TWO DIMENSIONAL BIN PACKING: INNOVATIONS AND STATISTICAL ANALYSIS

by

Todd Braithwaite

A thesis
Submitted to the Faculty of Graduate Studies and Research
through the Department of Economics, Mathematics, and Statistics
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

1997

Abstract

In this thesis, we introduce and analyze a new two dimensional bin packing algorithm. We focus on the problem of packing rectangles with known dimensions into a fixed width, infinite height bin so as to minimize the total height of the packing. To analyze the algorithm we use statistical methods to compare the known optimal pack heights with the pack heights obtained by the algorithm on a set of randomly generated test problems. This gives us a general technique to not only analyze a single algorithm, but also one which may be used to compare existing algorithms. The method to generate test problems is another contribution of this thesis.

Acknowledgements

I would express my gratitude to everyone who helped along the way. Specifically, I would like to thank Dr. Caron and Dr. Hlynka for their help throughout the creation of this thesis and throughout my studies at the University of Windsor. I would also like to thank Dr. Lashkari and Dr. McDonald for serving on my thesis committee.

LIST OF FIGURES

Figure 1.1 Packing of $L = \{p_1, p_2, p_3, p_4\}$ onto a four foot by twelve foot sheet of drywall.

Figure 2.1 An optimal packing of Example 2.1. This is also the partition of a 30 by 10 bin, created by 2D-BPGen.

Figure 2.2 The packing of Example 2.1 given by the NFDH algorithm into a bin of width 10.

Figure 2.3 The packing of Example 2.1 given by the FFDH algorithm into a bin of width 10.

Figure 2.4 The packing of Example 2.1 given by the BL algorithm into a bin of width 10.

Figure 2.5 The packing of Example 2.1 given by the UD algorithm into a bin of width 10.

Figure 3.1 (a) The packing of Example 2.1 given by our new BU algorithm, into a bin of width 10, when piece rotation is allowed. The slack is set to zero.

Figure 3.1 (b) The packing of Example 2.1 given by our new BU algorithm, into a bin of width 10, when piece rotation is allowed. The slack

is set to five.

Figure 3.2 (a) Using the BU algorithm on Example 2.1, we begin our first pack run by selecting the tallest piece and packing in the bottom left corner. This piece defines our pack height.

Figure 3.2 (b) The BU algorithm continues by selecting the remaining bottom row pieces, for the first pack run, and packing them beside the piece already chosen, by decreasing height from left to right.

Figure 3.2 (c) The BU algorithm packs on top of the already packed bottom row pieces. No piece is allowed to be packed higher than the pack run height.

Figure 3.2 (d) The packing of Example 2.1 given by our new BU algorithm, into a bin of width 10, when pieces have fixed orientations.

Figure 3.3 The step by step changes in the column definitions, that occur in the BU algorithm, when creating the first pack run in Fig. 3.2. The column modifications occur between Fig. 3.2 (b) and Fig. 3.2 (c).

Figure 4.1 The probability that each of the four possible partitions of a bin of height 1 and width 3 will be generated by the 2D-BPGen algorithm.

Figure 5.1 A histogram showing the distribution of the number of pieces in a

bin for 10,000 repetitions of a bin of height 20 and width 20. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 5.2 A histogram showing the distribution of the number of pieces in a bin for 10,000 repetitions of a bin of height 40 and width 10. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 5.3 A histogram showing the distribution of the number of pieces in a bin for 10,000 repetitions of a bin of height 10 and width 40. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 5.4 A histogram showing the distribution of the area of the pieces in a bin for 10,000 repetitions of a bin of height 20 and width 20. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 5.5 A histogram showing the distribution of the area of the pieces in a bin for 10,000 repetitions of a bin of height 40 and width 10. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 5.6 A histogram showing the distribution of the area of the pieces in a bin for 10,000 repetitions of a bin of height 10 and width 40. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.1 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 20 and width 20. All pieces have fixed but optimal orientation. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.2 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 20 and width 20. All possible maximum piece size possibilities are considered. Pieces are not allowed to rotate and are given optimal orientations.

Figure 6.3 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 40 and width 10. All pieces have fixed but optimal orientation. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.4 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 40 and width 10. All possible maximum piece size possibilities are considered. Pieces are not allowed to rotate and are given optimal orientations.

Figure 6.5 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 10 and width 40. All pieces have fixed but optimal orientation. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.6 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 10 and width 40. All possible maximum piece size possibilities are considered. Pieces are not allowed to rotate and are given optimal orientations.

Figure 6.7 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 20 and width 20. All pieces have fixed but random orientation. For all bins the maximum piece size allowed was

equal to that of the full bin.

Figure 6.8 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 20 and width 20. All possible maximum piece size possibilities are considered. Pieces are not allowed to rotate, yet are given random orientations.

Figure 6.9 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 40 and width 10. All pieces have fixed but random orientation. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.10 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 40 and width 10. All possible maximum piece size possibilities are considered. Pieces are not allowed to rotate, yet are given random orientations.

Figure 6.11 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 10 and width 40. All pieces have fixed but random

orientation. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.12 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 10 and width 40. All possible maximum piece size possibilities are considered. Pieces are not allowed to rotate, but are given random orientations.

Figure 6.13 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 20 and width 20. All pieces are permitted to rotate. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.14 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 20 and width 20. All possible maximum piece size possibilities are considered. Pieces are allowed to rotate.

Figure 6.15 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 40 and width 10. All pieces are permitted to rotate. For

all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.16 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 40 and width 10. All possible maximum piece size possibilities are considered. Pieces are allowed to rotate.

Figure 6.17 A histogram showing the distribution of the pack heights given by the Bottom-Up algorithm for 10,000 repetitions of a bin of height 10 and width 40. All pieces are permitted to rotate. For all bins the maximum piece size allowed was equal to that of the full bin.

Figure 6.18 A contour plot showing the average performance, of 100 repetitions, of the Bottom-Up algorithm, for a bin of height 10 and width 40. All possible maximum piece size possibilities are considered. Pieces are allowed to rotate.

Figure A.1 The one dimensional pack given by the FFD algorithm for the list $L = \{8, 7, 6, 6, 5, 5, 3, 2, 2, 2, 2\}$.

1 Introduction

1.1 Introduction

In this thesis we introduce and analyze a new algorithm for the two dimensional bin packing problem [3]. We are given a finite list $L = \{p_1, ..., p_n\}$ of rectangular pieces. Each piece, p_i has known height h_i and known width w_i. We wish to "pack" all pieces into a bin with fixed width $W \geq max\{w_i, i = 1, ..., n\}$ and infinite height. Furthermore, we wish to find the packing that will minimize H, where H is such that the packed pieces fit into a W by H bin so that no rectangles overlap. Numerous applications of this type are discussed in [3]. We consider two variations of this problem. In one we allow 90° rotations, thus giving each piece two possible orientations. Pieces are only allowed to rotate if they will still fit in the bin. In the other we do not allow the pieces to rotate.

The problem of two dimensional bin packing is a generalization of the one dimensional bin packing problem (see Appendix A) when instead of minimizing the bin height we fix the bin height and minimize the total number of bins. If all the rectangles have the same width as the bin, i.e., if $w_i = W$, $\forall i$, then the two dimensional problem (without rotation) is equivalent to the one dimensional problem. The problem of one dimensional

bin packing is known to be NP-complete [12]. Thus, the two dimensional problem is also NP-complete (see Appendix B). For this reason, we have limited our research to a fast heuristic for the problem. In chapter 2 we will present some of the existing heuristic algorithms. We introduce our new algorithm, which we call the Bottom-Up algorithm in chapter 3.

With all these algorithms we are led to the question, "Is one algorithm better than another?". One way to address this question is to compare bounds on the extent to which a solution from an algorithm can deviate from optimality. The smaller the bound, the better the algorithm. These worst case bounds give us information about an algorithm, however these bounds are often difficult to find. It is also not reasonable to state that one algorithm is superior to another based solely on their respective worst case bounds. It is quite possible for one algorithm to have a smaller worst case bound than another, yet produce lower quality solutions on almost all test problems. Instead of using worst case bounds to characterize the performance of our new algorithm we create large test problem sets, with bins of known optimal height, so that we can analyze the average performance of the algorithm. The algorithm used to create these problem sets is new. This test problem generator, named 2D-BPGen, is explained in chapter 4. We discuss three test problem sets, created by 2D-BPGen, in chapter 5.

In chapter 6 we make use of our three test problem sets to analyze the average performance of the Bottom-Up algorithm. Within this section we give a method to compare two algorithms according to their performance on a set of test problems. We will use our method to compare our two variations of the Bottom-Up algorithm.

1.2 Motivation

The Bottom-Up algorithm was developed to solve a specific application of two dimensional bin packing. The problem was to reduce the waste involved in the installation of drywall in residential units. Drywall sheets have a fixed width of four feet and can range in height from eight feet to twelve feet, in one foot intervals. We modeled the sheets as bins. The first step was to partition the walls into pieces which were small enough to fit onto a single drywall sheet. All the piece dimensions were integers with inches as the unit of measurement. Thus, for all i, $w_i \leq 48$ and $h_i \leq 144$. Typically $h_i > w_i$. The pieces were placed in either the group with $w_i < 48$ or the group with $w_i = 48$. The pieces with $w_i < 48$ were packed into new four foot wide bins using the Bottom-Up algorithm. The heights of these bins were at most twelve feet. These bins were then treated as new pieces with $w_i = 48$. The original pieces with $w_i = 48$ and the newly created pieces with $w_i = 48$ were

then packed onto four foot by twelve foot drywall sheets using the First Fit Decreasing algorithm (FFD), which is discussed in appendix A. After all the pieces were packed, the sheets were reduced to the smallest possible drywall sheet height.

In Figure 1.1 we show a list $L = \{p_1, ..., p_4\}$ and how this list would be packed onto a sheet of drywall with $H = 144$ and $W = 48$. The piece p_1 has a width, w_i, equal to 48. This piece is handled after the other pieces. Pieces p_2, p_3, and p_4 all have width's, w_i, less than 48. These pieces are packed onto a newly created four foot wide piece, labeled p_5. We now pack pieces p_1 and p_5, using one dimensional bin packing techniques onto our drywall sheet.

1.3 Goals

The main goals of this thesis are to introduce and analyze a new two dimensional bin packing algorithm, to present a new two dimensional bin packing test problem generator, and to develop a statistical technique to compare algorithms.

In [17] it is stated that "it is usually difficult to evaluate and compare the performance of heuristic algorithms, other than by running them on large problem sets with known optimal solutions." Yet this technique is often not

included when an algorithm is analyzed. In fact, this author was unable to find a problem set with known optimal solutions. This was the motivation for our development of the 2D-BPGen algorithm to generate random two dimensional test problems with known optimal pack heights. The 2D-BPGen algorithm allows for bins of varying width and optimal pack height, thus allowing us to vary the types of test problems sent to the Bottom-Up algorithm. This problem generator will be defined in detail in chapter 4. We will also attempt to give some feel as to the types of test problems that are created by the 2D-BPGen algorithm. The final thing we aim to do is to give a statistical method of comparing algorithms. To illustrate this method we make use of it to compare two variants of the Bottom-Up algorithm.

2 Literature Survey

There are many existing two dimensional bin packing algorithms that can be found in present literature. A few of these algorithms are, Bottom-Left [3], First Fit Decreasing Height [6], Next Fit Decreasing Height [6], Split-Fit [6], LFOLD [11], and Up-Down [2]. We will give a brief outline of the First Fit Decreasing Height, Next Fit Decreasing Height, Bottom-Left and Up-Down algorithms. The algorithms are compared mostly by their respective worst case bounds. These bounds tell us the extent to which the heuristic solution can deviate from optimality. These bounds are discussed in [6] and explained in the following manner. For L, an arbitrary list of rectangles, all assumed to have width no more than 1, let OPT(L) denote the minimum possible bin height within which the rectangles in L can be packed, and let A(L) denote the height actually used by a particular algorithm when applied to L. Absolute performance bounds for various algorithms are bounds of the form

$$A(L) \leq \beta \cdot OPT(L)$$

for all lists L. In contrast asymptotic performance bounds are of the form

$$A(L) \leq \beta \cdot OPT(L) + \gamma$$

for all lists L. These bounds are asymptotic in regards to the number of pieces. The difference between these two types of bounds will be illustrated as we discuss other two dimensional bin packing algorithms.

In order to improve the understanding of two dimensional bin packing algorithms we make use of a common example throughout this section. This example will be packed using each of the algorithms.

Example 2.1 : Let $p_i = (h_i, w_i)$. Let $L\star$=$\{(8,6),(21,1),(6,4),(16,3),(14,3),(2,7),(13,4),(6,1),(19,2),(1,7)\}$. For this list, $p_1 = (8,6), p_2 = (21,1), ..., p_{10} = (1,7)$. An optimal pack for this list $L\star$ is shown in Figure 2.1.

2.1 Next Fit Decreasing Height (NFDH)

For the Next Fit Decreasing Height algorithm [6] the list L is ordered by non-increasing height. NFDH is a level oriented algorithm. This means that as the pieces are packed into the bin a sequence of levels are formed. The pieces are packed so that they rest on one of the levels. The first level is defined by the bottom of the bin. (This level is referred to as the bottom row throughout this thesis.) Each ensuing level is defined by a horizontal line drawn at the height corresponding to the top of the maximum height piece placed on the preceding level. NFDH packs the pieces in the order given by L. The pieces are left justified on a given level, beginning with

the bottom of the bin, so that no two pieces overlap. This continues until there is no longer enough space to the right to fit the next piece in the list. Packing on the current level is finished and the next level is now defined. Packing continues on this new level beginning with the piece that would not fit on the previous level. This process continues until all the pieces in L have been packed. The height at which the next level would have been defined is referred to as NFDH(L). This is the height which NFDH uses to pack all the pieces of list L into the given bin. Figure 2.2 shows the packing of Example 2.1 when applying NFDH. To begin the list is ordered so that $L\star = \{p_2, p_9, p_4, p_5, p_7, p_1, p_3, p_8, p_6, p_{10}\}$. The first piece, p_2, is placed at the (0,0) position of the bin. We continue placing pieces, working from left to right, on the same level as p_2 until there is insufficient space remaining to pack the next piece in the list. Thus, we pack pieces p_9, p_4, and p_5 beside p_2. We cannot fit p_7 on this current level. Piece p_7 is placed at $(0,H_1)$, thus defining our next level. The next piece, p_1, is placed to the right of p_7. When we place p_3 we must once again begin a new level. Piece p_8 fits beside p_3 and then p_6 and p_{10} each define a new level.

In [6] the absolute performance bound is given to be

$$NFDH(L) \leq 3 \cdot OPT(L)$$

for all lists L. It has also been proven that there exist lists L for which NFDH(L) is arbitrarily close to $3 \cdot OPT(L)$. The asymptotic performance bound, when the height and the width of each piece is no more than 1 has been proved in [6] to be

$$NFDH(L) \leq 2 \cdot OPT(L) + 1$$

for all L, and the multiplicative constant 2 cannot be improved upon. For Example 2.1 $NFDH(L\star) = 1.4 \cdot OPT(L\star) + 1$.

2.2 First Fit Decreasing Height (FFDH)

The First Fit Decreasing Height algorithm is a level oriented algorithm and is quite similar to the Next Fit Decreasing algorithm. Once again we assume that the list L is ordered by non-increasing height. The difference is that at any point in the packing the next piece to be placed is left justified on the lowest level on which it will fit. Only when this piece will not fit on any of the current levels is a new level started. FFDH(L) refers to the height which FFDH uses to pack all the pieces of list L into the given bin. Figure 2.3 shows the packing of Example 2.1 when applying FFDH. This packing is extremely similar to NFDH, except for the placement of p_8. For this piece we were allowed to look back at all previous levels to find the lowest level where

the piece would fit. The piece p_8 fit, and thus was placed, on the first level. In NFDH we were forced to pack only on the current level or start a new level if the piece did not fit. It is stated in [6] that $FFDH(L) \leq NFDH(L)$ for all lists L. For Example 2.1 $FFDH(L\star) = NFDH(L\star)$.

The absolute performance bound is given to be

$$FFDH(L) \leq 2.7 \cdot OPT(L)$$

for all lists L. This bound is also tight. The asymptotic performance bound, when the height and the width of each piece is no more than 1 has been proved in [6] to be

$$FFDH(L) \leq 1.7 \cdot OPT(L) + 1$$

for all L, and that the multiplicative constant 1.7 is the smallest possible. In fact for all lists of squares it is proven that

$$FFDH(L) \leq 1.5 \cdot OPT(L) + 1$$

and that the multiplicative constant 1.5 is the smallest possible.

2.3 Bottom-Leftmost (BL)

The Bottom-Leftmost algorithm was first introduced in [3]. The BL algorithm packs pieces in the order given by the list L. Each piece is placed

in the lowest possible place in the bin and then left-justified at this height. This type of packing is shown in Figure 2.4 using Example 2.1. It was proved in [3] that if L is ordered on decreasing widths then,

$$BL(L) \leq 3 \cdot OPT(L).$$

In [5] a lower bound result was proven. This result states that there exists a list L for which

$$BL(L) \geq 1.25 \cdot OPT(L).$$

Also it was shown in [3] that if only square pieces are considered then

$$BL(L) \leq 2 \cdot OPT(L).$$

2.4 Up-Down (UD)

For the following description of the Up-Down algorithm [2] we assume that the bin width has been normalized to 1. Subsequently all piece widths have scaled down by the same factor used to normalize the bin width. The UD algorithm divides a list, $L = \{p_1, ..., p_n\}$, into 5 sublists, L_j, according to the piece widths. For $j = 1, 2, 3, 4, L_j = \{p_i | w_i \in (\frac{1}{j+1}, \frac{1}{j}], i = 1, ..., n\}$. These pieces are ordered by non-increasing width. The final sublist, L_5, is a list of the pieces of L with widths at most $1/5$. The pieces of L_5 are

ordered by non-increasing height. The pieces in L_1 are packed by the BL algorithm. The top height which these pieces reach is defined as H_1, where $H_0 = 0$. This region between the heights H_1 and H_0 is defined as R_1. The pieces of $L_j, j = 2, 3, 4$, are packed in one of two manners. If the piece, $p_i \in L_j$ fits in a column along the right side of the bin, from a height H_{j-1} downward then it is packed in this position. Otherwise, the piece is packed using the BL algorithm, starting at a height of H_{j-1}. When all the pieces of L_j are placed, H_j is set as the highest height a piece from L_j reaches. R_j is then defined as the region between H_j and H_{j-1}. The pieces of L_5 are packed only after all the pieces in the other sublists have been packed. They are placed using a generalized NFDH algorithm, named GNFDH [2]. This algorithm packs pieces in between the BL pieces and the column pieces in a region $R_j, j = 1, 2, 3, 4$. As soon as any piece does not fit using the GNFDH algorithm all remaining pieces are packed using the NFDH algorithm at the height H_4. The overall pack height used by UD is H_5. For a more detailed description of the UD algorithm we refer the reader to [2]. In Figure 2.5 we show the packing of our list $L\star$, from Example 2.1, as given by the UD algorithm. For this example $L_1 = \{p_6, p_{10}, p_1\}$. These pieces are packed, using the BL algorithm, between the heights H_0 and H_1. We then pack the pieces of L_2, where $L_2 = \{p_3, p_7\}$. The piece p_3 is packed in a column along

the right side of the bin, from the height H_1 downward. p_7 will not fit in the column down along the right side of the bin and thus is packed using the BL algorithm at the height H_1. For L_3 both p_4 and p_5 are too tall to fit along the right side of the bin. Again the BL algorithm is used to pack these pieces, this time at a height H_2. The sublist L_4 is empty and thus $H_4 = H_3$. The remaining pieces, p_2, p_9, and p_8, are in L_5. We attempt to pack these pieces using the GNFDH algorithm. The first piece considered is p_2. Since this piece will fit in any of the spaces between the BL pieces and the column pieces it is packed at the height H_4 by the NFDH algorithm. All remaining pieces, p_9, and p_8 are also packed using the NFDH algorithm at the height H_4. It should be noted that p_8 would fit at a lower height in the pack, however since p_2 did not fit at a lower height no remaining piece in L_5 is allowed to be packed using the GNFDH algorithm.

It was proved in [2] that for any list L of rectangles of height at most H,

$$UD(L) \leq \frac{5}{4} \cdot OPT(L) + \frac{53}{8} \cdot H.$$

Moreover, the asymptotic bound of 5/4 is tight. As stated in [2] this bound is an improvement of the previous best algorithm which had a bound of 4/3. In our example $UD(L\star) = 61$, $OPT(L\star) = 30$, and $H = 21$. Therefore, $\frac{5}{4} \cdot OPT(L\star) + \frac{53}{8} \cdot H = 176.625$.

2.5 Summary

One of the main differences between the Bottom-Up algorithm, which will be introduced in the next chapter, and the other algorithms discussed is the fact that the BU algorithm uses piece area as a factor for choosing the packing order. This approach was used with the drywall application in mind. On a drywall sheet the longest edges are beveled at the factory. These edges are called factory edges. These factory edges give the wall a better finish by giving a nice recess for filling the joints between drywall pieces. For this reason it is desirable to have certain pieces placed along the factory edges of the drywall sheet. These pieces we refer to as factory edge pieces. Most of these pieces are skewed with h_i much larger than w_i. These pieces were handled in a special manner, in an attempt to best make use of the edges of the drywall sheet. This means that we do not want to pack all the tall and thin pieces on the same pack run. Therefore we do not want the piece height to be the deciding factor as to the packing order. At the same time we want to always place the largest piece possible so as to best make use of the drywall sheet area. For these reasons the piece area was used as the main deciding factor for the packing order.

Also of importance when designing the Bottom-Up algorithm was to

insure that before any area was left empty that every remaining piece in the list had been considered to make use of the space. In both the NFDH and FFDH algorithms the space above the pieces packed at any given level height and below the ensuing level height was never considered for piece placement. The BL algorithm always packs a piece at the lowest height possible and thus always considers all remaining space before placing a piece. This approach however seemed unnatural. If a space was left at a low height and a piece remaining in the list would fit why then was it not placed immediately instead of waiting to see if the piece would still fit at a later point in time? Let us look back to the packing of Example 2.1 given by the BL algorithm in Figure 2.4. After we have packed pieces p_1 to p_4 we have a space 2 units wide between p_3 and p_2. At this point p_9 would fit very well in this space. The problem is that before we can pack p_9 we must pack pieces p_5, p_6, p_7, and p_8. By the time these pieces are handled p_9 will no longer fit in this space. Some of this space is used by p_8, however p_9 would have been a better fit. From examples like this it was felt that the order of the pieces in a list was far too important to the overall pack given by the BL algorithm. How would one decide how to order the pieces?

The Up-Down algorithm attempts to make use of any remaining space at lower levels before placing a piece at a higher level. However this algo-

rithm would not work for our drywall application for one main reason. As previously mentioned the drywall application was often called on to pack tall, thin pieces. If the tallest piece of a given list is also the thinnest piece the UD algorithm will always pack this piece close to the top of the pack. This practice will often lead to poor results. In Example 2.1 p_2 is the tallest and thinnest piece. In Figure 2.5 we can see how such a piece can result in a poor pack.

For all the reasons discussed above it was felt that we would be better served to create a new two dimensional algorithm which better suited our needs. The Bottom-Up algorithm is the result of this decision.

3 Bottom-Up Algorithm

The Bottom-Up algorithm is designed to give a two dimensional packing of a given set of rectangular pieces. The user can decide if they wish to fix all piece orientations or to allow all the pieces to rotate 90 degrees. Once the algorithm chooses the position of a piece, that piece is no longer allowed to change location.

The Bottom-Up algorithm was created by using the FFDH and NFDH algorithms as templates. Like both the FFDH and NFDH algorithms, the BU algorithm is level oriented. However, unlike the other two algorithms, the BU algorithm makes use of the spaces between levels. Another added feature of the BU algorithm is that the level heights do not have to be equal to the tallest piece on the level. The level height varies according to the "slack" the user defines. In this thesis we will only test the algorithm under the condition where "slack" is set to zero.

If we allowed non-zero slack then any piece would be allowed to surpass the level height by at most the slack value. If this occurred then the level height would be updated to the height reached by this given piece. If we refer to Figure 3.1 we can see how the pack of Example 2.1 differs when the slack variable is changed. In (a) we see the pack of the list $L\star$ with the

slack variable equal to zero. In (b) we set the slack variable equal to 5. In both packs we allow rotation. It is important to note how the level heights in (a) are all equal to the tallest piece packed on the pack run. In (b) the pack run heights are allowed to be larger than the tallest piece. An example of this can be seen in the first pack run. The piece p_8 is packed above p_9, causing the pack run height to increase. Although it may first seem that a non-zero slack would be beneficial to the overall pack, in many cases it was found that a large slack can give worse results. This is shown in Figure 3.1.

In the following subsections we describe the packing approaches used by the Bottom-Up algorithm for the two variations of two dimensional bin packing previously mentioned. We begin by describing how the Bottom-Up algorithm packs a list when piece orientations are fixed. We use this description and Example 2.1 to better illustrate the algorithm. Following this the algorithm outline for fixed piece orientation is given. We then explain how the algorithm differs when 90 degree piece rotation is permitted. Example 2.1 is again used to better highlight these differences.

3.1 No Rotation

Before we begin to pack, the pieces are sorted by non-increasing height. Any ties are sorted by non-increasing width. Packing starts by placing pieces into

what we shall refer to as a pack run. The width of a pack run is equal to the bin width. The height of a pack run is defined by the first piece chosen to be placed on it. This piece has the largest height of all the pieces remaining to be packed. For the first pack run this piece is p_1. This first piece is placed in the bottom corner of the pack run. The remaining pieces for the bottom row are chosen in the following manner. We find the minimum i such that the w_i is less than or equal to the remaining length along the bottom of the pack run and the area of p_i is not smaller than any other such piece. This continues until the remaining length along the bottom of the pack run is either zero or too small to permit any piece to fit. These bottom row pieces are now sorted by non-increasing height and arranged from left to right along the bottom of the pack run in the exact manner of the NFDH algorithm. This creates a stair-like structure, with the top step to the far left. No piece will be allowed to be packed at a height which places any of the piece higher than the height of the pack run. Any piece which has been packed is deleted from L. With this new list we proceed to pack up, into the open area remaining between the bottom row pieces and the pack run height. To do this we define columns. Any "step" of the stairs defines a column. If there is any remaining area left in the bottom row this space is merged with the column to its left. This is done since we have already

found that no piece will fit in this space. If a piece had fit it would have been selected along with the other bottom row pieces. Also, if any neighboring columns have the same height, they are merged into a single column. We attempt to pack on the lowest column (step). We search our list for the piece which uses the maximum amount of area defined by the column width and the difference between the column height and the pack run height. This piece is positioned on the column so that it is next to either a side of the pack run or the tallest neighboring column. If the width of the piece is equal to the width of the column then the column height is simply increased by the amount of the piece height. If the width of the piece is less than the width of the column then the piece defines a new column. In this case a new column is inserted and the dimensions of the old column are corrected accordingly. If this new column is beside a column of the same height, then these two columns are merged. At any point if no piece will fit in the given space defined between the column and the pack run height, then the column is merged with its lowest neighboring column. If there is a placed piece, it is deleted from the list and the process of building up the lowest column is continued. We finish a pack run when we have only one column at a height equal to the pack run height.

If there are any remaining pieces which have yet to be packed, we create

a new pack run by the same process as described above. This creates a set of pack runs. These pack runs are now placed in the bin, one on top of the other. You could think of this as a one-dimensional bin packing of the pack runs. However, since we are only dealing with one bin, the order in which the pack runs are placed in the bin is not important. At this point we have packed all the pieces into our bin and our resulting bin height is found by taking the sum of all the pack run heights.

Figure 3.2 gives a step by step packing of Example 2.1. Before we begin packing, the pieces are sorted by non-increasing height. If more than one piece has the same height then these pieces are also sorted by non-increasing width. In (a) we begin packing the first pack run by selecting the tallest piece, p_2, and packing it in the position (0,0). We set the pack run height equal to the height of p_2. We then select pieces p_7, p_4 and p_9. We place these pieces beside p_2, working from left to right in the order of non-increasing height. This is shown in (b). These bottom row pieces define our columns. The columns, and how they are modified throughout the packing of this first pack run, are given in Figure 3.3. Our shortest column, column 4, is defined by p_7. We now select p_3 since it is the piece with the largest area which fits in the 8 by 4 space above p_7. Since this piece uses the whole column width, the column height is increased by the piece height to a value of 19. Once

again we find the shortest column, column 3, which is defined by p_4. When we search for a piece to fit in the 5 by 3 space above column 3, we find that no such piece exists in our given list. Since no piece fits above this column we merge this column with column 2, which is defined by p_9. The width of column 2 is increased so that it covers column 3. We now set column 3 equal to column 4 and delete column 4. In other words we now have only 3 columns, instead of 4. Now, columns 2 and 3 are the same height and thus we again merge columns. The width of column 2 is increased so that it covers column 3 and column 3 is deleted. This leaves us with 2 columns. Column 2 has the smallest height and thus we search for the piece with the largest area that fits in the 2 by 9 space above the column. This piece is p_6. Since column 2 is the right-most column p_6 is packed to the far right side of the bin. Since this piece does not use the full column width of column 2 we must insert a new column. We define column 3 to have the height of column 2 plus the height of p_6. The width of column 3 is equal to the width of p_6 and the x-coordinate of column 3 is set as the left x-coordinate of p_6. The width of column 2 is redefined by subtracting the width of p_6. Once again column 2 is our shortest column. No piece will fit in the space above column 2 and we again merge columns. The end result is that we are left with only 1 column, where the column height is equal to the pack run height. This

tells us that we are finished packing on this pack run. This final pack of the first pack run is given in Figure 3.2 (c). Since we still have pieces remaining we must begin a new pack run, which we handle in the same manner as the first pack run. This second pack run, as well as the final pack of $L\star$, is shown in Figure 3.2 (d).

3.1.1 Bottom-Up Outline For No Rotation

We are given a set $L = \{p_i | i = 1, ..., n\}$ where each piece p_i has height h_i and width w_i. We assume that $h_1 \geq h_2 \geq ... \geq h_n$ and that if $h_i = h_j$ then $w_i \geq w_j$. We are also given a bin width, W. The position at which each piece is packed into the bin is given by the (x,y)-coordinates of its bottom left corner, with the origin in the bottom left corner of the bin.

Variable Explanations:

t: Pack run number.

H_t: Height of pack run t.

$L\prime$: Set consisting of all packed pieces.

$L\prime\prime$: Set consisting of all bottom row pieces.

q: Number of columns.

X: x-coordinate for positioning bottom row pieces.

R: Remaining unused width of the bottom row.

$position_i$: (x,y)-coordinate position of piece p_i.

$height_q$: Height of column q.

$width_q$: Width of column q.

x_q: x-coordinate of the left corner of column q.

c_q: Vector consisting of all the information for column q.

Algorithm Outline:

$t = 0$

$H_0 = 0$

$L = \{p_1, ..., p_n\}$

$L\prime = \emptyset$

$q = 0$

while $\|L\prime\| < n$

begin

$L\prime\prime = \emptyset$

$X = 0$

$t = t + 1$

$R = W$

$i = \min\{j | p_j \notin L\prime, j = 1, ..., n\}$

$H_t = h_i$

$R = R - w_i$

$L\prime = L\prime \cup \{p_i\}$

$L\prime\prime = L\prime\prime \cup \{p_i\}$

while $\exists p_j \ni w_j \leq R, p_j \notin L\prime, j = 1, ..., n$

begin

$i = \min\{j | p_j, p_k \notin L\prime; w_j \leq R; h_j w_j \geq h_k w_k; j, k = 1, ..., n\}$

$R = R - w_i$

$L\prime = L\prime \cup \{p_i\}$

$L\prime\prime = L\prime\prime \cup \{p_i\}$

end

while $\| L\prime\prime \| > 0$

begin

$i = \min\{j | p_j, p_k \in L\prime\prime; h_j \geq h_k; j, k = 1, ..., n\}$

$position_i = (X, H_{t-1})$

$X = X + w_i$

$L\prime\prime = L\prime\prime \setminus \{p_i\}$

end

$for\ i = 1\ to\ n$

$begin$

$if\ p_i \in L\prime\ then$

$begin$

$q = q + 1$

$height_q = h_i$

$width_q = w_i$

$x_q = position_i \cdot (1, 0)$

$c_q = (height_q, width_q, x_q)$

end

end

$if\ R \neq 0\ then$

$width_q = width_q + R$

$if\ \exists r \ni height_r = height_{r+1}, r = 1, ..., (s-1)\ then$

$begin$

$width_r = width_r + width_{r+1}$

$for\ u = (r+1)\ to\ q$

$c_u = c_{u+1}$

$q = q - 1$

end

while $q > 1$

begin

$m = \min\{r | height_r \leq height_z, r, z = 1, ..., s\}$

if $\exists p_j \ni w_j \leq width_m and h_j \leq H_t - height_m$ *then*

begin

$i = \min\{j | p_j, p_k \notin L\prime; w_j \leq width_m; h_j \leq H_t - height_m; h_j w_j \geq h_k w_k; j, k = 1, ..., n\}$

if $m \neq q$ *then*

$position_i = (x_m, H_{t-1} + height_m)$

else

$position_i = (x_m + width_m - w_i, H_{t-1} + height_m)$

$L\prime = L\prime \cup \{p_i\}$

if $w_i = width_m$ *then*

$height_m = height_m + h_i$

else

begin

if $m \neq q$ *then*

begin

$q = q + 1$

$for\ u = s\ to\ (m+1), step-1$

$c_u = c_{u-1}$

$c_m = (height_m + h_i, w_i, x_m)$

$c_{m+1} = (height_m, width_m - w_i, x_m + w_i)$

end

$else$

$begin$

$q = q + 1$

$c_{q-1} = (height_{q-1}, width_{q-1} - w_i, x_{q-1})$

$c_q = (height_{s-1} + h_i, w_i, W - w_i)$

end

end

end

$else$

$begin$

$if\ m = q\ then$

$begin$

$c_{q-1} = (height_{q-1}, width_{q-1} + width_q, x_{q-1})$

$q = q - 1$

end

else

begin

if $height_{m-1} \leq height_{m+1}$ *then*

$c_{m-1} = (height_{m-1}, width_{m-1} + width_m, x_{m-1})$

else

$c_{m+1} = (height_{m+1}, width_m + width_{m+1}, x_m)$

for $u = m$ *to* $(q-1)$

$c_u = c_{u+1}$

$q = q - 1$

end

if $\exists r \ni height_r = height_{r+1}, r = 1, ..., (s-1)$ *then*

begin

$width_r = width_r + width_{r+1}$

$c_u = c_{u+1}, u = (r+1), ..., s$

$q = q - 1$

end

end

end

end

3.2 Rotation

If rotation is allowed, each piece is set to the orientation in which $h_i \geq w_i$. As in the previous algorithm we sort our list by non-increasing piece height. Ties are again handled by sorting on non-increasing piece width. The first piece to be chosen for a pack run is the tallest remaining piece. The remaining bottom row pieces are selected in the same manner that they were in the no rotation algorithm. We find the largest area piece whose width is no larger than the remaining bin width. This continues until the bin width is completely used or no piece will fit in the remaining width. At this time the pieces are placed, once again in decreasing height, from left to right. Take note that our pack may already differ from the pack given in the no rotation algorithm. This is due to the fact that some pieces in our list were given different orientations to begin with. It should be noted that, if at this point there is only one piece on the pack run, the piece is rotated (if possible) so that the pack run height is minimized.

Once again we use our bottom row pieces to define columns. We select

the lowest column and search our list for a piece to fit on top of this column. Previously, in the no rotation algorithm, when selecting this piece, we only checked that the width of a piece was no larger than the width of the column and the height of a piece was no larger than the difference between the pack run height and the column height. However, we now allow for piece rotation and must check the possibility that a piece may fit, in the area above the column, if we change its orientation. Thus we select the piece with the maximum area such that at least one orientation of the piece will fit in the area above the column. If both orientations are feasible we choose the orientation so that the smaller dimension is placed parallel to the bottom of the bin. This process of building up the lowest column continues until all columns have been merged into one, or all pieces have been packed. If any pieces remain we define a new pack run. We are again left with a set of pack runs, which we pack one on top of another in our bin. The sum of the pack run heights gives us the overall pack height for the algorithm.

In Figure 3.1 (a) we show how the Bottom-Up algorithm packs Example 2.1. It should be noted that before we begin packing, pieces p_6 and p_{10} are rotated so their heights are greater than their widths. This causes a slightly different order, when we sort on non-increasing height, than we had when we packed Example 2.1 with no rotation. The pack of $L\star$ with rotation is

almost identical to that of no rotation, shown in Figure 3.2. Other than how the pieces are ordered before packing, there are three differences in the pack allowing rotation. First, p_6 is rotated back to its original orientation so that it will fit on the first pack run. Secondly, p_{10} is packed in its new orientation and is selected before p_8, since it has a larger area. Finally p_8 is packed above p_{10}.

4 Test problem creation

In order to analyze the Bottom-Up algorithm it became necessary to be able to create randomly generated test problems which had known optimal packing heights. This was essential since at the time of this paper no set of test problems could be found for two-dimensional algorithms. Thus the 2D-BPGen algorithm was developed. It should be stated that the test problems created by this algorithm are in no means meant to be a benchmark for two-dimensional bin packing. However it is this author's intent that the test problems given by this random partitioning algorithm, and the 2D-BPGen algorithm itself, be available for the purpose of comparison between algorithms.

The 2D-BPGen algorithm takes a bin, of given height and width, and partitions it into a list of randomly selected rectangles which completely fill the bin with no empty space. These rectangles are only allowed to be as large as the given maximum height and width dimensions set by the user. It is important to note here that the partitions created by 2D-BPGen are not all equally likely. To illustrate this fact we define Example 4.1 to be a bin with width three and height one. As shown in Figure 4.1, there exist four distinct partitions. Also shown in the figure are the respective probabilities

that 2D-BPGen will partition the bin in the according fashion. The width of the first rectangle selected has an equal probability of being 3, 2, or 1. Thus there is a probability of 1/3 that our bin will consist of only one piece. If we are given that our first piece chosen has width 2, then there is only one unit remaining and therefore we must select piece of width 1. Therefore our partition of two pieces, where the first piece is of width 2, also occurs with probability 1/3. Both our final two partitions begin with a piece of width 1. At this stage we have to make a choice. Our second piece can either have a width of 2 or a width of 1. The probability we chose a width of 2 is 2/3. If this is the case then our bin is done. This type of partition has a probability of 2/9. If we chose width 1, which has a probability of 1/3, then we must select a third piece with width 1. This type of partition has a probability of 1/9. As can be seen, the probabilities are not equal.

For the 2D-BPGen algorithm the user defines the bin height, H, bin width , W, maximum piece height, h, and maximum piece width, w. The algorithm randomly, using the rand function provided in Microsoft visual C++, selects an integer piece height from one to h and an integer piece width from one to w. These dimensions define the first piece in our list. This piece is placed in the bottom left corner which is defined as (0,0). We then select the rest of our bottom row pieces. We move from left to

right randomly selecting our piece dimensions. The height of these pieces is selected in the exact same manner as the height of the first piece. The width, however, is set to the minimum of the random number selected, from 1 to w, and the remaining bin width on the bottom row. In so doing we guarantee that all of the bin width will be used.

We have now defined our bottom row and must now build up to use the remaining bin height. To do this we use the bottom row pieces to define columns. Each piece defines a column. A column is defined by its height, left most x coordinate, and right most x coordinate. If any two neighboring columns are the same height we merge them into one column. It is now time to begin building up the columns. The smallest column is found and we randomly choose our piece dimensions. The piece height is taken to be the minimum of the difference between the column height and bin height and the random height from 1 to h. The piece width is the minimum between the column width and the random width from 1 to w. This piece is then placed at the left most x coordinate of the column. If this piece does not use the full column width then a new column is inserted as defined by the newest piece and the old columns dimensions are updated. Once again we check neighboring columns to see if any share the same height. If so these columns are merged. This process of building up the lowest column continues until

all columns have reached the bin height and have thus been merged into one column.

Now that we have defined our random partition of a fixed bin we output the information to an input file to be read by a packing algorithm. For each piece the length and width is output. For any algorithm, such as the Bottom-Left algorithm, which does not sort the list L, it is necessary to randomly scramble the order in which the pieces are output. Also it may be necessary to give each piece a random orientation so that we are not giving a bias to any algorithm that does not allow rotation by giving a list containing pieces with "optimal orientation". This consideration of random piece orientation versus optimal piece orientation will be discussed further in chapter 6. Along with the piece lengths and widths we output the number of pieces and the bin width and height of the bin which was partitioned. Any other information needed for the packing algorithm is also output. By repeatedly running this partitioning algorithm we can create a set of randomized test problems with known optimal heights. These test problems can then be given to a packing algorithm to test how close to the optimal packing the algorithm can reach.

Example 2.1 was in fact created by the 2D-BPGen algorithm. The bin height was set to 30, the bin width was set to 10 and the maximum allowable

piece size was set as the full bin. The exact partition given by 2D-BPGen is the optimal pack shown in Figure 2.1. The piece numbers have been scrambled so that no bias was shown to any algorithm. The first piece selected was p_6. The algorithm randomly chose a height of 2 and a width of 7. The next piece was p_4. The height to this piece was randomly chosen as 16. The random width could have been any number from 3 to 10. Any of these values would have given a width of 3 since only 3 units remained on the bottom row. These two pieces made up our bottom row and as such defined our columns. Column 1, defined by p_6, was the lowest and thus we began building up on this column. The height of p_{10} was chosen as 1. The width could have been any number from 7 to 10 since the full column width was used. For p_2 the random width must have been 1 and the height must have been 21. We continued building up by selecting p_1, then p_3. When the random height for p_9 was chosen it could have been any number from 19 to 30. This is because the piece uses the full remaining height. The random number chosen for the width of p_9 could have been any number form 2 to 10 since it to uses the full length available. Again we continue by selecting p_5, p_7 and finally p_8. After p_8 is chosen we only have one column whose height is equal to 30, the height of the bin. This is our signal that our bin has been completely partitioned.

4.1 Test Problem Generator Outline

In this section we give the outline of the 2D-BPGen algorithm. For 2D-BPGen the user inputs the bin width, W, and the bin height, H. This bin height will be the optimal pack height for the list of pieces randomly selected by 2D-BPGen. The user must also give a maximum piece width, w and maximum piece height, h. We wish to create a list $L = \{p_i | i = 1, ..n\}$, where $p_i = (h_i, w_i)$ as defined in chapter3. For all pieces $h_i \leq h$ and $w_i \leq w$. This list L will pack optimally in an H by W bin.

Variable Explanations:

$position_i$: (x,y)-coordinates of p_i within the bin.

$height_r$: Height of column r.

$width_r$: Width of column r.

x_r: the x-coordinate of the left corner of column r.

c_r: $(height_r, width_r, x_r)$, array containing all the information for column r.

$columns$: Number of columns.

R: Remaining unused width of the bottom row.

Algorithm Outline:

$h_1 = random\ number\ from\ 1\ to\ h$

$w_1 = random\ number\ from\ 1\ to\ w$

$p_1 = (h_1, w_1)$

$position_1 = (0, 0)$

$x_1 = 0$

$c_1 = (h_1, w_1, x_1)$

$n = 1$

$R = W - w_1$

$while\ R \neq 0$

$begin$

$n = n + 1$

$h_n = random\ number\ from\ 1\ to\ h$

$w_n = \min(random\ number\ from\ 1\ to\ w, R)$

$p_n = (h_n, w_n)$

$x_n = x_{n-1} + w_{n-1}$

$position_n = (x_n, 0)$

$c_n = (h_n, w_n, x_n)$

$R = R - w_n$

$columns = n$

end

$if\ \exists r \ni height_r = height_{r+1}, r = 1, ..., (n-1)\ then$

begin

$width_r = width_r + width_{r+1}$

$c_t = c_{t+1}, s = (t+1), ..., columns$

$columns = columns - 1$

end

$while\ \min\{height_r, r = 1, ..., columns\} \neq H$

begin

$m = \min\{t | height_t \leq height_u, t, u = 1, ..., columns\}$

$n = n + 1$

$h_n = \min(random\ number\ from\ 1\ to\ h, H - height - m)$

$w_n = \min(random\ number\ from\ 1\ to\ w, width_m)$

$p_n = (h_n, w_n)$

$position_n = (x_m, height_m)$

$if\ w_n = width_m\ then$

$height_m = height_m + h_n$

else

begin

$columns = columns + 1$

$for\ t = columns\ to\ (m+1), step - 1$

$c_t = c_{t-1}$

$c_m = (height_m + h_n, w_n, x_m)$

$c_{m+1} = (height_{m+1}, width_{m+1} - w_n, x_{m+1} + w_n)$

end

$if\ \exists r \ni height_r = height_{r+1}, r = 1, ..., (n-1)\ then$

$begin$

$width_r = width_r + width_{r+1}$

$c_t = c_{t+1}, s = (t+1), ..., columns$

$columns = columns - 1$

end

end

5 Analysis of Test Problem Generator

The 2D-BPGen algorithm was used to create three main sets of test problems. The first set is divided up into three subsets of 10,000 randomly generated test problems each. These subsets correspond to 3 different sized bins. These bins have the following respective dimensions: 20 height by 20 width, 40 height by 10 width, and 10 height by 40 width. For all three of these subsets the maximum piece size is the full bin size. This set of 30,000 problems, which is used later in chapter 6, was used to give an understanding of the types of test problems being created by 2D-BPGen. In Figures 5.1 to 5.3 we give the distribution of the number of pieces per bin for each of the three subsets of test problems. For each we note that 2D-BPGen partitions the bin into approximately 7 pieces most often. We also see that, rarely is any bin partitioned into any more than 20 pieces. This is not all that surprising given that we allow such a large maximum piece size. It is also interesting to note the similarities between the three graphs. In Figures 5.4 to 5.6 we give the distribution of the piece areas for each of the problem subsets. As would be expected the frequencies of the smaller area pieces are high. Since we do not allow any empty space when a bin is partitioned we would expect that many bins would need smaller pieces to fully fill the bin.

An interesting feature of these graphs are the "peaks" and "valleys". Many of these can be explained by the number of factorizations of the given area. For example, an area such as 16 has 5 factorizations, (1,16), (16,1), (2,8), (8,2), and (4,4), which give possible piece dimensions of length and width. An area such as 11 only has 2 such factorizations, (1,11) and (11,1). Another reason for many of the "valleys" is that many of the piece areas, less than bin area, do not give any valid piece dimensions. For example, any piece area which is a prime number greater than the largest bin dimension does not give a valid piece. A piece area of 41 is not possible for a bin of height 20 and width 20. Neither factorizations, (1,41) or (41,1), will fit within the bin. We also again note the similarities in the three graphs.

The second of the three main sets of test problems also consists of three subsets. These subsets again represent the same three bin sizes used in the first set of problems. This time, however each subset consists of 40,000 randomly generated test problems. These 40,000 test problems consist of 100 repetitions of each of the possible maximum piece sizes for the given bin. For the bin sizes we are using there are 400 different maximum piece sizes ranging from 1 by 1 to the full bin. This set of test problems is used in chapter 6 to give average performance values of the BU algorithm under all possible maximum piece size conditions.

The final set consists of 16,000 randomly generated test problems. For all of these problems the maximum piece size has a width of 20 and height of 5. Also all the problems have a bin width of 20. The bin height ranges from 5 to 80 in steps of 5. Each step consists of 1,000 repetitions. The set, like the previous two, is also used in chapter 6.

6 Analysis of Bottom-Up

In this section we analyze two variants of the Bottom-Up algorithm. We first consider the variation when piece rotation is not permitted by the algorithm. For this version of the algorithm we consider two cases. In one case each piece is fixed in the orientation given by the 2D-BPGen algorithm. We refer to this as optimal piece orientation. Given that all pieces have optimal orientation, we know that there exists a pack using the optimal height. For the second case we fix each piece with a random orientation. We then consider the second variation of the Bottom-Up algorithm and allow piece rotations of 90 degrees. In order to perform the analysis of the Bottom-Up algorithm we use the test problem sets created by 2D-BPGen and discussed in chapter5. Three different bin sizes are considered. These bin sizes have the following dimensions: 20 height by 20 width, 40 height by 10 width, and 10 height by 40 width. For each variation of the algorithm we examine the Bottom-Up algorithm's performance in three ways.

Using the first set of test problems we give the distribution of the pack heights achieved by the Bottom-Up algorithm, for each bin size. With these 10,000 test problems we give the mean pack height and the standard deviation. For each of these problems the maximum piece size allowed is equal

to the full bin.

To analyze the BU algorithm under all possible maximum piece size settings, we use the second test problem set. For each maximum piece size the problem set contains 100 randomly partitioned bins. To characterize the algorithm's average performance, under a certain maximum piece size setting, we calculate the average value of the Bottom-Up algorithm's pack height divided by the optimal pack height. For example, if this value is equal to 1.5 we can state that the BU pack height, on average for this problem set, is 1.5 times greater than the optimal pack height. For each of our three bin sizes there exist 400 different maximum piece size settings. The average ratio for each maximum piece size setting, between the BU algorithm height and the optimal height, is shown with the use of a contour plot.

The third and final test problem set is used to find regression equations to characterize the average performance of the Bottom-Up algorithm. These equations are intended to complement the worst case bounds discussed in chapter 2. Instead of an absolute performance bound we give an "absolute average equation", of the form $ALG(L) = \beta \cdot OPT(L)$. For the asymptotic performance bound we give an "asymptotic average equation". This equation is of the form $ALG(L) = \beta \cdot OPT(L) + \gamma \cdot H$, where H is the maximum piece height in the given list L. In order to find these equations we force

the regression through the origin. For each equation we give the multiple R value. It is very important to note that the assumptions of the regression model are not satisfied by the data given by our test problem set. Specifically, the assumptions of normality of the residuals and constant variance are unsatisfied. Thus although the regression equations can be used to obtain a description of the relationship between variables, they cannot be used to test hypotheses about regression parameters.

6.1 No Piece Rotation Allowed

For the following chapter the Bottom-Up algorithm is not allowed to change the piece orientations once the list has been input. Given a list of pieces, with dimensions and orientations, created by the 2D-BPGen algorithm it is certain that there exists a pack such that the pieces will fit into a bin of known optimal height. The fact that it is known that the pieces with these orientations will pack optimally gives a bias to the no piece rotation version of the Bottom-Up algorithm. By allowing a piece to rotate out of its "optimal orientation" we can no longer guarantee that the given list of pieces will still pack into a bin of known optimal height. In order to remove this bias, we randomly select the piece orientations for the given list. A piece is only rotated if it will still fit in the bin, i.e. $h_i \leq W$. We then pack this

modified list. For the rotation algorithm this list of random orientations will be packed the same as the list of optimal orientations. To analyze the Bottom-Up algorithm when no piece rotation is allowed we consider both the possibility of optimal piece orientation and that of random piece orientation. Thus we can observe the effect of optimal piece orientation on the BU algorithm.

6.1.1 Optimal Piece Orientation

For the Bottom-Up algorithm, under the condition that each piece is given optimal orientation, the asymptotic average equation is found to be

$$BU(L) = 1.034 \cdot OPT(L) + 0.3 \cdot H$$

The absolute average equation is

$$BU(L) = 1.061 \cdot OPT(L)$$

For both equations the multiple R value was 0.998.

20 Height by 20 Width Bin In Figure 6.1 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 20, and the maximum piece size allowed is equal to the full optimal bin size.

For this distribution the sample mean is 23.28. The sample standard deviation is 2.958. Figure 6.2 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 20 and width 20, given optimal piece orientation, achieves reasonable results except when the maximum piece height is relatively large in comparison to the optimal height, and the maximum piece width is small. As long as the maximum piece width is large, the algorithm performs well. The worst average performance shown by the contour lines is 1.5. This value occurs when the maximum piece size allowed has height 20 and width 4.

40 Height by 10 Width Bin In Figure 6.3 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 40, and the maximum piece size allowed is equal to the full optimal bin size. For this distribution the sample mean is 46.5. The sample standard deviation is 5.681. Figure 6.4 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 40 and width 10, given optimal piece orientation, achieves excellent results for all possible maximum piece size selections. The worst average performance shown by the contour lines is only 1.3. Much like the results for bins of height 20 and width 20, the

largest value occurs when the maximum piece size allowed has large height and small width.

10 Height by 40 Width Bin In Figure 6.5 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 10. and the maximum piece size allowed is equal to the full optimal bin size. For this distribution the sample mean is 11.38. The sample standard deviation is 1.49. Figure 6.6 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 10 and width 40, given optimal piece orientation, achieves reasonable results for most possible maximum piece size selections. The performance of the algorithm is poor when the maximum piece size allowed is approximately equal to the optimal bin height and the maximum piece width is extremely small. The worst average performance shown by the contour lines is 1.7. This result is comparable to the results for the other bin sizes considered.

6.1.2 Random Piece Orientation

In order to give each piece a random orientation we randomly select the value 0 or 1 as the piece information is input. Each of these values have

equal probability. If we receive a zero, and the height of the piece is less that or equal to the bin width, we rotate the piece. Otherwise we leave the piece with its current orientation. Once the complete list of pieces has been read in we "lock" all piece orientations.

The asymptotic average equation for the BU algorithm, when pieces are given random orientation, is found to be

$$BU(L) = 0.958 \cdot OPT(L) + 0.369 \cdot H$$

For this equation the multiple R value is 0.993. The absolute average equation is

$$BU(L) = 1.06 \cdot OPT(L)$$

The multiple R value is 0.987.

20 Height by 20 Width Bin In Figure 6.7 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 20, and the maximum piece size allowed is equal to the full optimal bin size. For this distribution the sample mean is 25.94. The sample standard deviation is 3.534. Figure 6.8 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 20 and width 20, given optimal

piece orientation, achieves good results for all possible maximum piece size selections. The worst results occur when either maximum piece dimension greatly differs from the other. The largest contour value is only 1.3.

40 Height by 10 Width Bin In Figure 6.9 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 40, and the maximum piece size allowed is equal to the full optimal bin size. For this distribution the sample mean is 47.39. The sample standard deviation is 5.652. Figure 6.10 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 40 and width 10, given optimal piece orientation, achieves good results for all maximum piece size settings. It should be noted that the scale used for the contour lines is increased to two decimal places for this plot. The largest contour line value is 1.34. This occurs when the maximum piece height is slightly smaller then the optimal bin height, and the maximum piece width is small.

10 Height by 40 Width Bin In Figure 6.11 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 10, and

the maximum piece size allowed is equal to the full optimal bin size. For this distribution the sample mean is 23.53. The sample standard deviation is 8.343. Figure 6.12 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 10 and width 40, given optimal piece orientation, achieves extremely poor results when the maximum piece width is large. The highest contour value is 4.1. This means that the average pack height for the Bottom-Up algorithm is 4.1 times the optimal pack height. In this case, when the optimal bin height is significantly smaller than the bin width, it is not surprising that the results given, when pieces are given random orientations, decline considerably from the results found when pieces were given optimal orientations. If a piece having a relatively large width is rotated, when it is read into the algorithm, we greatly increase the bin height needed to fit such a piece.

6.2 Piece Rotation Allowed

If pieces are allowed to rotate, the Bottom-Up algorithm, gives the asymptotic average equation

$$BU(L) = 0.939 \cdot OPT(L) + 0.386 \cdot H$$

The multiple R value is 0.993. The absolute average equation, given by the Bottom-Up algorithm is

$$BU(L) = 1.056 \cdot OPT(L)$$

For this equation the multiple R value is 0.984.

20 Height by 20 Width Bin In Figure 6.13 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 20, and the maximum piece size allowed is equal to the full optimal bin size. For this distribution the sample mean is 26.94. The sample standard deviation is 3.643. Figure 6.14 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 20 and width 20, given optimal piece orientation, achieves fair results when both dimensions are less than half of their maximum value. When either dimension is large the results decline. The worst results occur when the maximum piece height greatly differs from the maximum piece width. The largest contour value is 1.5.

40 Height by 10 Width Bin In Figure 6.15 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 40,

and the maximum piece size allowed is equal to the full optimal bin size. For this distribution the sample mean is 46.27. The sample standard deviation is 5.59. Figure 6.16 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 40 and width 10, given optimal piece orientation, achieves excellent results for all maximum piece size settings. Only when the maximum piece height is approximately equal to the optimal bin height, and the maximum piece width is small, is the performance moderately poor. The largest contour line value is only 1.3.

10 Height by 40 Width Bin In Figure 6.17 we show the distribution of the pack heights given by the Bottom-Up algorithm on our first set of test problems. For these problems the optimal pack height is equal to 10, and the maximum piece size allowed is equal to the full optimal bin size. For this distribution the sample mean is 27.01. The sample standard deviation is 6.668. Figure 6.18 demonstrates that the average performance of the Bottom-Up algorithm, for bins of height 10 and width 40, given optimal piece orientation, achieves extremely poor results when the maximum piece width is larger than the maximum piece height. The highest contour value is 3.4. Similar to the case when pieces are given random orientations, it is not surprising that results decline when allowing piece rotation for a

bin whose optimal pack height is smaller then the bin width. If the BU algorithm rotates a piece whose optimal orientation had a large width we greatly increase the height needed in order to pack.

6.3 Summary

By using the graphs described in the previous sections, specifically the contour plots, we can select the most reasonable version of the Bottom-Up algorithm for a given list of pieces. When given a list of pieces and a bin width, we can approximate the optimal bin height by summing over all the piece areas and dividing by the bin width. Depending on the relative size of the approximated bin height to the bin width, we can refer to the contour plots for the bin size most representative of the bin dimensions. At this point, we can select the version of the Bottom-Up algorithm, which gives the smallest contour plot value for the maximum piece size selection for the given list of pieces. For example, if we roughly estimated our bin height as approximately equal to our bin width, we can refer to our contour plots for bins of height 20 and width 20. Given that the pieces that we wish to pack have heights no larger than 1/4 of the approximated bin height, and widths which can be as large as the bin width, we can check the appropriate point on our contour plots. For our 20 height by 20 width bins, with a maximum

piece height of 5 and maximum piece width of 20, we can see from our contour plots (Figures 6.2, 6.8, and 6.14) that the no rotation version of the BU algorithm may give the best results.

7 Statistical Tests

In the preceding sections we have used graphical methods to analyze an algorithm's average performance. With these methods we increase our knowledge about an algorithm's strengths and limitations. However, we are still left with the question, "Is one algorithm better than another?". We attempt to answer this question by applying two well known statistical tests. In order to illustrate these statistical methods we shall compare our different variations of the Bottom-Up algorithm.

7.1 Testing Means

It would be useful to have a method of testing if an algorithm's mean packing height, μ, is a specific value, μ_0. To test such a hypothesis we can make use of the central limit theorem, if we have a large sample size. For our population of pack heights, given by an algorithm on a set of test problems, the variance of the population is unknown.

To test the null hypothesis $H_0 : \mu = \mu_0$ versus the alternative hypothesis $H_1 : \mu \neq \mu_0$ we use the following test statistic,

$$t = \frac{\bar{y} - \mu_0}{s/\sqrt{n}}.$$

For this equation, the sample mean, the sample standard deviation ,and the

sample size are denoted by $\bar{y}$, s, and n respectively.

We reject the null hypothesis if $|t| > t_{\frac{\alpha}{2},n-1}$, where $t_{\frac{\alpha}{2},n-1}$ denotes the upper $\alpha/2$ percentage point of the t distribution with $n-1$ degrees of freedom.

Example 7.1 : We wish to test if the mean pack height of the rotation version of the Bottom-Up algorithm is equal to the optimal pack height plus 5, for bins of height 20 and width 20 and maximum piece size equal to the full bin. To test this we make use of the results shown in Figure 6.13, from our first test problem set. Thus, for this example $\bar{y} = 26.94$, $s = 3.643$, and $n = 10,000$.

The hypotheses to be tested are:

$$H_0 : \mu = 25$$

$$H_1 : \mu \neq 25$$

The value of the test statistic is $t = 53.25$. If $\alpha = 0.05$, we find $t_{0.025,9999} \leq 2$. Therefore, we reject H_0 and conclude that the mean pack height differs from 25, at $\alpha = 0.05$

7.2 Sign Test

In order to answer the question, "Is one algorithm better than another?", we use the sign test to compare two algorithms according to their performance on a set of test problems. It should be noted that for these test problems it is not necessary to know the optimal pack height. The data consists of observations on a random sample, $(X_i, Y_i), i = 1, ..., n'$. For our purposes the pack height given by Algorithm 1 is denoted by X_i and the pack height given by Algorithm 2 is denoted by Y_i. The number of test problems in our set is n'. We assume that the bivariate random variables $(X_i, Y_i), i = 1, ..., n'$, are mutually independent.

For each pair, (X_i, Y_i), a comparison is made. The pair is classified as "+" if $X_i < Y_i$, "−" if $X_i > Y_i$, and "0" if $X_i = Y_i$. We wish to test if the expected pack height of Algorithm 1 is greater than or equal to the expected pack height of Algorithm 2. Thus our null hypothesis is $H_0 : P(+) \leq P(-)$, and our alternative hypothesis is $H_1 : P(+) > P(-)$. P(+) is the probability that $X_i < Y_i$. The test statistic, T, for our test is equal to the total number of "+" pairs.

For our decision rule we first disregard all the pairs which are tied. Thus, we let n equal the total number of "+" pairs plus the total number of "−"

pairs. For $n > 20$ we find the value $t = \frac{1}{2}(n + z_\alpha \cdot \sqrt{n})$, where z_p is the quantile of the standard normal random variable Z such that $P(Z \leq z_p) = p$. We reject H_0, at a level of significance α, if $T \geq n - t$

Example 7.2 : We wish to test if the Bottom-Up algorithm performs better when we allow rotation compared to fixed random orientations. We will only test for bins of height 20 and width 20, with the maximum piece size allowable equal the full bin. We will use the results from our first test problem set. We let the rotation algorithm be represented as Algorithm 2. For this example $n_\prime = 10,000$ and $\alpha = 0.05$.

The hypotheses to be tested are:

H_0: The pack height when allowing rotation is less than or equal to the pack height when pieces are fixed with random orientations for the Bottom-Up algorithm. H_1: The pack height when allowing rotation is greater than the pack height when pieces are fixed with random orientations for the Bottom-Up algorithm.

For our data it was found that, $n = 7925$, $T = 4838$, $z_{.05} = -1.645$ and $t = 3889.279$. Therefore, since $T > (n - t)$, where $n - t = 4035.72$, we reject H_0, at $\alpha = .05$. We state that for the Bottom-Up algorithm, the pack height for rotation is greater than the pack height for random orientations, for bins of height 20 and width 20. Thus if we wish to pack pieces into bins which

are assumed to be of height 20 and width 20, where the maximum possible piece size is equal to the full bin, we should receive better results using the no rotation version of the BU algorithm as opposed to the rotation version.

Take note that the Wilcoxon paired signed rank test is likely to be more powerful than the sign test when we have symmetry of the population distribution for the paired differences. If the sampled paired differences come from a population with a normal distribution, then the two-sample t test is the most powerful test. The sign test was selected since we desired a test that could be used to compare any two algorithms, regardless of the distribution of the sample paired differences.

8 Conclusions

In the previous sections we have introduced and analyzed both the 2D-BPGen algorithm for creating two dimensional bin packing test problems and the Bottom-Up algorithm for packing two dimensional bin packing problems. It is the feeling of this author that these algorithms have been shown to be both practical and useful. It is felt that the Bottom-Up algorithm is at least competitive with the other two dimensional bin packing algorithms available in current literature. Within this thesis we have given a method of statistically testing the hypothesis that the Bottom-Up algorithm may in fact perform better than other algorithms, for given test problems. We leave this and the following topics for future research.

First, consideration is needed in testing if there exists a better slack setting for the Bottom-Up algorithm. It may be, that for certain parameters, a non-zero slack value may enhance the algorithm's performance. The second topic involves modifying the order in which pieces are packed by the BU algorithm. In our drywall application it was necessary to use area as a deciding factor when selecting pieces. If we instead selected pieces in the same order as the FFDH algorithm, could we then prove that $BU(L) \leq FFDH(L)$ for all L? Would this change improve the algorithm's performance?

A One Dimensional Bin Packing

The standard one dimensional bin packing problem can be stated as follows. We are given a list $L = \{p_1, p_2, \ldots, p_n\}$ and an infinite number of bins each with capacity C where each piece, p_i, has size $h_i \leq C$. We wish to pack these pieces into the bins so that the capacity of no bin is exceeded and the minimal number of bins is used. Also no two pieces are allowed to overlap.

The zero-one integer program for the bin packing problem, as given in [18], is

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} y_j \\ \text{subject to}: \quad & \sum_{i=1}^{n} h_i x_{ij} \leq C y_j \qquad j = 1, \ldots, n \\ & \sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n \\ & y_j, x_{ij} \in \{0, 1\} \end{aligned}$$

where $x_{ij} = 1$ if piece i is placed in bin j, 0 otherwise and $y_j = 1$ if bin j is used, 0 otherwise. Note that in this classical formulation of the problem, the objective is to minimize the number of bins. This is equivalent to minimizing the overall waste since, for n bins, the waste is $nC - \sum_{i=1}^{n} h_i$, a constant. This result generalizes to higher dimensions.

The problem of one dimensional bin packing is known to be NP-complete [12]. Thus various heuristic algorithms have been developed. We give a brief outline of one such heuristic algorithm. More information and results on one dimensional bin packing can be found in [7], [13], [14], [15], [16], [17], [18], and [20].

A.1 First Fit Decreasing (FFD)

The First Fit Decreasing algorithm is discussed in [17] and described in the following manner. Let the bins be indexed as $B_1, B_2, ...$, with each initially filled to a level zero and with a maximum capacity of one. Let the values $h_1, h_2, ..., h_n$ be arranged into non-increasing order so that $h_i \geq h_j \ \forall i, j$. To place h_i, find the least j such that B_j is filled to a level $\beta \leq 1 - h_i$, and place h_i in B_j. B_j is now filled to a level $\beta + h_i$. For more information on FFD refer to [17]. For our drywall problem the h_i represented the height of the four foot wide pieces to be packed onto the drywall sheet. An example of how FFD packs a given list is given in Figure A.1.

In [17] it is proved that for any given list L,

$$FFD(L) \leq \frac{11}{9} \cdot OPT(L) + 4.$$

Here FFD(L) represents the number of bins used for FFD to pack the list

L. OPT(L) is the minimum number of bins required to pack the list L. It is also proved in [17] that for any given list L,

$$FFD(L) \geq \frac{11}{9} \cdot OPT(L) - 2.$$

B NP-Completeness

A problem is defined in [13] as an abstract description coupled with a question requiring an answer. There are two types of problems that are of specific interest to theoreticians. These are decision problems and search problems. Decision problems are those in which the answer to the problem is either "yes" or "no". Search problems involve finding a structure, out of a (large) set of possible structures, that has specified properties. The theory of NP-completeness is restricted to decision problems and thus unless otherwise stated we will concern ourselves with problems of this nature.

B.1 Algorithms, Time Complexity and Computation

An algorithm is a set of finite steps which solves a given problem. Thus if the true answer to the posed question is yes, then the algorithm, obviously, should return a yes. Often there are many algorithms which solve the same problem. It is therefore necessary to be able to compare algorithms. The amount of time that an algorithm takes to solve a problem is a standard measure of how "good" the algorithm is. This amount of time is measured by the time complexity of the algorithm. The time complexity of an algorithm measures the time taken for the algorithm to solve the problem in

terms of the length of input. This time to solve can also be measured by computational complexity which is the number of operations that solve a problem.

It is important for us to define two types of computational algorithms; deterministic and non-deterministic. A program is considered to be deterministic of it has a finite set of states, including an initial state, and two halting states. Also, the program has a transition function which alters the current states of the computation. A deterministic program starts at the initial state and applies the transition function repeatedly in a step-by-step manner until either of the two halting states are reached. This is the way in which most people think of computer programs.

Non-deterministic computation, on the other hand, is quite different than deterministic computation. A non-deterministic program consists of two stages, a guessing stage and a checking stage. The guessing stage guesses a structure which may or may not be a solution to the problem. This is determined in the checking stage in a normal deterministic manner.

B.2 The Classes P and NP

The theory of computational complexity involves classifying problems according to how difficult, "easy" or "hard", they are to solve. This classi-

fication scheme includes the classes P and NP. The class P is defined in [13] as the set of recognition (decision) problems for which there exists a polynomial-time algorithm. This class includes the problems that are formally considered "easy". The class NP is the set of all problems which can be solved by a non-deterministic polynomial time algorithm. Take note that the class NP contains the class P since the checking stage of a non-deterministic algorithm may be replaced by the polynomial time algorithm that solves the problem.

The class NP consists of all recognition problems with the following property: for any "yes" instance of the problem there exists a proof of this fact that can be verified in polynomial time. [13]

B.3 NP-Complete Problems

A problem is NP-Complete if it is in NP, and every problem in NP is translatable to it in polynomial time. Thus, if there is a polynomial-time algorithm for any one of these NP-Complete problems, then there is a polynomial-time algorithm for every problem in NP and then P=NP. At present no one has been able to find an algorithm to solve any of these problems in polynomial time. However, no one has been able to prove that a polynomial-time algorithm does not exist. Although this question of whether the classes P and

NP are the same is one that at this moment does not have a known answer, there is widespread belief that $P \neq NP$.

The class NP and the notion of "complete" problems for NP were first introduced by Cook in 1971. In his paper, he showed that the problem known as SATISFIABILITY was NP-Complete. He did this by showing that every other problem in NP could be encoded as an appropriate special case of SATISFIABILITY. For a proof of this and more detailed information on NP-Complete we refer the reader to [12]. Once the first NP-Complete problem had been established it became easy to show that other problems were also NP-Complete. To do so requires providing a polynomial transformation from a known NP-Complete problem to the candidate problem. One needs to show that the known problem, such as SATISFIABILITY, is a special case of the new problem. In [12] it is shown that one-dimensional bin packing is NP-complete. Since one-dimension bin packing is a special case of two-dimensional bin packing, then two-dimensional bin packing is also NP-complete.

B.4 NP-hard

The term NP-hard refers to any problem that is at least as hard as any problem in NP. Thus, the NP-Complete problems are precisely the inter-

section of the class of NP-hard problems with the class NP. In particular, optimization problems whose recognition versions are NP-Complete, such as bin-packing, are NP-hard, since solving the optimization version is at least as hard as solving the recognition version [13].

B.5 Heuristics

A heuristic algorithm is a program which finds near-optimal, "good", solutions with reasonable computational time without guaranteeing optimality. Since we know that bin packing is NP-complete we are justified in stating that it is not practical to search for an algorithm to find the optimal solution. In fact, even if we felt that we had the optimal packing solution, for a given set of rectangular objects, we would still need to check every possible combination. This would be extremely time consuming. Of course this is unless the solution we found had no empty, "wasted", space. Therefore we focus on searching for an algorithm which performs "reasonably well" and "fast".

References

[1] Brenda S. Baker. A new proof for the first-fit decreasing bin-packing algorithm. *Journal of Algorithms*, 6:49–70, 1985.

[2] Brenda S. Baker, Donna J. Brown, and Howard P. Katseff. A $\frac{5}{4}$ algorithm for two-dimensional packing. *Journal of Algorithms*, 2:348–368, 1981.

[3] Brenda S. Baker, E.G. Coffman, and Ronald L. Rivest. Orthogonal packing in two dimensions. *SIAM Journal on Computing*, 9:846–855, 1980.

[4] B.-E. Bengtsson. Packing rectangular pieces - a heuristic approach. *The Computer Journal*, 25:353–357, 1982.

[5] Donna J. Brown. An improved bl lower bound. *Information Processing Letters*, 11:37–39, 1980.

[6] E.G. Coffman, Jr., M.R. Garey, D.S. Johnson, and R.E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9:808–826, 1980.

[7] E.G. Coffman, Jr. and George S. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithms.* John Wiley and Sons, 1991.

[8] W.J. Conover. *Practical Nonparametric Statistics, Second Edition.* John Wiley and Sons, 1980.

[9] Harald Dyckhoff and Ute Finke. *Cutting and Packing in Production and Distribution.* Physica-Verlag, 1992.

[10] Robert J. Fowler, Michael S. Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are np-complete. *Information Processing Letters*, 12:133–137, 1981.

[11] Greg N. Frederickson. Probabilistic analysis for simple one- and two-dimensional bin packing algorithms. *Information Processing Letters*, 11:156–161, 1980.

[12] M.R. Garey and D.S. Johnson. *A Guide to the Theory of NP-Completeness.* W.H.Freeman and Company, 1979.

[13] Saul I. Gass and Carl M. Harris. *Encyclopedia of Operations Research and Management Science.* Kluwer Acamdemic Publishers, 1996.

[14] Micha Hofri. *Probabilistic Analysis of Algorithms.* Springer-Verlag, 1987.

[15] Micha Hofri. *Analysis of Algorithms.* Oxford University Press, 1995.

[16] Micha Hofri and Sami Kamhi. A stochastic analysis of the nfd bin-packing algorithm. *Journal of Algorithms*, 7:489–509, 1986.

[17] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. Performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3:299–325, 1974.

[18] Silvano Martello and Paulo Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley and Sons, 1990.

[19] Douglas C. Montgomery. *Design and Analysis of Experiments, Third Edition.* John Wiley and Sons, 1991.

[20] David Simchi-Levi. New worst case results for the bin-packing problem. *Naval Research Logisitics*, 41:579–585, 1994.

[21] Daniel D.K.D.B. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Processing Letters*, 10:37–40, 1980.

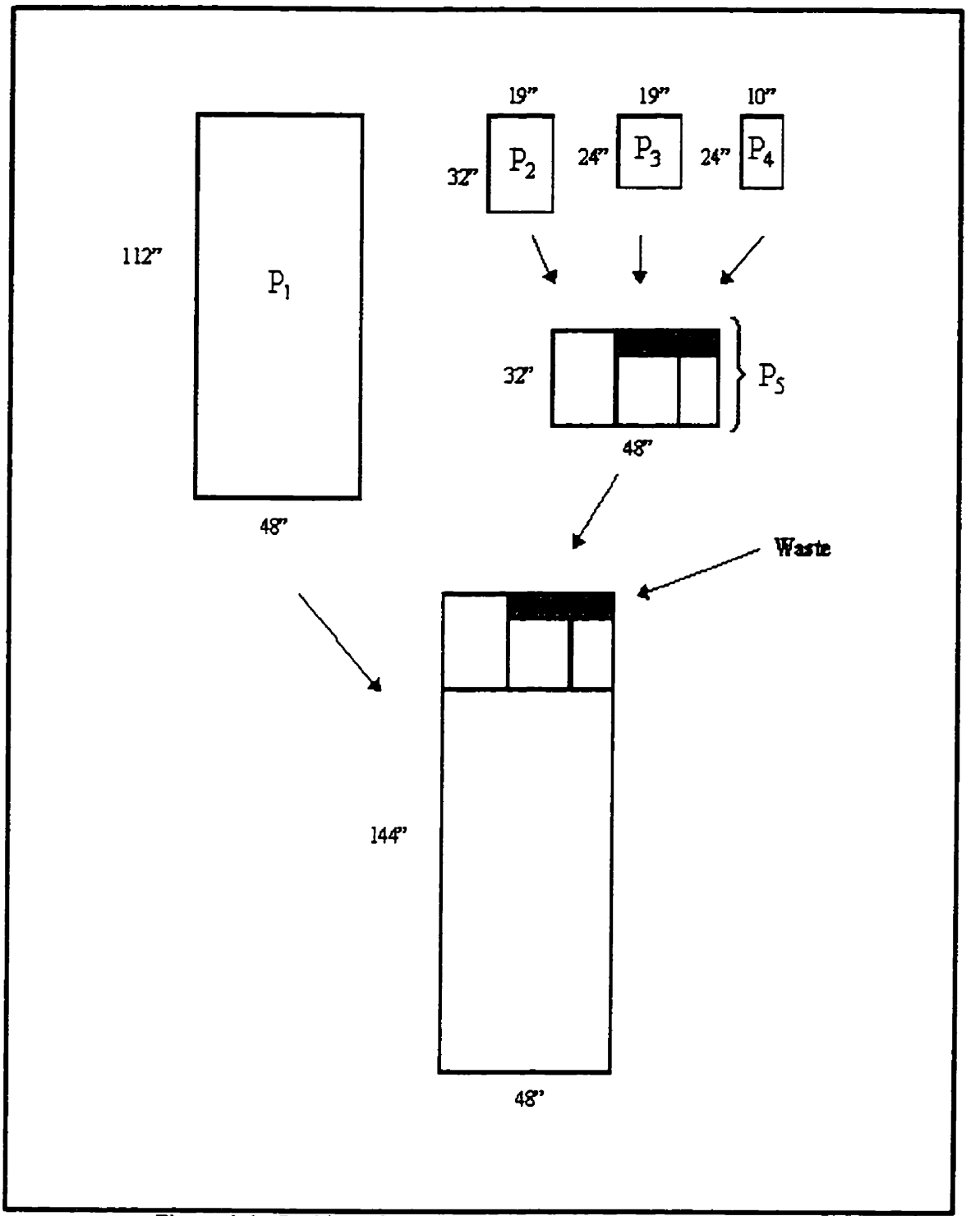


Figure 1.1: Packing of $L=\{p_1, p_2, p_3, p_4\}$ onto a sheet of drywall.

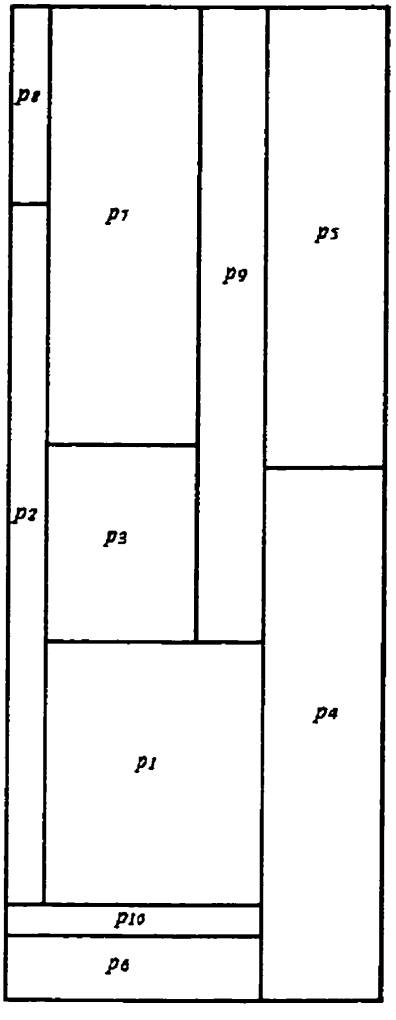


Figure 2.1: An optimal packing of Example 2.1 into a bin of height 30 and width 10.

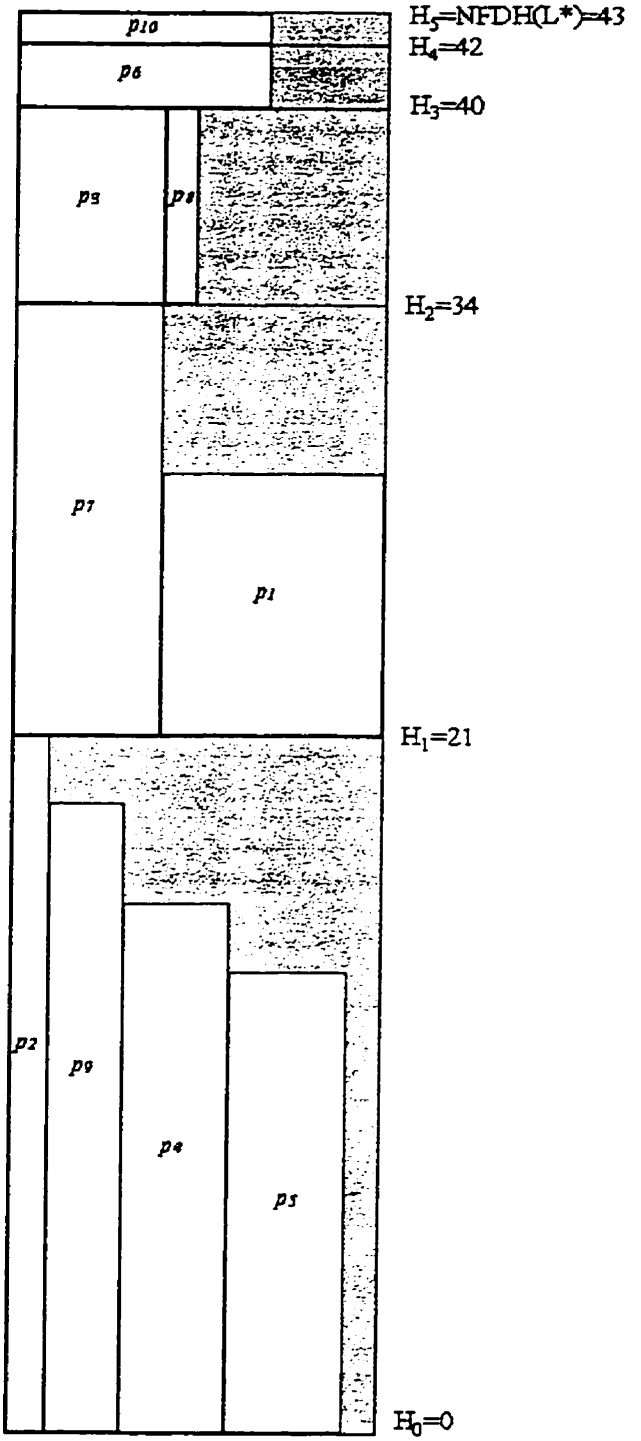


Figure 2.2: The packing of Example 2.1, into a bin of width 10, given by the NFDH algorithm.

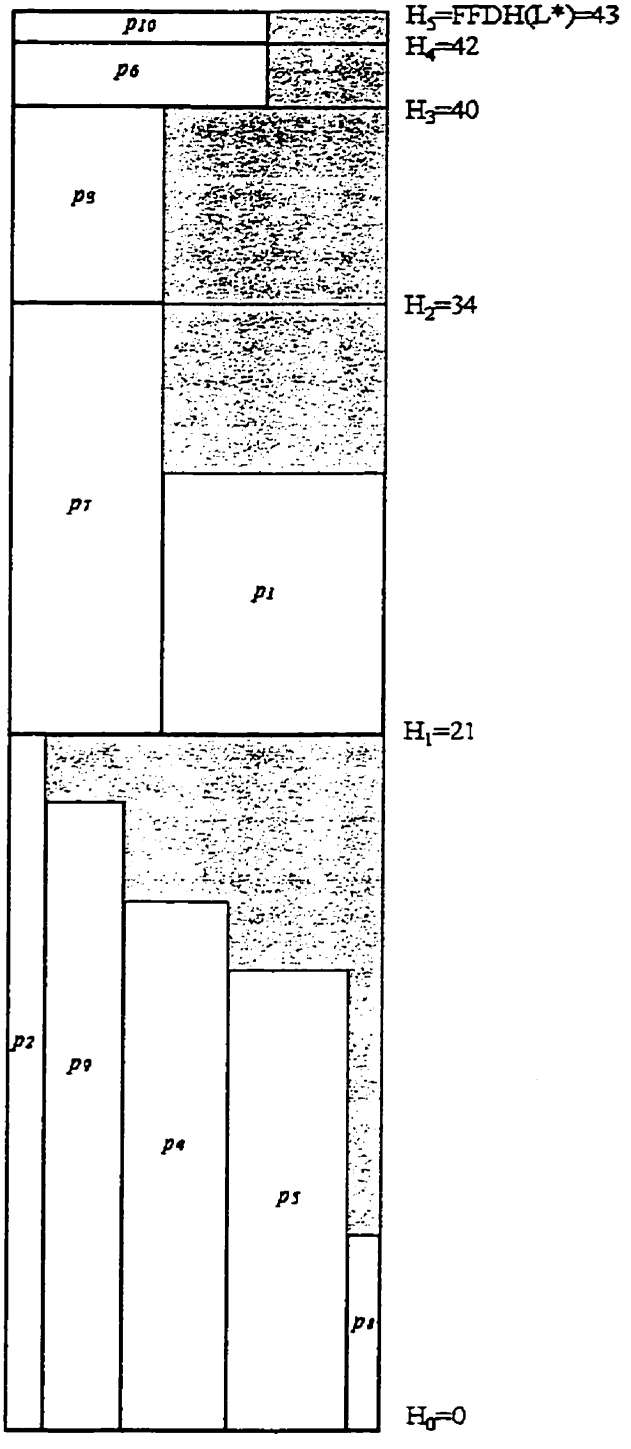


Figure 2.3: The packing of Example 2.1, into a bin of width 10, given by the FFDH algorithm.

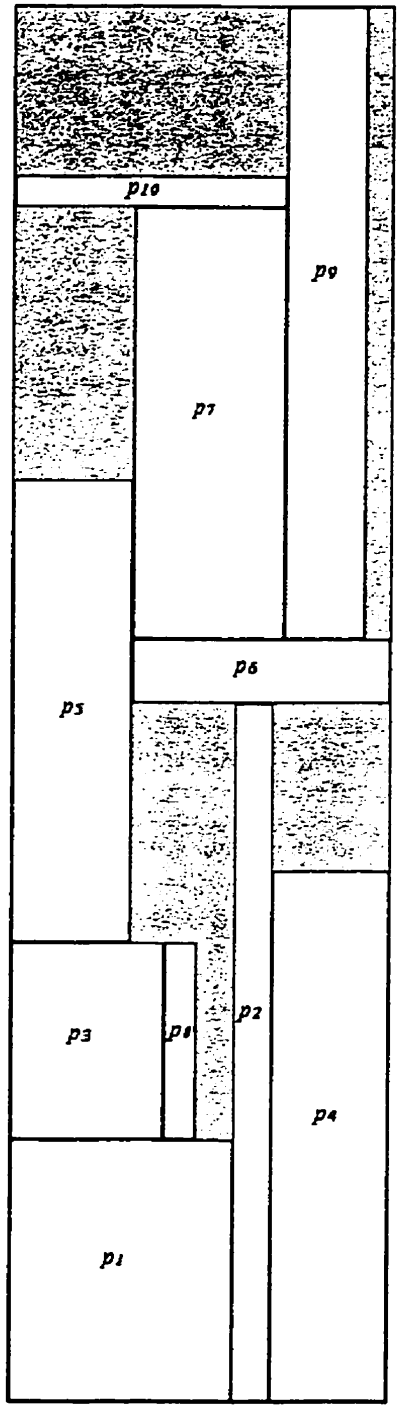


Figure 2.4: The packing of Example 2.1, into a bin of width 10, given by the BL algorithm. BL(L*)=41.

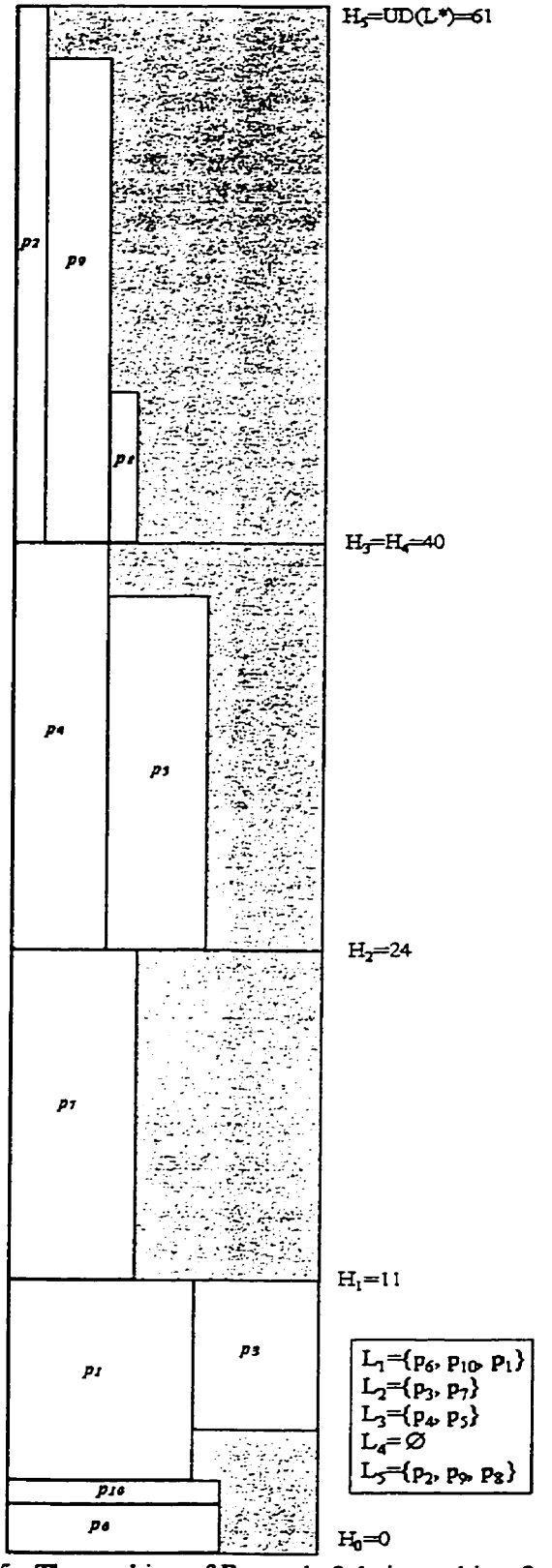
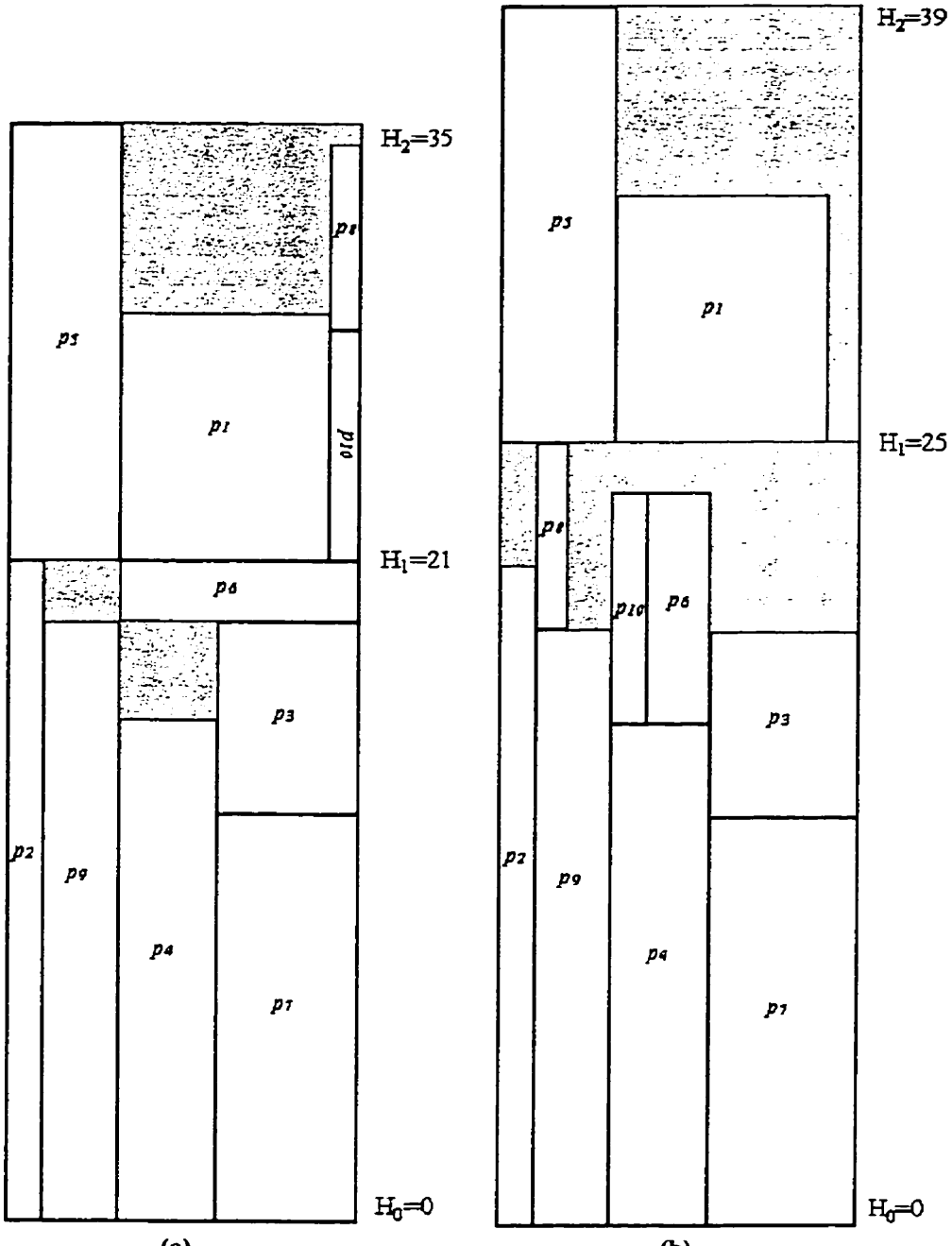


Figure 2.5: The packing of Example 2.1, into a bin of width 10, given by the UD algorithm.



(a) (b)

Figure 3.1: In (a) we show the pack given by BU, of Example 2.1, when slack is set to zero. In (b) we show the pack given by BU when slack is set to five. For both packs we allow rotation.

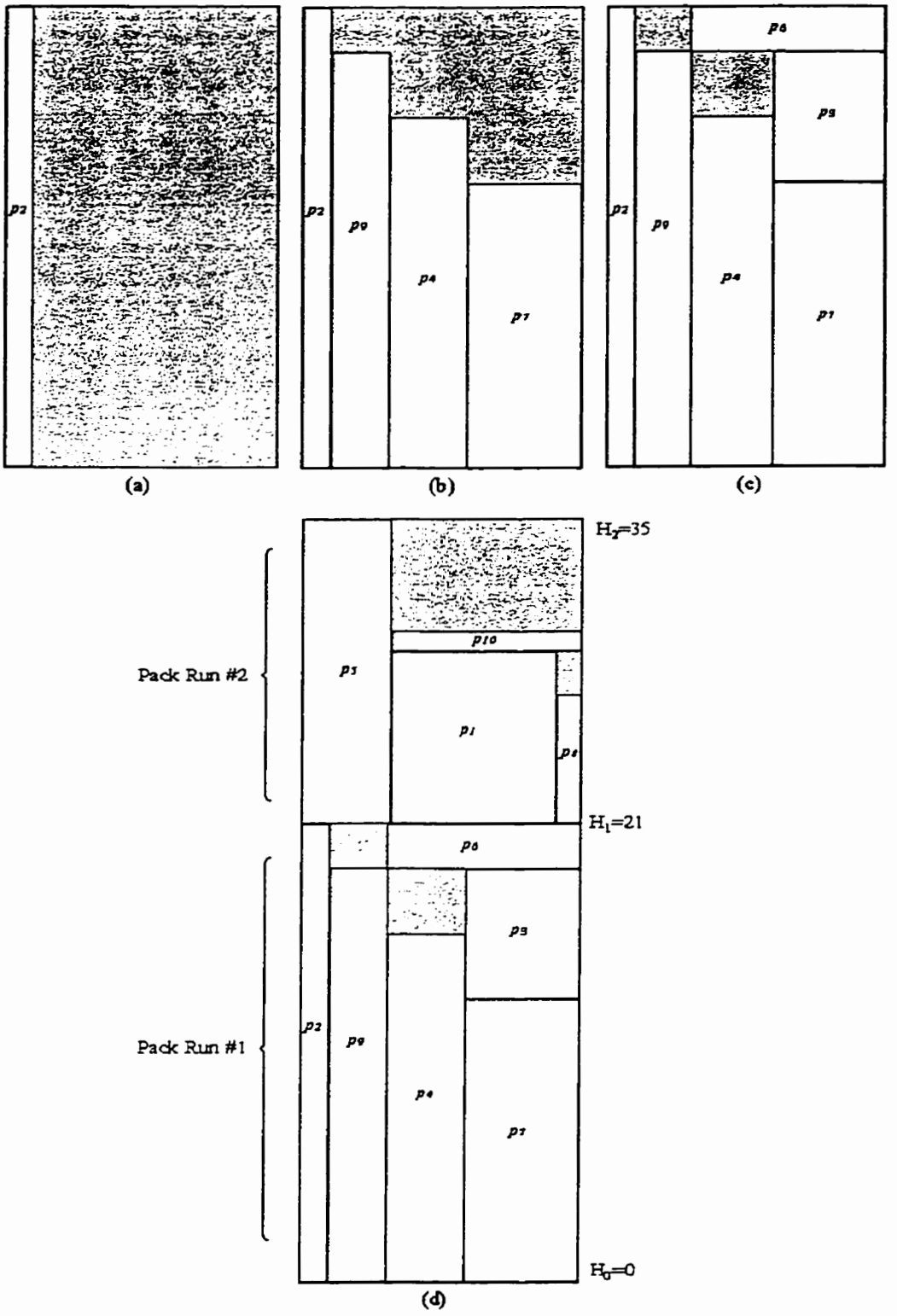


Figure 3.2: In (a) we select our first piece for our first pack run. In (b) we show the bottom row pieces packed in decreasing height. We show the completed first pack run in (c) and then the final pack in (d). Our list is given by Example 2.1. We pack in a bin of width 10 with fixed piece orientation.

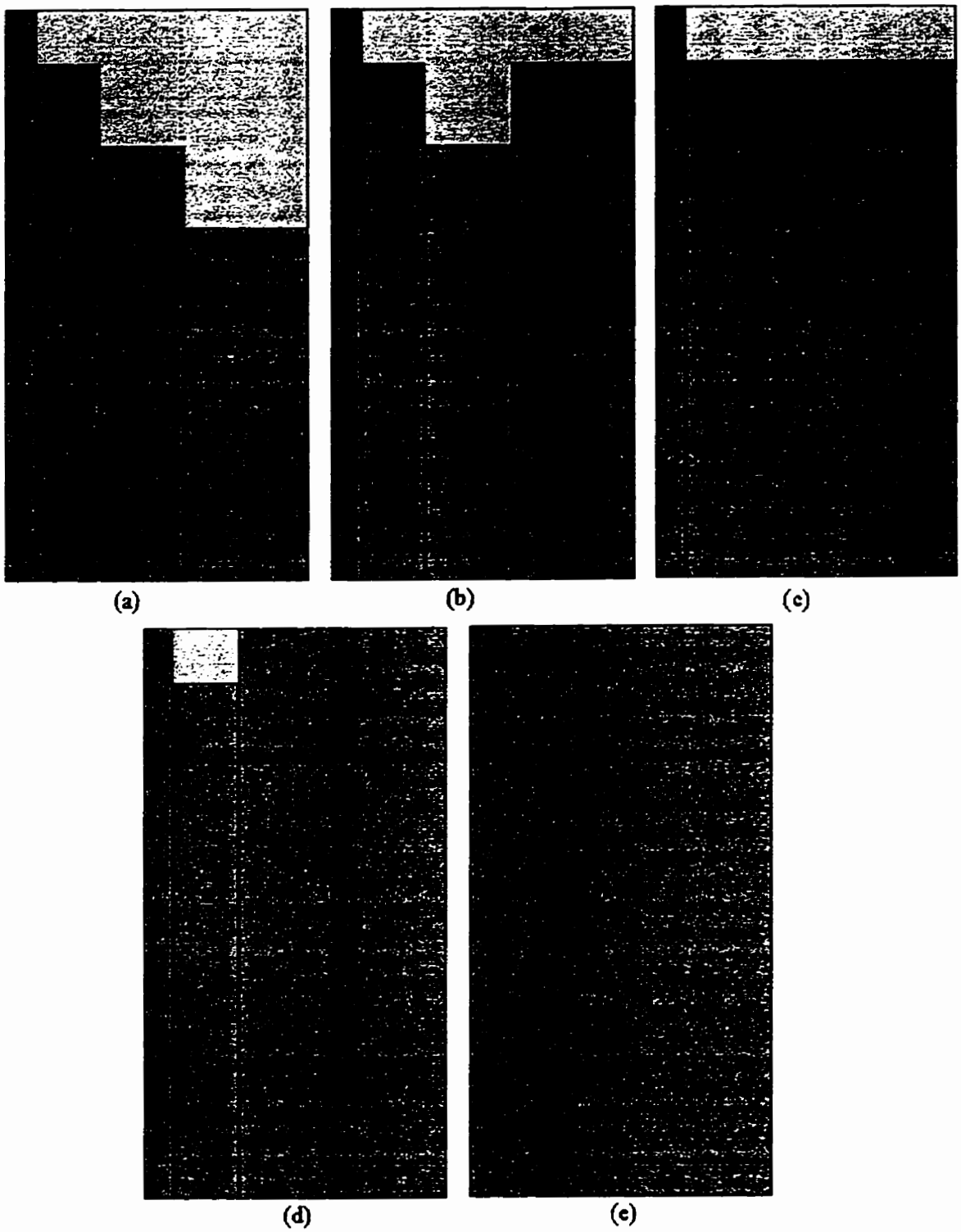


Figure 3.3: The step by step changes in the column definitions for the first pack run from Fig. 3.2.

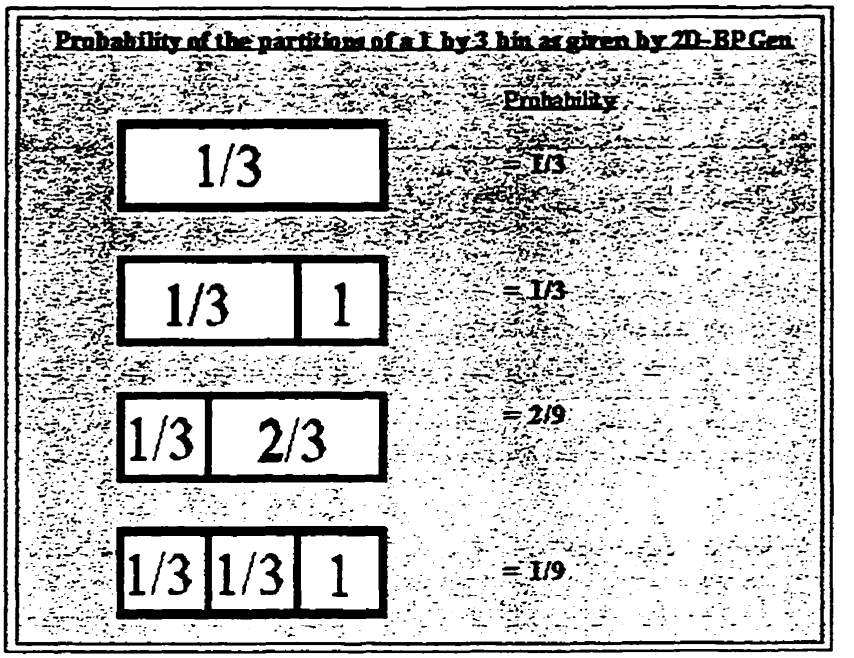

Probability of the partitions of a 1 by 3 bin as given by 2D-BPGen
Probability
1/3
= 1/3
1/3
1
= 1/3
1/3
2/3
= 2/9
1/3
1/3
1
= 1/9

Figure 4.1

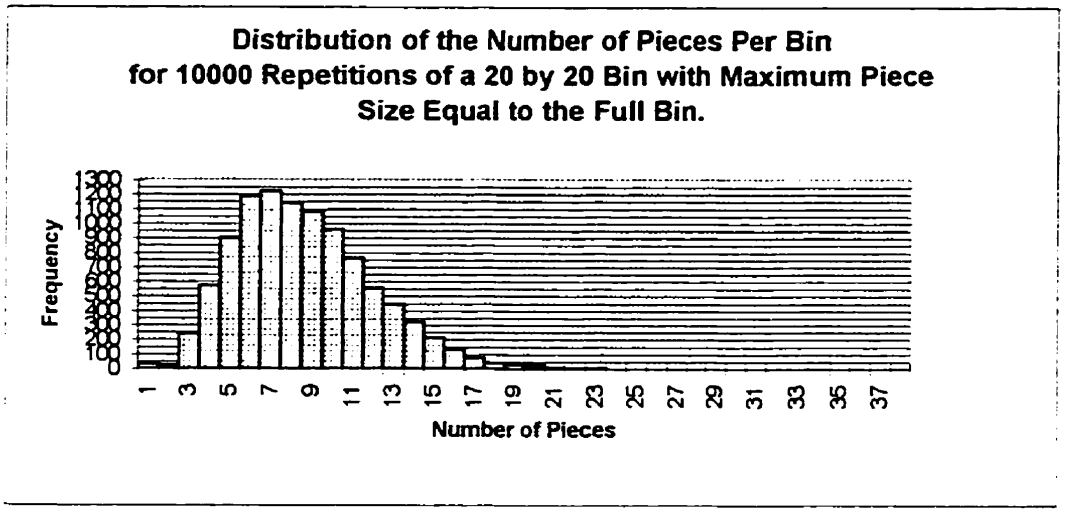


Figure 5.1

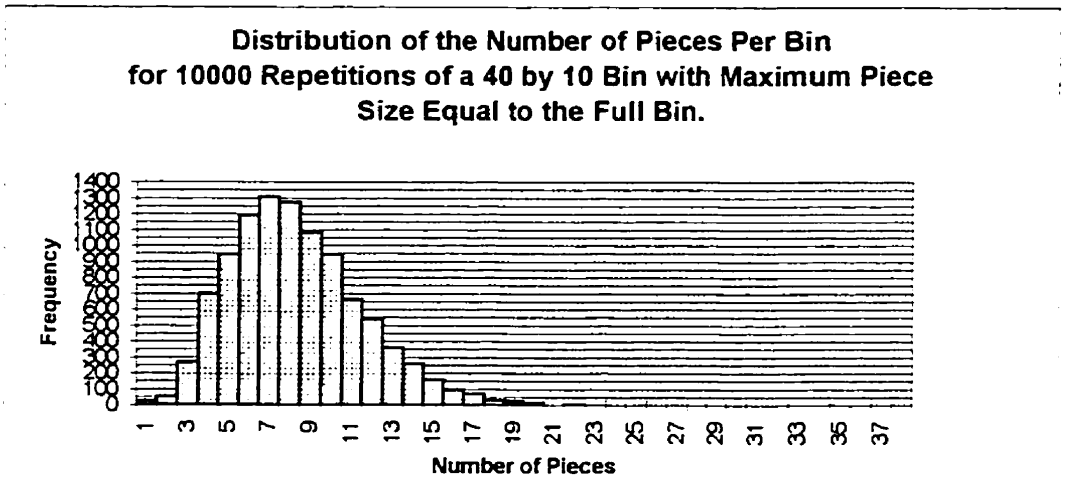


Figure 5.2

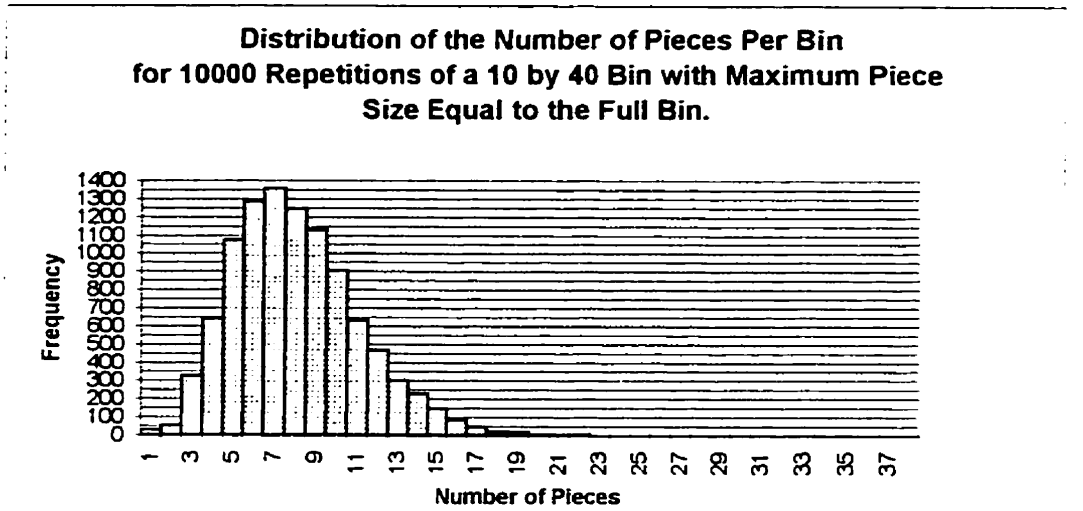


Figure 5.3

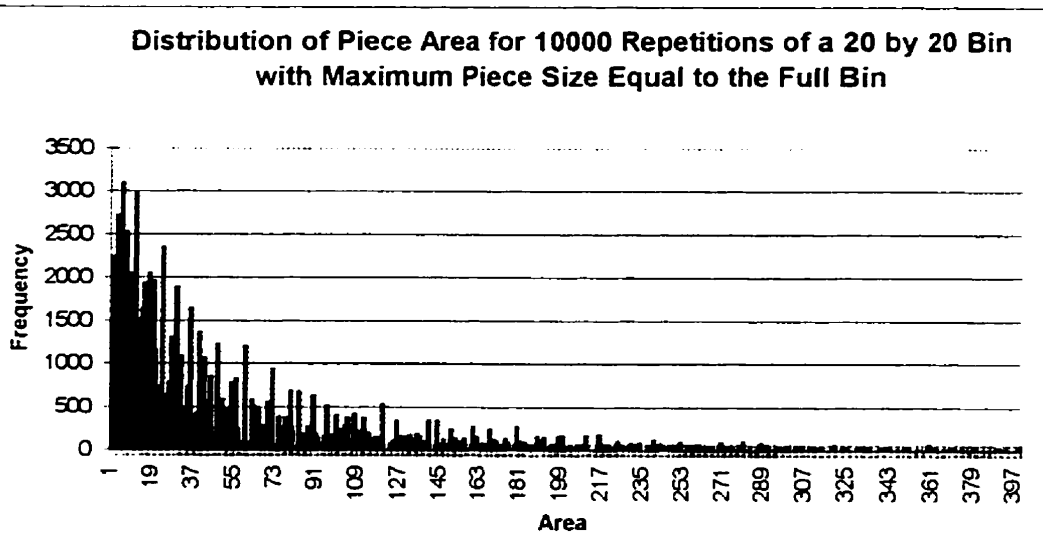


Figure 5.4

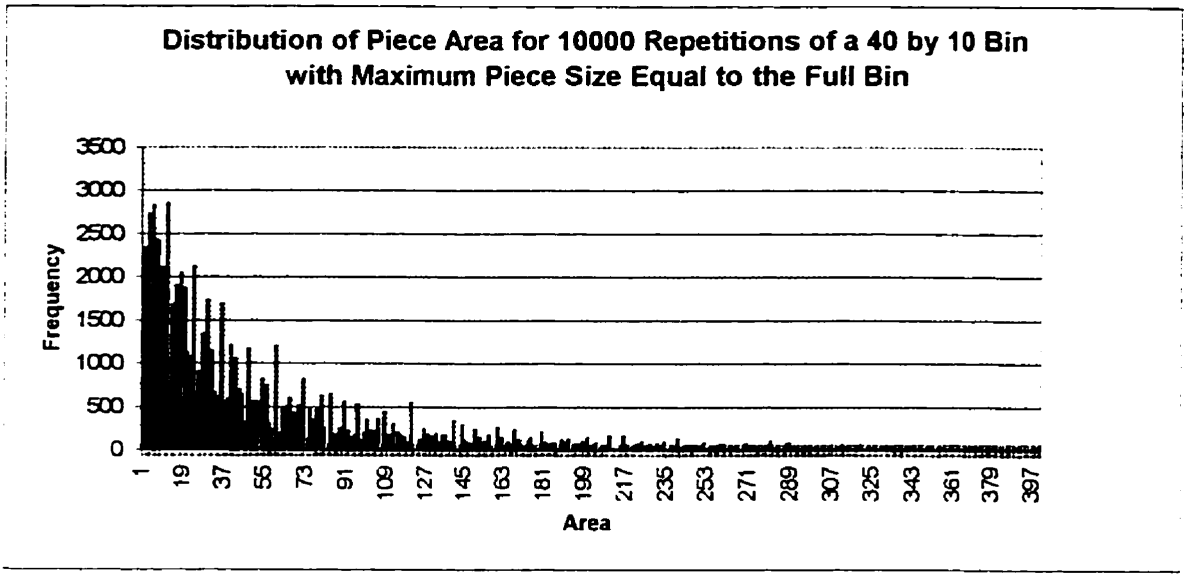


Figure 5.5

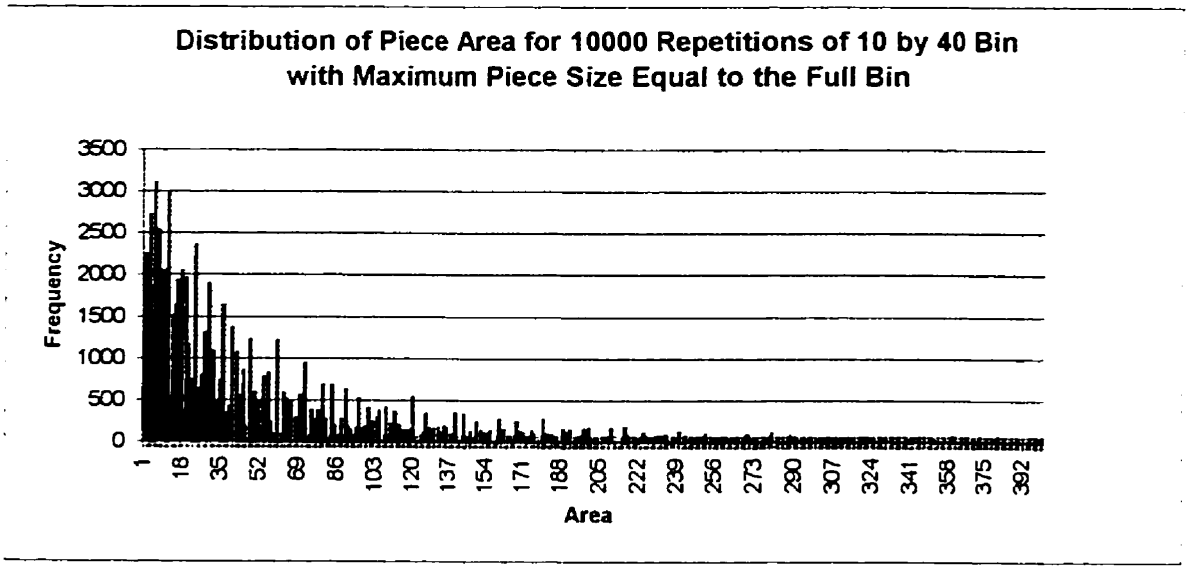


Figure 5.6

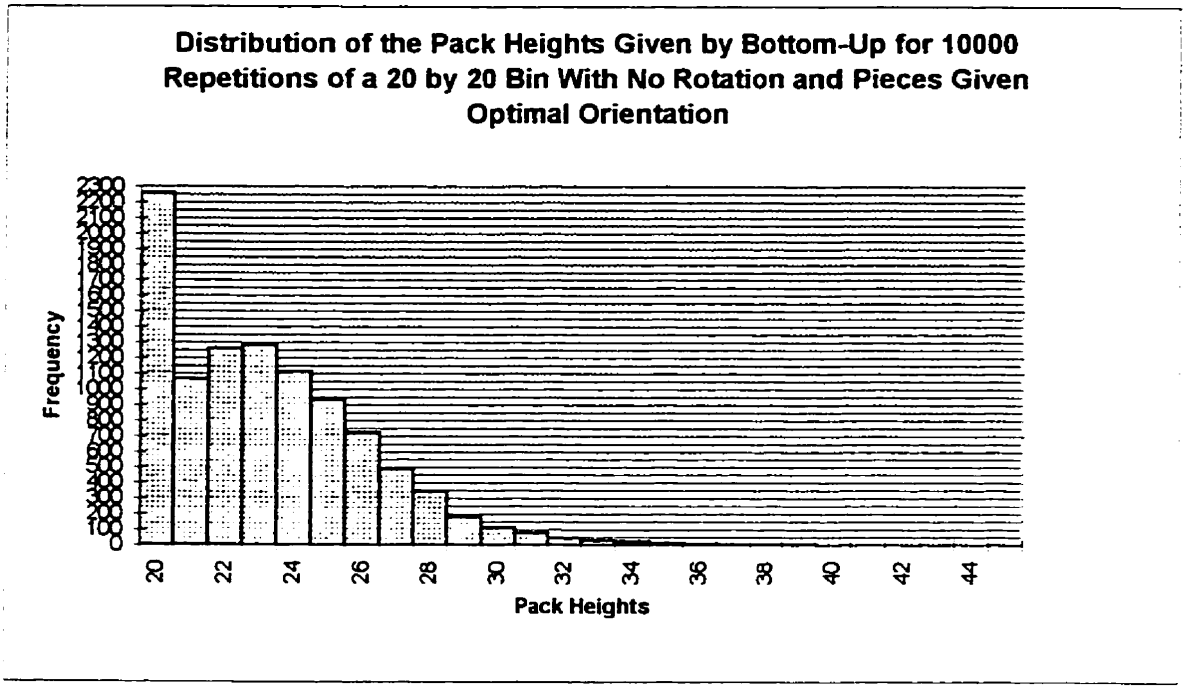

Distribution of the Pack Heights Given by Bottom-Up for 10000 Repetitions of a 20 by 20 Bin With No Rotation and Pieces Given Optimal Orientation
Frequency
Pack Heights
20
22
24
26
28
30
32
34
36
38
40
42
44

Figure 6.1

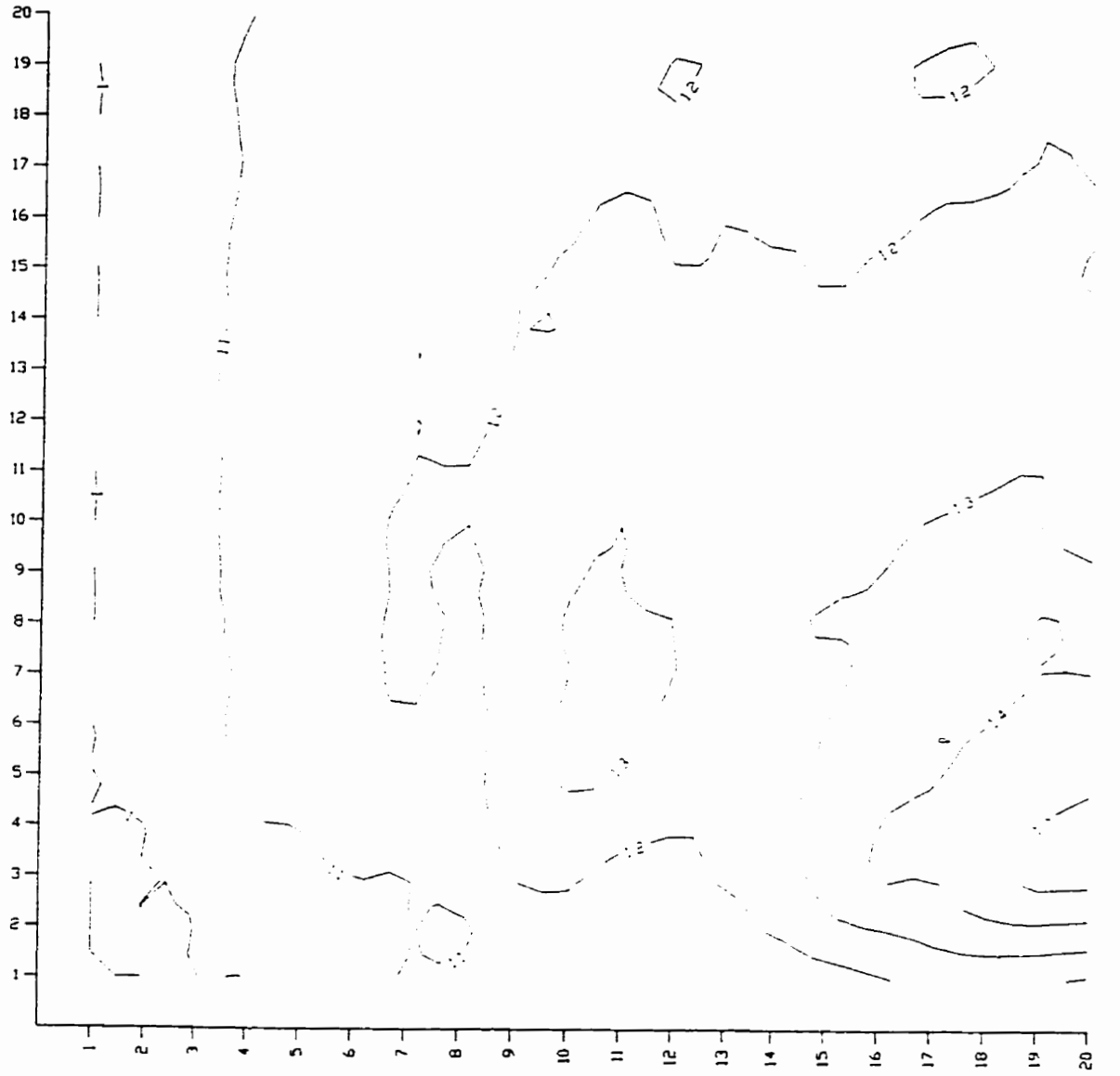


Figure 6.2: The average performance given by the Bottom-Up algorithm for 20 by 20 bins, with no rotation and optimal piece orientation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

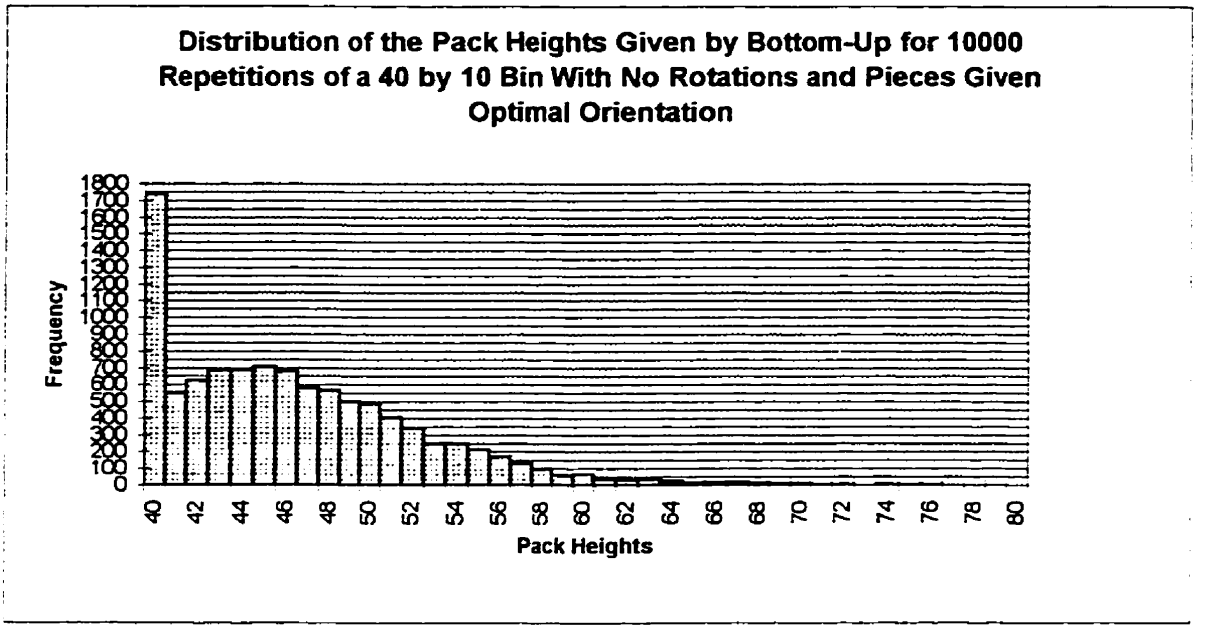

Distribution of the Pack Heights Given by Bottom-Up for 10000 Repetitions of a 40 by 10 Bin With No Rotations and Pieces Given Optimal Orientation
Frequency
Pack Heights
40
42
44
46
48
50
52
54
56
58
60
62
64
66
68
70
72
74
76
78
80

Figure 6.3

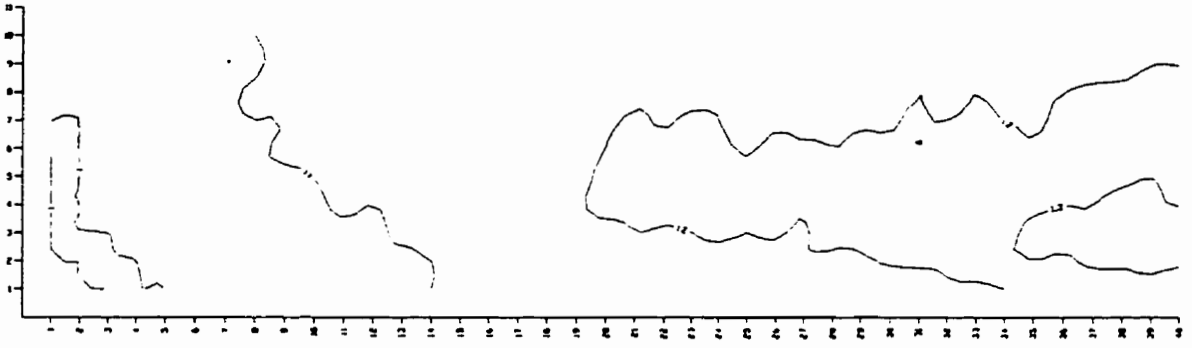


Figure 6.4: The average performance given by the Bottom-Up algorithm for 40 by 10 bins, with no rotation and optimal piece orientation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

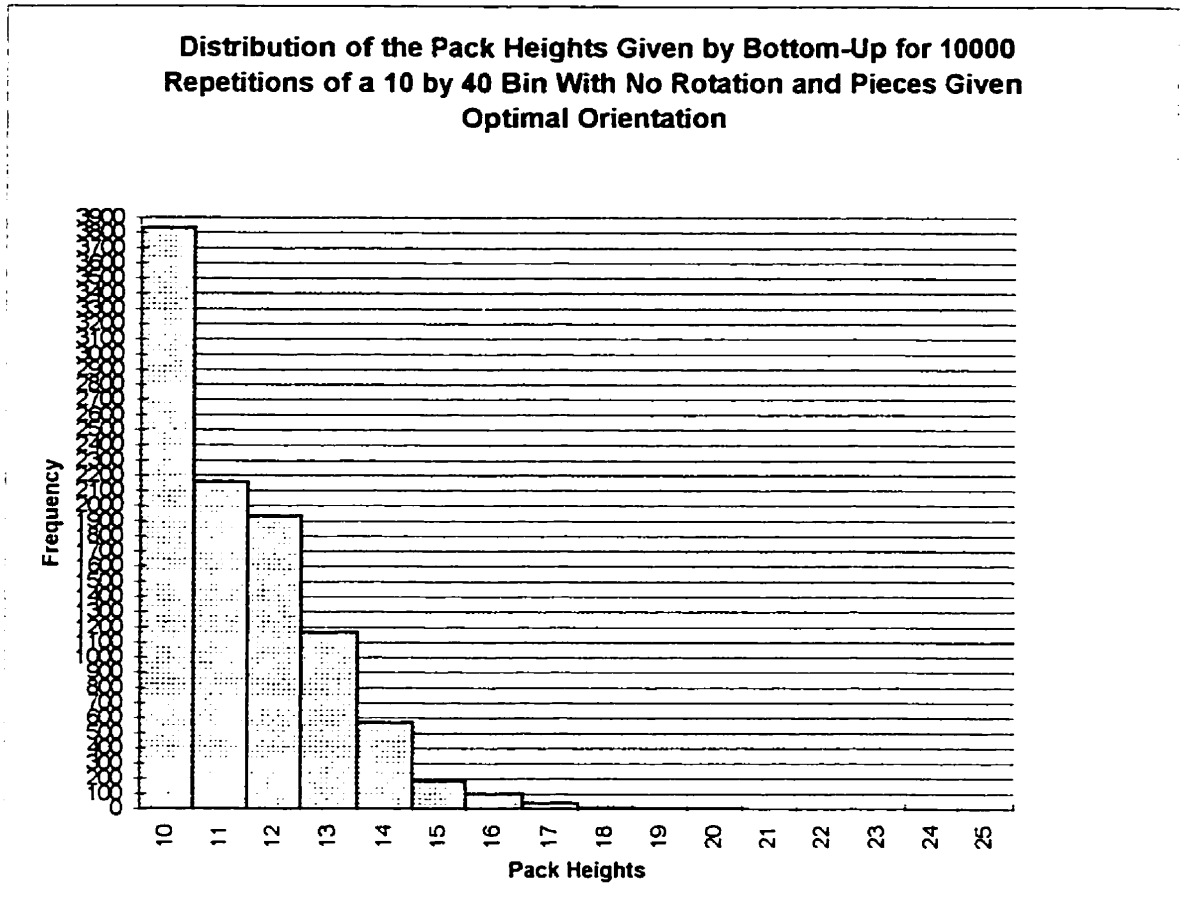

Distribution of the Pack Heights Given by Bottom-Up for 10000 Repetitions of a 10 by 40 Bin With No Rotation and Pieces Given Optimal Orientation
Frequency
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
Pack Heights

Figure 6.5

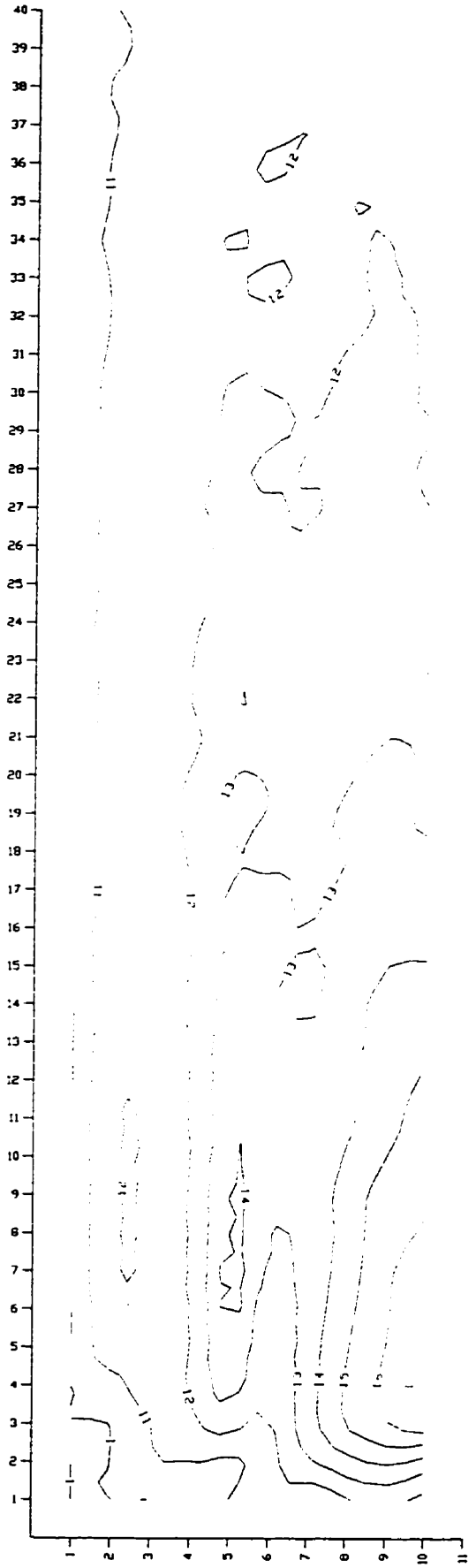


Figure 6.6: The average performance given by the Bottom-Up algorithm for 10 by 40 bins, with no rotation and optimal piece orientation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

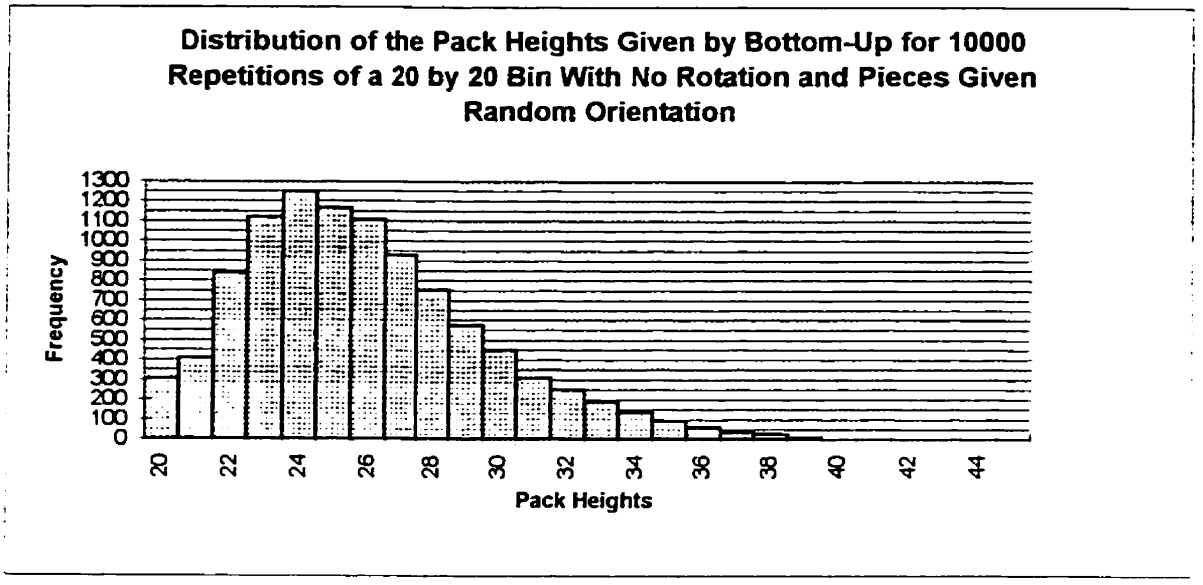

Distribution of the Pack Heights Given by Bottom-Up for 10000 Repetitions of a 20 by 20 Bin With No Rotation and Pieces Given Random Orientation
Frequency
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
20
22
24
26
28
30
32
34
36
38
40
42
44
Pack Heights

Figure 6.7

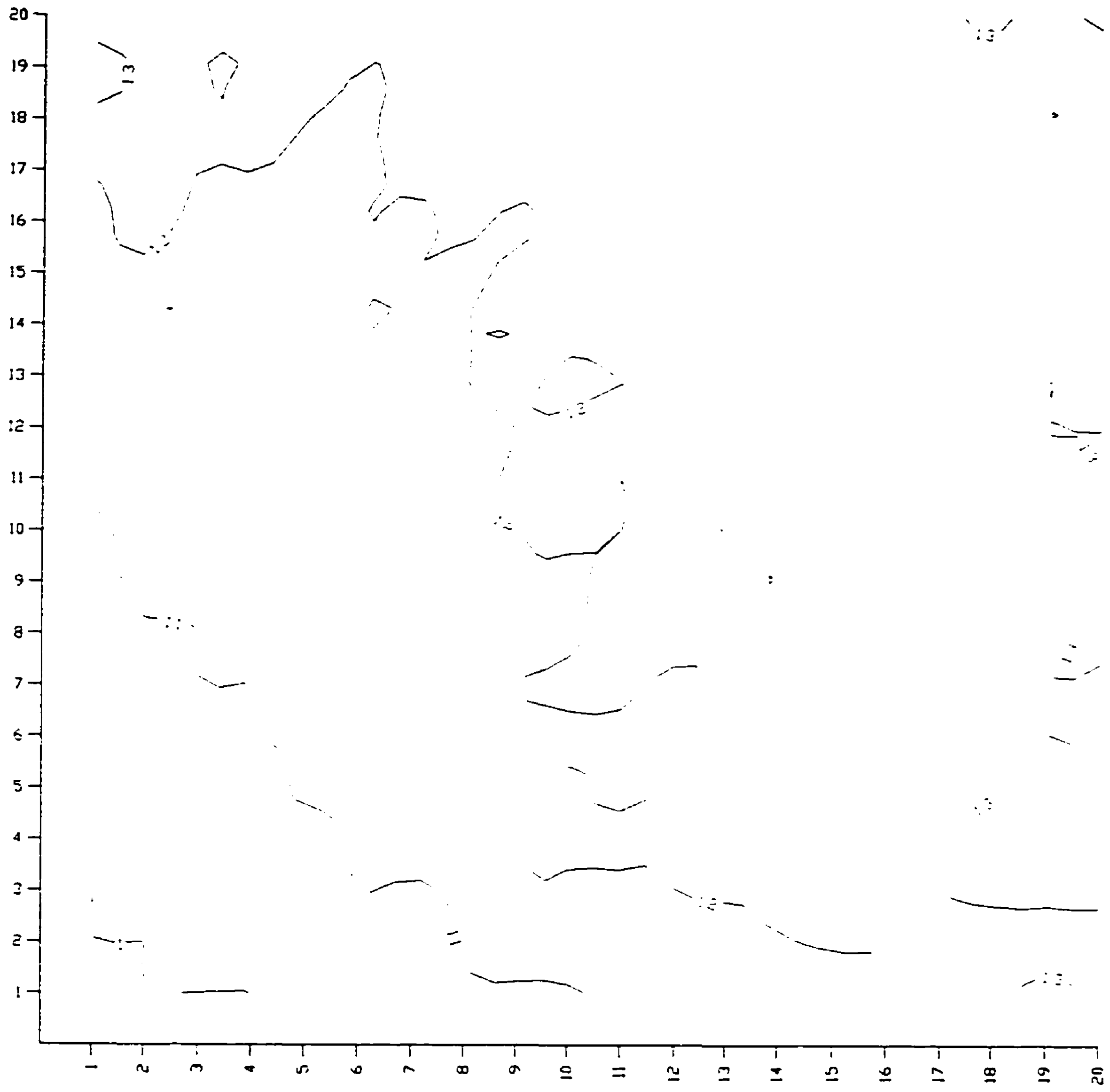


Figure 6.8: The average performance given by the Bottom-Up algorithm for 20 by 20 bins, with no rotation and random piece orientation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

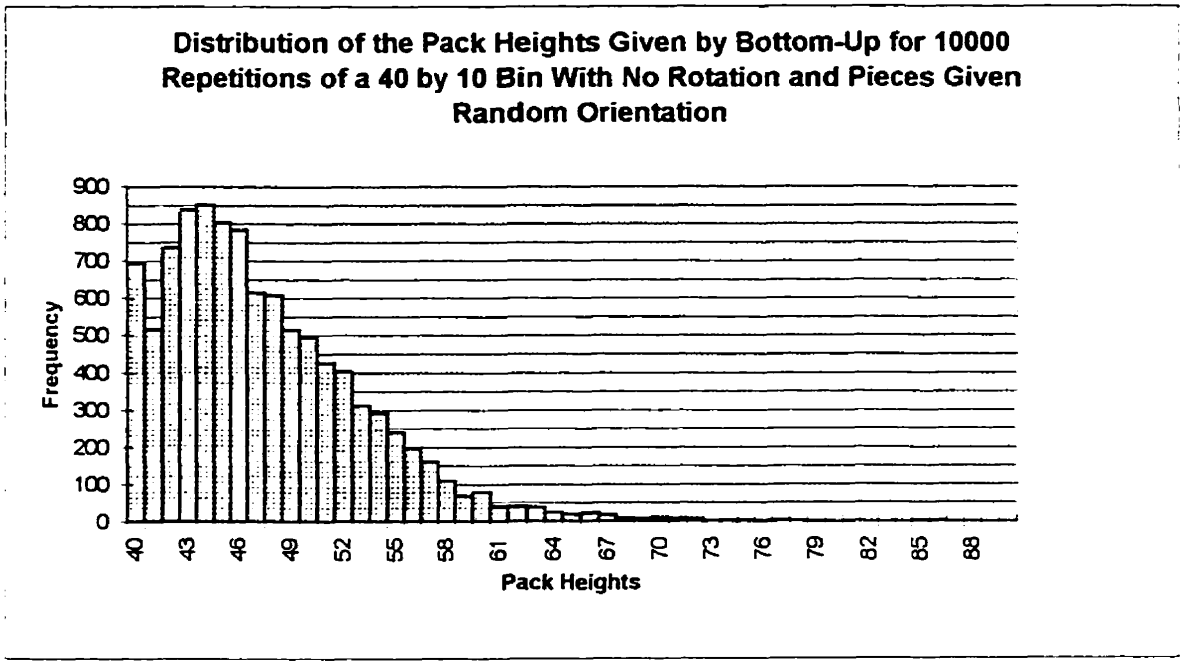

Distribution of the Pack Heights Given by Bottom-Up for 10000 Repetitions of a 40 by 10 Bin With No Rotation and Pieces Given Random Orientation
Frequency
900
800
700
600
500
400
300
200
100
0
40
43
46
49
52
55
58
61
64
67
70
73
76
79
82
85
88
Pack Heights

Figure 6.9

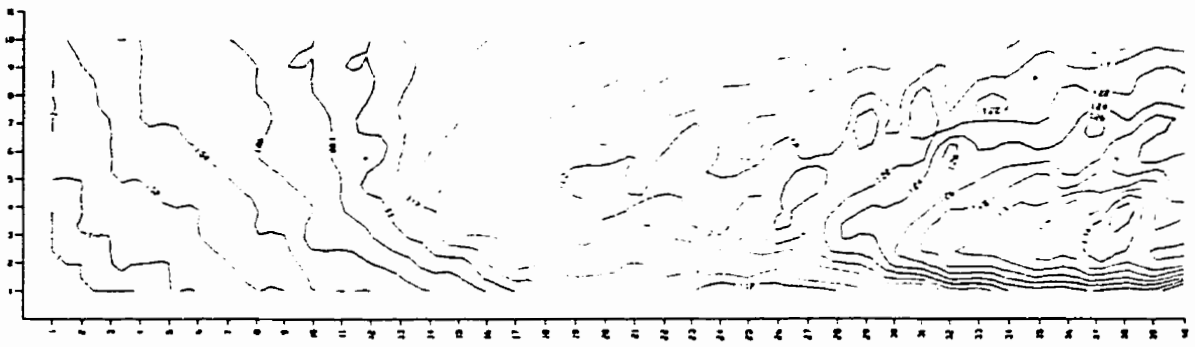


Figure 6.10: The average performance given by the Bottom-Up algorithm for 40 by 10 bins, with no rotation and random piece orientation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

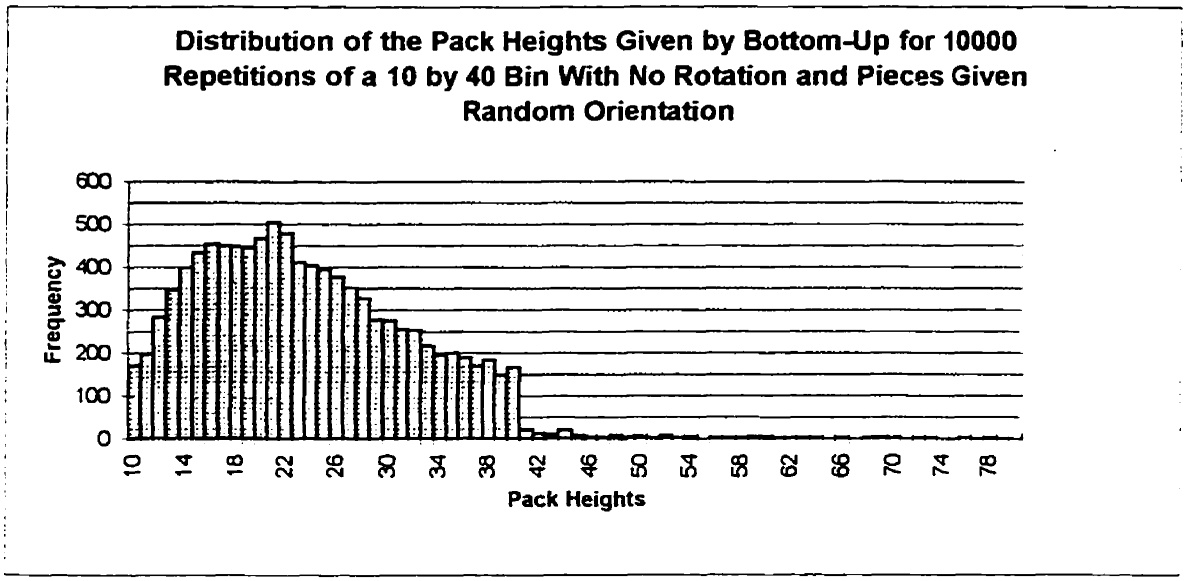

Distribution of the Pack Heights Given by Bottom-Up for 10000 Repetitions of a 10 by 40 Bin With No Rotation and Pieces Given Random Orientation
Frequency
600
500
400
300
200
100
0
10
14
18
22
26
30
34
38
42
46
50
54
58
62
66
70
74
78
Pack Heights

Figure 6.11

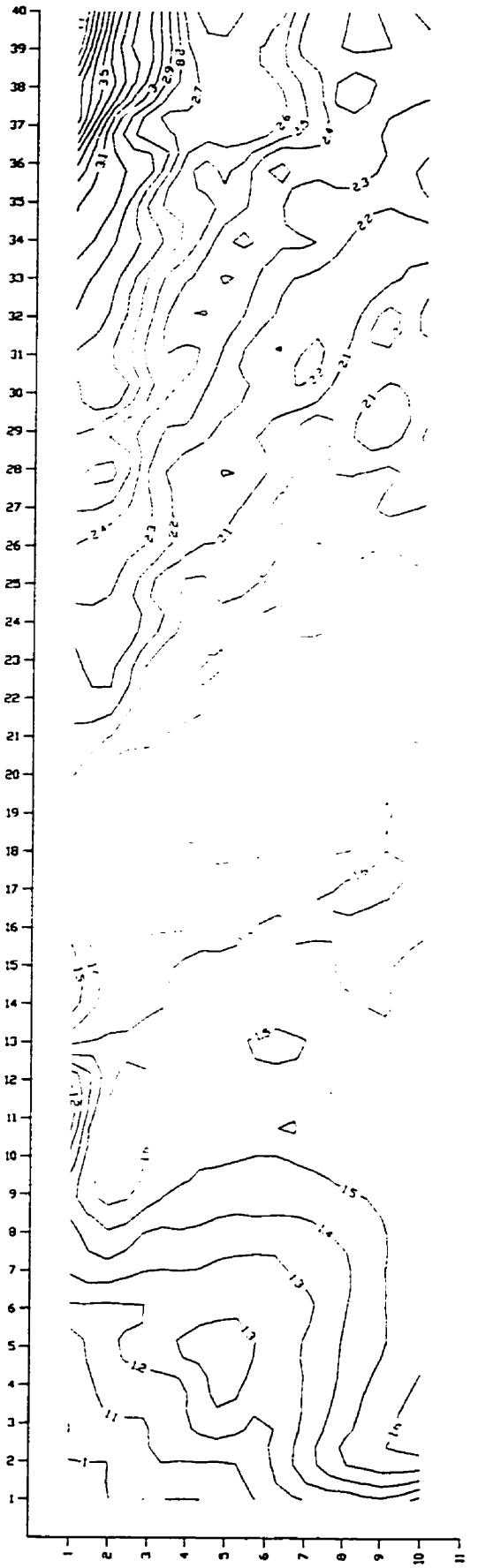


Figure 6.12: The average performance given by the Bottom-Up algorithm for 10 by 40 bins, with no rotation and random piece orientation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

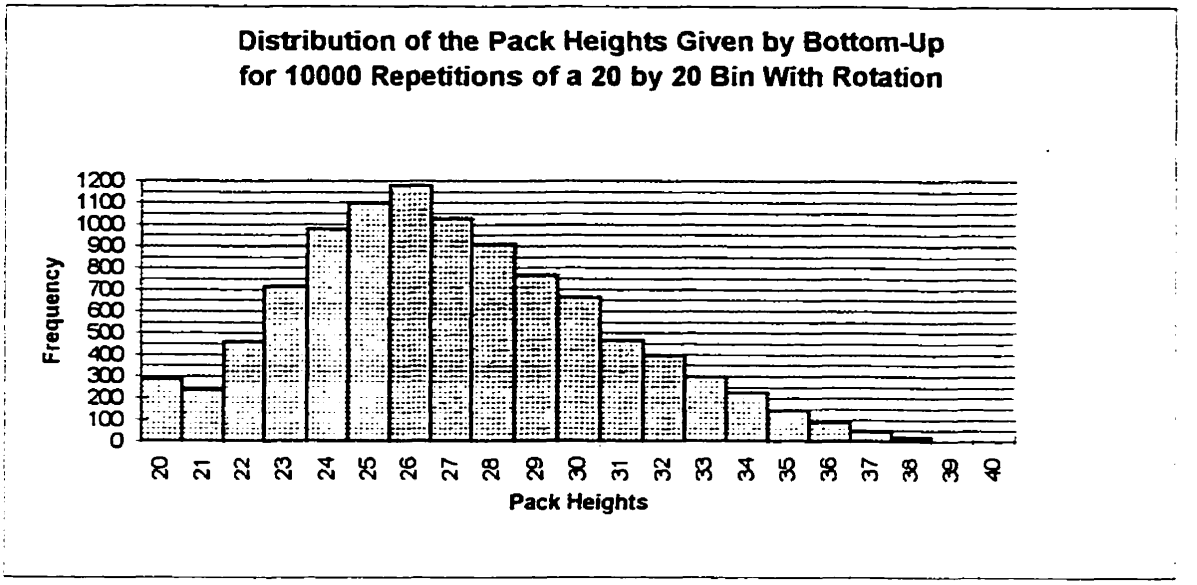

Distribution of the Pack Heights Given by Bottom-Up
for 10000 Repetitions of a 20 by 20 Bin With Rotation
Frequency
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
Pack Heights

Figure 6.13

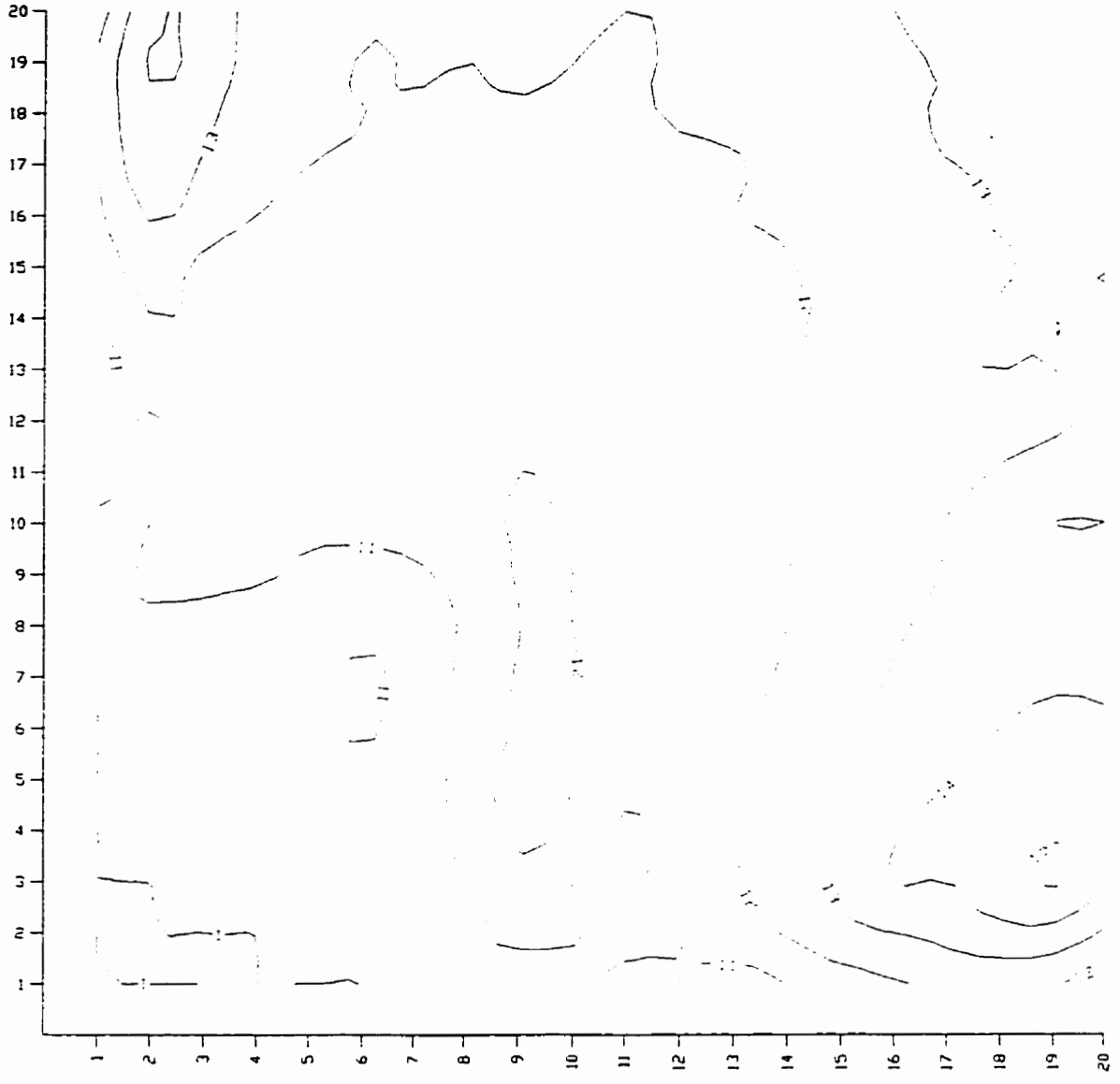


Figure 6.14: The average performance given by the Bottom-Up algorithm for 20 by 20 bins, with rotation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

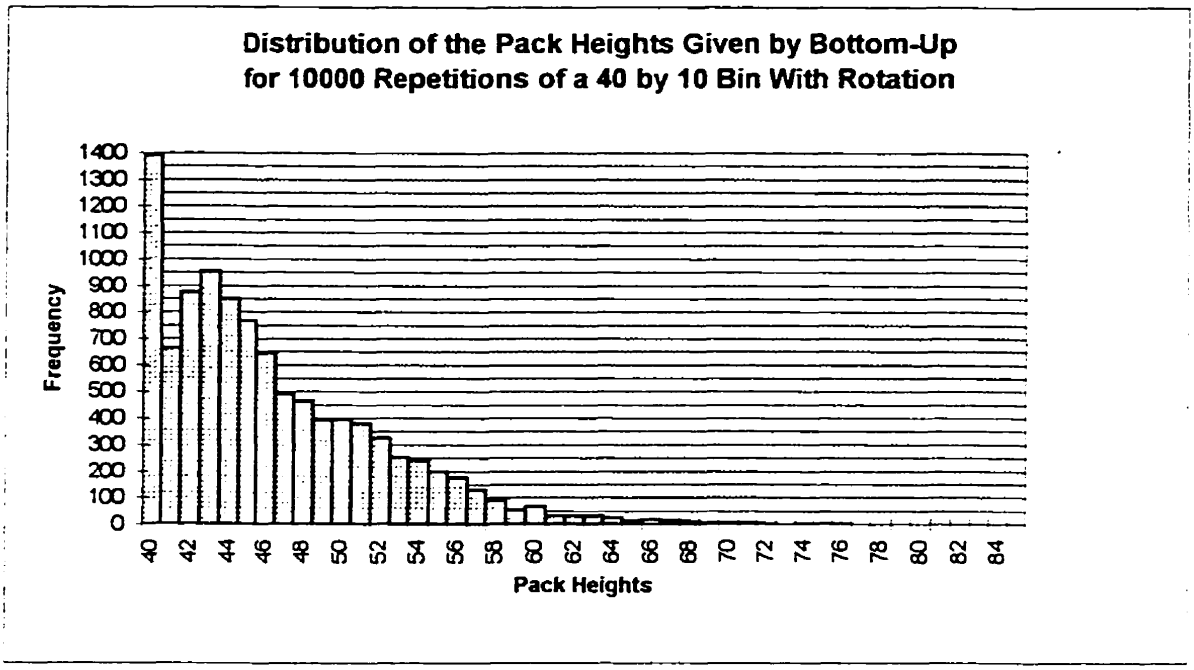

Distribution of the Pack Heights Given by Bottom-Up
for 10000 Repetitions of a 40 by 10 Bin With Rotation
Frequency
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
40
42
44
46
48
50
52
54
56
58
60
62
64
66
68
70
72
74
76
78
80
82
84
Pack Heights

Figure 6.15

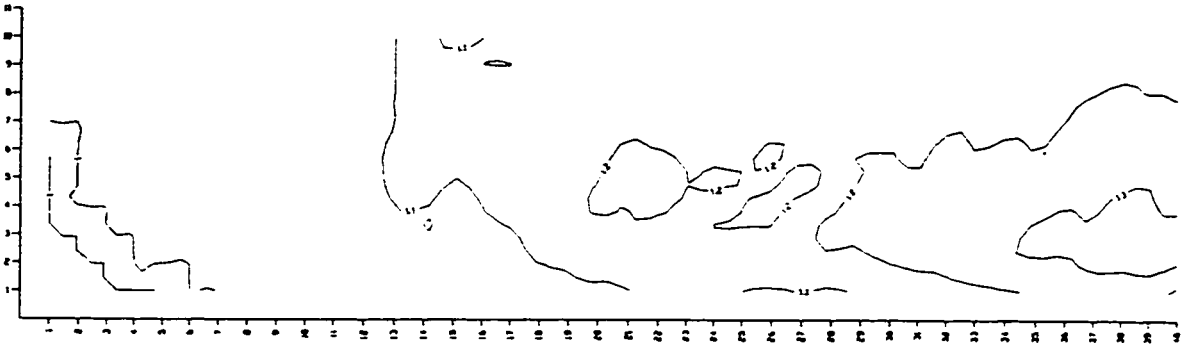


Figure 6.16: The average performance given by the Bottom-Up algorithm for 40 by 10 bins, with rotation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

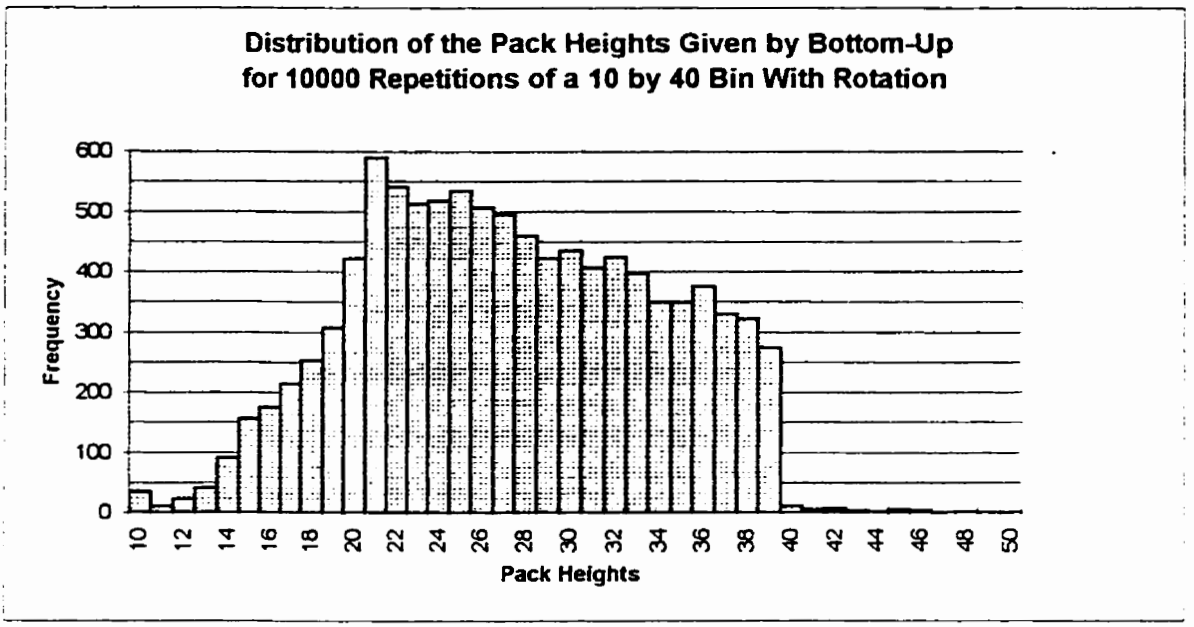

Distribution of the Pack Heights Given by Bottom-Up
for 10000 Repetitions of a 10 by 40 Bin With Rotation
Frequency
0
100
200
300
400
500
600
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
50
Pack Heights

Figure 6.17

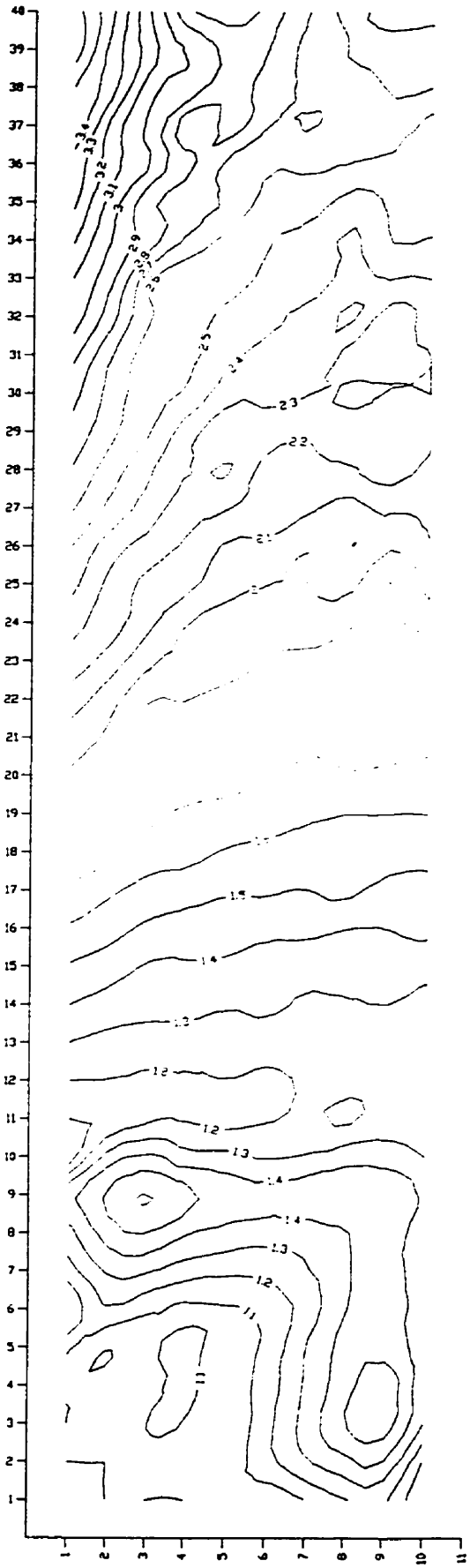


Figure 6.18: The average performance given by the Bottom-Up algorithm for 10 by 40 bins, with rotation, over all possible maximum piece size selections. The horizontal axis gives the maximum piece height allowed and the vertical axis gives the maximum piece width allowed. The contour lines show the average BU pack height divided by the optimal pack height.

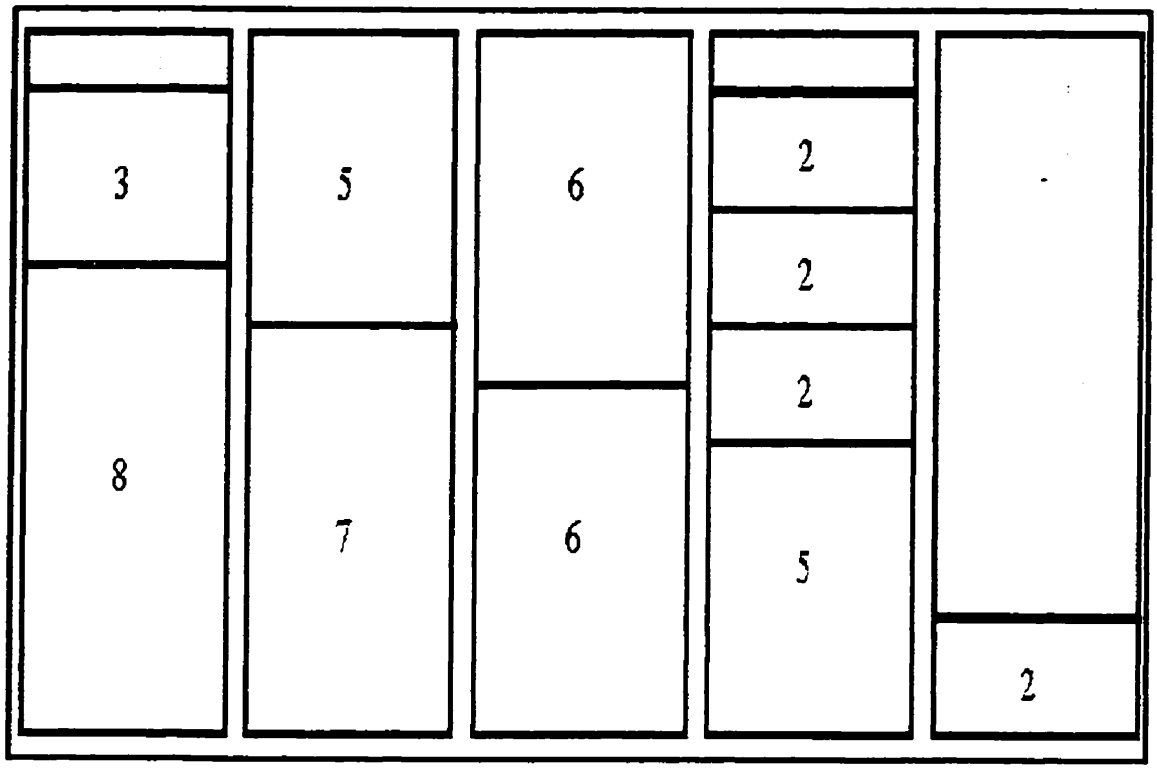
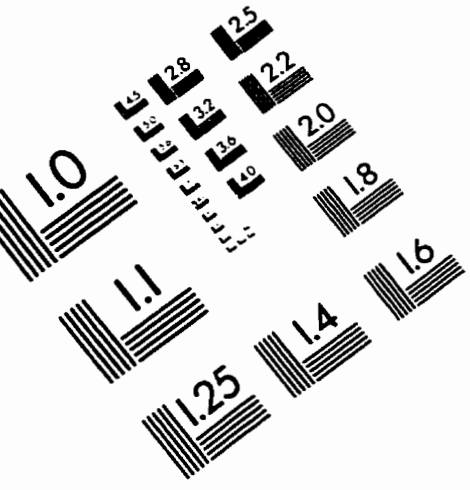


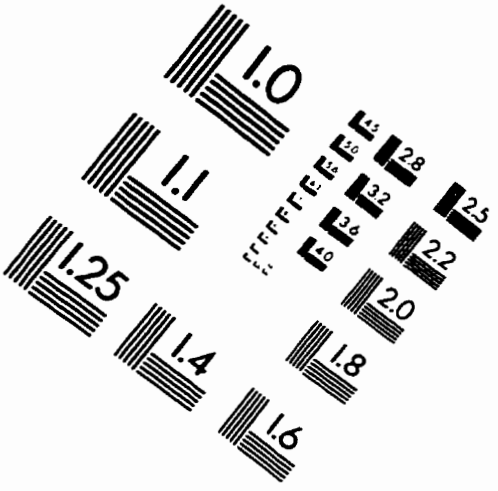
Figure A.1: The one dimensional pack given by FFD of the list L={8,7,6,6,5,5,3,2,2,2,2} into a bin of height 12. The optimal pack is 4 bins. FFD packs using 5 bins.

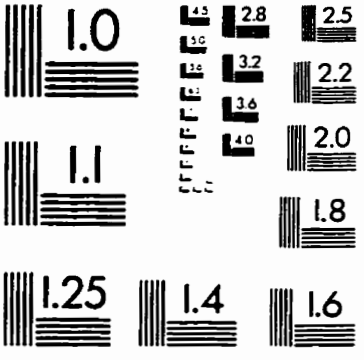
VITA AUCTORIS

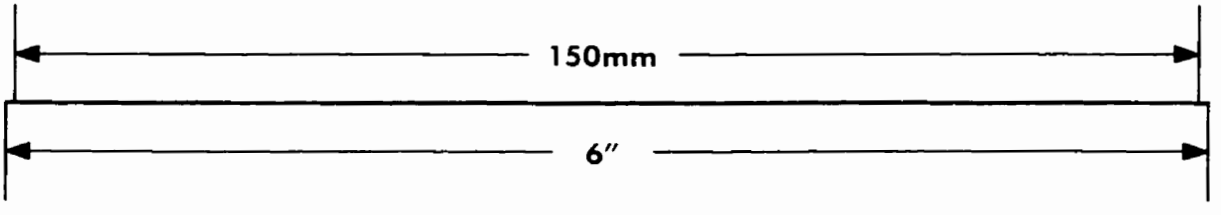
NAME	Todd Braithwaite
PLACE OF BIRTH	Windsor, Ontario
YEAR OF BIRTH	1973
EDUCATION	Vincent Massey Secondary School, Windsor 1987-1992
	University of Windsor, Windsor, Ontario, 1992-1996 B.Sc., Honours Mathematics and Statistics
	University of Windsor, Windsor, Ontario 1996-1997 M.Sc., Statistics

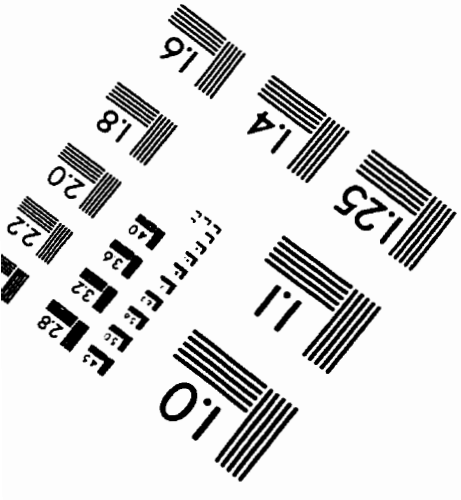
IMAGE EVALUATION TEST TARGET (QA-3)

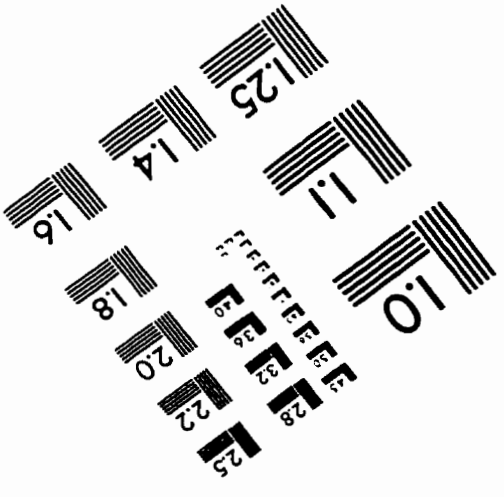












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989