OPERATIONS RESEARCH
Vol. 52, No. 4, July–August 2004, pp. 655–671
ISSN 0030-364X | EISSN 1526-5463 | 04 | 5204 | 0655

DOI 10.1287/opre.1040.0109
© 2004 INFORMS

A New Placement Heuristic for the Orthogonal Stock-Cutting Problem

E. K. Burke, G. Kendall, G. Whitwell
School of Computer Science and Information Technology, University of Nottingham, Jubilee Campus, Nottingham, NG8 1BB, United Kingdom {ekb@cs.nott.ac.uk, gxk@cs.nott.ac.uk, gxw@cs.nott.ac.uk}

This paper presents a new best-fit heuristic for the two-dimensional rectangular stock-cutting problem and demonstrates its effectiveness by comparing it against other published approaches. A placement algorithm usually takes a list of shapes, sorted by some property such as increasing height or decreasing area, and then applies a placement rule to each of these shapes in turn. The proposed method is not restricted to the first shape encountered but may dynamically search the list for better candidate shapes for placement. We suggest an efficient implementation of our heuristic and show that it compares favourably to other heuristic and metaheuristic approaches from the literature in terms of both solution quality and execution time. We also present data for new problem instances to encourage further research and greater comparison between this and future methods.

Subject classifications: production/scheduling: cutting stock/trim; production/scheduling: approximations/heuristic; computers/computer science: artificial intelligence.
Area of review: Optimization.
History: Received May 2002; revision received January 2003; accepted June 2003.

1. Introduction

The field of cutting and packing motivates many areas of operations research. The focus of this paper is the two-dimensional orthogonal stock-cutting problem. It has applications to areas such as dynamic memory allocation, multiprocessor scheduling problems, and general layout problems (Coffman et al. 1978, Garey and Johnson 1981, Coffman and Leighton 1989, Dyckhoff 1990). These problems have a similar logical structure and can be modelled by a set of rectangular pieces that must be arranged on a predefined stock sheet so that each rectangular piece does not overlap with another. It occurs, with different constraints, within manufacturing industries including paper, wood, glass, and metal cutting. For example, paper cutting is generally concerned with the guillotine packing (where only vertical or horizontal straight cuts across the entire sheet region are allowed) of rectangular items from a stock roll of fixed width, whereas applications in metal and shipbuilding are often concerned with the cutting of irregular shapes from a stock sheet. However, in most industrial applications the goals are similar: to produce good quality arrangements of items on the stock sheet in order to maximise material utilisation and, therefore, minimise wastage. The time allowed for any specific problem is usually dependent on the material cost and the urgency of a solution. For example, when packing onto a sheet with a two-inch thickness you would probably allow the packing algorithm more time, as a small improvement in the solution quality can result in large cost savings. With a sheet of one millimeter thickness, there may be more emphasis in finding solutions quickly, as small reductions in packing quality may only yield negligible savings. These processes are most heavily associated with mass-production operations, and it is usually very important to produce better-quality solutions, in less time, with less wastage in order to maximise profits. In some industrial situations, this optimisation task is still undertaken by skilled experts. However, due to the large costs, performance falloff, and the liability inherent with employed labour, automated-packing approaches have been more widely used in recent years. The solution quality of automated-packing approaches can often be equal or better than that of their human counterparts (Roberts 1984, Li and Milenkovic 1995) and are usually performed more quickly (Hower et al. 1996). There have been many approaches to producing automated-packing algorithms. These range from mathematical linear programming approaches to problem-specific heuristic algorithms and, more recently, the application of metaheuristic methodologies (Dowsland and Dowsland 1992, 1995; Hopper and Turton 2001).

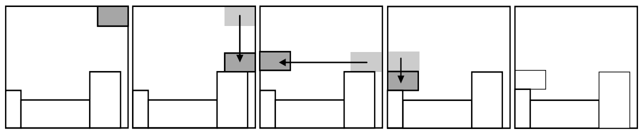
This paper addresses the problem of placing rectangles onto a larger rectangular object in order to minimise the height of the nest. We allow nonguillotine packings in which we are not restricted to only performing full horizontal or vertical cuts from one sheet edge to another (unlike guillotine packing). All items in the set of rectangles must be packed onto the object sheet, and we allow rotations of 90 degrees.

The two-dimensional variant of the stock-cutting problem is NP hard due to the combinatorial explosion encountered as the problem size increases (Garey and Johnson 1979). This area has been the subject of several decades of research from its formulation in the 1950s, where Paull (1956) addressed the newsprint layout problem. Due to the extensive nature of this research field and its diverse problem instances, many researchers have provided overviews and categorised bibliographies (Dowsland and Dowsland 1992, Sweeny and Paternoster 1992, Dyckhoff 1990, Coffman et al. 1984, Golden 1976, Gilmore 1966). The approaches can be broadly categorised into three methods: exact, heuristic, and metaheuristic.

Exact methods were investigated by Gilmore and Gomory (1961) in what is considered to be the first real industry-applicable research into stock cutting. They used linear programming techniques to solve problem instances to optimality. However, only small problem instances could be solved due to the computational time required. Christofides and Whitlock (1977) used a tree-search method to solve the two-dimensional guillotine stock-cutting problem to optimality, as does Beasley (1985a) with the nonguillotine variant. However, once again, with larger instances the method becomes time infeasible. Beasley (1985b) compares both optimal and heuristic algorithms using dynamic programming. Hifi and Zissimopolous (1997) presented an exact algorithm that improves on the approach used by Christofides and Whitlock. Recently, Cung et al. (2000) developed a new version of the algorithm proposed in Hifi and Zissimopolous (1997) that uses a best-first branch-and-bound approach to solve exactly some variants of two-dimensional stock-cutting problems. A major drawback of these methods is that they cannot provide good results for large instances of the problem (see our comparison and discussion in §4.3.1). Heuristic methods are required to provide good, although, of course, not necessarily optimal solutions.

For such instances, there have been many heuristic approaches that have been formulated to produce good packings in an acceptable time frame, even with large stock-cutting cases. Albano and Orsini (1979) used a heuristic method that packs similar rectangles into strips for large-problem instances ranging between 400 to 4,000 rectangles. Bengtsson (1982) presented a heuristic solution that achieved trim loss of between 2% and 5% by sorting rectangles and then arranging them into piles. The most documented heuristic approaches are the bottom-left (BL) and bottom-left-fill (BLF) methods (Baker et al. 1980, Chazelle 1983). Jakobs (1996) used a bottom-left method that takes as input a list of rectangles and places each one in turn onto the stock sheet. The placement strategy first places the rectangle in the top-right location and makes successive moves of sliding it as far down and left as possible (Figure 1). Lui and Teng (1999, Figure 2) developed an improved bottom-left heuristic, giving downward movement priority so that shapes only slide leftwards if no downwards movement is possible. It was shown that, unlike Jakobs' method, there was always at least one rectangle sequence that could be decoded into the optimal solution.

Figure 1. A bottom-left method (Jakobs 1996).



The second method, bottom-left-fill, is a modified version of the bottom-left placement heuristic. One implementation maintains a list of location points in a bottom-left ordering to indicate where the shapes may be placed. When placing a shape, the algorithm starts with the lowest and leftmost point, places the shape and left justifies it, then checks whether the shape would overlap with any other shape and that it stays within the confines of the sheet. If it does not overlap, the shape is placed and the point list is updated to indicate any new points. If the shape would overlap, the next point in the point list is selected until the shape can be placed without overlap occurring. Figure 3 shows an example of available points for placement.

In Figure 3, when placing a fifth shape the algorithm tries to place at the bottommost point first (with points at the same height being resolved by leftmost first). Therefore, bottom-left-fill can overcome the problem of holes by the storage of possible location points. We can show the difference between the two placement heuristics by showing the stages undertaken when adding a fifth shape to the above packing (Figure 4).

Figure 4 shows that bottom-left-fill is able to fill holes by using fitting rectangles later in the packing, whereas using bottom-left results in a hole in the final packing. This hole will never be filled (even if a later rectangle would fit), therefore the space can be considered to be wastage. An important aspect of these algorithms is that the sequence of rectangles supplied can greatly influence the quality of the solution. In experiments between the algorithms, Hopper and Turton (2001) found that bottom-left-fill outperformed bottom-left by up to 25%, and that preordering the shapes by decreasing widths or decreasing heights for both algorithms increased the packing quality by up to 10% compared to random sequences. The main advantage for using bottom-left is its time complexity of $O(N^2)$ (Hopper and Turton 2001). The bottom-left-fill algorithm is disadvantaged by its worse time complexity of $O(N^3)$ (Chazelle 1983). Consequently, execution times can

Figure 2. An improved bottom-left method (Liu and Teng 1999).

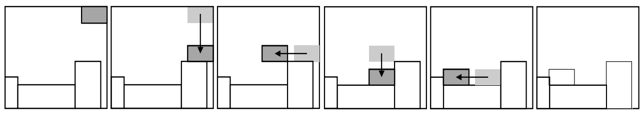
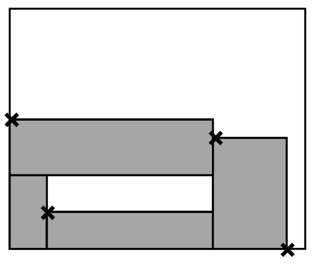


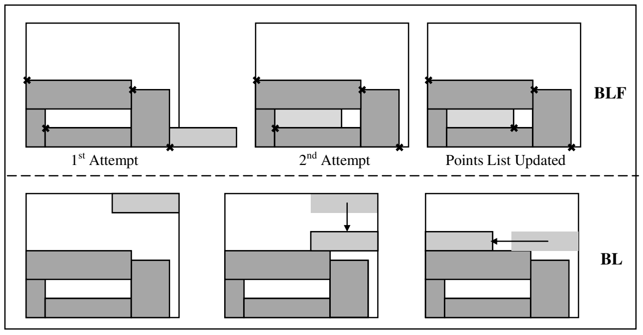
Figure 3. Storing placement locations for one implementation of bottom-left-fill.



become large for bottom-left-fill with increasing problem sizes. An overriding feature of these approaches is that they pack each shape according to a set of rules. Therefore, the solution will be the same if supplied with an identical rectangle input sequence. They attempt to produce good-quality packings in a single run, reducing the time required to achieve solutions.

A recent trend has been to utilise metaheuristic approaches in producing packings for the problem. These are usually hybridised algorithms involving the generation of input sequences that are then interpreted by placement heuristics such as bottom-left or bottom-left-fill (Jakobs 1996, Ramesh Babu and Ramesh Babu 1999, Hopper and Turton 2001). However, alternative methods have been developed that place all shapes on the stock sheet and then apply small movements to the shapes to maximise the packing density and minimise the penalty function for overlapping shapes (Dagli and Hajakbari 1990, Dowsland and Dowsland 1992). Dagli and Hajakbari performed one of the first applications of metaheuristics to this problem. They used a simulated annealing algorithm to pack rectangular and irregular shapes onto a sheet of finite dimensions (Dagli and Hajakbari 1990). Other work has since appeared on the applications of simulated annealing to the stock-cutting problem. Lai and Chan (1996) used a simulated annealing algorithm. They test their algorithm with real data from a printing company and report that their algorithm performs well with problems where less than 15 rectangles need to be packed. Faina (1999) applies simulated annealing to both the guillotine and nonguillotine varieties of the problem and shows that for larger problems, nonguillotine outperforms guillotine with little extra computation overhead. Genetic algorithms have also been extensively studied for this problem over the last decade. Kroger (1995) used genetic algorithms for the guillotine variant of bin packing and introduced the idea of "metarectangles," which are groupings of rectangles that can pack well together and can be treated as one single rectangle. Jakobs (1996) uses a genetic algorithm for the packing of polygons using rectangular enclosures and a bottom-left heuristic. Ramesh Babu and Ramesh Babu (1999) detail improvements on the genetic method proposed by Jakobs. It is able to pack onto many finite sheets by including an ordering of sheets within the chromosome of each population member. Hopper and Turton (1999) evaluate the use of the bottom-left-fill heuristic with genetic algorithms on the nonguillotine rectangle-nesting problem. They found that a genetic algorithm with bottom-left-fill placement outperformed a genetic algorithm with bottom-left placement. Other problems where evolutionary methods have been applied include bin packing (Falkenauer 1996) and the guillotinable variant of the orthogonal stock-cutting problem (Valenzuela and Wang 2001). Dagli and Poshyanonda (1997) compare the nesting of rectangular patterns using two methods involving artificial neural networks. The first neural network method was trained through back propagation. The second method involved a neural network/genetic algorithm combination. Their methods produced an average waste of 7.88%. Hopper and Turton (2001) compare several metaheuristics including genetic algorithms, simulated annealing, naïve evolution, hill climbing, and random searches. Genetic algorithms, simulated annealing, and naïve evolution (with bottom-left-fill decoder) all gave similar results, and the authors show that they are better than using the bottom-left-fill heuristic with a height- or width-sorted input sequence, although extra computation time is required.

Figure 4. A comparison of the BL and BLF methods when adding a rectangle.



In general, metaheuristic algorithms represent the more sophisticated heuristic methods. As they operate, they generate a number of different solutions; i.e., the metaheuristic algorithms are guided through the problem's search space by previous attempts. However, there are usually search variables that must be tuned to achieve best performance.

2. A New Heuristic Algorithm for the Problem (Best Fit)

2.1. An Overview

Unlike the bottom-left and bottom-left-fill methods that make placements that are based on the sequence of rectangles supplied to them, our proposed method dynamically selects the next rectangle for placement during the packing stage. This enables the algorithm to make informed decisions about which rectangle should be packed next and where it should be placed. We adopt a *best-fit* type strategy. This is, essentially, a greedy algorithm that attempts to produce good-quality packings by examining the lowest available space within the stock sheet and then placing the rectangle that best fits the space available. There are some inherent problems with respect to the time complexity of the algorithm and the quality of the solution. In order to find the lowest available *gap*, we need to examine the stock sheet and current assignment of shapes. Once the lowest gap is found, we must examine the list of rectangles to find the best-fitting shape. Both of these operations must be conducted at each step of the algorithm, and therefore they contribute to increasing the time complexity. We address methods that can be employed to reduce this complexity bottleneck in §2.2.

At the beginning of the packing process there will not be any shapes assigned to locations on the stock sheet. Therefore, the lowest available gap or "niche" for placement of a shape will be the entire width of the stock sheet. As shapes are placed, the *lowest gap* will change with respect to both location and width. We always select and place the best-fitting shape. There are three possibilities: (i) There is a shape with a dimension (either width or height) that exactly fits the gap, (ii) there is a shape with a dimension smaller than the gap, or (iii) there are no shapes that will fit the gap. In the first case, the placement of the shape is easy because there is a perfect fit (where there are several shapes that would exactly fit, the rectangle with the largest area is chosen). However, in the second case the shape that consumes the largest portion of the gap is placed. As this shape does not completely fill the gap, we need to choose how to place the shape within the gap. We have called this the *niche-placement policy*. The third case gives rise to an important property of using the best-fit methodology. If none of the available rectangles can fit within the lowest gap, then we can regard the relevant space as wastage. This is clear to see because if none of the shapes fit the space now, then none of the remaining shapes will be able to fit in the space in future iterations. Unlike the bottom-left placement heuristic, this method only ever creates holes that cannot be filled, and unlike bottom-left-fill, all holes that are created can be "forgotten." This leads to another important "feature" of the best-fit heuristic. Many algorithms, such as bottom-left and bottom-left-fill, require a costly "overlap" function. This performs an overlap test between the current shape and each of the shapes that have previously been placed onto the sheet. Obviously, the more rectangles that have been packed, the more overlap tests we have to perform, thus resulting in the process becoming slower as each rectangle is placed. However, because of the best-fit approach and the implementation (presented in §2.2), we do not require this operation, as we are always sure that the shapes we are placing do not overlap with other rectangles.

Niche-Placement Policies. Our heuristic is a combination of three niche-placement policies, with each policy indicating how a shape could be placed when it does not fit exactly into the lowest niche. We never deliberately create a niche, but if no rectangle can fit the gap completely, there is no option but to create one. We assume that the stock-sheet sides are "infinitely tall," which can be thought of as corresponding to a roll of material (which is infinitely long!). Our three policies can be summarised as follows:

(a) *Place at Leftmost.* The leftmost niche-placement policy places a nonexact fitting rectangle at the left side of the niche. Note: If a rightmost niche-placement policy were used, the resultant packing would be a mirror image of the leftmost policy.

(b) *Place Next to Tallest Neighbour.* In this placement policy we examine the two rectangles on the stock sheet that define the lowest gap. We then place the best-fitting rectangle within the gap next to the tallest gap-defining rectangle (Figure 5). If the lowest gap is defined by a rectangle and the sheet side, we place next to the sheet side.

(c) *Place Next to Shortest Neighbour.* This placement policy is the opposite of the tallest-neighbour policy described above. This time, we place rectangles next to the shortest neighbour. An example is given in Figure 6.

Figure 5. Placement next to tallest neighbour.

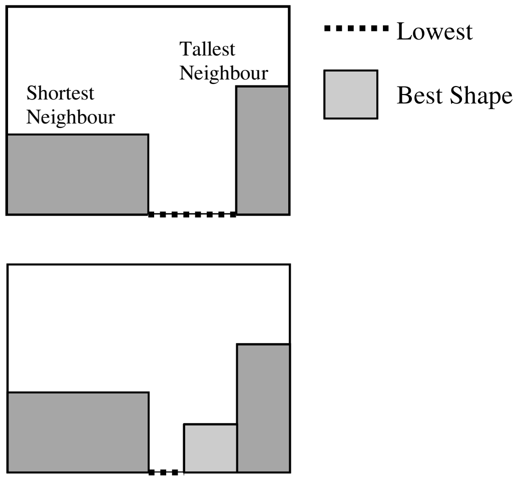
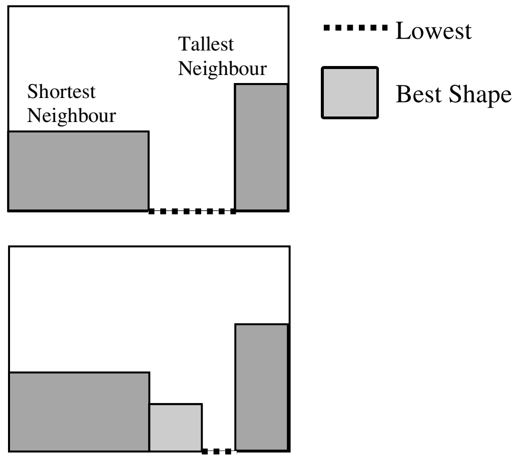


Figure 6. Placement next to shortest neighbour.



Improving the Packing. A drawback of using the proposed method is that it can create poorer-quality packings due to "towers." Towers are produced when long thin rectangles have not been placed until the latter stages of a packing. The algorithm may place these rectangles in *portrait* orientation near to the top of the nest, where they negatively affect the solution quality. Due to this difficulty, we conduct a further step after all rectangles have been placed. This searches for the shape that is currently adding the greatest height to the nest (i.e., a tower) and removes it from the packing. Then the shape is rotated by 90 degrees and replaced on top of the nest. If the solution quality is improved by this step, then we look for a new "tower" and perform the step again. We continue with this process until there is no improvement in solution quality. The procedure is described in more detail in the implementation section below.

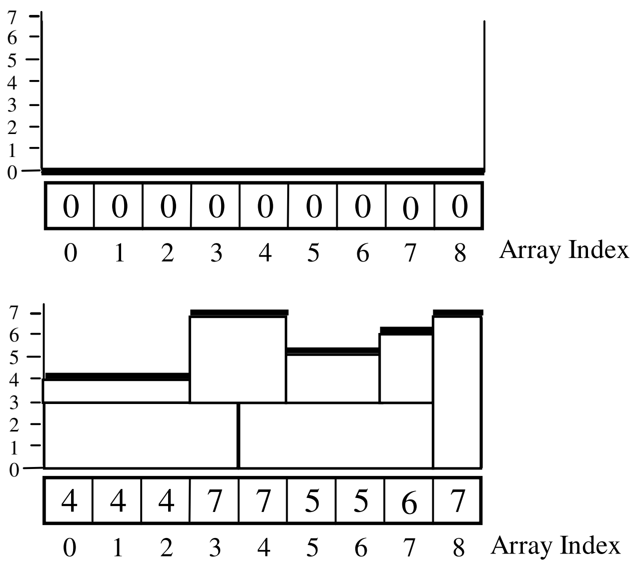
2.2. Implementation

The description of the algorithm above requires a search for both the best-fitting rectangle and the lowest space within the stock sheet at every time step of the algorithm. These are computationally expensive operations and the algorithm would benefit if we could simplify or remove them. The inputs to the algorithm are a list of rectangles in random order, and the width of the stock sheet.

Preprocessing Stage. First, we address the problem of representation. Some algorithms store the required information using location points as in the bottom-left-fill algorithms (Chazelle 1983) and/or collision detection by the sliding of shapes such as the bottom-left algorithms (Jakobs 1996, Liu and Teng 1999). Others avoid the problem by packing in rows with a new row starting from the highest point of the previous one (Bengtsson 1982). However, with these methods we require the use of free location lists that must be traversed and/or collision detections on each rectangle placement. Collision detection is computationally expensive due to the need to examine all of the already placed rectangles when adding a shape. Therefore, when adding the last piece in a problem of 500 shapes, we need to perform 499 collision tests to ensure that there are no overlaps. Our approach to this problem is to store the stock sheet as a linear array that has a number of elements equal to the width of the stock sheet. Each element of the array holds the total height of the packing at that x coordinate of the stock sheet. Two examples are given in Figure 7 to show a sheet of width nine units when empty, and the same sheet during packing.

Therefore, the coordinate of the lowest space of the stock sheet can be found by locating the smallest-valued entry of the array. The width of the gap can be found by examining how many consecutive array items of equal value exist. In the nonempty sheet in Figure 7, the lowest available space is located at $x = 0$ and has a width of three.

Figure 7. Storing the skyline of packings on a sheet of width nine units when empty, and filling.



The other problem we encounter is due to using the best-fit methodology. The rectangle data is defined as a list of rectangles each denoted by a (width, height) pair. In an unsorted list we must examine all n rectangles to be sure that there is not a "better"-fitting rectangle at each rectangle placement. However, we can sort the list of rectangles once before packing commences so that we reduce the number of rectangles we need to examine to $(1/2)n$ (on average per-rectangle placement). The first stage of this restructuring is to rotate any rectangle for which the height is greater than the width. For example:

$$\{(3,5),(5,2),(1,1),(7,3),(1,2)\} \text{ becomes } \{(5,3),(5,2),(1,1),(7,3),(2,1)\}.$$

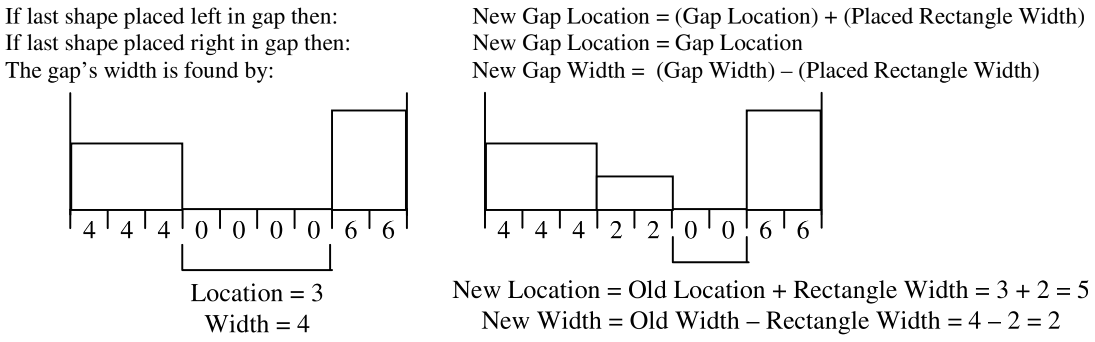
Next, the list of rectangles is sorted into decreasing width order (resolving equal widths by decreasing heights):

$$\{(5,3),(5,2),(1,1),(7,3),(2,1)\} \text{ becomes } \{(7,3),(5,3),(5,2),(2,1),(1,1)\}.$$

This list of rectangles can now be examined for the best-fitting rectangle without the need to search the entire list. For example, suppose we require a shape to fill a gap of six units. The first rectangle in the list is examined, $(7,3)$. Note that it could fill three units of the gap if rotated. The second rectangle in the list, $(5,3)$, can occupy a gap of five units. At this point we can terminate, as we know that all remaining rectangles have dimensions of equal or less than five.

Assume that we have the same list of rectangles, but that this time there is a gap of four units. The first rectangle in the list is examined, $(7,3)$. It can fill three units if

Figure 8. Finding a new gap when the old gap has not been completely filled.


If last shape placed left in gap then:
If last shape placed right in gap then:
The gap's width is found by:
New Gap Location = (Gap Location) + (Placed Rectangle Width)
New Gap Location = Gap Location
New Gap Width = (Gap Width) – (Placed Rectangle Width)
4 4 4 0 0 0 0 6 6
Location = 3
Width = 4
4 4 4 2 2 0 0 6 6
New Location = Old Location + Rectangle Width = 3 + 2 = 5
New Width = Old Width – Rectangle Width = 4 – 2 = 2

rotated. The second rectangle is examined, $(5, 3)$. Although it has an equal capacity to fill the gap as the first rectangle, we prefer to pack larger rectangles first, and so the search continues. The third rectangle, $(5, 2)$, is not better than Rectangle 1. We must continue because there may be a rectangle with a width of four units. The fourth rectangle is examined, $(2, 1)$. Now the width of this rectangle is worse than our current best, so we can terminate. The first rectangle would be returned as the best-fitting rectangle.

Note also that as soon as a rectangle that fits exactly is found, we terminate. This reduces the search time of the process and, due to the list structure, rectangle dimensions decrease as we proceed through the rectangle list. In general, it is better to place shapes with larger dimensions earlier in the packing than towards the end of the packing, where they may be allowed to protrude at the top of the layout and affect solution quality.

Packing Stage. First, the stock sheet is examined to find the lowest available gap (initially at $x = 0$, $y = 0$, and lasting the entire sheet width). The rectangle list is examined and the best-fitting rectangle returned. This is placed within the gap depending on the current placement policy (as described in §2.1). The rectangle is assigned coordinates and removed from the rectangle list. Finally, the relevant stock-sheet array elements are incremented by the rectangle height. The process continues: Find the position of the lowest gap, find the width of the lowest gap, find the best-fitting rectangle, assign coordinates, remove the rectangle from the rectangle list, and update the stock-sheet array. If the best-fitting rectangle does not completely fill the gap, then there is no need to locate the lowest gap for the next rectangle because it is a portion of the recent gap and can be found as shown in Figure 8.

If a gap is found for which no remaining rectangle can fit, then this is wasted space, and the stock-sheet array elements that reference the gap are raised up to the lowest neighbour. For example, assume there is a gap of two units, but that there are no rectangles with dimension two units or less within our rectangle list (Figure 9).

In Figure 9 there is a gap for which no rectangle is small enough to fit. Each neighbour of the gap is examined, a height of two to the left and six to the right. The array elements that define the gap are raised to the lowest neighbour (in this case, to a height of two). We can now recheck for the lowest gap and continue with packing. In raising some of the array elements it cannot be assumed that the new lowest gap is at a height of two because there may be more gaps at a height of zero or one, so the array must be rechecked.

Postprocessing Stage. Once every rectangle is packed, we proceed through all of the rectangles to find if any are protruding from the top of the packing and negatively affecting solution quality. When we find the highest-positioned rectangle, if the rectangle is orientated in such a way that its height is greater than its width, then we remove it from the packing and reduce the stock-sheet array by the relevant rectangle height. Note that if the rectangle is found orientated with width greater than height, then we cannot improve (reduce) the height of the packing, as this rectangle is in the lowest position possible. We then rotate the rectangle so that its width has the larger dimension and try to

Figure 9. Procedure when no rectangle will fit gap.

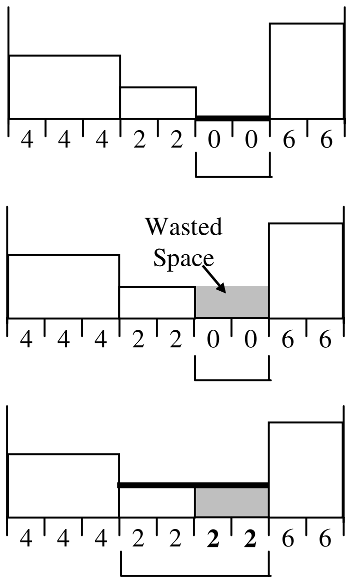
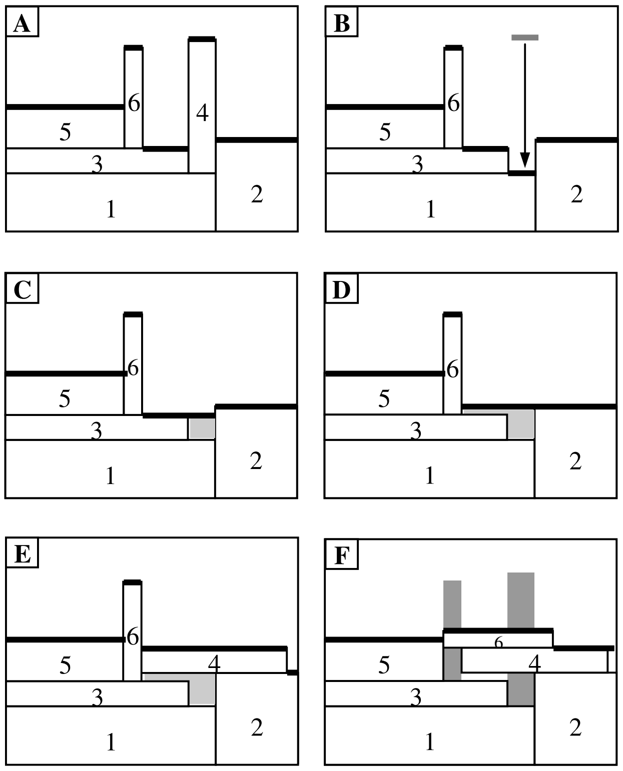


Figure 10. The processing of "towers."



pack the rectangle as before in "normal" packing but with the constraint that it must be packed in the width > height orientation. If the rectangle were allowed to rotate again, then it would be placed back in exactly the same position. We continue with this process until we cannot improve the quality of solution (Figure 10). This would occur when we remove the highest shape of the nest, rotate it, and in placing it in the lowest available position, it would give a worse solution than that achieved previously.

Figure 10A shows the solution after packing with the best-fit algorithm. To apply postprocessing to give better solutions, the tallest shape is removed (Shape 4) and the skyline is decreased appropriately as in Figure 10B. The removed shape is rotated and an attempt is made to reinsert it in the lowest part of the nest. As this shape will not fit, the lowest gap is raised to its lowest neighbour to make a more sizable gap, as in Figure 10C. As it still will not fit, the gap is raised once more (see Figure 10D). Now this gap is large enough to accommodate the shape, so it is placed as shown in Figure 10E. If this new arrangement improves the solution, it is accepted (as in this case). The same operation is performed with the next-highest shape (Shape 6). Figure 10F shows Shape 6 placed in its new position. If it enhances the quality of solution, it is accepted (as in this case). As all previous attempts have produced better-quality packings, the highest shape is selected once more. The highest shape is Shape 6 once again and its width is greater than its height, so we terminate and return the packing as the final solution.

Floating-Point Data. As the implementation is based on the faster integer data type, we must convert floating-point data to integer format by multiplying each rectangle by a scaling factor, depending on the degree of accuracy required. Some of the test problems from the literature, which we have used in §4, gave floating-point data (Valenzuela and Wang 2001).

Summary of Process. The final algorithm is based on the best solution returned after trying three packings with each utilising a different placement policy. The whole process can be summarised by the following pseudocode:

```
Obtain Stock Sheet Dimensions
Obtain List of $n$ Rectangles
Rotate each Rectangle so that Width $\geqslant$ Height
Sort Rectangle List by Decreasing Width (resolving equal
    widths by decreasing heights)
Initialize Skyline Array of $n$ Elements
for Each Placement Policy (Leftmost, Tallest Neighbour,
    Smallest Neighbour) do
    while Rectangles Not Packed do
      Find Lowest Gap
      if (Find Best-Fitting Rectangle == True) then
        Place Best-Fitting Rectangle Using Placement
          Policy
        Raise Array to Appropriately Reflect Skyline
      else
        Raise Gap to Lowest Neighbour
      end if
    end while
    while Optimisation Not Finished do
      Find Highest Shape
      if (Shape Width $\geqslant$ Shape Height) then
        Optimisation Finished
      end if
      Remove Highest Shape
      Reduce Array to Reflect Skyline
      Rotate Shape by 90 Degrees
      if (Shape Fits) then
        Place Best-Fitting Rectangle Using Placement
          Policy
        Raise Array to Appropriately Reflect Skyline
      else
        Raise Gap to Lowest Neighbour
      end if
      if (Packing Better == False) then
        Optimisation Finished
      end if
    end while
end for
Return Best Solution
```

Table 1. Test data from the literature.

Data source	Problem category	Test problems given	Number of rectangles	Optimal height	Object dimensions
Hopper and Turton (2001)	C1	P1, P2, P3	16 or 17	20	20×20
	C2	P1, P2, P3	25	15	40×15
	C3	P1, P2, P3	28 or 29	30	60×30
	C4	P1, P2, P3	49	60	60×60
	C5	P1, P2, P3	72 or 73	90	60×90
	C6	P1, P2, P3	97	120	80×120
	C7	P1, P2, P3	196 or 197	240	160×240
Valenzuela and Wang (2001)	Nice	P1, P2, P3, P4, P5, P6	25, 50, 100, 200, 500, 1,000	100	100×100
	Path	P1, P2, P3, P4, P5, P6	25, 50, 100, 200, 500, 1,000	100	100×100
Ramesh Babu and Ramesh Babu (1999)		P1	50	375	Width = 1,000

3. Benchmark Problems

In order to compare the relative performance of the presented best-fit heuristic to other heuristic and metaheuristic approaches, we used several test problems from the literature. Perhaps the most extensive data sets given for this problem are found in Hopper and Turton (2001), where 21 problem sets of rectangle data are presented in seven different-sized categories (each category has three problems of similar size and object dimension). Valenzuela and Wang (2001) provide floating-point data sets of both similarly dimensioned rectangles (named "nice" data) and vastly differing dimensions (named "path" data). Each category has data ranging from 25 to 1,000 rectangles. In Ramesh Babu and Ramesh Babu (1999), the authors use a test problem to compare their GA method against the GA method proposed by Jakobs (1996). Table 1 presents an overview of the test data from the literature.

Randomly Generated Problems. As we wanted to extensively test our approach, other test problems were generated at random. The method chosen to do this involved supplying the dimensions of a large rectangle, the number of smaller rectangles to be cut from this larger rectangle, and finally the smallest dimension allowed for any rectangle. A list of rectangles was maintained—initially with only the specified large rectangle. The algorithm selects any rectangle from the list and makes a vertical or horizontal guillotine cut through it at a random point to create two new rectangles. This process continues, making sure that the minimum dimension is observed, until the larger rectangle has been divided into the desired number of rectangles. This allows us to produce data sets for which we know the optimal solution. Table 2 shows an overview of our generated data. The complete data set is produced in Appendix A to allow other researchers to access the data.

4. Experimentation and Results

For our experiments we want to compare how the proposed heuristic method compared to conventional methods such as bottom-left, bottom-left-fill, and published metaheuristic methods. First of all, however, we present a series of experiments that investigates the options for employing different niche-placement policies within our algorithm.

4.1. A Comparison of the Placement Policies

We supplied the proposed best-fit heuristic with Hopper and Turton's data set and compared the performance of the three different placement policies we used (LM—Leftmost, TN—Tallest Neighbour, SN—Smallest Neighbour). Table 3 shows that each policy appears to have equal ability in finding good solutions (ticks indicate the best placement policy for each data set).

After 21 problems, we see that all three policies perform within a cumulative height of only 10 units difference, and we can see that each policy creates an outright best solution in several of the problems. This result validates our decision to allow all three policies to be tried within our heuristic. As an illustrative example, Figure 11 shows the solutions obtained by each placement policy with problem C2P3 from Hopper and Turton (2001). The problem has 25 shapes and an optimal solution of 15. The leftmost placement policy obtains a solution with a height of 18. The tallest-neighbour policy obtains a height of 16. The smallest-neighbour policy achieves a height of 17.

Table 2. Generated benchmark problems.

New test problems	Number of rectangles	Optimal height	Object dimensions
N1	10	40	40×40
N2	20	50	30×50
N3	30	50	30×50
N4	40	80	80×80
N5	50	100	100×100
N6	60	100	50×100
N7	70	100	80×100
N8	80	80	100×80
N9	100	150	50×150
N10	200	150	70×200
N11	300	150	70×200
N12	500	300	100×300
N13	3,152	960	640×960

Table 3. Comparison of placement policies and their cumulative performances.

		Placement policy solution height				Best policy		
Category	Problem	LM	TN	SN	Optimal	LM	TN	SN
C1	P1	21	22	21	20	✓		✓
	P2	22	22	22	20	✓	✓	✓
	P3	24	24	24	20	✓	✓	✓
C2	P1	17	16	17	15		✓	
	P2	16	17	16	15	✓		✓
	P3	18	16	17	15		✓	
C3	P1	32	32	32	30	✓	✓	✓
	P2	34	34	34	30	✓	✓	✓
	P3	33	35	35	30	✓		
C4	P1	63	64	65	60	✓		
	P2	64	66	62	60			✓
	P3	62	63	63	60	✓		
C5	P1	94	93	94	90		✓	
	P2	93	92	96	90		✓	
	P3	94	93	93	90		✓	✓
C6	P1	124	123	124	120		✓	
	P2	124	124	122	120			✓
	P3	124	128	125	120	✓		
C7	P1	246	247	250	240	✓		
	P2	246	244	246	240		✓	
	P3	245	246	248	240	✓		
Total		1,796	1,801	1,806	1,725	12	11	9

4.2. Comparison Against Bottom-Left and Bottom-Left-Fill Algorithms

In Hopper and Turton's comparison of bottom-left and bottom-left-fill algorithms, bottom-left-fill produced the better packings, although it took the longest time. Also, the use of preordering shapes by decreasing width or decreasing height gave better-quality solutions. As we are using a different quality measure from that of Hopper and Turton, involving the total height of the packing and not the density of the packing, the bottom-left and bottom-left-fill algorithms have been coded and tested using data from Hopper and Turton. Table 4 shows that the bottom-left-fill heuristic performs better than bottom-left and that the proposed best-fit heuristic outperforms bottom-left-fill in all but one of the categories, even when preordering is allowed (DW = decreasing width, DH = decreasing height, for unsorted we supply random sequences and take the average). The best solutions are shown in bold type.

4.3. Comparison Against Metaheuristic Approaches

Hopper and Turton (2001) also show that the best packings are achieved when using the bottom-left-fill algorithm along with a metaheuristic/evolutionary algorithm (simulated annealing or genetic algorithm). We implemented both of these algorithms to obtain a comparison with the new best-fit heuristic. For both of the metaheuristics, we operate on the rectangle input sequence and then decode the sequences using bottom-left-fill. We adapt the fitness function to use two distinct evaluators, a primary evaluator of the height of the nest and a secondary evaluator based on the sheet area used. This is necessary in order to distinguish packings with identical heights. We allow the metaheuristics five runs on each problem. The two metaheuristic implementations are briefly described as follows:

(a) *Genetic Algorithm/Bottom-Left-Fill Decoder* (*GA + BLF*). We implemented a genetic algorithm approach with

Figure 11. The solutions achieved by the placement policies on problem C2P3 from Hopper and Turton (2001).

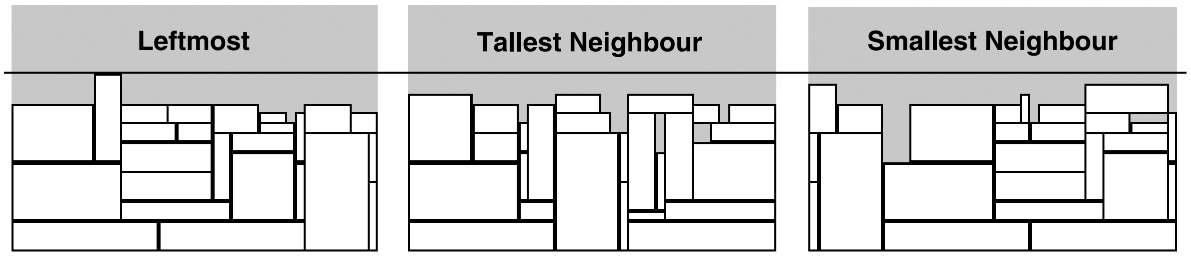


Table 4. Comparison of the best-fit heuristic to bottom-left and bottom-left-fill heuristics (% over optimal).

	C1			C2			C3			C4			C5			C6			C7		
Problem:	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
No. of rectangles:	16	17	16	25	25	25	28	29	28	49	49	49	72	73	72	97	97	97	196	197	296
BL	45	40	35	53	80	67	40	43	40	32	37	30	27	32	30	33	39	34	22	41	31
BL-DW	30	20	20	13	27	27	10	20	17	17	22	22	16	18	13	22	25	18	16	19	17
BL-DH	15	**10**	**5**	13	73	13	10	10	13	12	13	6.7	4.4	10	7.8	8.3	8.3	9.2	5	10	7.1
BLF	30	35	25	47	73	47	37	50	33	25	25	27	20	23	21	20	18	21	15	20	17
BLF-DW	10	15	15	13	20	20	10	13	13	10	5	10	5.6	6.7	5.6	5	4.2	4.2	4.6	3.3	2.9
BLF-DH	10	**10**	**5**	13	73	13	10	**6.7**	13	10	5	5	4.4	5.6	4.4	5	2.5	6.7	3.8	2.9	3.8
NEW BF	**5**	**10**	20	**6.7**	**6.7**	**6.7**	**6.7**	13	**10**	**5**	**3.3**	**3.3**	**3.3**	**2.2**	**3.3**	**2.5**	**1.7**	**3.3**	**2.9**	**1.7**	**2.1**

a population size of 50 and generation size of 1,000. Initially, each member of the population is a random ordering and orientation of the input rectangles. At each generation, we decode the sequences using bottom-left-fill to obtain packings. To improve the population after each generation, we select parents using linearly normalised roulette-wheel selection and then apply partially matched crossover with a probability of 60%. We also use two mutations of probability 3%, one for the random swapping of rectangles within an input sequence and another for the flipping of a rectangle's orientation. We use elitism to preserve the best two solutions of the population, and we replace the others with the children. We return the best solution found.

(b) *Simulated Annealing/Bottom-Left-Fill Decoder* (*SA + BLF*). We allowed the simulated annealing algorithm 50,000 iterations (same as the genetic algorithm method). The initial solution is a randomly generated input sequence. To move to a new candidate solution, we randomly select one of three neighbourhood operators: (i) swaps the orientation of a randomly chosen rectangle, (ii) moves a randomly chosen rectangle to a random position, (iii) swaps two randomly chosen rectangles. The starting temperature was calculated based on the total area of the input rectangles for each problem. We used a geometric cooling schedule of 0.9999, which is applied after every iteration. If we do not improve for 10 consecutive iterations, we reheat by applying a 1.01 times increase to the temperature. This scheme has the effect of cooling to about a 10% acceptance rate, where we remain until the search is completed. After 50,000 iterations, we return the best solution found during the process.

4.3.1. Experiments on Published Data. Table 5 shows the resultant solutions from using the metaheuristic methods and our new best-fit heuristic with test problems from the literature. For the metaheuristics, all times shown indicate the time taken to achieve the solution and not the time taken to complete the run. Therefore, the total time taken to finish the search will be of a longer duration. All experiments were performed on a PC with an 850 MHz CPU and 128 MB RAM. For all instances, the best solution is shown in bold.

The final columns of Table 5 show that the best-fit heuristic equals or outperforms both the GA + BLF and the SA + BLF metaheuristic methods on most of the data. On closer inspection of the cases where best fit cannot better the metaheuristic methods, we see that this occurs on problems involving fewer shapes. It seems that with up to around 50 shapes, which we call small problems, the metaheuristic methods produce the better solutions. As there are fewer possible sequences of shapes, they are able to evaluate a larger area within the search space and are therefore able to produce better solutions. On large problems, where there are more than 50 shapes to be packed, the best-fit heuristic provides the better solutions. Figure 12 shows our best-fit solution to the problem given in Ramesh Babu and Ramesh Babu (1999).

With the data of Ramesh Babu and Ramesh Babu, our best-fit heuristic obtains a solution of 40 mm, which equals the height returned from our implemented metaheuristic methods and their genetic method. It gives a better result than the solution of Jakobs' genetic method, which only achieved a solution of height of 45 mm. The execution time of the genetic algorithm of Ramesh Babu and Ramesh Babu is given as 521 seconds on a 100 MHz processor. However, our best-fit solution was achieved in 0.01 seconds.

As an illustrative example, a comparison of the solutions obtained by each of the examined methods is given in Figure 13, using problem C7P2 of 197 shapes, and where we know the optimal solution is 240 from Hopper and Turton (2001). Best fit achieves a solution with height of 244, GA + BLF finds a solution with height 251 using a population of 50 and several generations of 1,000, and SA + BLF finds a solution with height 253 after several 50,000-iteration runs.

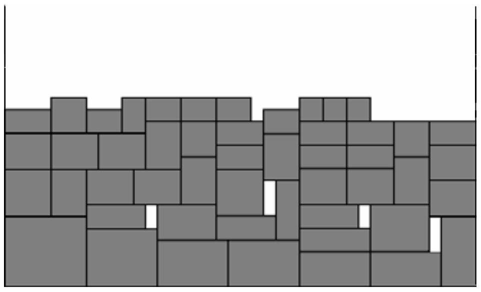
With the floating-point-based data of Valenzuela and Wang (2001), the best-fit heuristic consistently performed better than the other methods. Due to the floating-point nature of the data and the integer-based implementations, all methods took longer to execute than similarly sized integer problems due to the required scaling. However, because the best-fit heuristic is a one-pass method, the overhead is not exceptional.

4.3.2. Experiments on New Randomly Generated Data. In many real-world industrial applications there are time constraints and economic reasons for reaching good solutions quickly, so to compare the various methods

Table 5. Comparison of the best-fit heuristic against the metaheuristic methods (GA + BLF, SA + BLF).

					GA + BLF			SA + BLF			Best Fit		GA_{best} − BF		SA_{best} − BF	
Data set	Cat.	Problem	Number of shapes	Optimal height	Best	Worst	Time (s)	Best	Worst	Time (s)	Sol.	Time (s)	Abs.	% Impv.	Abs.	% Impv.
Hopper	C1	P1	16	20	**20**	21	3.4	**20**	21	1.1	21	<0.01	−1	−5.0	−1	−5.0
		P2	17	20	**21**	21	0.5	**21**	21	0.8	22	<0.01	−1	−4.8	−1	−4.8
		P3	16	20	**20**	21	7.1	**20**	21	0.8	24	<0.01	−4	−20.0	−4	−20.0
	C2	P1	25	15	16	16	1.3	16	16	6.5	**16**	<0.01	**0**	**0**	**0**	**0**
		P2	25	15	16	16	2.2	16	16	13.9	**16**	<0.01	**0**	**0**	**0**	**0**
		P3	25	15	16	16	1.0	16	16	13.6	**16**	<0.01	**0**	**0**	**0**	**0**
	C3	P1	28	30	32	32	7.4	32	33	20.3	**32**	<0.01	**0**	**0**	**0**	**0**
		P2	29	30	**32**	32	12.4	**32**	32	22.5	34	<0.01	−2	−6.3	−2	−6.3
		P3	28	30	**32**	32	11.6	**32**	33	18.3	33	<0.01	−1	−3.1	−1	−3.1
	C4	P1	49	60	64	64	35	64	64	65	**63**	<0.01	**1**	**1.6**	**1**	**1.6**
		P2	49	60	63	64	48	64	64	46	**62**	<0.01	**1**	**1.6**	**2**	**3.1**
		P3	49	60	62	63	61	63	64	70	**62**	<0.01	**0**	**0**	**1**	**1.6**
	C5	P1	72	90	95	95	236	94	95	501	**93**	0.01	**2**	**2.1**	**1**	**1.1**
		P2	73	90	95	96	440	95	96	285	**92**	0.01	**3**	**3.2**	**3**	**3.2**
		P3	72	90	95	96	150	95	95	425	**93**	0.01	**2**	**2.1**	**2**	**2.1**
	C6	P1	97	120	127	127	453	127	128	854	**123**	0.01	**4**	**3.1**	**4**	**3.1**
		P2	97	120	126	127	866	126	128	680	**122**	0.01	**4**	**3.2**	**4**	**3.2**
		P3	97	120	126	127	946	126	126	912	**124**	0.01	**2**	**1.6**	**2**	**1.6**
	C7	P1	196	240	255	255	4,330	255	256	4,840	**247**	0.01	**8**	**3.1**	**8**	**3.1**
		P2	197	240	251	253	5,870	253	253	5,100	**244**	0.01	**7**	**2.8**	**9**	**3.6**
		P3	196	240	254	255	5,050	255	255	6,520	**245**	0.01	**9**	**3.5**	**10**	**3.9**
Valenzuela	Nice	P1	25	100	108.2	109.5	185	109.3	110.9	192	**107.4**	0.02	**0.8**	**0.7**	**1.9**	**1.7**
		P2	50	100	112.0	113.5	956	112.6	113.3	1,358	**108.5**	0.01	**3.5**	**3.1**	**4.1**	**3.6**
		P3	100	100	113.0	114.0	2,580	113.3	113.8	4,130	**107.0**	0.01	**6.0**	**5.3**	**6.3**	**5.6**
		P4	200	100	113.2	114.0	1.6×10^4	113.4	113.8	2.3×10^4	**105.3**	0.03	**7.9**	**7.0**	**8.1**	**7.1**
		P5	500	100	111.9	112.3	5.1×10^4	111.9	112.2	7.3×10^4	**103.5**	0.08	**8.4**	**7.5**	**8.4**	**7.5**
		P6	1,000	100	—	—	—	—	—	—	**103.7**	0.26				
	Path	P1	25	100	**106.7**	108.4	148	109.6	110.8	122	110.1	0.04	−3.4	−3.2	−0.5	−0.5
		P2	50	100	**107.0**	108.0	2,417	108.7	109.5	1,010	113.8	0.24	−6.8	−6.4	−5.1	−4.7
		P3	100	100	109.0	110.1	4,190	109.0	111.2	6,865	**107.3**	0.15	**1.7**	**1.6**	**1.7**	**1.6**
		P4	200	100	108.8	109.0	1.6×10^4	110.3	111.3	2.7×10^4	**104.1**	0.33	**4.7**	**4.3**	**6.2**	**5.6**
		P5	500	100	111.1	111.5	8.8×10^4	112.4	113.0	1.4×10^5	**103.7**	0.19	**7.4**	**6.7**	**8.7**	**7.7**
		P6	1,000	100	—	—	—	—	—	—	**102.8**	0.23				
Ramesh Babu		P1	50	375	400	400	5.7	400	400	6.4	**400**	0.01	**0**	**0**	**0**	**0**

Figure 12. The resultant packing of our heuristic with data from Ramesh Babu and Ramesh Babu.



we should consider the time used in finding the solution. Table 6 uses our new data sets to show both solution quality and difference in time taken to find the solution. Once again, the best solutions are presented in bold.

The execution times from Tables 5 and 6 indicate that the best-fit heuristic can reach good solutions in a much quicker time than the other methods. In fact, on the newly generated problems in Table 6, best fit completes all solutions in a combined time of less than two seconds. Due to the use of populations with the genetic algorithm and neighbourhood functions with simulated annealing, there is an increase in execution time (because of their associated operations).

The power of the approach that we have presented is particularly illustrated by considering the largest of our problems, N13. Note that the GA + BLF and SA + BLF methods were unable to complete our N13 due to excessive time requirements (interpolated to approximately 26–40 weeks of execution time). However, the best-fit heuristic was able to pack the 3,152 shapes in less than two seconds, and it gave a solution of 964, which is only four units above optimal. The solution is presented in Figure 14.

5. Summary

In this paper we have described a new heuristic approach based on the best-fit methodology and have presented an efficient implementation of the heuristic. Using data provided by other researchers in the field of cutting and packing, our new heuristic has produced better results than the bottom-left and bottom-left-fill heuristics for most of the problems. The new heuristic has produced better-quality packings than the metaheuristic hybrids when given problems of 50 shapes or more. We obtained these solutions in a significantly shorter period of time by several orders of magnitude. The method also performs very well with data sets containing similar-sized rectangles and ones containing different-sized rectangles (such as the problems represented by data from Valenzuela and Wang 2001). In Hopper and Turton (2001), the authors concluded that the methods that yielded the best results were GA + BLF and SA + BLF. We have demonstrated that, even with extremely large problems, the proposed best-fit heuristic is able to outperform currently published and established heuristic and metaheuristic methods to produce solutions that are very close to optimal, using very small execution times. Many other areas of operations research, including memory allocation and multiprocessor scheduling, share a similar

Figure 13. A comparison of the best-fit heuristic to the metaheuristic methods using problem C7P2 from Hopper and Turton.

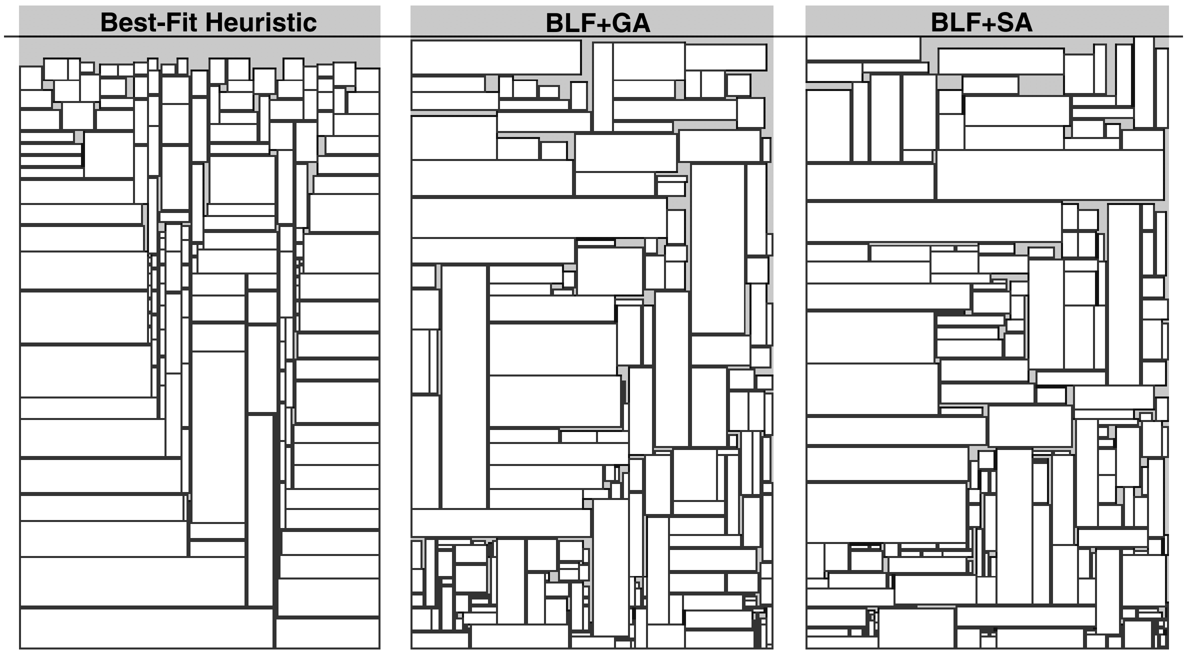
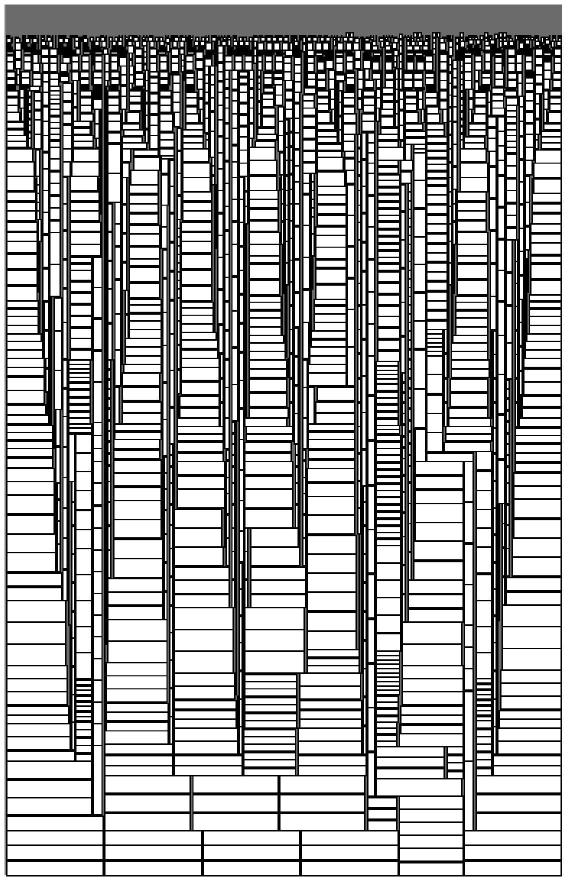


Table 6. A comparison of execution time.

			GA + BLF		SA + BLF		BF Heuristic		GA_{best} − BF		SA_{best} − BF	
Problem	No. of shapes	Optimal height	Result	Time (s)	Result	Time (s)	Result	Time (s)	Abs.	% Impv.	Abs.	% Impv.
N1	10	40	**40**	1.02	**40**	0.24	45	<0.01	−5	−12.5	−5	−12.5
N2	20	50	**51**	9.2	52	8.14	53	<0.01	−2	−3.9	−1	−1.9
N3	30	50	52	2.6	52	39.5	**52**	**<0.01**	**0**	**0**	**0**	**0**
N4	40	80	83	12.6	83	84	**83**	**<0.01**	**0**	**0**	**0**	**0**
N5	50	100	106	52.3	106	228	**105**	**0.01**	**1**	**0.9**	**1**	**0.9**
N6	60	100	103	261	103	310	**103**	**0.01**	**0**	**0**	**0**	**0**
N7	70	100	**106**	671	**106**	554	107	0.01	−1	−0.9	−1	−0.9
N8	80	80	85	1,142	85	810	**84**	**0.01**	**1**	**1.2**	**1**	**1.2**
N9	100	150	155	4,431	155	1,715	**152**	**0.01**	**3**	**1.9**	**3**	**1.9**
N10	200	150	154	2×10^4	154	6,066	**152**	**0.02**	**2**	**1.3**	**2**	**1.3**
N11	300	150	155	8×10^4	155	3×10^4	**152**	**0.03**	**3**	**1.9**	**3**	**1.9**
N12	500	300	313	4×10^5	312	6×10^4	**306**	**0.06**	**7**	**2.2**	**6**	**1.9**
N13	3,152	960	—	—	—	—	**964**	**1.37**				

logical structure to the problem in this paper. Therefore, the techniques proposed in this paper could be applied to these other areas with the possibility of similar improvements in solution quality.

Figure 14. The resultant packing from data N13 involving 3,152 rectangles.



Note. A packing of height 964 is achieved using best fit, which is only 4 over optimum.

Appendix A. Generated Benchmark Problems

N1: 10 shapes, 40 × 40.

Width	Height	Width	Height
7	6	4	4
40	16	7	8
5	20	4	20
24	24	5	4
7	4	7	6

N2: 20 shapes, 30 × 50.

Width	Height	Width	Height
23	9	14	6
19	4	6	6
12	21	5	4
6	4	4	6
7	13	6	4
9	4	7	6
4	6	14	11
23	6	4	7
16	6	8	4
4	14	14	4

N3: 30 shapes, 30 × 50.

Width	Height	Width	Height	Width	Height
10	6	6	4	5	4
4	4	3	15	3	5
4	5	9	6	3	3
3	5	3	3	5	8
10	5	3	4	5	16
3	5	6	18	10	13
7	13	7	5	3	3
3	4	3	4	10	17
8	32	5	3	5	14
3	23	3	8	5	3

N4: 40 shapes, 80 × 80.

Width	Height	Width	Height	Width	Height	Width	Height
61	38	4	4	5	4	5	72
7	4	5	4	10	7	5	52
9	5	8	10	4	7	5	4
5	4	4	4	20	7	9	7
5	7	5	5	5	7	5	12
7	7	32	4	5	12	9	33
9	15	7	7	5	4	4	8
4	4	5	4	11	7	9	34
4	4	5	8	8	21	4	4
32	31	7	24	9	4	29	4

N5: 50 shapes, 100 × 100.

Width	Height	Width	Height	Width	Height
25	10	7	10	7	6
74	8	6	10	25	10
27	19	11	10	26	8
64	34	53	7	36	6
74	6	20	6	6	6
39	6	46	10	6	7
48	7	18	6	20	6
11	10	10	6	16	22
8	10	10	10	20	14
14	6	9	6	10	6
6	10	7	6	14	8
26	6	11	6	8	6
27	6	7	6	7	6
6	10	6	8	16	6
8	10	9	7	8	6
31	10	47	7	15	6
23	6	10	7		

N6: 60 shapes, 50 × 100.

Width	Height	Width	Height	Width	Height
3	3	50	4	3	5
9	4	6	25	5	8
26	30	5	3	50	3
5	5	24	30	3	19
4	21	4	12	8	3
14	24	5	3	3	8
4	5	5	4	4	3
3	4	5	3	4	3
3	4	5	4	4	3
3	5	12	6	35	13
5	5	5	5	4	3
5	5	16	6	4	3
3	3	3	3	5	5
7	5	7	5	3	3
3	5	4	14	5	21
3	5	5	5	4	7
5	3	4	5	35	12
3	6	7	21	4	15
3	3	5	24	4	5
3	5	5	8	50	3

N7: 70 shapes, 80 × 100.

Width	Height	Width	Height	Width	Height
6	82	2	4	3	40
3	2	2	2	3	4
3	38	8	2	8	2
19	23	3	2	3	20
7	36	28	22	2	2
46	12	22	2	2	2
3	4	3	6	3	4
3	2	2	2	2	2
2	82	37	22	3	2
3	15	24	2	23	38
3	2	13	2	2	2
3	22	11	2	11	38
2	2	4	33	4	2
3	37	9	12	12	7
2	2	5	2	2	4
2	2	25	5	3	2
3	2	2	2	3	52
3	40	2	2	3	14
6	2	3	2	3	2
7	2	2	2	18	4
7	7	3	4	19	59
2	2	3	22	23	2
18	2	9	22		
13	7	3	5		

N8: 80 shapes, 100 × 80.

Width	Height	Width	Height	Width	Height
17	6	12	6	3	5
3	5	37	15	5	16
16	5	3	4	20	5
4	4	3	3	4	5
4	3	34	11	4	5
4	5	4	5	8	6
4	4	23	4	10	3
12	4	23	4	3	4
3	11	37	3	3	7
3	4	7	50	11	19
40	13	20	3	13	43
8	20	3	5	5	20
3	9	4	7	5	11
4	28	10	6	4	4
9	11	7	6	9	3
3	22	5	8	3	4
4	3	3	3	3	14
4	3	4	4	10	6
20	4	9	24		
4	35	11	8		
4	4	4	3		
23	34	3	5		
4	29	8	10		
3	12	7	3		
11	8	3	3		
4	21	3	5		
7	35	16	5		
23	35	7	4		
3	4	3	10		
4	5	12	10		
3	15	18	4		

N9: 100 shapes, 50 × 150.

W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H
5	3	4	3	4	4	4	3	3	3	4	5	3	13	3	8	4	5	3	10
3	5	47	7	4	3	3	40	5	3	3	5	4	3	3	4	3	4	4	9
5	3	4	4	8	7	4	5	3	20	47	4	3	3	4	32	3	39	15	50
5	5	11	3	3	3	24	4	3	3	3	3	5	3	3	5	19	17	12	27
4	5	3	4	4	4	4	3	3	4	4	28	4	3	8	3	10	5	22	4
4	15	4	5	28	5	3	3	3	6	3	5	28	11	3	5	6	54	8	4
5	5	19	6	9	5	3	3	4	5	3	3	3	3	3	4	9	4	3	4
3	3	3	5	3	4	3	5	6	5	4	4	4	3	4	3	7	3	16	5
44	35	4	3	12	23	3	3	7	3	4	4	10	3	4	9	3	54	18	4
3	5	5	3	28	7	9	5	3	8	3	35	10	5	5	3	3	5	3	14

N10: 200 shapes, 70 × 150.

W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H
2	3	2	2	2	2	6	2	2	2	2	2	4	6	2	2	3	12	3	2	5	2	6	2
3	3	2	2	3	2	3	2	2	2	2	2	51	3	2	3	2	2	14	3	3	3	8	3
3	21	2	3	3	2	2	2	2	8	13	2	3	2	6	3	16	31	2	2	4	4	2	2
54	33	3	3	25	3	25	3	20	2	2	2	9	2	34	2	2	2	3	12	42	3	2	2
8	5	3	2	2	2	2	8	2	3	2	3	5	9	2	2	3	2	2	2	2	2	3	3
2	2	2	2	2	2	2	2	4	2	2	2	2	2	13	33	13	3	4	2	3	11	2	3
3	2	3	62	2	2	2	2	2	3	2	3	25	2	2	2	2	2	3	3	2	3	2	3
2	2	3	5	2	4	2	3	3	8	2	2	2	2	16	3	2	6	2	3	2	2	2	3
3	2	2	2	60	25	3	3	2	11	2	3	3	2	4	2	16	4	27	2	2	2	2	2
5	3	2	3	2	2	5	13	2	3	19	8	3	3	2	7	8	3	3	2	31	28	19	2
2	2	2	2	59	18	4	3	5	12	9	4	2	3	2	5	2	5	2	4	2	2	2	2
29	3	6	6	5	28	4	2	32	2	17	8	32	2	2	2	3	2	7	5	2	3	2	2
3	3	3	3	3	2	2	2	2	2	5	2	11	3	2	2	3	3	5	2	2	2	20	2
2	3	2	2	12	3	2	2	3	2	2	2	17	6	2	2	2	2	3	10	2	2		
6	50	2	2	29	2	48	3	5	2	2	3	3	2	7	2	2	2	5	3	2	7		
3	2	2	2	2	2	2	2	2	3	5	3	2	2	8	3	2	2	14	3	2	2		
22	6	7	2	3	6	3	3	3	3	23	2	2	2	2	2	2	3	18	3	5	6		

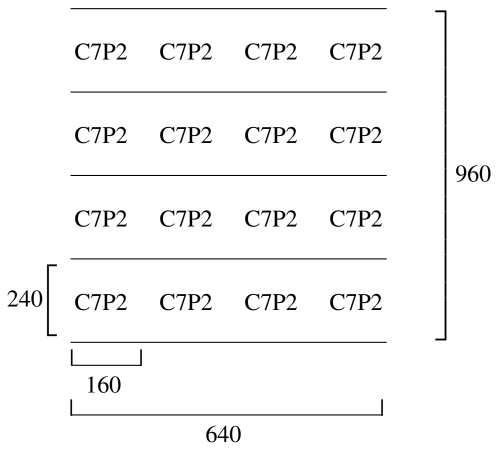
N11: 300 shapes, 70 × 150.

W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H
3	3	3	4	3	20	3	2	3	17	70	7	2	2	3	3	2	3	2	2	2	6	32	10
17	3	3	2	2	5	19	23	3	2	2	3	2	2	6	9	2	3	3	3	3	3	3	3
21	9	2	3	3	3	3	2	2	2	32	6	9	11	3	2	3	3	8	4	11	4	2	2
2	3	2	5	2	2	3	2	63	3	4	2	4	16	2	13	2	2	9	3	5	5	2	3
2	11	2	10	3	2	2	2	2	2	4	2	3	3	3	2	6	17	9	2	9	5	2	2
2	3	2	3	7	3	2	3	2	2	2	7	3	3	2	2	3	5	8	2	4	6	3	12
3	3	3	2	2	3	19	19	4	3	2	3	2	3	2	3	3	2	6	2	13	17	3	9
31	17	2	3	10	11	3	2	3	2	2	2	2	2	15	2	2	2	2	4	2	2	2	5
2	2	3	2	5	4	2	5	8	2	15	17	7	2	2	2	4	2	2	3	2	14	14	17
2	3	3	3	3	2	2	3	4	3	3	2	2	6	2	3	3	2	8	3	14	17	12	2
20	3	2	5	3	2	3	3	3	10	2	3	23	6	2	2	2	7	10	2	3	3	2	2
2	3	2	3	38	3	2	3	3	2	2	2	18	2	2	2	10	4	3	6	3	8	11	4
2	3	2	2	48	2	3	2	3	2	3	3	3	2	3	4	3	2	30	5	27	2	3	4
70	4	12	14	2	5	3	3	2	2	3	2	4	8	3	3	2	2	3	2	10	21	3	3
5	4	2	3	6	2	4	11	2	2	30	4	3	2	24	2	2	3	10	3	2	4	7	3
23	2	3	30	3	3	2	3	2	2	21	3	10	2	4	13	4	3	3	4	6	2	2	3
5	3	3	3	8	16	2	3	3	3	3	3	8	3	3	16	2	3	2	2	21	2	3	2
2	12	2	3	2	3	3	3	2	23	2	2	10	2	26	6	3	2	3	2	4	2	4	3
2	3	2	3	2	3	2	3	2	3	2	3	2	8	2	2	3	3	5	2	2	4	3	2
2	3	8	3	3	2	3	5	2	3	3	3	5	2	3	2	3	3	7	3	7	3	16	2
7	2	4	2	15	2	2	2	18	2	3	5	3	5	2	3	6	16	3	4	2	3	5	5
2	3	2	3	15	6	2	3	2	3	3	3	3	2	5	11	2	2	2	2	3	2	4	11
5	16	2	2	2	3	2	2	3	2	3	3	2	2	70	5	2	2	3	3	3	2	5	17
2	2	24	7	2	3	4	6	2	3	27	4	5	4	3	5	25	2	12	4	19	5	3	4
15	6	3	3	3	3	5	3	3	3	3	3	5	16	2	2	2	2	2	2	10	4	2	4

N12: 500 shapes, 100×300.

W	H	W	H	W	H	W	H	W	H	W	H	W	H	W	H
5	4	3	6	3	5	3	3	4	28	3	5	4	75	3	6
3	13	3	3	5	5	15	23	3	4	5	5	5	6	3	3
49	3	3	3	3	3	3	3	3	4	3	5	22	5	18	24
3	5	5	34	4	4	5	3	5	4	4	5	4	6	9	3
5	5	3	5	3	3	3	3	5	5	3	5	9	23	4	5
3	5	3	3	3	3	3	4	3	4	3	4	3	7	5	7
5	5	5	3	3	3	19	12	3	4	3	5	12	31	4	11
3	3	3	3	3	16	4	3	3	3	3	5	31	3	9	3
4	10	3	4	3	5	5	18	3	3	4	3	15	8	3	12
3	4	3	5	12	3	3	3	4	3	3	4	4	3	36	33
4	11	5	3	3	5	3	4	3	3	3	5	3	9	3	4
3	7	3	21	3	3	33	18	3	3	3	12	3	12	16	8
9	8	4	4	3	4	3	4	3	3	3	4	3	3	3	6
7	33	3	28	5	3	3	7	3	3	5	5	36	27	3	4
5	5	3	3	4	4	3	10	5	3	3	3	3	8	3	5
5	3	3	14	5	3	4	3	3	6	3	3	5	51	4	26
4	5	31	5	5	4	4	5	3	3	5	3	3	35	4	3
5	4	3	3	5	5	5	11	3	12	28	5	3	4	4	10
5	3	4	4	3	18	4	4	3	3	4	5	5	8	3	4
3	5	3	5	3	3	3	3	22	10	3	3	3	3	40	10
12	5	5	3	3	6	3	10	4	6	3	5	3	10	9	18
3	3	5	3	3	3	9	34	20	4	5	3	5	3	5	3
3	4	5	20	3	4	3	3	4	5	3	4	4	16	4	4
3	3	3	26	3	3	3	3	3	3	3	5	3	4	6	98
3	5	5	3	3	5	4	3	3	3	3	10	3	3	3	3
5	3	17	3	4	3	5	16	4	4	3	4	4	5	9	5
3	5	5	7	3	3	3	36	3	7	3	4	3	3	3	3
3	7	5	9	7	6	4	5	3	3	3	4	3	3	3	3
5	5	3	7	4	3	5	4	3	3	3	3	18	3	3	4
5	3	54	13	3	3	3	21	3	4	8	4	5	4	10	27
5	3	10	10	12	19	3	3	12	4	3	4	3	21	3	21
3	7	14	3	3	17	4	3	5	4	3	68	3	23	10	17
5	3	3	4	3	3	3	4	3	8	7	15	3	3	3	3
3	3	3	3	3	3	5	3	5	5	5	4	5	13	3	9
3	4	3	5	3	5	3	4	3	44	3	3	5	23	3	51
5	4	3	3	3	5	3	3	18	5	5	20	43	11	8	98
4	4	3	3	3	3	3	5	3	5	6	5	3	3	6	7
3	4	3	3	4	3	3	4	3	3	3	3	31	61	3	33
3	3	3	3	4	3	4	10	4	3	3	4	3	3	7	20
4	8	3	3	3	3	21	3	4	6	4	3	3	3	3	8
5	4	5	6	4	3	3	3	9	5	3	5	4	5	4	3
5	3	5	3	3	4	4	5	3	5	4	10	3	11	4	13
4	3	5	5	3	4	3	4	3	7	9	3	3	7	3	4
12	33	3	4	3	12	3	7	4	3	5	11	10	125	6	35
3	5	5	3	3	3	7	4	3	4	4	3	3	5	3	5
3	3	4	3	3	3	11	3	3	29	5	4	4	5	3	4
4	4	3	3	3	3	3	3	3	3	3	3	5	6	3	6
3	5	3	4	3	3	3	3	3	5	5	3	5	94	3	4
5	5	14	27	5	10	4	3	4	7	3	4	6	8	4	3
5	4	3	3	3	3	3	5	12	3	4	14	4	3	24	4
7	3	3	4	3	3	8	7	3	3	18	25	4	4	3	6
5	3	11	3	3	3	3	3	8	58	4	3	3	3	4	3
5	4	3	3	3	5	4	24	14	5	3	4	5	18	5	6
5	3	4	4	3	13	5	3	5	4	10	25	4	5	30	4
3	4	3	3	3	3	3	3	4	5	11	49	5	5	3	3
3	4	3	4	3	3	4	4	3	4	3	3	3	3	17	3
3	5	3	3	4	5	3	5	5	4	5	4	3	10	4	5
3	3	4	5	4	5	4	3	3	4	5	3	4	3	5	5
3	4	4	5	3	3	18	25	4	7	15	4	3	3	3	6
7	3	3	5	54	19	3	3	3	4	3	3	3	25		
5	8	18	9	5	3	3	3	3	5	8	3	4	3		
34	12	4	16	12	3	3	3	6	16	4	5	4	9		
5	5	3	4	5	10	3	4	3	5	6	12	3	3		

N13: 3,152 shapes, 640 × 960. Obtained by a 4 × 4 extension of problem C7P2 (197 shapes) from Hopper and Turton (2001).



Acknowledgments

The authors thank the UK Engineering and Physical Sciences Research Council (EPSRC) and Esprit Automation Ltd. for supporting this work. They also thank the anonymous reviewers for their helpful comments.

References

Albano, A., R. Orsini. 1979. A heuristic solution of the rectangular cutting stock problem. *Comput. J.* **23**(4) 338–343.

Baker, B. S., E. G. Coffman, Jr., R. L. Rivest. 1980. Orthogonal packings in two dimensions. *SIAM J. Comput.* **9**(4) 808–826.

Beasley, J. E. 1985a. An exact two-dimensional non-guillotine cutting tree search procedure. *Oper. Res.* **33**(1) 49–64.

Beasley, J. E. 1985b. Algorithms for unconstrained two-dimensional guillotine cutting. *J. Oper. Res. Soc.* **36**(4) 297–306.

Bengtsson, B. E. 1982. Packing rectangular pieces—A heuristic approach. *Comput. J.* **25**(3) 353–357.

Chazelle, B. 1983. The bottom-left bin packing heuristic: An efficient implementation. *IEEE Trans. Comput.* **32**(8) 697–707.

Christofides, N., C. Whitlock. 1977. An algorithm for two-dimensional cutting problems. *Oper. Res.* **25**(1) 30–44.

Coffman, E. G., Jr., F. T. Leighton. 1989. A provably efficient algorithm for dynamic storage allocation. *J. Comput. System Sci.* **38** 2–35.

Coffman, E. G., Jr., M. R. Garey, D. S. Johnson. 1978. An application of bin packing to multiprocessor scheduling. *SIAM Comput.* **7** 1–17.

Coffman, E. G., Jr., M. R. Garey, D. S. Johnson. 1984. Approximation algorithms for bin packing—An updated survey. G. Ausiello, M. Lucertini, P. Serafini, eds. *Algorithm Design for Computer Systems Design.* Springer-Verlag, New York 49–106.

Cung, V. D., M. Hifi, B. Le Cun. 2000. Constrained two-dimensional cutting stock problems—A best-first branch-and-bound algorithm. *Internat. Trans. Oper. Res.* **7** 185–210.

Dagli, C. H., A. Hajakbari. 1990. Simulated annealing approach for solving stock cutting problem. *Proc. IEEE Internat. Conf. Systems, Man, and Cybernetics*. Los Angeles, CA, 221–223.

Dagli, C. H., P. Poshyanonda. 1997. New approaches to nesting rectangular patterns. *J. Intelligent Manufacturing* **3**(3) 177–190.

Dowsland, K. A., W. B. Dowsland. 1992. Packing problems. *Eur. J. Oper. Res.* **56** 2–14.

Dowsland, K. A., W. B. Dowsland. 1995. Solution approaches to irregular nesting problems. *Eur. J. Oper. Res.* **84** 506–521.

Dyckhoff, H. 1990. A typology of cutting and packing problems. *Eur. J. Oper. Res.* **44** 145–159.

Faina, L. 1999. An application of simulated annealing to the cutting stock problem. *Eur. J. Oper. Res.* **114** 542–556.

Falkenauer, E. 1996. A hybrid grouping genetic algorithm for bin packing. *J. Heuristics* **2**(1) 5–30.

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, CA.

Garey, M. R., D. S. Johnson. 1981. Approximation algorithms for bin packing problems: A survey. D. Ausiello, M. Lucertini, eds. *Analysis and Design of Algorithms in Combinatorial Optimization. CISM Courses and Lectures*, Vol. 266. Springer Verlag, New York, 147–172.

Gilmore, P. C. 1966. The cutting stock problem. *IBM Proc. Combinatorial Problems* 211–224.

Gilmore, P. C., R. E. Gomory. 1961. A linear programming approach to the cutting stock problem. *Oper. Res.* **9** 849–859.

Golden, B. L. 1976. Approaches to the cutting stock problem. *AIIE Trans.* **8** 265–274.

Hifi, M., V. Zissimopolous. 1997. Constrained two-dimensional cutting: An improvement of Christofides and Whitlock's exact algorithm. *J. Oper. Res. Soc.* **48** 324–331.

Hopper, E., B. C. H. Turton. 1999. A genetic algorithm for a 2D industrial packing problem. *Comput. Indust. Engrg.* **37** 375–378.

Hopper, E., B. C. H. Turton. 2001. An empirical investigation of metaheuristic and heuristic algorithms for a 2D packing problem. *Eur. J. Oper. Res.* **128** 34–57.

Hower, W., M. Rosendahl, D. Köstner. 1996. Evolutionary algorithm design. *Artificial Intelligence in Design '96.* Kluwer Academic Publishers, Dordrecht, Germany, 663–680.

Jakobs, S. 1996. On genetic algorithms for the packing of polygons. *Eur. J. Oper. Res.* **88** 165–181.

Kroger, B. 1995. Guillotineable bin packing: A genetic approach. *Eur. J. Oper. Res.* **84** 645–661.

Lai, K. K., J. W. M. Chan. 1996. Developing a simulated annealing algorithm for the cutting stock problem. *Comput. Indust. Engrg.* **32**(1) 115–127.

Li, Z., V. Milenkovic. 1995. Compaction and separation algorithms for non-convex polygons and their applications. *Eur. J. Oper. Res.* (Special Issue on Cutting and Packing) **84** 539–561.

Liu, D., H. Teng. 1999. An improved BL-algorithm for genetic algorithms of the orthogonal packing of rectangles. *Eur. J. Oper. Res.* **112** 413–419.

Paull, A. E. 1956. Linear programming: A key to optimum newsprint production. *Pulp Paper Magazine Canada* **57** 85–90.

Ramesh Babu, A., N. Ramesh Babu. 1999. Effective nesting of rectangular parts in multiple rectangular sheets using genetic and heuristic algorithms. *Internat. J. Production Res.* **37**(7) 1625–1643.

Roberts, S. A. 1984. Application of heuristic techniques to the cutting stock problem for worktops. *J. Oper. Res. Soc.* **35** 369–377.

Sweeny, P. E., E. R. Paternoster. 1992. Cutting and packing problems: A categorized application-orientated research bibliography. *J. Oper. Res. Soc.* **43**(7) 691–706.

Valenzuela, C. L., P. Y. Wang. 2001. Heuristics for large strip packing problems with guillotine patterns: An empirical study. *Proc. 4th Metaheuristics Internat. Conf.*, University of Porto, Porto, Portugal, 417–421.