Volume 24 (2), pp. 157–183
http://www.orssa.org.za

ORiON
ISSN 0529-191-X

A survey and comparison of heuristics for the 2D oriented on-line strip packing problem

N Ntene* JH van Vuuren†

Received: 11 August 2008; Revised: 22 October 2008; Accepted: 27 October 2008

Abstract

The two dimensional oriented *on-line* strip packing problem requires items to be packed, one at a time, into a strip of fixed width and infinite height so as to minimise the total height of the packing. The items may neither be rotated nor overlap. In this paper, ten heuristics from the literature are considered for the special case where the items are rectangles. Six modifications to some of these heuristics are proposed, along with two entirely new shelf algorithms. The performances and efficiencies of all the algorithms are compared in terms of the total packing height achieved and computation time required in each case, when applied to 542 benchmark data sets documented in the literature.

Key words: Heuristic packing, on-line packing, shelf algorithm, strip packing.

1 Introduction

The two dimensional *strip packing problem* involves packing a list of items (in this case, rectangles) into a bin (referred to as a strip) of fixed width and infinite height. The objective is to minimise the total packing height in the strip for which rectangles do not overlap. Each rectangle L_i is specified by the pair of dimensions $(h(L_i), w(L_i))$ referring to its height and width respectively. Ntene and Van Vuuren [22] conducted a survey on heuristics for solving *offline* strip packing problems approximately. These are problems where the entire set of rectangles to be packed is known in advance. There are, however, applications where the entire set of rectangles to be packed is not known in advance and problems of this nature are referred to as *on-line* packing problems. Applications of this class of problems include warehouse storage [2, 3], VLSI design [14] and scheduling with a shared resource [3, 6, 20].

In an *on-line* environment, rectangles are packed one at a time; rectangle L_{i+1} only becomes available once rectangle L_i has been packed [2, 13, 14, 19, 20]. Another condition

*Department of Logistics, University of Stellenbosch, Private Bag X1, Matieland, 7602, Republic of South Africa.

†Corresponding author: Department of Logistics, University of Stellenbosch, Private Bag X1, Matieland, 7602, Republic of South Africa, email: vuuren@sun.ac.za

for a system to be fully *on-line* is that once a rectangle has been packed it may not be moved at a later stage of the packing. The challenge in *on-line* packing problems is due to the potential volatility of rectangle heights that have yet to be packed [19].

The main objective in this paper is to examine and compare the time efficiencies and performances of a number of existing heuristics for *on-line* packing problems in the literature, and to propose some improvements or suggest altogether new algorithmic approaches. The paper is organised as follows. In §2 the mechanisms behind a number of existing *level algorithms* for *on-line* packing problems are reviewed and illustrated by means of a numerical example. In §3 a number of *shelf algorithms* from the literature are briefly described and illustrated by means of an example. A number of algorithms for solving *on-line* packing problems with additional constraints (approximately) are discussed and illustrated by means of an example in §4. Then a number of possible modifications to some of these procedures considered in §2–4 are presented in §5. Two entirely new shelf algorithms are presented in §6 and finally all the algorithms are tested on a large set of existing benchmark problem instances so that their performances and time-efficiencies may be compared statistically in §7.

To illustrate the packing patterns produced by the various algorithms mentioned above, all algorithms are applied to an example instance requiring 10 rectangles to be packed into a strip of width 15 units. This is the same example instance used by Ortmann *et al.* [23] to facilitate comparisons for offline packing algorithms. The rectangle dimensions (height, width) for the example instance are shown in Table 1.

L_1	L_2	L_3	L_4	L_5	L_6	L_7	L_8	L_9	L_{10}
$(14,5)$	$(5,4)$	$(4,9)$	$(15,1)$	$(6,11)$	$(6,2)$	$(4,6)$	$(2,5)$	$(6,10)$	$(1,7)$

Table 1: *Dimensions* $(h(L_i), w(L_i))$ *of rectangles* $L_1, \ldots, L_{10}$ *used as example instance in §2–6.*

2 Level Algorithms

The algorithms considered in this section are a slight variation on the algorithms investigated in [22], namely the *next fit decreasing height* (NFDH) [10], the *first fit decreasing height* (FFDH) [10] and the *best fit decreasing height* (BFDH) [11] algorithms. Since we are dealing with *on-line* packing problems, we do away with the pre-ordering condition in each of these original algorithms.

In the *next fit level* (NFL) algorithm [11], rectangles are packed (one at a time and in the order given) on the current level, left justified. The first level corresponds with the bottom of the strip. If there is insufficient horizontal space on the current level to pack the next rectangle, a horizontal line is drawn across the upper edge of the tallest rectangle on the current level so as to create a new level above the current level. All levels below the current level are never revisited.

In the *first fit level* (FFL) algorithm [11], rectangles are packed (one by one in the order given) on the lowest level into which they fit both height-wise and width-wise; if a rectangle does not fit into any existing level, then a new level is created exactly as in the NFL algorithm and the rectangle in question is packed on that level.

The *best fit level* (BFL) algorithm [11] is similar to the FFL algorithm, except that each rectangle is placed on the lowest level (into which it fits both height-wise and width-wise) with minimum residual horizontal space (the space between the right-most edge of the last rectangle packed on a level and the right-hand boundary of the strip).

For our example instance in Table 1, total packing heights of 46, 45 and 42 units are obtained by the NFL, FFL and BFL algorithms respectively, as shown in Figure 1(a)–(c).

3 Shelf algorithms

In shelf algorithms, rectangles are also packed on horizontal planes (referred to as shelves) of fixed height as in the case of level algorithms. However, this class of algorithms differs from the class of level algorithms in that additional space (called *free space*) is intentionally left between the top-most edge of the tallest rectangle on a shelf and the position of the next shelf so as to accommodate (to some degree) potential volatility in the heights of rectangles yet to be packed. However, in a level algorithm, the position of a level coincides with the top-most edge of the tallest rectangle on the previous level. The name *shelf algorithm* is derived from the situation where books are packed in a stack of bookshelves [2].

Shelf algorithms were first designed by Baker *et al.* [2] who modified two existing *offline* heuristics, namely the NFDH and FFDH algorithms [10]. The resulting two shelf algorithms are referred to as the *next fit shelf* (NFS_r) and *first fit shelf* (FFS_r) algorithms, where $0 < r < 1$ is a parameter, and these algorithms are described in §3.1. In these shelf algorithms, the objective is to pack rectangles of similar heights $r^{k+1} < h(L_i) \leq r^k$ on a single shelf of fixed height r^k (for some integer k). The parameter r is a measure of how much free space is allowed on each shelf to accommodate variations in the heights of rectangles to come. A small value of r (approximately equal to zero) results in large-sized shelves — hence allowing for rectangles with large variations in height to be packed on the same level. On the other hand, a large value of r (approximately equal to 1) allows rectangles of almost similar heights to be packed on one level due to the small shelf heights created [25]. For the shelf algorithms applied to our example instance in Table 1, a value of $r = 0.6$ was selected for illustrative purposes.

Coffman [12] modified the BFDH algorithm [11] to arrive at the so-called *best fit shelf* (BFS_r) algorithm, also described in §3.1, which differs from the NFS_r and FFS_r algorithms in a manner analogous to the difference between the NFL, FFL and BFL level algorithms. As Csirik and Woeginger [14] mention, shelf algorithms are based on one dimensional bin packing procedures: after determining an appropriate shelf on which a rectangle may be packed, so that it fits height-wise, the problem then becomes the one dimensional bin packing problem of determining amongst which of the shelves of appropriate height the rectangle should be packed (during this last stage only one dimension, namely width, is of interest, because it has been determined that height-wise the rectangle will fit). It is on this basis that another shelf algorithm, known as the *harmonic shelf* (HS_{M_r}) algorithm is reviewed in §3.2.

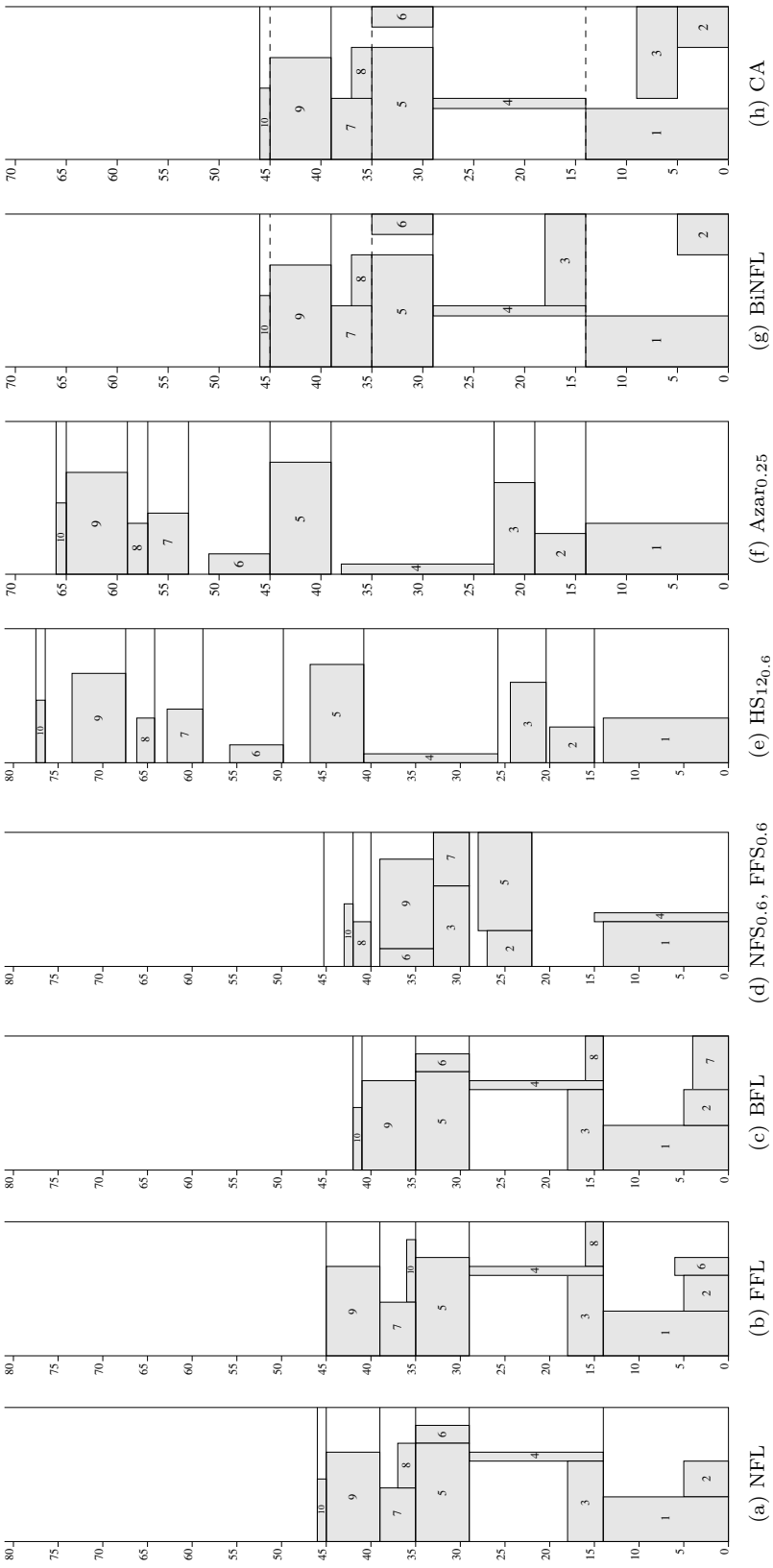


Figure 1: *Packings produced by the algorithms described in §2–4 for the example instance of the strip packing problem in Table 1. Rectangle L_i is denoted by i in the figure, for all $i \in \{1, \ldots, 10\}$.*

3.1 The Next Fit Shelf, First Fit Shelf and Best Fit Shelf algorithms

The *next fit shelf* (NFS_r) algorithm [2] with parameter $0 < r < 1$ is a natural modification of the NFDH algorithm [10], the difference being that the rectangles are not sorted in the NFS_r algorithm; they are merely packed in the order given. In this algorithm, a value of r is initially selected for the entire packing. Before packing each rectangle, the smallest integer k is computed for which $r^{k+1} < h(L_i) \leq r^k$; here r^k is referred to as the *appropriate height* of the shelf to pack rectangle L_i. A rectangle is packed on the highest shelf of appropriate height. If a shelf of appropriate height for rectangle L_i does not exist, a new shelf of appropriate height is created above the top-most shelf and rectangle L_i is packed there, left justified. If a shelf of appropriate height exists, but there is insufficient space to accommodate the rectangle, this shelf is closed off and a new shelf of the same (appropriate) height is created above the top-most level.

The *first fit shelf* (FFS_r) algorithm [2] with parameter $0 < r < 1$ is a modification of the FFDH algorithm [10] and it is similar to the NFS_r algorithm, except that a rectangle is placed left justified on the *lowest* shelf of appropriate height instead of on the highest shelf of appropriate height.

The *best fit shelf* (BFS_r) algorithm [12] with parameter $0 < r < 1$ is a modification of the *best fit decreasing height* (BFDH) algorithm [11]. The difference between the FFS_r and BFS_r algorithms is that once the parameter r has been selected and different values of k determined, the latter procedure packs a rectangle on the lowest shelf of appropriate height *with minimum residual horizontal space.*

As shown in Figure 1(d), a total packing height of 45.27 units is obtained via all three of the $\text{NFS}_{0.6}$, $\text{FFS}_{0.6}$ and $\text{BFS}_{0.6}$ algorithms for our example instance in Table 1.

3.2 The Harmonic Shelf algorithm

Csirik and Woeginger [14] combined a one dimensional bin packing algorithm, called the $harmonic_M$ algorithm and proposed by Lee and Lee [19], with the principles of shelf algorithms. The $harmonic_M$ algorithm is used to partition the interval (0,1] non-uniformly into M intervals $I_1, \ldots, I_M$, where $I_p = (1/(p+1), 1/p]$, $1 \leq p < M$ and $I_M = (0, 1/M]$. A reasonable value of M is considered to be in the range $3 \leq M \leq 12$. This harmonic partition allows a rectangle to be classified according to the interval into which it fits width-wise.

The *harmonic shelf* (HS_{M_r}) algorithm does not only aim to pack rectangles of similar heights on the same shelf; over and above this objective the rectangles should also have similar widths. Before rectangle L_i is packed, two decisions have to be made. The first decision is to determine the appropriate shelves onto which a rectangle may be packed in terms of its height by selecting a value for r and computing a value of k for which $r^{k+1} < h(L_i) \leq r^k$. The second decision is to determine the interval I_p into which the rectangle belongs width-wise, by computing the value of p for which $1/(p+1) < w(L_i) < 1/p$. Only rectangles belonging to I_p, with r^k as the appropriate height may be packed onto such a shelf. If no shelf of appropriate height exists or if there is insufficient horizontal space on all shelves of appropriate height, then a new shelf of appropriate height is created above the current top-most shelf. In our example instance in Table 1, a total packing height of

77.60 units is obtained via the $HS_{12_{0.6}}$ algorithm, as depicted in Figure 1(e). Values of $M = 12$ and $r = 0.6$ were used in this example for illustration purposes.

4 Packings observing the tetris constraint

In all the algorithms reviewed thus far, it was assumed that a rectangle may be packed onto any shelf inside the strip as long as it fits. However, there are applications, such as packing boxes from the back of a delivery vehicle, where rectangles have to be transferred through all succeeding levels before being packed (for example, in order to reach the lower levels of the strip which model the front of the vehicle). This constraint is also found in the game *Tetris* where rectangles drop from the top of the strip to reach lower levels and the player has to avoid being blocked by rectangles already packed in other levels. Three existing algorithms in the literature, taking this additional constraint into consideration, are reviewed in this section.

4.1 The Azar_Y algorithm

This algorithm is from a paper by Azar and Epstein [1]. In the Azar_Y algorithm, the rectangle widths are assumed to be in the range (0,1] and the strip has width 1, without loss of generality. However, there is no restriction on the rectangle heights. The Azar_Y algorithm partitions the strip into horizontal levels by means of a real threshold constant $0 < Y < \frac{1}{2}$. Rectangles of particular heights ($2^{j-1} < h(L_i) \leq 2^j$) and widths ($2^{-x-1} < w(L_i) \leq 2^{-x}$) are packed on the same level, referred to as an (x, j) level (where $j \in \mathbb{Z}$ and $x \in \mathbb{N}$).

A rectangle whose width is at least Y is referred to as a *buffer*. When the next rectangle to be packed arrives, it is classified either as a *buffer* or *non-buffer*, depending on its width. If it is a *buffer*, a new level, whose height is equal to the height of the *buffer*, is created above the top-most level and the rectangle is packed there, left justified. This means that *buffers* are packed on their own within levels. If the rectangle is a *non-buffer*, it is classified as an (x, j) rectangle, for some $j \in \mathbb{Z}$ and some $x \in \mathbb{N}$. The first *non-buffer* rectangle packed on a level determines the height of the level as 2^j and this level becomes an (x, j) level. If a rectangle fits on an (x, j) level and it can reach such a level without being blocked by any of the *buffers*, then it is placed on that level. However, if no such level exists, if the rectangle does not fit on a particular (x, j) level or if the rectangle is blocked, then a new level of height 2^j is created above the top-most level. For our example instance in Table 1, a total packing height of 66 units is obtained via the $\text{Azar}_{0.25}$ algorithm, as depicted in Figure 1(f), where the value of $Y = 0.25$ was chosen for illustrative purposes.

4.2 The Bi-level Next Fit algorithm

The *Bi-level next fit* (BiNFL) algorithm [9] is a modification of the NFL algorithm described in §2. As the name suggests, the algorithm packs two levels at a time, referred to as the *lower* and *upper* levels. The height of the *lower* level is determined by the height of the tallest rectangle packed on it.

The first rectangle L_i to be packed on a bi-level is placed on the *lower* level, left justified. If the next rectangle L_{i+1} to be packed fits on the *lower* level, it is placed there, right justified. All other rectangles that follow and fit on the *lower* level are placed there, right justified, next to the previous rectangle packed. If there is not enough room for a rectangle to be packed on the *lower* level, packing proceeds on the *upper* level. A horizontal line is drawn along the top-most edge of the tallest rectangle on the *lower* level and this becomes the lower boundary of the *upper* level.

If, on the *upper* level, rectangle L_{i+1} is the first rectangle to be packed (because it failed to fit on the *lower* level), it is packed left justified on top of L_i since it is the only rectangle on the *lower* level. Subsequent rectangles are packed left justified on this level provided there is sufficient space (see Figure 2(a)). If, on the other hand, L_{i+2} is the first rectangle to be packed on the *upper* level, it is packed above the shorter of L_i and L_{i+1} (because these are the only two rectangles on the *lower* level), justified against the same strip boundary as the shorter of rectangles L_i and L_{i+1}; this scenario is depicted in Figures 2(b) and (c). If there are more than two rectangles on the *lower* level, the first rectangle packed on the *upper* level is packed above the shorter of the first left justified or the first of the right justified rectangles on the *lower* level. If a rectangle does not fit on the *upper* level, a new bi-level is created above the top-most level and similar steps are carried out as defined for the *lower* and *upper* levels until all rectangles are packed.

A total packing height of 46 units is obtained for our example instance in Table 1, as shown in Figure 1(g), with the *lower* and *upper* levels within each bi-level separated by dashed lines.

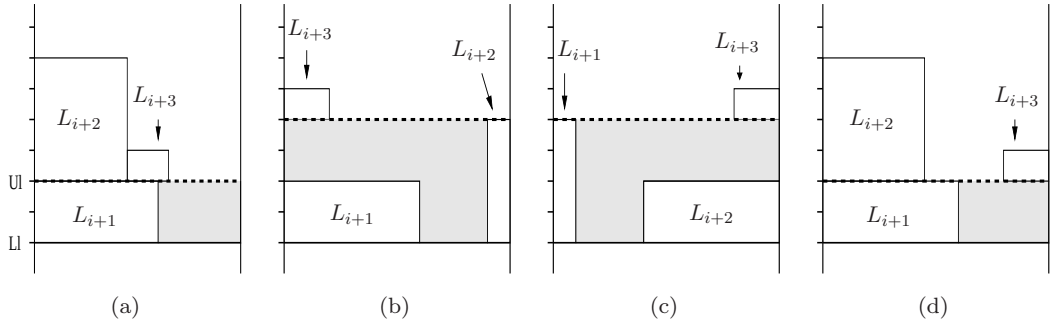


Figure 2: *(a)–(c) Examples of patterns resulting from a BiNFL packing. (d) In the CA algorithm the second rectangle packed on the* upper *level is right justified provided only one rectangle is packed on the* lower *level. Ul and Ll represent the lower boundaries of the* upper *and* lower *levels respectively.*

4.3 The Compression algorithm

The *compression algorithm* (CA) [9] is an extension of the BiNFL algorithm. It exploits certain patterns (when only one or two rectangles are packed on the *lower* level) that result from a BiNFL packing. In the CA algorithm, packing on the *lower* level proceeds in a manner similar to a BiNFL packing. However, if rectangle L_i ($i \geq 3$) is the first

rectangle to be packed on the *upper* level, it is justified according to the shorter of the first left justified or first right justified rectangles on the *lower* level, and it is slid down onto the *lower* level provided there is sufficient space (see Figures 2(b) and (c)) — this process is called *compression*. If rectangle L_i ($i \geq 3$) is the second rectangle to be packed (*i.e.* if there is one rectangle on each level, each of them left justified), it is right justified and if there is sufficient room on the *lower* level, this rectangle is compressed down onto the *lower* level (see Figure 2(d)). Subsequent rectangles that fit on the *lower* level may also be shifted next to previously compressed rectangles. Packing continues on the *upper* level as in the BiNFL algorithm for rectangles that may not be compressed down. A rectangle that fails to fit on the *upper* level is placed in a new bi-level that is created above the top-most level and previous bi-levels are never revisited. In our example instance in Table 1, a total packing height of 46 units is obtained via the CA algorithm, as shown in Figure 1(h).

5 Proposed Modifications

A number of modifications to some of the algorithms reviewed in §2–4 are proposed in this section.

5.1 The Modified Next Fit, First Fit and Best Fit Level algorithms

As the name suggests, the *modified next fit level* (MNFL) algorithm is a newly proposed variation on the NFL algorithm described in §2. In the MNFL algorithm, the first rectangle packed on a level determines the height of that level. If a rectangle is encountered that does not fit onto the current level, that level is closed off in both these algorithms and a new current level is created above it. The NFL algorithm is expected to perform poorly if the rectangles are presented in an order in which they tend to increase in height. However, if the rectangles are presented in an order in which they tend to decrease in height, then the algorithm is expected to perform well. The MNFL algorithm differs from the NFL algorithm in that in the latter procedure, level heights are determined by the tallest rectangle packed on a level, while in the former procedure, level heights are determined by the first rectangle packed on the level. For our example instance in Table 1, a total packing height of 44 units is obtained via the MNFL algorithm, as shown in Figure 5(a).

In the *modified first fit level* (MFFL) algorithm, the height of each level corresponds to the height of the first rectangle packed on that level. The MFFL and FFL algorithms differ in a manner analogous to the difference between the MNFL and NFL algorithms. A total packing height of 41 units is obtained via the MFFL algorithm for our example instance in Table 1, as illustrated in Figure 5(b).

The *modified best fit level* (MBFL) algorithm is similar to the BFL algorithm, except that in the BFL algorithm the height of a level is determined by the height of the tallest rectangle packed on the level, while in the MBFL algorithm the height of a level is determined by the first rectangle packed on the level. A total packing height of 40 units is obtained via the MBFL algorithm for our example instance in Table 1, as illustrated in Figure 5(c).

5.2 The Compression Part Fit algorithm

Downey [15] mentions that the CA algorithm (described in §4.3) is far from optimal, because it only takes a few patterns into consideration (where it may be possible to compress rectangles from the *upper* to the *lower* level). The *compression part fit* (CPF) algorithm is proposed to accommodate more patterns occurring within a *bi-level*. An idea originally introduced by Burke *et al.* [7] of using a linear array whose size equals the width of the strip is employed. Each element of the array is used to store the height of rectangles packed at that coordinate of the array. However, the drawback of using such an array is that it requires the dimensions of the rectangles and the strip to be integers. Two versions of the CPF algorithm are proposed for use when dealing with floating point data. The first version involves rounding the dimensions (up or down) to the nearest integer, which may not necessarily represent a true packing, but it maintains the characteristics of the data. On the other hand, the second version wastes space by rounding *up* the dimensions to the nearest integer, thereby creating a feasible packing for the original rectangles.

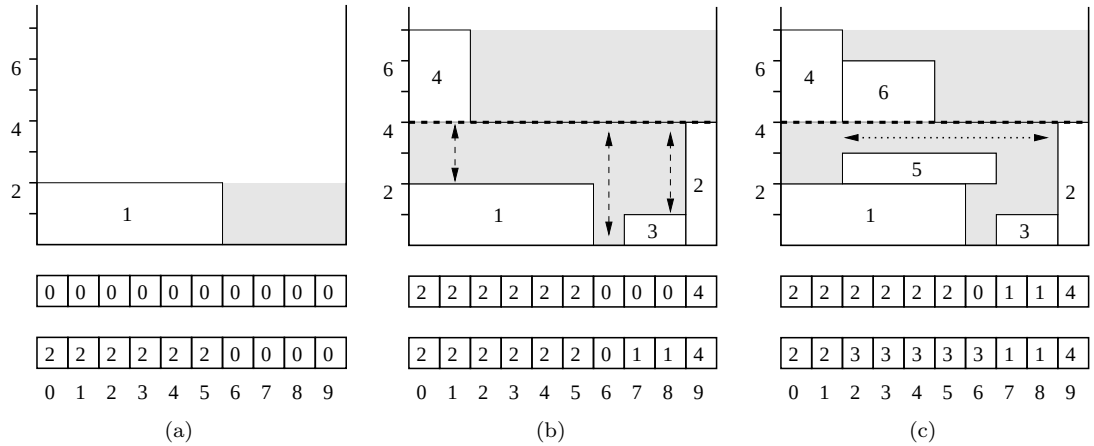


Figure 3: *Examples of how a linear array is populated when a new bi-level is created. (a) The upper linear array containing zeros represents a new bi-level with no rectangles packed. The lower linear array stores the height of the rectangle packed on the* lower *level from coordinates 0 to 5. (b) The upper linear array stores the heights of the first and second rectangles packed. The lower linear array stores height of the third rectangle and the* vertical space *is indicated by the dashed arrows at certain coordinates of the linear array. (c) The fifth rectangle has been compressed down onto the* lower *level by the CFF algorithm. The* horizontal space *is indicated by the horizontal dotted arrow.*

Bi-level Stage. Packing on the *lower* level proceeds exactly as in the BiNFL algorithm, except that a linear array is used to represent the various heights of rectangles packed on the *lower* level only. Before any packing takes place on a bi-level, the linear array contains only zeros. On the *upper* level, the CPF algorithm differs from the BiNFL algorithm in that rectangles are always packed left justified. A *vertical space* on the *lower* level is defined as the space between the lower boundary of the *upper* level and the upper edge of rectangles packed on the *lower* level (or sometimes the lower boundary of the *lower* level) at each coordinate of the linear array. Three vertical spaces of heights $2, 4$ and 3 units are indicated by dashed vertical arrows in Figure 3(b) at coordinates 1, 6 and 8 respectively. A

horizontal space on the *lower* level, on the other hand, is defined as the space between the left-hand edge of a rectangle being considered for compression downwards and the nearest left-hand edge of a rectangle packed on the *lower* level at a height given in the linear array at the coordinate corresponding to the left-hand edge of the rectangle. A horizontal space of 7 units is shown in Figure 3(c) by the horizontal dotted arrows, computed from the coordinates 2 to 8 at a height of 3 (given in the lower linear array, at coordinate 2 which corresponds to the left-hand edge of the sixth rectangle).

Compression Stage. For a rectangle to be compressed down onto the *lower* level, two conditions must be satisfied:

1. The height of the rectangle must exceed the height of the vertical space. The width of a rectangle may be covered by a single value (Figure 4(a)) or different values of the vertical space (Figure 4(b)). If more than one value of the vertical space covers the entire width of the rectangle, the height of the rectangle must exceed the smallest value of the vertical spaces.
2. The width of the rectangle must not exceed the width of the horizontal space.

Provided that the two conditions above are satisfied, the rectangle in question is compressed down so that its bottom edge rests on the top edge of a rectangle on the lower level. The algorithm is expected to perform better if the tallest rectangle on the *upper* level may be compressed onto the *lower* level. A total packing height of 35 units is obtained when the CPF algorithm is applied to our example instance in Table 1, as shown in Figure 6(a).

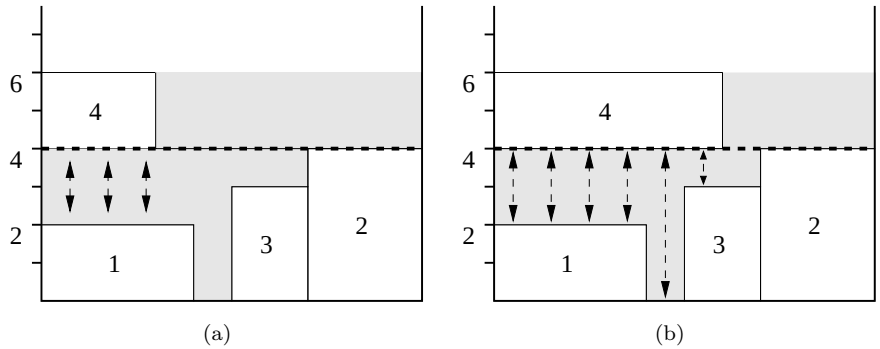
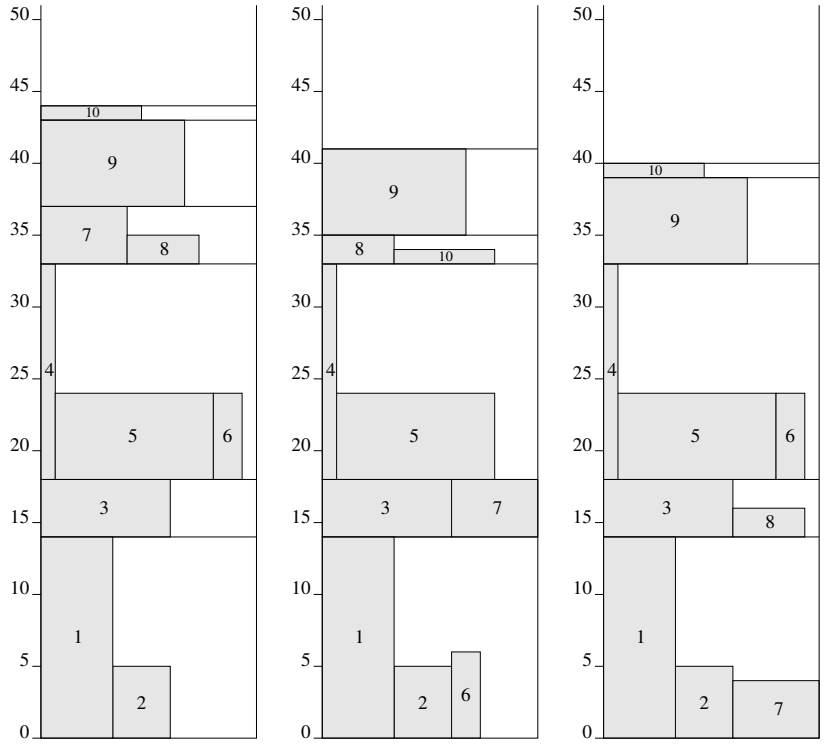


Figure 4: *Example of how the width of a rectangle is covered by: (a) one value of vertical space, or (b) more than one value of vertical space.*

5.3 The Compression Full Fit algorithm

The steps of the *compression full fit* (CFF) algorithm and the CPF algorithm are similar in all respects, except for condition 1 of the compression stage. In the CFF algorithm, a rectangle is compressed down onto the *lower* level provided its height is less than or equal to the vertical space covering the entire width of the rectangle. The advantage of doing this is that the residual vertical space (the vertical space remaining after a rectangle is compressed down) may be considered again when packing the next rectangle. Before rectangle 5 was compressed down in Figure 3(c), there were vertical and horizontal spaces



(a) MNFL (b) MFFL (c) MBFL

Figure 5: *Packings produced by the modified algorithms described in §5.1 for the example instance of the strip packing problem in Table 1. Rectangle L_i is denoted by i in the figure, for all $i \in \{1, \ldots, 10\}$.*

of 2 and 7 units respectively at coordinate 2. After rectangle 5 was compressed down onto the *lower* level, a vertical space of 1 unit resulted. If rectangle 6 had a height of 1 unit, then it would be compressed down onto the *lower* level. The idea in the CFF algorithm is to increase the probability of packing more rectangles on the *upper* level by utilising the space remaining after compression of a rectangle onto the *lower* level. Once a rectangle is compressed onto the lower level, the space it was supposed to occupy on the upper level may be used to pack other rectangles. The algorithm is expected to perform better if the tallest rectangle on the *upper* level may be compressed onto the *lower* level and if more rectangles fit onto the *upper* level. The latter implies an increased probability of creating fewer levels, hence possibly leading to a decrease in the overall strip height. When the CFF algorithm is applied to our example instance in Table 1, a total packing height of 46 units is obtained, as illustrated in Figure 6(b).

5.4 The Compression Combo algorithm

The *compression combo* (CC) algorithm is a combination of the first conditions of the compression stages of the CPF and CFF algorithms. In the CC algorithm, any rectangle may be compressed down onto the *lower* level regardless of whether it fits fully or partially onto the *lower* level, as long as the second condition is satisfied, namely that the width of

the rectangle to be compressed down is at most the width of the horizontal space. When the CC algorithm is applied to our example instance in Table 1, a total packing height of 35 units is again obtained, as illustrated in Figure 6(a).

6 Two New Shelf Algorithms

In this section two new shelf algorithms are suggested. The algorithms highlight two different methods of creating *free space* in between shelves, based on the packing history, so as to cater for the volatility in heights of rectangles still to be packed.

6.1 The Shelf Deviation algorithm

In the newly proposed *shelf deviation* (SDev) algorithm the notion of a shelf *type* refers to a collection of shelves of equal height and the objective is to increase these fixed heights as more *types* are created. A $type_1$ shelf only accommodates rectangles of height $0 < h(L_i) \leq h(L_1)$ where L_1 is the first rectangle to be packed (*i.e.* the height of the first rectangle determines the height of the first shelf *type*). A rectangle whose height fits within this range is referred to as a $type_1$ rectangle. The height of a subsequent shelf of $type_j$ ($j \geq 2$) equals the height of the first rectangle packed on the shelf together with a certain proportion, referred to as the *shelf height increase proportion*. This proportion is computed as the *standard deviation* (stdev) of the rectangle heights already packed on all shelves, *i.e.* $h(type_j) = h(L_{i+1}) + stdev(h(L_1), \ldots, h(L_{i+1}))$. In general, $type_j$ shelves can accommodate rectangles of height $h(type_{j-1}) < h(L_i) \leq h(type_j)$, where $j \geq 2$.

Rectangles are classified according to the shelf *type* to which they belong and are packed onto the lowest shelf of that *type*. New shelf *types* are created above the top-most shelf each time the next rectangle has a height exceeding the height of all existing shelf *types*. It is not necessary for two consecutive shelves to be of the same *type* — the shelf *types* may be interspersed, as long as rectangles are placed onto appropriate shelf *types*. If there is insufficient horizontal space to accommodate a rectangle, a new shelf of the appropriate *type* is created above the top-most shelf for that rectangle. In our example instance in Table 1, a total strip height of 90.80 units is obtained via the SDev algorithm, as shown in Figure 6(c). A pseudocode listing of the steps of this algorithm is given in the appendix.

6.2 The Shelf Difference algorithm

The *shelf difference* (SDiff) algorithm differs from the SDev algorithm only in the way the shelf height increase proportion is computed. In the SDiff algorithm, a $type_1$ shelf is still determined by the height of the first rectangle packed. For a subsequent shelf of type $type_j$ ($j \geq 2$), instead of computing the standard deviation, the shelf height increase is taken as the difference between the height of the rectangle to be packed and the previous shelf height added to the height of the previous shelf *type*, *i.e.* $h(type_j) = (h(L_{i+1}) - h(type_{j-1})) + h(L_{i+1})$. A total packing height of 86 units is obtained when the SDiff algorithm is applied to our example instance in Table 1, as shown in Figure 6(d). A pseudocode listing of the steps of this algorithm is also given in the appendix.

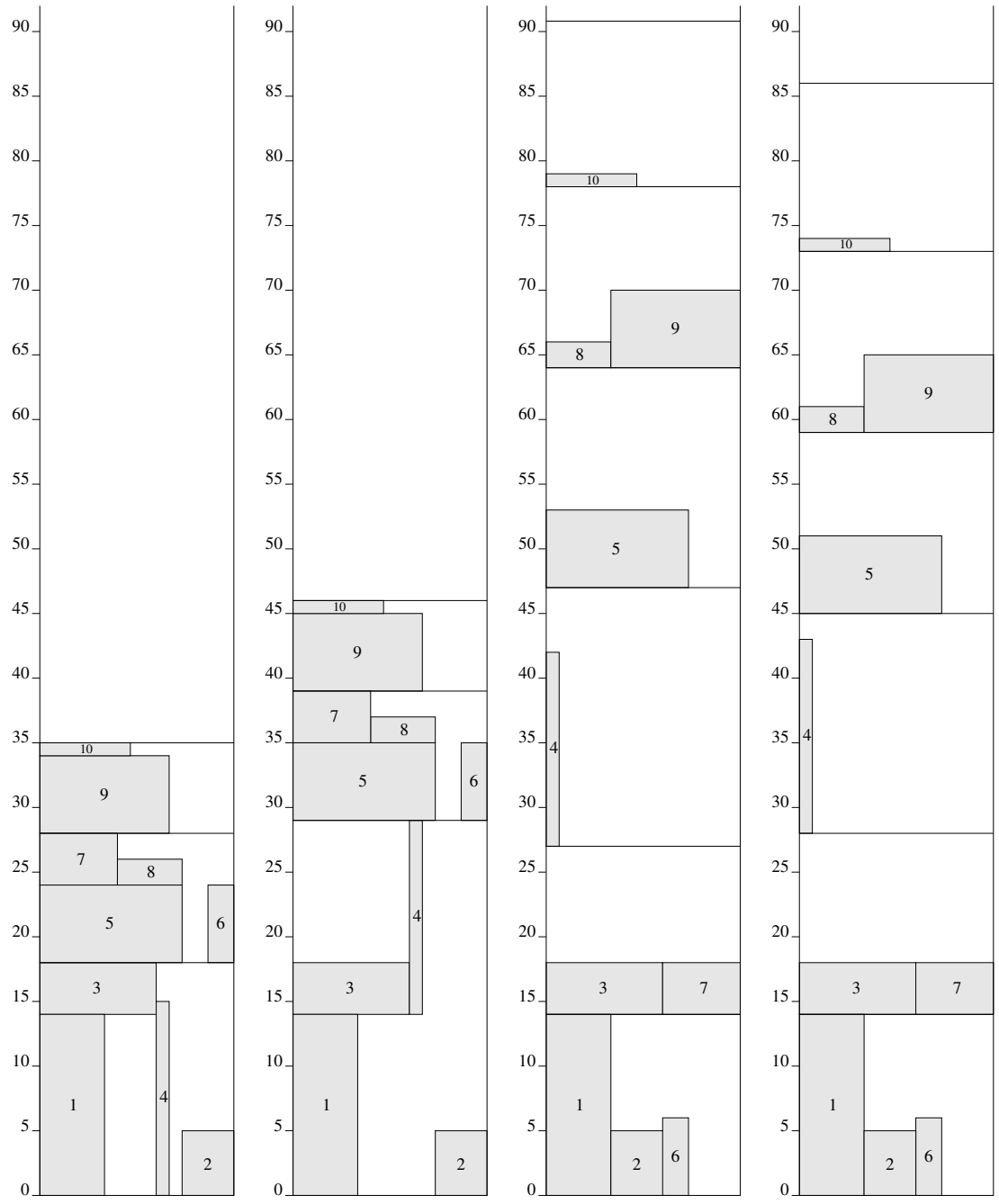


Figure 6: *Packings produced by the algorithms described in §5.2 –6.2 for the example instance of the strip packing problem in Table 1. Rectangle L_i is denoted by i in the figure, for all $i \in \{1, \ldots, 10\}$.*

7 Comparison of algorithmic results

The efficiencies of and solution qualities obtained by the algorithms presented in §2–6 were compared by applying them to the 542 benchmark instances of Beasley [4, 5], Burke *et al.* [7], Christofides and Whitlock [8], Hopper and Turton [16, 17] and Mumford-Valenzuela [21]. For a full description on how these benchmark data sets were generated, the reader is referred to [22]. Each algorithm's performance was measured by means of the mean packing height obtained as well as by the mean execution time, computed over all benchmark data sets. Statistical tools used in the comparison of each algorithm's performance include the *student's t-test*, *ANalyses Of VAriance* (ANOVA) and the *chi-squared* test. All these tests were carried out at a 5% level of significance. The *t-test* and *ANOVA* were used to compare the mean packing heights obtained by the algorithms over the 542 instances, while the *chi-squared* test was used to compare the frequencies with which the various algorithms obtained the smallest packing height and to determine whether, statistically, there were any significant differences between these frequencies. Where the results from the ANOVA indicated significant differences, the method of *Least Significance Difference* (LSD) was employed to determine between which algorithms these differences arose.

While testing the algorithms, it was observed that in most of the 542 data sets, the initial rectangles have larger heights than the rectangles towards the ends of the packing lists. Hence each algorithm was tested three times on each data set, by changing the order in which rectangles enter the system from the data set list—either in the *normal or forward* order, in the *reverse* order and in a *random* order.

7.1 Level algorithms

The level algorithms from the literature for *online* packing problems described in §2 were compared with the suggested modifications in §5. The results shown in the first section of Figure 7 indicate that the mean packing height obtained in the *forward* traversal order of the data sets is smaller than in the *reverse* order. This is because in the *forward* order, packing typically begins with rectangles of greater height and for those algorithms that allow revisiting of existing levels, the smaller rectangles may be inserted on any available level with sufficient space—thus decreasing the probability of creating new levels. An ANOVA was carried out separately for each order and in all instances the results revealed that there are significant differences between the mean packing heights obtained. In all three traversal orders, the newly suggested MFFL algorithm obtained the smallest mean packing height, although the LSD indicated that there were no significant differences between the mean packing heights obtained by the MFFL, BFL, FFL and MBFL algorithms (indicated "✕" by entries in Table 2). There were significant differences between the mean packing heights obtained by algorithms that do not revisit existing levels (NFL, MNFL, BiNFL) and those allowing existing levels to be revisited (FFL, BFL, MFFL, MBFL), as expected.

In terms of the algorithmic frequencies of obtaining the smallest packing height (which may be seen in the first section of Figure 9) the results of the chi-squared test revealed that there were significant differences between those frequencies achieved by the various algorithms. The MFFL algorithm has the largest frequency in all traversal orders—hence

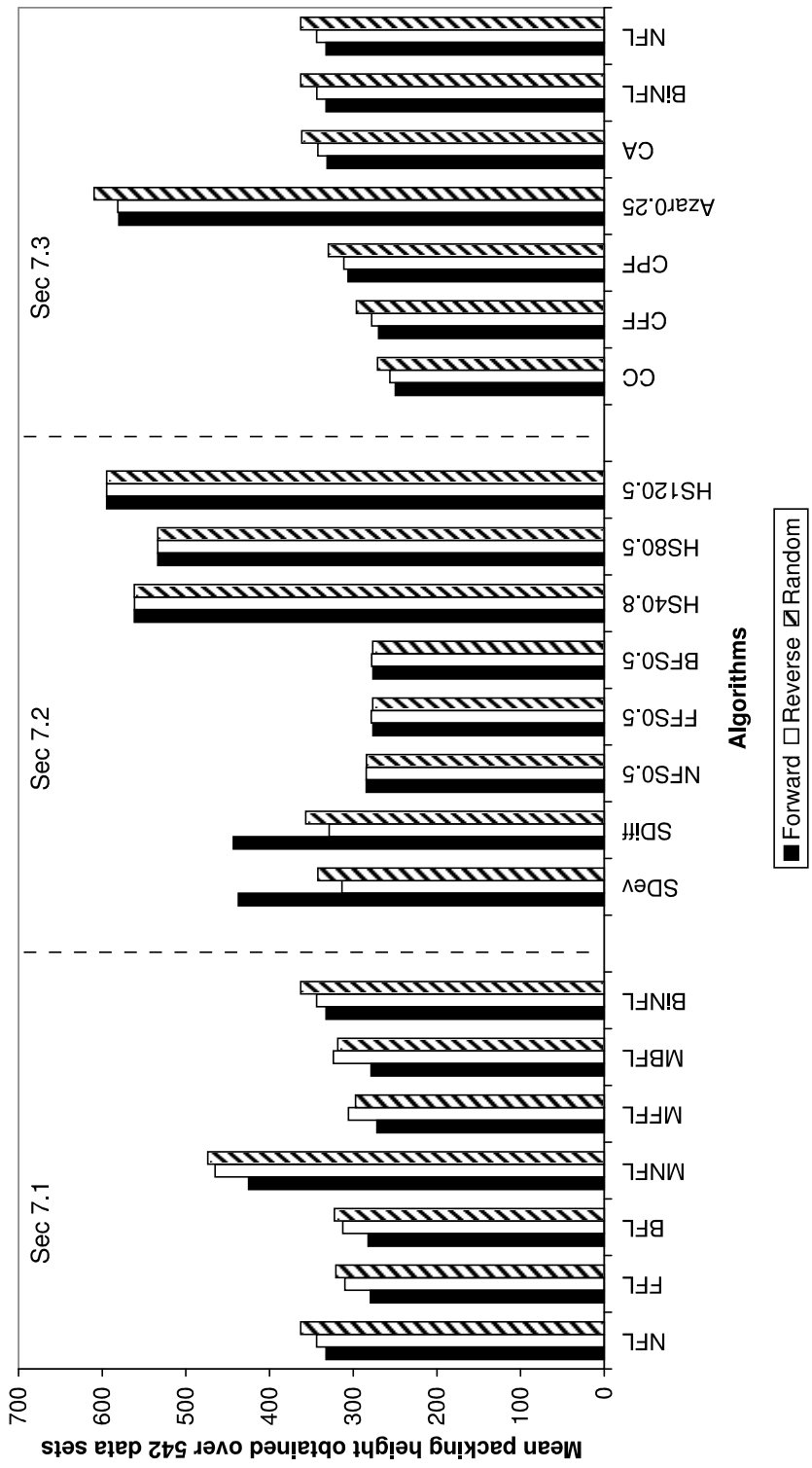


Figure 7: *Comparison of the average packing heights obtained over 542 benchmark data sets by the algorithms described in §2–6. As mentioned in the text, each algorithm was tested for 3 different orders in which rectangles enter the system from the benchmark lists: forward, reverse and random.*

Level algorithms

		NFL	FFL	BFL	MNFL	MFFL	MBFL
NFL	332.627						
FFL	279.426	✓					
BFL	282.215	✓	×				
MNFL	425.084	✓	✓	✓			
MFFL	271.750	✓	×	×	✓		
MBFL	278.584	✓	×	×	✓	×	
BiNFL	332.627	×	✓	✓	✓	✓	✓

		NFL_R	FFL_R	BFL_R	$MNFL_R$	$MFFL_R$	$MBFL_R$
NFL_R	343.823						
FFL_R	310.039	✓					
BFL_R	312.634	✓	×				
$MNFL_R$	465.227	✓	✓	✓			
$MFFL_R$	305.739	✓	×	×	✓		
$MBFL_R$	323.638	×	×	×	✓	×	
$BiNFL_R$	343.833	×	✓	✓	✓	✓	×

		NFL_{Ra}	FFL_{Ra}	BFL_{Ra}	$MNFL_{Ra}$	FFL_{Ra}	$MBFL_{Ra}$
NFL_{Ra}	362.742						
FFL_{Ra}	320.633	✓					
BFL_{Ra}	322.501	✓	×				
$MNFL_{Ra}$	473.743	✓	✓	✓			
$MFFL_{Ra}$	297.316	✓	×	×	✓		
$MBFL_{Ra}$	318.585	✓	×	×	✓	×	
$BiNFL_{Ra}$	362.742	×	✓	✓	✓	✓	✓

Shelf algorithms

		SDev	SDiff	$NFS_{0.5}$	$FFS_{0.5}$	$BFS_{0.5}$	$HS_{4_{0.8}}$	$HS_{8_{0.5}}$
SDev	437.459							
SDiff	443.299	×						
$NFS_{0.5}$	284.532	✓	✓					
$FFS_{0.5}$	276.580	✓	✓	×				
$BFS_{0.5}$	276.499	✓	✓	×	×			
$HS_{4_{0.8}}$	561.612	✓	✓	✓	✓	✓		
$HS_{8_{0.5}}$	533.532	✓	✓	✓	✓	✓	×	
$HS_{12_{0.5}}$	594.578	✓	✓	✓	✓	✓	×	✓

		$SDev_R$	$SDiff_R$	$NFS_{0.5R}$	$FFS_{0.5R}$	$BFS_{0.5R}$	$HS_{4_{0.8R}}$	$HS_{8_{0.5R}}$
$SDev_R$	313.592							
$SDiff_R$	328.616	×						
$NFS_{0.5R}$	284.464	×	✓					
$FFS_{0.5R}$	278.543	×	✓	×				
$BFS_{0.5R}$	277.985	×	✓	×	×			
$HS_{4_{0.8R}}$	561.216	✓	✓	✓	✓	✓		
$HS_{80.5R}$	533.380	✓	✓	✓	✓	✓	×	
$HS_{12_{0.5R}}$	594.494	✓	✓	✓	✓	✓	×	✓

		$SDev_{Ra}$	$SDiff_{Ra}$	$NFS_{0.5Ra}$	$FFS_{0.5Ra}$	$BFS_{0.5Ra}$	$HS_{4_{0.8Ra}}$	$HS_{8_{0.5Ra}}$
$SDev_{Ra}$	342.360							
$SDiff_{Ra}$	356.738	×						
$NFS_{0.5Ra}$	284.263	✓	✓					
$FFS_{0.5Ra}$	276.922	✓	✓	×				
$BFS05_{Ra}$	276.923	✓	✓	×	×			
$HS_{4_{0.8Ra}}$	561.499	✓	✓	✓	✓	✓		
$HS_{80.5Ra}$	533.671	✓	✓	✓	✓	✓	×	
$HS_{12_{0.5Ra}}$	594.659	✓	✓	✓	✓	✓	×	✓

Special case algorithms

		CC	CFF	CPF	$Azar_{0.25}$	CA	BiNFL
CC	249.664						
CFF	269.568	×					
CPF	306.458	✓	✓				
$Azar_{0.25}$	580.181	✓	✓	✓			
CA	331.367	✓	✓	×	✓		
BiNFL	332.627	✓	✓	×	✓	×	
NFL	332.627	✓	✓	×	✓	×	×

		CCR	CFF_R	CPF_R	$Azar_{0.25R}$	CA_R	$BiNFL_R$
CC_R	256.172						
CFF_R	278.014	×					
CPF_R	311.205	✓	×				
$Azar_{0.25R}$	581.475	✓	✓	✓			
CA_R	342.191	✓	✓	×	✓		
$BiNFL_R$	343.644	✓	✓	×	✓	×	
NFL_R	343.823	✓	✓	×	✓	×	×

		CC_{Ra}	CFF_{Ra}	CPF_{Ra}	$Azar_{0.25Ra}$	CA_{Ra}	$BiNFL_{Ra}$
CC_{Ra}	271.036						
CFF_{Ra}	296.136	×					
CPF_{Ra}	329.928	✓	×				
$Azar_{0.25Ra}$	609.721	✓	✓	✓			
CA_{Ra}	361.416	✓	✓	×	✓		
$BiNFL_{Ra}$	362.742	✓	✓	×	✓	×	
NFL_{Ra}	362.742	✓	✓	×	✓	×	×

Table 2: *LSD results for level, shelf and special case algorithms. A block containing a ✓ (resp. ×) indicates that there are (resp. not) significant differences between the means of the algorithms in the corresponding row and column. The subscripts R and Rand refer to the* reverse *and* random *orders of traversing the data sets respectively.*

this is statistically the best algorithm within this class of level algorithms, at a 5% level of significance.

Further tests were also carried out to determine whether the data set traversal order plays a significant role in each algorithm's performance measured. The results shown in Table 4 indicate that, in terms of the mean packing height obtained, order does not play a significant role in the NFL and BiNFL algorithms. However, when it comes to a frequency analysis, it is only for the MNFL algorithm that traversal order is unimportant (see Table 4).

The class of FFL, MFFL, BFL and MBFL algorithms considered to have achieved better performances in terms of smallest mean packing height obtained, have correspondingly longer execution times than the poorer performing class of NFL, MNFL and BiNFL algorithms (see Table 4). This is an expected result, because in the former class of algorithms the strip has to be searched from the bottom upwards for a level with sufficient space and this is time consuming — particularly for a large number of levels. Ideally the best performing algorithm should achieve the smallest packing height in the quickest time. However, the results indicate that a trade-off exists between algorithms that yield better solutions, but which take longer to execute, and algorithms yielding solutions of lesser quality, but which exhibit faster execution times.

Another investigation was carried out in terms of the aspect ratios of the 1 626 data sets (a combination of all three traversal orders for all 542 benchmark data sets). From the 1 626 instances, only instances where an algorithm obtained the smallest packing height were selected and the *standard deviation* (stdevAR) and *mean* (meanAR) of the aspect ratios of the rectangles in these instances were computed. The fraction $stdevAR/meanAR$, known as the *coefficient of variation* (CV), was used to reflect the variation of rectangle aspect ratios relative to the mean. The numbers of data sets for which each algorithm obtained the smallest height associated with values of the CV are shown in Figure 8. If, for instance, a value of 3 is selected for the CV, it may be seen in the figure that the BFL, MBFL, FFL and MFFL algorithms were all able to obtain the smallest packing height, on average. Of these algorithms, the MFFL algorithm obtained the smallest packing height for the largest number of data sets (825).

An interesting question is the following: Given a data set with a known CV value, which level algorithm should be recommended to give the best solution, on expectation? To answer this question, the CV values for test instances where each algorithm obtained the smallest height were analysed. The objective was to determine a threshold CV value beyond which significant differences occur between frequencies in obtaining the smallest packing height by each level algorithm and below which any of the level algorithms may be used. This was achieved by starting with the smallest value and iteratively determining the frequency with which each level algorithm obtained the smallest packing height for that particular CV value. At each iteration, before the CV value was increased, a chi-squared test was performed to determine whether there were any significant differences between the frequencies obtained by each level algorithm. As the value of CV was increased, a point was reached where a slight increment results in significant differences between the frequencies obtained by each level algorithm. We call such a point the *threshold CV value*, and this value was found to be 0.438 in the case of level algorithms. This means for data

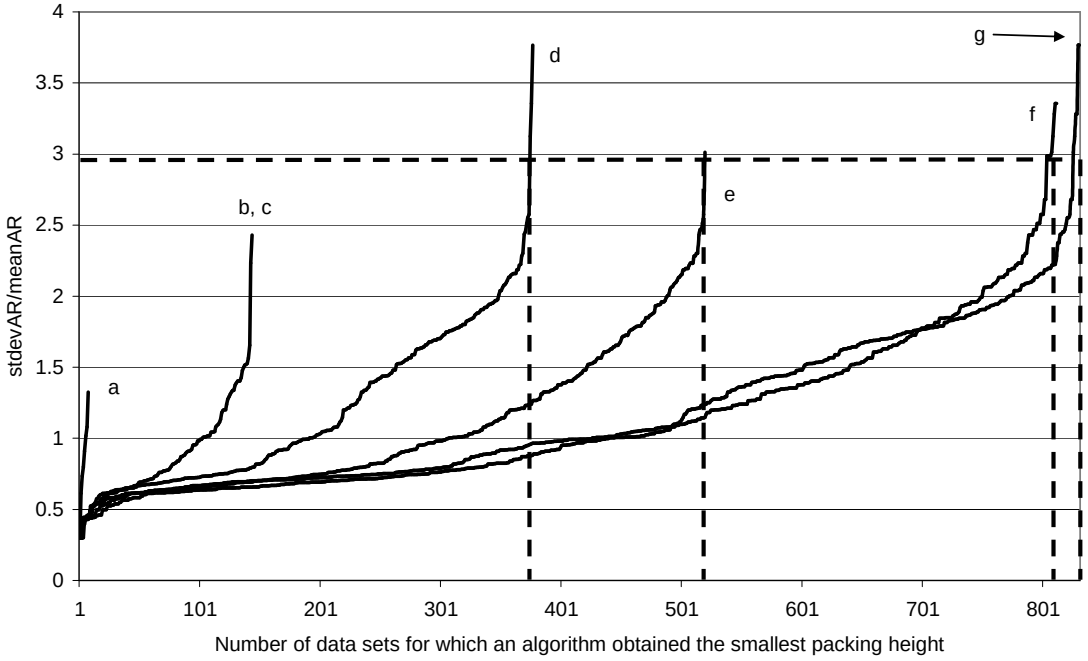


Figure 8: *Aspect ratio analysis for level algorithms: a – MNFL, b – NFL, c – BiNFL, d – MBFL, e – BFL, f – FFL and g – MFFL.*

sets with CV values below the threshold, any of the algorithms may be used, but for values greater than the threshold, the MFFL algorithm is recommended.

7.2 Shelf algorithms

When comparing the shelf algorithms, the algorithms discussed in §3 pose a problem, because they depend on a parameter $0 < r < 1$. Over and above this, the HS_{M_r} algorithms also depend on the value of a parameter $3 \leq M \leq 12$. Hence each of the algorithms was implemented with the representative parameter values $r = 0.2, 0.5, 0.8$ and $M = 4, 8, 12$ resulting in six classes of the algorithms $(\mathrm{NFS}_r, \mathrm{FFS}_r, \mathrm{BFS}_r, \mathrm{HS}_{4_r}, \mathrm{HS}_{8_r}, \mathrm{HS}_{12_r})$. An ANOVA was performed on each of these six classes for the three different traversal orders and the results are shown in Table 3. In the NFS_r class, no significant differences were observed between the $\mathrm{NFS}_{0.5}$ and $\mathrm{NFS}_{0.8}$ algorithms. However, the $\mathrm{NFS}_{0.5}$ algorithm was selected for further comparisons since it achieved a smaller mean packing height over all benchmark sets. For algorithmic instances whose mean heights showed no significant difference, a selection of algorithms to be used for the purposes of further comparison was simply based on the algorithmic instance achieving a smaller mean packing height. Hence the following algorithms were selected in all traversal orders: $\mathrm{FFS}_{0.5}, \mathrm{BFS}_{0.5}, \mathrm{HS}_{4_{0.8}}, \mathrm{HS}_{8_{0.5}}$ and $\mathrm{HS}_{12_{0.5}}$.

The selected shelf algorithms and the two new shelf algorithms were compared in terms of the mean packing height obtained and the results are shown in the second section of Figure 7. The results indicate that, in terms of the mean packing height obtained, the

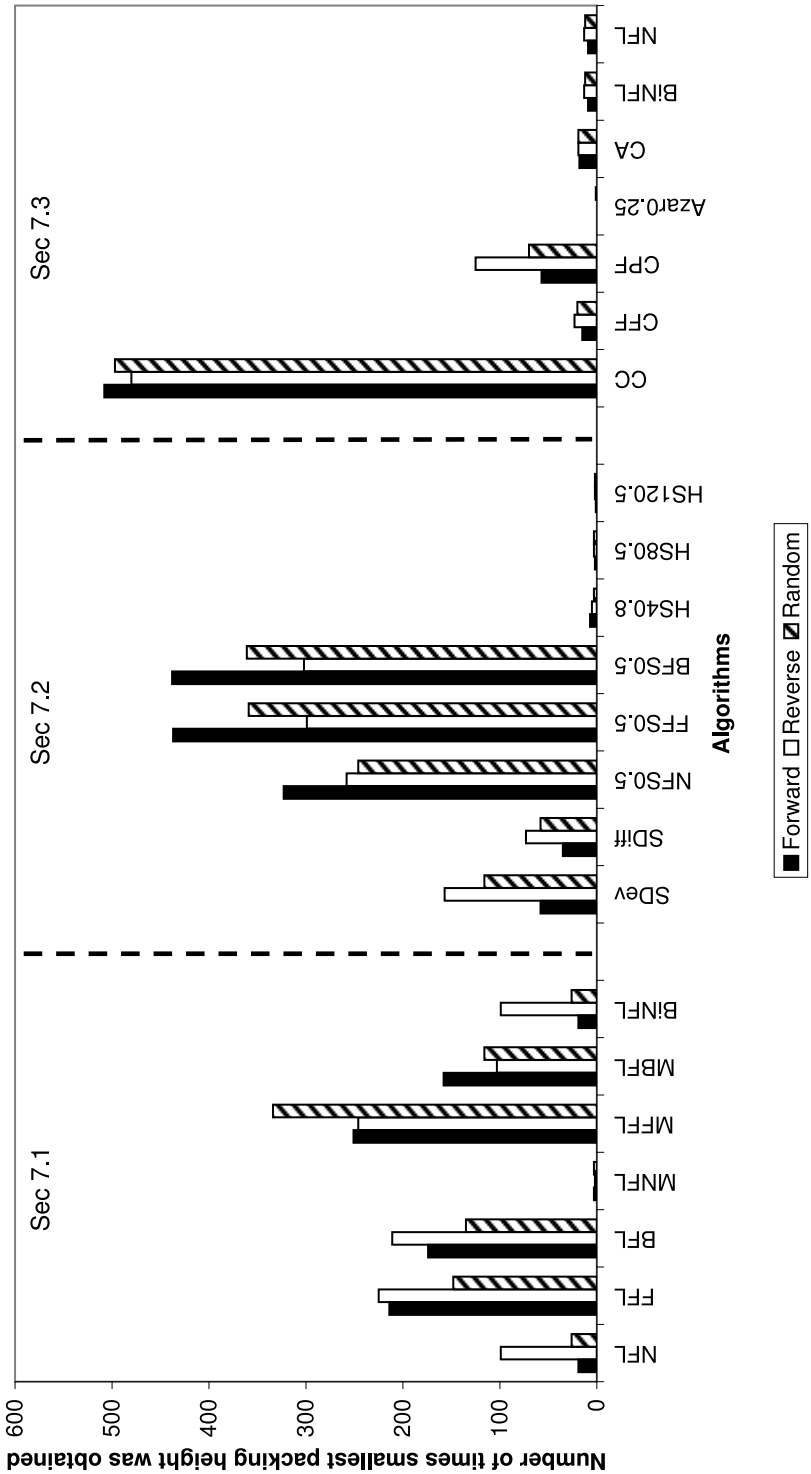


Figure 9: *Comparison of the number of times the smallest packing height was obtained by the various algorithms.*

$\mathrm{NFS}_{0.5}, \mathrm{FFS}_{0.5}$ and $\mathrm{BFS}_{0.5}$ algorithms achieved the best performance, followed by the new SDev and SDiff algorithms.

Considering the algorithms individually and comparing the mean packing heights obtained per traversal order, results from the ANOVA indicated that there were no significant differences, except with the SDev and SDiff algorithms. The results shown in Table 4 indicate that the two algorithms perform better in the *reverse* order. This was an expected result, because the SDev and SDiff algorithms rely on the first rectangle packed and ideally this rectangle must have the smallest height possible. As mentioned, the majority of the benchmark data sets in *reverse* order start with rectangles of relatively small height, hence leading to small increments of each shelf height with an overall smaller total packing height. The mean packing heights obtained by the $\mathrm{NFS}_{0.5}, \mathrm{FFS}_{0.5}$ and $\mathrm{BFS}_{0.5}$ algorithms were not expected to be similar, because a rectangle is classified according to its height, but depending on the widths of the rectangles that are packed first, it may sometimes be necessary to create an additional shelf of appropriate height due to insufficient space on existing shelves of appropriate height. The HS algorithmic instances, on other hand, were expected to yield similar mean packing heights regardless of the order, because the algorithm takes both height and width of the rectangles into consideration before packing on a level.

The results of the chi-squared test indicated that only the HS algorithmic instances display no significant difference with respect to the frequency with which they achieve the smallest packing heights, as illustrated in Table 4 (columns 14–16). The shelf algorithms with parameter r achieve the largest frequency, followed by the SDev and SDiff algorithms (see Figure 9). Based on the results in Table 4 the SDev and SDiff algorithms require shorter execution times than the known shelf algorithms from the literature. A threshold value of 0.456 was computed for the class of shelf algorithms. The $\mathrm{FFS}_{0.5}$ algorithm is recommended for use when dealing with data sets with a CV value larger than this threshold.

7.3 Special case algorithms obeying the tetris constraint

Because the Azar_Y algorithm depends on the threshold constant $0 < Y < 1/2$, three representative values $Y = 0.2, 0.25, 0.3$ were selected in order to determine only one value that may be used for further comparisons with other algorithms obeying the tetris constraint. An ANOVA was carried out and the results revealed that there were no significant differences between the mean packing heights obtained by these three algorithmic instances. The $\mathrm{Azar}_{0.25}$ algorithm was selected, because upon carrying out a chi-squared test, significant differences were found between the frequencies in obtaining the smallest packing heights, showing that the $\mathrm{Azar}_{0.25}$ algorithm achieved the largest frequency (297).

The results shown in the third section of Figure 7 indicate that the newly proposed CC algorithm obtained the smallest mean packing height in the class of algorithms obeying the tetris constraint. An ANOVA was carried out separately for each algorithm to decide whether the traversal order in which rectangles enter the system affects the performance of an algorithm. The results shown in Table 4 indicate that there are no significant differences between mean packing heights obtained per traversal order by each algorithm.

Forward Order

	Height	$NFS_{0.2}$	$NFS_{0.5}$		Height	$FFS_{0.2}$	$FFS_{0.5}$		Height	BFS$_{0.2}$	BFS$_{0.5}$		Height	HS$_{4_{0.2}}$	HS$_{4_{0.5}}$		Height	HS$_{8_{0.2}}$	HS$_{8_{0.5}}$		Height	HS$_{12_{0.2}}$	HS$_{12_{0.5}}$
NFS$_{0.2}$	414.0018			FFS$_{0.2}$	396.7122			BFS$_{0.2}$	394.7841			HS$_{4_{0.2}}$	948.1646			HS$_{8_{0.2}}$	773.7797			HS$_{12_{0.2}}$	845.9694		
NFS$_{0.5}$	284.5318	✓		FFS$_{0.5}$	276.5798	✓		BFS$_{0.5}$	276.4986	✓		HS$_{4_{0.5}}$	608.5404	✓		HS$_{8_{0.5}}$	533.5324	✓		HS$_{12_{0.5}}$	594.578	✓	
NFS$_{0.8}$	336.8876	✓	×	FFS$_{0.8}$	333.8336	✓	✓	BFS$_{0.8}$	333.805	✓	✓	HS$_{4_{0.8}}$	561.6116	✓	×	HS$_{8_{0.8}}$	549.6021	✓	×	HS$_{12_{0.8}}$	623.055	✓	×

Reverse Order

	Height	NFS$_{0.2R}$	NFS$_{0.5R}$		Height	FFS$_{0.2R}$	FFS$_{0.5R}$		Height	BFS$_{0.2R}$	BFS$_{0.5R}$		Height	HS$_{4_{0.2R}}$	HS$_{4_{0.5R}}$		Height	HS$_{8_{0.2R}}$	HS$_{8_{0.5R}}$		Height	HS$_{12_{0.2R}}$	HS$_{12_{0.5R}}$
NFS$_{0.2R}$	413.6771			FFS$_{0.2R}$	400.8192			BFS$_{0.2R}$	398.7085			HS$_{4_{0.2R}}$	947.507			HS$_{8_{0.2R}}$	772.8757			HS$_{12_{0.2R}}$	846.5967		
NFS$_{0.5R}$	284.4645	✓		FFS$_{0.5R}$	278.5429	✓		BFS$_{0.5R}$	277.9848	✓		HS$_{4_{0.5R}}$	608.5404	✓		HS$_{8_{0.5R}}$	533.3796	✓		HS$_{12_{0.5R}}$	594.4944	✓	
NFS$_{0.8R}$	337.1546	✓	×	FFS$_{0.8R}$	334.4034	✓	✓	BFS$_{0.8R}$	334.3017	✓	✓	HS$_{4_{0.8R}}$	561.2164	✓	×	HS$_{8_{0.8R}}$	549.7103	✓	×	HS$_{12_{0.8R}}$	624.0787	✓	×

Random Order

	Height	NFS$_{0.2Ra}$	NFS$_{0.5Ra}$		Height	FFS$_{0.2Ra}$	FFS$_{0.5Ra}$		Height	BFS$_{0.2Ra}$	BFS$_{0.5Ra}$		Height	HS$_{4_{0.2Ra}}$	HS$_{4_{0.5Ra}}$		Height	HS$_{8_{0.2Ra}}$	HS$_{8_{0.5Ra}}$		Height	HS$_{12_{0.2Ra}}$	HS$_{12_{0.5Ra}}$
NFS$_{0.2Ra}$	414.2122			FFS$_{0.2Ra}$	395.8708			BFS$_{0.2Ra}$	395.7897			HS$_{4_{0.2Ra}}$	947.6693			HS$_{8_{0.2Ra}}$	773.9753			HS$_{12_{0.2Ra}}$	847.1502		
NFS$_{0.5Ra}$	284.2634	✓		FFS$_{0.5Ra}$	276.922	✓		BFS$_{0.5Ra}$	276.923	✓		HS$_{4_{0.5Ra}}$	608.6096	✓		HS$_{8_{0.5Ra}}$	533.6707	✓		HS$_{12_{0.5Ra}}$	594.6587	✓	
NFS$_{0.8Ra}$	335.5847	✓	×	FFS$_{0.8Ra}$	333.832	✓	✓	BFS$_{0.8Ra}$	333.693	✓	✓	HS$_{4_{0.8Ra}}$	561.4988	✓	×	HS$_{8_{0.8Ra}}$	549.9066	✓	×	HS$_{12_{0.8Ra}}$	624.1357	✓	×

Table 3: *LSD results for shelf algorithms for the values* $r = 0.2, 0.5, 0.8$ *and* $M = 4, 8, 12$.

		Analysis of variance					Chi-squared test					Time		
		Fwd	**Rev**	**Rand**	F_{calc}	F_{crit}	**Frd**	**Rev**	**Rand**	χ^2_{calc}	χ^2_{crit}	**Fwd**	**Rev**	**Rand**
Level algorithms	NFL	332.627	343.823	362.742	1.583	3.001	19	99	26	**81.792**	**5.990**	13.465	9.771	14.554
	FFL	279.426	310.039	320.633	**4.276**	**3.001**	214	225	148	**17.727**	**5.990**	21.077	17.647	21.216
	BFL	282.215	312.634	322.501	**4.143**	**3.001**	174	211	135	**16.665**	**5.990**	21.899	17.472	20.823
	MNFL	425.084	465.227	473.743	**3.402**	**3.001**	3	2	3	0.25	5.99	13.044	9.771	12.423
	MFFL	271.75	305.739	297.316	**3.228**	**3.001**	251	246	334	**17.639**	**5.990**	20.456	16.93	19.454
	MBFL	278.584	323.638	318.585	**6.176**	**3.001**	158	103	116	**13.151**	**5.990**	21.055	17.619	20.07
	BiNFL	332.627	343.833	362.742	1.583	3.001	19	99	26	**81.792**	**5.990**	11.373	11.056	12.432
Shelf algorithms	SDev	437.459	313.592	342.36	**23.409**	**3.001**	58	157	116	**44.852**	**5.990**	10.742	10.103	11.478
	SDiff	443.299	328.616	356.738	**18.809**	**3.001**	35	73	58	**13.241**	**5.990**	11.188	11.623	13.066
	$NFS_{0.5}$	284.532	284.464	284.263	0	3.001	323	258	246	**12.452**	**5.990**	15.86	16.369	17.426
	$FFS_{0.5}$	276.58	278.543	276.922	0.007	3.001	437	299	359	**26.236**	**5.990**	18.406	16.285	18.096
	$BFS_{0.5}$	276.499	277.985	276.923	0.004	3.001	438	302	361	**25.346**	**5.990**	16.686	15.199	16.887
	$HS_{4_{0.8}}$	561.612	561.216	561.499	0	3.001	7	5	3	1.6	5.99	15.246	14.769	15.544
	$HS_{8_{0.5}}$	533.532	533.38	533.671	0	3.001	2	3	3	0.25	5.99	12.773	12.301	13.347
	$HS_{12_{0.5}}$	594.578	594.494	594.659	0	3.001	1	2	2	0.4	5.99	12.906	13.38	14.007
Special case alg.	CC	249.664	256.172	271.036	1.111	3.001	508	480	497	0.804	5.99	11.236	11.506	9.29
	CFF	269.568	278.014	296.136	1.618	3.001	15	23	20	1.69	5.99	10.86	10.946	9.909
	CPF	306.458	311.205	329.928	1.147	3.001	57	125	70	**31.024**	**5.990**	11.799	12.707	11.887
	$Azar_{0.25}$	580.181	581.475	609.721	0.897	3.001	0	0	1	2	5.99	12.847	13.556	15.495
	CA	331.367	342.191	361.416	1.648	3.001	18	19	19	0.036	5.99	11.12	11.327	12.673
	BiNFL	332.627	343.644	362.742	1.583	3.001	9	13	12	0.765	5.99	11.373	12.198	12.432
	NFL	332.627	343.823	362.742	1.583	3.001	9	13	12	0.765	5.99	13.465	11.056	14.554

Table 4: *Results from the ANOVA, chi-square test and execution times of the level, shelf and special case algorithms obeying the* tetris *constraint. Bold faced entries indicate that significantly different results are achieved for different traversal orders of the data sets.*

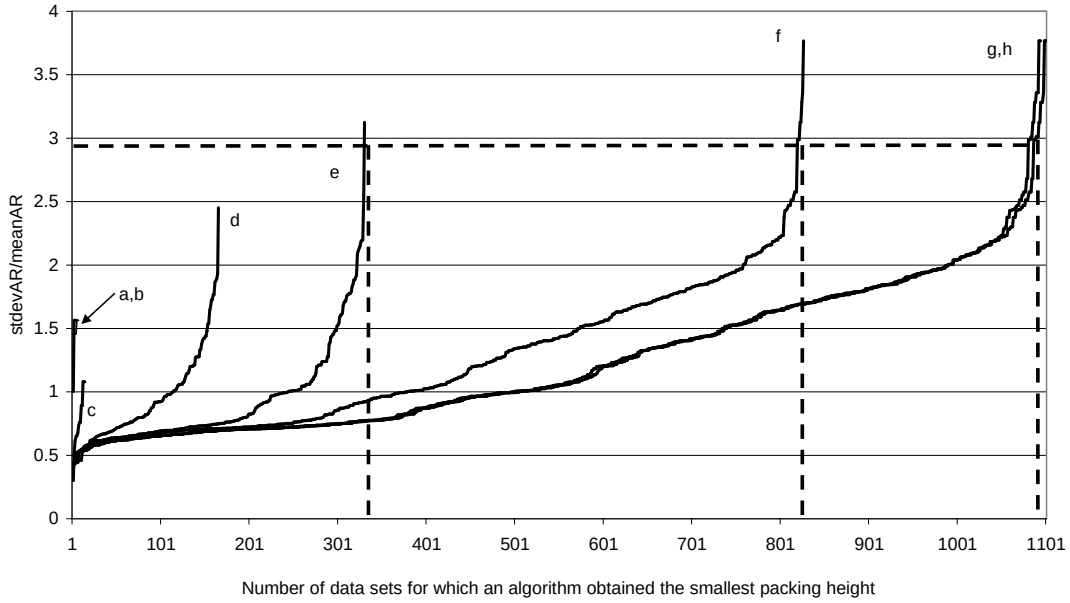


Figure 10: *Aspect ratio analysis for shelf algorithms:* a – $\mathrm{HS}_{8_{0.5}}$, b – $\mathrm{HS}_{12_{0.5}}$, c – $\mathrm{HS}_{4_{0.8}}$, d – *SDiff*, e – *SDev*, f – $\mathrm{NFS}_{0.5}$, g – $\mathrm{FFS}_{0.5}$ *and* h – $\mathrm{BFS}_{0.5}$.

Comparing the frequencies of obtaining the smallest packing height separately for each algorithm, the results of the chi-squared test (see Table 4) showed that only the CPF algorithm is affected by the order in which rectangles enter the system, achieving the largest frequency in the *reverse* traversal order.

When comparing all the algorithms obeying the tetris constraint, the results of the ANOVA indicate that there are significant differences in terms of their mean packing heights obtained by the various algorithms over all 542 test instances. The results from the LSD (see Table 2) suggest that the newly proposed CC and CFF algorithms are the best performing algorithms with no distinguishable difference between the mean packing heights obtained. However, in terms of the frequency of obtaining the smallest packing height, the two algorithms are distinguishable with the CC algorithm achieving the largest frequency, as may be seen in the results of the chi-squared test.

A CV threshold value of 0.443 was computed, implying that for data sets with CV values smaller than the threshold, any of the special case algorithms may be used. However, for CV values larger than the threshold, the CC algorithm is recommended.

8 Final Remarks

We have investigated a number of on-line algorithms from the literature and classified them into the three classes of *level*, *shelf* and *special case* algorithms. For each class, we were able to find a threshold value for the *coefficient of variation* (CV) such that, given a data set with CV value above this threshold, certain heuristics are recommended above

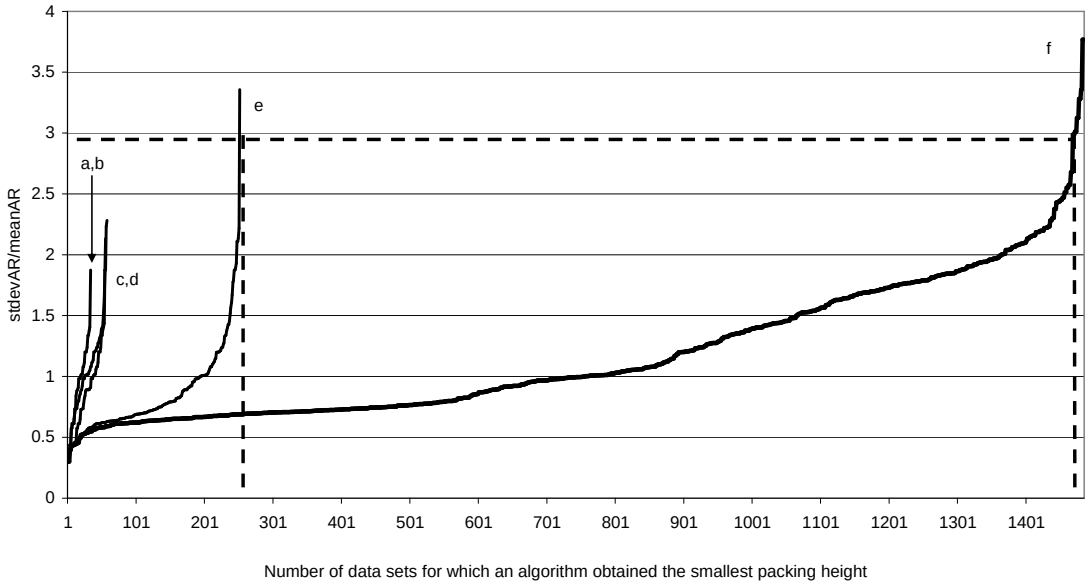


Figure 11: *Aspect ratio analysis for special case algorithms: a – BiNFL, b – NFL, c – CCF, d – CA, e – CPF and f – CC.*

others. In particular, for level (resp. shelf) algorithms, data sets with CV values beyond 0.438 (resp. 0.456), the MFFL (resp. $\text{FFS}_{0.5}$) algorithm is recommended. For special case algorithms obeying the *Tetris* constraint, the CC algorithm is recommended for CV values beyond a threshold value of 0.443.

Two new shelf algorithms (SDev and SDiff) were introduced with an entirely different way of generating additional space within shelves. Instead of using a parameter value (as in some of the classical shelf algorithms from the literature), the new algorithms use the history of the rectangles packed to determine how much free space to create. In the SDev algorithm, the standard deviation of the heights of the rectangles already packed is used while in the SDiff algorithm, the difference in height between the previous shelf and the rectangle to be packed is used. The advantage of the new algorithms is that they do not rely on the selection of any parameter value which, if badly chosen, may lead to poor performance of the algorithm. The new algorithms achieve a better performance than the HS_{M_r} algorithm and can even perform better than the NFS_r, FFS_r and BFS_r algorithms for certain values of the parameter r.

Three modifications (CPF, CFF and CC algorithms) to the CA algorithm [9] have also been proposed, which take more patterns into consideration. When tested on benchmark data sets, the CC algorithm obtained the smallest packing height with the highest frequency.

Finally, it is worth mentioning that in terms of execution time, all algorithms were able to provide a solution to any benchmark instance within 1 second on a 2.00 GHz processor with 224 MB of RAM.

Acknowledgments

Work towards this paper was supported financially by the Third World Organisation for Women in Science (TWOWS), by the South African National Research Foundation (Gun 2072815) and by the Harry Crossley Foundation. We are grateful to Dr Werner Gründlingh and Mr John Part for respectively typesetting assistance and programming advice.

References

[1] Azar Y & Epstein L, 1997, *On two-dimensional packing*, Journal of Algorithms, **25(2)**, pp. 321–332.

[2] Baker BS & Schwarz JS, 1983, *Shelf algorithms for two-dimensional packing problems*, SIAM Journal on Computing, **12(3)**, pp. 508–525.

[3] Bartholdi III JJ, Vate JHV & Zhang J, 1989, *Expected performance of the shelf heuristic for 2-dimensional packing*, Operations Research Letters, **8**, pp. 11–16.

[4] Beasley JE, 1985, *Algorithms for unconstrained two-dimensional guillotine cutting*, Journal of the Operational Research Society, **36(4)**, pp. 297–306.

[5] Beasley JE, 1985, *An exact two-dimensional non-guillotine cutting tree search procedure*, Operations Research, **33(1)**, pp. 49–64.

[6] Brown DJ, Baker BS & Katseff HP, 1982, *Lower bounds for on-line two-dimensional packing algorithms*, Acta Informatica, **18**, pp. 207–225.

[7] Burke EK, Kendall G & Whitwell G, 2004, *A new placement heuristic for the orthogonal stock-cutting problem*, Operations Research, **52(4)**, pp. 655–671.

[8] Christofides N & Whitlock C, 1977, *An algorithm for two–dimensional cutting problems*, Operations Research, **25(1)**, pp. 31–44.

[9] Coffman Jr. EG, Downey PJ & Winkler P, 2002, *Packing rectangles in a strip*, Acta Informatica, **38**, pp. 673–693.

[10] Coffman Jr. EG, Garey DS & Tarjan RE, 1980, *Performance bounds for level oriented two-dimensional packing algorithms*, SIAM Journal on Computing, **9(4)**, pp. 808–826.

[11] Coffman Jr. EG & Shor PW, 1990, *Average-case analysis of cutting and packing in two dimensions*, European Journal of Operational Research, **44**, pp. 134–144.

[12] Coffman Jr. EG & Shor PW, 1993, *Packings in two dimensions: Asymptotic average-case analyisis of algorithms*, Algorithmica, **9**, pp. 253–277.

[13] Coppersmith D & Raghavan P, 1989, *Multidimensional on-line bin packing: Algorithms and worst-case analysis*, Operations Research Letters, **8**, pp. 17–20.

[14] Csirik J & Woeginger GJ, 1997, *Shelf algorithms for on-line strip packing*, Information Processing Letters, **63**, pp. 171–175.

[15] Downey PJ, 2006, *Personal communication via e-mail*, 2006 March 23, contactable at `pete@cs.arizona.edu`.

[16] Hopper E & Turton BCH, 2002, *Problem generators for rectangular packing problems*, Studia Informatica Universalis, **2(1)**, pp. 123–136.

[17] Hopper E & Turton BCH, 2001, *An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem*, European Journal of Operational Research, **128(1)**, pp. 34–57.

[18] Imreh C, 2001, *Online strip packing with modifiable boxes*, Operations Research Letters, **29**, pp. 79–85.

[19] Lee CC & Lee DT, 1985, *A simple on-line bin packing algorithm*, Journal of the Association of Computing Machinery, **32(3)**, pp. 562–572.

[20] Miyazawa FK & Wakabayashi Y, 1998, *Parametric on-line packing*, In Anais do XXX Simpsio Brasileiro de Pesquisa Operacional / Workshop da III Oficina Nacional de Problemas de Corte and Empacotamento, Curitiba-Pr, pp. 109–121.

[21] Mumford-Valenzuela C, Wang PY & Vick J, 2001, *Heuristic for large strip packing problems with guillotine patterns: An empirical study*, Proceedings of the 4th Metaheuristics International Conference, pp. 417–421.

[22] Ntene N & Van Vuuren JH, 2008, *A survey and comparison of guillotine heuristics for the 2D oriented offline strip packing problem*, Submitted.

[23] Ortmann FG, Ntene N & van Vuuren JH, 2008, *New and improved level heuristics for the rectangular strip packing and variable-sized bin packing problems*, European Journal of Operational Research (Submitted).

[24] Ramanan P, Brown DJ, Lee CC & Lee DT, 1989, *On-line bin packing in linear time*, Journal of Algorithms, **10**, pp. 305–326.

[25] Smith H, 2006, *Strip packing algorithms*, [Online], [cited 2006, March 15], Available from: `http://users.cs.cf.ac.uk/C.L.Mumford/ heidi/Algorithms.html`

Appendix

Algorithm 1: Shelf Deviation and Shelf Difference algorithms
Input: Dimensions of the rectangles $\langle w(L_i), h(L_i)\rangle$ and the strip width W.
Output: The height H of the packing obtained in the strip.

$h(type_{0,1}) \leftarrow 0$, $h(type_{1,1}) \leftarrow h(L_1)$, $H \leftarrow h(type_{1,1})$
$w(type_{1,1}) \leftarrow W - w(L_1)$
$i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow 1$, NumTypes $\leftarrow 1$, NumShelfType$_1 \leftarrow 1$
while there is a rectangle to be packed **do**
  $i \leftarrow i + 1$ (going to the next rectangle)
  while $j \leq$ NumTypes Or rectangle is not packed **do**
    $k \leftarrow 1$
    if $h(type_{j-1,k}) < h(L_i) \geq h(type_{j,k})$ **then**
      while $k \leq$ NumTypes$_j$ **do**
        if $w(type_{j,k}) \geq w(L_i)$ **then**
          pack rectangle
        else $\{w(type_{j,k}) < w(L_i)\}$
          $k \leftarrow k + 1$ (move on to the next shelf of the same type)
        end if
      end while
      if $k >$ NumShelfType$_j$ **then**
        NumShelfType$_j \leftarrow$ NumShelfType$_j + 1$ (increase the number of shelves of this particular type)
        $w(type_{j,k}) = W - w(L_i)$
        $H \leftarrow H + h(type_{j,k})$
      end if
    else $\{h(type_{j-1,k}) \geq h(L_i)$ or $h(L_i) < h(type_{j,k})\}$
      $j \leftarrow j + 1$ (move on to the next type)
    end if
  end while
  if $j >$ NumTypes **then**
    create a new shelf type
    NumTypes $\leftarrow$ NumTypes $+ 1$, $k \leftarrow 1$
    $proportion \leftarrow stdev(h(L_1), \ldots, h(L_i))$ **SDev algorithm**
    $proportion \leftarrow (h(L_i) - h(type_{j-1,k}))$ **SDiff algorithm**
    $h(type_{j,k}) \leftarrow proportion + h(L_i)$
    $H \leftarrow H + h(type_{j,k})$
  end if
end while