Chapter 24

HEURISTICS FOR LARGE STRIP PACKING PROBLEMS WITH GUILLOTINE PATTERNS: AN EMPIRICAL STUDY

Christine L. Mumford-Valenzuela [1], Pearl Y. Wang *[2], Janis Vick [2]

[1] *Department of Computer Science, Cardiff University, UK*
christine@cs.cf.ac.uk

[2] *Department of Computer Science, George Mason University, USA*
pwang@cs.gmu.edu

Abstract In this paper we undertake an empirical study which examines the effectiveness of eight simple strip packing heuristics on data sets of different sizes with various characteristics and known optima. We restrict this initial study to techniques that produce guillotine patterns (also known as slicing floorplans). Guillotine patterns are important industrially. Our chosen heuristics are simple to code, have very fast execution times, and provide a good starting point for our research. In particular, we examine the performance of the eight heuristics as the problems become larger, and demonstrate the effectiveness of a preprocessing routine that rotates some of the rectangles by 90 degrees before the heuristics are applied. We compare the heuristic results to those obtained using a good genetic algorithm (GA) that also produces guillotine patterns. Our findings suggest that the GA is better on problems of up to about 200 rectangles, but thereafter certain of the heuristics become increasingly effective as the problem size becomes larger, producing better results much more quickly than the GA.

Keywords: Strip packing, heuristics, genetic algorithm.

*Partially supported by the NASA Goddard Space Flight Center (NAG–5–9781)

1. Introduction

We undertake an empirical study in which we compare the performance of some well-known strip packing heuristics and a good genetic algorithm on a range of two-dimensional rectangular packing problems. We restrict our study to problems involving *guillotine patterns* which are produced using a series of vertical and horizontal edge–to–edge cuts. Many applications of two-dimensional cutting and packing in the glass, wood, and paper industries are restricted to guillotine patterns. Packing heuristics that produce these patterns are generally simple to code and fast to execute.

The packing problem under consideration involves the orthogonal placement of a set of rectangles into a rectangular strip (or bin) of given width and infinite height so that no rectangles are overlapping. The goal of this strip packing problem is to minimize the height of the packing. We compare the performance of our GA, adapted from [Valenzuela and Wang, 2001; Valenzuela and Wang, 2000], with the performance of eight strip packing heuristics [Coffman et al., 1984; Baker and Schwarz, 1980; Sleator, 1980; Golan, 1981] on data sets of various sizes with a variety of characteristics.

These data sets were generated using a suite of data generation programs developed in [Wang and Valenzuela, 2001] that produce data sets with optimal guillotine packings of zero waste. The software allows a set of basic rectangles to be cut from a large enclosing rectangle of given dimensions, and options are available which allow the user to control various characteristics: the number of pieces in the data set, the maximum and minimum height–to–width ratios of the pieces, and the ratio of the area of the largest piece to that of the smallest piece.

Several recent comparative studies on strip packing have reported superior performances for metaheuristic algorithms over simple heuristic approaches (for examples see [Hopper and Turton, 2001; Hwang et al., 1994; Kröger, 1995]). We note, however, that in most of these cases, the problem sizes are restricted to 100 rectangles or less. Our interests lie in examining a larger range of problem sizes and types.

2. Simple Heuristic Algorithms for Guillotine Packing

In this section, we review the heuristic algorithms used for our comparative study. The sets of rectangles to be packed may first be presequenced by arranging them in order of non-increasing height or width. They are then placed in the bin, one at a time, in a deterministic fashion. Pre-sequencing is employed by four of the heuristic algorithms we use:

those based on the *level oriented* approach introduced by [Coffman et al., 1980]. To avoid pre-sequencing, two similar approaches utilize shelves for packing the rectangles and were proposed by [Baker and Schwarz, 1980]. A seventh algorithm was formulated by [Sleator, 1980] and splits the rectangular bin vertically into two sub-bins after packing some initial pieces. This idea is further extended by [Golan, 1981] who repeatedly splits the bin into smaller sub-bins and packs the pieces, sorted by decreasing width, into ever narrower bins as the algorithm progresses. All eight heuristic algorithms produce guillotine patterns. Other algorithms which produce non-guillotine patterns are reviewed in [Coffman et al., 1984].

The Level Oriented Algorithms

To implement a level oriented algorithm, the items are first pre-sequenced by non–increasing height, and then the packing is constructed as a series of *levels*, each rectangle being placed so that its bottom rests on one of these levels. The first level is simply the bottom of the bin. Each subsequent level is defined by a horizontal line drawn through the top of the tallest rectangle on the previous level.

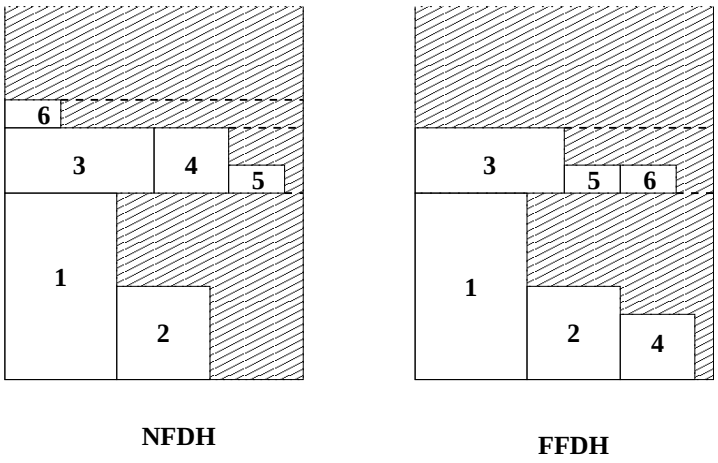

Figure 24.1. Some NFDH and FFDH packings

In the Next Fit Decreasing Height (NFDH) algorithm, rectangles are packed left justified on a level until the next rectangle will not fit, in which case it is used to start a new level above the previous one, on which packing proceeds. The run time complexity of NFDH (excluding

the sort) is linear, just placing one rectangle after another in sequence. The First Fit Decreasing Height (FFDH) algorithm places each rectangle left justified on the first (i.e. lowest) level in which it will fit. If none of the current levels has room, then a new level is started.

The FFDH heuristic can be easily modified into a Best Fit Decreasing Height (BFDH) and a Worst Fit Decreasing Height (WFDH) heuristic. BFDH packs each piece onto the best level (i.e. the one with least remaining horizontal space) on which it will fit. WFDH packs the piece into the largest remaining horizontal space where it will fit. These four approaches are two-dimensional analogues of classical one-dimensional packing heuristics. The run time complexity for FFDH, BFDH and WFDH is $O(n \lg n)$ where n is the number of rectangles being packed. Figure 24.1 illustrates the NFDH and FFDH packings for the same six rectangles.

For the level packing algorithms, the asymptotic worst case performance has been shown to be twice the optimum height for the NFDH algorithm and 1.7 times the optimum height for the FFDH algorithm. The asymptotic worst case performance bounds for BFDH and WFDH are, respectively, at least twice and 1.7 times the optimum height. These are the asymptotic bounds for their one-dimensional analogues and easily serve as lower bounds on their worst case performance in two dimensions as level-oriented algorithms.

The Shelf Algorithms

The on-line packing heuristics proposed by [Baker and Schwarz, 1980] can be used when pre-ordering of the input data is to be avoided. These heuristics are modifications of the Next-Fit and First-Fit approaches. Rectangles whose dimensions are between 0 and 1 are packed on *shelves* whose heights are set by a parameter r where $0 < r < 1$. Shelf sizes have the form r^k and each rectangle of height h, $r^{k+1} \leq h \leq r^k$ is packed into a shelf having height r^k. Thus, the motivation for the use of r is to limit the amount of vertical wasted space which is allowed on each shelf.

Using the Next-Fit approach, the Next-Fit Shelf (NFS_r) algorithm packs each piece as far to the left as possible on the highest shelf with height r^k where the rectangle height h, $r^{k+1} \leq h \leq r^k$. If there is no room on this shelf, a new one is created having height r^k. In the First-Fit Shelf (FFS_r) algorithm, the lowest shelf of the correct height on which the rectangles will fit is chosen. Examples of NFS_r and FFS_r packing are shown in Figure 24.2.

For the on-line shelf packing algorithms, the asymptotic worst case performance has been shown to be $\frac{2}{r}$ of the optimum height for the NFS_r

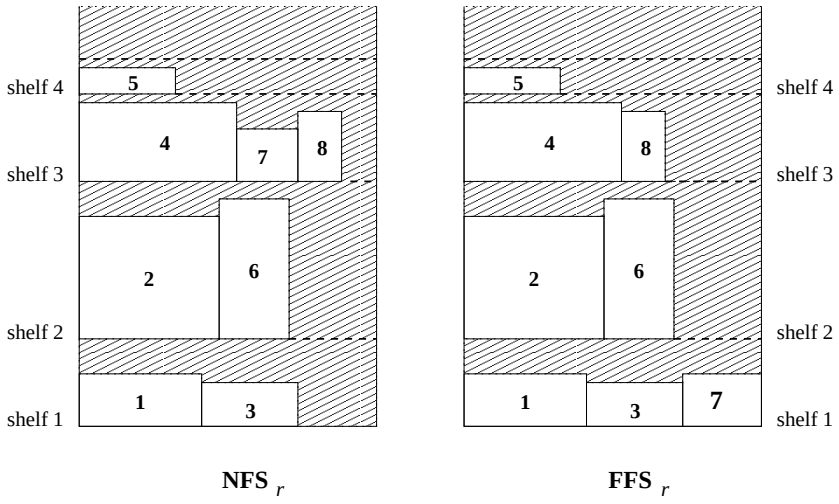

Figure 24.2. Two shelf packing algorithms

algorithm and $\frac{1.7}{r}$ times the optimum height for the FFS_r algorithm. These bounds are higher than for the level-oriented heuristics, but no pre-sequencing of the rectangles is required.

Sleator's Algorithm

Unlike the shelf heuristics, the heuristic proposed by [Sleator, 1980] is not an on-line algorithm. It also packs rectangles having width at most one. Initially, the rectangles with width greater than $\frac{1}{2}$ are stacked vertically beginning at the bottom left corner of the bin. A horizontal line (h_0) is then drawn through the top of the last packed rectangle. The area of the rectangles below this line is denoted by A_0.

Next, the remaining rectangles of the list are sorted by decreasing height. They are placed along the h_0 line until a piece is encountered which won't fit, or until no more rectangles remain in the list. In the first case, the bin is next split in half vertically into two open-ended bins with width exactly $\frac{1}{2}$. We let A_1 be the area of the rectangles in the left half-bin. All remaining packed and unpacked rectangles have area designated as A_2. Notice that one rectangle may be "split" in two, with part of it residing in the left half bin and the other part in the right half bin.

The remaining unpacked rectangles are packed into the two half-bins using a level approach: the half-bin with the current lowest level (defined by the tallest piece) is packed from left to right by placing rectangles

along the level until a half-bin edge is reached. The first rectangle packed this way defines a new level in this half-bin. If it is lower than the current level in the other half-bin, then packing resumes on this level in this half-bin. Otherwise, packing proceeds using the current level of the other half-bin as the site where the remaining pieces should be placed. An illustration of this process is shown in Figure 24.3.

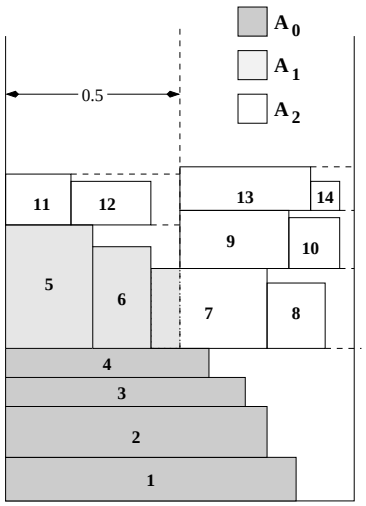

Figure 24.3. The Sleator packing algorithm

[Sleator, 1980] was able to prove that the absolute worst case error bound for the packing height of this heuristic was 2.5 times the optimal height and that this is a tight bound.

The Split Algorithm

The Split algorithm is an extension of Sleator's approach. It packs the pieces in order of non–increasing width. We can imagine that for each piece that is packed the original bin is split into two, and then into two again when the next piece is packed, and so on. As the rectangles to be packed are sequenced according to width, after packing some pieces, those left to be packed are narrower and thus easy to fit into one of newly created bins. If possible, we pack pieces side by side with previously packed pieces. When this is not possible, we pack pieces on top of previously packed pieces. When bins are split, closed bins (M) are created below in which no further pieces can be packed.

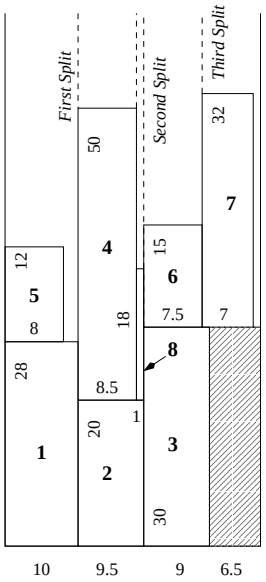

Figure 24.4. The Split algorithm packing process

A brief example is shown above where the seven rectangles are to be packed. Rectangle 1 is packed first. Then rectangle 2 is placed to its right, splitting the open-ended bin into two sub-bins. (There is no M in this case.) Rectangle 3 is placed in the right sub-bin (it can't fit next to piece 1 in the left sub-bin), causing another split of the right sub-bin. When the fourth piece is packed, it will not fit to the right of the last piece packed in any sub-bin. Hence it is stacked on top of rectangle 2 in the middle sub-bin because that is the lowest position where a piece can be stacked in the three sub-bins. Similarly pieces 5 and 6 will be stacked in the first and third sub-bins.

Subsequent to this, the heuristic checks the three sub-bins to see if piece 7 can be packed next to the last pieced packed into each sub-bin. Piece 7 can be placed next to piece 6 in the third sub-bin but is too wide to be placed next to either piece 5 or 4 in the first and second sub-bins. Thus, the placement of piece 7 splits the third sub-bin into three parts: two narrower sub-bins and a lower rectangle. Finally, piece 8 could be placed to the right of either piece 5, 4, or 7. However, the second choice reflects the sub-bin with the lowest current level and so piece 8 is packed there which causes another split.

Figure 24.4 illustrates the splitting process that is encountered when packing a bin using this approach. The worst–case performance of the Split algorithm is three times the optimum height. Full details of the Split algorithm are given in [Golan, 1981].

3. Our Genetic Algorithm

Our genetic approach to solving packing and placement problems is based on a normalized postfix representation which offers a unique encoding for each guillotine pattern and covers the search space efficiently. Our postfix representation provides a blueprint for a recursive bottom-up construction of a packing or placement by combining rectangles together in pairs. The general technique that we use has proven effective for the VLSI floorplanning problem [Valenzuela and Wang, 2001; Valenzuela and Wang, 2000]. However, that problem is more complex than the strip packing problem of the present study: although the individual rectangular components for VLSI placement have fixed areas, it is possible to vary their height and width dimensions to obtain closer packings. The GA that we developed for VLSI floorplanning incorporates a sophisticated area optimization routine that is not needed for the present strip packing study.

The Representation and Decoder

The postfix expressions that we use to encode placements for our GA utilize the binary operators '+' and '$*$' which signal that one rectangle should be placed on top of the other (+), or by the side of the other ($*$). Alternatively, + and $*$ represent horizontal and vertical cuts, when viewed from a top-down, cutting perspective. By repeatedly combining pairs of rectangles or sub-assemblies together, a complete layout can be generated. Wasted space will be created in the combining process when rectangles or sub-assemblies have different height or width dimensions. The objective of the GA is to pack the rectangles into a given fixed width bin as tightly as possible.

Note that a complete postfix expression of n rectangles will contain exactly $n-1$ operators. Also, at any point during the evaluation of a postfix expression, the cumulative total of operators must be less than the cumulative total of rectangles. The integers $1 \ldots n$ represent the n rectangles in a problem. *Normalized* postfix expressions are characterized by strings of alternating $*$ and + operators separating the rectangle IDs. Figure 24.5 illustrates a slicing floorplan, its slicing tree representation, and the corresponding normalized postfix expression.

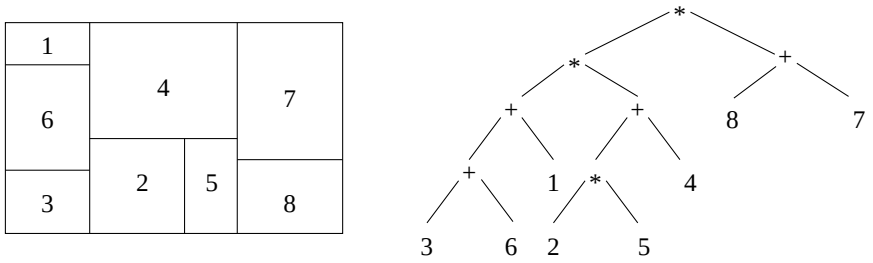

Figure 24.5. Slicing Trees and Normalized Postfix Strings

The representation used for our GA is order based and consists of an array of records, with one record for each of the basic rectangles of the data set. Each record contains three fields:

- *a rectangle ID field*: this identifies one of the basic rectangles from the set {1, 2, 3,..., n}
- *an op-type flag*: this boolean flag distinguishes two types of normalized postfix chains, $+ = +*+*+*+\ldots$ and $* = *+*+*+*\ldots$
- *a chain length field*: this field specifies the maximum length of the operator chain consecutive with the rectangle identified in the first field.

Starting with a given array of records produced by the GA, our decoder will construct a legal normalized postfix expression. It does this by transcribing the rectangular IDs given in the first field of each record in the sequence specified, while at the same time inserting chains of alternating operators following each rectangle ID, as specified in the second field of each record (i.e., either $+*+*+\ldots$ or $*+*+*\ldots$). The length of each individual chain of alternating operators is determined by examining the third field of each record which specifies chain length.

The chain lengths specified in the records are adopted whenever possible. If such an adoption produces an illegal postfix expression or partial expression, however, the decoder steps in to correct it, either by adding operators to, or subtracting operators from, an offending chain. Below is an example showing an encoded string and its normalized postfix interpretation:

rectangle 5	rectangle 2	rectangle 4	rectangle 1	rectangle 3
op-type $*$	op-type $+$	op-type $*$	op-type $*$	op-type $+$
length 2	length 1	length 0	length 2	length 0

Postfix expression generated: 5 2 + 4 1 ∗ + 3 +

The Genetic Algorithm

```
Procedure GA
begin
    Generate N random permutations {N is the population size}
    Evaluate the objective function (i.e. bin height)for each structure and store it
    Store best-so-far
    Repeat
        For each member of the population
            This individual becomes the first parent
            Select a second parent at random
            Apply crossover to produce offspring
            Apply a single mutation to the offspring
            Evaluate the objective function produced by offspring
            If offspring is a duplicate, delete it
            else
                    If offspring better than weaker parent then it replaces it in
                                population
                    If offspring better than best-so-far then it replaces best-so-far
        Endfor
    Until stopping condition satisfied
    Print best-so-far
End
```

Figure 24.6. Algorithm 2 A simple steady state genetic algorithm

Figure 24.6 gives the simple genetic algorithm (GA) which we use. It is an example of a steady state GA, and it uses the weaker parent replacement strategy first described in [Cavicchio, 1970]. In this scheme, a newly generated offspring will replace the weaker of its two parents if it has a better fitness value than that parent. The current GA is similar to the GAs used in [Valenzuela and Wang, 2000; Valenzuela and Wang, 2001], but differs in the following important ways:

- The area optimization routine designed for rectangles with fixed area but flexible height and width dimensions is not needed here because all rectangles have fixed dimensions (as mentioned earlier).

- Duplicates in the population are deleted.

- Selection based on fitness values has been abandoned in favour of simple uniform selection. The weaker parent replacement strategy seems to be sufficient to advance the genetic search, thus reducing the computational requirement of the GA.

- A rotation heuristic has been added which will rotate rectangles through 90 degrees when this is locally effective.

In the current GA population, duplicates are deleted as soon as they arise. To save computation time, we delete *phenotypic* duplicates i.e., new offspring are deleted if they duplicate a current population member's height (within sampling error). Ideally we should delete *genotypic* duplicates, but this would involve performing time consuming pairwise comparisons on the genetic array of records encoding the postfix expressions.

Each time the GA iterates through its inner loop, it selects two parents and applies *cycle crossover, CX* [Oliver et al., 1987] to the permutations of records to produce a single offspring. A single mutation is then applied which is selected from three alternatives M1, M2 or M3: M1 swaps the position of two rectangles, M2 switches the op-type flag from $+$ to $*$ or vice versa, and M3 increments or decrements (with equal probability) the length field. The pairs of parents are selected in the following way: the first parent is selected deterministically in sequence, but the second parent is selected uniformly, at random.

Once decoded, the postfix expression is evaluated, the corresponding slicing layout generated, and the bin height calculated and recorded.

For the present study, we have incorporated a rotation heuristic into our GA. This performs 90 degree rotations when it is locally effective during the construction of the layout from the postfix expression. This means that individual rectangles and sub-assemblies can be rotated when they are combined in pairs, if this results (locally) in a reduction of wasted space. For each pairwise combination of rectangles A and B, there are four alternative configurations: A combined with B, 'A rotated' combined with B, A combined with 'B rotated', and 'A rotated' combined with 'B rotated'. All four alternatives are tried each time a pair or rectangles or sub-assemblies are combined, and the best configuration selected.

The GA handles the fixed width constraint in the strip packing problem (i.e. we are packing rectangles into a bin of fixed width and infinite height) by rejecting packings that are too wide, and repeatedly generating new ones until the width constraint is satisfied.

Our current goal is to compare the performance of a good genetic algorithm with the classical strip packing heuristics on a range of data sets of different sizes with different characteristics. We do not claim that our GA is the best in existence; we do not know whether it is or not. Evidence that our genetic packing techniques are good, however, was provided first by the results obtained for VLSI floorplanning in [Valenzuela and Wang, 2000; Valenzuela and Wang, 2001], and more directly

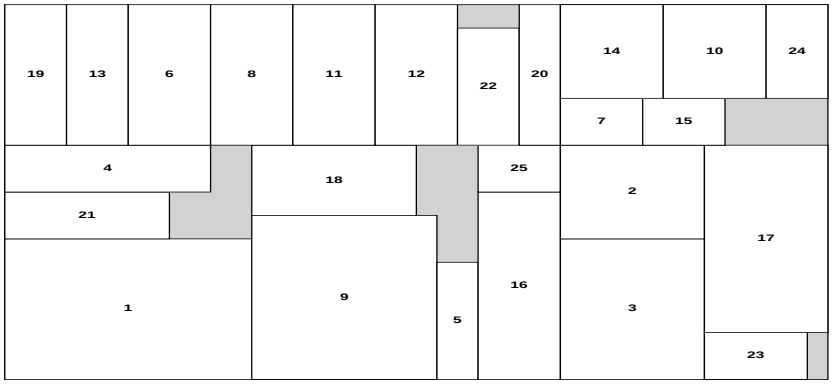

Figure 24.7. GA packing for the 25 rectangle problem from (Jakobs, 1996): width = 40, height = 16

for some small strip packing results: we matched the best solutions obtained by Liu and Teng in a recent paper [Liu and Teng, 1999].

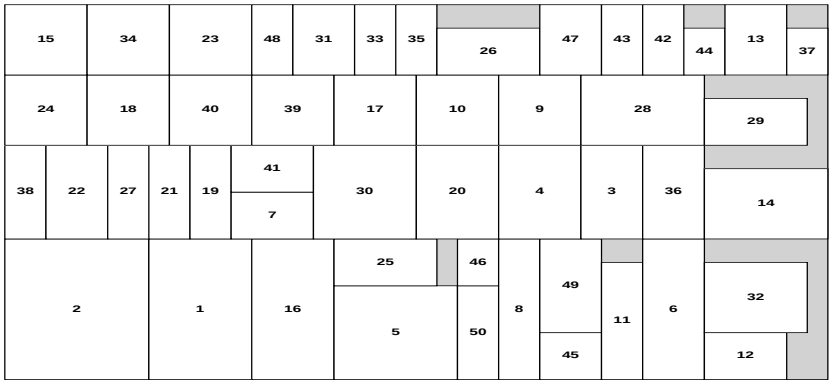

Figure 24.8. GA packings for the 50 rectangle problem from (Jakobs, 1996): width = 40, height = 16

For these strip packing experiments, the 25 and 50 rectangle problems from [Jakobs, 1996] were used. We were able to match the results of Liu and Teng, despite our use of a much more restrictive search engine: in [Liu and Teng, 1999], a non-guillotine Bottom Left placement heuristic is used; our search space was confined to guillotine patterns. Figures 24.7 and 24.8 shows some typical packings obtained by our GA for these problems.

4. Experimental Results

In this section, we present results for the eight strip packing heuristics and our GA when they are applied to two different types of data sets. In one type of data set, the rectangles are all similar in shape and size. These are referred to as the *Nice* data. In the other type of data set, the variation in rectangle shape and size is more extreme; these are the pathological or *Path* data sets.

All of the data sets that were employed for our experiments were generated using the techniques described in [Wang and Valenzuela, 2001] which produce sets of rectangles by applying guillotine cuts at appropriate locations within a 100×100 rectangle. Data sets of size n are created by making $n-1$ guillotine cuts, and each resulting data set can be packed into a strip of width 100 with an optimum height of 100 (and zero waste). Simple constraints may be applied during the cutting process to guarantee that the data sets have the different shape and size characteristics that were mentioned above. For the majority of our experiments, we used data sets with sizes $n = 25, 50, 100, 200$ and 500 for both the *Nice* and *Path* types, but in a final set of experiments, we applied only the most successful of the heuristics to some larger problems.

Comparing the Eight Heuristics

In the first set of experiments, the eight packing heuristics (NFDH, FFDH, BFDH, WFDH, NFS_r,FFS_r, Split and Sleator) were applied to 50 *Nice.n* and 50 *Path.n* data sets having sizes $n = 25, 50, 100$, and 200. In addition, ten data sets of each type with $n = 500$ rectangles were tested. For the *Nice.n* data sets, the height-to-width ratio of all n rectangles fell in the range $1/4 \leq H/W \leq 4$, and the maximum ratio of the areas of any two rectangles in the set was 7. For the *Path.n* data, the H/W ratio lay in the range $1/100 \leq H/W \leq 100$, and the maximum area ratio was 100.

Figure 24.9 illustrates the packings produced by FFDH, FFS (with $r = 0.5$), Sleator's, and the Split heuristic for a sample $Nice.50$ data set, and Figure 24.10 shows the results of applying the same heuristics to a sample $Path.50$ data set.

The mean heights of the packings generated by all eight strip packing heuristics were calculated and are shown in Tables 24.1–24.6. The runtimes of each of the eight algorithms took less than one second when applied to each data set on a Toshiba Tecra computer with a 1.2 GHz Pentium III processor and 256 MBytes RAM.

Tables 24.1 and 24.2 give mean packing heights for both *Nice* and *Path* data sets. Recall that the rectangles in a data set are reordered by

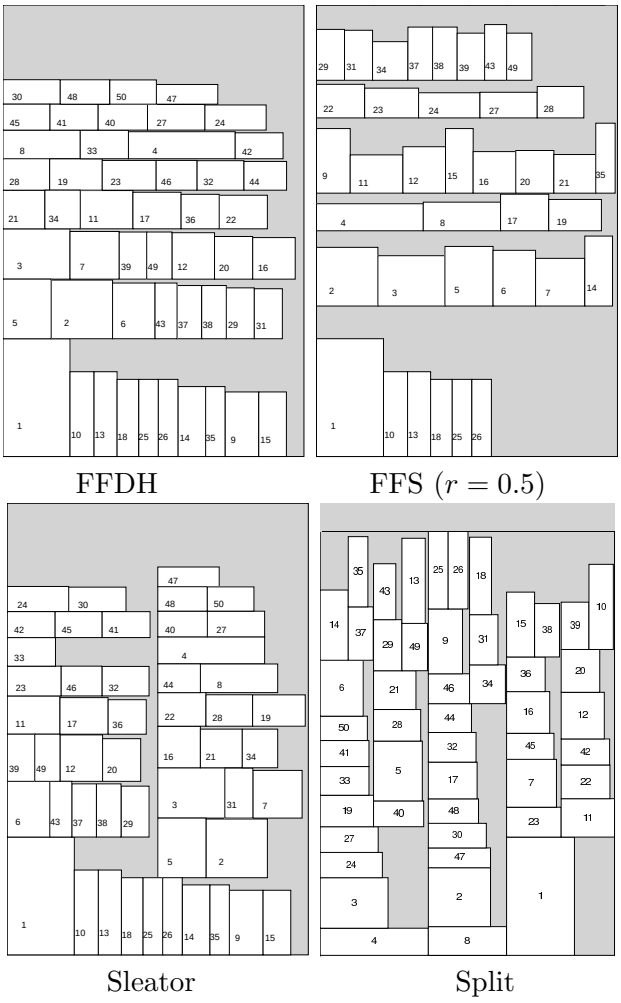

Figure 24.9. Packings of a *Nice*.50 data set

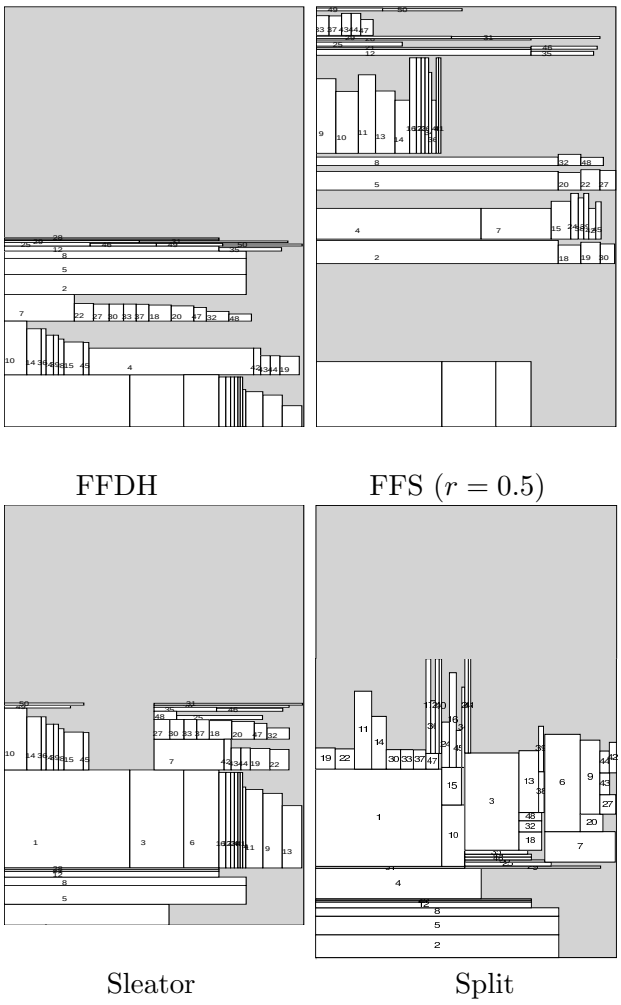

Figure 24.10. Packings of a *Path*.50 data set

Table 24.1. Mean packing heights for eight heuristics applied to *Nice.n* data sets. Optimum height is 100.

Heuristic	Nice.25 (50 sets)	Nice.50 (50 sets)	Nice.100 (50 sets)	Nice.200 (50 sets)	Nice.500 (10 sets)
NFDH	134.2	125.9	120.4	115.0	109.0
FFDH	130.4	121.9	117.7	113.1	108.2
BFDH	130.4	121.9	117.7	113.1	108.2
WFDH	132.6	123.6	119.0	114.0	109.0
NFS 0.5	210.8	195.7	193.3	173.4	168.2
NFS 0.6	210.2	189.9	181.3	165.3	154.7
NFS 0.7	211.5	192.3	176.5	165.3	151.2
FFS 0.5	206.5	189.1	189.4	169.8	167.4
FFS 0.6	207.8	185.5	176.9	162.8	151.7
FFS 0.7	209.4	190.1	174.9	163.2	150.2
Split	141.8	140.4	138.1	138.6	138.0
Sleator	133.8	125.5	119.2	113.9	108.5

Table 24.2. Mean packing heights for eight heuristics applied to *Path.n* data sets. Optimum height is 100.

Heuristic	Path.25 (50 sets)	Path.50 (50 sets)	Path.100 (50 sets)	Path.200 (50 sets)	Path.500 (10 sets)
NFDH	152.9	157.0	154.9	152.4	144.7
FFDH	149.4	149.9	149.6	147.8	142.0
BFDH	149.4	149.9	149.6	147.8	142.0
WFDH	150.4	152.3	151.6	148.8	142.8
NFS 0.5	223.1	241.3	250.7	255.8	246.7
NFS 0.6	241.9	266.8	268.4	269.1	280.0
NFS 0.7	290.5	310.6	316.3	308.8	294.0
FFS 0.5	219.4	238.7	245.8	252.6	242.0
FFS 0.6	240.7	265.9	266.1	266.1	276.5
FFS 0.7	289.7	309.4	314.9	307.8	293.2
Split	169.4	170.2	168.3	165.1	158.6
Sleator	145.7	139.4	137.9	134.3	125.4

Table 24.3. Mean packing heights for eight heuristics applied to preprocessed ($W \geq H$) *Nice.n* data sets. Optimum height is 100.

Heuristic	Nice.25 (50 sets)	Nice.50 (50 sets)	Nice.100 (50 sets)	Nice.200 (50 sets)	Nice.500 (10 sets)
NFDH	126.1	120.5	114.9	110.9	105.4
FFDH	118.9	115.5	110.7	108.2	105.4
BFDH	118.9	115.5	110.7	108.3	105.4
WFDH	121.4	117.3	111.7	109.2	105.7
NFS 0.5	193.2	175.2	172.4	161.8	163.3
NFS 0.6	177.2	170.7	160.8	150.0	144.0
NFS 0.7	167.4	160.2	150.9	144.5	134.0
FFS 0.5	184.8	168.6	167.7	158.4	160.2
FFS 0.6	168.6	164.0	157.3	146.5	141.5
FFS 0.7	163.4	155.9	148.0	141.6	132.2
Split	138.3	137.1	137.4	138.2	138.9
Sleator	138.0	127.7	119.3	113.6	108.6

Table 24.4. Mean packing heights for eight heuristics applied to preprocessed ($W \geq H$) *Path.n* data sets. Optimum height is 100.

Heuristic	Path.25 (50 sets)	Path.50 (50 sets)	Path.100 (50 sets)	Path.200 (50 sets)	Path.500 (10 sets)
NFDH	132.2	131.2	130.7	125.8	118.0
FFDH	121.4	120.6	118.1	115.3	110.6
BFDH	121.1	120.3	118.0	115.2	110.6
WFDH	123.0	117.3	120.8	117.5	112.6
NFS 0.5	193.7	197.5	199.9	191.3	174.0
NFS 0.6	180.9	183.8	184.0	179.2	159.2
NFS 0.7	174.2	183.7	179.0	172.6	166.7
FFS 0.5	187.2	188.5	187.6	180.0	166.0
FFS 0.6	177.5	177.7	174.0	170.5	153.0
FFS 0.7	172.5	181.0	173.8	167.7	159.9
Split	129.6	134.3	136.4	137.4	137.9
Sleator	131.3	134.8	133.7	130.0	122.8

decreasing height in the level algorithms and by decreasing width in the Split algorithm. Tables 24.3 and 24.4 show the mean packing heights when a preprocessing of the sets of rectangles is applied so that tall rectangles are rotated through 90^o, i.e. the rectangles are reoriented so that their width W is greater than or equal to their height H for each rectangle. Tables 24.5 and 24.6 show the mean packing heights when rectangles are first preprocessed by rotating wide rectangles through 90^o. Following the preprocessing in the two later cases, the level-oriented, Sleator, and Split algorithms will again sort the rectangles by decreasing height or width as before.

Table 24.5. Mean packing heights for eight heuristics applied to preprocessed ($H \geq W$) *Nice.n* data sets. Optimum height is 100.

Heuristic	Nice.25 (50 sets)	Nice.50 (50 sets)	Nice.100 (50 sets)	Nice.200 (50 sets)	Nice.500 (10 sets)
NFDH	131.5	124.7	118.9	113.5	109.2
FFDH	125.9	123.8	117.3	112.5	108.5
BFDH	125.9	123.8	117.3	112.5	108.5
WFDH	127.7	124.3	117.8	112.8	108.7
NFS 0.5	212.4	188.9	196.9	174.6	168.5
NFS 0.6	198.6	192.0	178.2	163.6	158.8
NFS 0.7	202.4	191.4	181.8	161.0	149.9
FFS 0.5	208.5	187.0	194.8	171.8	167.2
FFS 0.6	197.2	190.2	176.2	161.8	157.3
FFS 0.7	202.4	191.0	180.5	160.2	148.1
Split	137.4	136.6	136.5	137.0	138.6
Sleator	127.8	121.7	116.5	112.1	108.1

From the tables, we can see that for the unrotated data sets, the NFDH, FFDH, BFDH and WFDH level heuristics and Sleator's heuristic tend to perform the best. Since the shelf heuristics are on-line algorithms, they are expected to do worse than the others. The shelf heuristics perform less effectively when there are larger variations of rectangle shapes in the pathological data sets as compared to the *Nice.n* sets.

It is also clear from examining Tables 24.3 and 24.4, that preprocessing the data by rotating the rectangles to ensure that $W \geq H$ will improve the results significantly for the level-oriented and shelf heuristics. Sleator's algorithm produces consistent solutions despite the changes in orientation for both types of data sets.

In addition, one can observe that the level-oriented, shelf, and Sleator's heuristics become more effective as the problem size increases, but the

Table 24.6. Mean packing heights for eight heuristics applied to preprocessed ($H \geq W$) *Path.n* data sets. Optimum height is 100.

Heuristic	Path.25 (50 sets)	Path.50 (50 sets)	Path.100 (50 sets)	Path.200 (50 sets)	Path.500 (10 sets)
NFDH	151.6	151.9	153.6	150.6	147.3
FFDH	149.1	149.3	150.3	148.9	146.5
BFDH	149.1	149.3	150.3	148.9	146.5
WFDH	149.5	149.7	151.0	149.3	146.8
NFS 0.5	209.5	226.6	241.1	261.6	251.0
NFS 0.6	240.1	259.2	252.8	264.5	272.0
NFS 0.7	300.9	313.8	304.1	301.8	301.8
FFS 0.5	209.5	224.6	239.6	260.9	251.1
FFS 0.6	240.1	259.2	252.2	264.4	270.1
FFS 0.7	300.9	313.8	304.1	301.8	301.8
Split	163.4	162.2	160.2	155.6	153.7
Sleator	143.1	135.0	131.7	126.8	124.1

Split algorithm performs consistently throughout the range of problem instances tested. Overall, the best solutions to the problems were obtained when the rectangles were rotated so that $W \geq H$ and the FFDH or BFDH heuristic is used. They produced very similar packings.

Comparing the Best Heuristics with our GA

Table 24.7 compares the results of our GA to the best packings obtained by the simple heuristic algorithms for the test data sets. For each category of data, single runs of the GA were applied to each problem, and the means and standard deviations calculated. In each case, the GA was allowed to run until no improvement to the best solution had been observed for 100 generations. Using a population size of 1,000, the runtimes for the GA took only a few minutes for the smaller problems of up to 100 rectangles, but for the larger problems, runtimes were unpredictable because of our flexible stopping condition and could take several hours.

Parameters such as the population size and stopping criterion were arrived at after some initial experimentation with our GA, with the aim of producing good quality solutions in a reasonable amount of time. These settings could be viewed as somewhat arbitrary and better results can certainly be obtained if longer run times are used. However, in our view it was difficult to justify the vast computing resources that would

be required to lift the solution by a small percentage when considering that the simple heuristics ran in a fraction of a second.

Table 24.7. Comparison of GA with results from the eight heuristics on rotated ($W \geq H$) data sets. Optimum height is 100.

Data set type	No. of sets	GA mean (std)	Heuristic mean (std)	Heuristic used	sig at 0.1%
Nice.25	50	107.3 (1.6248)	118.9 (4.085)	F/BFDH	yes
Nice.50	50	107.8 (2.0806)	115.5 (2.9384)	F/BFDH	yes
Nice.100	50	108.6 (1.6111)	110.7 (2.1615)	F/BFDH	yes
Nice.200	50	111.3 (2.2485)	108.2 (1.1170)	FFDH	yes
Nice.500	10	120.8 (2.7234)	105.4 (0.5788)	N/F/BFDH	yes
Path.25	50	104.4 (1.6950)	121.1 (7.8641)	BFDH	yes
Path.50	50	108.5 (3.9577)	120.3 (6.6813)	BFDH	yes
Path.100	50	112.6 (7.4351)	118.0 (5.4290)	BFDH	yes
Path.200	50	116.7 (7.6316)	115.2 (4.0875)	BFDH	no
Path.500	10	120.8 (4.3667)	110.6 (1.6088)	F/BFDH	yes

Each row of Table 24.7 compares the GA with the best performing heuristic for a particular category of problem. A Student's t-test reveals highly significant (0.1 %) differences between the mean packing heights of the GA and the best performing heuristic in all cases except for the *Path.200* data sets. The GA outperforms the heuristics for the smaller problems up to and including *Nice.100* and *Path.100* (although its run-time is very much longer), while the heuristics do better on the larger problems. The best performing heuristics are usually FFDH and BFDH, as mentioned previously.

Applying the Heuristics to Some Really Large Problems

To complete our study, the best performing heuristics (FFDH and BFDH) were applied to some very large data sets containing 1,000–5,000 rectangles. Ten data sets were generated for each size and type category. The resulting packing heights are presented in Table 24.8. The heuristics were each applied to the unrotated and rotated data sets ($W \geq H$ and $H \geq W$) as before.

It is clear from Table 24.8 that the results for the FFDH and BFDH heuristics continue to improve (i.e. get closer to the optimum of 100) as the data sets get larger. Furthermore, better packing results are observed when the data sets are first preprocessed so that $W \geq H$ for each rectangle, although the preprocessing appears to have much more

Table 24.8. Results for heuristics on large data sets. Means are tabulated for 10 data sets in each category. Optimum height is 100.

Data set type	Unrotated		$W \geq H$		$H \geq W$	
	FFDH	BFDH	FFDH	BFDH	FFDH	BFDH
Nice.1000	106.1	106.1	104.2	104.2	105.8	105.8
Nice.2000	104.5	104.5	103.0	103.0	104.2	104.2
Nice.5000	103.5	103.5	101.9	101.9	103.4	103.4
Path.1000	141.9	141.9	109.4	109.4	141.3	141.3
Path.2000	134.4	134.4	107.2	107.2	135.2	135.2
Path.5000	130.5	130.5	105.1	105.1	131.9	131.9

effect on the *Path.n* data sets than it does on the *Nice.n* data sets. Preprocessing the data set to make $H \geq W$ for each rectangle produces results very similar to those obtained when the heuristics are applied directly to the unrotated data.

5. Summary and Future Work

In this paper, we have tested a number of well-known classical on-line and off-line strip packing heuristics on a range of data sets for which the optimum is known, and we have compared these results with those produced by a good genetic algorithm (GA). Although the GA found better solutions to these problems for the data set sizes up to about $n = 100$ rectangles, some of the classical strip packing heuristics, particularly Best Fit Decreasing Height and First Fit Decreasing Height, performed much better than the GA on these types of data sets as the size of the set increased. Furthermore, the simple heuristics run in a fraction of the time that it takes the GA. We also discovered that the performance of many of the simple heuristics could be enhanced if the data sets undergo a simple preprocessing routine involving the rotation of some of the rectangles by 90^o.

Probablistic studies have been performed, e.g. [Karp et al., 1984] which analyze the expected wasted space of some of these heuristic algorithms when n gets large. As an example, it is known [Coffman and Shor, 1993] that the asymptotic packing efficiency of BFDH approaches 100% for data sets whose rectangles have uniformly distributed heights and widths. Our experiments appear to confirm this predicted asymptotic behavior. A larger study is currently underway to determine if the individual characteristics of these *Nice.n* and *Path.n* data sets have any impact upon the observed performance of the strip heuristics in the

average and worst cases. We also plan to experiment with techniques for seeding an initial population with packings produced using simple heuristics, and then applying our GA with a view to improving upon the heuristic output. In order to do this effectively, however, we may have to remove the locally applied rotation heuristic used by our current GA, as this will probably change the orientations of some of the rectangles packed by the simple strip packing heuristics and produce different results. We also need to investigate suitable methods for injecting the small number of seeded individuals that can be produced using the simple strip packing heuristics into a larger and varied population.

References

Baker, B.S. and Schwarz J.S. Shelf Algorithms for Two-Dimensional Packing Problems. *SIAM Journal of Computing*, 12:508–525, 1983.

Coffman Jr., E. G., Garey, M. R. and Johnson, D. S. Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms. *SIAM Journal of Computing*, 9:808–826, 1980.

Coffman Jr, E. G., Garey, M. R. and Johnson, D. S. Approximation Algorithms for Bin Packing – An Updated Survey In G. Ausiello and N. Lucertini and P. Serafini, editors, *Algorithm Design for Computer Systems Design*, pages 49–106, Springer-Verlag, Vienna, 1984.

Cavicchio, D. J. *Adaptive Search Using Simulated Evolution.* PhD dissertation, University of Michigan, Ann Arbor, 1970.

Coffman Jr., E.G. and Shor P.W. Packings in Two Dimensions: Asymptotic Average-Case Analysis of Algorithms. *Algorithmica*, 9: 253–277, 1993.

Golan, I. Two Orthogonal Oriented Algorithms for Packing in Two Dimensions. Computer Center M.O.D, 1979/311/MHM, P.O. Box 2250, Haifi, Israel, 1979

Golan, I. Performance Bounds for Orthogonal, Oriented Two-Dimensional Packing Algorithms. *SIAM J. Comput*, 10(3):571–582, 1981.

Hopper, E. and Turton, B. C. H. An empirical investigation of meta-heuristic and heuristic algorithms for a 2D packing problem. *European Journal of Operational Research*, 128:34–57, 2001.

Hwang, S. M., Kao, C. Y. and Horng, J. T. On solving rectangle bin packing problems using genetic algorithms. In *Proceedings of the 1994 IEEE International Conference on Systems, Man and Cybernetics*, pages 1583–1590, 1994.

Jakobs, S. On Genetic Algorithms for the Packing of Polygons. *European Journal of Operational Research*, 88:165–181, 1996.

Karp, R. M., Luby M., and Marchetti-Spaccamela, A. Probabilistic analysis of multi-dimensional bin-packing problems. In *Proceedings of the 16th ACM Symposium on the Theory of Computing*, pages 289–298, 1984.

Kröger, B. Guillotineable bin packing: A genetic approach. *European Journal of Operational Research*, 84:645–661, 1995.

Liu, D. and Teng, H. An improved BL-algorithm for genetic algorithm of the orthogonal packing of rectangles. *European Journal of Operational Research*, 112:413–420, 1999.

Oliver, I. M., Smith, D J. and Holland, J. R. C. A study of permutation crossover operators on the traveling salesman problem. In *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230, 1987.

Sleator, D. A 2.5 Times Optimal Algorithm for Packing in Two Dimensions. *Information Processing Letters* , 1:37–40, 1980.

Valenzuela, C. L. and Wang, P. Y. A Genetic Algorithm for VLSI Floorplanning. In *Parallel Problem Solving from Nature — PPSN VI*, Lecture Notes in Computer Science 1917, pages 671–680, Springer, 2000.

Valenzuela, C. L. and Wang, P. Y. VLSI Placement and Area Optimization Using a Genetic Algorithm to Breed Normalized Postfix Expressions. *IEEE Transactions on Evolutionary Computation, Vol. 6, No. 4, pages 390–401, August 2002.*

Wang, P. Y and Valenzuela, C. L. Data Set Generation for Rectangular Placement Problems. *European Journal of Operational Research*, 134(2):150–163, 2001.