 3. In this table, average deviations are shown for each problem class and size group, for both the oriented and non-oriented case. The total number of bins used is also provided. Similar results for TS, A-B, and our CHBP are provided in Table 4. Examining the detailed results, the robustness of our algorithms is also evident. Comparing constructive heuristics for the oriented case, CH performs best in 49 of the 50 class/problem-size groups, whereas FC, KP and GBL meet the best deviation in 11, 9 and 7 cases, respectively. Similarly, for the case where item rotation is permitted, CH results in the best deviation in 44 out of the 50 groups whereas FC, KP, and GBL do so only in 19, 21, and 18 cases, respectively. The same robustness can be identified when comparing the three more advanced algorithms. For the oriented case, over the 50 problem groups, CHBP performs best in 47 cases, whereas TS and A-B do so for 9 and 22, respectively. Similarly, in the non-oriented case CHBP achieves the best deviation in 47 out of the 50 cases, compared with 20 and 23 cases for TS and A-B, respectively.

5. Conclusions

A new constructive bin-oriented heuristic (CH) for the two-dimensional bin packing problem with guillotine cuts was presented. The algorithm constructs solutions incrementally; at each stage adding a simple pattern to the previously committed items. The main strength of CH is in generating multiple simple patterns and in the way these patterns are compared to select the best. The comparison uses the concept of area-sufficiency in addition to classical total area covered maximization. The sufficiency criterion guards against overuse of easily packed items early on in the construction of the solution by giving preference to efficient patterns consisting of larger items. Moreover, the criterion is dynamic, as the limit on average item area is adjusted when the set of unpacked items is reduced.

Computational analysis shows that CH is very effective when compared to other fast methods available in the literature. The experiments also show that using the sufficiency criterion has a positive impact on the results. CH, which is a simple constructive heuristic, managed to outperform even the tabu-search heuristic of Lodi et al. [1] for the oriented case.

Improved results were obtained when the heuristic was enhanced with biasing the sufficiency condition (CHB). Further improvements were obtained when a postoptiomization routine was used (CHBP) that reapplied the algorithm seeding it with the most avoided items. The CHBP outperformed all published two-dimensional bin packing metaheuristics in terms of both solution quality and computation time.

A natural future research direction would be to examine whether a similar approach could achieve the same quality of solutions when used for the bin packing problem without the guillotine constraint. It would also be interesting to examine whether the proposed algorithms can be incorporated into a more sophisticated non-polynomial scheme that could attempt to solve the problem to optimality. Finally, it would be of great interest to evaluate the effectiveness of the sufficiency principle on other packing and related combinatorial problems.

Acknowledgements

The authors would like to thank Dr. Sergey Polyakovsky for providing detailed data on the performance of the GBL and A-B algorithms as well as the lower bound calculations for the 2DBP|R|* case.

Bibliography

[1] S. Martello A. Lodi and D Vigo. Heuristics and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems. *INFORMS Journal on Computing*, 11(4):345–357, 1999.

[2] S. Martello A. Lodi and D Vigo. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141:241–252, 2002.

[3] F. Alvelos, T. Chan, P. Vilaca, E. Silva, and J. M. Valerio de Carvalho. Sequence based heuristics for two-dimensional bin packing problems. *Engineering Optimization*, 41(8):773–791, 2009.

[4] J.O. Berkey and P.Y. Wang. Two-dimensional finite bin-packing algorithms. *Journal of the Operational Research Society*, 38(5):423–429, 1987.

[5] F. Clautiaux, A. Jouglet, and J. El Hayek. A new lower bound for the non-oriented two-dimensional bin-packing problem. *Operations Research Letters*, 35(3):365–373, 2007.

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

[7] A. Lodi, S. Martello, and D Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathimatics*, 123:379–296, 2002.

[8] D. Vigo M Dell Amico, S. Martello. A lower bound for the non-oriented two-dimensional bin packing problem. *Discrete Applied Mathematics*, 118:12–24, 2002.

[9] S. Martello and D. Vigo. Exact solution of the two-dimensional finite bin packing problem. *Management Science*, 44(3):388–399, 1998.

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

[11] J. Puchinger and G.R. Raidl. Models and algorithms for three-stage two-dimensional bin packing. *European Journal of Operational Research*, 183(3):1304–1327, 2007.

[12] G. Wäscher, H Haussner, and H Schumann. An Improved Typology of cutting and packing problems. *European Journal of Operational Research*, 183:1109–1130, 2007.

Appendix

This section contains the detailed results of our new heuristics, CH and CHBP, and other heuristics with which CH and CHBP were compared in Section 4. Results for all algorithms are presented in terms of deviations from lower bounds. Additionally, for the new heuristics proposed in this work the total number of bins is also provided in columns followed by 'bins'.

Table 3: Detailed results of CH and three other fast heuristics

Class	n	2BP|O|G					2BP|R|G				
		FC	KP	GBL	CH	CH-bins	FC	KP	GBL	CH	CH-bins
1	20	1.14	1.13	1.08	1.00	71	1.06	1.06	1.06	1.01	67
	40	1.09	1.10	1.08	1.02	136	1.06	1.05	1.06	1.02	131
	60	1.07	1.07	1.06	1.03	202	1.06	1.05	1.03	1.03	200
	80	1.06	1.06	1.04	1.01	277	1.07	1.06	1.02	1.02	276
	100	1.06	1.05	1.04	1.03	325	1.05	1.03	1.03	1.03	323
	All	1.084	1.082	1.058	1.016	1011	1.060	1.048	1.039	1.024	997
2	20	1.10	1.00	1.20	1.00	10	1.00	1.00	1.00	1.00	10
	40	1.10	1.10	1.10	1.10	20	1.10	1.10	1.10	1.10	20
	60	1.15	1.15	1.20	1.05	26	1.05	1.15	1.10	1.05	26
	80	1.07	1.07	1.10	1.07	33	1.03	1.07	1.07	1.03	32
	100	1.03	1.03	1.08	1.03	40	1.03	1.03	1.03	1.00	39
	All	1.090	1.070	1.137	1.050	129	1.042	1.070	1.060	1.037	127
3	20	1.18	1.18	1.21	1.08	54	1.16	1.10	1.19	1.04	49
	40	1.14	1.15	1.19	1.08	98	1.15	1.15	1.12	1.07	96
	60	1.11	1.12	1.14	1.06	144	1.16	1.09	1.13	1.05	138
	80	1.10	1.10	1.18	1.07	199	1.10	1.07	1.09	1.07	194
	100	1.09	1.09	1.11	1.06	233	1.12	1.09	1.09	1.06	228
	All	1.124	1.128	1.166	1.068	728	1.140	1.103	1.124	1.057	705
4	20	1.00	1.00	1.10	1.00	10	1.00	1.00	1.00	1.00	10
	40	1.00	1.10	1.10	1.00	19	1.00	1.00	1.00	1.00	19
	60	1.10	1.20	1.25	1.10	25	1.10	1.10	1.15	1.10	25
	80	1.10	1.13	1.17	1.10	33	1.10	1.10	1.10	1.10	33
	100	1.10	1.10	1.09	1.07	39	1.07	1.07	1.10	1.07	39
	All	1.060	1.106	1.142	1.053	126	1.054	1.054	1.070	1.053	126
5	20	1.14	1.13	1.09	1.00	65	1.04	1.04	1.06	1.00	59
	40	1.11	1.09	1.10	1.03	122	1.07	1.08	1.08	1.03	116
	60	1.10	1.10	1.09	1.04	185	1.08	1.08	1.07	1.05	180
	80	1.09	1.09	1.09	1.04	251	1.08	1.07	1.06	1.06	249
	100	1.09	1.09	1.09	1.05	292	1.09	1.08	1.07	1.07	290
	All	1.106	1.100	1.091	1.032	915	1.072	1.070	1.066	1.042	894
6	20	1.00	1.00	1.10	1.00	10	1.00	1.00	1.00	1.00	10
	40	1.40	1.50	1.40	1.40	19	1.40	1.40	1.40	1.40	19
	60	1.10	1.10	1.20	1.05	22	1.05	1.05	1.10	1.05	22
	80	1.00	1.00	1.03	1.00	30	1.00	1.00	1.00	1.00	30
	100	1.10	1.10	1.20	1.07	34	1.07	1.10	1.13	1.07	34
	All	1.120	1.140	1.187	1.103	115	1.104	1.110	1.127	1.103	115
7	20	1.10	1.10	1.14	1.02	56	1.19	1.17	1.22	1.13	53
	40	1.11	1.07	1.12	1.05	115	1.15	1.15	1.18	1.09	108
	60	1.08	1.06	1.10	1.04	163	1.18	1.16	1.18	1.10	154
	80	1.06	1.06	1.09	1.05	235	1.15	1.15	1.16	1.08	216
	100	1.04	1.04	1.07	1.03	277	1.17	1.16	1.17	1.09	261
	All	1.078	1.066	1.105	1.039	846	1.167	1.157	1.184	1.099	792
8	20	1.16	1.12	1.12	1.02	59	1.12	1.12	1.15	1.08	54
	40	1.08	1.07	1.09	1.03	116	1.18	1.18	1.19	1.09	106
	60	1.06	1.06	1.10	1.04	165	1.17	1.17	1.19	1.09	155
	80	1.06	1.05	1.08	1.02	227	1.16	1.15	1.16	1.09	214
	100	1.06	1.04	1.07	1.04	284	1.17	1.17	1.17	1.09	263
	All	1.084	1.068	1.091	1.030	851	1.160	1.159	1.170	1.088	792
9	20	1.01	1.01	1.01	1.01	144	1.00	1.00	1.00	1.01	144
	40	1.02	1.02	1.00	1.01	280	1.01	1.01	1.01	1.02	279
	60	1.02	1.01	1.00	1.00	439	1.01	1.01	1.00	1.01	438
	80	1.02	1.02	1.00	1.00	578	1.01	1.01	1.01	1.01	576
	100	1.01	1.01	1.00	1.00	696	1.01	1.01	1.00	1.00	694
	All	1.016	1.014	1.001	1.004	2137	1.008	1.008	1.005	1.007	2131
10	20	1.14	1.16	1.08	1.05	44	1.13	1.10	1.17	1.13	42
	40	1.09	1.10	1.08	1.02	75	1.08	1.08	1.09	1.08	75
	60	1.10	1.10	1.13	1.05	103	1.08	1.07	1.08	1.09	103
	80	1.12	1.12	1.15	1.07	132	1.06	1.06	1.07	1.07	131
	100	1.10	1.08	1.11	1.07	163	1.07	1.05	1.08	1.05	161
	All	1.110	1.112	1.110	1.051	517	1.083	1.071	1.097	1.083	512
Overall		1.087	1.089	1.109	1.045	7375	1.089	1.085	1.094	1.059	7191

Table 4: Detailed results of CHBP and two metaheuristics

Class	n	2BP|O|G				2BP|R|G			
		TS	A-B	CHBP	CHBP-bins	TS	A-B	CHBP	CHBP-bins
1	20	1.11	1.01	1.00	71	1.03	1.01	1.00	66
	40	1.08	1.02	1.00	134	1.03	1.02	1.01	129
	60	1.05	1.03	1.02	201	1.03	1.01	1.00	195
	80	1.04	1.00	1.00	275	1.05	1.00	1.00	271
	100	1.05	1.02	1.01	321	1.02	1.01	1.00	314
	All	1.066	1.017	1.008	1002	1.030	1.010	1.003	975
2	20	1.00	1.00	1.00	10	1.00	1.00	1.00	10
	40	1.10	1.10	1.10	20	1.10	1.00	1.00	19
	60	1.15	1.00	1.05	26	1.15	1.00	1.00	25
	80	1.07	1.00	1.07	33	1.03	1.00	1.00	31
	100	1.03	1.00	1.00	39	1.03	1.00	1.00	39
	All	1.070	1.020	1.043	128	1.062	1.000	1.000	124
3	20	1.18	1.05	1.03	52	1.07	1.04	1.02	48
	40	1.12	1.08	1.07	97	1.10	1.07	1.04	94
	60	1.07	1.07	1.03	140	1.07	1.05	1.03	136
	80	1.08	1.06	1.05	196	1.03	1.04	1.02	186
	100	1.09	1.05	1.04	230	1.07	1.05	1.04	223
	All	1.108	1.062	1.046	715	1.069	1.050	1.031	687
4	20	1.00	1.00	1.00	10	1.00	1.00	1.00	10
	40	1.10	1.00	1.00	19	1.00	1.00	1.00	19
	60	1.20	1.10	1.10	25	1.10	1.10	1.10	25
	80	1.10	1.10	1.10	33	1.03	1.03	1.10	33
	100	1.10	1.03	1.07	39	1.03	1.00	1.03	38
	All	1.100	1.047	1.053	126	1.032	1.027	1.047	125
5	20	1.13	1.02	1.00	65	1.00	1.02	1.00	59
	40	1.09	1.04	1.02	121	1.04	1.04	1.03	115
	60	1.07	1.04	1.02	183	1.04	1.05	1.03	176
	80	1.08	1.06	1.03	247	1.05	1.04	1.02	240
	100	1.09	1.06	1.04	288	1.06	1.06	1.04	282
	All	1.092	1.043	1.021	904	1.039	1.043	1.023	872
6	20	1.00	1.00	1.00	10	1.00	1.00	1.00	10
	40	1.50	1.40	1.40	19	1.40	1.30	1.30	18
	60	1.10	1.05	1.05	22	1.05	1.00	1.00	21
	80	1.00	1.00	1.00	30	1.00	1.00	1.00	30
	100	1.10	1.10	1.07	34	1.07	1.07	1.07	34
	All	1.140	1.110	1.103	115	1.104	1.073	1.073	113
7	20	1.08	1.04	1.00	55	1.11	1.13	1.11	52
	40	1.07	1.05	1.03	112	1.05	1.08	1.05	104
	60	1.05	1.04	1.02	160	1.06	1.09	1.05	148
	80	1.05	1.05	1.04	233	1.06	1.08	1.06	211
	100	1.04	1.03	1.02	275	1.07	1.08	1.07	255
	All	1.058	1.043	1.023	835	1.071	1.092	1.068	770
8	20	1.12	1.02	1.00	58	1.06	1.11	1.06	53
	40	1.04	1.03	1.02	114	1.07	1.09	1.08	105
	60	1.03	1.04	1.03	163	1.06	1.08	1.06	150
	80	1.03	1.03	1.01	226	1.08	1.09	1.07	210
	100	1.04	1.04	1.02	279	1.08	1.08	1.07	258
	All	1.052	1.032	1.015	840	1.071	1.090	1.066	776
9	20	1.00	1.00	1.00	143	1.00	1.00	1.00	143
	40	1.01	1.00	1.00	279	1.01	1.00	1.00	275
	60	1.01	1.00	1.00	438	1.01	1.00	1.00	435
	80	1.01	1.00	1.00	577	1.01	1.00	1.00	573
	100	1.01	1.00	1.00	695	1.01	1.00	1.00	693
	All	1.008	1.000	1.001	2132	1.008	1.000	1.000	2119
10	20	1.14	1.05	1.05	44	1.10	1.10	1.10	41
	40	1.09	1.03	1.00	74	1.07	1.06	1.05	73
	60	1.08	1.08	1.05	103	1.06	1.07	1.06	100
	80	1.10	1.06	1.06	130	1.06	1.06	1.06	130
	100	1.07	1.08	1.07	163	1.05	1.05	1.04	159
	All	1.096	1.058	1.044	514	1.067	1.068	1.062	503
Overall		1.079	1.043	1.036	7311	1.068	1.045	1.037	7064