A Parallel Stochastic Optimization Algorithm for Solving 2D Bin Packing Problems

Roy P. Pargas

Department of Computer Science
Clemson University
Clemson, SC 29631
email : pargas@cs.clemson.edu

Rajat Jain

Department of Computer Science
Clemson University
Clemson, SC 29631
email : rajat@cs.clemson.edu

Abstract

This study describes a stochastic approach to the problem of packing two-dimensional figures in a rectangular area efficiently. The techniques employed are similar to those used in genetic algorithms or in simulated annealing algorithms, algorithmic methods which we group under the general classification: stochastic optimization. A parallel processing system, an Intel i860 hypercube, is used to speed up execution; execution time is quite lengthy due to the costly process of evaluating the lengths of layouts. Load balancing is quite efficient; near-perfect load balancing is achieved. Four different data sets are tested, the simplest consisting of 100 rectangles of various sizes, the most complex consisting of 129 figures, each of seven possible shapes and of differing sizes. The goal of a minimum of 80% efficiency or utilization based on bin length, is achieved in all runs performed.

1 Introduction

Bin packing is one of those problems that is quite easy to state but extremely difficult to solve (NP-hard in the strong sense [9]). In order to put two- and three-dimensional bin packing in perspective, we define one-dimensional bin packing: Given a finite set of objects $X = \{x_1, x_2, \ldots, x_n\}$ with associated sizes $S = \{s_1, s_2, \ldots, s_n\}$ such that $0 \leq s_i \leq 1$, partition X into k disjoint subsets $X_1, X_2, \ldots, X_p$ such that the sum of the sizes of the objects in each partition is at most 1 and that p is minimal. Two-dimensional bin packing can be intuitively defined as packing a finite number of two-dimensional objects, always squares or rectangles, into a two-dimensional bin of a given height and infinite length (assuming the bin lies on its side and is filled to the right). The objective is to pack the the objects as efficiently as possible, minimizing the total length required. Three-dimensional bin packing is an extension of the two-dimensional problem; the objects usually studied are cubes or rectangular solids and the bin is has a square or rectangular base and infinite height or length.

Applications of two-dimensional bin-packing can be found in the apparel industry in which patterns for garments (called markers) are laid out on plies of cloth. A ply has a given width and the objective is to pack the marker so as to minimize the required length. The aerospace industry has the same problem; here the material is aluminum and the pieces to be cut are component parts of the fuselage or wing. Both of these applications fall into what is called the *cutting stock problem*. Yet another application can be found in the scheduling of tasks and the allocation of memory to those tasks in a computer system. Bin width is the amount of memory required and bin length is time. The objective is maximum utilization (or minimal waste) of memory over time. The study of three-dimensional bin packing has been found useful in job scheduling problems in partitionable mesh connected systems. The objective is to minimize completion time for all jobs.

Virtually all of the work on two- and three-dimensional problems has focused on rectangles or rectangular solids, respectively. The primary reason, of course, is to make the problem mathematically tractable in order to derive bounds on or measures of algorithm performance and quality of solution. These measures include: (1) space and time complexity of the algorithm, (2) asymptotic efficiency of the algorithm (as the optimum length of the bin becomes much greater than the dimensions of the largest figure), and (3) probabilistic analyses of the algorithm using ran-

1043-0989/93 $03.00 © 1993 IEEE

dom data,

Few, if any, of the studies found in the literature provide any practical guidance to the person faced with the task of developing software to solve a real problem. In our view, there is great need for work in developing practical heuristics to solve large two- and three-dimensional bin-packing problems. This then is the purpose of this paper: *to present an approach which can be used to solve actual problems in two-dimensional bin packing.* The approach combines new algorithmic techniques in stochastic optimization with new computer systems (multiprocessors) providing, we believe, a powerful tool to solve this difficult problem. The approach can be extended easily to three-dimensional bin packing problems.

Stochastic optimization is a generic term for optimization heuristics which include such approaches as *genetic algorithms* and *simulated annealing* ([6], [7], [10]). Such techniques employ algorithms that mimic natural processes, such as selection and mutation in natural evolution, or metallurgical processes, such as the annealing of metals, in an attempt to evolve solutions to large and difficult problems. The problems typically have solution spaces that are so large that traditional search techniques, for example *branch and bound*, or optimization techniques, such as *linear programming*, are totally impractical.

A separate development in computing is the emergence of very powerful computer systems called multiprocessor systems. Such systems can provide dramatic improvements in computation speed, up to N times faster than sequential algorithms, if parallel algorithms can be developed that take advantage of the multiprocessing capability without suffering excessive overhead in processor-to-processor communication (for distributed-memory multiprocessor systems) or contention for memory (in shared-memory multiprocessor systems).

This algorithm described in this paper combines the software power of stochastic optimization algorithms with the scalable hardware strength of distributed multiprocessor systems to solve difficult two-dimensional bin packing problems. The rest of the paper is organized as follows. In Section 2, we present briefly related work. Section 3 describes the algorithm in detail. Section 4 explains the use of a parallel processing system, specifically an Intel iPSC/860 hypercube. In Section 5, the results of empirical runs on four problem sets tested are presented, including details on the performance of the multiprocessor system used. Section 6 draws conclusions and lists several extensions that can be done to this work.

2 Previous Work

The vast majority of the work that one finds in bin packing is in *one-dimensional* bin packing. A small sample includes papers by Coffman, So, Hoi and Yao [5], Friesen and Kuhk [8], Hofri [11], Hofri and Kamhi [12], Johnson, Demers, Ullman, Garey, and Graham [13], Coffman, Garey, Johnson, and Tarjan [3], and Rhee and Talagrand [17].

Work in two-dimensional bin packing is much rarer and virtually all involve only rectangular figures. Such work includes that of Baker and Schwarz [1] who propose an algorithm which places rectangular figures one layer at a time; the algorithms produced are called *shelf* algorithms. Their algorithm does not require that the rectangles be considered in decreasing height or width order, unlike earlier algorithms. Probabilistic studies include papers by Csirik, Frenk, Galambos and Kan [2] and Coffman and Lagarias [4]. The latter paper proposes an $O(n \log n)$ to pack squares no larger than $w \times w$ in an infinite strip of width w so as to minimize the height or span of the packing.

Work in three-dimensional bin packing is rarer still. An example is a paper by Li and Cheng [14] which extends two-dimensional shelf (or level) packing algorithms for rectangles to three-dimensional layer-by-layer algorithms for rectangular solids.

3 Algorithm

The algorithm described in this section has several components: (1) Initialization, (2) Selection, (3) Solution Generation, (4) Evaluation, and (5) Termination. We first define a few terms and describe some of the structures used before describing the algorithm components in detail.

A *figure* is a right-angled polygon loosely resembling one of the block letters: *C, E, F, H, I, L, T.* Examples are shown in Figure 1. We assume some predefined unit length; we also assume that the length of each edge of a figure is a multiple of this unit length. As a result, each figure occupies an integral number of unit squares. This has significance in the evaluation function defined in Section 3.4.

For a given problem with N figures, each figure is numbered between 0 and $(N-1)$. A figure has a default initial orientation, but may be rotated 90°, 180°, or 270°. Figure 2 shows a block figure E in orientations 0 through 3.

A *solution* is a structure with the following format:

$$[(f_1, o_1), (f_2, o_2), \ldots, (f_N, o_N), L]$$

Figure 1

Figure 2

where f_i, $0 \leq f_i \leq (N-1)$ represents the ith of N figures, o_j, $0 \leq o_j \leq 3$ is the figure's current orientation, and L is the length (cost) of the solution. A bin has a predefined height H as shown in Figure 3. The length L is the distance between the left edge of the bin and the right edge of the rightmost figure after the figures are placed during evaluation of a solution. The placement of a figure increases the value of L if its right edge is further to the right than the previous rightmost figure. More details on the process of evaluation of a solution are provided in Section 3.4 below.

As an example, if $N = 5$, then $[(1,3), (4,0), (3,1), (2,3), (0,2), 11]$ is a solution where the figures are processed in the order 1, 4, 3, 2, and 0. Moreover, the initial orientations of the figures are 3, 0, 1, 3, and 2, respectively. This solution has been evaluated and its length L is 11.

The algorithm in the remainder of this section works not with a single solution at a time but with a set, called a *population*, of solutions.

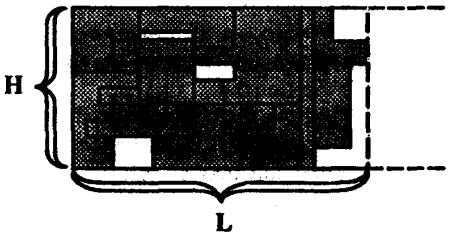


Figure 3

```
Line
 1  Initialize the population
 2      by generating a random solution,
 3      determining its length L, and inserting
 4      it into the population according to L ;
    repeat
 5      Select a solution using a linearly
            biased random number generator ;
 6      Generate a new solution
 7          either by perturbing the
            solution selected (probability p)
 8          or by generating a random
            solution (probability (1 − p)) ;
 9      Determine the value, L,
            of the new solution ;
10      Insert the solution into the population
            maintaining sorted order ;
11      Increment the iteration count ;
12  until [(Population has converged)
13      or (No improvement of best solution)
14      or (NumIterations > MaxIterations)] ;
```

Figure 4

The search for a solution is summarized by the algorithm shown in Figure 4. Lines 1-4 initialize the population, generating solutions randomly, evaluating them, and inserting them into the population, sorted by length. After initialization, new solutions are produced either by selecting a solution using a linearly biased random number generator (line 5, described in Section 3.1) and perturbing it (line 7), or by random generation of new solutions (line 8). The algorithm terminates when any one of three conditions (lines 12, 13, 14) is met.

3.1 Initialization

The population is initialized randomly, i.e., we generate an integer between 0 and $(N-1)$ for each f_i and an integer between 0 and 3 for each o_i. Evaluation of each solution (Section 3.4) produces its length L. The population is then sorted on L, from best to worst.

3.2 Biased Selection

Selection of solutions is performed using a linearly biased random number generator [16]. The effect is that the probability of selecting a solution is linearly proportional to its relative position in the population;

the better the solution, the higher the probability of its being selected. The probability of selecting the best solution of the population is B times greater than the probability of selecting the worst solution, where B is a parameter (a bias) provided by the user. Values of B used range from 1.5 through 3.0.

3.3 Solution Generation

Generating a new solution from one or more solutions already in the population (called *recombination* in genetic algorithms and *perturbation* in simulated annealing algorithms) can be performed in many different ways. Several were tried in this study; the most successful of these is described below. This strategy was used to produce the results described in Section 5.

New solutions are generated in either of two ways. (a) In the first, a new solution is "derived" from an existing solution. An existing solution, S, is selected from the population using linear bias. Several solutions in the "neighborhood" of S are randomly selected and evaluated. If the best of these neighborhood solutions is *at least as good* as S (i.e., the length of the best solution is equal to or less than the length of S), then the best solution is inserted into the population. Otherwise, all "derived" solutions are abandoned and no solution is inserted into the population.

Two solutions, S_1 and S_2,

$$S_1 = [(f_1, o_1), (f_2, o_2), \ldots, (f_N, o_N), L_1] \ ,$$
$$S_2 = [(F_1, O_1), (F_2, O_2), \ldots, (F_N, O_N), L_2]$$

are defined to be *neighbors* if the sequence in which their figures are placed are identical except for exactly two figures, i.e., if $f_i = F_i$ for all but two values of i. This definition imposes a simple graphical structure on the solution space in which solutions are nodes and two nodes are adjacent *iff* the solutions they represent are neighbors. Note also that each solution has exactly ${}_N C_2 = N(N+1)/2$ neighbors. We can decide to make the search of a solution's neighborhood exhaustive, i.e., examine its every neighbor. To reduce the amount of time required per iteration, however, we choose instead to select and evaluate a fixed number (50) of neighbors, randomly chosen, in each iteration. This is accomplished by randomly generating two numbers i and j between 0 and $N-1$, inclusive, and by swapping figures f_i and f_j in the solution. This is done in approximately 90% of the iterations.

(b) The second method simply generates a new solution randomly, as was done during initialization. The purpose of generating a solution randomly is to introduce new permutations, possibly very different from those present in the population, in order to prevent the population from converging prematurely. This is done in approximately 10% of the iterations.

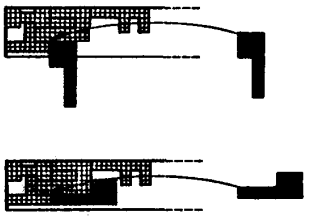


Figure 5

To summarize, in each iteration, a solution S is selected using a linearly biased random number generator with bias B, and, with probability $p \approx 0.9$, fifty of its neighbors are randomly generated and evaluated in search of a better solution. With the remaining probability $1-p \approx 0.1$ a new solution is randomly generated and evaluated (see Figure 4, line 8).

3.4 Evaluation

Evaluation of a solution S involves assigning a location in the two-dimensional bin to each figure such that no two figures overlap. Recall that figures are right-angled polygons and that the length of each edge is an integer number of units. The figures are assigned locations in the order defined by the permutation. The evaluation algorithm employs a two-dimensional bit-array, called A, of height H and of arbitrary length. A represents the two-dimensional bin and each element of A represents one square unit of space in the bin. Each square in the bin is either occupied (by a figure) or is empty. Two figures are not allowed to occupy the same cell of A.

The array A is initially empty; as figures are placed, the appropriate cells of the array are marked occupied. At any time in the placement process, checking the bit-array A reveals which areas remain free and which are occupied. Figure 5 shows an an L-shaped figure (shaded) being placed in A. The figure, currently in orientation 2, does not fit in the available space. Turning the figure 90°, i.e., incrementing the orientation by 1, however, allows the figure to be placed neatly in the space available.

The complete algorithm used in evaluating a solu-

tion is given in Figure 6. Recall that a solution is of the form

$$[(f_1, o_1), (f_2, o_2), \ldots, (f_N, o_N), L]$$

and the purpose of the evaluation process is to compute the length L. The algorithm places each of the figures f_1 through f_N in the order given by the permutation. When figure f_i is to be placed, the algorithm scans the cells of A for the next (or at the start of repeat-until loop, the first) empty cell. When the cell is found, f_i is placed with orientation o_i and with the leftmost-uppermost square unit on the empty cell.

If successful, i.e., if figure f_i can be placed such that no part of the figure lies on an occupied cell of A, figure f_i is considered placed and the cells of A on which it lies are marked occupied. If unsuccessful, the algorithm attempts three more times with this empty cell, once with the figure rotated 90° relative to orientation o_i, once with the figure rotated 180°, and once with the figure rotated 270°. If all four attempts are unsuccessful, the algorithm scans A for the next empty cell and repeats the process. The search is finite because, in the worst case, the current figure can be placed in the column to the right of the rightmost figure already placed in A (see Figure 7).

Scanning the cells of A in search of the next empty cell can be conducted in many different ways. This study used a simple linear scan in column major order. Other search techniques are suggested in Section 6.

We end this section with two final notes: (1) evaluation of solutions is deterministic, and (2) there exists some permutation of the figures with correct orientations that, when sent to the evaluation function, produces the optimal length.

3.5 Termination

There are three conditions under which the process terminates (Figure 4, lines 12, 13, 14): (a) the population converges, (b) no improvement in the best solution has occurred for a prespecified number of iterations, or (c) a predefined maximum number of iterations is reached. Condition (a), convergence, is defined as having occurred when the difference between the lengths of the best and worst solutions in the population is less than or equal to some user specified value. In this study, this value is zero, i.e., convergence occurs when all solutions in the population have the same length. Condition (b) terminates execution if the process finds itself in a rut. Condition (c), on the other hand, is used as a practical limit on the total amount of effort spent on the process.

```
Line
 1  Receive a solution
 2  Set the solution length L to 0
    foreach figure in the solution
 3      Prepare to systematically scan cells of A
    repeat
 4      Find the next empty cell of A
 5      Attempt to place the figure with the pres-
            cribed orientation and with its leftmost
            uppermost square unit on the empty cell
 6      if unsuccessful, try figure by rotating 90°
 7      if unsuccessful, try figure by rotating 180°
 8      if unsuccessful, try figure by rotating 270°
    until (figure is successfully placed)
 9      Mark "occupied" all cells covered
            by the placed figure
    end foreach
10  Return the solution with its length L
```

Figure 6

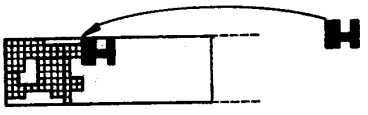


Figure 7

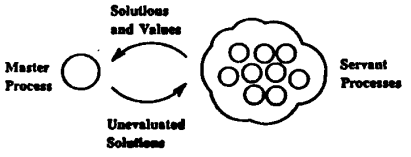


Figure 8

Set	Num Figs	Shapes	N	Pop'n Size	Num Iter	Lin Bias	Ran (%)	Time (min)	Opt Len	Best Sol	Uti l (%)
1	100	I	32	200	1000	2	10	31	37	40	92
2	126	I,L,T	32	200	1000	2	10	63	54	60	90
3	43	C,E,F,H,I,L,T	32	200	1000	2	10	28	45	55	81
4	129	C,E,F,H,I,L,T	32	200	500	2	10	135	135	162	83

Figure 9

4 Parallel Processing

By far the most time-consuming component of the entire process is the evaluation of a solution (section 3.4). Over 95% of execution time is spent evaluating the lengths of solutions. For this reason, the algorithm is designed such that it can execute on a single processor if necessary, but can take advantage of multiple processors if available. This section describes the parallel algorithm.

The algorithm is divided into two independent processes: Process 0 and Process 1. Process 0 manages the population of solutions: collecting the initial solutions during initializing, selecting solutions in order to form new ones, inserting new solutions, deciding whether a new solution will be generated from an existing solution or whether an entirely random solution will be formed, and checking for convergence or termination. In short, Process 0 does everything except evaluate solutions. That is the job of Process 1. Process 0 sends a solution to Process 1 for evaluation. Process 1 takes a solution and places each of the figures in the manner described in Section 3.4. At the end of this operation, Process 1 returns both the solution and the final length L to Process 0. Note that Process 1 *must* return the entire solution, and not just its length, because it may have changed the orientation of one or more of the figures.

If there are multiple processors available, one of the processors may execute Process 0 and the each of the remaining processors may execute Process 1. This defines a very simple multiprocessing strategy. Process 0, in effect, is a master process which farms out the time-consuming task of solution-evaluation to servant processes, each of which is executing on a different processor (Figure 8).

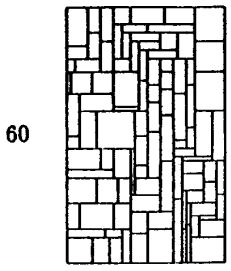
As each servant process completes its evaluation and returns the solution and length to the master process, it receives a new solution (or solutions) to evaluate. The master process therefore distributes solutions for evaluation and collects evaluated solutions continuously. The master process may see to it that each servant process has at least one solution waiting to be evaluated so as to guarantee that all of the processors remain busy, i.e., at no time is a processor idle since after the processor completes evaluation of a solution, another is immediately available. Because all processors are busy all the time except possibly at the very end of the entire process, near-perfect load balancing is achieved quite easily.

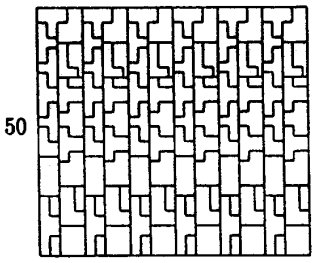
The multiprocessor used in this study is an Intel i860 hypercube. A node on this distributed memory system holds an Intel i860 processor with 8 Mbytes of memory. The host processor runs System V UNIX and the node processors run Intel's proprietary NX operating system (version 3.3.1). Process 0 executes on node 0 of the hypercube; Process 1 runs on every other available node. For example, when 32 processors are used, Process 1 runs on each of nodes 1 through 31. The parallel algorithm, however, is not designed exclusively for a hypercube or, for that matter, for a distributed memory multiprocessor system. The design is simple and generic enough to run on *any* multiprocessor system, distributed or shared, even on a loosely coupled network of workstations.

5 Results

Figure 9 describes the four sets of data used in this study and summarizes the results obtained. Problem set 1 consists of 100 I-shaped blocks, i.e., rectangles. Set 2 is more difficult with more figures and figures of three different shapes (I, L, and T). Set 3 has fewer shapes but a greater variety of them (C, E, F, H, I, L, and T). Finally set 4 is the most difficult with the largest number of all seven shapes.

Each problem set is solved using $N = 32$ nodes of an Intel iPSC/860 hypercube. The population size is 200 and the maximum number of iterations allowed is either 1000 or 500. Selection of solutions uses a linearly biased random number generator with a bias of 2. Random generation of solutions is performed in 10% of the iterations (hence perturbation of solutions is performed 90% of the time). Execution time ranges





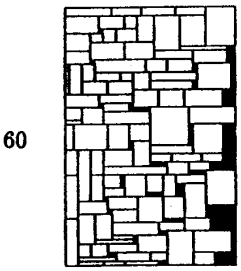


Figure 10

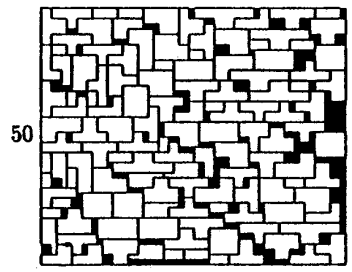
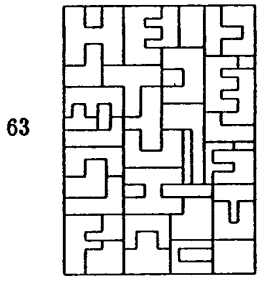


Figure 11



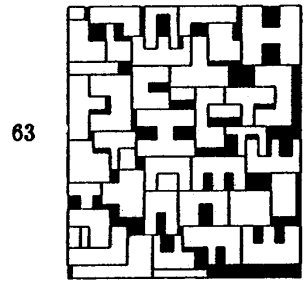
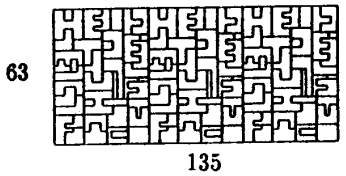


Figure 12



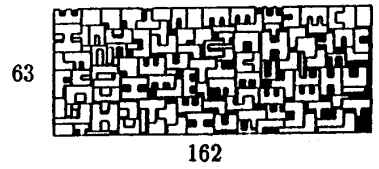


Figure 13

from 28 minutes for problem set 3 through 135 minutes for set 4. The next two columns give the optimum length for each of the problem sets and the length of the best solution found by the algorithm. From these, we can compute utilization or efficiency as

$$100 \times \frac{OptimumLength}{BestSolutionFound}\ \%.$$

Figures 10-13 show the optimal solutions and the best solutions found by the algorithm for each of the problem sets 1-4, respectively. The placement algorithm is able to generate the optimal placement when provided with the correct permutation and orientation of figures.

6 Conclusions and Future Work

This paper presents an novel approach to the problem of packing two-dimensional figures in a rectangular area of fixed height and unbounded length. The figures are right-angled polygons of seven different shapes loosely resembling the block letters C, E, F, H, I, L, and T. This differs radically from most algorithms proposed for two-dimensional bin packing which concentrate solely on rectangles. Another difference in the approach proposed is the use of techniques in stochastic optimization; the algorithm gradually evolves a population of solutions with the goal in mind of steadily improving the best solution. A third difference is the use of multiple processors to reduce execution time. The parallel algorithm is straightforward and is inherently load-balanced. A goal of at least 80% utilization is achieved for each of four problem sets tested.

Placement of the figures is accomplished with the use of a bit array representing the two-dimensional

bin. This makes the addition of more complex figures straightforward; each distinct shape is simply a separate case in the placement and evaluation process. The same bit array may be extended to three-dimensions allowing algorithm described in this paper to be extended to a three-dimensional bin packing algorithm. Work in both of these extensions is already in progress. Additional two-dimensional shapes being considered include figures such as polygons with non-right angles and figures with arcs. Three-dimensional shapes currently being studied include rectangular solids, L-shaped solids, and T-shaped solids.

Finally, anyone interested in working with the four problem sets used in this study may request the data by writing or sending email to the authors.

Acknowledgement

The authors wish to thank the Oak Ridge National Laboratory for its support of this research through the use of its Intel iPSC/860 hypercube.

References

[1] Baker, B.S. and Jerald S. Schwarz, Shelf Algorithms for Two-Dimensional Packing Problems, *SIAM Journal of Computing 12* : 3, August 1983, pp. 508-525.

[2] Csirik, J., J.B.G. Frenk, G. Galambos, and AHG Rinnooy Kan, Probabilistic Analysis of Algorithms for Dual Bin Packing Problems, *Journal of Algorithms 12*, 1991, pp. 189-203.

[3] Coffmann, E.G., M.R. Garey, D.S. Johnson, and R.E. Tarjan, Performance Bounds for Level-Oriented Two-dimensional Packing Algorithms, *SIAM Journal of Computing 9*, 1980, pp. 808-826.

[4] Coffman Jr., E.G. and J.C. Lagarias, Algorithms for Packing Squares : A Probabilistic Analysis, *SIAM Journal of Computing 18* : 1, February 1989, pp. 166-185.

[5] Coffman Jr., E.G., K. So, M. Hoi, and A.C. Yao, A Stochastic Model of Bin Packing, *Information and Control 44*, 1980, pp. 105-115.

[6] Davis, Lawrence, *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann, Los Altos, California, 1987.

[7] Davis, Lawrence, *Handbook of Genetic Algorithms*, Van Nostrand Reihhold, New York, 1991.

[8] Friesen, D.K. and F.S. Kuhl, Analysis of a Hybrid Algorithm for Packing Unequal Bins, *SIAM Journal of Computing 17*: 1, February 1988, pp. 23-40.

[9] Garey, Michael R. and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, 1979.

[10] Goldberg, David, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Massachusetts, 1989.

[11] Hofri, Micha, A Probabilistic Analysis of the Next-fit Bin Packing Algorithm, *Journal of Algorithms 5*, 1984, pp. 547-556.

[12] Hofri, Micha and Sami Kamhi, A Stochastic Analysis of the Next-fit Decreasing Bin Packing Algorithm, *Journal of Algorithms 7*, 1986, pp. 489-509.

[13] Johnson, D.S., A. Demers, M.R. Ullman, M.R. Garey, and R.L. Graham, Worst-case Performance Bounds for Simple One-dimensional Packing Algorithms, *SIAM Journal of Computing 3*, 1974, pp. 299-326.

[14] Li, Keqin and Kam Hoi Cheng, On Three-dimensional Packing, *SIAM Journal of Computing 19* : 5, October 1990, pp. 847-867.

[15] Li, K. and K.H. Cheng, Job scheduling in PMCS using a 2DBS as the the system partition scheme, *Proceedings of the 1990 International Conference on Parallel Processing 1*, August 1990, pp. 119-122.

[16] Pargas, Roy P. A Linearly-Biased Random Number Generator, Clemson University Technical Report 92-1006, October 1992.

[17] Rhee, Wansoo T. and Michel Talagrand, Optimal Bin Packing with Items of Random Sizes, *SIAM Journal of Computing 18* : 1, February 1989, pp. 139-151.